



[翻译]理解PYTHON中的装饰器

- [翻译]理解python中的装饰器
 - python的函数是对象
 - 函数引用
 - 手工装饰器
 - 装饰器阐述
 - 最后回答问题
 - 向装饰器函数传递参数
 - 装饰方法
 - 向装饰器传递参数
 - 练习：一个装饰装饰器的装饰器
 - 装饰器使用最佳实践
 - 装饰器为何那么有用

分享

有人翻译过了，很多转载，暂时没找到原文，各个地方的排版不一样，排版（代码格式），代码注解等都不怎么好

练练手，顺手一翻吧，权当加深理解

来源[stackoverflow](#)上的问题 [链接](#)

很长哦(应该是巨长...分了三次搞完)，要有耐心看完

python的函数是对象

要理解装饰器，首先，你必须明白，在python中，函数是对象. 这很重要.

简单例子来理解为什么

```
def shout(word="yes"):
    return word.capitalize()+"!"

print shout()
# outputs : 'Yes!'

# 作为一个对象，你可以讲函数赋值给另一个对象
scream = shout

# 注意到这里我们并没有使用括号：我们不是调用函数，而是将函数'shout'赋给变量'scream'
# 这意味着，你可以通过'scream'调用'shout'

print scream()
# outputs : 'Yes!'

# 不仅如此，你可以删除老的名称'shout'，但是通过'scream'依旧可以访问原有函数

del shout
try:
    print shout()
except NameError, e:
    print e
    #outputs: "name 'shout' is not defined"

print scream()
# outputs: 'Yes!'
```

好了，记住这点，我们将会很快用到它。

Python函数另一个有趣的特性是，函数可以被定义在另一个函数里面

```
def talk():
    # 你可以定义一个函数
    def whisper(word="yes"):
        return word.lower()+"..."

    # ... 并且立刻调用
    print whisper()

# 每次当你调用"talk"，都会定义"whisper"
# 并且在"talk"中被调用
talk()
# outputs:
# "yes..."

#但是在"talk"外部，函数"whisper"不存在！
try:
```

```
print whisper()
except NameError, e:
    print e
#outputs : "name 'whisper' is not defined"
```

函数引用

好了，到这里了，接下来是有意思的部分，我们刚才看到 函数是对象，然后：

1.函数可以赋值给一个变量

2.函数可以定义在另一个函数内部

即，这也意味着一个函数可以返回另一个函数:-) ,让我们来看另一段代码

```
def getTalk(type="shout"):

    # 定义函数
    def shout(word="yes"):
        return word.capitalize()+"!"

    def whisper(word="yes") :
        return word.lower()+"...";

    # 返回函数
    if type == "shout":
        # 没有使用"()", 并不是要调用函数, 而是要返回函数对象
        return shout
    else:
        return whisper

# 如何使用?

# 将函数返回值赋值给一个变量
talk = getTalk()

# 我们可以打印下这个函数对象
print talk
#outputs : <function shout at 0xb7ea817c>

# 这个对象是函数的返回值
print talk()
#outputs : Yes!

# 不仅如此, 你还可以直接使用之
```

```
print getTalk("whisper")()  
#outputs : yes...
```

但是稍等，如果你可以返回一个函数，那么你也可以将函数作为参数传递

```
def doSomethingBefore(func):  
    print "I do something before then I call the function you gave me"  
    print func()  
  
doSomethingBefore(scream)  
#outputs:  
#I do something before then I call the function you gave me  
#Yes!
```

好了，现在你已经了解要理解装饰器的每件事。

装饰器就是封装器，可以让你在被装饰函数之前或之后执行代码，而不必修改函数本身

手工装饰器

如何书写一个装饰器

```
# 装饰器是一个以另一个函数为参数的函数  
def my_shiny_new_decorator(a_function_to_decorate):  
  
    # 在这里，装饰器定义一个函数： 包装器。  
    # 这个函数将原始函数进行包装，以达到在原始函数之前、之后执行代码的目的  
    def the_wrapper_around_the_original_function():  
  
        # 将你要在原始函数之前执行的代码放到这里  
        print "Before the function runs"  
  
        # 调用原始函数(需要带括号)  
        a_function_to_decorate()  
  
        # 将你要在原始函数之后执行的代码放到这里  
        print "After the function runs"  
  
    # 代码到这里，函数'a_function_to_decorate'还没有被执行  
    # 我们将返回刚才创建的这个包装函数  
    # 这个函数包含原始函数及要执行的附加代码，并且可以被使用  
    return the_wrapper_around_the_original_function  
  
# 创建一个函数  
def a_stand_alone_function():  
    print "I am a stand alone function, don't you dare modify me"
```

```

a_stand_alone_function()
#outputs: I am a stand alone function, don't you dare modify me

# 好了，在这里你可以装饰这个函数，扩展其行为
# 将函数传递给装饰器，装饰器将动态地将其包装在任何你想执行的代码中，然后返回一个新的函数
a_stand_alone_function_decorated = my_shiny_new_decorator(a_stand_alone_function)

# 调用新函数，可以看到装饰器的效果
a_stand_alone_function_decorated()
#outputs:
#Before the function runs
#I am a stand alone function, don't you dare modify me
#After the function runs

```

到这里，或许你想每次调用[a_stand_alone_function](#)都使用

[a_stand_alone_function_decorated](#)替代之 很简单，只需要将[a_stand_alone_function](#)用[my_shiny_new_decorator](#)装饰返回

```

a_stand_alone_function = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function()
#outputs:
#Before the function runs
#I am a stand alone function, don't you dare modify me
#After the function runs

# 这就是装饰器做的事情！

```

装饰器阐述

前面的例子，使用装饰器语法

```

@my_shiny_new_decorator
def another_stand_alone_function():
    print "Leave me alone"

another_stand_alone_function()
#outputs:
#Before the function runs
#Leave me alone
#After the function runs

```

是的，就是这么简单. @decorator是下面代码的简写

```

another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)

```

装饰器只是 装饰器模式的python实现

python代码中还存在其他几个经典的设计模式，以方便开发，例如迭代器iterators

当然，你可以累加装饰器

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "<\_____/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#tomatoes#"
        func()
        print "~salad~"
    return wrapper

def sandwich(food="--ham--"):
    print food

sandwich()
#outputs: --ham--

#累加两个装饰器
sandwich = bread(ingredients(sandwich))
sandwich()
#outputs:
#</''''''\>
# #tomatoes#
# --ham--
# ~salad~
#<\_____/>
```

使用python装饰器语法

```
@bread
@ingredients
def sandwich(food="--ham--"):
    print food

sandwich()
#outputs:
#</''''''\>
```

```
# #tomatoes#
# --ham--
# ~salad~
#<\_____/>
```

装饰器位置的顺序很重要

```
@ingredients
@bread
def strange_sandwich(food="--ham--"):
    print food

    strange_sandwich()
#outputs:
##tomatoes#
#</' '''\>
# --ham--
#<\_____/>
# ~salad~'
```

最后回答问题

```
# bold装饰器
def makebold(fn):
    def wrapper():
        # 在前后加入标签
        return "<b>" + fn() + "</b>"
    return wrapper

# italic装饰器
def makeitalic(fn):
    def wrapper():
        # 加入标签
        return "<i>" + fn() + "</i>"
    return wrapper

@makebold
@makeitalic
def say():
    return "hello"

print say()
#outputs: <b><i>hello</i></b>

# 等价的代码
def say():
    return "hello"
```

```
say = makebold(makeitalic(say))

print say()

#outputs: <b><i>hello</i></b>
```

好了，到这里你可以高兴地离开了，或者来看下一些装饰器高级的用法

向装饰器函数传递参数

这不是黑魔法，你只需要让包装传递参数：

```
def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print "I got args! Look:", arg1, arg2
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
```

当你调用装饰器返回的函数，实际上是调用包装函数，所以给包装函数传递参数即可将参数传给装饰器函数

```
@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print "My name is", first_name, last_name

print_full_name("Peter", "Venkman")
# outputs:
#I got args! Look: Peter Venkman
#My name is Peter Venkman
```

装饰方法

Python中对象的方法和函数是一样的，除了对象的方法首个参数是指向当前对象的引用(self)。这意味着你可以用同样的方法构建一个装饰器，只是必须考虑self

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3 # very friendly, decrease age even more :-)
        return method_to_decorate(self, lie)
    return wrapper

class Lucy(object):

    def __init__(self):
        self.age = 32

    @method_friendly_decorator
```



```
def sayYourAge(self, lie):
    print "I am %s, what did you think?" % (self.age + lie)

l = Lucy()
l.sayYourAge(-3)
#outputs: I am 26, what did you think?
```

当然，你可以构造一个更加通用的装饰器，可以作用在任何函数或对象方法上，而不必关系其参数 使用

```
*args, **kwargs
```

如下代码

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    # 包装函数可以接受任何参数
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print "Do I have args?:"
        print args
        print kwargs
        # 然后你可以解开参数, *args, **kwargs
        # 如果你对此不是很熟悉, 可以参考 http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print "Python is cool, no argument here."

function_with_no_argument()
#outputs
#Do I have args?:
#()
#{ }
#Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print a, b, c

function_with_arguments(1,2,3)
#outputs
#Do I have args?:
#(1, 2, 3)
#{ }
#1 2 3
```

```

@a_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c, platypus="Why not ?"):
    print "Do %s, %s and %s like platypus? %s" %\
        (a, b, c, platypus)

function_with_named_arguments("Bill", "Linus", "Steve", platypus="Indeed!")
#outputs
#Do I have args ? :
#('Bill', 'Linus', 'Steve')
#{'platypus': 'Indeed!'}
#Do Bill, Linus and Steve like platypus? Indeed!

class Mary(object):
    def __init__(self):
        self.age = 31

    @a_decorator_passing_arbitrary_arguments
    def sayYourAge(self, lie=-3): # You can now add a default value
        print "I am %s, what did you think ?" % (self.age + lie)

m = Mary()
m.sayYourAge()
#outputs
# Do I have args?:
#(<__main__.Mary object at 0xb7d303ac>,)
#{ }
#I am 28, what did you think?

```

向装饰器传递参数

好了，现在你或许会想是否可以向装饰器本身传递参数

装饰器必须使用函数作为参数，所以这看起来会有些复杂，你不能直接传递参数给装饰器本身

在开始处理这个问题前，看一点提醒

```

# 装饰器是普通的方法
def my_decorator(func):
    print "I am a ordinary function"
    def wrapper():
        print "I am function returned by the decorator"
        func()
    return wrapper

```

```
# 所以，你可以不通过@调用它

def lazy_function():
    print "zzzzzzzz"

decorated_function = my_decorator(lazy_function)
#outputs: I am a ordinary function

# It outputs "I am a ordinary function", because that's just what you do:

# 调用一个函数，没有什么特别
@my_decorator
def lazy_function():
    print "zzzzzzzz"

#outputs: I am a ordinary function
```

上面两个形式本质上是相同的，"my_decorator" 被调用.所以当你使用"@my_decorator",告诉python一个函数被变量"my_decorator"标记 这十分重要,因为你提供的标签直接指向装饰器...或者不是，继续

```
# 声明一个用于创建装饰器的函数
def decorator_maker():

    print "I make decorators! I am executed only once: "+\
        "when you make me create a decorator."

    def my_decorator(func):
        print "I am a decorator! I am executed only when you decorate a function."

        def wrapped():
            print ("I am the wrapper around the decorated function. "
                  "I am called when you call the decorated function. "
                  "As the wrapper, I return the RESULT of the decorated "
                  function.")
            return func()

        print "As the decorator, I return the wrapped function."
        return wrapped

    print "As a decorator maker, I return a decorator"
    return my_decorator

# Let's create a decorator. It's just a new function after all.
# 创建一个装饰器，本质上只是一个函数
new_decorator = decorator_maker()
```

```
#outputs:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator

# 使用装饰器装饰函数

def decorated_function():
    print "I am the decorated function."

decorated_function = new_decorator(decorated_function)
#outputs:
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function

# 调用被装饰函数
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the
decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

我们跳过中间变量，做同样的事情

```
def decorated_function():
    print "I am the decorated function."
decorated_function = decorator_maker()(decorated_function)
#outputs:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

# 最后:
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the
decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

使用装饰器语法，更简短

```
@decorator_maker()
def decorated_function():
    print "I am the decorated function."
#outputs:
```

```
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

#最终:
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the
decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

到这里，我们使用@调用一个函数

回到问题，向装饰器本身传递参数，如果我们可以通过函数去创建装饰器，那么我们可以传递参数给这个函数，对么？

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):

    print "I make decorators! And I accept arguments:", decorator_arg1,
decorator_arg2

    def my_decorator(func):
        # 这里能传递参数的能力，是闭包的特性
        # 更多闭包的内容，参考 http://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-python
        print "I am the decorator. Somehow you passed me arguments:",
decorator_arg1, decorator_arg2

        # 不要搞混了装饰器参数和函数参数
        def wrapped(function_arg1, function_arg2) :
            print ("I am the wrapper around the decorated function.\n"
                "I can access all the variables\n"
                "\t- from the decorator: {0} {1}\n"
                "\t- from the function call: {2} {3}\n"
                "Then I can pass them to the decorated function"
                .format(decorator_arg1, decorator_arg2,
                    function_arg1, function_arg2))
            return func(function_arg1, function_arg2)

        return wrapped

    return my_decorator

@decorator_maker_with_arguments("Leonard", "Sheldon")
def decorated_function_with_arguments(function_arg1, function_arg2):
```

```

print ("I am the decorated function and only knows about my arguments: {0}"
      " {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments("Rajesh", "Howard")
#outputs:
#I make decorators! And I accept arguments: Leonard Sheldon
#I am the decorator. Somehow you passed me arguments: Leonard Sheldon
#I am the wrapper around the decorated function.
#I can access all the variables
#  - from the decorator: Leonard Sheldon
#  - from the function call: Rajesh Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Rajesh Howard

```

好了, that's it.参数可以设置为变量

```

c1 = "Penny"
c2 = "Leslie"

@decorator_maker_with_arguments("Leonard", c1)
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ("I am the decorated function and only knows about my arguments:"
          " {0} {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments(c2, "Howard")
#outputs:
#I make decorators! And I accept arguments: Leonard Penny
#I am the decorator. Somehow you passed me arguments: Leonard Penny
#I am the wrapper around the decorated function.
#I can access all the variables
#  - from the decorator: Leonard Penny
#  - from the function call: Leslie Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Leslie Howard

```

你可以看到, 你可以使用像其它函数一样使用这个方法向装饰器传递参数.如果你愿意你甚至可以使用 `arg *kwargs`.

但是记住, 装饰器仅在Python代码导入时被调用一次,之后你不能动态地改变参数.当你使用"import x",函数已经被装饰, 所以你不能改变什么

练习: 一个装饰装饰器的装饰器

作为奖励, 我将展示创建可以处理任何参数的装饰器代码片段. 毕竟, 为了接收参数, 必须使用另一个函数来创建装饰器

让我们来给装饰器写一个装饰器:

```
# 装饰 装饰器 的装饰器 (好绕.....)
def decorator_with_args(decorator_to_enhance):
    """
    这个函数将作为装饰器使用
    它必须装饰另一个函数
    它将允许任何接收任意数量参数的装饰器
    方便你每次查询如何实现
    """

    # 同样的技巧传递参数
    def decorator_maker(*args, **kwargs):

        # 创建一个只接收函数的装饰器
        # 但是这里保存了从创建者传递过来的参数
        def decorator_wrapper(func):

            # 我们返回原始装饰器的结果
            # 这是一个普通的函数, 返回值是另一个函数
            # 陷阱: 装饰器必须有这个特殊的签名, 否则不会生效
            return decorator_to_enhance(func, *args, **kwargs)

        return decorator_wrapper

    return decorator_maker
```

使用:

```
# 你创建这个函数是作为一个装饰器, 但是给它附加了一个装饰器
# 别忘了, 函数签名是: "decorator(func, *args, **kwargs)"
@decorator_with_args
def decorated_decorator(func, *args, **kwargs):
    def wrapper(function_arg1, function_arg2):
        print "Decorated with", args, kwargs
        return func(function_arg1, function_arg2)
    return wrapper

# 然后, 使用这个装饰器(your brand new decorated decorator)

@decorated_decorator(42, 404, 1024)
def decorated_function(function_arg1, function_arg2):
    print "Hello", function_arg1, function_arg2

decorated_function("Universe and", "everything")
#outputs:
#Decorated with (42, 404, 1024) {}
```

```
#Hello Universe and everything
```

```
# Whoooot!
```

我知道，到现在你一定会有这种感觉，就像你听一个人说“在理解递归之前，你必须首先了解递归”，但是现在，掌握这儿你有没有觉得很棒？

装饰器使用最佳实践

- 这是Python2.4的新特性，所以确保你的代码在2.4及之上的版本运行
- 装饰器降低了函数调用的性能，记住这点
- You can not un-decorate a function. There are hacks to create decorators that can be removed but nobody uses them. So once a function is decorated, it's done. For all the code.
- 装饰器包装函数，所以很难debug

Python2.5解决了最后一个问题，它提供functools模块，包含functools.wraps.这个函数会将装饰函数的名称，模块，文档字符串拷贝给封装函数,有趣的是，functools.wraps是一个装饰器:-)

```
# 调试，打印函数的名字
def foo():
    print "foo"

print foo.__name__
#outputs: foo

# 但当你使用装饰器，这一切变得混乱
def bar(func):
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

print foo.__name__
#outputs: wrapper

# "functools" 可以改变这点
import functools
```



```
def bar(func):
    # 我们所说的 "wrapper", 封装 "func"
    @functools.wraps(func)
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

# 得到的是原始的名称, 而不是封装器的名称
print foo.__name__
#outputs: foo
```

装饰器为何那么有用

现在的问题是, 我们用装饰器来做什么? 看起来很酷很强大, 但是如果有实践的例子会更好.好了, 有1000种可能。经典的用法是, 在函数的外部, 扩展一个函数的行为（你不需要改变这个函数）, 或者, 为了调试的目的（我们不修改的原因是这是临时的）, 你可以使用装饰器扩展一些函数,而不用在这些函数中书写相同的函数实现一样的功能

DRY原则, 例子:

```
def benchmark(func):
    """
    装饰器打印一个函数的执行时间
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print func.__name__, time.clock()-t
        return res
    return wrapper

def logging(func):
    """
    装饰器记录函数日志
    """
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print func.__name__, args, kwargs
        return res
```

```

return wrapper

def counter(func):
    """
    记录并打印一个函数的执行次数
    """
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print "{0} has been used: {1}x".format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper

@counter
@benchmark
@logging
def reverse_string(string):
    return str(reversed(string))

print reverse_string("Able was I ere I saw Elba")
print reverse_string("A man, a plan, a canoe, pasta, heros, rajahs, a coloratura,
maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag
again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a
peon, a canal: Panama!")

#outputs:
#reverse_string ('Able was I ere I saw Elba',) {}
#wrapper 0.0
#wrapper has been used: 1x
#able was I ere I saw elbA
#reverse_string ('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura,
maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag
again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a
peon, a canal: Panama!',) {}
#wrapper 0.0
#wrapper has been used: 2x
#!amanaP :lanac a ,noep a ,stah eros ,raj a ,hsac ,oloR a ,tur a ,mapS ,snip
,eperc a ,)lemac a ro( niaga gab ananab a ,gat a ,nat a ,gab ananab a ,gag a
,inoracam ,elacrep ,epins ,spam ,arutaroloc a ,shajar ,soreh ,atsap ,eonac a ,nalp
a ,nam A

```

装饰器意味着，你可以用正确的方法实现几乎所有的事情，而不必重写他们

```

@counter
@benchmark
@logging
def get_random_futurama_quote():

```

```
import httplib
conn = httplib.HTTPConnection("slashdot.org:80")
conn.request("HEAD", "/index.html")
for key, value in conn.getresponse().getheaders():
    if key.startswith("x-b") or key.startswith("x-f"):
        return value
return "No, I'm ... doesn't!"

print get_random_futurama_quote()
print get_random_futurama_quote()

#outputs:
#get_random_futurama_quote () {}
#wrapper 0.02
#wrapper has been used: 1x
#The Laws of science be a harsh mistress.
#get_random_futurama_quote () {}
#wrapper 0.01
#wrapper has been used: 2x
#Curse you, merciful Poseidon!
```

Python本身提供了一些装饰器: property,staticmethod,等等,

Django使用装饰器去管理缓存和权限. Twisted to fake inlining asynchronous functions calls.
用途广泛

EDIT: 鉴于这个回答的完美, 人们希望我去回答metaclass,我这样做了

版权声明: 自由转载-非商用-非衍生-保持署名 | [Creative Commons BY-NC-ND 3.0](#)






如果你觉得我的文章或项目对你有所帮助, **You can buy me a coffee:)**

[上一篇: \[翻译\]Python中yield的解释](#)[下一篇: \[翻译整理\]stackoverflow python 百问](#)

2 条评论

wklken's blog

 dongguangming ▾ Recommended 2  分享

按评分高低排序 ▾



加入讨论...

**Yang Zhonggang** · 4个月前

三明治那个例子好棒。

^ | ▾ · 回复 · 分享 ▾

**xavierskip** · 1年前

留个脚印，下次需要用到装饰器的时候，来看看。

^ | ▾ · 回复 · 分享 ▾

在 WKLKEN'S BLOG 上还有.....

这是什么?

vim插件: quickrun[快速执行]

1条评论 · 3个月前

**Richard** — c++用quickrun貌似不能输入类似cin的语句都不能用**Logstash+ElasticSearch+Kibana- 实现相对通用的数据收集分析**

2 条评论 · 4个月前

**kimileonis** —**重读>**

2 条评论 · 10个月前

**Lotaku** — 这就是所谓的“温故而知新”吧
^ _ ^ ...**Python-基础-时间日期处理小结**

2 条评论 · 6个月前

**caimaoy** — 楼主，广告是你的公司吗？ 订阅 在您的网站上使用Disqus 隐私

DISQUS

COPYRIGHT © 2015 WKLKEN

HOSTED ON VULTR . POWERED BY PELICAN. SOCIAL ICONS BY FONT-AWESOME.