Golang构建HTTP服务（二）--- Handler，ServeMux与中间件

人世间 + 关注
2016.12.22 19:22* 字数 1929 阅读 4709 评论 0 喜欢 5 赞赏 1

Golang标准库http包提供了基础的http服务，这个服务又基于Handler接口和ServeMux结构的做Mutilpexer。实际上，go的作者设计Handler这样的接口，不仅提供了默认的ServeMux对象，开发者也可以自定义ServeMux对象。

本质上ServeMux只是一个路由管理器，而它本身也实现了Handler接口的ServeHTTP方法。因此围绕Handler接口的方法ServeHTTP，可以轻松的写出go中的中间件。

在go的http路由原理讨论中，追本溯源还是讨论Handler接口和ServeMux结构。下面就基于这两个对象开始更多关于go中http的故事吧。

介绍http库源码的时候，创建http服务的代码很简单，实际上代码隐藏了很多细节，才有了后来的流程介绍。本文的目的主要是把这些细节暴露，从更底层的方式开始，一步步隐藏细节，完成样例代码的一样的逻辑。了解更多http包的原理之后，才能基于此构建中间件。

自定义的Handler

标准库http提供了Handler接口，用于开发者实现自己的handler。只要实现接口的ServeHTTP方法即可。

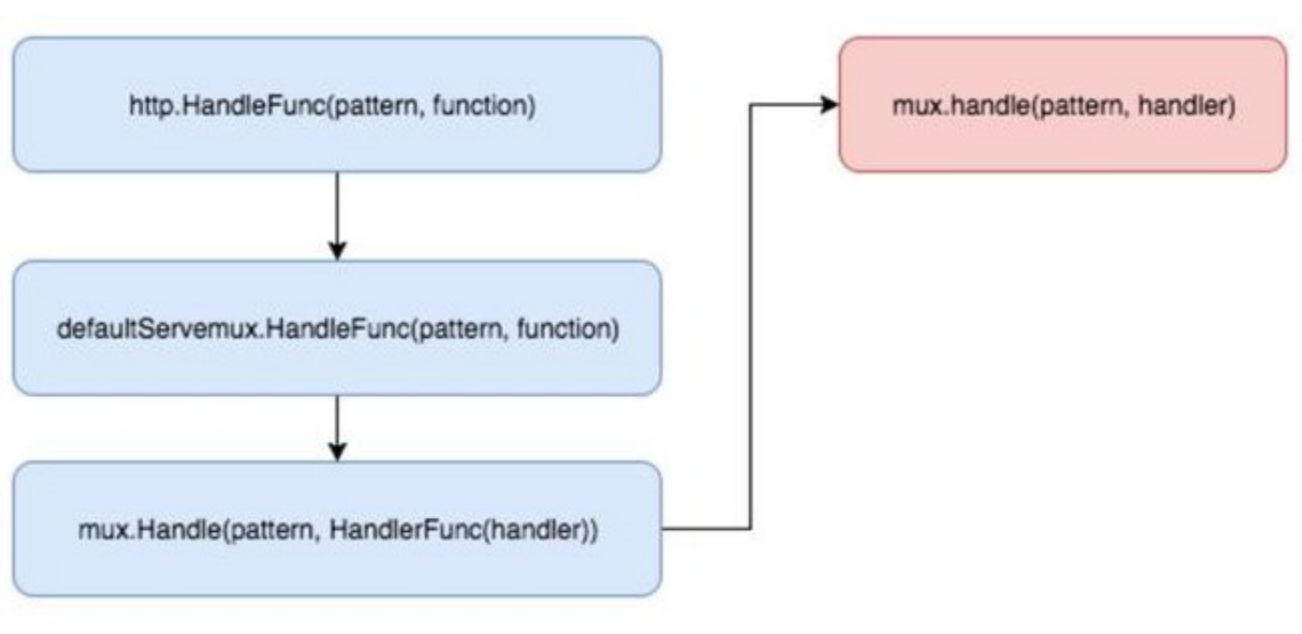
> 关于约定名词 **handler函数**，**handler处理器**，**handler**，请参考http原理与源码笔记中的定义。不然对下文的描述将会很困惑。

```
type textHandler struct {
    responseText string
}

func (th *textHandler) ServeHTTP(w http.ResponseWriter, r *http.Request){
    fmt.Fprintf(w, th.responseText)
}

type indexHandler struct {}

func (ih *indexHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")

    html := `<doctype html>
        <html>
        <head>
          <title>Hello World</title>
        </head>
        <body>
        <p>
          <a href="/welcome">Welcome</a> |  <a href="/message">Message</a>
        </p>
        </body>
</html>`
    fmt.Fprintln(w, html)
}

func main() {
    mux := http.NewServeMux()

    mux.Handle("/", &indexHandler{})

    thWelcome := &textHandler{"TextHandler !"}
    mux.Handle("/text",thWelcome)

    http.ListenAndServe(":8000", mux)
}
```

上面自定义了两个handler结构，都实现了ServeHTTP方法。我们知道，NewServeMux可以创建一个ServeMux实例，ServeMux同时也实现了ServeHTTP方法，因此代码中的mux也是一种handler。把它当成参数传给http.ListenAndServe方法，后者会把mux传给Server实例。因为指定了handler，因此整个http服务就不再是DefaultServeMux，而是mux，无论是在注册路由还是提供请求服务的时候。

有一点值得注意，这里并没有使用HandleFunc注册路由，而是直接使用了mux注册路由。当没有指定mux的时候，系统需要创建一个默认的defaultServeMux，此时我们已经有了mux，因此不再需要http.HandleFucn方法了，直接使用mux的Handle方法注册即可。

此外，Handle第二个参数是一个handler(处理器)，并不是HandleFunc的一个handler函数，其原因也是因为mux.Handle本质上就需要绑定url的pattern模式和handler(处理器)即可。既然indexHandler是handle(处理器)，当然就能作为参数，一切请求的处理过程，都交给器实现的接口方法ServeHTTP就行了。这个过程有点饶，如果不甚了解，建议先阅读http原理与源码笔记了解注册路由的本质。下图



handleFunc-handle.jpeg

左边的12两步只是为了创建一个ServeMux实例，然后调用实例的Handle方法，右边的直接就调用了mux实例的Handle方法。

创建handler处理器

上面费劲口舌罗嗦，不就是1，2，3与3的差别么，并且1，2的两步操作，封装程度更高，开发者只需要写函数即可，不用再定义结构。代码更简洁，因此，下面将直接创建handler函数，调用go的方法将函数转变成handler(处理器)。

```
func text(w http.ResponseWriter, r *http.Request){
    fmt.Fprintln(w, "hello world")
}

func index(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html")

    html := `<doctype html>
        <html>
        <head>
          <title>Hello World</title>
        </head>
        <body>
        <p>
          <a href="/welcome">Welcome</a> |  <a href="/message">Message</a>
        </p>
        </body>
</html>`
    fmt.Fprintln(w, html)
}

func main() {
    mux := http.NewServeMux()
    mux.Handle("/", http.HandlerFunc(index))
    mux.HandleFunc("/text", text)
    http.ListenAndServe(":8000", mux)
}
```

代码中使用了 `http.HandlerFunc` 方法直接将一个handler函数转变成实现了handler(处理器)。等价与图中的3的步骤。

而 `mux.HandleFunc("/text", text)` 就更进一步，与图中的2步骤一致，与 `defaultServemux.HandleFunc(pattern, function)` 的用法一样。

使用默认的DefaultServeMux

经过了上面两个过程的转化，隐藏了更多的细节，代码与defaultServeMux的方式越来越像。下面再去掉自定义的ServeMux，只需要修改main函数的逻辑如下：

```
func main() {
    http.Handle("/", http.HandlerFunc(index))
    http.HandleFunc("/text", text)
    http.ListenAndServe(":8000", nil)
}
```

上述的代码就和前文的例子一样，当代码中不显示的创建serveMux对象，http包就默认创建一个DefaultServeMux对象用来做路由管理器mutilplexer。

自定义Server

默认的DefaultServeMux创建的判断来自server对象，如果server对象不提供handler，才会使用默认的serveMux对象。既然ServeMux可以自定义，那么Server对象一样可以。

使用http.Server 即可创建自定义的server对象：

```
func main(){
    http.HandleFunc("/", index)

    server := &http.Server{
        Addr: ":8000",
        ReadTimeout: 60 * time.Second,
        WriteTimeout: 60 * time.Second,
    }
    server.ListenAndServe()
}
```

自定义的serverMux对象也可以传到server对象中。

```
func main() {

    mux := http.NewServeMux()
    mux.HandleFunc("/", index)

    server := &http.Server{
        Addr: ":8000",
        ReadTimeout: 60 * time.Second,
        WriteTimeout: 60 * time.Second,
        Handler: mux,
    }
    server.ListenAndServe()
}
```

可见go中的路由和处理函数之间关系非常密切，同时又很灵活。通过巧妙的使用Handler接口，可以设计出优雅的中间件程序。

中间件Middleware

所谓中间件，就是连接上下级不同功能的函数或者软件，通常进行一些包裹函数的行为，为被包裹函数提供添加一些功能或行为。前文的HandleFunc就能把签名为 `func(w http.ResponseWriter, r *http.Reqeust)` 的函数包裹成handler。这个函数也算是中间件。

这里我们以HTTP请求的中间件为例子，提供一个log中间件，能够打印出每一个请求的log。

go的http中间件很简单，只要实现一个函数签名为 `func(http.Handler) http.Handler` 的函数即可。http.Handler是一个接口，接口方法我们熟悉的为serveHTTP。返回也是一个handler。因为go中的函数也可以当成变量传递或者或者返回，因此也可以在中间件函数中传递定义好的函数，只要这个函数是一个handler即可，即实现或者被handlerFunc包裹成为handler处理器。

```
func middlewareHandler(next http.Handler) http.Handler{
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        // 执行handler之前的逻辑
        next.ServeHTTP(w, r)
        // 执行完毕handler后的逻辑
    })
}
```

这种方式在Elixir的Plug框架中很流行，思想偏向于函数式范式。熟悉python的朋友一定也想到了装饰器。闲话少说，来看看go是如何实现的吧：

```
func loggingHandler(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        log.Printf("Started %s %s", r.Method, r.URL.Path)
        next.ServeHTTP(w, r)
        log.Printf("Comleted %s in %v", r.URL.Path, time.Since(start))
    })
}

func main() {
    http.Handle("/", loggingHandler(http.HandlerFunc(index)))
    http.ListenAndServe(":8000", nil)
}
```

loggingHandler即是一个中间件函数，将请求的和完成的时间处理。可以看见请求或go的输出：

```
2016/12/04 21:18:13 Started GET /
2016/12/04 21:18:13 Comleted / in 13.365µs
2016/12/04 21:18:20 Started GET /
2016/12/04 21:18:20 Comleted / in 17.541µs
```

既然中间件是一种函数，并且签名都是一样，那么很容易就联想到函数一层包一层的中间件。再添加一个函数，然后修改main函数：

```
func hook(next http.Handler) http.Handler{
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Println("before hook")
        next.ServeHTTP(w, r)
        log.Println("after hook")

    })
}

func main() {
    http.Handle("/", hook(loggingHandler(http.HandlerFunc(index))))
    http.ListenAndServe(":8000", nil)
}
```

在loggingHandler再包了一层hook，可以看到输出为：

```
2016/12/04 21:26:30 before hook
2016/12/04 21:26:30 Started GET /
2016/12/04 21:26:30 Comleted / in 14.016µs
2016/12/04 21:26:30 after hook
```

函数调用形成了一条链，可以是在这条链上做很多事情。当然go的写法上，比起elixir的 `|>` 的符号，优雅性略差。

总结

通过对http包的源码学习，我们了解了Handler接口和ServeMux结构。并且知道如何配合他们实现go的中间件函数。当然，对于几个约定名词，handler函数，handler处理器和handler对象的理解，是掌握它们关系的关键因素，而handler处理器和handler对象的关系，恰恰又是go接口使用的经典例子，让go具有一些动态类型的特性。

了解了http服务如何构建之后，处理请求和返回响应就是下一个故事。而实现处理逻辑恰恰在我们一直在强调的ServeHTTP接口方法中。

接下来将会更详细的讨论请求和响应相关的函数对象。