

## 34 | 编程范式游记 ( 5 ) - 修饰器模式

2018-01-25 陈皓

左耳听风

[进入课程 >](#)



在上一篇文章中，我们领略了函数式编程的趣味和魅力，主要讲了函数式编程的主要技术。还记得有哪些吗？递归、Map、Reduce、Filter 等，并利用 Python 的 Decorator 和 Generator 功能，将多个函数组合成了管道。

此时，你心中可能会有个疑问，这个 decorator 又是怎样工作的呢？这就是本文中要讲述的内容，“Decorator 模式”，又叫“修饰器模式”，或是“装饰器模式”。


### Python 的 Decorator

Python 的 Decorator 在使用上和 Java 的 Annotation ( 以及 C# 的 Attribute ) 很相似，就是在方法名前面加一个 @XXX 注解来为这个方法装饰一些东西。但是，Java/C# 的 Annotation 也很让人望而却步，太过于复杂了。你要玩它，需要先了解一堆 Annotation 的类库文档，感觉几乎就是在学另外一门语言。

是语言层面的玩法：一种函数式编程的技巧。


这是我最喜欢的一个模式了，也是一个挺好玩儿的东西，这个模式动用了函数式编程的一个技术——用一个函数来构造另一个函数。

好了，我们先来点感性认识，看一个 Python 修饰器的 Hello World 代码。

 复制代码

```
1 def hello(fn):
2     def wrapper():
3         print "hello, %s" % fn.__name__
4         fn()
5         print "goodbye, %s" % fn.__name__
6     return wrapper
7
8 @hello
9 def Hao():
10     print "i am Hao Chen"
11
12 Hao()
```

代码的执行结果如下：

 复制代码

```
1 $ python hello.py
2 hello, Hao
3 i am Hao Chen
4 goodbye, Hao
```

你可以看到如下的东西：

1. 函数 Hao 前面有个 @hello 的“注解”，hello 就是我们前面定义的函数 hello；
2. 在 hello 函数中，其需要一个 fn 的参数（这就是用来做回调的函数）；

对于 Python 的这个 @注解语法糖（syntactic sugar）来说，当你在用某个 @decorator 来修饰某个函数 func 时，如下所示：

[复制代码](#)

```
1 @decorator
2 def func():
3     pass
```

其解释器会解释成下面这样的语句：

[复制代码](#)

```
1 func = decorator(func)
```

嘿！这不就是把一个函数当参数传到另一个函数中，然后再回调吗？是的。但是，我们需要注意，那里还有一个赋值语句，把 decorator 这个函数的返回值赋值回了原来的 func。

我们再来看一个带参数的玩法：

[复制代码](#)

```
1 def makeHtmlTag(tag, *args, **kwargs):
2     def real_decorator(fn):
3         css_class = " class='{0}'".format(kwargs["css_class"]) \
4             if "css_class" in kwargs else ""
5         def wrapped(*args, **kwargs):
6             return "<"+tag+css_class+">" + fn(*args, **kwargs) + "</"+tag+">"
7         return wrapped
8     return real_decorator
9
10 @makeHtmlTag(tag="b", css_class="bold_css")
11 @makeHtmlTag(tag="i", css_class="italic_css")
12 def hello():
13     return "hello world"
14
15 print hello()
16
```

在上面这个例子中，我们可以看到：`makeHtmlTag`有两个参数。所以，为了让 `hello = makeHtmlTag(arg1, arg2)(hello)` 成功，`makeHtmlTag` 必需返回一个 `decorator`（这就是为什么我们在 `makeHtmlTag` 中加入了 `real_decorator()`）。

这样一来，我们就可以进入到 `decorator` 的逻辑中去了——`decorator` 得返回一个 `wrapper`，`wrapper` 里回调 `hello`。看似那个 `makeHtmlTag()` 写得层层叠叠，但是，已经了解了本质的我们觉得写得很自然。

我们再来看一个为其它函数加缓存的示例：


复制代码

```
1 from functools import wraps
2 def memoization(fn):
3     cache = {}
4     miss = object()
5
6     @wraps(fn)
7     def wrapper(*args):
8         result = cache.get(args, miss)
9         if result is miss:
10             result = fn(*args)
11             cache[args] = result
12         return result
13
14     return wrapper
15
16 @memoization
17 def fib(n):
18     if n < 2:
19         return n
20     return fib(n - 1) + fib(n - 2)
```

上面这个例子中，是一个斐波那契数列的递归算法。我们知道，这个递归是相当没有效率的，因为会重复调用。比如：我们要计算 `fib(5)`，于是其分解成 `fib(4) + fib(3)`，而

而我们用 decorator，在调用函数前查询一下缓存，如果没有才调用，有了就从缓存中返回值。一下子，这个递归从二叉树式的递归成了线性的递归。wraps 的作用是保证 fib 的函数名不被 wrapper 所取代。

除此之外，Python 还支持类方式的 decorator。

 复制代码

```
1 class myDecorator(object):
2     def __init__(self, fn):
3         print "inside myDecorator.__init__()"
4         self.fn = fn
5
6     def __call__(self):
7         self.fn()
8         print "inside myDecorator.__call__()"
9
10 @myDecorator
11 def aFunction():
12     print "inside aFunction()"
13
14 print "Finished decorating aFunction()"
15
16 aFunction()
17
18 # 输出:
19 # inside myDecorator.__init__()
20 # Finished decorating aFunction()
21 # inside aFunction()
22 # inside myDecorator.__call__()
```

上面这个示例展示了，用类的方式声明一个 decorator。我们可以看到这个类中有两个成员：

1. 一个是 `__init__()`，这个方法是在我们给某个函数 decorate 时被调用，所以，需要有一个 `fn` 的参数，也就是被 decorate 的函数。
2. 一个是 `__call__()`，这个方法是在我们调用被 decorate 的函数时被调用的。

我们来看一个实际点的例子。下面这个示例展示了通过 URL 的路由来调用相关注册的函数示例：


[复制代码](#)

```
1 class MyApp():
2     def __init__(self):
3         self.func_map = {}
4
5     def register(self, name):
6         def func_wrapper(func):
7             self.func_map[name] = func
8             return func
9         return func_wrapper
10
11    def call_method(self, name=None):
12        func = self.func_map.get(name, None)
13        if func is None:
14            raise Exception("No function registered against - " + str(name))
15        return func()
16
17 app = MyApp()
18
19 @app.register('/')
20 def main_page_func():
21     return "This is the main page."
22
23 @app.register('/next_page')
24 def next_page_func():
25     return "This is the next page."
26
27 print app.call_method('/')
28 print app.call_method('/next_page')
```

注意：上面这个示例中 decorator 类不是真正的 decorator，其中也没有 `__call__()`，并且，wrapper 返回了原函数。所以，原函数没有发生任何变化。

## Go 语言的 Decorator


Python 有语法糖，所以写出来的代码比较酷。但是对于没有修饰器语法糖这类语言，写出来的代码会是怎么样的？我们来看一下 Go 语言的代码。

 复制代码

```
1 package main
2
3 import "fmt"
4
5 func decorator(f func(s string)) func(s string) {
6     return func(s string) {
7         fmt.Println("Started")
8         f(s)
9         fmt.Println("Done")
10    }
11 }
12
13 func Hello(s string) {
14     fmt.Println(s)
15 }
16
17 func main() {
18     decorator(Hello)("Hello, World!")
19 }
```

可以看到，我们动了一个高阶函数 `decorator()`，在调用的时候，先把 `Hello()` 函数传进去，然后其返回一个匿名函数。这个匿名函数中除了运行了自己的代码，也调用了被传入的 `Hello()` 函数。

这个玩法和 Python 的异曲同工，只不过，Go 并不支持像 Python 那样的 `@decorator` 语法糖。所以，在调用上有些难看。当然，如果要想让代码容易读一些，你可以这样：

 复制代码

```
1 hello := decorator(Hello)
2 hello("Hello")
```

我们再来看一个为函数 log 消耗时间的例子：

 复制代码

```
1 type SumFunc func(int64, int64) int64
2
```

```
6
7 func timedSumFunc(f SumFunc) SumFunc {
8     return func(start, end int64) int64 {
9         defer func(t time.Time) {
10             fmt.Printf("--- Time Elapsed (%s): %v ---\n",
11                 getFunctionName(f), time.Since(t))
12         }(time.Now())
13         return f(start, end)
14     }
15 }
16
17 func Sum1(start, end int64) int64 {
18     var sum int64
19     sum = 0
20     if start > end {
21         start, end = end, start
22     }
23     for i := start; i <= end; i++ {
24         sum += i
25     }
26     return sum
27 }
28
29 func Sum2(start, end int64) int64 {
30     if start > end {
31         start, end = end, start
32     }
33     return (end - start + 1) * (end + start) / 2
34 }
35
36 func main() {
37
38     sum1 := timedSumFunc(Sum1)
39     sum2 := timedSumFunc(Sum2)
40
41     fmt.Printf("%d, %d\n", sum1(-10000, 10000000), sum2(-10000, 10000000))
42 }
```

关于上面的代码：

有两个 Sum 函数，Sum1() 函数就是简单地做个循环，Sum2() 函数动用了数据公式。  
(注意：start 和 end 有可能有负数的情况。)

代码中使用了 Go 语言的反射机制来获取函数名。



再来看一个 HTTP 路由的例子：

复制代码

```
1 func WithServerHeader(h http.HandlerFunc) http.HandlerFunc {
2     return func(w http.ResponseWriter, r *http.Request) {
3         log.Println("--->WithServerHeader()")
4         w.Header().Set("Server", "HelloServer v0.0.1")
5         h(w, r)
6     }
7 }
8
9 func WithAuthCookie(h http.HandlerFunc) http.HandlerFunc {
10    return func(w http.ResponseWriter, r *http.Request) {
11        log.Println("--->WithAuthCookie()")
12        cookie := &http.Cookie{Name: "Auth", Value: "Pass", Path: "/"}
13        http.SetCookie(w, cookie)
14        h(w, r)
15    }
16 }
17
18 func WithBasicAuth(h http.HandlerFunc) http.HandlerFunc {
19    return func(w http.ResponseWriter, r *http.Request) {
20        log.Println("--->WithBasicAuth()")
21        cookie, err := r.Cookie("Auth")
22        if err != nil || cookie.Value != "Pass" {
23            w.WriteHeader(http.StatusForbidden)
24            return
25        }
26        h(w, r)
27    }
28 }
29
30 func WithDebugLog(h http.HandlerFunc) http.HandlerFunc {
31    return func(w http.ResponseWriter, r *http.Request) {
32        log.Println("--->WithDebugLog")
33        r.ParseForm()
34        log.Println(r.Form)
35        log.Println("path", r.URL.Path)
36        log.Println("scheme", r.URL.Scheme)
37        log.Println(r.Form["url_long"])
38        for k, v := range r.Form {
39            log.Println("key:", k)
40            log.Println("val:", strings.Join(v, ""))
41        }
42        h(w, r)
43    }
44 }
```

```
48 }
```

上面的代码中，我们写了多个函数。有写 HTTP 响应头的，有写认证 Cookie 的，有检查认证 Cookie 的，有打日志的.....在使用过程中，我们可以把其嵌套起来使用，在修饰过的函数上继续修饰，这样就可以拼装出更复杂的功能。

[复制代码](#)

```
1 func main() {
2     http.HandleFunc("/v1/hello", WithServerHeader(WithAuthCookie(hello)))
3     http.HandleFunc("/v2/hello", WithServerHeader(WithBasicAuth(hello)))
4     http.HandleFunc("/v3/hello", WithServerHeader(WithBasicAuth(WithDebugLog(hello))))
5     err := http.ListenAndServe(":8080", nil)
6     if err != nil {
7         log.Fatal("ListenAndServe: ", err)
8     }
9 }
```

当然，如果一层套一层不好看的话，我们可以使用 pipeline 的玩法——我们需要先写一个工具函数——用来遍历并调用各个 decorator：

[复制代码](#)

```
1 type HttpHandlerDecorator func(http.HandlerFunc) http.HandlerFunc
2
3 func Handler(h http.HandlerFunc, decors ...HttpHandlerDecorator) http.HandlerFunc {
4     for i := range decors {
5         d := decors[len(decors)-1-i] // iterate in reverse
6         h = d(h)
7     }
8     return h
9 }
```

然后，我们就可以像下面这样使用了。


[复制代码](#)

这样的代码是不是更易读了一些？pipeline 的功能也就出来了。

不过，对于 Go 的修饰器模式，还有一个小问题——好像无法做到泛型，就像上面那个计算时间的函数一样，它的代码耦合了需要被修饰的函数的接口类型，无法做到非常通用。如果这个事解决不了，那么，这个修饰器模式还是有点不好用的。

因为 Go 语言不像 Python 和 Java，Python 是动态语言，而 Java 有语言虚拟机，所以它们可以干许多比较变态的事儿，然而 Go 语言是一个静态的语言，这意味着其类型需要在编译时就要搞定，否则无法编译。不过，Go 语言支持的最大的泛型是 `interface{}`，还有比较简单的 reflection 机制，在上面做做文章，应该还是可以搞定的。

废话不说，下面是我用 reflection 机制写的一个比较通用的修饰器（为了便于阅读，我删除了出错判断代码）。

 复制代码

```
1 func Decorator(decoPtr, fn interface{}) (err error) {
2     var decoratedFunc, targetFunc reflect.Value
3
4     decoratedFunc = reflect.ValueOf(decoPtr).Elem()
5     targetFunc = reflect.ValueOf(fn)
6
7     v := reflect.MakeFunc(targetFunc.Type(),
8         func(in []reflect.Value) (out []reflect.Value) {
9             fmt.Println("before")
10            out = targetFunc.Call(in)
11            fmt.Println("after")
12            return
13        })
14
15     decoratedFunc.Set(v)
16     return
17 }
```

上面的代码动用了 `reflect.MakeFunc()` 函数制作出了一个新的函数。其中的 `targetFunc.Call(in)` 调用了被修饰的函数。关于 Go 语言的反射机制，推荐官方文章


上面这个 `Decorator()` 需要两个参数：

第一个是出参 `decoPtr`，就是完成修饰后的函数。

第二个是入参 `fn`，就是需要修饰的函数。


这样写是不是有些二？的确是。不过，这是我个人在 Go 语言里所能写出来的最好的代码了。如果你知道更优雅的写法，请你一定告诉我！

好的，让我们来看一下使用效果。首先，假设我们有两个需要修饰的函数：

 复制代码

```
1 func foo(a, b, c int) int {
2     fmt.Printf("%d, %d, %d \n", a, b, c)
3     return a + b + c
4 }
5
6 func bar(a, b string) string {
7     fmt.Printf("%s, %s \n", a, b)
8     return a + b
9 }
```

然后，我们可以这样做：

 复制代码

```
1 type MyFoo func(int, int, int) int
2 var myfoo MyFoo
3 Decorator(&myfoo, foo)
4 myfoo(1, 2, 3)
```

你会发现，使用 `Decorator()` 时，还需要先声明一个函数签名，感觉好傻啊。一点都不泛型，不是吗？谁叫这是有类型的静态编译的语言呢？

嗯。如果你不想声明函数签名，那么也可以这样：

```
2 decorator(mybar, bar)
3 mybar("hello, ", "world!")
```

好吧，看上去不是那么的漂亮，但是 it does work。看样子 Go 语言目前本身的特性无法做成像 Java 或 Python 那样，对此，我们只能多求 Go 语言多放糖了！

## 小结

好了，讲了那么多的例子，看了那么多的代码，我估计你可能有点晕，让我们来做个小结吧。

通过上面 Python 和 Go 修饰器的例子，我们可以看到，所谓的修饰器模式其实是在做下面的几件事。

表面上看，修饰器模式就是扩展现有的一个函数的功能，让它可以干一些其他的事，或是在现有的函数功能上再附加上一些别的功能。

除了我们可以感受到**函数式编程**下的代码扩展能力，我们还能感受到函数的互相和随意拼装带来的好处。

但是深入一下，我们不难发现，Decorator 这个函数其实是可以修饰几乎所有的函数的。于是，这种可以通用于其它函数的编程方式，可以很容易地将一些非业务功能的、属于控制类型的代码给抽象出来（所谓的控制类型的代码就是像 for-loop，或是打日志，或是函数路由，或是求函数运行时间之类的非业务功能性的代码）。

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。**这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。**

[编程范式游记（1）- 起源](#)

[编程范式游记（2）- 泛型编程](#)

[编程范式游记（3）- 类型系统和泛型的本质](#)

[编程范式游记（4）- 函数式编程](#)


[编程范式游记（5）- 修饰器模式](#)

[编程范式游记 \( 8 \) - Go 语言的委托模式](#)

[编程范式游记 \( 9 \) - 编程的本质](#)

[编程范式游记 \( 10 \) - 逻辑编程范式](#)

[编程范式游记 \( 11 \) - 程序世界里的编程范式](#)

 极客时间

# 左耳朵耗子

全年独家专栏《左耳听风》

20000 名程序员的练级攻略

陈皓

资深技术专家  
骨灰级程序员



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 33 | [编程范式游记 \( 4 \) - 函数式编程](#)

下一篇 35 | [编程范式游记 \( 6 \) - 面向对象编程](#)

## 精选留言 (9)

 写留言



楊\_宵夜

2018-02-02

 8

越看越觉得装饰器模式是属于AOP思想的一种实现😄。

**mingnub**

2018-02-14

3

其实Java装饰器和Python装饰器还是差别挺大的，Python装饰器是一个高阶函数，Java的则真的是"注解"，只是起到一个打标签的作用，还要另外的类来检查特定标签进行特定处理。

**superK**

2018-01-25

3

耗子叔，我看你博客和文章很久了，从coolshell就开始了，现在也快30了，但是越来越焦虑，他们都说是30岁程序员的普遍情况，希望耗子叔能以过来人的身份写下这方面的文章，为我们指点下迷路

**浩子**

2018-01-25

2

耗子哥，文章写的很有意思。最近也在相继学习Go语言。不过我很纠结，我是一名.net的技术主管，最近想开拓其他语言的方向。可是却不知道从何下手，比较感兴趣的有Go，Java，Python。可是时间总是有限的。不知道从哪方面进行深入研究。

展开



2018-12-15

1

基本没看懂，后面的总结基本知道装饰器是干嘛的

展开

**杨智晓**

2018-11-16

1

哎，Go语言的语法真是看着别扭，虽然知道Go强劲

展开

**恒**

2018-09-19

1



下载APP



亮出

2018-07-26



编程的例子，有github么

展开 ▾



秋天

2018-04-26



python和go基本语法要看看上面有的函数例子，没看懂。