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。

而Python使用了一种相对于Decorator Pattern和Annotation来说非常优雅的方法，这种方法不需要你去掌握什么复杂的OO模型或是Annotation的各种类库规定，完全就是语言层面的玩法：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。

```
def hello(fn):
    def wrapper():
        print "hello, %s" % fn.__name__
        fn()
        print "goodbye, %s" % fn.__name__
    return wrapper

@hello
def Hao():
    print "i am Hao Chen"

Hao()
```

代码的执行结果如下：

```
$ python hello.py
hello, Hao
i am Hao Chen
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）；
3. hello函数中返回了一个inner函数 `wrapper`，这个 `wrapper`函数回调了传进来的 `fn`，并在回调前后加了两条语句。

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：

```
@decorator
def func():
    pass
```

其解释器会解释成下面这样的语句：

```
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：

```
def makeHtmlTag(tag, *args, **kwds):
    def real_decorator(fn):
        css_class = " class='{0}'".format(kwds["css_class"]) \
                                     if "css_class" in kwds else ""
        def wrapped(*args, **kwds):
            return "<"+tag+css_class+">" + fn(*args, **kwds) + "</"+tag+">"
        return wrapped
    return real_decorator

@makeHtmlTag(tag="b", css_class="bold_css")
@makeHtmlTag(tag="i", css_class="italic_css")
def hello():
    return "hello world"

print hello()

# 输出:
# <b class='bold_css'><i class='italic_css'>hello world</i></b>
```

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：

```
from functools import wraps
def memoization(fn):
    cache = {}
    miss = object()

    @wraps(fn)
    def wrapper(*args):
        result = cache.get(args, miss)
        if result is miss:
            result = fn(*args)
            cache[args] = result
        return result

    return wrapper

@memoization
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

上面这个例子中，是一个斐波那契数例的递归算法。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 `fib(4)` 分解成 `fib(3) + fib(2)`，`fib(3)` 又分解成`fib(2) + fib(1)`……你可看到，基本上来说，`fib(3)`，`fib(2)`，`fib(1)`在整个递归过程中被调用了至少两次。

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。

```
class myDecorator(object):
    def __init__(self, fn):
        print "inside myDecorator.__init__()"
        self.fn = fn

    def __call__(self):
        self.fn()
        print "inside myDecorator.__call__()"

@myDecorator
def aFunction():
    print "inside aFunction()"

print "Finished decorating aFunction()"

aFunction()

# 输出:
# inside myDecorator.__init__()
# Finished decorating aFunction()
# inside aFunction()
# 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。

从上面的输出中，可以看到整个程序的执行顺序。这看上去要比“函数式”的方式更易读一些。

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：

```
class MyApp():
    def __init__(self):
        self.func_map = {}

    def register(self, name):
        def func_wrapper(func):
            self.func_map[name] = func
            return func
        return func_wrapper

    def call_method(self, name=None):
        func = self.func_map.get(name, None)
        if func is None:
            raise Exception("No function registered against - " + str(name))
        return func()

app = MyApp()

@app.register('/')
def main_page_func():
    return "This is the main page."

@app.register('/next_page')
def next_page_func():
    return "This is the next page."

print app.call_method('/')
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。

还是从一个Hello World开始。

```
package main

import "fmt"

func decorator(f func(s string)) func(s string) {
    return func(s string) {
        fmt.Println("Started")
        f(s)
        fmt.Println("Done")
    }
}

func Hello(s string) {
    fmt.Println(s)
}

func main() {
    decorator(Hello)("Hello, World!")
}
```

可以看到，我们动用了一个高阶函数 `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：

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

我们再来看一个为函数log消耗时间的例子：

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

func getFunctionName(i interface{}) string {
    return runtime.FuncForPC(reflect.ValueOf(i).Pointer()).Name()
}

func timedSumFunc(f SumFunc) SumFunc {
    return func(start, end int64) int64 {
        defer func(t time.Time) {
            fmt.Printf("--- Time Elapsed (%s): %v ---\n",
                getFunctionName(f), time.Since(t))
        }(time.Now())
        return f(start, end)
    }
}

func Sum1(start, end int64) int64 {
    var sum int64
    sum = 0
    if start > end {
        start, end = end, start
    }
    for i := start; i <= end; i++ {
        sum += i
    }
    return sum
}

func Sum2(start, end int64) int64 {
    if start > end {
        start, end = end, start
    }
    return (end - start + 1) * (end + start) / 2
}

func main() {

    sum1 := timedSumFunc(Sum1)
    sum2 := timedSumFunc(Sum2)

    fmt.Printf("%d, %d\n", sum1(-10000, 10000000), sum2(-10000, 10000000))
}
```

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。
- 修饰器函数是 `timedSumFunc()`。

再来看一个 HTTP 路由的例子：

```
func WithServerHeader(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithServerHeader()")
        w.Header().Set("Server", "HelloServer v0.0.1")
        h(w, r)
    }
}

func WithAuthCookie(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithAuthCookie()")
        cookie := &http.Cookie{Name: "Auth", Value: "Pass", Path: "/"}
        http.SetCookie(w, cookie)
        h(w, r)
    }
}

func WithBasicAuth(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithBasicAuth()")
        cookie, err := r.Cookie("Auth")
        if err != nil || cookie.Value != "Pass" {
            w.WriteHeader(http.StatusForbidden)
            return
        }
        h(w, r)
    }
}

func WithDebugLog(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Println("--->WithDebugLog")
        r.ParseForm()
        log.Println(r.Form)
        log.Println("path", r.URL.Path)
        log.Println("scheme", r.URL.Scheme)
        log.Println(r.Form["url_long"])
        for k, v := range r.Form {
            log.Println("key:", k)
            log.Println("val:", strings.Join(v, ""))
        }
        h(w, r)
    }
}
func hello(w http.ResponseWriter, r *http.Request) {
    log.Printf("Received Request %s from %s\n", r.URL.Path, r.RemoteAddr)
    fmt.Fprintf(w, "Hello, World! "+r.URL.Path)
}
```

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。

```
func main() {
    http.HandleFunc("/v1/hello", WithServerHeader(WithAuthCookie(hello)))
    http.HandleFunc("/v2/hello", WithServerHeader(WithBasicAuth(hello)))
    http.HandleFunc("/v3/hello", WithServerHeader(WithBasicAuth(WithDebugLog(hello))))
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

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：

```
type HttpHandlerDecorator func(http.HandlerFunc) http.HandlerFunc

func Handler(h http.HandlerFunc, decors ...HttpHandlerDecorator) http.HandlerFunc {
    for i := range decors {
        d := decors[len(decors)-1-i] // iterate in reverse
        h = d(h)
    }
    return h
}
```

然后，我们就可以像下面这样使用了。

```
http.HandleFunc("/v4/hello", Handler(hello,
                WithServerHeader, WithBasicAuth, WithDebugLog))
```

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，其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，所以它们可以干好些比较变态的事，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）。

```
func Decorator(decoPtr, fn interface{}) (err error) {
    var decoratedFunc, targetFunc reflect.Value

    decoratedFunc = reflect.ValueOf(decoPtr).Elem()
    targetFunc = reflect.ValueOf(fn)

    v := reflect.MakeFunc(targetFunc.Type(),
        func(in []reflect.Value) (out []reflect.Value) {
            fmt.Println("before")
            out = targetFunc.Call(in)
            fmt.Println("after")
            return
        })

    decoratedFunc.Set(v)
    return
}
```

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——《The Laws of Reflection》，在这里我不多说了。

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？的确是的。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：

```
func foo(a, b, c int) int {
    fmt.Printf("%d, %d, %d \n", a, b, c)
    return a + b + c
}

func bar(a, b string) string {
    fmt.Printf("%s, %s \n", a, b)
    return a + b
}
```

然后，我们可以这样做：

```
type MyFoo func(int, int, int) int
var myfoo MyFoo
Decorator(&myfoo, foo)
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：

```
mybar := bar
Decorator(&mybar, bar)
mybar("hello,", "world!")
```

好吧，看上去不是那么的漂亮，但是it works。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（6）- 面向对象编程
- 编程范式游记（7）- 基于原型的编程范式
- 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