> 2021/2/16
>
> Feynman Technique
>
> Reference

装饰器 @decorator

何谓装饰

我们有一个函数, 想知道每次调用它都用了多长时间

```
def my(n):
    result = 0
    for i in range(n + 1):
        result += i
    return result
```

一种方法是我们修改函数为这个样子:

```
def my2(n):

    print("accumulate 开始运行")
    start = time.time()

    result = 0
    for i in range(n + 1):
        result += i

    end = time.time()
    print("accumulate 结束运行")
    print(f"花费时间为{int(end-start)}ms")

    return result
```

你的舍友也有一个函数, 想让你帮忙测速:

```
def gauss(n):
    result = (1 + n) * n / 2;
    return result
```

我们当然也可以再用上面的方法修改函数, 来实现测速.

不过这既破坏了原函数本身, 而且每当遇到一个新的函数都要修改, 太麻烦了!

然而, 我们可以定义一个 calc_time() 闭包函数, 它能实现给传入的函数前后增加计时的功能:

```
def calc_time(func):

    def func2(*args, **kwargs):

        print(f"{func.__name__} 开始运行")
        start = time.time()

        return_value = func(*args, **kwargs)

        end = time.time()
        print(f"{func.__name__} 结束运行")
        print(f"花费时间为{int(end-start)}ms")

        return return_value

    return func2
```

我们用这个闭包来得到新的带测速功能的函数, 然后调用新得到的函数即可:

```
# 得到新的函数
new_my = calc_time(my)
new_gauss = calc_time(gauss)

# 调用并测速
new_my(10**7)
new_gauss(10**7)
```

@ 运算符

用一个闭包来给很多的函数增添功能, 就是所谓"装饰"了.

而且在 python 中, 上面的写法还可以简化. 我们只需要在定义函数时写成这个样子:

```
@calc_time
def foo(n):
    result = sum(range(n))
    return result
```

这就等价于:

```
def foo(n):
    result = sum(range(n))
```

```
    return result
foo = calc_time(foo)
```

也就是说如果我们只想得到被装饰后的函数, 那么这种通过 `@` 的写法可以简化代码.

也就是说,

```
@dec
def f():
    pass
```

就等价于

```
def f():
    pass
f = dec(f)
```

单从语法的角度看, `@` 右边的东西只要是一个函数就可以.

所以如果我们见到下述写法:

```
@f(x, y)
def g():
    pass
```

其实就是因为 `f(x, y)` 返回值是一个函数, 然后被 `@` 的其实是 `f(x, y)` 的返回值.

举个例子: 比如我们想定制上述测速装饰器的功能, 可以选择是否打印"开始运行"和"结束运行"这两行字. 可以改写如下:

```
def calc_time(show_start, show_end):
    def decorator(func):
        def func2(*args, **kwargs):
            if show_start:
                print(f"{func.__name__} 开始运行")
            import time
            start = time.time()
            return_value = func(*args, **kwargs)
            end = time.time()
            if show_end:
                print(f"{func.__name__} 结束运行")
            print(f"花费时间为{int(end-start)}ms")
            return return_value
        return func2
    return decorator

@calc_time(show_start=True, show_end=False)
def test(N):
    s = 1
```

```
    for i in range(N):
        s += 1

test(10**7)
```

也就是说 `calc_time(show_start=True, show_end=False)` 返回值是 decorator 函数, 然后相当于 `@decorator`

不一定是闭包

最后说两句, 虽然我们用测速这个话题引出, 但 `@` 这个语法本身有很多别的用法.

比如我们只是想把某个def块中的东西作为一个代码块来执行:

```
def just_run(f):
    f()
    return "Done!"

@just_run
def just_a_block_of_code():
    print("code line 1")
    print("code line II")
    print("code line 三")
```

或者比如 Flask 框架中, 我们可能会在代码中这么写:

```
@app.route("/GO", methods=["POST"])
def Remainder():
    numIn = int(request.form["numIn"])
    Mode = request.form.getlist("MODE")
    Msg = ""
    for divisor in Mode:
        tempDivisor = int(divisor)
        Msg += f"{numIn}除以{tempDivisor}的余数是{numIn % tempDivisor}!<br>"
    return render_template("remainder.html", rtnMsg=Msg)
```

然后我们去翻看 app.route 的源代码:

```
def route(self, rule, **options):

    def decorator(f):
        endpoint = options.pop("endpoint", None)
        self.add_url_rule(rule, endpoint, f, **options)
        return f

    return decorator
```

所以上面我们的代码中的 `@app.route("/GO", methods=["POST"])` 就相当于:

```
app.add_url_rule('/GO', None, Remainder, {"methods":["POST"]})
```