python3之装饰器

2019.04.23 17:46:26 字数 847 阅读 650

一、装饰器介绍

装饰器也是一个函数，它是让其他函数在不改变变动的前提下增加额外的功能。
装饰器是一个闭包，把一个函数当作参数返回一个替代版的函数，本质是一个返回函数的函数（即返回值为函数对象）。
python3支持用@符号直接将装饰器应用到函数。
装饰器工作场景：插入日志、性能测试、事务处理等等。
函数被装饰器装饰过后，此函数的属性均已发生变化，如名称变为装饰器的名称。

1. 简单的装饰器

1.1. 被装饰的函数不带参数

```
"""入门装饰器：函数功能不带参数"""
def my_decorator(func):
    def inner():
        print("**********")
        print("要添加的功能代码")
        func()
    return inner

# script1()函数调用装饰器的第一种方法
def script1():
    print("测试")
runScript1 = my_decorator(script1)    # 运行script()函数的同时添加有my_decorator()函数的功能
runScript1()
# script1()函数调用装饰器的第二种方法：使用@符号，简单明了
@my_decorator
def script1():
    print("测试")
script1()
```

1.2. 被装饰的函数带参数

可变参数*args和关键字参数*kwargs添加函数通用的装饰器

```
"""入门装饰器：函数带参数"""
def my_decorator(func):
    def inner(*args, **kwargs):     # 可变参数*args和关键字参数**kwargs
        print("**********")
        print("要添加的功能代码")
        func(*args, **kwargs)
    return inner

# script2()函数调用装饰器的第一种方法：了解即可
def script2(arg):
    print("测试：%s" % arg)
runScript2 = my_decorator(script2)
runScript2("aaa")
# script2()函数调用装饰器的第二种方法：使用@符号，目前使用此方法
@my_decorator
def script2(arg):
    print("测试：%s" % arg)
script2("aaa")
```

2. 装饰器带参数

python3之装饰器


```
    def outer(func):
        def inner(*args, **kwargs):
            print("********")
            print("添加带装饰器参数%s的功能代码" % self.name)
            func(*args, **kwargs)
        return inner
    return outer

@my_decorator(name='settings')
def script3(arg):
    print("测试----%s" % arg)
script3("bbb")
```

3. 基于类封装的装饰器

__call __()方法是将实例成为一个可调用对象（即callable对象），同时不影响实例的构造，但可以改变实例的内部值。

3.1. 基于类封装的不带参数装饰器

通过类封装装饰器的实现方法：先通过构造函数__init __()传入函数；再通过__call __方法重载，并返回一个函数。

```
"""基于类封装的不带参数装饰器"""
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("********")
        print("要添加的功能代码")
        return self.func(*args, **kwargs)

@MyDecorator
def script4(arg):
    print("测试----%s" % arg)
script4("ccc")
```

3.2. 基于类封装的带参数装饰器

通过类封装装饰器的实现方法：先通过构造函数__init __()传入装饰器参数；再通过__call __方法传入被装饰的函数，并返回一个函数。

```
"""基于类封装的带参数装饰器"""
class MyDecorator:
    def __init__(self, name):
        self.name = name

    def __call__(self, func):
        def inner(*args, **kwargs):
            print("********")
            print("添加带装饰器参数%s的功能代码" % self.name)
            func(*args, **kwargs)
        return inner

@MyDecorator(name="settings")
def script4(arg):
    print("测试----%s" % arg)
script4("ddd")
```

二、常用的内置装饰器

1. @property装饰器

python3之装饰器

rr1990 关注 赞赏支持

添加装饰器@property时，函数类型是返回值的类型：如，<class 'str'>

- property对象的setter方法：表示给属性添加设置功能，即可修改属性值。
  若未添加设置属性，就设置新值，则会引发错误AttributeError: can't set attribute。
- property对象的deleter方法：表示给属性添加删除功能
  若添加删除属性，就删除属性则会引发错误AttributeError: can't delete attribute。

```
"""@property装饰器"""
class Test1:
    def __init__(self, name):
        self.__name = name

    @property               # 将函数由方法变为属性
    def get_name(self):
        return self.__name

    @get_name.setter            # 添加设置属性
    def get_name(self, value):
        if not isinstance(value, str):
            raise TypeError("参数应为字符串类型，但实际是%s类型" % type(value))
        else:
            self.__name = value

    @get_name.deleter           # 添加删除属性
    def get_name(self):
        del self.__name

test1 = Test1("launcher")
# 获取get_name类型
print(type(test1.get_name))      # 结果： <class 'str'>
# 获取get_name属性值
print(test1.get_name)             # 结果：launcher
# 给get_name属性设置新值：添加设置属性需使用装饰器@property的setter函数；
test1.get_name = "赋新值"
print(test1.get_name)           # 结果：赋新值
# 删除get_name属性：删除属性需使用装饰器@property的deleter函数；
del test1.get_name
print(test1.get_name)           # 结果:报错(AttributeError: 'Test1' object has no attribute '_Tes


"""@property实例：加减法运算"""
class Test2:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @property
    def add(self):
        return self.a + self.b

    @property
    def reduce(self):
        return self.a - self.b
print(Test2(3, 1).add)      # 结果：4
print(Test2(5, 2).reduce)   # 结果：3
```

2. 类对象中的方法

类对象中的方法：实例方法、类方法和静态方法

- 实例方法：函数中的第一个参数为self的方法
- 静态方法：使用@staticmethod装饰器来将类中的函数定义为静态方法。
  类中创建的一些方法，但该方法并不需要引用类或实例。静态方法通过类直接调用，无需创建对象，也无需传递self。
- 类方法：使用@classmethod装饰器来装饰类中的函数定义为类方法

python3之装饰器


```
"""实例方法、静态方法@staticmethod、类方法@classmethod"""
class Student:
    description = "学员统计信息"

    def __init__(self, name, age, sex):
        self.name = name
        self.age = age
        self.sex = sex

    def function(self, fun_type):
        return fun_type

    def instance_method(self):          # 实例方法
        use_type = self.function("实例方法")
        print("------%s------" % use_type)
        print(Student.description)
        print(self.name + '_' + str(self.age) + '_' + self.sex)

    @staticmethod           # 静态方法
    def static_method():
        student_info = Student("xiaoxiao", 20, "female")
        use_type = student_info.function("静态方法")
        print("------%s------" % use_type)
        print(Student.description)
        print(student_info.name + '_' + str(student_info.age) + '_' + student_info.sex)

    @classmethod        # 类方法
    def class_method(cls):
        student_info = cls("xiaoming", 23, "male")
        use_type = student_info.function("类方法")
        print("------%s------" % use_type)
        print(Student.description)
        print(student_info.name + '_' + str(student_info.age) + '_' + student_info.sex)

    def call_different_method(self):
        print("------同一类对象中调用实例方法、静态方法、类方法------")
        self.instance_method()
        self.static_method()
        self.class_method()

# 实例方法
Student("xiaohong", 19, "female").instance_method()
# 静态方法
Student.static_method()
# 类方法
Student.class_method()
# 同一类对象中某个函数调用实例/静态/类方法
Student("xiaohong", 19, "female").call_different_method()
```

三、使用三方已封装的装饰器

- 三方模块decorator

  先安装decorator模块，再导入from decorator import decorator
- 三方模块wrapt

  先安装wrapt模块，再导入import wrapt

```
# decorator三方模块
from decorator import decorator
@decorator
def My_decorator(func, *args, **kwargs):
    print("********")
    print("添加封装的功能内容")
    return func(*args, **kwargs)

@My_decorator
def testScript2():
    print("待装饰的函数")
testScript2()
```

python3之装饰器

rr1990 关注 赞赏支持

```python
@wrapt.decorator
def My_decorator(wrapped, instance, args, kwargs):
# instance参数即使用不使用也必须保留
    print("********")
    print("添加封装的功能内容")
    return wrapped(*args, **kwargs)

@My_decorator
def testScript1():
    print("待装饰的函数")
testScript1()

"""装饰器带参数"""
def My_decorator(name):
    @wrapt.decorator
    def inner(wrapped, instance, args, kwargs):
        print("********")
        print("添加封装的功能内容，且装饰器参数为%s" % name)
        return wrapped(*args, **kwargs)
    return inner

@My_decorator(set)
def testScript1():
    print("待装饰的函数")
testScript1()
```