Python装饰器简介

这里要介绍一点与算法无关，但对理解Python代码有帮助的知识——Python装饰器。虽然如果我们了解其他编程语言就能很容易看懂Python代码，但面对Python装饰器时却往往感到困惑，因为很多其他编程语言没有类似的概念。

装饰器（Decorator）是Python中一种工具，它允许我们在不修改原有函数代码的情况下，动态地扩展或增强函数的功能。装饰器本质上是一个高阶函数，它接收一个函数作为参数，并返回一个新的函数。

装饰器的作用

装饰器的主要作用是对函数或方法进行功能扩展，同时保持代码的简洁和可维护性。常见的应用场景包括：

- 日志记录：记录函数的调用信息、参数和返回值。
- 性能测试：测量函数的执行时间。
- 权限校验：在调用函数前检查用户权限。
- 缓存结果：对函数的结果进行缓存，避免重复计算。
- 事务处理：自动管理数据库事务。

装饰器的基本原理

装饰器本质上是一个函数，它接受一个函数作为输入，并返回一个新的函数。其核心思想是函数可以作为参数传递，并且函数可以嵌套定义。

一个简单的例子

```
# 定义一个装饰器my_decorator，它接受一个函数作为参数，并返回一个新的函数。
def my_decorator(func):
    def wrapper():
        print("函数执行前的操作")
        func()
        print("函数执行后的操作")
    return wrapper

# 使用装饰器my_decorator来装饰函数say_hello。
@my_decorator
def say_hello():
    print("Hello, World!")

# 调用装饰后的函数say_hello。
say_hello()
```

输出结果为：

```
函数执行前的操作
Hello, World!
函数执行后的操作
```

在这个例子中：

- my_decorator 是装饰器函数。
- wrapper 是内部函数，用于包装原函数。
- @my_decorator 是语法糖，等价于 say_hello = my_decorator(say_hello)。

带参数的装饰器

如果被装饰的函数需要参数，可以通过在wrapper函数中使用*args和**kwargs来支持任意参数。

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"传入的参数: {args}, {kwargs}")
        result = func(*args, **kwargs)
        print("函数执行完成")
        return result
    return wrapper

@my_decorator
def add(a, b):
    return a + b

print(add(3, 5))
```

输出结果为：

```
传入的参数: (3, 5), {}
函数执行完成
8
```

带参数的装饰器函数

如果装饰器本身需要参数，可以在外层再包一层函数。

```python
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

输出结果为：

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

类装饰器

除了函数装饰器，还可以使用类实现装饰器。类装饰器通常通过实现__call__方法来实现。

```
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("函数执行前的操作")
        result = self.func(*args, **kwargs)
        print("函数执行后的操作")
        return result

@MyDecorator
def say_hello():
    print("Hello, World!")

say_hello()
```

输出结果为：

```
函数执行前的操作
Hello, World!
函数执行后的操作
```

内置装饰器

Python 提供了一些内置的装饰器，常用的有：

- @staticmethod：将方法标记为静态方法，无需实例化即可调用。
- @classmethod：将方法标记为类方法，第一个参数是类本身（cls）。
- @property：将方法转换为属性访问。

例如：

```
class MyClass:
    @staticmethod
    def static_method():
        print("这是一个静态方法")

    @classmethod
    def class_method(cls):
        print(f"这是一个类方法，类名是 {cls.__name__}")

    @property
    def name(self):
        return "这是一个属性"

MyClass.static_method()
obj = MyClass()
obj.static_method()
obj.class_method()
print(obj.name)
```

输出结果为：

```
这是一个静态方法
这是一个静态方法
这是一个类方法，类名是 MyClass
这是一个属性
```

装饰器链

多个装饰器可以叠加使用，称为装饰器链。装饰器会从上到下依次应用。

```
def decorator1(func):
    def wrapper():
        print("装饰器1开始")
        func()
        print("装饰器1结束")
    return wrapper

def decorator2(func):
    def wrapper():
        print("装饰器2开始")
        func()
        print("装饰器2结束")
    return wrapper

@decorator1
@decorator2
def say_hello():
    print("Hello, World!")

say_hello()
```

输出结果为：

```
装饰器1开始
装饰器2开始
Hello, World!
装饰器2结束
装饰器1结束
```

注意事项

使用装饰器后，原函数的元信息（如__name__、**doc**）可能会丢失。可以使用functools.wraps解决这个问题。

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("装饰器操作")
        return func(*args, **kwargs)
    return wrapper
```

另外，装饰器会增加函数调用的额外开销，需合理使用。