深入理解Python内部函数和闭包，和它们的应用场景【Python进阶】

本文以内部函数为主线，深入讲解内部函数和闭包的应用场景和原理，学会后你的Python水平会再上一个台阶，对工作面试或实战应用都会很有帮助。

本文包括：

1. 函数是一等公民
2. 内部函数定义

3. 闭包
4. 应用场景 - 封装
5. 应用场景 - 函数生成器
6. 函应用场景 - 装饰器
7. 闭包实现原理

阅读到最后可以获得本文PDF资料和源代码下载，建议搜藏。

函数是一等公民

Python是面向对象的编程语言，对象是Python的一等公民，我们常用的字符串**str**，整数**int**，和其他**变量**都是对象。

函数也是对象，所以也是一等公民，这就意味着它和变量一样

1. 可以作为参数被传递
2. 可以在函数内部定义
3. 可以作为函数返回值
4. 可以被打印，有自己的类型
5. 函数可以赋值给变量

```
def say_hello():
    print('hello')

print(say_hello)

def say_something(some_func):
    for _ in range(3):
        some_func()

say_something(say_hello)
```

执行结果：

```
<function say_hello at 0x7ff3d35b9160>
hello
```

```
hello
hello
```

内部函数

把函数的内部定义函数，就是内部函数（有点像废话，但就那么个意思）。

```
def outter():
    print('我是外部函数')
    def inner():
        print('我是outter的内部函数')
    print('调用内部函数')
    inner()
    print('我再次调用内部函数，自己家的想用就用，随时用')
    inner()
    print('还可以返回给大家共用')
    return inner

#调用外部函数，并接受返回值
func = outter()

#调用outter返回的内部函数
print('在外部调用内部函数')
func()
```

注意： 调用的时候加小括号`inner()`，作为参数或者返回值的时候不加小括号`inner`，是引用这个函数对象。

执行结果：

```
我是外部函数
调用内部函数
我是outter的内部函数
我再次调用内部函数，自己家的想用就用，随时用
我是outter的内部函数
还可以返回给大家共用
在外部调用内部函数
我是outter的内部函数
```

闭包

如果内部函数只是把函数定义在函数的内部，那就没有多大意思了，它还有一个很大的特点，正因为这个特点，它才被称为**闭包clsure**。

> 学过JavaScript的非小白同学可能会对这个概念很熟悉。

内部函数还有一个很重要的特性：

1. 可以访问它所属的外部函数的局部变量，这些变量被称为nonlocal，或者enclosing变量
2. 可以携带这些nonlocal变量，让它们不会被回收

所以说Python中的闭包就是内部函数，准确点是使用了nonlocal变量的内部函数。

```
import random
def create_room():
    room_no = random.randint(1, 100)
    print(f'我创建了房间号：{room_no}')

    def toilet():
        print(f'我是{room_no}的内部厕所')
    print('上厕所')
    toilet()
    print('我再次上厕所，自己家的想用就用，随时用')
    toilet()
    print('还可以共享给大家共用')
    return toilet

#调用外部函数，并接受返回值
toilet = create_room()

#调用outter返回的内部函数
print('在外部使用内部厕所')
toilet()

print('在外部再次使用内部厕所')
toilet()
```

运行结果：

```
我创建了房间号：52
上厕所
我是52的内部厕所
我再次上厕所，自己家的想用就用，随时用
我是52的内部厕所
还可以共享给大家共用
在外部使用内部厕所
我是52的内部厕所
在外部再次使用内部厕所
我是52的内部厕所
```

- 在调用一个create_room的时候临时生成了房间号，一个局部变量room_no。
- 在内部函数toilet中可以直接访问外部函数的局部变量，这是内部函数的特性。
- 局部变量room_no本来在函数执行完就释放的，但由于内部函数toilet引用了它就不会被释放了，在外部调用的时候仍然可以引用到。这就形成了闭包。

说的这么玄乎，其实就是内部函数使用了外部函数的局部变量，所以局部变量被内部函数给封存了，也就不会释放了。

nonlocal关键词

内部函数也可以改写外部函数的变量值，但需要使用nonlocal关键词声明这是外部的变量。

> 回忆一下：函数内部修改全局变量，需要使用global关键词。

```python
import random

def create_room():
    room_no = random.randint(1, 100)
    print(f'我创建了房间号：{room_no}')

    def toilet():
        nonlocal room_no
        room_no = random.randint(1, 100)
        print(f'我是{room_no}的内部厕所')
```

```
    print('上厕所')
    toilet()
    print(f'房间号：{room_no}')
    print('我再次上厕所，自己家的想用就用，随时用')
    toilet()
    print(f'房间号：{room_no}')
    print('还可以共享给大家共用')
    return toilet


#调用外部函数，并接受返回值
toilet = create_room()

#调用outter返回的内部函数
print('在外部使用内部厕所')
toilet()

print('在外部再次使用内部厕所')
toilet()
```

使用nonlocal在内部函数改变外部变量room_no的值，所以每次上厕所，都会改变房间号（很神奇的房间😆）：

```
我创建了房间号：39
上厕所
我是43的内部厕所
房间号：43
我再次上厕所，自己家的想用就用，随时用
我是66的内部厕所
房间号：66
还可以共享给大家共用
在外部使用内部厕所
我是52的内部厕所
在外部再次使用内部厕所
我是29的内部厕所
```

其实内部函数就这么点东西了（后面再说它的实现原理），现在来看到底有什么实实在在的用处。

下面来说3个应用场景：

应用场景 - 封装

写在内部是因为只有在内部才有用，外部根本不需要，也不想让他们使用，就像上面的内部厕所的例子，实际上是不可能在外面使用的。这种场景叫做**封装**。

```
import random

def create_room():
    room_no = random.randint(1, 100)
    print(f'我创建了房间号：{room_no}')

    def toilet():
        print(f'欢迎进入{room_no}的VIP厕所')
        print('冲水')
        print('请君入厕')
        print('洗手')
        print('热毛巾')
        print('欢迎下次光临')

    print('上厕所')
    toilet()
    print('上厕所')
    toilet()
    print('上厕所')
    toilet()

#调用外部函数，并接受返回值
create_room()
```

- 上厕所是create_room所独有的配方，不希望外面使用
- 上厕所的过程独有配方是比较复杂的，有必要封装到函数内，否则每次上厕所都要重复这些代码

再总结一下：

1. 封装一方面不希望对外暴露函数
2. 也为了方便内部的重用

应用场景 - 函数生成器

内部函数可以方便的生成新的函数，看这个例子：

```python
def team_maker(type, level, temperature):
    '''
    type: 品种，如绿茶，红茶
    level: 等级，特级，一级，二级
    temper: 温度
    '''
    def tea(water):
        print('正在沏茶中')
        print(f'{type},{level}, {temperature}')
        print(f'{water}毫升给您冲好了')
    return tea


#创建符合我口味的沏茶函数
mytea = team_maker('绿茶', '特级', '66.6')
#创建符合她口味的沏茶函数
herteam = team_maker('红茶', '一级', '88.8')

print('我们来一杯')
mytea(500)
herteam(300)
print('多喝点')
mytea(800)
herteam(600)
print('完了，我喝醉了...，因为有她')
```

- 沏茶需要传入多个参数，有点麻烦，而每个人的口味相对比较固定
- 我们用内部函数创建了一个符合我口味的沏茶函数，以后调用这个函数就行了。我也给她创建了一个符合她口味的沏茶函数。

运行结果：

```
我们来一杯
正在沏茶中
绿茶,特级, 66.6
500毫升给您冲好了
正在沏茶中
红茶,一级, 88.8
300毫升给您冲好了
多喝点
正在沏茶中
绿茶,特级, 66.6
800毫升给您冲好了
正在沏茶中
红茶,一级, 88.8
600毫升给您冲好了
完了, 我喝醉了...
```

应用场景 - 装饰器

装饰器对Python至关重要。这也是内部函数的主要使用场景。

写到这里忽然有点累了，我想起来我有两篇还不错的装饰器的文章，直接拿来看吧。我就不重复码字了。

结合内部函数的文章和装饰器的文章，应该会通透了。

这两篇文章见本文底部相关阅读前两篇。

闭包实现原理

闭包携带了外部函数的变量，所以可以访问这些变量，而这些变量也不会被释放。具体是怎么实现的呢?

答案就1个字：**__closure__属性**。Python给内部函数添加了这个属性来携带内部函数用到的外部函数中的变量。

```
import random


def create_room():
    room_no = random.randint(1, 100)
    print(f'我创建了房间号：{room_no}')

    def toilet():
        print(f'我是{room_no}的内部厕所')
    return toilet


print('调用外部函数')
toilet = create_room()

print('打印一下toilet函数的变量，其中有一个是__closure__')
print(dir(toilet))
print('__closure__是一个包含它携带的变量的元组')
print(toilet.__closure__)
print('__closure__元组里是cell，通过cell_contents可以访问所携带的变量值')
print(toilet.__closure__[0].cell_contents)
```

执行结果：

```
import random


def create_room():
    room_no = random.randint(1, 100)
    print(f'我创建了房间号：{room_no}')

    def toilet():
        print(f'我是{room_no}的内部厕所')
    return toilet


print('调用外部函数')
toilet = create_room()
```

```
print('打印一下toilet函数的变量，其中有一个是__closure__')
print(dir(toilet))
print('__closure__是一个包含它携带的变量的元组')
print(toilet.__closure__)
print('__closure__元组里是cell，通过cell_contents可以访问所携带的变量值')
print(toilet.__closure__[0].cell_contents)
```

点赞是美德

你动动手，就是对我最大的鼓励。本文内容干货，首先建议收藏。

如果对你有帮助，请**点赞**，**在看**，**转发**。谢谢！

关注公众号回复**内部函数**，获取本文PDF文档和源码。