# 《源码探秘 CPython》64. 装饰器是怎么实现的?

原创 古明地觉 古明地觉的编程教室 2022-04-08 09:00





装饰器是Python的一个亮点，但并不神秘，因为它本质上就是使用了闭包的思想，只不过给我们提供了一个优雅的语法糖。

装饰器的本质就是高阶函数加上闭包，至于为什么要有装饰器，我觉得有句话说的非常好，装饰器存在的最大意义就是可以在不改动原函数的代码和调用方式的情况下，为函数增加一些新的功能。

```
def deco(func):
    print("都闪开，我要开始装饰了")
    def inner(*args, **kwargs):
        print("开始了")
        ret = func(*args, **kwargs)
        print("结束")
        return ret
    return inner

# 这一步就等价于foo = deco(foo)
# 因此上来就会打印deco里面的print
@deco
def foo(a, b):
    print(a, b)
print("---------")

# 此时再调用foo，已经不再是原来的foo了，而是deco里面的闭包inner
foo(1, 2)

# 整体输出如下：
"""
都闪开，我要开始装饰了
---------
开始了
1 2
结束
"""
```

如果不使用装饰器的话：

```
def deco(func):
    print("都闪开，我要开始装饰了")
    def inner(*args, **kwargs):
        print("开始了")
        ret = func(*args, **kwargs)
        print("结束")
        return ret
    return inner

def foo(a, b):
    print(a, b)
# 其实@deco就是一个语法糖，它本质上就是
foo = deco(foo)
print("-------")
foo(1, 2)
```

```
"""
都闪开，我要开始装饰了
-------
开始了
1 2
结束
"""
```

打印结果告诉我们，装饰器只是类似于foo=deco(foo)的一个语法糖罢了。

装饰器本质上就是使用了闭包，两者的字节码类似，这里就不再看了。还是那句话，@只是个语法糖，它和我们直接调用foo=deco(foo)是等价的，所以理解装饰器(decorator)的关键就在于理解闭包(closure)。

另外函数在被装饰器装饰之后，整个函数其实就已经变了，而为了保留原始信息我们一般会从functools中导入一个wraps函数。当然装饰器还可以写的更复杂，比如带参数的装饰器、类装饰器等等，不过这些都属于Python层级的东西了，我们就不说了。

装饰器还可以不止一个，如果一个函数被多个装饰器装饰，会有什么表现呢？

```
def deco1(func):
    def inner():
        return f"<deco1>{func()}</deco1>"
    return inner

def deco2(func):
    def inner():
        return f"<deco2>{func()}</deco2>"
    return inner

def deco3(func):
    def inner():
        return f"<deco3>{func()}</deco3>"
    return inner

@deco1
@deco2
@deco3
def foo():
    return "古明地觉"

print(foo())
```

打印结果是什么呢？可以简单的分析一下。

解释器还是从上到下解释，当执行到**@deco1**的时候，肯定要装饰了，但是它下面的哥们不是函数、也是一个装饰器，于是说：要不哥们，你先装饰。然后执行**@deco2**，但它下面还是一个装饰器，于是重复了刚才的话，把皮球踢给**@deco3**。

当执行**@deco3**的时候，发现下面终于是一个普通的函数了，于是装饰了。

当deco3装饰完毕之后，**foo=deco3(foo)**。然后deco2发现deco3已经装饰完毕，那么会对deco3装饰的结果再进行装饰，此时**foo=deco2(deco3(foo))**；同理，再经过deco1的装饰，最终得到了**foo = deco1(deco2(deco3(foo)))**。

于是最终输出：

```
# <deco1><deco2><deco3>古明地觉</deco3></deco2></deco1>
```

所以当有多个装饰器的时候，会从下往上装饰；然后执行的时候，会从上往下执行。

以上就是装饰器相关的内容，内容不多，因为核心都在上一篇介绍的**闭包**当中。当然啦，如果文章就这么结束了，感觉有点水了，所以我们再以问答的方式回顾一下前面的内容。

### 问：Python 中有几个名字空间，分别是什么？Python 变量以什么顺序进行查找？

总共有4个名字空间，分别是：

- 局部名字空间；
- 闭包名字空间；
- 全局名字空间；
- 内置名字空间；

Python 查找变量时，依次检查 局部 、闭包、全局、内置 这几个名字空间，直到变量被找到为止。如果几个空间都遍历完了还没找到，那么会抛出 NameError。以上也被称为 LEGB 规则。

> 注意：理解的时候可以按照 LEGB 规则来理解，但我们要知道局部变量和闭包变量是在 f_localsplus 中静态查找的。

### 问：如何在一个函数内部修改全局变量？

在函数内部用 global 关键字将变量声明为全局，然后再进行修改：

```
a = 1

def f():
    # 表示 a 是一个全局变量
    global a
    a = 2

print(a)  # 1
f()
print(a)  # 2
```

或者获取global名字空间，然后通过字典进行修改，因为全局变量是通过字典来存储的。

```
a = 1

def f():
    globals()["a"] = 2

print(a)  # 1
f()
print(a)  # 2
```

### 问：不使用 def 关键字的话，还有什么办法可以创建函数对象？

根据 Python 的对象模型，实例对象可以通过调用类型对象来创建。而函数的类型对象，虽然没有直接暴露给我们，但我们可以通过函数对象的__class__属性找到：

```
def f():
    pass
print(f.__class__)  # <class 'function'>
print((lambda: None).__class__)  # <class 'function'>
```

事实上，Python 将函数的类型对象暴露在 types 模块中，可通过 FunctionType 访问到：

```
from types import FunctionType

def f():
    pass
```

```
print(FunctionType)  # <class 'function'>
print(f.__class__ is FunctionType)  # True
```

当然它干的事情和我们本质上是一样的，我们看一下源码怎么实现的：

```
def _f(): pass
FunctionType = type(_f)
```

然后创建函数时，我们就可以根据类型对象来创建了，我们之前已经见过了。大部分情况下，只需传递3个参数即可：PyCodeObject、名字空间、函数名（全限定名）。但其实可以传递五个参数：

- PyCodeObject 对象；
- 全局名字空间；
- 函数名（__qualname__）；
- 默认值；
- 闭包变量；

```
def f(v):
    global value
    value = v

g = {}
# 对于new_f而言，g 就是它的全局名字空间
# 所以设置的全局变量 value 会体现在 g 中
new_f = type(f)(f.__code__, g, "new_f")
new_f(16)
print(g)  # {'value': 16}

new_f("古明地觉")
print(g)  # {'value': '古明地觉'}
```

是不是奇怪的知识又增加了呢？但还是那句话，这种做法没有什么实际用途，只是让我们能够更好地理解函数的机制。

**问：请介绍装饰器的运行原理，并说说你对 @xxx 这种写法的理解？**

装饰器用于包装函数对象，在不修改函数源码和调用方式的前提下、达到修改函数行为的目的。它的本质是高阶函数加上闭包，而**@xxx**只是它语法糖。

```
class Deco:

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("++++++")
        res = self.func(*args, **kwargs)
        print("******")
        return res

@Deco
def foo(a, b):
    return a + b

print(foo(1, 2))
"""
++++++
******
3
"""
```

还是那句话，装饰器本质是高阶函数加上闭包，而很多语言都有闭包，也可以多层函数嵌套。但是对于Python而言，装饰器显得格外的优雅。

> flask框架就用到了大量的装饰器，比如：@app.route("/")，不得不说，flask的作者真的是非常喜欢使用装饰器。还有它们团队开发的、用于处理命令行参数的click模块，也是大量使用了装饰器。

**问：Python 中的闭包变量(外层作用域的变量)可以被内部函数修改吗?**

显然是可以的，有两种方式：一种是通过nonlocal关键字，另一种是通过获取闭包变量的方式。

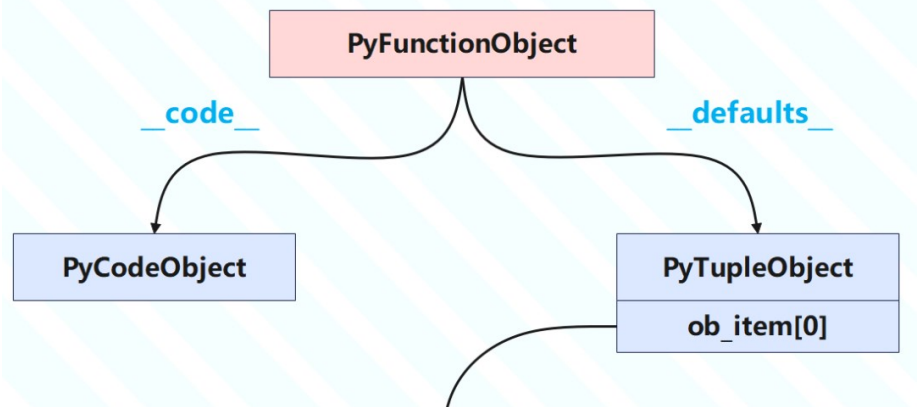
```
def f1():
    value = 0
    def f2():
        return value
    return f2

f = f1()
print(f())  # 0
f.__closure__[0].cell_contents = ">>>"
print(f())  # >>>
```

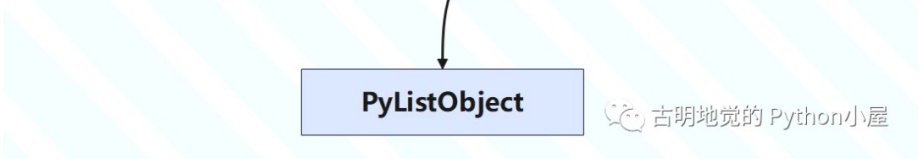
**问：执行以下程序将输出什么内容？并试着解释其中的原因。**

```
def add(n, l=[]):
    l.append(n)
    return l

print(add(1))  # [1]
print(add(2))  # [1, 2]
print(add(3))  # [1, 2, 3]
```

出现这种问题的原因就在于，函数在创建时便完成了默认值的初始化，并保存在函数对象的 __defaults__ 字段中，并且是不变的，永远是那一个对象：

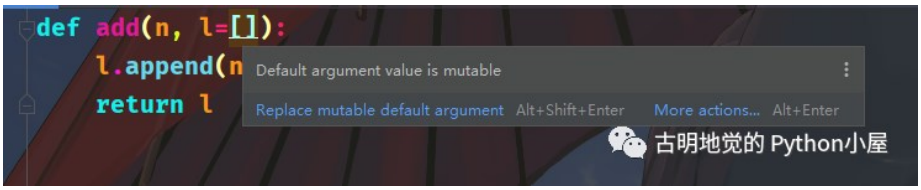
```
def add(n, l=[]):
    l.append(n)
    return l

print(add.__defaults__[0])  # []

print(add(1))  # [1]
print(add.__defaults__[0])  # [1]

print(add(2))  # [1, 2]
print(add.__defaults__[0])  # [1, 2]

print(add(3))  # [1, 2, 3]
print(add.__defaults__[0])  # [1, 2, 3]
```






显然在函数执行的时候，如果我们没有传递参数，那么虚拟机会从函数对象中取出默认值并设置到 f_localsplus（局部变量对应的内存区域）里面。但列表是可变对象，因此采用append的方式，显然每一次都会有变化的，因为指向的是同一个列表。

所以在给参数设置默认值的时候，不要设置成可变对象。如果你的 IDE 比较智能的话，比如PyCharm，那么会给你抛出警告。




我们看到飘黄了，因为默认参数的值是一个可变对象。

到目前为止，我们关于函数的内容就全部介绍完了，可以好好体会一下函数的底层实现。

下一篇文章我们来介绍生成器。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》65. 生成器的实现原理（上）

下一篇 ›
《源码探秘 CPython》63. 闭包是怎么实现的？

喜欢此内容的人还喜欢

Golang编程系列第二期：Go数据类型--基本类型，自定义类型、数组、切片
i运维

C++构造函数总结
AI研修 潜水摸大鱼

力扣 428. 序列化和反序列化 N 叉树 DFS
钰娘娘知识汇总