# 03 web开发

[03 web开发 1](#_Toc154047253)

[1 cookie和session 1](#_Toc154047254)

[一.Cookie和session 1](#_Toc154047255)

[1 cookie 2](#_Toc154047256)

[2.Session 6](#_Toc154047257)

[二 gin框架 9](#_Toc154047258)

[1.Gin环境搭建,程序的热加载,路由GET,POST,PUT,DELETE 9](#_Toc154047259)

[2. Gin HTML模板渲染以及模板语法,自定义模板函数,静态文件服务 18](#_Toc154047260)

[2.1.Gin HTML 模板渲染 18](#_Toc154047261)

[2.2.静态文件服务 29](#_Toc154047262)

[3.路由详解,自定义控制器,中间件 34](#_Toc154047263)

[3.1.路由详解 34](#_Toc154047264)

[3.2.Gin 中自定义控制器 43](#_Toc154047265)

[2.3.Gin 中间件 45](#_Toc154047266)

[4. 自定义Model以及Gin 文件上传 53](#_Toc154047267)

# 1 cookie和session

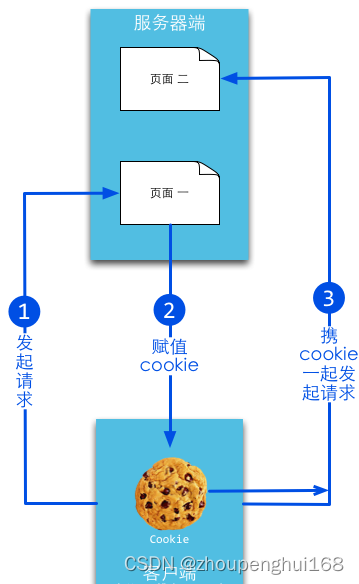
http是无状态协议,服务器不能记录浏览器的访问状态，也就是说服务器不能区分中两次请求是否由一个客户端发出.这样的设计严重阻碍的 Web 程序的设计,如：在进行网购时，买了一条裤子，又买了一个手机,由于 http协议是无状态的，如果不通过其他手段,服务器是不能知道用户到底买了什么,而cookie就是解决方案之一.

一.Cookie和session

## 1 cookie

        Cookie 实际上就是服务器保存在浏览器上的一段信息,浏览器有了Cookie之后，每次向服务器发送请求时都会同时将该信息发送给服务器，服务器收到请求后，就可以根据该信息处理请求

2.cookie的运行原理



(1).第一次向服务器发送请求时在服务器端创建 Cookie

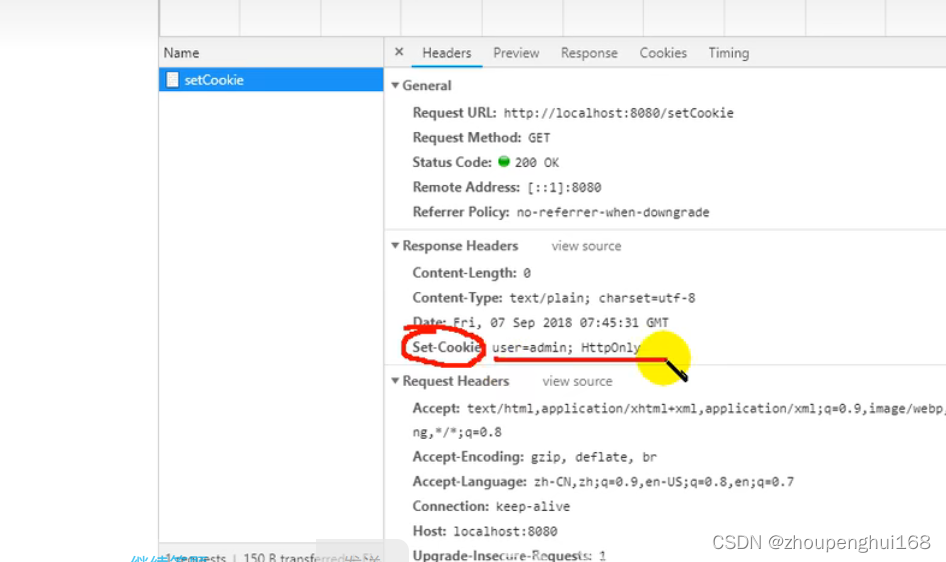
(2).将在服务器端创建的Cookie以响应头的方式发送给浏览器

(3).以后再发送请求时,浏览器就会携带着该Cookie

(4).服务器得到Cookie之后根据Cookie的信息来区分不同的用户

3.创建cookie并将它发送给浏览器

package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
  
// setCookie 添加Cookie  
func setCookie(w http.ResponseWriter, r \*http.Request){  
 // 创建Cookie  
 cookie := http.Cookie{  
 Name: "user",  
 Value: "admin",  
 HttpOnly: true,  
 }  
 // 将Cookie 发送给浏览器  
 w.Header().Set("set-Cookie", cookie.String())  
 // 添加第二个Cookie  
 w.Header().Add("set-Cookie", cookie.String())  
 fmt.Fprintf(w, "setCookie!\n")  
  
}  
  
func main(){  
 http.HandleFunc("/setCookie", setCookie)  
 http.ListenAndServe(":8080", nil)  
  
}

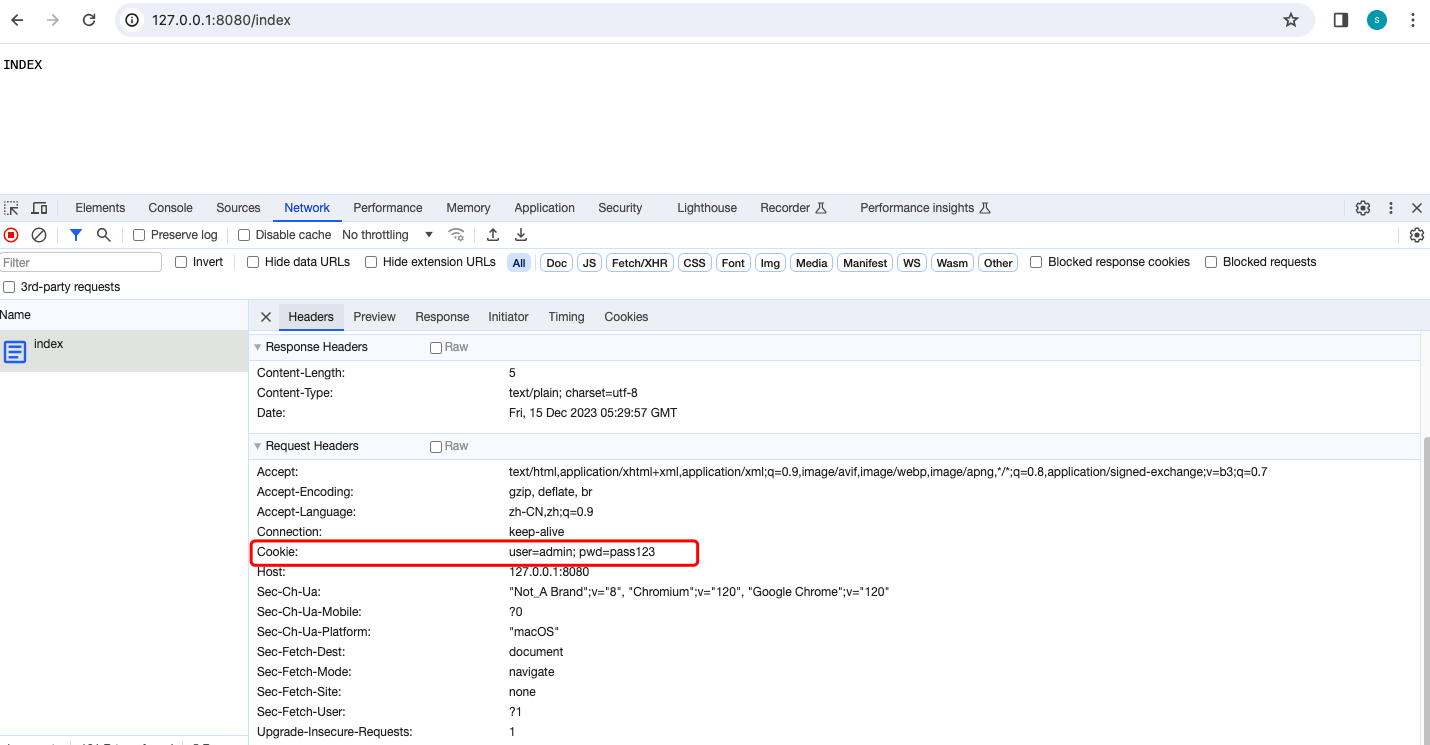


4.获取cookie

package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
  
// setCookie 添加Cookie  
func setCookie(w http.ResponseWriter, r \*http.Request){  
 // 创建Cookie  
 cookie := http.Cookie{  
 Name: "user",  
 Value: "admin",  
 HttpOnly: true,  
 }  
 // 将Cookie 发送给浏览器  
 w.Header().Set("set-Cookie", cookie.String())  
 // 添加第二个Cookie  
 w.Header().Add("set-Cookie", cookie.String())  
 fmt.Fprintf(w, "setCookie!\n")  
  
}  
  
func getCookies(w http.ResponseWriter, r \*http.Request) {  
 // 获取请求头中所有的cookies  
 cookies := r.Header["Cookie"]  
 fmt.Fprintf(w,"所有的cookies：", cookies)  
}  
  
func getCookie(w http.ResponseWriter, r \*http.Request) {  
 // 获取具体某一个cookie  
 cookie, \_ := r.Cookie("user")  
 fmt.Fprintf(w,"user的cookie",cookie)  
}  
  
func main(){  
 http.HandleFunc("/setCookie", setCookie)  
 http.HandleFunc("/getCookies", getCookies)  
 http.HandleFunc("/getCookie", getCookie)  
 http.ListenAndServe(":8080", nil)  
  
}

cookie一旦设置就是全局的：

package main  
  
import (  
 "fmt"  
 "net/http"  
)  
  
  
// setCookie 添加Cookie  
func setCookie(w http.ResponseWriter, r \*http.Request){  
 // 创建Cookie  
 cookie1 := http.Cookie{  
 Name: "user",  
 Value: "admin",  
 HttpOnly: true,  
 }  
 cookie2 := http.Cookie{  
 Name: "pwd",  
 Value: "pass123",  
 HttpOnly: true,  
 }  
 // 将Cookie 发送给浏览器  
 w.Header().Set("set-Cookie", cookie1.String())  
 // 添加第二个Cookie  
 w.Header().Add("set-Cookie", cookie2.String())  
 fmt.Fprintf(w, "setCookie!\n")  
  
}  
  
func getCookies(w http.ResponseWriter, r \*http.Request) {  
 // 获取请求头中所有的cookies  
 cookies := r.Header["Cookie"]  
 fmt.Fprintf(w,"所有的cookies：", cookies)  
}  
  
func getCookie(w http.ResponseWriter, r \*http.Request) {  
 // 获取具体某一个cookie  
 cookie, \_ := r.Cookie("user")  
 fmt.Fprintf(w,"user的cookie",cookie)  
}  
  
func index(w http.ResponseWriter, r \*http.Request) {  
 fmt.Fprintf(w,"INDEX")  
}  
  
func main(){  
 http.HandleFunc("/setCookie", setCookie)  
 http.HandleFunc("/getCookies", getCookies)  
 http.HandleFunc("/getCookie", getCookie)  
 http.HandleFunc("/index", index)  
 http.ListenAndServe(":8080", nil)  
  
}



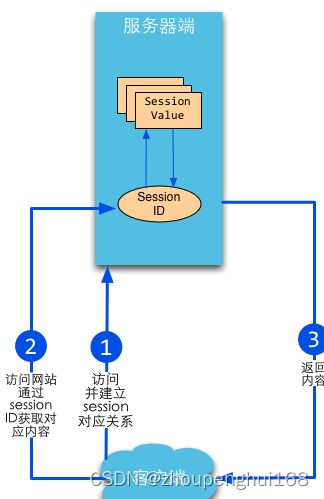
## 2.Session

1.简介

Cookie有一个非常大的局限，就是如果Cookie 很多，则无形的增加了客户端与服务端的数据传输量,而且由于浏览器对Cookie 数量的限制，注定不能在Cookie 中保存过多的信息,于是 Session 出现

Session的作用就是在服务器端保存一些用户的数据，然后传递给用户一个特殊的 Cookie, 这个Cookie对应着这个服务器中的一个Session,通过它就可以获取到保存用户信息的 Session ，进而就知道是哪个用户在发送请求

当然，由于 Session 存储在服务器中，肯定会消耗服务器的资源，所以 Session 一般都会有一个过期时间，服务器一般会定期检查并删除过期的 Session ，如果后来该用户再次访问服务器，可能就会面临重新登录等等措施，然后服务器新建一个 Session ，将 Session ID 通过 Cookie 的形式传送给客户端



(1).第一次向服务器发送请求时创建Session,给它设置一个全局唯一的 ID （可以通过 UUID 生成）

(2).创建一个Cookie,将Cookie 的Value 设置为 Session 的ID值，并将 Cookie 发送给浏览器 ·

(3).以后再发送请求浏览器就会携带着该 Cookie

(4).服务器获取Cookie并根据它的 Value 值找到服务器中对应的 Session,也就知道了请求是哪个用户发的

3.实现

Session一般需要三个组件配合完成，它们分别是 Manager、Provider 和 Session 三个类（接口）

步骤:

(1).浏览器通过 HTTP 协议向服务器请求路径 /content 的网页资源，对应路径上有一个 Handler 函数接收请求，解析 HTTP header 中的 cookie，得到其中存储的 sessionID，然后把这个 ID 发给 Manager

(2).Manager 充当一个 session 管理器的角色，主要存储一些配置信息，比如 session 的存活时间，cookie 的名字等等。而所有的 session 存在 Manager 内部的一个 Provider 中。所以 Manager 会把 sid（sessionID）传递给 Provider，让它去找这个 ID 对应的具体是哪个 session

(3).Provider 就是一个容器，最常见的应该就是一个散列表，将每个 sid 和对应的 session一一映射起来。收到 Manager 传递的 sid 之后，它就找到 sid 对应的 session 结构，也就是 Session 结构，然后返回它

(4).Session 中存储着用户的具体信息，由 Handler 函数中的逻辑拿出这些信息，生成该用户的 HTML 网页，返回给客户端

下面就来说说，为什么分成 Manager、Provider 和 Session

先从最底层的 Session 说。既然 session 就是键值对，为啥不直接用哈希表，而是要抽象出这么一个数据结构呢？

第一，因为 Session 结构可能不止存储了一个哈希表，还可以存储一些辅助数据，比如 sid，访问次数，过期时间或者最后一次的访问时间，这样便于实现 LRU、LFU 这样的算法。

第二，因为 session 可以有不同的存储方式。如果用编程语言内置的哈希表，那么 session 数据就是存储在内存中，如果数据量大，很容易造成程序崩溃，而且一旦程序结束，所有 session 数据都会丢失。所以可以有很多种 session 的存储方式，比如存入缓存数据库 Redis，或者存入 MySQL 等等。

因此，Session 结构提供一层抽象，屏蔽不同存储方式的差异，只要提供一组通用接口操纵键值对：

type Session interface {

// 设置键值对

Set(key, val interface{})

// 获取 key 对应的值

Get(key interface{}) interface{}

// 删除键 key

Delete(key interface{})

}

再说 Provider 为啥要抽象出来。我们上面那个图的 Provider 就是一个散列表，保存 sid 到 Session 的映射，但是实际中肯定会更加复杂。我们要时不时删除一些 session 吗，除了设置存活时间之外，还可以采用一些其他策略，比如 LRU 缓存淘汰算法，这样就需要 Provider 内部使用哈希链表这种数据结构来存储 session,因此，Provider 作为一个容器，就是要屏蔽算法细节，以合理的数据结构和算法组织 sid 和 Session 的映射关系，只需要实现下面这几个方法实现对 session 的增删查改：

type Provider interface {

// 新增并返回一个 session

SessionCreate(sid string) (Session, error)

// 删除一个 session

SessionDestroy(sid string)

// 查找一个 session

SessionRead(sid string) (Session, error)

// 修改一个session

SessionUpdate(sid string)

// 通过类似 LRU 的算法回收过期的 session

SessionGC(maxLifeTime int64)

}

最后说 Manager，大部分具体工作都委托给 Session 和 Provider 承担了，Manager 主要就是一个参数集合，比如 session 的存活时间，清理过期 session 的策略，以及 session 的可用存储方式。Manager 屏蔽了操作的具体细节，我们可以通过 Manager 灵活地配置 session 机制

# 二 gin框架

## 1.Gin环境搭建,程序的热加载,路由GET,POST,PUT,DELETE

Gin 的官网： <https://gin-gonic.com/zh-cn/>  
Gin Github 地址： <https://github.com/gin-gonic/gin>

1. Gin 环境搭建

要安装 Gin 软件包，需要先安装 Go 并设置 Go 工作区

go get -u [github.com/gin-gonic/gin](mailto:github.com/gin-gonic/gin@v1.7.7)

由于网络原因国内部分用户可能没法直接下载第三方包,go get 失败,Golang Gin中没法下载第三方包解决办法如下:

//打开终端并执行

go env -w GO111MODULE=on

go env -w GOPROXY=https://goproxy.cn,direct

mkdir gin\_demo

cd gin\_demo

go mod init gin\_demo

go mod tidy

将 gin 引入到代码中

import "github.com/gin-gonic/gin"

2. 如果使用诸如 http.StatusOK 之类的常量，则需要引入 net/http 包

import "net/http"

3.新建main.go配置路由

package main  
  
import (  
 "github.com/gin-gonic/gin"  
 "net/http"  
)  
  
func main() {  
 // 创建gin的默认路由引擎  
 r := gin.Default()  
 // 配置路由  
 r.GET("/", func(c \*gin.Context) {  
 c.JSON(200, gin.H{ // c.JSON：返回 JSON 格式的数据  
 "message": "Hello world!", })  
 })  
  
 // 浏览器访问请求127.0.0.1:8080/ping路由时,调用回调函数  
 r.GET("/ping", func(c \*gin.Context) {  
 //浏览器输出  
 //c.JSON(200, gin.H{  
 // "message": "pong",  
 //})  
 c.String(http.*StatusOK*, "值:%v", "你好")  
 })  
  
 //监听并在 0.0.0.0:8080 上启动服务(启动一个web服务)  
 r.Run()  
 // 监听并在 0.0.0.0:8000 上启动服务(启动一个web服务)  
 //r.Run("8000")  
}

4.运行项目

go run main.go

5.要改变默认启动的端口

r.Run(":9000")

6.golang程序的热加载

所谓热加载就是当对代码进行修改时，程序能够自动重新加载并执行，这在开发中

是非常便利的，可以快速进行代码测试，省去了每次手动重新编译,beego 中可以使用官方给提供bee工具来热加载项目，但是 gin 中并没有官方提供的热加载工具，这个时候要实现热加载就可以借助第三方的工具

工具 1（推荐）：https://github.com/gravityblast/fresh

//进入终端执行

go get github.com/pilu/fresh

//然后运行命令

fresh

//或者进入终端执行

go install github.com/pilu/fresh@latest

//然后运行命令

fresh

**工具 2**：<https://github.com/codegangsta/gin>

//进入终端执行

go get -u github.com/codegangsta/gin

//然后运行命令

gin run main.go

7.Gin框架中的路由

1).路由概述

路由（Routing）是由一个 URI（或者叫路径）和一个特定的 HTTP 方法（GET、POST 等）

组成的，涉及到应用如何响应客户端对某个网站节点的访问 ,RESTful API 是目前比较成熟的一套互联网应用程序的 API 设计理论，所以在设计路由的时候建议参考 RESTful API 指南,在 RESTful 架构中，每个网址代表一种资源，不同的请求方式表示执行不同的操作：

|  |
| --- |
| GET（SELECT） 从服务器取出资源（一项或多项） |
| POST（CREATE）在服务器新建一个资源 |
| PUT（UPDATE） 在服务器更新资源（客户端提供改变后的完整资源） |
| DELETE（DELETE） 从服务器删除资源 |

简单的路由配置

当用 GET 请求访问一个网址的时候

r.GET("网址", func(c \*gin.Context) {

c.String(200, "Get")

})

当用 POST 访问一个网址的时候

r.POST("网址", func(c \*gin.Context) {

c.String(200, "POST")

})

当用 PUT 访问一个网址的时候

r.PUT("网址", func(c \*gin.Context) {

c.String(200, "PUT")

})

当用 DELETE 访问一个网址的时候

r.DELETE("网址", func(c \*gin.Context) {

c.String(200, "DELETE")

})

路由里面获取 Get 传值

// 域名/news?aid=20

r.GET("/news", func(c \*gin.Context) {

aid := c.Query("aid")

c.String(200, "aid=%s", aid)

})

动态路由

// 域名/user/20

r.GET("/user/:uid", func(c \*gin.Context) {

    uid := c.Param("uid")

    c.String(200, "userID=%s", uid)

})

完整的代码案例如下:

package main  
  
import (  
 \_ "fmt"  
 "github.com/gin-gonic/gin"  
 "net/http"  
)  
  
func main() {  
 r := gin.Default() // 创建gin的默认路由引擎  
 //配置路由  
 // 浏览器访问请求127.0.0.1:8080/ping路由时,调用回调函数  
 r.GET("/ping", func(c \*gin.Context) {  
 //浏览器输出  
 //c.JSON(200, gin.H{  
 // "message": "pong",  
 //})  
 c.String(http.*StatusOK*, "值:%v", "你好")  
 })  
 r.GET("/news", func(c \*gin.Context) {  
 //使用http.StatusOK状态码  
 c.String(http.*StatusOK*, "新闻页11面")  
 })  
 r.POST("/add", func(c \*gin.Context) {  
 c.String(200, "这是一个POST请求,主要用于增加数据")  
 })  
 r.PUT("/edit", func(c \*gin.Context) {  
 c.String(200, "这是一个PUT请求,主要用于修改数据")  
 })  
 r.DELETE("/delete", func(c \*gin.Context) {  
 c.String(200, "这是一个DELETE请求,主要用于增删除数据")  
 })  
 r.Run() // 监听并在 0.0.0.0:8080 上启动服务(启动一个web服务)  
}

8.c.String() c.JSON() c.JSONP() c.XML() c.HTML()

返回一个字符串

r.GET("/news", func(c \*gin.Context) {

aid := c.Query("aid")

c.String(200, "aid=%s", aid)

})

返回一个 JSON 数据

//方法一:自己拼接JSON

r.GET("/json", func(c \*gin.Context) {

//返回json数据,使用 map[string]interface

//c.JSON(返回的状态码, 任意类型的数据(如:map,struct,...)

c.JSON(200, map[string]interface{}{

"success": true,

"msg": "你好",

})

})

//方法二:gin中的H函数

r.GET("/json2", func(c \*gin.Context) {

//返回json数据,使用gin中的H函数, gin.H 是 map[string]interface{}的缩写

c.JSON(200, gin.H{

"success": true,

"msg": "你好gin",

})

})

//方法三:使用结构体

r.GET("/json3", func(c \*gin.Context) {

//实例化一个结构体

a := &Article{

Title: "标题",

Desc: "说明",

Content: "内容",

}

c.JSON(200, a)

})

JSOPN: jsonp请求 主要用来解决跨域问题

//jsonp请求 主要用来解决跨域问题

//http://127.0.0.1:8080/jsonp?callback=call

//call({"title":"标题-jsonp","desc":"说明-jsonp","content":"内容-jsonp"});

r.GET("/jsonp", func(c \*gin.Context) {

//实例化一个结构体

a := &Article{

Title: "标题-jsonp",

Desc: "说明-jsonp",

Content: "内容-jsonp",

}

c.JSONP(200, a)

})

**返回 XML 数据**

//方法一:使用gin.H返回

r.GET("/xml", func(c \*gin.Context) {

c.XML(http.StatusOK, gin.H{

"success": true,

"msg": "成功xml",

})

})

//方法二:使用结构体

r.GET("/xmlStruct", func(c \*gin.Context) {

//实例化一个结构体

a := &Article{

Title: "标题-xmlStruct",

Desc: "说明-xmlStruct",

Content: "内容-xmlStruct",

}

c.XML(200, a)

})

**返回HTML数据**

//初始化路由

r := gin.Default()

//加载templates文件中所有模板文件,以便后续c.HTML()渲染文件时使用

r.LoadHTMLGlob("templates/\*")

r.GET("/news", func(c \*gin.Context) {

//使用模板文件渲染HTML文件

//前提: r.LoadHTMLGlob("templates/\*")

//HTML(状态码, 要渲染的文件名, 加载的参数)

c.HTML(http.StatusOK, "news.html", gin.H{

"title": "我是一个news",

})

})

完整代码案例如下:

package main  
  
import (  
 "github.com/gin-gonic/gin"  
 "net/http"  
)  
  
type Article struct {  
 Title string `json:"title"`  
 Desc string `json:"desc"`  
 Content string `json:"content"`  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //加载templates文件中所有模板文件,以便后续c.HTML()渲染文件时使用  
 r.LoadHTMLGlob("templates/\*")  
 //配置路由  
 r.GET("/", func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 r.GET("/json", func(c \*gin.Context) {  
 //返回json数据,使用 map[string]interface  
 //c.JSON(返回的状态码, 任意类型的数据(如:map,struct,...)  
 c.JSON(200, map[string]interface{}{  
 "success": true,  
 "msg": "你好",  
 })  
 })  
 r.GET("/json2", func(c \*gin.Context) {  
 //返回json数据,使用gin中的H函数  
 c.JSON(200, gin.H{  
 "success": true,  
 "msg": "你好gin",  
 })  
 })  
 r.GET("/json3", func(c \*gin.Context) {  
 //实例化一个结构体  
 a := &Article{  
 Title: "标题",  
 Desc: "说明",  
 Content: "内容",  
 }  
 c.JSON(200, a)  
 })  
 //jsonp请求 主要用来解决跨域问题  
 //http://127.0.0.1:8080/jsonp?callback=call  
 //call({"title":"标题-jsonp","desc":"说明-jsonp","content":"内容-jsonp"});  
 r.GET("/jsonp", func(c \*gin.Context) {  
 //实例化一个结构体  
 a := &Article{  
 Title: "标题-jsonp",  
 Desc: "说明-jsonp",  
 Content: "内容-jsonp",  
 }  
 c.JSONP(200, a)  
 })  
 r.GET("/xml", func(c \*gin.Context) {  
 c.XML(http.*StatusOK*, gin.H{  
 "success": true,  
 "msg": "成功xml",  
 })  
 })  
 r.GET("/news", func(c \*gin.Context) {  
 //使用模板文件渲染HTML文件  
 //前提: r.LoadHTMLGlob("templates/\*")  
 //HTML(状态码, 要渲染的文件名, 加载的参数)  
 c.HTML(http.*StatusOK*, "news.html", gin.H{  
 "title": "我是一个news",  
 })  
 })  
 r.GET("/goods", func(c \*gin.Context) {  
 //使用模板文件渲染HTML文件  
 //前提: r.LoadHTMLGlob("templates/\*")  
 //HTML(状态码, 要渲染的文件名, 加载的参数)  
 c.HTML(http.*StatusOK*, "goods.html", gin.H{  
 "title": "我是一个goods",  
 "price": 12.99,  
 })  
 })  
 r.Run() // 启动一个web服务  
}

## Gin HTML模板渲染以及模板语法,自定义模板函数,静态文件服务

### 2.1.Gin HTML 模板渲染

全部模板放在一个目录里面的配置方法

首先在项目根目录新建 templates 文件夹，然后在文件夹中新建 对应的index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<h1>这是一个 html 模板</h1>

<h3>{{.title}}</h3>

</body>

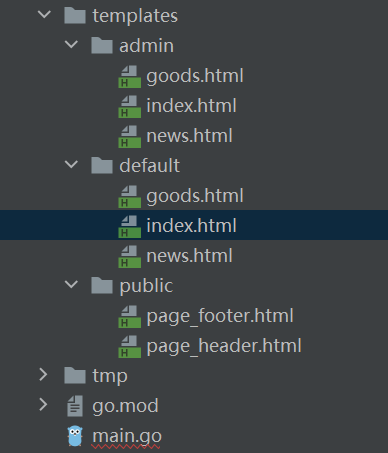
</html>

2.Gin 框架中使用 c.HTML 可以渲染模板，渲染模板前需要使用 LoadHTMLGlob()或者LoadHTMLFiles()方法加载模板

package main  
import (  
 "net/http"  
 "github.com/gin-gonic/gin"  
)  
  
func main() {  
 //初始化路由  
 router := gin.Default()  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 router.LoadHTMLGlob("templates/\*")  
 //router.LoadHTMLFiles("templates/template1.html", "templates/template2.html")  
  
 router.GET("/", func(c \*gin.Context) {  
 c.HTML(http.*StatusOK*, "index.html", gin.H{  
 "title": "首页",  
 })  
 })  
  
 router.GET("/", func(c \*gin.Context) {  
 c.HTML(http.*StatusOK*, "index.html", map[string]interface{}{  
 "title": "前台首页",  
 })  
 })  
 router.Run(":8080")  
}

模板放在不同目录里面的配置方法

目录如下：



Gin 框架中如果不同目录下面有同名模板的话我们需要使用下面方法加载模板  
**注意：**定义模板的时候需要通过 define 定义名称

//templates/admin/index.html

<!-- 相当于给模板定义一个名字 define end 成对出现-->

{{ define "admin/index.html" }}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<h1>后台模板</h1>

<h3>{{.title}}</h3>

</body>

</html>

{{ end }}

==============================分割线=================

templates/default/index.html

<!-- 相当于给模板定义一个名字 define end 成对出现-->

{{ define "default/index.html" }}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<h1>前台模板</h1>

<h3>{{.title}}</h3>

</body>

</html>

{{end}}

代码如下:

package main

import ( "net/http"

"github.com/gin-gonic/gin"

)

func main() {

router := gin.Default()

router.LoadHTMLGlob("templates/\*\*/\*")

router.GET("/", func(c \*gin.Context) {

c.HTML(http.StatusOK, "default/index.html", gin.H{

"title": "前台首页",

})

})

router.GET("/admin", func(c \*gin.Context) {

c.HTML(http.StatusOK, "admin/index.html", gin.H{

"title": "后台首页",

})

})

router.Run(":8080")

}

**注意：**如果模板在多级目录里面的话需要这样配置 r.LoadHTMLGlob("templates/\*\*/\*\*/\*")

/\*\* 表示目录

Gin 模板基本语法

(1)、{{.}} 输出数据

模板语法都包含在{{和}}中间，其中{{.}}中的点表示当前对象,当传入一个结构体对象时，可以根据.来访问结构体的对应字段。例如：

package main  
  
import (  
 "net/http"  
 "github.com/gin-gonic/gin"  
)  
  
type UserInfo struct {  
 Name string  
 Gender string  
 Age int  
}  
func main() {  
 router := gin.Default()  
 router.LoadHTMLGlob("templates/\*\*/\*")  
 user := UserInfo{  
 Name: "张三",  
 Gender: "男",  
 Age: 18,  
 }  
 router.GET("/", func(c \*gin.Context) {  
 c.HTML(http.*StatusOK*, "default/index.html", map[string]interface{}{  
 "title": "前台首页",  
 "user": user,  
 })  
 })  
 router.Run(":8080")  
}

**模板 <!-- 相当于给模板定义一个名字 define end 成对出现-->**

{{ define "default/index.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Document</title>  
</head>  
<body>  
<h1>前台模板</h1>  
<h3>{{.title}}</h3>  
<h4>{{.user.Name}}</h4>  
<h4>{{.user.Age}}</h4>  
</body>  
</html>  
{{end}}

(2).注释

{{*/\* a comment \*/*}}

注释，执行时会忽略,可以多行。注释不能嵌套，并且必须紧贴分界符始止

(3).变量

还可以在模板中声明变量，用来保存传入模板的数据或其他语句生成的结果,具体语法  
如下：

<h4>{{$obj := .title}}</h4>

<h4>{{$obj}}</h4>

(4).移除空格

有时候在使用模板语法的时候会不可避免的引入一下空格或者换行符，这样模板最终渲

染出来的内容可能就和想的不一样，这个时候可以使用{{-语法去除模板内容左侧的所有

空白符号， 使用-}}去除模板内容右侧的所有空白符号,例如:

{{- .Name -}}

注意：-要紧挨{{和}}，同时与模板值之间需要使用空格分隔

(5).比较函数

布尔函数会将任何类型的零值视为假，其余视为真.  
下面是定义为函数的二元比较运算的集合

|  |  |
| --- | --- |
| eq | 如果 arg1 == arg2 则返回真 |
| ne | 如果 arg1 != arg2 则返回真 |
| lt | 如果 arg1 < arg2 则返回真 |
| le | 如果 arg1 <= arg2 则返回真 |
| gt | 如果 arg1 > arg2 则返回真 |
| ge | 如果 arg1 >= arg2 则返回真 |

(6).条件判断

Go 模板语法中的条件判断有以下几种

{{if pipeline}}

T1

{{end}}

{{if pipeline}}

T1

{{else}}

T0

{{end}}

{{if pipeline}}

T1

{{else if pipeline}}

T0

{{end}}

{{if gt .score 60}}

及格

{{else}}

不及格

{{end}}

{{if gt .score 90}}

优秀

{{else if gt .score 60}}

及格

{{else}}

不及格

{{end}}

(6).range

Go 的模板语法中使用 range 关键字进行遍历，有以下两种写法，其中 pipeline 的值必须是数组、切片、字典或者通道

{{range $key,$value := .obj}}

{{$value}}

{{end}}

如果 pipeline 的值其长度为 0，不会有任何输出

{{range $key,$value := .obj}}

{{$value}}

{{else}}

pipeline 的值其长度为 0

{{end}}

如果 pipeline 的值其长度为 0，则会执行 T0

router.GET("/", func(c \*gin.Context) {

c.HTML(http.StatusOK, "default/index.html", map[string]interface{}{

"hobby": []string{

"吃饭", "睡觉", "写代码"

},

})

})

{{range $key,$value := .hobby}}

<p>{{$value}}</p>

{{end}}

(7).with

user := UserInfo{

Name: "张三",

Gender: "男",

Age: 18,

}

router.GET("/", func(c \*gin.Context) {

c.HTML(http.StatusOK, "default/index.html", map[string]interface{}{

"user": user,

})

})

以前要输出数据:

<h4>{{.user.Name}}</h4>

<h4>{{.user.Gender}}</h4>

<h4>{{.user.Age}}</h4>

现在要输出数据:

{{with .user}}

<h4>姓名：{{.Name}}</h4>

<h4>性别：{{.Gender}}</h4>

<h4>年龄：{{.Age}}</h4>

{{end}}

相当于 var .=.user

(8).预定义函数

执行模板时，函数从两个函数字典中查找：首先是模板函数字典，然后是全局函数字典,一般不在模板内定义函数，而是使用 Funcs 方法添加函数到模板里,预定义的全局函数如下:

and函数返回它的第一个 empty 参数或者最后一个参数；就是说"and x y"等价于"if x then y else x"；所有参数都会执行

or返回第一个非 empty 参数或者最后一个参数；亦即"or x y"等价于"if x then x else y"；所有参数都会执行

not返回它的单个参数的布尔值的否定

len返回它的参数的整数类型长度

index执行结果为第一个参数以剩下的参数为索引/键指向的值；如"index x 1 2 3"返回 x[1][2][3]的值；每个被索引的主体必须是数组、切片或者字典

print即 fmt.Sprint

printf即 fmt.Sprintf

println即 fmt.Sprintln

html返回与其参数的文本表示形式等效的转义 HTML。这个函数在 html/template 中不可用

urlquery以适合嵌入到网址查询中的形式返回其参数的文本表示的转义值。

这个函数在 html/template 中不可用

js返回与其参数的文本表示形式等效的转义 JavaScript

call执行结果是调用第一个参数的返回值，该参数必须是函数类型，其余参数作为调用该函数的参数；如"call .X.Y 1 2"等价于 go 语言里的 dot.X.Y(1, 2)；其中 Y 是函数类型的字段或者字典的值，或者其他类似情况；

call 的第一个参数的执行结果必须是函数类型的值（和预定义函数如 print 明显不同）；该函数类型值必须有 1 到 2 个返回值，如果有 2 个则后一个必须是 error 接口类型；如果有 2 个返回值的方法返回的 error 非 nil，模板执行会中断并返回给调用模板执行者该错误

(9).自定义模板函数

package main  
  
import ( "fmt"  
 "html/template"  
 "net/http"  
 "time"  
 "github.com/gin-gonic/gin"  
)  
  
func formatAsDate(t time.Time) string {  
 year, month, day := t.Date()  
 return fmt.Sprintf("%d/%02d/%02d", year, month, day)  
}  
  
func main() {  
 router := gin.Default()  
 //注册全局模板函数 注意顺序，注册模板函数需要在加载模板上面  
 router.SetFuncMap(template.FuncMap{  
 "formatDate": formatAsDate,  
 })  
 //加载模板  
 router.LoadHTMLGlob("templates/\*\*/\*")  
 router.GET("/", func(c \*gin.Context) {  
 c.HTML(http.*StatusOK*, "default/index.html", map[string]interface{}{  
 "title": "前台首页",  
 "now": time.Now(),  
 })  
 })  
 router.Run(":8080")  
}

模版中调用：

{{.now | formatDate}}

//或者

{{formatDate .now }}

4.嵌套 template

(1).新建 templates/deafult/page\_header.html

{{ define "default/page\_header.html" }}

<h1>这是一个头部</h1>

{{end}}

(2).外部引入

注意：

1).引入的名字为 page\_header.html 中定义的名字

2).引入的时候注意最后的点（.）

{{template "default/page\_header.html" .}}

<!-- 相当于给模板定义一个名字 define end 成对出现-->

{{ define "default/index.html" }}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

{{template "default/page\_header.html" .}}

</body>

</html>

{{end}}

### 2.2.静态文件服务

当渲染的HTML 文件中引用了静态文件时,需要配置静态 web 服务r.Static("/static", "./static"),前面的/static 表示路由,后面的./static 表示路径

func main() {

r := gin.Default()

r.Static("/static", "./static")

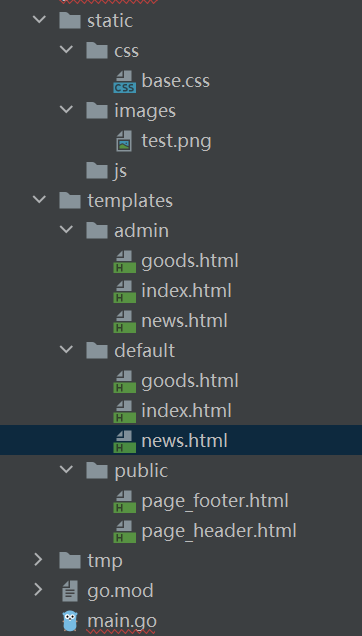
r.LoadHTMLGlob("templates/\*\*/\*")

// ... r.Run(":8080")

}

<link rel="stylesheet" href="/static/css/base.css" />

案例目录如下:



代码如下:

main.go

package main  
  
import (  
 "github.com/gin-gonic/gin"  
 "html/template"  
 "net/http"  
 "time"  
)  
  
type Article struct {  
 Title string `json:"title"`  
 Content string `json:"  
"`  
}  
  
//时间戳转换成日期函数  
func UnixToTime(timestamp int) string {  
 t := time.Unix(int64(timestamp), 0)  
 return t.Format("2006-01-02 15:04:05")  
}  
  
func Println(str1 string, str2 string) string {  
 return str1 + str2  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //自定义模板函数,必须在r.LoadHTMLGlob前面  
 r.SetFuncMap(template.FuncMap{  
 "UnixToTime":UnixToTime, //注册模板函数  
 "Println": Println,  
 })  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 r.LoadHTMLGlob("templates/\*\*/\*")  
 //配置静态web目录 第一个参数表示路由,第二个参数表示映射的目录  
 r.Static("/static", "./static")  
  
 //配置路由  
 //前台路由  
 r.GET("/", func(c \*gin.Context) {  
 //渲染模板文件  
 c.HTML(http.*StatusOK*, "default/index.html", gin.H{  
 "title": "首页",  
 "score": 88,  
 "hobby": []string{"吃饭", "睡觉", "打豆豆"}, // 切片  
 "newList":[]interface{}{ // 接口  
 &Article{  
 Title: "新闻标题1",  
 Content: "新闻内容1",  
 },  
 &Article{  
 Title: "新闻标题2",  
 Content: "新闻内容2",  
 },  
 },  
 "testSlice":[]string{}, // 空数组/空切片  
 "news": &Article{ // 结构体  
 Title: "新闻标题3",  
 Content: "新闻内容3",  
 },  
 "date": 1672648334,  
 })  
 })  
 r.GET("/news", func(c \*gin.Context) {  
 news := &Article{  
 Title: "新闻标题",  
 Content: "新闻内容",  
 }  
 c.HTML(http.*StatusOK*, "default/news.html", gin.H{  
 "title": "新闻详情",  
 "news": news,  
 })  
 })  
  
 //后台路由  
 r.GET("/admin", func(c \*gin.Context) {  
 //渲染模板文件  
 c.HTML(http.*StatusOK*, "admin/index.html", gin.H{  
 "title": "后台首页",  
 })  
 })  
 r.GET("/admin/news", func(c \*gin.Context) {  
 news := &Article{  
 Title: "后台新闻标题",  
 Content: "后台新闻内容",  
 }  
 c.HTML(http.*StatusOK*, "admin/news.html", gin.H{  
 "title": "后台新闻详情",  
 "news": news,  
 })  
 })  
 r.Run() // 启动一个web服务  
}

templates/admin/index.html

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "admin/index.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>后台首页</title>  
</head>  
<body>  
<h2>{{.title}}</h2>  
</body>  
</html>  
{{ end }}

templates/admin/news.html

{{ define "admin/news.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>后台新闻详情</title>  
</head>  
<body>  
<h2>{{.title}}</h2>  
</body>  
</html>  
{{ end }}

templates/default/index.html

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "default/index.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <link rel="stylesheet" href="/static/css/base.css">  
 <title>首页</title>  
</head>  
<body>  
{{ template "public/page\_header.html" .}}  
<h2>{{.title}}</h2>  
<img src="/static/images/test.png">  
<!-- 定义变量 把后台的变量赋值给$t -->  
{{ $t := .title}}  
<!-- 显示$t -->  
<h4> {{ $t }}</h4>  
<!-- if判断 -->  
{{if gt .score 90}}  
<div>优秀</div><br/>  
{{else if gt .score 80}}  
<div>良好</div><br/>  
{{else if gt .score 70}}  
<div>可以</div><br/>  
{{ else }}  
<div>合格</div><br/>  
{{end}}  
<!-- 循环数据 -->  
<ul>  
 <!-- 循环切片 -->  
 {{range $k, $v := .hobby}}  
 <li>{{$k}} => {{$v}}</li>  
 {{end}}  
</ul>  
<ul>  
 <!-- 循环结构体 -->  
 {{range $k, $v := .newList}}  
 <li>{{$k}} => {{$v.Title}} --- {{$v.Content}}</li>  
 {{end}}  
</ul>  
<ul>  
 <!-- 循环空数组/切片 判断 -->  
 {{range $k, $v := .testSlice}}  
 <li>{{$k}} => {{$v.Title}} --- {{$v.Content}}</li>  
 {{else}}  
 <li>空数据</li>  
 {{end}}  
</ul>  
<ul>  
 <!-- with 解析结构体-->  
 {{with .news}}  
 <li>{{.Title}} => {{.Content}}</li>  
 {{end}}  
</ul>  
<h4>  
 <!--预定义函数 -->  
 {{.title}}的长度为{{len .title}}  
</h4>  
<h4>  
 <!--自定义函数 -->  
 {{UnixToTime .date}}  
 {{Println .title .news.Title}}  
</h4>  
<!-- 嵌套 template -->  
{{ template "public/page\_footer.html" .}}  
</body>  
</html>  
{{end}}

templates/default/news.html

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "default/news.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>news</title>  
</head>  
<body>  
{{ template "public/page\_header.html" .}}  
  
<link rel="stylesheet" href="static/css/base.css">  
<h2>{{.title}}</h2>  
<h4>  
 {{.news.Title}}  
 {{.news.Content}}  
</h4>  
{{ template "public/page\_footer.html" .}}  
</body>  
</html>  
{{ end }}

templates/public/page\_footer.html

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "public/page\_footer.html" }}  
<h4>  
 公共的底部  
</h4>  
{{end}}

templates/public/page\_header.html

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "public/page\_header.html" }}  
<h1>  
 公共的 --- {{.title}}  
</h1>  
{{end}}

## 3.路由详解,自定义控制器,中间件

### 3.1.路由详解

路由（Routing）是由一个 URI（或者叫路径）和一个特定的 HTTP 方法（GET、POST 等）组成的，涉及到应用如何响应客户端对某个网站节点的访问,前面介绍了路由基础以及路由配置，这里详细讲讲路由传值、路由返回值

1.GET POST 以及获取 Get Post 传值

(1).Get 请求传值

// GET /user?uid=20&page=1

router.GET("/user", func(c \*gin.Context) {

uid := c.Query("uid")

page := c.DefaultQuery("page", "0")

c.String(200, "uid=%v page=%v", uid, page)

})

(2).动态路由传值

// 域名/user/20

r.GET("/user/:uid", func(c \*gin.Context) {

uid := c.Param("uid")

c.String(200, "userID=%s", uid)

})

(3).Post 请求传值 获取 form 表单数据

// 定义一个 default/add\_user.html 的页面

{{ define "default/add\_user.html" }}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<form action="/doAddUser" method="post">

用户名：<input type="text" name="username" />

密码: <input type="password" name="password" />

<input type="submit" value="提交">

</form>

</body>

</html>

{{end}}

// 通过 c.PostForm 接收表单传过来的数据

router.GET("/addUser", func(c \*gin.Context) {

c.HTML(200, "default/add\_user.html", gin.H{})

})

router.POST("/doAddUser", func(c \*gin.Context) {

username := c.PostForm("username")

password := c.PostForm("password")

age := c.DefaultPostForm("age", "20")

c.JSON(200, gin.H{

"usernmae": username,

"password": password,

"age": age,

})

})

(4).获取 GET POST 传递的数据绑定到结构体

为了能够更方便的获取请求相关参数，提高开发效率，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

//注意首字母大写

type Userinfo struct {

Username string `form:"username" json:"user"`

Password string `form:"password" json:"password"`

}

**Get 传值绑定到结构体**

//?username=zhangsan&password=123456

router.GET("/", func(c \*gin.Context) {

var userinfo Userinfo

if err := c.ShouldBind(&userinfo); err == nil {

c.JSON(http.StatusOK, userinfo)

} else {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

}

})

//返回数据

{"user":"zhangsan","password":"123456"}

**Post 传值绑定到结构体**

router.POST("/doLogin", func(c \*gin.Context) {

var userinfo Userinfo

if err := c.ShouldBind(&userinfo); err == nil {

c.JSON(http.StatusOK, userinfo)

} else {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

}

})

//返回数据

{"user":"zhangsan","password":"123456"}

5).获取 Post Xml 数据

在 API 的开发中，经常会用到 JSON 或 XML 来作为数据交互的格式，这个时候可以在 gin 中使用 c.GetRawData()获取数据

<?xml version="1.0" encoding="UTF-8"?>

<article>

<content type="string">我是张三</content>

<title type="string">张三</title>

</article>

type Article struct {

Title string `xml:"title"`

Content string `xml:"content"`

}

router.POST("/xml", func(c \*gin.Context) {

b, \_ := c.GetRawData() // 从 c.Request.Body 读取请求数据

article := &Article{}

if err := xml.Unmarshal(b, &article); err == nil {

c.JSON(http.StatusOK, article)

} else {

c.JSON(http.StatusBadRequest, err.Error())

}

})

代码汇总:

**main.go**

package main  
  
import (  
 "encoding/xml"  
 "github.com/gin-gonic/gin"  
 "html/template"  
 "net/http"  
 "time"  
)  
  
//时间戳转换成日期函数  
func UnixToTime(timestamp int) string {  
 t := time.Unix(int64(timestamp), 0)  
 return t.Format("2006-01-02 15:04:05")  
}  
  
func Println(str1 string, str2 string) string {  
 return str1 + str2  
}  
  
type UserInfo struct {  
 Username string `json:"username" form:"username"`  
 Password string `json:"password" form:"password"`  
}  
  
type Article struct {  
 Title string `json:"title" xml:"title"`  
 Content string `json:"content" xml:"content"`  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //自定义模板函数,必须在r.LoadHTMLGlob前面  
 r.SetFuncMap(template.FuncMap{  
 "UnixToTime": UnixToTime, //注册模板函数  
 "Println": Println,  
 })  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 r.LoadHTMLGlob("templates/\*\*/\*")  
 //配置静态web目录 第一个参数表示路由,第二个参数表示映射的目录  
 r.Static("/static", "./static")  
 //配置路由  
 //GET传值  
 r.GET("/", func(c \*gin.Context) {  
 username := c.Query("username")  
 age := c.Query("age")  
 page := c.DefaultQuery("page", "1")  
 c.JSON(http.*StatusOK*, gin.H{  
 "username": username,  
 "age": age,  
 "page": page,  
 })  
 })  
 //GET传值 获取文章id  
 r.GET("/article", func(c \*gin.Context) {  
 id := c.DefaultQuery("id", "1")  
 c.JSON(http.*StatusOK*, gin.H{  
 "id": id,  
 "conent": "文章详情",  
 })  
 })  
  
 //post演示  
 r.GET("/user", func(c \*gin.Context) {  
 //渲染模板  
 c.HTML(http.*StatusOK*, "default/user.html", gin.H{})  
 })  
 //获取表单post过来的数据  
 r.POST("/doAddUser", func(c \*gin.Context) {  
 username := c.PostForm("username")  
 password := c.PostForm("password")  
 age := c.DefaultPostForm("age", "20")  
 c.JSON(http.*StatusOK*, gin.H{  
 "username": username,  
 "password": password,  
 "age": age,  
 })  
 })  
 //获取GET POST传递的数据绑定到结构体  
 //r.POST("/getUser", func(c \*gin.Context) {  
 r.GET("/getUser", func(c \*gin.Context) {  
 user := &UserInfo{}  
 //绑定到对应的结构体  
 if err := c.ShouldBind(&user); err == nil {  
 c.JSON(http.*StatusOK*, user)  
 } else {  
 c.JSON(http.*StatusOK*, gin.H{  
 "err": err.Error(),  
 })  
 }  
 })  
 //获取 POST xml数据绑定到结构体  
 r.POST("/xml", func(c \*gin.Context) {  
 article := &Article{}  
 //获取c.Request.Body读取请求数据, 返回的是一个xml切片  
 xmlSliceData, \_ := c.GetRawData()  
 //解析xml切片  
 if err := xml.Unmarshal(xmlSliceData, &article); err == nil {  
 c.JSON(http.*StatusOK*, article)  
 } else {  
 c.JSON(http.*StatusBadRequest*, gin.H{  
 "err": err.Error(),  
 })  
 }  
 })  
  
 //动态路由传值  
 //http://127.0.0.1:8080/list/2221 http://127.0.0.1:8080/list/2  
 r.GET("/list/:id", func(c \*gin.Context) {  
 id := c.Param("id")  
 c.JSON(http.*StatusOK*, gin.H{  
 "id": id,  
 })  
 })  
 r.Run() // 启动一个web服务  
}

**default/user.html**

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->  
{{ define "default/user.html" }}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>news</title>  
</head>  
<body>  
{{ template "public/page\_header.html" .}}  
<form action="/doAddUser" method="post">  
 用户名:<input type="text" name="username" /> <br/> <br/>  
 密码:<input type="text" name="password" /><br/> <br/>  
 <input type="submit" value="提交">  
</form>  
{{ template "public/page\_footer.html" .}}  
</body>  
</html>  
{{ end }}

2.简单的路由组

官方文档: <https://gin-gonic.com/zh-cn/docs/examples/grouping-routes/>

package main  
  
import "github.com/gin-gonic/gin"  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //简单的路由分组  
 defaultRouters := r.Group("/")  
 defaultRouters.GET("/", func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 defaultRouters.GET("/news", func(c \*gin.Context) {  
 c.String(200, "新闻")  
 })  
  
  
 apiRouters := r.Group("/api")  
 { // 可以省略  
 apiRouters.GET("/", func(c \*gin.Context) {  
 c.String(200, "api首页")  
 })  
 apiRouters.GET("/userlist", func(c \*gin.Context) {  
 c.String(200, "一个userlist接口")  
 })  
 } // 可以省略  
  
 adminRouters := r.Group("/admin")  
 {  
 adminRouters.GET("/", func(c \*gin.Context) {  
 c.String(200, "后台首页")  
 })  
 adminRouters.GET("/user", func(c \*gin.Context) {  
 c.String(200, "用户列表")  
 })  
 adminRouters.GET("/article", func(c \*gin.Context) {  
 c.String(200, "新闻列表")  
 })  
 }  
  
 r.Run()  
}

3.Gin 路由文件 分组

(1).新建 routers 文件夹，routers文件下面新建 adminRoutes.go、apiRoutes.go、defaultRoutes.go

1).新建 adminRoutes.go

package routers  
  
import "github.com/gin-gonic/gin"  
  
//设置admin后台路由  
func AdminRoutersInit(r \*gin.Engine) {  
 adminRouters := r.Group("/admin")  
 {  
 adminRouters.GET("/", func(c \*gin.Context) {  
 c.String(200, "后台首页")  
 })  
 adminRouters.GET("/user", func(c \*gin.Context) {  
 c.String(200, "用户列表")  
 })  
 adminRouters.GET("/article", func(c \*gin.Context) {  
 c.String(200, "新闻列表")  
 })  
 }  
}

2).新建 apiRoutes.go

package routers  
  
import "github.com/gin-gonic/gin"  
  
//设置api路由  
func ApiRoutersInit(r \*gin.Engine) {  
 apiRouters := r.Group("/api")  
 {  
 apiRouters.GET("/", func(c \*gin.Context) {  
 c.String(200, "api首页")  
 })  
 apiRouters.GET("/userlist", func(c \*gin.Context) {  
 c.String(200, "一个userlist接口")  
 })  
 }  
}

3).新建 defaultRoutes.go

package routers  
  
import (  
 "github.com/gin-gonic/gin"  
)  
  
//设置前台后台路由  
func DefaultRoutersInit(r \*gin.Engine) {  
 defaultRouters := r.Group("/")  
 {  
 defaultRouters.GET("/", func(c \*gin.Context) {  
 c.HTML(200, "default/index.html", gin.H{  
 "msg":"前台首页",  
 })  
 })  
 defaultRouters.GET("/news", func(c \*gin.Context) {  
 c.String(200, "新闻")  
 })  
 }  
}

(2).配置 main.go

package main  
  
import (  
 "gin\_demo/routers"  
 "github.com/gin-gonic/gin"  
)  
  
func main() {  
 r := gin.Default()  
 //路由文件  
 routers.AdminRoutersInit(r)  
 routers.ApiRoutersInit(r)  
 routers.DefaultRoutersInit(r)  
}

### 3.2.Gin 中自定义控制器

1.控制器分组

当项目比较大的时候有必要对控制器进行分组

(1).新建 controller/admin/articleController.go

package admin  
  
import (  
 "github.com/gin-gonic/gin"  
 "net/http"  
)  
  
type ArticleController struct {  
  
}  
  
func (con ArticleController) Index(c \*gin.Context) {  
 c.String(http.*StatusOK*, "文章列表1")  
}  
  
func (con ArticleController) Add(c \*gin.Context) {  
 c.String(http.*StatusOK*, "文章添加2")  
}  
  
func (con ArticleController) Edit(c \*gin.Context) {  
 c.String(http.*StatusOK*, "文章编辑3")  
}

(2).新建 controller/admin/indexController.go

package admin  
  
import "github.com/gin-gonic/gin"  
  
type IndexController struct {  
  
}  
  
func (con IndexController) Index(c \*gin.Context) {  
 c.String(200, "后台首页")  
}

(3).新建 controller/admin/userController.go

package admin  
  
import "github.com/gin-gonic/gin"  
  
//定义一个UserController结构体,可以实例化结构体访问里面的方法  
type UserController struct {  
  
}  
  
func (con UserController) Index(c \*gin.Context) {  
 c.String(200, "用户列表1")  
 //con.success(c)  
}  
  
func (con UserController) Add(c \*gin.Context) {  
 c.String(200, "用户添加2")  
}  
  
func (con UserController) Edit(c \*gin.Context) {  
 c.String(200, "用户编辑3")  
}

(4).配置对应的路由:adminRoutes.go

package routers  
  
import (  
 "gin\_demo/controller/admin"  
 "github.com/gin-gonic/gin"  
)  
  
//设置admin后台路由  
func AdminRoutersInit(r \*gin.Engine) {  
 adminRouters := r.Group("/admin")  
 {  
 adminRouters.GET("/", admin.IndexController{}.Index)  
 //adminRouters.GET("/user", func(c \*gin.Context) {  
 // c.String(200, "用户列表")  
 //})  
 adminRouters.GET("/user", admin.UserController{}.Index)  
 adminRouters.GET("/user/add", admin.UserController{}.Add)  
 adminRouters.GET("/user/edit", admin.UserController{}.Edit)  
  
 adminRouters.GET("/article", admin.ArticleController{}.Index)  
 adminRouters.GET("/article/add", admin.ArticleController{}.Add)  
 adminRouters.GET("/article/edit", admin.ArticleController{}.Edit)  
 }  
}

**其他路由的配置方法类似**

2.控制器的继承

(1).新建 controller/admin/baseController.go

package admin  
  
import (  
 "github.com/gin-gonic/gin"  
 "net/http"  
)  
  
//基础控制器  
type BaseController struct {  
  
}  
  
func (con BaseController) success(c \*gin.Context) {  
 c.String(http.*StatusOK*, "成功")  
}  
  
func (con BaseController) error(c \*gin.Context) {  
 c.String(http.*StatusOK*, "失败")  
}

(2).userController 继承 baseController

package admin  
  
import "github.com/gin-gonic/gin"  
  
//定义一个UserController结构体,可以实例化结构体访问里面的方法  
type UserController struct {  
 BaseController // 继承基础控制器  
}  
  
func (con UserController) Index(c \*gin.Context) {  
 con.success(c)  
}  
  
func (con UserController) Add(c \*gin.Context) {  
 c.String(200, "用户添加2")  
}  
  
func (con UserController) Edit(c \*gin.Context) {  
 c.String(200, "用户编辑3")  
}

### 2.3.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

通俗的讲：中间件就是匹配路由前和匹配路由完成后执行的一系列操作

1.路由中间件

(1).初识中间件

Gin 中的中间件必须是一个 gin.HandlerFunc 类型，配置路由的时候可以传递多个 func 回调函数，最后一个 func 回调函数前面触发的方法都可以称为中间件

package main  
  
import (  
 "github.com/gin-gonic/gin"  
)  
  
//中间件  
func initMiddleware(c \*gin.Context) {  
 c.String(200, "--1.中间件one--")  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 //r.LoadHTMLGlob("templates/\*\*/\*")  
 //配置静态web目录 第一个参数表示路由,第二个参数表示映射的目录  
 //r.Static("/static", "./static")  
  
 //路由分组  
 defaultRouters := r.Group("/")  
 {  
 defaultRouters.GET("/", initMiddleware, func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 }  
  
 r.Run() // 启动一个web服务  
}

(2)c.Next()调用该请求的剩余处理程序

package main  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
//中间件  
func initMiddleware(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--1.中间件one--")  
 //调用该请求的剩余处理程序  
 c.Next() //   
  
 c.String(200, "--2.中间件one--")  
 end := time.Now().UnixNano()  
 // 计算耗时 Go 语言中的 Since()函数保留时间值，并用于评估与实际时间的差异  
 fmt.Println( end - start)  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
  
 //路由分组  
 defaultRouters := r.Group("/")  
 {  
 //中间件访问,调用该请求的剩余处理程序, c.Next(): 依次输出:--1.中间件one--首页--2.中间件one--  
 defaultRouters.GET("/", initMiddleware, func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 }  
  
 r.Run() // 启动一个web服务  
}

(3).一个路由配置多个中间件的执行顺序

package main  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
//中间件  
func initMiddleware(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--1.中间件one--")  
 //调用该请求的剩余处理程序  
 c.Next()  
  
 c.String(200, "--2.中间件one--")  
 end := time.Now().UnixNano()  
 // 计算耗时 Go 语言中的 Since()函数保留时间值，并用于评估与实际时间的差异  
 fmt.Println( end - start)  
}  
//中间件  
func initMiddlewareTwo(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--2.中间件two--")  
 //调用该请求的剩余处理程序  
 c.Next()  
 c.String(200, "--2.中间件two--")  
 end := time.Now().UnixNano()  
 fmt.Println( end - start)  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 //r.LoadHTMLGlob("templates/\*\*/\*")  
 //配置静态web目录 第一个参数表示路由,第二个参数表示映射的目录  
 //r.Static("/static", "./static")  
  
 //路由分组  
 defaultRouters := r.Group("/")  
 {  
 //多个中间件访问: 依次输出:--1.中间件one----2.中间件two--首页--2.中间件two----2.中间件one--  
 defaultRouters.GET("/", initMiddleware, initMiddlewareTwo, func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 }  
  
 r.Run() // 启动一个web服务  
}

控制台内容：

initMiddleware--1-执行中间件  
initMiddlewareTwo--1-执行中间件  
执行路由里面的程序  
initMiddlewareTwo--2-执行中间件  
initMiddleware--2-执行中间件

(4).c.Abort()终止调用该请求的剩余处理程序

Abort 是终止的意思， c.Abort() 表示终止调用该请求的剩余处理程序

package main  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
//中间件  
func initMiddlewareAbort(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--1.中间件abort--")  
 //调用该请求的剩余处理程序  
 //c.Next()  
 //终止该请求的剩余处理程序  
 c.Abort()  
  
 c.String(200, "--2.中间件abort--")  
 end := time.Now().UnixNano()  
 fmt.Println( end - start)  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
  
 //路由分组  
 defaultRouters := r.Group("/")  
 {  
 //中间件访问,终止该请求的剩余处理程序, c.Abort(): 依次输出:--1.中间件abort----2.中间件abort--  
 //不会执行最后的HandFunc程序  
 defaultRouters.GET("/", initMiddlewareAbort, func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 }  
  
 r.Run() // 启动一个web服务  
}

1. 全局中间件

package main  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
  
//中间件  
func initMiddleware(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--1.中间件one--")  
 //调用该请求的剩余处理程序  
 c.Next()  
 //终止该请求的剩余处理程序  
 //c.Abort()  
  
 c.String(200, "--2.中间件one--")  
 end := time.Now().UnixNano()  
 fmt.Println( end - start)  
}  
  
//中间件  
func initMiddlewareAbort(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--1.中间件abort--")  
 //调用该请求的剩余处理程序  
 //c.Next()  
 //终止该请求的剩余处理程序  
 c.Abort()  
  
 c.String(200, "--2.中间件abort--")  
 end := time.Now().UnixNano()  
 fmt.Println( end - start)  
}  
  
  
//中间件  
func initMiddlewareTwo(c \*gin.Context) {  
 start := time.Now().UnixNano()  
 c.String(200, "--2.中间件two--")  
 //调用该请求的剩余处理程序  
 c.Next()  
 //终止该请求的剩余处理程序  
 //c.Abort()  
  
 c.String(200, "--2.中间件two--")  
 end := time.Now().UnixNano()  
 fmt.Println( end - start)  
}  
  
func main() {  
 //初始化路由  
 r := gin.Default()  
  
 //全局中间件  
 r.Use(initMiddleware, initMiddlewareTwo)  
 //路由分组  
 defaultRouters := r.Group("/")  
 {  
 //中间件访问,终止该请求的剩余处理程序, c.Abort(): --1.中间件one----2.中间件two----1.中间件abort----2.中间件abort----2.中间件two----2.中间件one--  
 //不会执行最后的HandFunc程序  
 defaultRouters.GET("/", initMiddlewareAbort, func(c \*gin.Context) {  
 c.String(200, "首页")  
 })  
 }  
  
 r.Run() // 启动一个web服务  
}

3.在路由分组中配置中间件

(1).为路由组注册中间件有以下两种写法

写法 1：

userGroup := r.Group("/user", UserOper())

{

userGroup.GET("/index", func(c \*gin.Context) {

...

})

...

}

写法 2：

userGroup := r.Group("/user")

userGroup.Use(UserOper())

{

userGroup.GET("/index", func(c \*gin.Context) {

...

})

...

}

2).分组路由adminRoutes.go 中配置中间件

adminRoutes.go

package routers  
  
import (  
 "gin\_demo/controller/admin"  
 "gindemo/middlewares"  
 "github.com/gin-gonic/gin"  
)  
  
//设置admin后台路由  
func AdminRoutersInit(r \*gin.Engine) {  
 //路由分组: 配置全局中间件:middlewares.InitMiddleware  
 adminRouters := r.Group("/admin", middlewares.InitMiddleware)  
 {  
 adminRouters.GET("/", admin.IndexController{}.Index)// 实例化控制器,并访问其中方法  
  
 adminRouters.GET("/user", admin.UserController{}.Index)  
 adminRouters.GET("/user/add",admin.UserController{}.Add)  
 adminRouters.GET("/user/edit", admin.UserController{}.Edit)  
  
 adminRouters.GET("/article", admin.ArticleController{}.Index)  
 adminRouters.GET("/article/add", admin.ArticleController{}.Add)  
 adminRouters.GET("/article/edit", admin.ArticleController{}.Edit)  
 }  
}

middlewares/init.go

package middlewares  
  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
func InitMiddleware( c \*gin.Context) {  
 //判断用户是否登录  
 fmt.Println(time.Now())  
 fmt.Println(c.Request.URL)  
}

4.中间件和对应控制器之间共享数据

(1).设置值

c.Set("username","张三")

(2).获取值

username, \_ := c.Get("username")

(3).中间件设置值

middlewares/init.go

package middlewares  
  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
func InitMiddleware( c \*gin.Context) {  
 //判断用户是否登录  
 fmt.Println(time.Now())  
 fmt.Println(c.Request.URL)  
 //设置值, 和对应控制器之间共享数据  
 c.Set("username", "张三")  
}

控制器获取值 controlelers/adimn/indexController.go

package admin  
  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
)  
  
type IndexController struct {  
  
}  
  
func (con IndexController) Index(c \*gin.Context) {  
 //获取中间件中设置的username值,数据共享  
 username, \_ := c.Get("username")  
 fmt.Println(username)  
 //username是一个空接口类型,故要使用则需要用类型断言转换username  
 v, ok := username.(string)  
 if ok != true {  
 c.String(200, "后台首页--获取用户名失败")  
 } else {  
 c.String(200, "后台首页,用户名:" + v)  
 }  
  
}

5.中间件注意事项

(1).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

(2).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()）

package middlewares  
  
import (  
 "fmt"  
 "github.com/gin-gonic/gin"  
 "time"  
)  
  
func InitMiddleware( c \*gin.Context) {  
 //判断用户是否登录  
 fmt.Println(time.Now())  
 fmt.Println(c.Request.URL)  
 //设置值, 和对应控制器之间共享数据  
 c.Set("username", "张三")  
  
 //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()）  
 cCp := c.Copy()  
 go func() {  
 time.Sleep(2 \* time.*Second*)  
 fmt.Println("gin 中间件中使用 goroutine" + cCp.Request.URL.Path)  
 }()  
}

## 自定义Model以及Gin 文件上传

一.Gin 中自定义 Model

关于 Model

如果应用非常简单的话，我们可以在 Controller 里面处理常见的业务逻辑,但是如果有一个功能想在多个控制器、或者多个模板里面复用的话，那么就可以把公共的功能单独抽取出来作为一个模块（Model）,Model 是逐步抽象的过程，一般会在 Model里面封装一些公共的方法让不同 Controller 使用，也可以在 Model 中实现和数据库打交道

2.Model 里面封装公共的方法

新建 models/ tools.go

package models  
  
import (  
 "crypto/md5"  
 "fmt"  
 "time"  
)  
  
//时间戳转换成日期函数  
func UnixToTime(timestamp int) string {  
 t := time.Unix(int64(timestamp), 0)  
 return t.Format("2006-01-02 15:04:05")  
}  
  
//日期转换成时间戳  
func DateToUnix(str string) int64 {  
 template := "2006-01-02 15:04:05"  
 t, err := time.ParseInLocation(template, str, time.Local)  
 if err != nil {  
 return 0  
 }  
 return t.Unix()  
}  
  
//获取当前时间戳  
func GetUnix() int64 {  
 return time.Now().Unix()  
}  
  
//获取当前日期  
func GetDate() string {  
 template := "2006-01-02 15:04:05"  
 return time.Now().Format(template)  
}  
  
//获取年月日  
func GetDay() string {  
 template := "20060102"  
 return time.Now().Format(template)  
}  
  
func Md5(str string) string {  
 data := []byte(str)  
 return fmt.Sprintf("%x\n", md5.Sum(data))  
}  
func Hello(in string) (out string) {  
 out = in + "world"  
 return  
}

3.控制器中调用 Model

package controllers

import (

"gindemo/models"

)

day := models.GetDay()

4.调用 Model 注册全局模板函数

(1).main.go中

package main  
  
import (  
 "fmt"  
 "gindemo/models"  
 "github.com/gin-gonic/gin"  
 "html/template"  
)  
  
//main中的公共方法  
func Println(str1 string, str2 string) string {  
 return str1 + str2  
}  
  
func main() {  
 //初始化路由,会设置默认中间件:engine.Use(Logger(), Recovery())，可以使用gin.New()来设置路由  
 r := gin.Default()  
 //自定义模板函数,必须在r.LoadHTMLGlob前面  
 r.SetFuncMap(template.FuncMap{  
 "UnixToTime": models.UnixToTime, //注册模板函数  
 "Println": Println,  
 })  
 //加载templates中所有模板文件, 使用不同目录下名称相同的模板,注意:一定要放在配置路由之前才得行  
 r.LoadHTMLGlob("templates/\*\*/\*")  
 //配置静态web目录 第一个参数表示路由,第二个参数表示映射的目录  
 r.Static("/static", "./static")  
 ...  
}

(2).控制器使用模板中函数:方式一

控制器UserController

func (c UserController) Add(c \*gin.Context) {

c.HTML(http.StatusOK, "admin/user/add.html", gin.H{

"now": models.GetUnix(),

})

}

模板admin/user/add.html

<h2>{{.now | unixToDate}}</h2>

(3).控制器,模板中:方式二

控制器defaultController.go

package itying

import (

\_ "fmt"

"gindemo/models"

"github.com/gin-gonic/gin"

"net/http"

)

type DefaultController struct {

}

func (con DefaultController) Index(c \*gin.Context) {

fmt.Println(models.GetDay())

c.HTML(http.StatusOK, "default/index.html",gin.H{

"msg":"我是一个msg",

"t": 1629788010,

})

}

模板default/index.html：

<!-- 相当于给模板定义一个名字, define end 必须成对出现 -->

{{ define "default/index.html" }}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="stylesheet" href="/static/css/base.css">

<title>首页</title>

</head>

<body>

{{ template "public/page\_header.html" .}}

<h2>{{.msg}}</h2>

<h3>{{UnixToTime .t}}</h3>

{{ template "public/page\_footer.html" .}}

</body>

</html>

{{end}}

5.Golang Md5 加密

方法一

package models

import (

"crypto/md5"

)

func Md5(str string) string {

data := []byte(str)

return fmt.Sprintf("%x\n", md5.Sum(data))

}

方法二：

package models

import (

"crypto/md5"

)

func Md5(str string) string {

h := md5.New()

io.WriteString(h, str)

return fmt.Sprintf("%x\n", h.Sum(nil))

}

后续详细见gin框架教程v4.0