

李文周的博客

JPG程序员/全栈开发 -- 专注互联网技术，相信代码改变世界。Go语言学习QQ群：645090316 公众号：李文周

[首页](#) [归档](#) [关于](#)

Gin框架介绍及使用

2017年10月8日 | [Golang](#) | 12011 阅读

`Gin` 是一个用Go语言编写的web框架。它是一个类似于 `martini` 但拥有更好性能的API框架, 由于使用了 `httprouter` , 速度提高了近40倍。如果你是性能和高效的追求者, 你会爱上 `Gin` 。

Gin框架介绍

Go世界里最流行的Web框架, [Github](#)上有 `32K+` star。基于[httprouter](#)开发的Web框架。 [中文文档](#)齐全, 简单易用的轻量级框架。

Gin框架安装与使用

安装

下载并安装 `Gin` :

```
1 | go get -u github.com/gin-gonic/gin
```

第一个Gin示例:

```
1 package main
2
3 import (
4     "github.com/gin-gonic/gin"
5 )
6
7 func main() {
8     // 创建一个默认的路由引擎
9     r := gin.Default()
10    // GET: 请求方式; /hello: 请求的路径
11    // 当客户端以GET方法请求/hello路径时, 会执行后面的匿名函数
12    r.GET("/hello", func(c *gin.Context) {
13        // c.JSON: 返回JSON格式的数据
14        c.JSON(200, gin.H{
15            "message": "Hello world!",
16        })
17    })
18    // 启动HTTP服务, 默认在0.0.0.0:8080启动服务
19    r.Run()
20 }
```

将上面的代码保存并编译执行, 然后使用浏览器打开 `127.0.0.1:8080/hello` 就能看到一串JSON字符串。

RESTful API

REST与技术无关, 代表的是一种软件架构风格, REST是Representational State Transfer的简称, 中文翻译为“表征状态转移”或“表现层状态转化”。

推荐阅读[阮一峰 理解RESTful架构](#)

简单来说, REST的含义就是客户端与Web服务器之间进行交互的时候, 使用HTTP协议中的4个请求方法代表不同的动作。

- GET用来获取资源
- POST用来新建资源
- PUT用来更新资源
- DELETE用来删除资源。

只要API程序遵循了REST风格，那就可以称其为RESTful API。目前在前后端分离的架构中，前后端基本都是通过RESTful API来进行交互。

例如，我们现在要编写一个管理书籍的系统，我们可以查询对一本书进行查询、创建、更新和删除等操作，我们在编写程序的时候就要设计客户端浏览器与我们Web服务端交互的方式和路径。按照经验我们通常会设计成如下模式：

请求方法	URL	含义
GET	/book	查询书籍信息
POST	/create_book	创建书籍记录
POST	/update_book	更新书籍信息
POST	/delete_book	删除书籍信息

同样的需求我们按照RESTful API设计如下：

请求方法	URL	含义
GET	/book	查询书籍信息
POST	/book	创建书籍记录
PUT	/book	更新书籍信息
DELETE	/book	删除书籍信息

Gin框架支持开发RESTful API的开发。

```
1 func main() {
2     r := gin.Default()
3     r.GET("/book", func(c *gin.Context) {
4         c.JSON(200, gin.H{
5             "message": "GET",
6         })
7     })
8
9     r.POST("/book", func(c *gin.Context) {
10        c.JSON(200, gin.H{
11            "message": "POST",
12        })
13    })
14
15    r.PUT("/book", func(c *gin.Context) {
16        c.JSON(200, gin.H{
17            "message": "PUT",
18        })
19    })
20
21    r.DELETE("/book", func(c *gin.Context) {
22        c.JSON(200, gin.H{
23            "message": "DELETE",
24        })
25    })
26 }
```

开发RESTful API的时候我们通常使用[Postman](#)来作为客户端的测试工具。

Gin渲染

HTML渲染

我们首先定义一个存放模板文件的 `templates` 文件夹，然后在其内部按照业务分别定义一个 `posts` 文件夹和一个 `users` 文件夹。

`posts/index.html` 文件的内容如下：

```
1  {{define "posts/index.html"}}
2  <!DOCTYPE html>
3  <html lang="en">
4
5  <head>
6      <meta charset="UTF-8">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8      <meta http-equiv="X-UA-Compatible" content="ie=edge">
9      <title>posts/index</title>
10 </head>
11 <body>
12     {{.title}}
13 </body>
14 </html>
15 {{end}}
```

users/index.html 文件的内容如下:

```
1  {{define "users/index.html"}}
2  <!DOCTYPE html>
3  <html lang="en">
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8      <title>users/index</title>
9  </head>
10 <body>
11     {{.title}}
12 </body>
13 </html>
14 {{end}}
```

Gin框架中使用 `LoadHTMLGlob()` 或者 `LoadHTMLFiles()` 方法进行HTML模板渲染。

```
1 func main() {  
2     r := gin.Default()  
3     r.LoadHTMLGlob("templates/**/*.html")  
4     //r.LoadHTMLFiles("templates/posts/index.html", "templates/users/index.html")  
5     r.GET("/posts/index", func(c *gin.Context) {  
6         c.HTML(http.StatusOK, "posts/index.html", gin.H{  
7             "title": "posts/index",  
8         })  
9     })  
10  
11     r.GET("/users/index", func(c *gin.Context) {  
12         c.HTML(http.StatusOK, "users/index.html", gin.H{  
13             "title": "users/index",  
14         })  
15     })  
16  
17     r.Run(":8080")  
18 }
```

自定义模板函数

定义一个不转义相应内容的 `safe` 模板函数如下：

```
1 func main() {
2     router := gin.Default()
3     router.SetFuncMap(template.FuncMap{
4         "safe": func(str string) template.HTML{
5             return template.HTML(str)
6         },
7     })
8     router.LoadHTMLFiles("./index.tmpl")
9
10    router.GET("/index", func(c *gin.Context) {
11        c.HTML(http.StatusOK, "index.tmpl", "<a href='https://liwenzhou.com'>李文周的博客</a>")
12    })
13
14    router.Run(":8080")
15 }
```

在 `index.tmpl` 中使用定义好的 `safe` 模板函数:

```
1 <!DOCTYPE html>
2 <html lang="zh-CN">
3 <head>
4     <title>修改模板引擎的标识符</title>
5 </head>
6 <body>
7 <div>{{ . | safe }}</div>
8 </body>
9 </html>
```

静态文件处理

当我们渲染的HTML文件中引用了静态文件时, 我们只需要按照以下方式在渲染页面前调用 `gin.Static` 方法即可。

```
1 func main() {  
2     r := gin.Default()  
3     r.Static("/static", "./static")  
4     r.LoadHTMLGlob("templates/**/*.html")  
5     // ...  
6     r.Run(":8080")  
7 }
```

使用模板继承

Gin框架默认都是使用单模板，如果需要使用 `block template` 功能，可以通过 `"github.com/gin-contrib/multitemplate"` 库实现，具体示例如下：

首先，假设我们项目目录下的templates文件夹下有如下模板文件，其中 `home.html` 和 `index.html` 继承了 `base.html`：

```
1 templates  
2 ├── includes  
3 │   ├── home.html  
4 │   └── index.html  
5 ├── layouts  
6 │   └── base.html  
7 └── scripts.html
```

然后我们定义一个 `loadTemplates` 函数如下：


```
1 func loadTemplates(templatesDir string) multitemplate.Renderer {
2     r := multitemplate.NewRenderer()
3     layouts, err := filepath.Glob(templatesDir + "/layouts/*.tmpl")
4     if err != nil {
5         panic(err.Error())
6     }
7     includes, err := filepath.Glob(templatesDir + "/includes/*.tmpl")
8     if err != nil {
9         panic(err.Error())
10    }
11    // 为layouts/和includes/目录生成 templates map
12    for _, include := range includes {
13        layoutCopy := make([]string, len(layouts))
14        copy(layoutCopy, layouts)
15        files := append(layoutCopy, include)
16        r.AddFromFiles(filepath.Base(include), files...)
17    }
18    return r
19 }
```

我们在 `main` 函数中

```
1 func indexFunc(c *gin.Context){
2     c.HTML(http.StatusOK, "index.tmpl", nil)
3 }
4
5 func homeFunc(c *gin.Context){
6     c.HTML(http.StatusOK, "home.tmpl", nil)
7 }
8
9 func main(){
10    r := gin.Default()
11    r.HTMLRender = loadTemplates("./templates")
12    r.GET("/index", indexFunc)
13    r.GET("/home", homeFunc)
14    r.Run()
15 }
```

补充文件路径处理

关于模板文件和静态文件的路径，我们需要根据公司/项目的要求进行设置。可以使用下面的函数获取当前执行程序的路径。

```
1 func getCurrentPath() string {
2     if ex, err := os.Executable(); err == nil {
3         return filepath.Dir(ex)
4     }
5     return "./"
6 }
```

JSON渲染

```
1 func main() {
2     r := gin.Default()
3
4     // gin.H 是map[string]interface{}的缩写
5     r.GET("/someJSON", func(c *gin.Context) {
6         // 方式一：自己拼接JSON
7         c.JSON(http.StatusOK, gin.H{"message": "Hello world!"})
8     })
9     r.GET("/moreJSON", func(c *gin.Context) {
10        // 方法二：使用结构体
11        var msg struct {
12            Name    string `json:"user"`
13            Message string
14            Age     int
15        }
16        msg.Name = "小王子"
17        msg.Message = "Hello world!"
18        msg.Age = 18
19        c.JSON(http.StatusOK, msg)
20    })
21    r.Run(":8080")
22 }
```

XML渲染

注意需要使用具名的结构体类型。

```
1 func main() {
2     r := gin.Default()
3     // gin.H 是map[string]interface{}的缩写
4     r.GET("/someXML", func(c *gin.Context) {
5         // 方式一：自己拼接JSON
6         c.XML(http.StatusOK, gin.H{"message": "Hello world!"})
7     })
8     r.GET("/moreXML", func(c *gin.Context) {
9         // 方法二：使用结构体
10        type MessageRecord struct {
11            Name    string
12            Message string
13            Age     int
14        }
15        var msg MessageRecord
16        msg.Name = "小王子"
17        msg.Message = "Hello world!"
18        msg.Age = 18
19        c.XML(http.StatusOK, msg)
20    })
21    r.Run(":8080")
22 }
```

YAML渲染

```
1 r.GET("/someYAML", func(c *gin.Context) {
2     c.YAML(http.StatusOK, gin.H{"message": "ok", "status": http.StatusOK})
3 })
```

protobuf渲染

```
1 r.GET("/someProtoBuf", func(c *gin.Context) {
2     reps := []int64{int64(1), int64(2)}
3     label := "test"
4     // protobuf 的具体定义写在 testdata/protoexample 文件中。
5     data := &protoexample.Test{
6         Label: &label,
7         Repls:  reps,
8     }
9     // 请注意，数据在响应中变为二进制数据
10    // 将输出被 protoexample.Test protobuf 序列化了的数据
11    c.ProtoBuf(http.StatusOK, data)
12 })
```

获取参数

获取querystring参数

querystring 指的是URL中 `?` 后面携带的参数，例如：`/user/search?username=小王子&address=沙河`。获取请求的querystring参数的方法如下：

```
1 func main() {
2     //Default返回一个默认的路由引擎
3     r := gin.Default()
4     r.GET("/user/search", func(c *gin.Context) {
5         username := c.DefaultQuery("username", "小王子")
6         //username := c.Query("username")
7         address := c.Query("address")
8         //输出json结果给调用方
9         c.JSON(http.StatusOK, gin.H{
10             "message": "ok",
11             "username": username,
12             "address": address,
13         })
14     })
15     r.Run()
16 }
```

获取form参数

请求的数据通过form表单来提交，例如向 `/user/search` 发送一个POST请求，获取请求数据的方式如下：

```
1 func main() {
2     //Default返回一个默认的路由引擎
3     r := gin.Default()
4     r.POST("/user/search", func(c *gin.Context) {
5         // DefaultPostForm取不到值时会返回指定的默认值
6         //username := c.DefaultPostForm("username", "小王子")
7         username := c.PostForm("username")
8         address := c.PostForm("address")
9         //输出json结果给调用方
10        c.JSON(http.StatusOK, gin.H{
11            "message": "ok",
12            "username": username,
13            "address": address,
14        })
15    })
16    r.Run(":8080")
17 }
```

获取path参数

请求的参数通过URL路径传递，例如： `/user/search/小王子/沙河` 。获取请求URL路径中的参数的方式如下。

```
1 func main() {
2     //Default返回一个默认的路由引擎
3     r := gin.Default()
4     r.GET("/user/search/:username/:address", func(c *gin.Context) {
5         username := c.Param("username")
6         address := c.Param("address")
7         //输出json结果给调用方
8         c.JSON(http.StatusOK, gin.H{
9             "message": "ok",
10            "username": username,
11            "address": address,
12        })
13    })
14
15    r.Run(":8080")
16 }
```

参数绑定

为了能够更方便的获取请求相关参数，提高开发效率，我们可以基于请求的 `Content-Type` 识别请求数据类型并利用反射机制自动提取请求中 `QueryString`、`form表单`、`JSON`、`XML` 等参数到结构体中。下面的示例代码演示了 `.ShouldBind()` 强大的功能，它能够基于请求自动提取 `JSON`、`form表单` 和 `QueryString` 类型的数据，并把值绑定到指定的结构体对象。

```
1 // Binding from JSON
2 type Login struct {
3     User      string `form:"user" json:"user" binding:"required"`
4     Password  string `form:"password" json:"password" binding:"required"`
5 }
6
7 func main() {
8     router := gin.Default()
9
10    // 绑定JSON的示例 ({ "user": "q1mi", "password": "123456" })
11    router.POST("/loginJSON", func(c *gin.Context) {
12        var login Login
13
14        if err := c.ShouldBind(&login); err == nil {
```

```
15         fmt.Printf("login info:%#v\n", login)
16         c.JSON(http.StatusOK, gin.H{
17             "user":    login.User,
18             "password": login.Password,
19         })
20     } else {
21         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
22     }
23 })
24
25 // 绑定form表单示例 (user=q1mi&password=123456)
26 router.POST("/loginForm", func(c *gin.Context) {
27     var login Login
28     // ShouldBind()会根据请求的Content-Type自行选择绑定器
29     if err := c.ShouldBind(&login); err == nil {
30         c.JSON(http.StatusOK, gin.H{
31             "user":    login.User,
32             "password": login.Password,
33         })
34     } else {
35         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
36     }
37 })
38
39 // 绑定QueryString示例 (/loginQuery?user=q1mi&password=123456)
40 router.GET("/loginForm", func(c *gin.Context) {
41     var login Login
42     // ShouldBind()会根据请求的Content-Type自行选择绑定器
43     if err := c.ShouldBind(&login); err == nil {
44         c.JSON(http.StatusOK, gin.H{
45             "user":    login.User,
46             "password": login.Password,
47         })
48     } else {
49         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
50     }
51 })
52
53 // Listen and serve on 0.0.0.0:8080
54 router.Run(":8080")
55 }
```

ShouldBind 会按照下面的顺序解析请求中的数据完成绑定：

1. 如果是 GET 请求，只使用 Form 绑定引擎（query）。
2. 如果是 POST 请求，首先检查 content-type 是否为 JSON 或 XML，然后再使用 Form（form-data）。

文件上传

单个文件上传

文件上传前端页面代码：

```
1 <!DOCTYPE html>
2 <html lang="zh-CN">
3 <head>
4     <title>上传文件示例</title>
5 </head>
6 <body>
7 <form action="/upload" method="post" enctype="multipart/form-data">
8     <input type="file" name="f1">
9     <input type="submit" value="上传">
10 </form>
11 </body>
12 </html>
```

后端gin框架部分代码：


```
1 func main() {
2     router := gin.Default()
3     // 处理multipart forms提交文件时默认的内存限制是32 MiB
4     // 可以通过下面的方式修改
5     // router.MaxMultipartMemory = 8 << 20 // 8 MiB
6     router.POST("/upload", func(c *gin.Context) {
7         // 单个文件
8         file, err := c.FormFile("f1")
9         if err != nil {
10             c.JSON(http.StatusInternalServerError, gin.H{
11                 "message": err.Error(),
12             })
13             return
14         }
15
16         log.Println(file.Filename)
17         dst := fmt.Sprintf("C:/tmp/%s", file.Filename)
18         // 上传文件到指定的目录
19         c.SaveUploadedFile(file, dst)
20         c.JSON(http.StatusOK, gin.H{
21             "message": fmt.Sprintf("%s' uploaded!", file.Filename),
22         })
23     })
24     router.Run()
25 }
```

多个文件上传

```
1 func main() {
2     router := gin.Default()
3     // 处理multipart forms提交文件时默认的内存限制是32 MiB
4     // 可以通过下面的方式修改
5     // router.MaxMultipartMemory = 8 << 20 // 8 MiB
6     router.POST("/upload", func(c *gin.Context) {
7         // Multipart form
8         form, _ := c.MultipartForm()
9         files := form.File["file"]
10
11         for index, file := range files {
12             log.Println(file.Filename)
13             dst := fmt.Sprintf("C:/tmp/%s_%d", file.Filename, index)
14             // 上传文件到指定的目录
15             c.SaveUploadedFile(file, dst)
16         }
17         c.JSON(http.StatusOK, gin.H{
18             "message": fmt.Sprintf("%d files uploaded!", len(files)),
19         })
20     })
21     router.Run()
22 }
```

重定向

HTTP重定向

HTTP 重定向很容易。内部、外部重定向均支持。

```
1 r.GET("/test", func(c *gin.Context) {
2     c.Redirect(http.StatusMovedPermanently, "http://www.sogo.com/")
3 })
```

路由重定向

路由重定向, 使用 `HandleContext` :

```
1 | r.GET("/test", func(c *gin.Context) {
2 |     // 指定重定向的URL
3 |     c.Request.URL.Path = "/test2"
4 |     r.HandleContext(c)
5 | })
6 | r.GET("/test2", func(c *gin.Context) {
7 |     c.JSON(http.StatusOK, gin.H{"hello": "world"})
8 | })
```

Gin路由

普通路由

```
1 | r.GET("/index", func(c *gin.Context) {...})
2 | r.GET("/login", func(c *gin.Context) {...})
3 | r.POST("/login", func(c *gin.Context) {...})
```

此外, 还有一个可以匹配所有请求方法的 `Any` 方法如下:

```
1 | r.Any("/test", func(c *gin.Context) {...})
```

为没有配置处理函数的路由添加处理程序, 默认情况下它返回404代码, 下面的代码为没有匹配到路由的请求都返回 `views/404.html` 页面。

```
1 | r.NoRoute(func(c *gin.Context) {
2 |     c.HTML(http.StatusNotFound, "views/404.html", nil)
3 | })
```

路由组

我们可以将拥有共同URL前缀的路由划分为一个路由组。习惯性一对 `{}` 包裹同组的路由，这只是为了看着清晰，你用不用 `{}` 包裹功能上没什么区别。

```
1 func main() {
2     r := gin.Default()
3     userGroup := r.Group("/user")
4     {
5         userGroup.GET("/index", func(c *gin.Context) {...})
6         userGroup.GET("/login", func(c *gin.Context) {...})
7         userGroup.POST("/login", func(c *gin.Context) {...})
8     }
9     shopGroup := r.Group("/shop")
10    {
11        shopGroup.GET("/index", func(c *gin.Context) {...})
12        shopGroup.GET("/cart", func(c *gin.Context) {...})
13        shopGroup.POST("/checkout", func(c *gin.Context) {...})
14    }
15    r.Run()
16 }
17 }
```

路由组也是支持嵌套的，例如：

```
1 shopGroup := r.Group("/shop")
2 {
3     shopGroup.GET("/index", func(c *gin.Context) {...})
4     shopGroup.GET("/cart", func(c *gin.Context) {...})
5     shopGroup.POST("/checkout", func(c *gin.Context) {...})
6     // 嵌套路由组
7     xx := shopGroup.Group("xx")
8     xx.GET("/oo", func(c *gin.Context) {...})
9 }
```

通常我们将路由分组用在划分业务逻辑或划分API版本时。

路由原理

Gin框架中的路由使用的是[httprouter](#)这个库。

其基本原理就是构造一个路由地址的前缀树。

Gin中间件

Gin框架允许开发者在处理请求的过程中，加入用户自己的钩子（Hook）函数。这个钩子函数就叫中间件，中间件适合处理一些公共的业务逻辑，比如登录认证、权限校验、数据分页、记录日志、耗时统计等。

定义中间件

Gin中的中间件必须是一个 `gin.HandlerFunc` 类型。例如我们像下面的代码一样定义一个统计请求耗时的中间件。

```
1 // StatCost 是一个统计耗时请求耗时的中间件
2 func StatCost() gin.HandlerFunc {
3     return func(c *gin.Context) {
4         start := time.Now()
5         c.Set("name", "小王子") // 可以通过c.Set在请求上下文中设置值，后续的处理函数能够取到该值
6         // 调用该请求的剩余处理程序
7         c.Next()
8         // 不调用该请求的剩余处理程序
9         // c.Abort()
10        // 计算耗时
11        cost := time.Since(start)
12        log.Println(cost)
13    }
14 }
```

注册中间件

在gin框架中，我们可以为每个路由添加任意数量的中间件。

为全局路由注册

```
1 func main() {  
2     // 新建一个没有任何默认中间件的路由  
3     r := gin.New()  
4     // 注册一个全局中间件  
5     r.Use(StatCost())  
6  
7     r.GET("/test", func(c *gin.Context) {  
8         name := c.MustGet("name").(string) // 从上下文取值  
9         log.Println(name)  
10        c.JSON(http.StatusOK, gin.H{  
11            "message": "Hello world!",  
12        })  
13    })  
14    r.Run()  
15 }
```

为某个路由单独注册

```
1 // 给/test2路由单独注册中间件（可注册多个）  
2 r.GET("/test2", StatCost(), func(c *gin.Context) {  
3     name := c.MustGet("name").(string) // 从上下文取值  
4     log.Println(name)  
5     c.JSON(http.StatusOK, gin.H{  
6         "message": "Hello world!",  
7     })  
8 })
```

为路由组注册中间件

为路由组注册中间件有以下两种写法。

写法1:

```
1 shopGroup := r.Group("/shop", StatCost())
2 {
3     shopGroup.GET("/index", func(c *gin.Context) {...})
4     ...
5 }
```

写法2:

```
1 shopGroup := r.Group("/shop")
2 shopGroup.Use(StatCost())
3 {
4     shopGroup.GET("/index", func(c *gin.Context) {...})
5     ...
6 }
```

中间件注意事项

gin默认中间件

`gin.Default()` 默认使用了 `Logger` 和 `Recovery` 中间件, 其中:

- `Logger`中间件将日志写入 `gin.DefaultWriter`, 即使配置了 `GIN_MODE=release`。
- `Recovery`中间件会recover任何 `panic`。如果有`panic`的话, 会写入500响应码。

如果不想使用上面两个默认的中间件, 可以使用 `gin.New()` 新建一个没有任何默认中间件的路由。

gin中间件中使用goroutine

当在中间件或 `handler` 中启动新的 `goroutine` 时, **不能使用**原始的上下文 (`c *gin.Context`), 必须使用其只读副本 (`c.Copy()`)。

运行多个服务

我们可以在多个端口启动服务, 例如:

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "time"
7
8     "github.com/gin-gonic/gin"
9     "golang.org/x/sync/errgroup"
10 )
11
12 var (
13     g errgroup.Group
14 )
15
16 func router01() http.Handler {
17     e := gin.New()
18     e.Use(gin.Recovery())
19     e.GET("/", func(c *gin.Context) {
20         c.JSON(
21             http.StatusOK,
22             gin.H{
23                 "code": http.StatusOK,
24                 "error": "Welcome server 01",
25             },
26         )
27     })
28
29     return e
30 }
31
32 func router02() http.Handler {
33     e := gin.New()
34     e.Use(gin.Recovery())
35     e.GET("/", func(c *gin.Context) {
36         c.JSON(
37             http.StatusOK,
38             gin.H{
39                 "code": http.StatusOK,
40                 "error": "Welcome server 02",
41             },
42         )
43     })
44
45     return e
46 }
```



```
42     )
43 })
44
45 return e
46 }
47
48 func main() {
49     server01 := &http.Server{
50         Addr:      ":8080",
51         Handler:    router01(),
52         ReadTimeout: 5 * time.Second,
53         WriteTimeout: 10 * time.Second,
54     }
55
56     server02 := &http.Server{
57         Addr:      ":8081",
58         Handler:    router02(),
59         ReadTimeout: 5 * time.Second,
60         WriteTimeout: 10 * time.Second,
61     }
62     // 借助errgroup.Group或者自行开启两个goroutine分别启动两个服务
63     g.Go(func() error {
64         return server01.ListenAndServe()
65     })
66
67     g.Go(func() error {
68         return server02.ListenAndServe()
69     })
70
71     if err := g.Wait(); err != nil {
72         log.Fatal(err)
73     }
74 }
```

[3](#) 条评论

未登录用户 ▾



说点什么

① 支持 Markdown 语法

使用 GitHub 登录

预览

TanHeyi 发表于 4 个月前

这里面怎么渲染返回js的代码



Q1mi 发表于 3 个月前

@TanHeyi

这里面怎么渲染返回js的代码



没明白你的问题是什么

TanHeyi 发表于 3 个月前

@Q1mi

@TanHeyi

这里面怎么渲染返回js的代码



没明白你的问题是什么

Emmmm 意思大概就是网页提交一段js代码，再将这段代码返回到网页上，提交的js代码可以执行。已解决。。。

See Also

- Cookie和Session
- 第三方日志库logrus使用

- Go语言基础之单元测试
- Go语言基础之net/http
- Go语言基础之网络编程



© 2020 李文周的博客 By 李文周. 本站博客未经授权禁止转载. Powered by Hugo. Theme based on maupassant. 京ICP备15037996号-1.