

目 录

简介

gin路由

基本路由

Restful风格的API

API参数

URL参数

表单参数

上传单个文件

上传多个文件

routes group

404页面设置

路由原理

gin 数据解析和绑定

Json 数据解析和绑定

表单数据解析和绑定

URI数据解析和绑定

gin 渲染

各种数据格式的响应

HTML模板渲染

重定向

同步异步

gin 中间件

全局中间件

Next()方法

局部中间件

中间件练习

中间件推荐

会话控制

Cookie介绍

Cookie的使用

Cookie练习

Cookie的缺点

Sessions

参数验证

结构体验证

自定义验证

自定义验证v10

多语言翻译验证

其他

日志文件

Air实时加载

简介

简介

介绍

- Gin是一个golang的微框架，封装比较优雅，API友好，源码注释比较明确，具有快速灵活，容错方便等特点
- 对于golang而言，web框架的依赖要远比Python，Java之类的要小。自身的 `net/http` 足够简单，性能也非常不错
- 借助框架开发，不仅可以省去很多常用的封装带来的时间，也有助于团队的编码风格和形成规范

安装

要安装Gin软件包，您需要安装Go并首先设置Go工作区。

- 1.首先需要安装Go（需要1.10+版本），然后可以使用下面的Go命令安装Gin。

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

- 2.将其导入您的代码中：

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

- 3.（可选）导入net/http。例如，如果使用常量，则需要这样做http.StatusOK。

```
import "net/http"
```

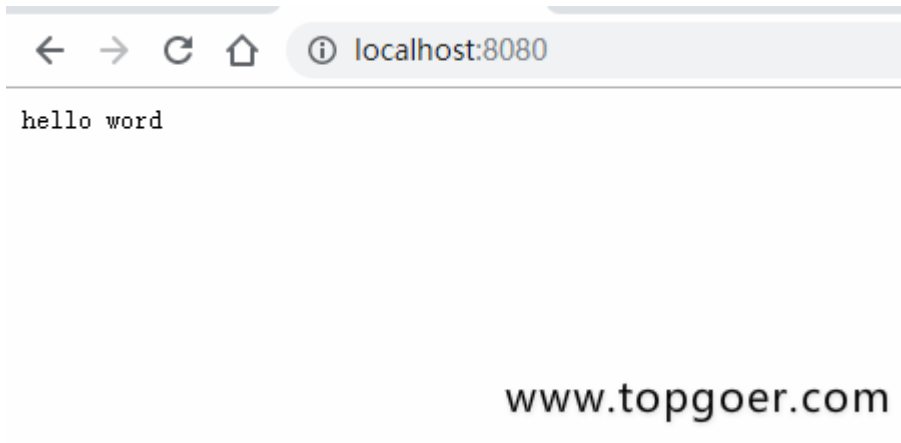
hello word

```
package main

import (
    "net/http"
```

```
    "github.com/gin-gonic/gin"
)

func main() {
    // 1. 创建路由
    r := gin.Default()
    // 2. 绑定路由规则, 执行的函数
    // gin.Context, 封装了request和response
    r.GET("/", func(c *gin.Context) {
        c.String(http.StatusOK, "hello World!")
    })
    // 3. 监听端口, 默认在8080
    // Run("里面不指定端口号默认为8080")
    r.Run(":8000")
}
```



gin路由

[基本路由](#)

[Restful风格的API](#)

[API参数](#)

[URL参数](#)

[表单参数](#)

[上传单个文件](#)

[上传多个文件](#)

[routes group](#)

[404页面设置](#)

[路由原理](#)

基本路由

基本路由

- gin 框架中采用的路由库是基于httprouter做的
- 地址为: <https://github.com/julienschmidt/httprouter>

```
package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.String(http.StatusOK, "hello word")
    })
    r.POST("/xxxpost", getting)
    r.PUT("/xxxput")
    //监听端口默认为8080
    r.Run(":8000")
}
```

Restful风格的API

Restful风格的API

- gin支持Restful风格的API
- 即Representational State Transfer的缩写。直接翻译的意思是“表现层状态转化”，是一种互联网应用程序的API设计理念：URL定位资源，用HTTP描述操作

1. 获取文章 /blog/getXxx Get blog/Xxx
2. 添加 /blog/addXxx POST blog/Xxx
3. 修改 /blog/updateXxx PUT blog/Xxx
4. 删除 /blog/delXxxx DELETE blog/Xxx

API参数

API参数

- 可以通过Context的Param方法来获取API参数
- localhost:8000/xxx/zhangsan

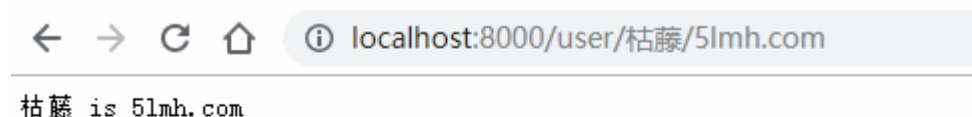
```
package main

import (
    "net/http"
    "strings"

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

func main() {
    r := gin.Default()
    r.GET("/user/:name/*action", func(c *gin.Context) {
        name := c.Param("name")
        action := c.Param("action")
        //截取/
        action = strings.Trim(action, "/")
        c.String(http.StatusOK, name+" is "+action)
    })
    //默认为监听8080端口
    r.Run(":8000")
}
```

输出结果:



www.topgoer.com

URL参数

URL参数

- URL参数可以通过DefaultQuery()或Query()方法获取
- DefaultQuery()若参数不村则，返回默认值，Query()若不存在，返回空串
- API ? name=zs

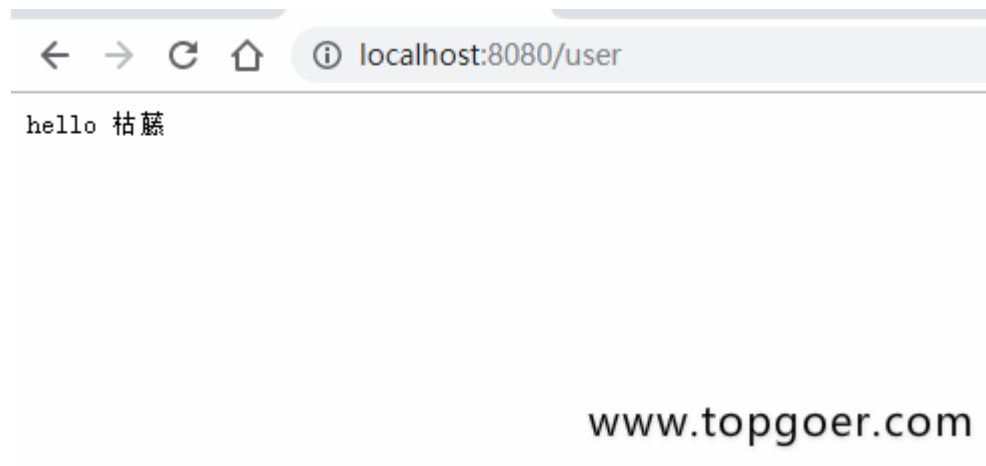
```
package main

import (
    "fmt"
    "net/http"

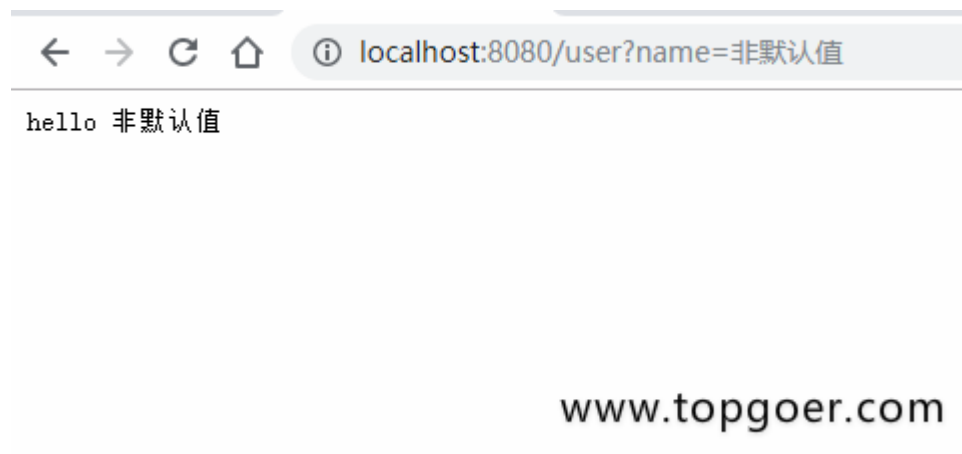
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    r.GET("/user", func(c *gin.Context) {
        //指定默认值
        //http://localhost:8080/user 才会打印出来默认的值
        name := c.DefaultQuery("name", "枯藤")
        c.String(http.StatusOK, fmt.Sprintf("hello %s", name))
    })
    r.Run()
}
```

不传递参数输出的结果:



传递参数输出的结果:




```
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    r.POST("/form", func(c *gin.Context) {
        types := c.DefaultPostForm("type", "post")
        username := c.PostForm("username")
        password := c.PostForm("userpassword")
        // c.String(http.StatusOK, fmt.Sprintf("username:%s,password:%s,type:%s",
, username, password, types))
        c.String(http.StatusOK, fmt.Sprintf("username:%s,password:%s,type:%s", u
sername, password, types))
    })
    r.Run()
}
```

输出结果:

浏览器地址: E:/goproject/src/github.com/student/gingo/ce4/index.html

用户名:

密码:

username: 枯藤, password: 123456, type: post

www.topgoer.com

上传单个文件

上传单个文件

- multipart/form-data格式用于文件上传
- gin文件上传与原生的net/http方法类似，不同在于gin把原生的request封装到c.Request中

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <form action="http://localhost:8080/upload" method="post" enctype="multipart/form-data">
    上传文件:<input type="file" name="file" >
    <input type="submit" value="提交">
  </form>
</body>
</html>
```

```
package main

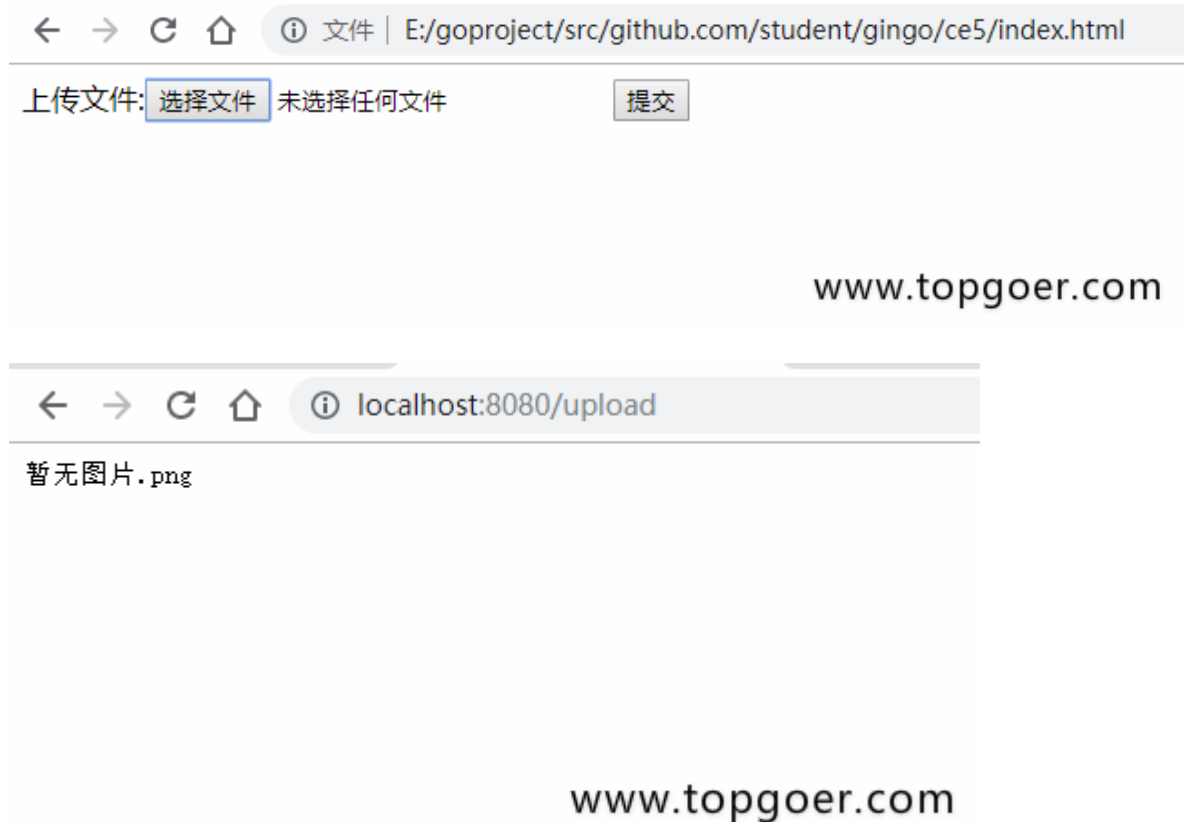
import (
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    //限制上传最大尺寸
    r.MaxMultipartMemory = 8 << 20
    r.POST("/upload", func(c *gin.Context) {
        file, err := c.FormFile("file")
        if err != nil {
            c.String(500, "上传图片出错")
        }
    })
}
```

上传单个文件

```
    }  
    // c.JSON(200, gin.H{"message": file.Header.Context})  
    c.SaveUploadedFile(file, file.Filename)  
    c.String(http.StatusOK, file.Filename)  
  })  
  r.Run()  
}
```

效果演示:



上传特定文件

有的用户上传文件需要限制上传文件的类型以及上传文件的大小,但是gin框架暂时没有这些函数(也有可能是我没找到),因此基于原生的函数写法自己写了一个可以限制大小以及文件类型的上传函数

```
package main  
  
import (  
    "fmt"  
    "log"  
    "net/http"
```

```
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    r.POST("/upload", func(c *gin.Context) {
        _, headers, err := c.Request.FormFile("file")
        if err != nil {
            log.Printf("Error when try to get file: %v", err)
        }
        //headers.Size 获取文件大小
        if headers.Size > 1024*1024*2 {
            fmt.Println("文件太大了")
            return
        }
        //headers.Header.Get("Content-Type")获取上传文件的类型
        if headers.Header.Get("Content-Type") != "image/png" {
            fmt.Println("只允许上传png图片")
            return
        }
        c.SaveUploadedFile(headers, "./video/"+headers.FileName)
        c.String(http.StatusOK, headers.FileName)
    })
    r.Run()
}
```

上传多个文件

上传多个文件

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <form action="http://localhost:8000/upload" method="post" enctype="multipart/form-data">
    上传文件:<input type="file" name="files" multiple>
    <input type="submit" value="提交">
  </form>
</body>
</html>
```

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
    "fmt"
)

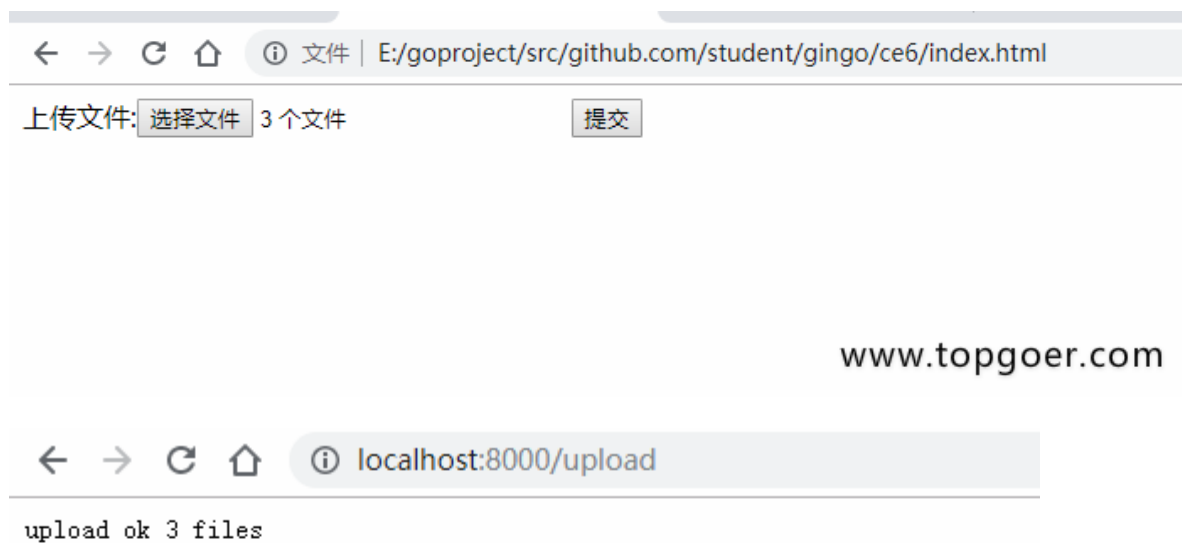
// gin的helloWorld

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 限制表单上传大小 8MB, 默认为32MB
    r.MaxMultipartMemory = 8 << 20
    r.POST("/upload", func(c *gin.Context) {
        form, err := c.MultipartForm()
        if err != nil {
            c.String(http.StatusBadRequest, fmt.Sprintf("get err %s", err.Error()))
        }
    })
}
```


上传多个文件

```
    }  
    // 获取所有图片  
    files := form.File["files"]  
    // 遍历所有图片  
    for _, file := range files {  
        // 逐个存  
        if err := c.SaveUploadedFile(file, file.Filename); err != nil {  
            c.String(http.StatusBadRequest, fmt.Sprintf("upload err %s", err.Error()))  
        }  
        return  
    }  
    c.String(200, fmt.Sprintf("upload ok %d files", len(files)))  
})  
//默认端口号是8080  
r.Run(":8000")  
}
```

效果演示:



routes group

routes group

- routes group是为了管理一些相同的URL

```
package main

import (
    "github.com/gin-gonic/gin"
    "fmt"
)

// gin的helloWorld

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 路由组1, 处理GET请求
    v1 := r.Group("/v1")
    // {} 是书写规范
    {
        v1.GET("/login", login)
        v1.GET("/submit", submit)
    }
    v2 := r.Group("/v2")
    {
        v2.POST("/login", login)
        v2.POST("/submit", submit)
    }
    r.Run(":8000")
}

func login(c *gin.Context) {
    name := c.DefaultQuery("name", "jack")
    c.String(200, fmt.Sprintf("hello %s\n", name))
}

func submit(c *gin.Context) {
    name := c.DefaultQuery("name", "lily")
}
```

routes group

```
c.String(200, fmt.Sprintf("hello %s\n", name))  
}
```

效果演示:

```
C:\Users\Administrator>curl http://127.0.0.1:8000/v1/login?name=zhangsan  
hello zhangsan  
C:\Users\Administrator>curl http://127.0.0.1:8000/v1/submit -X POST  
404 page not found  
C:\Users\Administrator>curl http://127.0.0.1:8000/v2/submit -X POST  
hello lily  
www.topgoer.com
```

404页面设置

gin框架实现404页面

```
package main

import (
    "fmt"
    "net/http"

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

func main() {
    r := gin.Default()
    r.GET("/user", func(c *gin.Context) {
        //指定默认值
        //http://localhost:8080/user 才会打印出来默认的值
        name := c.DefaultQuery("name", "枯藤")
        c.String(http.StatusOK, fmt.Sprintf("hello %s", name))2020-08-05 09:22:1
1 星期三
    })
    r.NoRoute(func(c *gin.Context) {
        c.String(http.StatusNotFound, "404 not found2222")
    })
    r.Run()
}
```

路由原理

路由原理

- httprouter会将所有路由规则构造一颗前缀树
- 例如有 root and as at cn com

路由原理

- httprouter会将所有路由规则构造一颗前缀树
- 例如有 root and as at cn com

三、gin 数据解析和绑定

- 1. json 数据解析和绑定
 - 客户端传参，后端接收并解析到结构
- 2. 表单数据解析和绑定
- 3. URI 数据解析和绑定

3 页, 共 14 页 1794 个字 英语(美国)

gin 数据解析和绑定

[Json 数据解析和绑定](#)

[表单数据解析和绑定](#)

[URI数据解析和绑定](#)

Json 数据解析和绑定

Json 数据解析和绑定

- 客户端传参，后端接收并解析到结构体

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

// 定义接收数据的结构体
type Login struct {
    // binding:"required"修饰的字段，若接收为空值，则报错，是必须字段
    User    string `form:"username" json:"user" uri:"user" xml:"user" binding:"required"`
    Pssword string `form:"password" json:"password" uri:"password" xml:"password" binding:"required"`
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // JSON绑定
    r.POST("loginJSON", func(c *gin.Context) {
        // 声明接收的变量
        var json Login
        // 将request的body中的数据，自动按照json格式解析到结构体
        if err := c.ShouldBindJSON(&json); err != nil {
            // 返回错误信息
            // gin.H封装了生成json数据的工具
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        // 判断用户名密码是否正确
        if json.User != "root" || json.Pssword != "admin" {
            c.JSON(http.StatusBadRequest, gin.H{"status": "304"})
            return
        }
    })
}
```

```
c. JSON(http.StatusOK, gin.H{"status": "200"})
})
r. Run(":8000")
}
```

效果演示:

```
C:\Users\Administrator>curl http://127.0.0.1:8000/loginJSON -H 'content-type:application/json' -d {"user":"root","password":"admin"} -X POST
{"error":"invalid character 'u' looking for beginning of object key string"}

C:\Users\Administrator>curl http://127.0.0.1:8000/loginJSON -H 'content-type:application/json' -d "{\"user\":\"root\", \"password\":\"admin\"}" -X POST
{"status":"200"}

C:\Users\Administrator>curl http://127.0.0.1:8000/loginJSON -H 'content-type:application/json' -d "{\"user\":\"root\", \"password\":\"admin2\"}" -X POST
{"status":"304"}
```


表单数据解析和绑定

表单数据解析和绑定

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <form action="http://localhost:8000/loginForm" method="post" enctype="application/x-www-form-urlencoded">
    用户名<input type="text" name="username"><br>
    密码<input type="password" name="password">
    <input type="submit" value="提交">
  </form>
</body>
</html>
```

```
package main

import (
    "net/http"

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

// 定义接收数据的结构体
type Login struct {
    // binding:"required"修饰的字段, 若接收为空值, 则报错, 是必须字段
    User string `form:"username" json:"user" uri:"user" xml:"user" binding:"required"`
    Pssword string `form:"password" json:"password" uri:"password" xml:"password" binding:"required"`
}

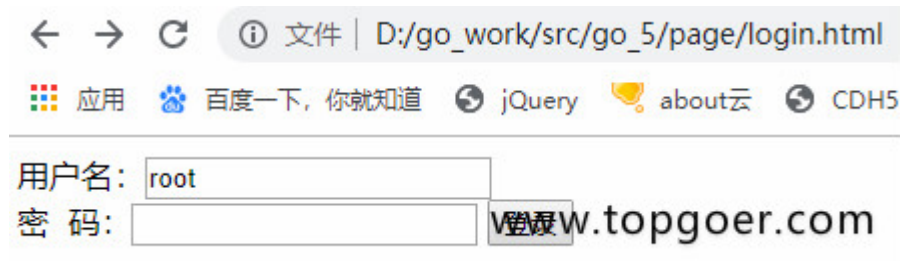
func main() {
    // 1. 创建路由
```

```

// 默认使用了2个中间件Logger(), Recovery()
r := gin.Default()
// JSON绑定
r.POST("/loginForm", func(c *gin.Context) {
    // 声明接收的变量
    var form Login
    // Bind()默认解析并绑定form格式
    // 根据请求头中content-type自动推断
    if err := c.Bind(&form); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    // 判断用户名密码是否正确
    if form.User != "root" || form.Pssword != "admin" {
        c.JSON(http.StatusBadRequest, gin.H{"status": "304"})
        return
    }
    c.JSON(http.StatusOK, gin.H{"status": "200"})
})
r.Run(":8000")
}

```

效果展示:



URI数据解析和绑定

URI数据解析和绑定

```
package main

import (
    "net/http"

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

// 定义接收数据的结构体
type Login struct {
    // binding:"required"修饰的字段, 若接收为空值, 则报错, 是必须字段
    User    string `form:"username" json:"user" uri:"user" xml:"user" binding:"required"`
    Pssword string `form:"password" json:"password" uri:"password" xml:"password" binding:"required"`
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // JSON绑定
    r.GET("/:user/:password", func(c *gin.Context) {
        // 声明接收的变量
        var login Login
        // Bind()默认解析并绑定form格式
        // 根据请求头中content-type自动推断
        if err := c.ShouldBindUri(&login); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        // 判断用户名密码是否正确
        if login.User != "root" || login.Pssword != "admin" {
            c.JSON(http.StatusBadRequest, gin.H{"status": "304"})
            return
        }
        c.JSON(http.StatusOK, gin.H{"status": "200"})
    })
}
```

```
r.Run(":8000")  
}
```

效果演示:

```
C:\Users\Administrator>curl http://127.0.0.1:8000/root/admin  
{"status": "200"}
```

gin 渲染

各种数据格式的响应

HTML模板渲染

重定向

同步异步

各种数据格式的响应

各种数据格式的响应

- json、结构体、XML、YAML类似于java的properties、ProtoBuf

```
package main

import (
    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/testdata/protoexample"
)

// 多种响应方式
func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 1. json
    r.GET("/someJSON", func(c *gin.Context) {
        c.JSON(200, gin.H{"message": "someJSON", "status": 200})
    })
    // 2. 结构体响应
    r.GET("/someStruct", func(c *gin.Context) {
        var msg struct {
            Name    string
            Message string
            Number  int
        }
        msg.Name = "root"
        msg.Message = "message"
        msg.Number = 123
        c.JSON(200, msg)
    })
    // 3. XML
    r.GET("/someXML", func(c *gin.Context) {
        c.XML(200, gin.H{"message": "abc"})
    })
    // 4. YAML响应
    r.GET("/someYAML", func(c *gin.Context) {
        c.YAML(200, gin.H{"name": "zhangsan"})
    })
}
```

```
// 5. protobuf格式, 谷歌开发的高效存储读取的工具
// 数组? 切片? 如果自己构建一个传输格式, 应该是什么格式?
r.GET("/someProtoBuf", func(c *gin.Context) {
    reps := []int64{int64(1), int64(2)}
    // 定义数据
    label := "label"
    // 传protobuf格式数据
    data := &protoexample.Test{
        Label: &label,
        Reps:   reps,
    }
    c.ProtoBuf(200, data)
})

r.Run(":8000")
}
```

HTML模板渲染

HTML模板渲染

- gin支持加载HTML模板, 然后根据模板参数进行配置并返回相应的数据, 本质上就是字符串替换
- LoadHTMLGlob()方法可以加载模板文件

```
package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.LoadHTMLGlob("tem/*")
    r.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", gin.H{"title": "我是测试", "ce": "123456"})
    })
    r.Run()
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>{{.title}}</title>
</head>
<body>
    fgkjdskjds{{.ce}}
</body>
</html>
```

目录结构:



- 如果你的目录结构是下面的情况



代码如下：

```
package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.LoadHTMLGlob("tem/**/*")
    r.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "user/index.html", gin.H{"title": "我是测试", "address": "www.5lmh.com"})
    })
    r.Run()
}
```

```
{{ define "user/index.html" }}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>{{.title}}</title>
</head>
```

```

<body>
  fgkjds kjdsh{{. address}}
</body>
</html>
{{ end }}

```

- 如果你想进行头尾分离就是下面这种写法了:



```

package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.LoadHTMLGlob("tem/**/*")
    r.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "user/index.html", gin.H{"title": "我是测试", "address": "www.5lmh.com"})
    })
    r.Run()
}

```

user/index.html 文件代码:

```

{{ define "user/index.html" }}
{{ template "public/header" . }}
  fgkjds kjdsh{{. address}}
{{ template "public/footer" . }}
{{ end }}

```

public/header.html 文件代码:

```
{{define "public/header"}}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <meta http-equiv="X-UA-Compatible" content="ie=edge">  
  <title>{{.title}}</title>  
</head>  
  <body>  
  
{{end}}
```

public/footer.html 文件代码:

```
{{define "public/footer"}}  
</body>  
</html>  
{{ end }}
```

- 如果你需要引入静态文件需要定义一个静态文件目录

```
r.Static("/assets", "./assets")
```

重定向

重定向

```
package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.GET("/index", func(c *gin.Context) {
        c.Redirect(http.StatusMovedPermanently, "http://www.51mh.com")
    })
    r.Run()
}
```

同步异步

同步异步

- goroutine机制可以方便地实现异步处理
- 另外，在启动新的goroutine时，不应该使用原始上下文，必须使用它的只读副本

```
package main

import (
    "log"
    "time"

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

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 1. 异步
    r.GET("/long_async", func(c *gin.Context) {
        // 需要搞一个副本
        copyContext := c.Copy()
        // 异步处理
        go func() {
            time.Sleep(3 * time.Second)
            log.Println("异步执行: " + copyContext.Request.URL.Path)
        }()
    })
    // 2. 同步
    r.GET("/long_sync", func(c *gin.Context) {
        time.Sleep(3 * time.Second)
        log.Println("同步执行: " + c.Request.URL.Path)
    })

    r.Run(":8000")
}
```

gin 中间件

[全局中间件](#)

[Next\(\)方法](#)

[局部中间件](#)

[中间件练习](#)

[中间件推荐](#)

全局中间件

全局中间件

- 所有请求都经过此中间件

```
package main

import (
    "fmt"
    "time"

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

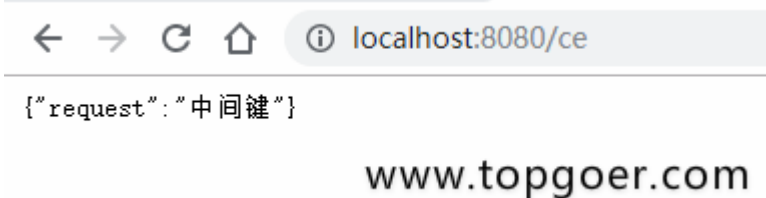
// 定义中间
func MiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()
        fmt.Println("中间件开始执行了")
        // 设置变量到Context的key中, 可以通过Get()取
        c.Set("request", "中间件")
        status := c.Writer.Status()
        fmt.Println("中间件执行完毕", status)
        t2 := time.Since(t)
        fmt.Println("time:", t2)
    }
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 注册中间件
    r.Use(MiddleWare())
    // {}为了代码规范
    {
        r.GET("/ce", func(c *gin.Context) {
            // 取值
            req, _ := c.Get("request")
            fmt.Println("request:", req)
            // 页面接收
        })
    }
}
```

```
        c.JSON(200, gin.H{"request": req})
    })
}
r.Run()
```

输出结果:

```
[GIN-debug] Environment variable PORT is undefined. Using port :8080 by default
[GIN-debug] Listening and serving HTTP on :8080
中间键开始执行了
中间键执行完毕 200
time 0s
request: 中间键
[GIN] 2019/10/23 - 16:42:27 |?[97;42m 200 ?[0m|      5.9979ms |      ::1 |?[97;44m www.topgoer.com
```



注意:

请注意黑色的数据里面有一步算时间差没有执行(需要学习Next就懂了)

Next()方法

Next()方法

```
package main

import (
    "fmt"
    "time"

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

// 定义中间
func MiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()
        fmt.Println("中间件开始执行了")
        // 设置变量到Context的key中, 可以通过Get()取
        c.Set("request", "中间件")
        // 执行函数
        c.Next()
        // 中间件执行完后续的一些事情
        status := c.Writer.Status()
        fmt.Println("中间件执行完毕", status)
        t2 := time.Since(t)
        fmt.Println("time:", t2)
    }
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 注册中间件
    r.Use(MiddleWare())
    // {}为了代码规范
    {
        r.GET("/ce", func(c *gin.Context) {
            // 取值
            req, _ := c.Get("request")
            fmt.Println("request:", req)
        })
    }
}
```

Next()方法

```
// 页面接收
c.JSON(200, gin.H{"request": req})
})

}
r.Run()
}
```

输出结果:

```
中间件开始执行了
request: 中间件
中间件执行完毕 200
time: 998.6µs
[GIN] 2019/10/23 - 16:58:31 | 200 | 1.9987ms | ::1 | GET /ce
```



局部中间件

局部中间件

```
package main

import (
    "fmt"
    "time"

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

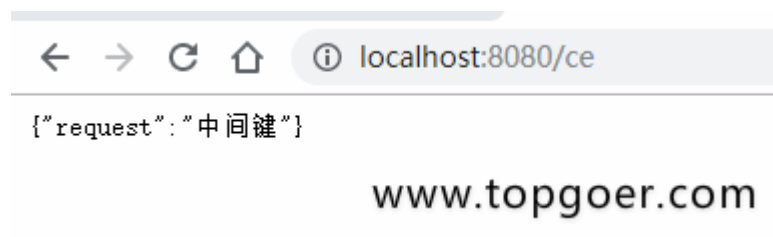
// 定义中间
func MiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()
        fmt.Println("中间件开始执行了")
        // 设置变量到Context的key中, 可以通过Get()取
        c.Set("request", "中间件")
        // 执行函数
        c.Next()
        // 中间件执行完后续的一些事情
        status := c.Writer.Status()
        fmt.Println("中间件执行完毕", status)
        t2 := time.Since(t)
        fmt.Println("time:", t2)
    }
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    //局部中间键使用
    r.GET("/ce", MiddleWare(), func(c *gin.Context) {
        // 取值
        req, _ := c.Get("request")
        fmt.Println("request:", req)
        // 页面接收
        c.JSON(200, gin.H{"request": req})
    })
}
```

```
r.Run()  
}
```

效果演示:

```
中间件开始执行了  
request: 中间件  
中间件执行完毕 200  
time: 998.6µs  
[GIN] 2019/10/23 - 16:58:31 |?|97;42m 200 ?[0m| 1.9987ms | ::1 |?|97;44m GET ?[0m /ce
```



中间件练习

中间件练习

- 定义程序计时中间件，然后定义2个路由，执行函数后应该打印统计的执行时间，如下：

```
package main

import (
    "fmt"
    "time"

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

// 定义中间
func myTime(c *gin.Context) {
    start := time.Now()
    c.Next()
    // 统计时间
    since := time.Since(start)
    fmt.Println("程序用时: ", since)
}

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 注册中间件
    r.Use(myTime)
    // {}为了代码规范
    shoppingGroup := r.Group("/shopping")
    {
        shoppingGroup.GET("/index", shopIndexHandler)
        shoppingGroup.GET("/home", shopHomeHandler)
    }
    r.Run(":8000")
}

func shopIndexHandler(c *gin.Context) {
    time.Sleep(5 * time.Second)
}
```

```
func shopHomeHandler(c *gin.Context) {  
    time.Sleep(3 * time.Second)  
}
```

效果演示:

```
[GIN-debug] Listening and serving HTTP on :8000  
程序用时: 5.0017493s  
[GIN] 2019/10/16 - 15:44:42 | 200 | 5.0017493s | 127.0.0.1 | GET | /shopping/index  
程序用时: 3.0004205s  
[GIN] 2019/10/16 - 15:44:45 | 200 | 3.0004205s | 127.0.0.1 | GET | www.topgoer.com
```

中间件推荐

列表

谷歌翻译欢迎查看原文 <https://github.com/gin-gonic/contrib/blob/master/README.md>

- [RestGate](#) - REST API端点的安全身份验证
- [staticbin](#) - 用于从二进制数据提供静态文件的中间件/处理程序
- [gin-cors](#) - CORS杜松子酒的官方中间件
- [gin-csrf](#) - CSRF保护
- [gin-health](#) - 通过[gocraft/health](#)报告的中间件
- [gin-merry](#) - 带有上下文的漂亮 打印 错误的中间件
- [gin-revision](#) - 用于Gin框架的修订中间件
- [gin-jwt](#) - 用于Gin框架的JWT中间件
- [gin-sessions](#) - 基于mongodb和mysql的会话中间件
- [gin-location](#) - 用于公开服务器的主机名和方案的中间件
- [gin-nice-recovery](#) - 紧急恢复中间件，可让您构建更好的用户体验
- [gin-limit](#) - 限制同时请求；可以帮助增加交通流量
- [gin-limit-by-key](#) - 一种内存中的中间件，用于通过自定义键和速率限制访问速率。
- [ez-gin-template](#) - gin简单模板包装
- [gin-hydra](#) - gin中间件Hydra
- [gin-glog](#) - 旨在替代Gin的默认日志
- [gin-gomonitor](#) - 用于通过Go-Monitor公开指标
- [gin-oauth2](#) - 用于OAuth2
- [static](#) gin框架的替代静态资产处理程序。
- [xss-mw](#) - XssMw是一种中间件，旨在从用户提交的输入中“自动删除XSS”
- [gin-helmet](#) - 简单的安全中间件集合。
- [gin-jwt-session](#) - 提供JWT / Session / Flash的中间件，易于使用，同时还提供必要的调整选项。也提供样品。
- [gin-template](#) - 用于gin框架的html / template易于使用。
- [gin-redis-ip-limiter](#) - 基于IP地址的请求限制器。它可以与redis和滑动窗口机制一起使用。

- [gin-method-override](#) - `_method`受Ruby的同名机架启发而被POST形式参数覆盖的方法
- [gin-access-limit](#) - `limit`-通过指定允许的源CIDR表示法的访问控制中间件。
- [gin-session](#) - 用于Gin的Session中间件
- [gin-stats](#) - 轻量级和有用的请求指标中间件
- [gin-statsd](#) - 向statsd守护进程报告的Gin中间件
- [gin-health-check](#) - `check`-用于Gin的健康检查中间件
- [gin-session-middleware](#) - 一个有效，安全且易于使用的Go Session库。
- [ginception](#) - 漂亮的例外页面
- [gin-inspector](#) - 用于调查http请求的Gin中间件。
- [gin-dump](#) - Gin中间件/处理程序，用于转储请求和响应的标头/正文。对调试应用程序非常有帮助。
- [go-gin-prometheus](#) - Gin Prometheus metrics exporter
- [ginprom](#) - Gin的Prometheus指标导出器
- [gin-go-metrics](#) - Gin middleware to gather and store metrics using [rcrowley/go-metrics](#)
- [ginrpc](#) - Gin 中间件/处理器自动绑定工具。通过像beego这样的注释路线来支持对象注册

会话控制

[Cookie介绍](#)

[Cookie的使用](#)

[Cookie练习](#)

[Cookie的缺点](#)

[Sessions](#)

Cookie介绍

Cookie介绍

- 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两次请求是否由同一个客户端发出
- Cookie就是解决HTTP协议无状态的方案之一，中文是小甜饼的意思
- 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
- Cookie由服务器创建，并发送给浏览器，最终由浏览器保存

Cookie的用途

- 测试服务端发送cookie给客户端，客户端请求时携带cookie

Cookie的使用

Cookie的使用

- 测试服务端发送cookie给客户端，客户端请求时携带cookie

```
package main

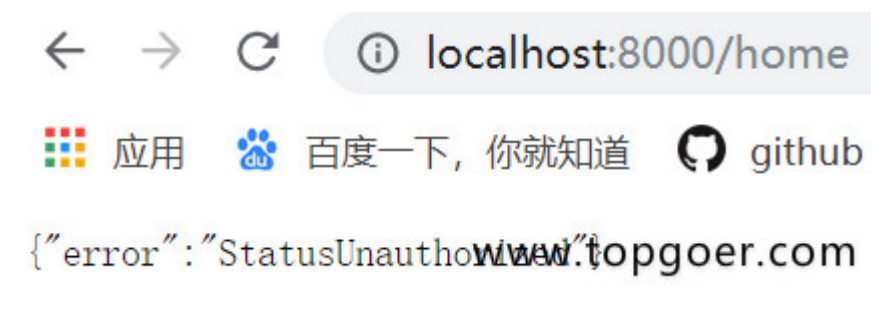
import (
    "github.com/gin-gonic/gin"
    "fmt"
)

func main() {
    // 1. 创建路由
    // 默认使用了2个中间件Logger(), Recovery()
    r := gin.Default()
    // 服务端要给客户端cookie
    r.GET("cookie", func(c *gin.Context) {
        // 获取客户端是否携带cookie
        cookie, err := c.Cookie("key_cookie")
        if err != nil {
            cookie = "NotSet"
            // 给客户端设置cookie
            // maxAge int, 单位为秒
            // path, cookie所在目录
            // domain string, 域名
            // secure 是否智能通过https访问
            // httpOnly bool 是否允许别人通过js获取自己的cookie
            c.SetCookie("key_cookie", "value_cookie", 60, "/",
                "localhost", false, true)
        }
        fmt.Printf("cookie的值是: %s\n", cookie)
    })
    r.Run(":8000")
}
```

Cookie练习

Cookie练习

- 模拟实现权限验证中间件
 - 有2个路由，login和home
 - login用于设置cookie
 - home是访问查看信息的请求
 - 在请求home之前，先跑中间件代码，检验是否存在cookie
- 访问home，会显示错误，因为权限校验未通过



- 然后访问登录的请求，登录并设置cookie



- 再次访问home，访问成功



```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func AuthMiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        // 获取客户端cookie并校验
        if cookie, err := c.Cookie("abc"); err == nil {
            if cookie == "123" {
                c.Next()
                return
            }
        }
        // 返回错误
        c.JSON(http.StatusUnauthorized, gin.H{"error": "err"})
        // 若验证不通过, 不再调用后续的函数处理
        c.Abort()
        return
    }
}

func main() {
    // 1. 创建路由
    r := gin.Default()
    r.GET("/login", func(c *gin.Context) {
        // 设置cookie
        c.SetCookie("abc", "123", 60, "/",
            "localhost", false, true)
        // 返回信息
        c.String(200, "Login success!")
    })
    r.GET("/home", AuthMiddleWare(), func(c *gin.Context) {
        c.JSON(200, gin.H{"data": "home"})
    })
    r.Run(":8000")
}
```

访问 /home 和 /login进行测试

Cookie的缺点

Cookie的缺点

- 不安全，明文
- 增加带宽消耗
- 可以被禁用
- cookie有上限

Sessions

Sessions

gorilla/sessions为自定义session后端提供cookie和文件系统session以及基础结构。

主要功能是：

- 简单的API：将其用作设置签名（以及可选的加密）cookie的简便方法。
- 内置的后端可将session存储在cookie或文件系统中。
- Flash消息：一直持续读取的session值。
- 切换session持久性（又称“记住我”）和设置其他属性的便捷方法。
- 旋转身份验证和加密密钥的机制。
- 每个请求有多个session，即使使用不同的后端也是如此。
- 自定义session后端的接口和基础结构：可以使用通用API检索并批量保存来自不同商店的session。

代码：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/sessions"
)

// 初始化一个cookie存储对象
// something-very-secret应该是一个你自己的密匙，只要不被别人知道就行
var store = sessions.NewCookieStore([]byte("something-very-secret"))

func main() {
    http.HandleFunc("/save", SaveSession)
    http.HandleFunc("/get", GetSession)
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        fmt.Println("HTTP server failed, err:", err)
    }
    return
}
```

```

    }
}

func SaveSession(w http.ResponseWriter, r *http.Request) {
    // Get a session. We're ignoring the error resulted from decoding an
    // existing session: Get() always returns a session, even if empty.

    // 获取一个session对象, session-name是session的名字
    session, err := store.Get(r, "session-name")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // 在session中存储值
    session.Values["foo"] = "bar"
    session.Values[42] = 43
    // 保存更改
    session.Save(r, w)
}

func GetSession(w http.ResponseWriter, r *http.Request) {
    session, err := store.Get(r, "session-name")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    foo := session.Values["foo"]
    fmt.Println(foo)
}

```

删除session的值:

```

// 删除
// 将session的最大存储时间设置为小于零的数即为删除
session.Options.MaxAge = -1
session.Save(r, w)

```

官网地址: <http://www.gorillatoolkit.org/pkg/sessions>

参数验证

结构体验证

自定义验证

自定义验证v10

多语言翻译验证

结构体验证

结构体验证

用gin框架的数据验证，可以不用解析数据，减少if else，会简洁许多。

```
package main

import (
    "fmt"
    "time"

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

//Person ..
type Person struct {
    //不能为空并且大于10
    Age      int      `form:"age" binding:"required,gt=10"`
    Name     string   `form:"name" binding:"required"`
    Birthday time.Time `form:"birthday" time_format:"2006-01-02" time_utc:"1"`
}

func main() {
    r := gin.Default()
    r.GET("/51mh", func(c *gin.Context) {
        var person Person
        if err := c.ShouldBind(&person); err != nil {
            c.String(500, fmt.Sprintf(err))
            return
        }
        c.String(200, fmt.Sprintf("%#v", person))
    })
    r.Run()
}
```

演示地址:

<http://localhost:8080/51mh?age=11&name=枯藤&birthday=2006-01-02>

自定义验证

自定义验证

都在代码里自己看吧

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gin-gonic/gin"
    "gopkg.in/go-playground/validator.v9"
)

/*
    对绑定解析到结构体上的参数，自定义验证功能
    比如我们需要对URL的接受参数进行判断，判断用户名是否为root如果是root通过否则
    返回false
*/
type Login struct {
    User      string `uri:"user" validate:"checkName"`
    Pssword  string `uri:"password"`
}

// 自定义验证函数
func checkName(fl validator.FieldLevel) bool {
    if fl.Field().String() != "root" {
        return false
    }
    return true
}

func main() {
    r := gin.Default()
    validate := validator.New()
    r.GET("/:user/:password", func(c *gin.Context) {
        var login Login
        //注册自定义函数，与struct tag关联起来
        err := validate.RegisterValidation("checkName", checkName)
        if err := c.ShouldBindUri(&login); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        }
    })
}
```

```

        return
    }
    err = validate.Struct(login)
    if err != nil {
        for _, err := range err.(validator.ValidationErrors) {
            fmt.Println(err)
        }
        return
    }
    fmt.Println("success")
})
r.Run()
}

```

示例2:

```

package main

import (
    "net/http"
    "reflect"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/binding"
    "gopkg.in/go-playground/validator.v8"
)

// Booking contains binded and validated data.
type Booking struct {
    //定义一个预约的时间大于今天的时间
    CheckIn time.Time `form:"check_in" binding:"required,bookabledate" time_format:"2006-01-02"`
    //gtfield=CheckIn退出的时间大于预约的时间
    CheckOut time.Time `form:"check_out" binding:"required,gtfield=CheckIn" time_format:"2006-01-02"`
}

func bookableDate(
    v *validator.Validate, topStruct reflect.Value, currentStructOrField reflect.Value,
    field reflect.Value, fieldType reflect.Type, fieldKind reflect.Kind, param string
) bool {
    //field.Interface().(time.Time)获取参数值并且转换为时间格式

```

```
    if date, ok := field.Interface().(time.Time); ok {
        today := time.Now()
        if today.Unix() > date.Unix() {
            return false
        }
    }
    return true
}

func main() {
    route := gin.Default()
    //注册验证
    if v, ok := binding.Validator.Engine().(*validator.Validate); ok {
        //绑定第一个参数是验证的函数第二个参数是自定义的验证函数
        v.RegisterValidation("bookabledate", bookableDate)
    }

    route.GET("/51mh", getBookable)
    route.Run()
}

func getBookable(c *gin.Context) {
    var b Booking
    if err := c.ShouldBindWith(&b, binding.Query); err == nil {
        c.JSON(http.StatusOK, gin.H{"message": "Booking dates are valid!"})
    } else {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    }
}

// curl -X GET "http://localhost:8080/51mh?check_in=2019-11-07&check_out=2019-11-20"
// curl -X GET "http://localhost:8080/51mh?check_in=2019-09-07&check_out=2019-11-20"
// curl -X GET "http://localhost:8080/51mh?check_in=2019-11-07&check_out=2019-11-01"
```

自定义验证v10

介绍

Validator 是基于 tag（标记）实现结构体和单个字段的值验证库，它包含以下功能：

- 使用验证 tag（标记）或自定义验证器进行跨字段和跨结构体验证。
- 关于 slice、数组和 map，允许验证多维字段的任何或所有级别。
- 能够深入 map 键和值进行验证。
- 通过在验证之前确定接口的基础类型来处理类型接口。
- 处理自定义字段类型（如 sql 驱动程序 Valuer）。
- 别名验证标记，它允许将多个验证映射到单个标记，以便更轻松地定义结构体上的验证。
- 提取自定义的字段名称，例如，可以指定在验证时提取 JSON 名称，并在生成的 FieldError 中使用该名称。
- 可自定义 i18n 错误消息。
- Web 框架 gin 的默认验证器。

安装：

使用 go get:

```
go get github.com/go-playground/validator/v10
```

然后将 Validator 包导入到代码中：

```
import "github.com/go-playground/validator/v10"
```

变量验证

Var 方法使用 tag（标记）验证方式验证单个变量。

```
func (*validator.Validate).Var(field interface{}, tag string) error
```

它接收一个 interface{} 空接口类型的 field 和一个 string 类型的 tag，返回传递的非法值得无效验证错误，否则将 nil 或 ValidationErrors 作为错误。如果错误不是 nil，则需要断言错误去访问错误数组，例如：

```
validationErrors := err.(validator.ValidationErrors)
```

如果是验证数组、`slice` 和 `map`，可能会包含多个错误。

示例代码：

```
func main() {
    validate := validator.New()
    // 验证变量
    email := "admin@admin.com"
    email := ""
    err := validate.Var(email, "required,email")
    if err != nil {
        validationErrors := err.(validator.ValidationErrors)
        fmt.Println(validationErrors)
        // output: Key: '' Error:Field validation for '' failed on the 'email' tag
        // output: Key: '' Error:Field validation for '' failed on the 'required' tag
    }
    return
}
```

结构体验证

结构体验证结构体公开的字段，并自动验证嵌套结构体，除非另有说明。

```
func (*validator.Validate).Struct(s interface{}) error
```

它接收一个 `interface{}` 空接口类型的 `s`，返回传递的非法值得无效验证错误，否则将 `nil` 或 `ValidationErrors` 作为错误。如果错误不是 `nil`，则需要断言错误去访问错误数组，例如：

```
validationErrors := err.(validator.ValidationErrors)
```

实际上，`Struct` 方法是调用的 `StructCtx` 方法，因为本文不是源码讲解，所以此处不展开赘述，如有兴趣，可以查看源码。

示例代码：

```
func main() {
    validate = validator.New()
    type User struct {
        ID int64 `json:"id" validate:"gt=0"`
    }
}
```

```

    Name string `json:"name" validate:"required"`
    Gender string `json:"gender" validate:"required,oneof=man woman"`
    Age uint8 `json:"age" validate:"required,gte=0,lte=130"`
    Email string `json:"email" validate:"required,email"`
}

user := &User{
    ID: 1,
    Name: "frank",
    Gender: "boy",
    Age: 180,
    Email: "gopher@88.com",
}

err = validate.Struct(user)
if err != nil {
    validationErrors := err.(validator.ValidationErrors)
    // output: Key: 'User.Age' Error:Field validation for 'Age' failed on the 'lte' tag
    // fmt.Println(validationErrors)
    fmt.Println(validationErrors.Translate(trans))
    return
}
}

```

细心的读者可能已经发现，错误输出信息并不友好，错误输出信息中的字段不仅没有使用备用名（首字母小写的字段名），也没有翻译为中文。通过改动代码，使错误输出信息变得友好。

注册一个函数，获取结构体字段的备用名称：

```

validate.RegisterTagNameFunc(func(fld reflect.StructField) string {
    name := strings.SplitN(fld.Tag.Get("json"), ",", 2)[0]
    if name == "-" {
        return "j"
    }
    return name
})

```

错误信息翻译为中文：

```

zh := zh.New()
uni = ut.New(zh)
trans, _ := uni.GetTranslator("zh")
_ = zh_translations.RegisterDefaultTranslations(validate, trans)

```

标签

通过以上章节的内容，读者应该已经了解到 Validator 是一个基于 tag（标签），实现结构体和单个字段的值验证库。

本章节列举一些比较常用的标签：

标签	描述
eq	等于
gt	大于
gte	大于等于
lt	小于
lte	小于等于
ne	不等于
max	最大值
min	最小值
oneof	其中一个
required	必需的
unique	唯一的
isDefault	默认值
len	长度
email	邮箱格式

转自：Golang语言开发栈

多语言翻译验证

多语言翻译验证

当业务系统对验证信息有特殊需求时，例如：返回信息需要自定义，手机端返回的信息需要是中文而pc端发挥返回的信息需要时英文，如何做到请求一个接口满足上述三种情况。

```
package main

import (
    "fmt"

    "github.com/gin-gonic/gin"
    "github.com/go-playground/locales/en"
    "github.com/go-playground/locales/zh"
    "github.com/go-playground/locales/zh_Hant_TW"
    ut "github.com/go-playground/universal-translator"
    "gopkg.in/go-playground/validator.v9"
    en_translations "gopkg.in/go-playground/validator.v9/translations/en"
    zh_translations "gopkg.in/go-playground/validator.v9/translations/zh"
    zh_tw_translations "gopkg.in/go-playground/validator.v9/translations/zh_tw"
)

var (
    Uni      *ut.UniversalTranslator
    Validate *validator.Validate
)

type User struct {
    Username string `form:"user_name" validate:"required"`
    Tagline  string `form:"tag_line" validate:"required,lt=10"`
    Tagline2 string `form:"tag_line2" validate:"required,gt=1"`
}

func main() {
    en := en.New()
    zh := zh.New()
    zh_tw := zh_Hant_TW.New()
    Uni = ut.New(en, zh, zh_tw)
    Validate = validator.New()

    route := gin.Default()
```

```

route.GET("/51mh", startPage)
route.POST("/51mh", startPage)
route.Run(":8080")
}

func startPage(c *gin.Context) {
    //这部分应放到中间件中
    locale := c.DefaultQuery("locale", "zh")
    trans, _ := Uni.GetTranslator(locale)
    switch locale {
    case "zh":
        zh_translations.RegisterDefaultTranslations(Validate, trans)
        break
    case "en":
        en_translations.RegisterDefaultTranslations(Validate, trans)
        break
    case "zh_tw":
        zh_tw_translations.RegisterDefaultTranslations(Validate, trans)
        break
    default:
        zh_translations.RegisterDefaultTranslations(Validate, trans)
        break
    }

    //自定义错误内容
    Validate.RegisterTranslation("required", trans, func(ut ut.Translator) error {
        return ut.Add("required", "{0} must have a value!", true) // see univers
al-translator for details
    }, func(ut ut.Translator, fe validator.FieldError) string {
        t, _ := ut.T("required", fe.Field())
        return t
    })

    //这块应该放到公共验证方法中
    user := User{}
    c.ShouldBind(&user)
    fmt.Println(user)
    err := Validate.Struct(user)
    if err != nil {
        errs := err.(validator.ValidationErrors)
        sliceErrs := []string{}
        for _, e := range errs {
            sliceErrs = append(sliceErrs, e.Translate(trans))
        }
        c.String(200, fmt.Sprintf("%#v", sliceErrs))
    }
}

```

```
}  
c.String(200, fmt.Sprintf("#v", "user"))  
}
```

正确的链接: [http://localhost:8080/testing?user_name=枯藤](http://localhost:8080/testing?user_name=枯藤&tag_line=9&tag_line2=33&locale=zh)

[&tag_line=9&tag_line2=33&locale=zh](http://localhost:8080/testing?user_name=枯藤&tag_line=9&tag_line2=33&locale=zh)

http://localhost:8080/testing?user_name=枯藤&tag_line=9&tag_line2=3&locale=en

返回英文的验证信息

http://localhost:8080/testing?user_name=枯藤&tag_line=9&tag_line2=3&locale=zh

返回中文的验证信息

查看更多的功能可以查看官网 gopkg.in/go-playground/validator.v9

其他

其他

日志文件

[Air实时加载](#)

日志文件

日志文件

```
package main

import (
    "io"
    "os"

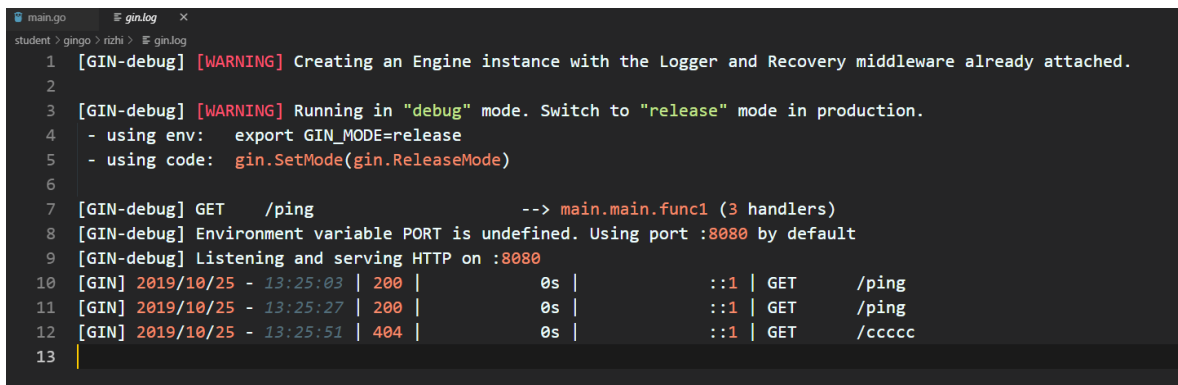
    "github.com/gin-gonic/gin"
)

func main() {
    gin.DisableConsoleColor()

    // Logging to a file.
    f, _ := os.Create("gin.log")
    gin.DefaultWriter = io.MultiWriter(f)

    // 如果需要同时将日志写入文件和控制台，请使用以下代码。
    // gin.DefaultWriter = io.MultiWriter(f, os.Stdout)
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })
    r.Run()
}
```

效果演示:



```
main.go gin.log x
student > gingo > rizi > gin.log
1 [GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.
2
3 [GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
4 - using env:   export GIN_MODE=release
5 - using code: gin.SetMode(gin.ReleaseMode)
6
7 [GIN-debug] GET    /ping          --> main.main.func1 (3 handlers)
8 [GIN-debug] Environment variable PORT is undefined. Using port :8080 by default
9 [GIN-debug] Listening and serving HTTP on :8080
10 [GIN] 2019/10/25 - 13:25:03 | 200 |      0s |      ::1 | GET    /ping
11 [GIN] 2019/10/25 - 13:25:27 | 200 |      0s |      ::1 | GET    /ping
12 [GIN] 2019/10/25 - 13:25:51 | 404 |      0s |      ::1 | GET    /cccc
13
```

Air实时加载

Air实时加载

本章我们要介绍一个神器——Air能够实时监听项目的代码文件，在代码发生变更之后自动重新编译并执行，大大提高gin框架项目的开发效率。

为什么需要实时加载？

之前使用Python编写Web项目的时候，常见的Flask或Django框架都是支持实时加载的，你修改了项目代码之后，程序能够自动重新加载并执行（live-reload），这在日常的开发阶段是十分方便的。

在使用Go语言的gin框架在本地做开发调试的时候，经常需要在变更代码之后频繁的按下Ctrl+C停止程序并重新编译再执行，这样就不是很方便。

Air介绍

怎样才能基于gin框架开发时实现实时加载功能呢？像这种烦恼肯定不会只是你一个人的烦恼，所以我报着肯定有现成轮子的心态开始了全网大搜索。果不其然就在Github上找到了一个工具：Air[1]。它支持以下特性：

- 彩色日志输出
- 自定义构建或二进制命令
- 支持忽略子目录
- 启动后支持监听新目录
- 更好的构建过程

安装Air

Go

这也是最经典的安装方式：

```
go get -u github.com/cosmtrek/air
```

MacOS

```
curl -fLo air https://git.io/darwin_air
```

Linux

```
curl -fLo air https://git.io/linux_air
```

Windows

```
curl -fLo air.exe https://git.io/windows_air
```

Dcoker

```
docker run -it --rm \  
-w "<PROJECT>" \  
-e "air_wd=<PROJECT>" \  
-v $(pwd):<PROJECT> \  
-p <PORT>:<APP SERVER PORT> \  
cosmtrek/air \  
-c <CONF>
```

然后按照下面的方式在docker中运行你的项目：

```
docker run -it --rm \  
-w "/go/src/github.com/cosmtrek/hub" \  
-v $(pwd):/go/src/github.com/cosmtrek/hub \  
-p 9090:9090 \  
cosmtrek/air
```

使用Air

为了敲命令更简单更方便，你应该把 `alias air='~/air'` 加到你的 `.bashrc` 或 `.zshrc` 中。

首先进入你的项目目录：

```
cd /path/to/your_project
```

最简单的用法就是直接执行下面的命令：


```
# 首先在当前目录下查找`.air.conf`配置文件，如果找不到就使用默认的
air -c .air.conf
```

推荐的使用方法是：

```
# 1. 在当前目录创建一个新的配置文件.air.conf
touch .air.conf

# 2. 复制`.air.conf.example`中的内容到这个文件，然后根据你的需要去修改它

# 3. 使用你的配置运行air，如果文件名是`.air.conf`，只需要执行`air`。
air
```

air_example.conf示例

完整的air_example.conf示例配置如下，可以根据自己的需要修改。

```
# [Air] (https://github.com/cosmtrek/air) TOML 格式的配置文件

# 工作目录
# 使用. 或绝对路径，请注意`tmp_dir`目录必须在`root`目录下
root = "."
tmp_dir = "tmp"

[build]
# 只需要写你平常编译使用的shell命令。你也可以使用`make`
cmd = "go build -o ./tmp/main ."
# 由`cmd`命令得到的二进制文件名
bin = "tmp/main"
# 自定义的二进制，可以添加额外的编译标识例如添加 GIN_MODE=release
full_bin = "APP_ENV=dev APP_USER=air ./tmp/main"
# 监听以下文件扩展名的文件。
include_ext = ["go", "tpl", "tmpl", "html"]
# 忽略这些文件扩展名或目录
exclude_dir = ["assets", "tmp", "vendor", "frontend/node_modules"]
# 监听以下指定目录的文件
include_dir = []
# 排除以下文件
exclude_file = []
# 如果文件更改过于频繁，则没有必要在每次更改时都触发构建。可以设置触发构建的延迟时间
delay = 1000 # ms
# 发生构建错误时，停止运行旧的二进制文件。
stop_on_error = true
```

```
# air的日志文件名, 该日志文件放在你的`tmp_dir`中
log = "air_errors.log"

[log]
# 显示日志时间
time = true

[color]
# 自定义每个部分显示的颜色。如果找不到颜色, 使用原始的应用程序日志。
main = "magenta"
watcher = "cyan"
build = "yellow"
runner = "green"

[misc]
# 退出时删除tmp目录
clean_on_exit = true
```

本文转自: <https://mp.weixin.qq.com/s/QgL3jx8Au7YbRjlfslFgJg>