链滴

python 装饰器在接口自动化测试中的应用

作者：zyjImmortal
原文链接：https://ld246.com/article/1589952739988
来源网站：链滴

在讲解装饰器在接口自动化测试项目的应用之前，我们先来介绍一下python装饰器到底是个什么

装饰器

说装饰器就不得不提一下函数这个一等公民了，在python中函数有几个特性先来了解一下

函数的一些特性

- 函数也是对象

在python中函数也是对象，可以把函数赋值给变量，比如下面这样：

```
def func(message):
    print("打印一条message: {}".format(message))

send_message = func
send_message("123")
```

我们把函数 func 赋予了变量 send_message，这样之后你调用 send_message，就相当于是调用函数 func()

- 把函数当做参数

函数也可以当做参数传递给另一个函数使用，比如：

```
def func(message):
    print("打印一条message: {}".format(message))

def call_func(func, message):
    func(message)
```

- 函数的嵌套

函数的嵌套就是说在函数里再定义一个函数，比如这样：

```
def call_func(message):
    def func(message):
        print("打印一条message: {}".format(message))
    return func(message)
```

上面在call_func的内部又定义了一个函数func，并在call_func里调用了这个内部的函数，调用后作为call_func的返回值返回

- 函数的返回值也可以是函数对象

我们修改一下上面的例子。如下：

```
def call_func():
    def func(message):
        print("打印一条message: {}".format(message))
    return func

result = call_func()
result("hello world")
```

函数 call_func() 的返回值是函数对象 func 本身，之后，我们将其赋予变量 result，再调用 result（‘ello world’），最后输出了'打印一条message: hello world'.

简单的装饰器

```
def my_decorator(func):
    def wrapper():
        print('wrapper of decorator')
        func()
    return wrapper

def greet():
    print('hello world')

greet = my_decorator(greet)
greet()

# 输出
wrapper of decorator
hello world
```

变量 greet 指向了内部函数 wrapper()，而内部函数 wrapper() 中又会调用原函数 greet()，因此，后调用 greet() 时，就会先打印'wrapper of decorator'，然后输出'hello world'。这里的函数 my_decorator() 就是一个装饰器，它把真正需要执行的函数 greet() 包裹在其中，并且改变了它的行为，但原函数 greet() 不变.

语法糖@

```
def my_decorator(func):
```

```
    def wrapper():
        print('wrapper of decorator')
        func()
    return wrapper

@my_decorator
def greet():
    print('hello world')

greet()
```

这里的@，我们称之为语法糖，@my_decorator就相当于前面的greet=my_decorator(greet)语句，只不过更加简洁。因此，如果你的程序中有其它函数需要做类似的装饰，你只需在它们的上方加上@decorator就可以了，这样就大大提高了函数的重复利用和程序的可读性。

带参数的装饰器

如果原函数 greet() 是需要接收参数，因为被装饰函数是在装饰器里执行，那就需要把函数接收的参传递到装饰器里，该怎么办呢？很简单，只需在装饰器的嵌套函数上增加入参就行，比如

```
def my_decorator(func):
    def wrapper(message):
        print('wrapper of decorator')
        func(message)
    return wrapper


@my_decorator
def greet(message):
    print(message)


greet('hello world')

# 输出
wrapper of decorator
hello world
```

不过一般不这么一个个的写，麻烦，直接这样搞：

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print('wrapper of decorator')
        func(*args, **kwargs)
    return wrapper
```

装饰器也是可以接收参数的

装饰器还有更大程度的灵活性，可以接受自己定义的参数，可以给装饰器本身传递参数

```
def repeat(num):
    def my_decorator(func):
        def wrapper(*args, **kwargs):
```

```
            for i in range(num):
                print('wrapper of decorator')
                func(*args, **kwargs)
        return wrapper
    return my_decorator


@repeat(4)
def greet(message):
    print(message)

greet('hello world')
```

类装饰器

类也可以作为装饰器。类装饰器主要依赖于函数__call__()，每当你调用一个类的示例时，函数__call__()就会被执行一次。

```
class Request:
    def __init__(self, func):
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print('num of calls is: {}'.format(self.num_calls))
        return self.func(*args, **kwargs)

@Request
def example():
    print("hello world")

example()

# 输出
num of calls is: 1
hello world

example()

# 输出
num of calls is: 2
hello world

...
```

这个类装饰器还不支持接收参数，后面我们实战的装饰器时可以支持结束参数的。

装饰器在接口自动化测试项目中应用

至此我们介绍完了装饰器，下面我们基于之前的理论，来进行一次实战。

需求是希望通过装饰器来实现接口的请求，能够自定义请求方法、请求的根路径、公共参数、header设置等功能。

```
class Request:

    def __init__(self, url='', method='get'):
        """
        self.url = url  # 请求路径
        self.method = method # 请求方法
        self.func_return = None # 被装饰器标记的方法的返回参数
        self.func_im_self = None  # 被装饰器标记的方法的类的实例
        self.session = None # 当前使用的会话对象

    def __call__(self, func):
        self.func = func
        self.is_class = False
        try:
            if inspect.getfullargspec(self.func).args[0] == 'self':
                self.is_class = True
        except IndexError:
            pass

        def fun_wrapper(*args, **kwargs):
          # 调用被装饰标记的方法，这个方法会返回请求接口所需要的返回值
            self.func_return = self.func(*args, **kwargs) or {}
            self.func_im_self = args[0] if self.is_class else object
            self.create_url()
            self.create_session()
            self.session.headers.update(getattr(self.func_im_self, 'headers', {}))
            self.decorator_args.update(getattr(self.func_im_self, 'common_params', {}))
            self.decorator_args.update(self.func_return)
            return Request(self.method, self.url, self.session)
        return fun_wrapper

def create_url(self):
    """
    生成http请求的url，跟路径和接口路由进行拼接
    """
    base_url = getattr(self.func_im_self, 'base_url', '')
    self.url = self.func_return.pop('url', None) or self.url
    self.url = ''.join([base_url, self.url])
```

使用的时候要定义一个类，比如下面这样：

```
class AdvertService:

    def __init__(self):
        self.common_params = {} # 定义接口请求的公共参数
        self.headers = {} # 定义请求的header
        self.base_url = self._config.AD_ADMIN_ROOT_URL

    @Request(url= “/v3/advert/create” , method='post')
    def _create_ad(self, advert: Advert):
        return dict(json=advert)
```

上面的header会被自动的添加的session的header里，common_params也会被添加到参数里，base url和装饰器里传的url会被拼接成一个完整的url去请求接口。

以上实战的具体代码，当然这只是一部分，并不是完整的，后面争取写个系列文章，将这个接口自动测试项目整体介绍一下，欢迎大家关注，多多交流！

原文地址https://www.immortalp.com/articles/2020/05/20/1589952737702.html