装饰器

完全遵循开放封闭原则

在不改变原函数的代码以及调用方式的前提下，为其增加新的功能

引入场景

我们现在需要测试一段代码的效率，通常来讲，我们可以写成如下的方式：

```
import time

def index():
    """ There's a lot of code here"""
    time.sleep(0.5)
    print("Welcome to my index")

def timmer(f):
    """Used to test function run time"""

    start_time = time.time()
    f()
    end_time = time.time()

    print(f"The run time is {end_time - start_time}")

timmer(index)

# output
Welcome to my index
The run time is 0.5008871555328369
```

　这段代码大家应该很容易理解，但是这并不符合我们装饰器的规定，因为我们上面讲到过，装饰器是：在不改变原函数的代码以及调用方式的前提下，为其增加新的功能，很明显我们调用index的时候并不是直接index()直接调用，所以这里我们需要运用到简单的闭包知识。

改进版本一

```
import time

def index():
    """ There's a lot of code here"""
    time.sleep(0.5)
    print("Welcome to my index")

def timmer(f):
    """Used to test function run time"""
    def inner():

        start_time = time.time()
        f() # It can be seen as index()
        end_time = time.time()

        print(f"The run time is {end_time - start_time}")
    return inner

index = timmer(index)
index()

# output
Welcome to my index
The run time is 0.5008871555328369
```

这就完成了一个非常非常简单的一个装饰器了，但是我们仔细想一想，假设我们有许许多多的函数全部需要去计算他们的执行效率的话，那么我们需要写许许多多的 xxx = 装饰器(xxx)然后xxx()

这样一想是不是十分的麻烦，我们可以拿一个场景来测试一下，方便大家的理解，我们现在多了一个测试函数，是一个日记函数。

```
import time


def index():
    """ There's a lot of code here"""
    time.sleep(0.5)
    print("Welcome to my index")


def diary():
    """ There's a lot of code here"""
    time.sleep(0.88)
    print("Welcome to my diary")


def timmer(f):
    """Used to test function run time"""
    def inner():

        start_time = time.time()
        f()
        end_time = time.time()

        print(f"The run time is {end_time - start_time}")
    return inner
```

```
index = timmer(index)
diary = timmer(diary)
diary()
index()

# output
Welcome to my diary
The run time is 0.8808135986328125
Welcome to my index
The run time is 0.5008871555328369
```

假如我们需要测试一千个函数，那我们岂不是需要多写一千行吗?

所以我们需要想办法去解决这些问题

改进版本二

```
import time

def timmer(f):
    """Used to test function run time"""
    def inner():

        start_time = time.time()
        f()
        end_time = time.time()

        print(f"The run time is {end_time - start_time}")
    return inner


@timmer  # index = timmer(index)
def index():
    """ There's a lot of code here"""
    time.sleep(0.5)
    print("Welcome to my index")


@timmer  # index = timmer(diary)
def diary():
    """ There's a lot of code here"""
    time.sleep(0.88)
    print("Welcome to my diary")


diary()
index()

# output
Welcome to my diary
The run time is 0.8808534145355225
Welcome to my index
The run time is 0.5008211135864258
```

@timmer 装饰在index上面，那么就相当于：index = timmer(index)

然后我们执行 index() == inner() ---> f() == index()

但是我们怎么解决传参和返回值的问题呢？

改进版本三

```
import time

def timmer(f):
    """Used to test function run time"""
    def inner(*args,**kwargs):

        # args = (1)

        start_time = time.time()
        r = f(*args,**kwargs)  # --> f(1) = index(1)
        end_time = time.time()

        print(f"The run time is {end_time - start_time}")

        return r

    return inner


@timmer  # index = timmer(index)
def index(nums):
    """ There's a lot of code here"""
    time.sleep(0.5)
    print("Welcome to my index")
    return f"in index my nums is {nums}"


@timmer  # index = timmer(diary)
def diary(nums):
    """ There's a lot of code here"""
    time.sleep(0.88)
    print("Welcome to my diary")
    return f"in diary my nums is {nums}"


a = diary(1)
b = index(2)
print(a)
print(b)

# output
Welcome to my diary
The run time is 0.8808488845825195
Welcome to my index
The run time is 0.50087571144104
in diary my nums is 1
in index my nums is 2
```

总结：

标准版装饰器

```
def wrapper(f):
    def inner(*args, **kargs):
        """ Operations before decorated functions """
        ret = f(*args, **kargs)  # ret is the return value of the decorated 
function
        """ Operations before decorated functions """
        return ret  # return value of the decorated function

    return inner
```

装饰器的本质就是函数。