一、Gin中渲染模板

1.1、常规用法

```
func main() {
    r := gin.Default()
    r.LoadHTMLGlob("templates/**/*")
    //r.LoadHTMLFiles("templates/posts/index.html",
"templates/users/index.html")
    r.GET("/posts/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "posts/index.html", gin.H{
            "title": "posts/index",
        })
    })

    r.GET("users/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/index.html", gin.H{
            "title": "users/index",
        })
    })

    r.Run(":8080")
}
```

1.2、自定义模板函数

```
func main() {
    // 默认路由
    r := gin.Default()

    // gin框架自定义模板函数
    r.SetFuncMap(template.FuncMap{
        // 自定义函数名 safe
        // 匿名函数 func (str string) template.HTML ---> 自定义模板函数实现的功能
        "safe": func(str string) template.HTML {
            return template.HTML(str)
        },
    })

    // gin框架模板解析,Gin框架中使用LoadHTMLGlob()或者LoadHTMLFiles()方法进行HTML模板渲
染。
    r.LoadHTMLFiles("template/index.html")

    r.GET("/index", func(c *gin.Context) {
        // HTTP请求
        c.HTML(http.StatusOK, "index.html", gin.H{
            "title": "<a href='https://baidu.com'>tzh666</a>",
        })
    })

    r.Run(":9090")
}
```

```
<body>
    {{ .title | safe}}
</body>
```

1.3、静态文件处理

```
func main() {
    r := gin.Default()
    r.Static("/static", "./static")
    r.LoadHTMLGlob("templates/**/*")
   // ...
    r.Run(":8080")
}
```

二、JSON、XML数据渲染

2.1、JSON数据渲染

```
func main() {

    r := gin.Default()

    // gin.H 是map[string]interface{}的缩写
    r.GET("/json1", func(c *gin.Context) {
        // 方式一,自己拼接JSON格式数据
        data := map[string]interface{}{
            "name": "tzh",
            "age":  18,
        }
        c.JSON(http.StatusOK, data)
    })
    r.GET("/json2", func(c *gin.Context) {
        // 方式一,自己拼接JSON格式数据
        c.JSON(http.StatusOK, gin.H{
            "name": "tzh1",
            "age":  18,
        })
    })
    // 方式二：使用结构体,首字母大写才能传到前端去,但是可以用 `` 做灵活定制化操作,这样返回给前端的就是name  password。
    type msg struct {
        Name     string `json:"name" xml:"name"`
        Password string `json:"passwoed" xml:"password"`
    }
    r.GET("/json3", func(c *gin.Context) {
        // 方式一,自己拼接JSON格式数据
        data := msg{
            "tzh",
            "tzh",
```

```
        }
        // JSON序列号
        c.JSON(http.StatusOK, data)
    })

    r.Run(":9090")
}
```

2.2、XML数据渲染

注意需要使用具名的结构体类型

```
func main() {
    r := gin.Default()
    // gin.H 是map[string]interface{}的缩写
    r.GET("/someXML", func(c *gin.Context) {
        // 方式一：自己拼接JSON
        c.XML(http.StatusOK, gin.H{"message": "Hello world!"})
    })
    r.GET("/moreXML", func(c *gin.Context) {
        // 方法二：使用结构体
        type MessageRecord struct {
            Name    string
            Message string
            Age     int
        }
        var msg MessageRecord
        msg.Name = "小王子"
        msg.Message = "Hello world!"
        msg.Age = 18
        c.XML(http.StatusOK, msg)
    })
    r.Run(":8080")
}
```

2.3、YAML数据渲染

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

2.4、protobuf渲染

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

三、Gin参数获取

3.1、获取querystring参数

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

```
func main() {
    r := gin.Default()

    // 获取浏览器的请求，获取ur发起请求携带的query string 参数
    r.GET("/web", func(c *gin.Context) {
        // 通过c.Query获取请求中携带的参数  <http://127.0.0.1:9090/web?query=tzh>
        // name := c.Query("query")

        // 取不到就用默认值  <http://127.0.0.1:9090/web?xxx>
        // name := c.DefaultQuery("query", "zhangsan")

        // GetQuery 有两个返回值,可做判断用 <http://127.0.0.1:9090/web?query=tzh> 取不到返回false
        name, ok := c.GetQuery("query")
        if !ok {
            fmt.Println("取值失败")
            return
        }
        // <http://127.0.0.1:9090/web?query=tzh&age=18>
        age, _ := c.GetQuery("query")
        c.JSON(http.StatusOK, gin.H{
            "name": name,
            "age":  age,
        })
    })

    r.Run(":9090")
}
```

3.2、获取form参数

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

```
func main() {
    //Default返回一个默认的路由引擎
    r := gin.Default()
    r.POST("/Login", func(c *gin.Context) {
        username := c.PostForm("username")
        address := c.PostForm("address")

        // DefaultPostForm取不到值时会返回指定的默认值
        username1 := c.DefaultPostForm("username", "森森")
        address1 := c.DefaultPostForm("address", "北京")

        username2, _ := c.GetPostForm("username")
        address2, _ := c.GetPostForm("address")
```

```
        //输出json结果给调用方
        c.JSON(http.StatusOK, gin.H{
            "message":   "ok",
            "Nsername":  username,
            "Address":   address,
            "Nsername1": username1,
            "Address1":  address1,
            "Nsername2": username2,
            "Address2":  address2,
        })
    })
    r.Run(":8080")
}
```

3.3、获取json参数

当前端请求的数据通过JSON提交时，例如向`/json`发送一个POST请求，则获取请求参数的方式如下：

```
r.POST("/json", func(c *gin.Context) {
    // 注意：下面为了举例子方便，暂时忽略了错误处理
    b, _ := c.GetRawData()  // 从c.Request.Body读取请求数据
    // 定义map或结构体
    var m map[string]interface{}
    // 反序列化
    _ = json.Unmarshal(b, &m)

    c.JSON(http.StatusOK, m)
})
```

3.4、获取path参数

```
func main() {
    //Default返回一个默认的路由引擎
    r := gin.Default()
    r.GET("/user/search/:username/:address", func(c *gin.Context) {
        username := c.Param("username")
        address := c.Param("address")
        //输出json结果给调用方
        c.JSON(http.StatusOK, gin.H{
            "message":  "ok",
            "username": username,
            "address":  address,
        })
    })

    r.Run(":8080")
}
```

3.5、参数绑定 （结构体类型数据）

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

```go
// Binding from JSON
type Login struct {
    User     string `form:"user" json:"user" binding:"required"`
    Password string `form:"password" json:"password" binding:"required"`
}

func main() {
    router := gin.Default()

    // 绑定JSON的示例 ({"user": "q1mi", "password": "123456"})
    router.POST("/loginJSON", func(c *gin.Context) {
        var login Login

        if err := c.ShouldBind(&login); err == nil {
            fmt.Printf("login info:%#v\n", login)
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        }
    })

    // 绑定form表单示例 (user=q1mi&password=123456)
    router.POST("/loginForm", func(c *gin.Context) {
        var login Login
        // ShouldBind()会根据请求的Content-Type自行选择绑定器
        if err := c.ShouldBind(&login); err == nil {
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        }
    })

    // 绑定QueryString示例 (/loginQuery?user=q1mi&password=123456)
    router.GET("/loginForm", func(c *gin.Context) {
        var login Login
        // ShouldBind()会根据请求的Content-Type自行选择绑定器
        if err := c.ShouldBind(&login); err == nil {
            c.JSON(http.StatusOK, gin.H{
                "user":     login.User,
                "password": login.Password,
            })
        } else {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        }
    })

    // Listen and serve on 0.0.0.0:8080
    router.Run(":8080")
}
```

`ShouldBind`会按照下面的顺序解析请求中的数据完成绑定：

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

四、Gin文件上传

4.1、单个文件上传

```
func main() {
    r := gin.Default()
    r.LoadHTMLFiles("views/index.html")
    r.GET("/uploadfile", func(c *gin.Context) {
        // 返回一个页面
        c.HTML(http.StatusOK, "index.html", gin.H{})
    })

    // 处理multipart forms提交文件时默认的内存限制是32 MiB
    // 可以通过下面的方式修改
    // r.MaxMultipartMemory = 8 << 20  // 8 MiB
    r.POST("/uploadfile", func(c *gin.Context) {
        // 从请求中读取文件，<input type="file" name="uf1" id="">
        file, err := c.FormFile("uf1")
        if err != nil {
            c.JSON(http.StatusInternalServerError, gin.H{
                "status": "error",
                "error":  err.Error(),
            })
        } else {
            // 将读取到的文件保存到本地服务器中
            log.Println(file.Filename)
            dst := fmt.Sprintf("./file/%s", file.Filename)
            c.SaveUploadedFile(file, dst)
            c.JSON(http.StatusOK, gin.H{
                "status": "ok",
            })
        }
    })

    r.Run(":9090")
}
```

```
<!DOCTYPE html>
<html lang="zh-CN">

<head>
    <meta charset="UTF-8">
    <title>文件上传</title>
</head>

<body>
    <form action="/uploadfile" method="post" enctype="multipart/form-data">
        <input type="file" name="uf1" id=""> <br>
        <input type="submit" value="点击上传">
    </form>
```

```
</body>

</html>
```

4.2、多个文件上传

```
func main() {
    router := gin.Default()
    // 处理multipart forms提交文件时默认的内存限制是32 MiB
    // 可以通过下面的方式修改
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // Multipart form
        form, _ := c.MultipartForm()
        files := form.File["file"]

        for index, file := range files {
            log.Println(file.Filename)
            dst := fmt.Sprintf("C:/tmp/%s_%d", file.Filename, index)
            // 上传文件到指定的目录
            c.SaveUploadedFile(file, dst)
        }
        c.JSON(http.StatusOK, gin.H{
            "message": fmt.Sprintf("%d files uploaded!", len(files)),
        })
    })
    router.Run()
}
```

五、重定向

5.1、HTTP重定向

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

5.2、路由重定向，使用`HandleContext`：

```
r.GET("/test", func(c *gin.Context) {
    // 指定重定向的URL
    c.Request.URL.Path = "/test2"
    r.HandleContext(c)
})
r.GET("/test2", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{"hello": "world"})
})
```

六、Gin路由

6.1、普通路由

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

此外，还有一个可以匹配所有请求方法的`Any`方法如下：

`Any`可以处理GET、POST、DELETE等请求，在func里面自己进行判断即可

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

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

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

6.2、路由组

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

访问变成：127.0.0.1:8080/user/index 127.0.0.1:8080/user/login

```
func main() {
    r := gin.Default()
    userGroup := r.Group("/user")
    {
        userGroup.GET("/index", func(c *gin.Context) {...})
        userGroup.GET("/login", func(c *gin.Context) {...})
        userGroup.POST("/login", func(c *gin.Context) {...})

    }
}
```

路由组也是支持嵌套的，例如：

```
shopGroup := r.Group("/shop")
    {
        shopGroup.GET("/index", func(c *gin.Context) {...})
        shopGroup.GET("/cart", func(c *gin.Context) {...})
        shopGroup.POST("/checkout", func(c *gin.Context) {...})
        // 嵌套路由组
        xx := shopGroup.Group("xx")
        xx.GET("/oo", func(c *gin.Context) {...})
    }
```

七、中间件

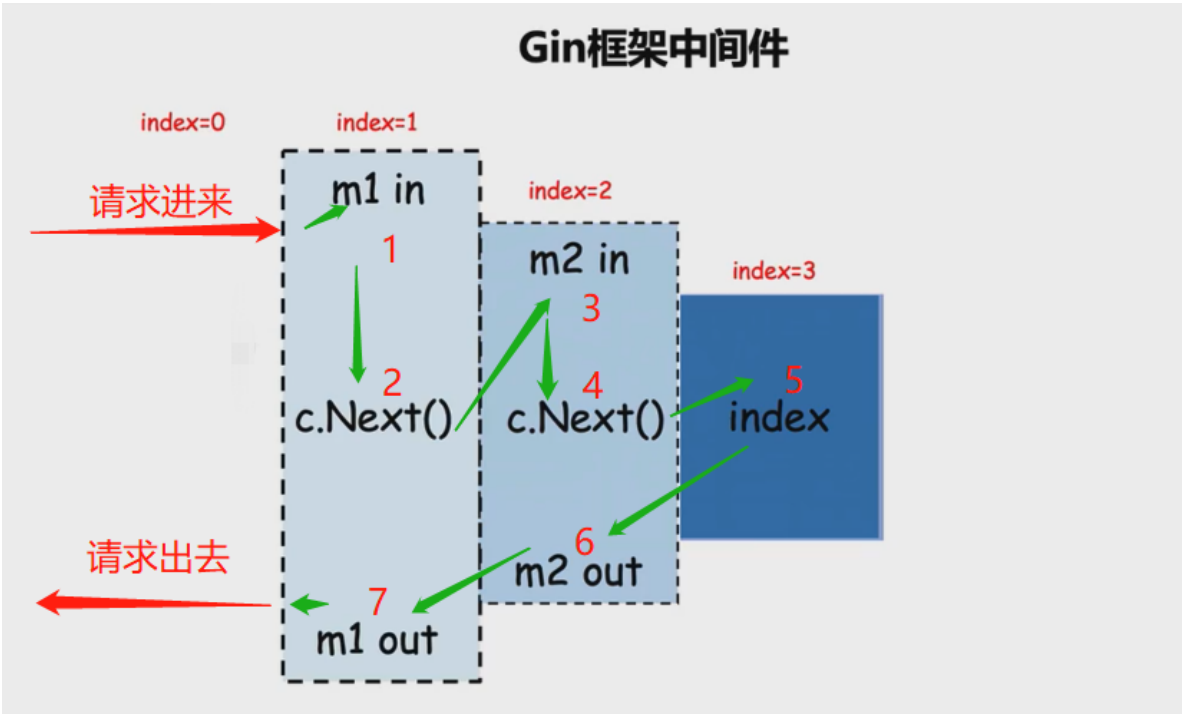
Gin框架允许开发者在处理请求的过程中，加入用户自己的钩子（Hook）函数。这个钩子函数就叫中间件，中间件适合处理一些公共的业务逻辑，比如**登录认证、权限校验、数据分页、记录日志、耗时统计**等。

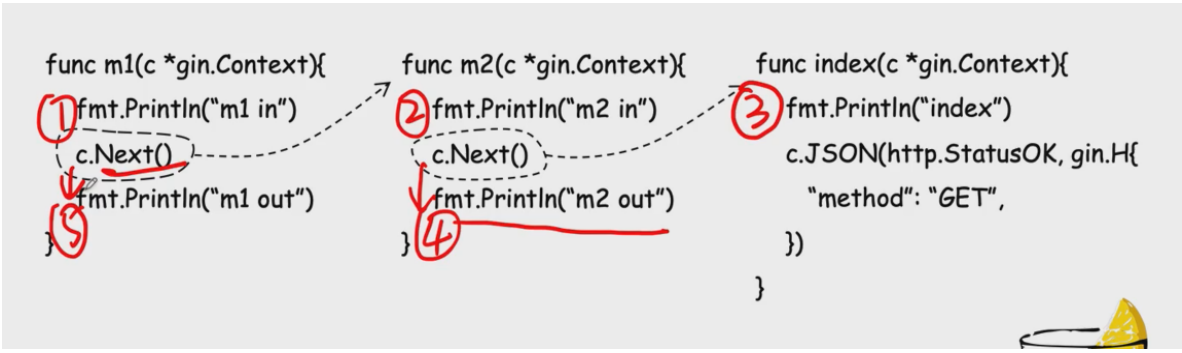
7.1、定义中间件

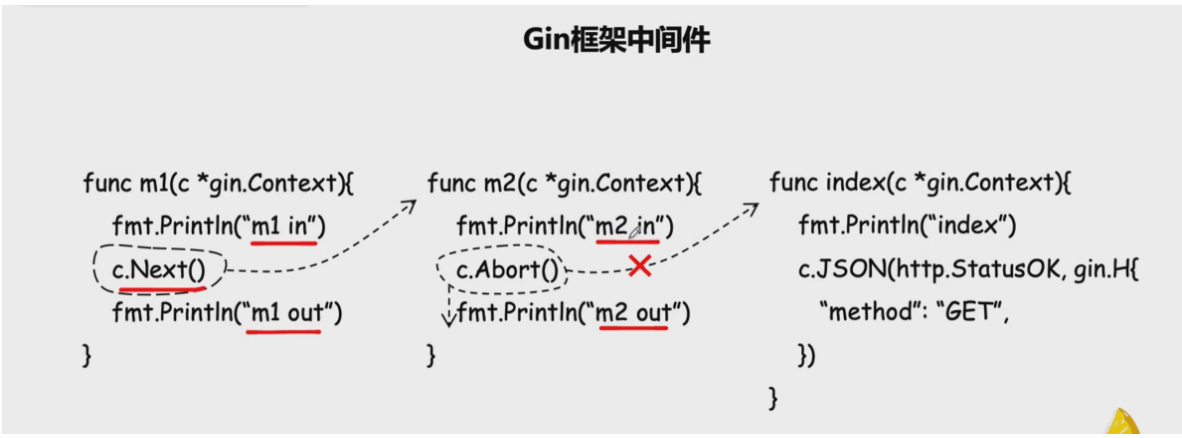
Gin中的中间件必须是一个`gin.HandlerFunc`类型。例如我们像下面的代码一样定义一个统计请求耗时的中间件。

```
// 平时写这个常用的也是一个 gin.HandlerFunc类型  <也是一个中间件>
func indexHandler(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "x": "1",
    })
}

// 自定义中间件,统计耗时
func m1(c *gin.Context) {
    start := time.Now()

    // 可以通过c.Set在请求上下文中设置值，后续的处理函数能够取到该值
    c.Set("name", "小王子")

    // 调用该请求的剩余处理程序
    c.Next()

    // 不调用该请求的剩余处理程序
    // c.Abort()
    // 计算耗时
    cost := time.Since(start)
    fmt.Println(cost)
}

func main() {
    r := gin.Default()

    // GET(relativePath string, handlers ...HandlerFunc)
    r.GET("/index", m1, indexHandler) // <会先执行m1>

    r.Run(":9090")
}
```

7.2、注册中间件 <一般使用闭包形式写中间件>







7.2.1、为全局路由注册

```go
func main() {
    //r := gin.Default()
    // 新建一个没有任何默认中间件的路由
    r := gin.New()
    // 注册一个全局中间件 m1
    r.Use(m1)

    // GET(relativePath string, handlers ...HandlerFunc)
    r.GET("/index", m1, indexHandler) // <会先执行m1>

    r.Run(":9090")
}
```

7.2.2、为某个路由单独注册

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

7.2.3、为路由组注册中间件

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


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

7.3、中间件注意事项

7.3.1、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。

7.3.2、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()`）

八、运行多个服务

我们可以在多个端口启动服务，例如：

```
package main

import (
    "log"
    "net/http"
    "time"

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

```
    "golang.org/x/sync/errgroup"
)

var (
    g errgroup.Group
)

func router01() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 01",
            },
        )
    })

    return e
}

func router02() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 02",
            },
        )
    })

    return e
}

func main() {
    server01 := &http.Server{
        Addr:         ":8080",
        Handler:      router01(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    server02 := &http.Server{
        Addr:         ":8081",
        Handler:      router02(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }
   // 借助errgroup.Group或者自行开启两个goroutine分别启动两个服务
    g.Go(func() error {
        return server01.ListenAndServe()
    })
```

```
    g.Go(func() error {
        return server02.ListenAndServe()
    })

    if err := g.Wait(); err != nil {
        log.Fatal(err)
    }
}
```