GO 语言中文网
http://studygolang.com

过滤器和WebSocket在Go语言中文网的应用

芝麻到家 徐新华

polaris

2015.9.5

芝麻到家

提纲

- 一、过滤器（拦截器）/装饰器（中间件）介绍
- 二、过滤器在Go语言中文网的应用
- 三、WebSocket介绍
- 四、WebSocket在Go语言中文网的应用

拦截器

- 拦截器，在AOP（Aspect-Oriented Programming）中用于在某个方法或字段被访问之前，进行拦截然后在之前或之后加入某些操作。拦截是AOP的一种实现策略。
- 拦截器是动态拦截Action调用的对象。它提供了一种机制使开发者可以定义在一个action执行的前后执行特定的代码，也可以在一个action执行前阻止其执行。同时也是提供了一种可以提取action中可重用的部分的方式。

过滤器

- 过滤器是一个程序，它先于与之相关的servlet或JSP页面运行在服务器上。过滤器可附加到一个或多个servlet或JSP页面上，并且可以检查进入这些资源的请求信息。
- 过滤器是一段代码，可被配置在控制器动作执行之前或之后执行。例如， 访问控制过滤器将被执行以确保在执行请求的动作之前用户已通过身份验证；性能过滤器可用于测量控制器执行所用的时间。
- YII中的说明： 一个动作可以有多个过滤器。过滤器执行顺序为它们出现在过滤器列表中的顺序。过滤器可以阻止Action及后面其他过滤器的执行

装饰器(Decorator)

- 通常给对象添加功能，要么直接修改对象添加相应的功能，要么派生对应的子类来扩展，抑或是使用对象组合的方式。显然，直接修改对应的类这种方式并不可取。
- 装饰模式能够实现动态的为对象添加功能，是从一个对象外部来给对象添加功能。
- 在面向对象的设计中，而我们也应该**尽量使用对象组合，而不是对象继承**来扩展和复用功能。装饰器模式就是基于对象组合的方式，可以很灵活的给对象添加所需要的功能。
- 装饰器模式的本质就是**动态组合**。动态是手段，组合才是目的。总之，装饰模式是通过把复杂的功能简单化，分散化，然后再运行期间，根据需要来动态组合的这样一个模式。

中间件

- 中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。
- 但现在很多框架（比如现在Go语言的一些Web框架），会说自己是微型的，支持各种中间件(Middleware)，可插拔。比如，权限验证中间件

小结

- 名字叫法、实现可能不太相同
- 目的类似：不改变原有对象（或功能、业务逻辑）的情况下，动态扩展功能；也是一种代码复用

- 本PPT使用过滤器(Filter)的叫法

过滤器在Go语言中文网的应用

- ListenAndServe

  func ListenAndServe(addr string, handler Handler) error

- ServeMux　　DefaultServeMux
- net/http Handler 接口

  type Handler interface {

  　　ServeHTTP(ResponseWriter, *Request)

  }

- Handler 的一个实现HandlerFunc

  type HandlerFunc func(ResponseWriter, *Request)

过滤器在Go语言中文网的应用

- net/http 的 decorator
  - func StripPrefix(prefix string, h Handler) Handler

```
func StripPrefix(prefix string, h Handler) Handler {
        if prefix == "" {
                return h
        }
        return HandlerFunc(func(w ResponseWriter, r *Request) {
                if p := strings.TrimPrefix(r.URL.Path, prefix); len(p) < len(r.URL.Path) {
                        r.URL.Path = p
                        h.ServeHTTP(w, r)
                } else {
                        NotFound(w, r)
                }
        })
}
```

过滤器在Go语言中文网的应用

- net/h
  - func

```
func T

}
```

```
func (h *timeoutHandler) ServeHTTP(w ResponseWriter, r *Request) {
        done := make(chan bool, 1)
        tw := &timeoutWriter{w: w}
        go func() {
                h.handler.ServeHTTP(tw, r)
                done <- true
        }()
        select {
        case <-done:
                return
        case <-h.timeout():
                tw.mu.Lock()
                defer tw.mu.Unlock()
                if !tw.wroteHeader {
                        tw.w.WriteHeader(StatusServiceUnavailable)
                        tw.w.Write([]byte(h.errorBody()))
                }
                tw.timedOut = true
        }
}
```

andler

过滤器在Go语言中文网的应用

- Decorator/中间件 核心思想

```
// 类似的，参数和返回值类型可以为 http.Handler
func logHandlerFunc(fn http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Before")
    fn(w, r)
    fmt.Fprintln(w, "After")
  }
}
```

```
http.HandlerFunc("/hello", logHandlerFunc(func(w http.ResponseWriter, r *http.Request){
    fmt.Fprintln(w, "Hello World!")
}))

// Output:
// Before
// Hello World!
// After
```

过滤器在Go语言中文网的应用

- 过滤器

```go
// 自定义Handler，以便应用过滤器
type MyHandler struct {
    *http.ServeMux
    // FilterChain 过滤器链
    FilterChain *FilterChain

    path string
}

func NewMyHandler(path string) *MyHandler {...
}

func (this *MyHandler) ServeHTTP(rw http.ResponseWriter, req *http.Request) {
    // 执行当前Route的FilterChain
    filterChain := this.FilterChain
    if filterChain != nil {
        filterChain.Run(0, this, rw, req)
        return
    }
    // 没有设置FilterChain时，直接执行Handler
    this.ServeMux.ServeHTTP(rw, req)
}
```

过滤器在Go语言中文网的应用

- 过滤器核心代码

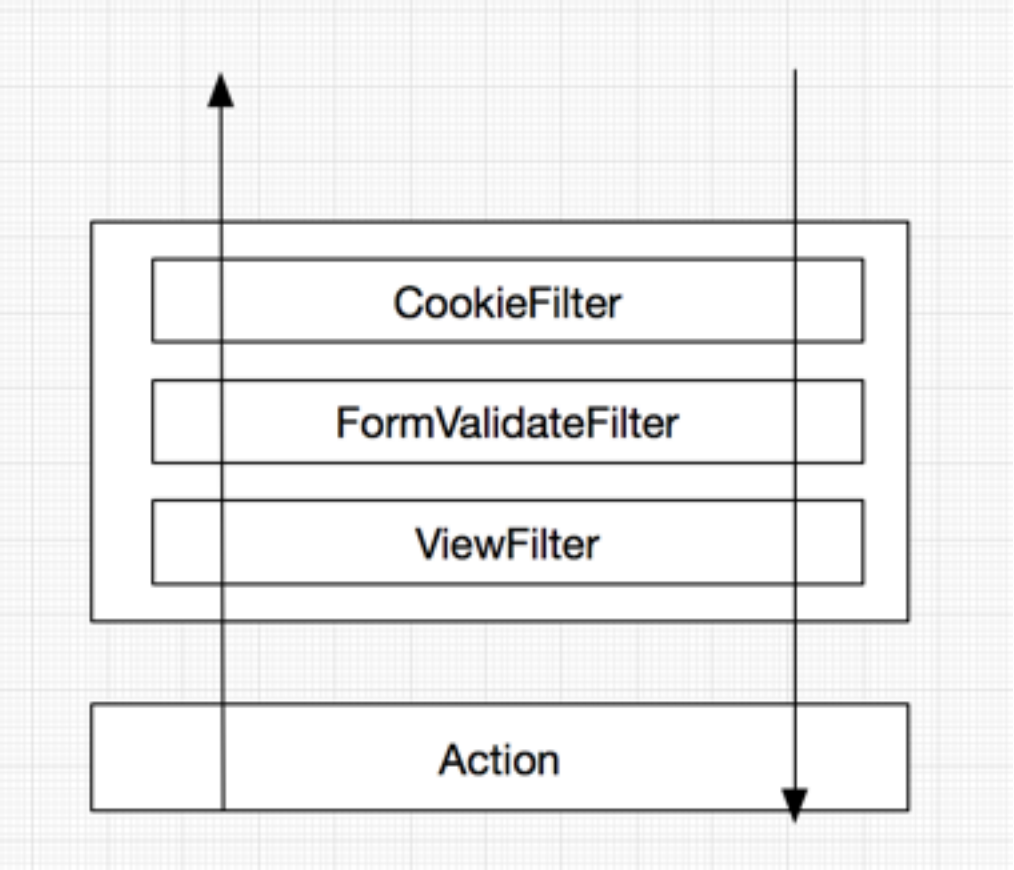
```
// 过滤器接口
type Filter interface {
    // 在Handler执行之前 执行
    PreFilter(http.ResponseWriter, *http.Request) bool
    // 在PreFilter返回false时，执行PreErrorHandle处理错误的情况
    PreErrorHandle(http.ResponseWriter, *http.Request)
    // 在Handler执行之后 执行
    PostFilter(http.ResponseWriter, *http.Request) bool
}
```

过滤器在Go语言中文网的应用

- 过滤器核心代码

```
// Run 运行过滤器链
func (this *FilterChain) Run(cur int, handler *MyHandler, rw http.ResponseWriter, req *http.Request) {
    if this.cur < len(this.filters) {
        i := this.cur
        this.cur++
        if this.filters[i].PreFilter(rw, req) {
            this.Run(cur, handler, rw, req)
            this.filters[i].PostFilter(rw, req)
        } else {
            // 错误处理中，过滤器链不应该往下执行了。
            this.filters[i].PreErrorHandle(rw, req)
        }
    } else {
        // 执行真正的逻辑
        handler.ServeMux.ServeHTTP(rw, req)
    }
}
```

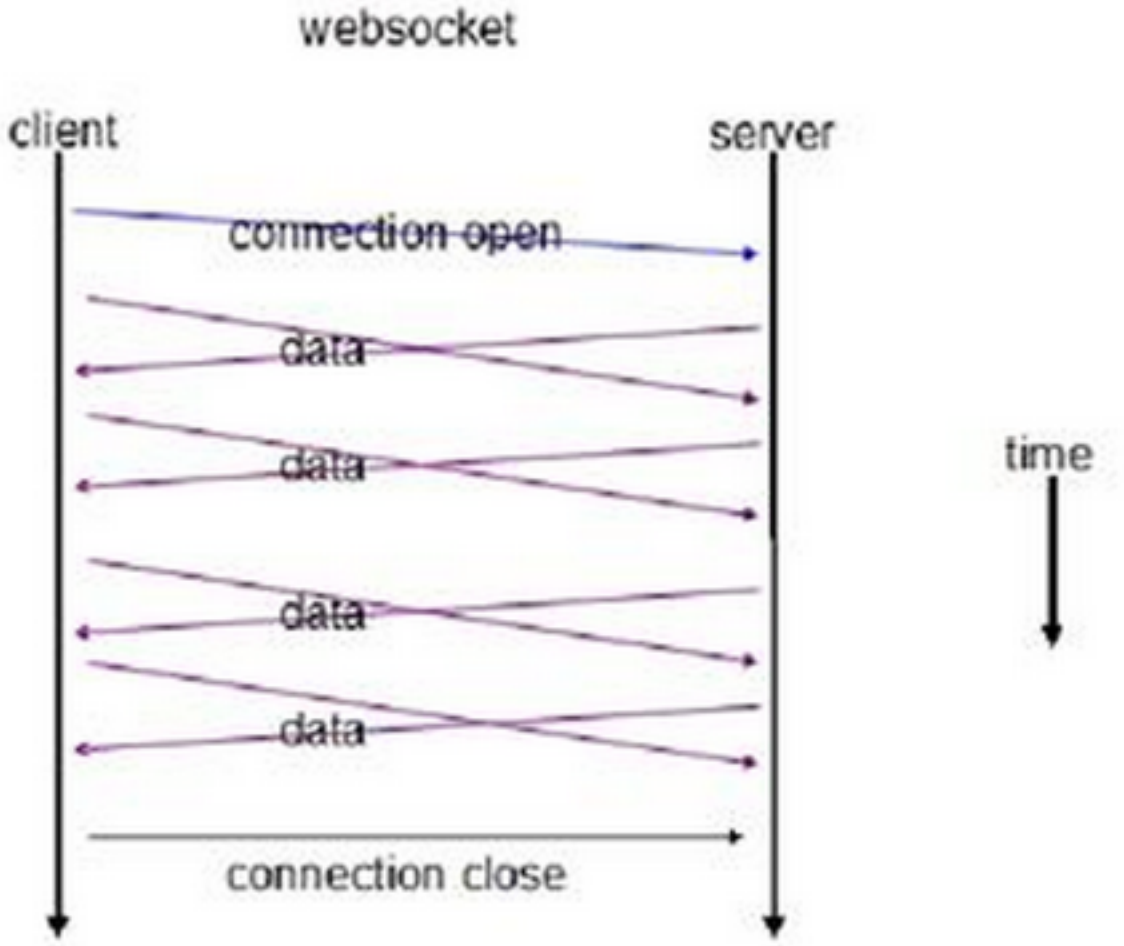
过滤器在Go语言中文网的应用



WebSock 介绍

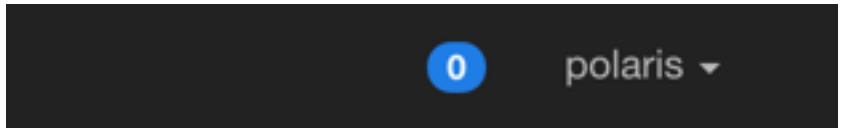
- WebSocket p 浏览器与服务器全双工
- http://www. ocket/

图 2.WebSocket 请求响应客户端服务器交互图



WebSocket在Go语言中文网的应用

- 具体应用场景
  - ➢用户进来 — 给所有在线用户广播，更新在线人数
  - ➢用户离开— 给所有在线用户广播，更新在线人数
  - ➢被@或站内信



WebSocket在Go语言中文网的应用


```
userData := service.Book.AddUser(user, serverId)
// 给自己发送消息，告诉当前在线用户数、历史最高在线人数
onlineInfo := map[string]int{"online": service.Book.Len(), "maxonline": service.MaxOnlineNum()}
message := service.NewMessage(service.WsMsgOnline, onlineInfo)
err = websocket.JSON.Send(wsConn, message)
if err != nil {
    logger.Errorln("Sending onlineusers error:", err)
}
var clientClosed = false
for {
    select {
    case message := <-userData.MessageQueue(serverId):
        if err := websocket.JSON.Send(wsConn, message); err != nil {
            clientClosed = true
        }
        // 心跳
    case <-time.After(30e9):
        if err := websocket.JSON.Send(wsConn, ""); err != nil {
            clientClosed = true
        }
    }
    if clientClosed {
        service.Book.DelUser(user, serverId)
        break
    }
}
// 用户退出时需要变更其他用户看到的在线用户数
if !service.Book.UserIsOnline(user) {
    message := service.NewMessage(service.WsMsgOnline, map[string]int{"online": service.Book.Len()})
    go service.Book.BroadcastAllUsersMessage(message)
}
```

WebSocket在Go语言中文网的应用

- 使用 golang.org/x/net/websocket 包
- 关键数据结构

```
type Message struct {

func (this *UserData) Len() int {...
}

func (this *UserData) MessageQueue(serverId int) chan *Message {
    return this.serverMsgQueue[serverId]
}

type UserData struct {
    // 该用户收到的消息(key为serverId)
    serverMsgQueue map[int]chan *Message
    lastAccessTime time.Time
    onlineDuartion time.Duration

    rwMutex sync.RWMutex
}
```

WebS

- 关键数

```
// 增加一个用户到book中（有可能是用户的另一个请求）
// user为UID或IP地址的int表示
func (this *book) AddUser(user, serverId int) *UserData {…
}

// 删除用户
func (this *book) DelUser(user, serverId int) {…
}

// 判断用户是否还在线
func (this *book) UserIsOnline(user int) bool {…
}

// 在线用户数
func (this *book) Len() int {…
}

// 给某个用户发送一条消息
func (this *book) PostMessage(uid int, message *Message) {…
}

// 给所有用户广播消息
func (this *book) BroadcastAllUsersMessage(message *Message) {…
}

// 给除了自己的其他用户广播消息
func (this *book) BroadcastToOthersMessage(message *Message, myself int) {…
}
```

小结

- 站内通知
- 聊天
- ……

Q&A