[详解Python的装饰器](https://www.cnblogs.com/cicaday/p/python-decorator.html)

Python中的装饰器是你进入Python大门的一道坎，不管你跨不跨过去它都在那里。

**为什么需要装饰器**

我们假设你的程序实现了say\_hello()和say\_goodbye()两个函数。

def say\_hello():

print "hello!"

def say\_goodbye():

print "hello!" # bug here

if \_\_name\_\_ == '\_\_main\_\_':

say\_hello()

say\_goodbye()

但是在实际调用中，我们发现程序出错了，上面的代码打印了两个hello。经过调试你发现是say\_goodbye()出错了。老板要求调用每个方法前都要记录进入函数的名称，比如这样：

[DEBUG]: Enter say\_hello()

Hello!

[DEBUG]: Enter say\_goodbye()

Goodbye!

好，小A是个毕业生，他是这样实现的。

def say\_hello():

print "[DEBUG]: enter say\_hello()"

print "hello!"

def say\_goodbye():

print "[DEBUG]: enter say\_goodbye()"

print "hello!"

if \_\_name\_\_ == '\_\_main\_\_':

say\_hello()

say\_goodbye()

很low吧？ 嗯是的。小B工作有一段时间了，他告诉小A可以这样写。

def debug():

import inspect

caller\_name = inspect.stack()[1][3]

print "[DEBUG]: enter {}()".format(caller\_name)

def say\_hello():

debug()

print "hello!"

def say\_goodbye():

debug()

print "goodbye!"

if \_\_name\_\_ == '\_\_main\_\_':

say\_hello()

say\_goodbye()

是不是好一点？那当然，但是每个业务函数里都要调用一下debug()函数，是不是很难受？万一老板说say相关的函数不用debug，do相关的才需要呢？

那么装饰器这时候应该登场了。

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

概括的讲，装饰器的作用就是**为已经存在的函数或对象添加额外的功能**。

**怎么写一个装饰器**

在早些时候 (Python Version < 2.4，2004年以前)，为一个函数添加额外功能的写法是这样的。

def debug(func):

def wrapper():

print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)

return func()

return wrapper

def say\_hello():

print "hello!"

say\_hello = debug(say\_hello) # 添加功能并保持原函数名不变

上面的debug函数其实已经是一个装饰器了，它对原函数做了包装并返回了另外一个函数，额外添加了一些功能。因为这样写实在不太优雅，[在后面版本的Python中支持了@语法糖](mailto:%E5%9C%A8%E5%90%8E%E9%9D%A2%E7%89%88%E6%9C%AC%E7%9A%84Python%E4%B8%AD%E6%94%AF%E6%8C%81%E4%BA%86@%E8%AF%AD%E6%B3%95%E7%B3%96)，下面代码等同于早期的写法。

def debug(func):

def wrapper():

print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)

return func()

return wrapper

@debug

def say\_hello():

print "hello!"

这是最简单的装饰器，但是有一个问题，如果被装饰的函数需要传入参数，那么这个装饰器就坏了。因为返回的函数并不能接受参数，你可以指定装饰器函数wrapper接受和原函数一样的参数，比如：

def debug(func):

def wrapper(something): # 指定一毛一样的参数

print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)

return func(something)

return wrapper # 返回包装过函数

@debug

def say(something):

print "hello {}!".format(something)

这样你就解决了一个问题，但又多了N个问题。因为函数有千千万，你只管你自己的函数，别人的函数参数是什么样子，鬼知道？还好Python提供了可变参数\*args和关键字参数\*\*kwargs，有了这两个参数，装饰器就可以用于任意目标函数了。

def debug(func):

def wrapper(\*args, \*\*kwargs): # 指定宇宙无敌参数

print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)

print 'Prepare and say...',

return func(\*args, \*\*kwargs)

return wrapper # 返回

@debug

def say(something):

print "hello {}!".format(something)

至此，你已完全掌握初级的装饰器写法。

**高级一点的装饰器**

带参数的装饰器和类装饰器属于进阶的内容。在理解这些装饰器之前，最好对函数的闭包和装饰器的接口约定有一定了解。(参见http://betacat.online/posts/python-closure/)

**带参数的装饰器**

假设我们前文的装饰器需要完成的功能不仅仅是能在进入某个函数后打出log信息，而且还需指定log的级别，那么装饰器就会是这样的。

def logging(level):

def wrapper(func):

def inner\_wrapper(\*args, \*\*kwargs):

print "[{level}]: enter function {func}()".format(

level=level,

func=func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

return inner\_wrapper

return wrapper

@logging(level='INFO')

def say(something):

print "say {}!".format(something)

# 如果没有使用@语法，等同于

# say = logging(level='INFO')(say)

@logging(level='DEBUG')

def do(something):

print "do {}...".format(something)

if \_\_name\_\_ == '\_\_main\_\_':

say('hello')

do("my work")

是不是有一些晕？你可以这么理解，当带参数的装饰器被打在某个函数上时，比如@logging(level='DEBUG')，它其实是一个函数，会马上被执行，只要这个它返回的结果是一个装饰器时，那就没问题。细细再体会一下。

**基于类实现的装饰器**

装饰器函数其实是这样一个接口约束，它必须接受一个callable对象作为参数，然后返回一个callable对象。在Python中一般callable对象都是函数，但也有例外。只要某个对象重载了\_\_call\_\_()方法，那么这个对象就是callable的。

class Test():

def \_\_call\_\_(self):

print 'call me!'

t = Test()

t() # call me

像\_\_call\_\_这样前后都带下划线的方法在Python中被称为内置方法，有时候也被称为魔法方法。重载这些魔法方法一般会改变对象的内部行为。上面这个例子就让一个类对象拥有了被调用的行为。

回到装饰器上的概念上来，装饰器要求接受一个callable对象，并返回一个callable对象（不太严谨，详见后文）。那么用类来实现也是也可以的。我们可以让类的构造函数\_\_init\_\_()接受一个函数，然后重载\_\_call\_\_()并返回一个函数，也可以达到装饰器函数的效果。

class logging(object):

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print "[DEBUG]: enter function {func}()".format(

func=self.func.\_\_name\_\_)

return self.func(\*args, \*\*kwargs)

@logging

def say(something):

print "say {}!".format(something)

**带参数的类装饰器**

如果需要通过类形式实现带参数的装饰器，那么会比前面的例子稍微复杂一点。那么在构造函数里接受的就不是一个函数，而是传入的参数。通过类把这些参数保存起来。然后在重载\_\_call\_\_方法是就需要接受一个函数并返回一个函数。

class logging(object):

def \_\_init\_\_(self, level='INFO'):

self.level = level

def \_\_call\_\_(self, func): # 接受函数

def wrapper(\*args, \*\*kwargs):

print "[{level}]: enter function {func}()".format(

level=self.level,

func=func.\_\_name\_\_)

func(\*args, \*\*kwargs)

return wrapper #返回函数

@logging(level='INFO')

def say(something):

print "say {}!".format(something)

**内置的装饰器**

内置的装饰器和普通的装饰器原理是一样的，只不过返回的不是函数，而是类对象，所以更难理解一些。

**@property**

在了解这个装饰器前，你需要知道在不使用装饰器怎么写一个属性。

def getx(self):

return self.\_x

def setx(self, value):

self.\_x = value

def delx(self):

del self.\_x

# create a property

x = property(getx, setx, delx, "I am doc for x property")

以上就是一个Python属性的标准写法，其实和Java挺像的，但是太罗嗦。[有了@语法糖](mailto:%E6%9C%89%E4%BA%86@%E8%AF%AD%E6%B3%95%E7%B3%96)，能达到一样的效果但看起来更简单。

@property

def x(self): ...

# 等同于

def x(self): ...

x = property(x)

属性有三个装饰器：setter, getter, deleter ，都是在property()的基础上做了一些封装，因为setter和deleter是property()的第二和第三个参数，[不能直接套用@语法](mailto:%E4%B8%8D%E8%83%BD%E7%9B%B4%E6%8E%A5%E5%A5%97%E7%94%A8@%E8%AF%AD%E6%B3%95)。getter装饰器和不带getter的属性装饰器效果是一样的，估计只是为了凑数，本身没有任何存在的意义。经过@property装饰过的函数返回的不再是一个函数，而是一个property对象。

>>> property()

<property object at 0x10ff07940>

**@staticmethod，@classmethod**

有了@property装饰器的了解，这两个装饰器的原理是差不多的。@staticmethod返回的是一个staticmethod类对象，而@classmethod返回的是一个classmethod类对象。他们都是调用的是各自的\_\_init\_\_()构造函数。

class classmethod(object):

"""

classmethod(function) -> method

"""

def \_\_init\_\_(self, function): # for @classmethod decorator

pass

# ...

class staticmethod(object):

"""

staticmethod(function) -> method

"""

def \_\_init\_\_(self, function): # for @staticmethod decorator

pass

# ...

[装饰器的@语法就等同调用了这两个类的构造函数](mailto:%E8%A3%85%E9%A5%B0%E5%99%A8%E7%9A%84@%E8%AF%AD%E6%B3%95%E5%B0%B1%E7%AD%89%E5%90%8C%E8%B0%83%E7%94%A8%E4%BA%86%E8%BF%99%E4%B8%A4%E4%B8%AA%E7%B1%BB%E7%9A%84%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0)。

class Foo(object):

@staticmethod

def bar():

pass

# 等同于 bar = staticmethod(bar)

至此，我们上文提到的装饰器接口定义可以更加明确一些，装饰器必须接受一个callable对象，其实它并不关心你返回什么，可以是另外一个callable对象（大部分情况），也可以是其他类对象，比如property。

**装饰器里的那些坑**

装饰器可以让你代码更加优雅，减少重复，但也不全是优点，也会带来一些问题。

**位置错误的代码**

让我们直接看示例代码。

def html\_tags(tag\_name):

print 'begin outer function.'

def wrapper\_(func):

print "begin of inner wrapper function."

def wrapper(\*args, \*\*kwargs):

content = func(\*args, \*\*kwargs)

print "<{tag}>{content}</{tag}>".format(tag=tag\_name, content=content)

print 'end of inner wrapper function.'

return wrapper

print 'end of outer function'

return wrapper\_

@html\_tags('b')

def hello(name='Toby'):

return 'Hello {}!'.format(name)

hello()

hello()

在装饰器中我在各个可能的位置都加上了print语句，用于记录被调用的情况。你知道他们最后打印出来的顺序吗？如果你心里没底，那么最好不要在装饰器函数之外添加逻辑功能，否则这个装饰器就不受你控制了。以下是输出结果：

begin outer function.

end of outer function

begin of inner wrapper function.

end of inner wrapper function.

<b>Hello Toby!</b>

<b>Hello Toby!</b>

**错误的函数签名和文档**

装饰器装饰过的函数看上去名字没变，其实已经变了。

def logging(func):

def wrapper(\*args, \*\*kwargs):

"""print log before a function."""

print "[DEBUG] {}: enter {}()".format(datetime.now(), func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

return wrapper

@logging

def say(something):

"""say something"""

print "say {}!".format(something)

print say.\_\_name\_\_ # wrapper

为什么会这样呢？[只要你想想装饰器的语法糖@代替的东西就明白了](mailto:%E5%8F%AA%E8%A6%81%E4%BD%A0%E6%83%B3%E6%83%B3%E8%A3%85%E9%A5%B0%E5%99%A8%E7%9A%84%E8%AF%AD%E6%B3%95%E7%B3%96@%E4%BB%A3%E6%9B%BF%E7%9A%84%E4%B8%9C%E8%A5%BF%E5%B0%B1%E6%98%8E%E7%99%BD%E4%BA%86)。@等同于这样的写法。

say = logging(say)

logging其实返回的函数名字刚好是wrapper，那么上面的这个语句刚好就是把这个结果赋值给say，say的\_\_name\_\_自然也就是wrapper了，不仅仅是name，其他属性也都是来自wrapper，比如doc，source等等。

使用标准库里的functools.wraps，可以**基本**解决这个问题。

from functools import wraps

def logging(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

"""print log before a function."""

print "[DEBUG] {}: enter {}()".format(datetime.now(), func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

return wrapper

@logging

def say(something):

"""say something"""

print "say {}!".format(something)

print say.\_\_name\_\_ # say

print say.\_\_doc\_\_ # say something

看上去不错！主要问题解决了，但其实还不太完美。因为函数的签名和源码还是拿不到的。

import inspect

print inspect.getargspec(say) # failed

print inspect.getsource(say) # failed

如果要彻底解决这个问题可以借用第三方包，比如wrapt。后文有介绍。

[**不能装饰@staticmethod**](mailto:%E4%B8%8D%E8%83%BD%E8%A3%85%E9%A5%B0@staticmethod)**或者 @classmethod**

当你想把装饰器用在一个静态方法或者类方法时，不好意思，报错了。

class Car(object):

def \_\_init\_\_(self, model):

self.model = model

@logging # 装饰实例方法，OK

def run(self):

print "{} is running!".format(self.model)

@logging # 装饰静态方法，Failed

@staticmethod

def check\_model\_for(obj):

if isinstance(obj, Car):

print "The model of your car is {}".format(obj.model)

else:

print "{} is not a car!".format(obj)

"""

Traceback (most recent call last):

...

File "example\_4.py", line 10, in logging

@wraps(func)

File "C:\Python27\lib\functools.py", line 33, in update\_wrapper

setattr(wrapper, attr, getattr(wrapped, attr))

AttributeError: 'staticmethod' object has no attribute '\_\_module\_\_'

"""

前面已经解释了@staticmethod这个装饰器，其实它返回的并不是一个callable对象，而是一个staticmethod对象，那么它是不符合装饰器要求的（比如传入一个callable对象），你自然不能在它之上再加别的装饰器。要解决这个问题很简单，只要把你的装饰器放在@staticmethod之前就好了，因为你的装饰器返回的还是一个正常的函数，然后再加上一个@staticmethod是不会出问题的。

class Car(object):

def \_\_init\_\_(self, model):

self.model = model

@staticmethod

@logging # 在@staticmethod之前装饰，OK

def check\_model\_for(obj):

pass

**如何优化你的装饰器**

嵌套的装饰函数不太直观，我们可以使用第三方包类改进这样的情况，让装饰器函数可读性更好。

**decorator.py**

[decorator.py](http://pythonhosted.org/decorator/documentation.html) 是一个非常简单的装饰器加强包。你可以很直观的先定义包装函数wrapper()，再使用decorate(func, wrapper)方法就可以完成一个装饰器。

from decorator import decorate

def wrapper(func, \*args, \*\*kwargs):

"""print log before a function."""

print "[DEBUG] {}: enter {}()".format(datetime.now(), func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

def logging(func):

return decorate(func, wrapper) # 用wrapper装饰func

你也可以使用它自带的@decorator装饰器来完成你的装饰器。

from decorator import decorator

@decorator

def logging(func, \*args, \*\*kwargs):

print "[DEBUG] {}: enter {}()".format(datetime.now(), func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

decorator.py实现的装饰器能完整保留原函数的name，doc和args，唯一有问题的就是inspect.getsource(func)返回的还是装饰器的源代码，你需要改成inspect.getsource(func.\_\_wrapped\_\_)。

**wrapt**

[wrapt](http://wrapt.readthedocs.io/en/latest/quick-start.html)是一个功能非常完善的包，用于实现各种你想到或者你没想到的装饰器。使用wrapt实现的装饰器你不需要担心之前inspect中遇到的所有问题，因为它都帮你处理了，甚至inspect.getsource(func)也准确无误。

import wrapt

# without argument in decorator

@wrapt.decorator

def logging(wrapped, instance, args, kwargs): # instance is must

print "[DEBUG]: enter {}()".format(wrapped.\_\_name\_\_)

return wrapped(\*args, \*\*kwargs)

@logging

def say(something): pass

使用wrapt你只需要定义一个装饰器函数，但是函数签名是固定的，必须是(wrapped, instance, args, kwargs)，注意第二个参数instance是必须的，就算你不用它。当装饰器装饰在不同位置时它将得到不同的值，比如装饰在类实例方法时你可以拿到这个类实例。根据instance的值你能够更加灵活的调整你的装饰器。另外，args和kwargs也是固定的，注意前面**没有星号**。在装饰器内部调用原函数时才带星号。

如果你需要使用wrapt写一个带参数的装饰器，可以这样写。

def logging(level):

@wrapt.decorator

def wrapper(wrapped, instance, args, kwargs):

print "[{}]: enter {}()".format(level, wrapped.\_\_name\_\_)

return wrapped(\*args, \*\*kwargs)

return wrapper

@logging(level="INFO")

def do(work): pass

关于wrapt的使用，建议查阅官方文档，在此不在赘述。

* <http://wrapt.readthedocs.io/en/latest/quick-start.html>

**小结**

Python的装饰器和Java的注解（Annotation）并不是同一回事，和C#中的特性（Attribute）也不一样，完全是两个概念。

装饰器的理念是对原函数、对象的加强，相当于重新封装，所以一般装饰器函数都被命名为wrapper()，意义在于包装。函数只有在被调用时才会发挥其作用。比如@logging装饰器可以在函数执行时额外输出日志，@cache装饰过的函数可以缓存计算结果等等。

而注解和特性则是对目标函数或对象添加一些属性，相当于将其分类。这些属性可以通过反射拿到，在程序运行时对不同的特性函数或对象加以干预。比如带有Setup的函数就当成准备步骤执行，或者找到所有带有TestMethod的函数依次执行等等。

至此我所了解的装饰器已经讲完，但是还有一些内容没有提到，比如装饰类的装饰器。有机会再补充。谢谢观看。