一、课前准备

> 📦开发环境
>
> - GoLand 2019.2 EAP
> - GoLang 1.10+
> - 采用 Go Modules 进行管理

二、课堂主题

> 说明：快速掌握Gin框架和Web编程

三、课堂目标

> 说明：完成本章课程案例

四、知识点（1小时40分钟）

Web 编程主要涉及这几个方面:

- web路由 (uri解析/ restful设计)
- 中间件 (业务和非业务分离)
- validator验证 (业务代码的验证和检验)
- db/orm/sql builder (结构体映射，序列化等)

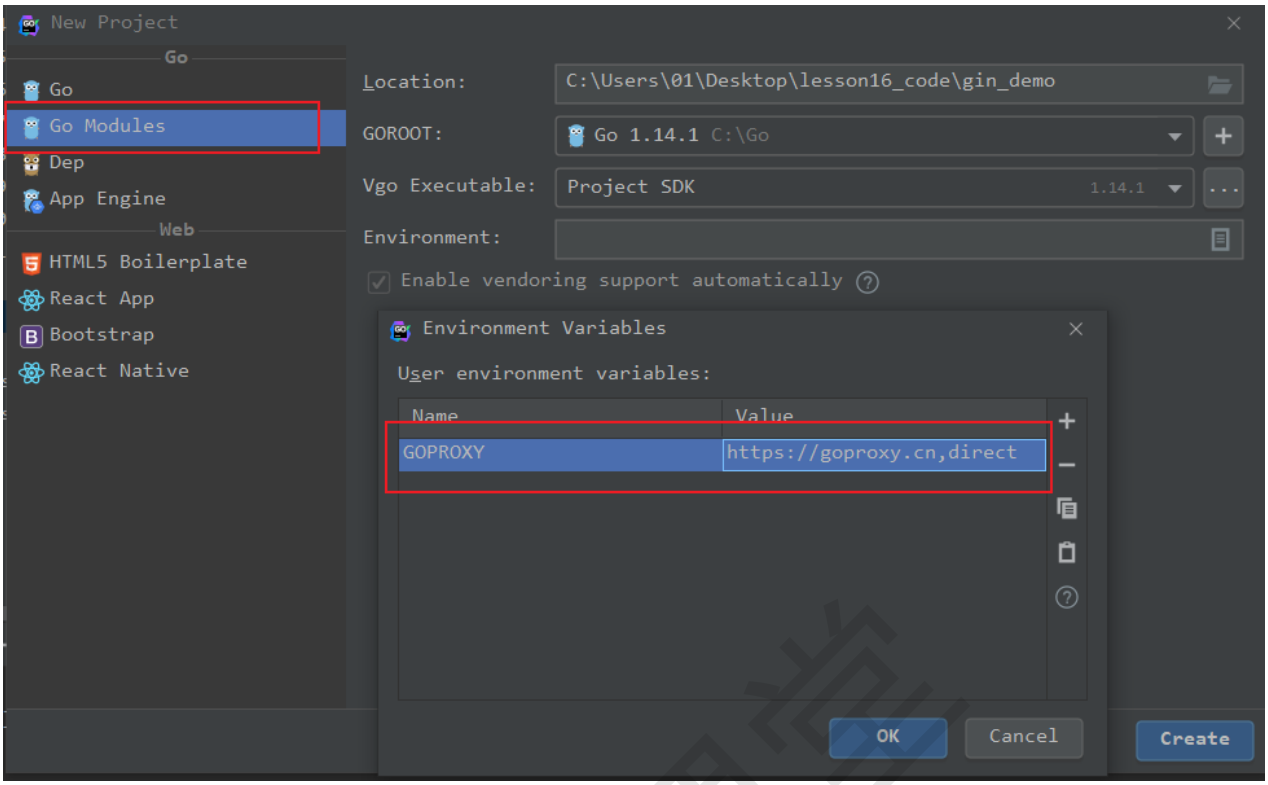
> Gin is a HTTP web framework written in Go (Golang). It features a Martini-like API with much better performance -- up to 40 times faster. If you need smashing performance, get yourself some Gin.

Gin是用Golang开发的一个微框架，类似Martini的API，重点是小巧、易用、性能好很多，也因为 httprouter 的性能提高了40倍。

1. gin初识

创建第一个gin项目

用 `GoLand` 新建项目的时候，我们选择 `Go Modules(vgo)` ，建议添加goproxy，填写我们的项目地址和项目名称，我们命名为 `GinHello` 。



点击 `Create` ，此时 `Goland` 为我们生成了项目目录， `Go` 项目的目录永远是那么的简单，比 `Java` 的 `Maven` 或者 `Gradle` 生成的项目目录简单多了。

```
GinHello
|
|-go.mod
```

项目目录包括一个 `Go module` 文件。 `go mod` 是 `Go` 官方引入的一个依赖管理工具。

添加依赖

通过 `go mod` 文件进行依赖的。

```
require github.com/gin-gonic/gin v1.4.0
```

我们把上面的依赖进行添加到 `go module` 中， `GoLand` 会自动帮我们进行依赖的下载和管理。

(自定义安装）使用go-get安装Gin

在命令行下执行安装

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

`go module` 模式下，下载的依赖信息都会保存在 `$GOPATH/pkg/mod/` 。

Hello Gin

当完成依赖的添加，就可以开始写代码了。

新建一个 `main.go` 文件。

```
package main
import (
 "github.com/gin-gonic/gin"
)
func main() {
 router := gin.Default()
 router.Run()
}
```

`Gin` 只需要两行代码就可以把我们的服务跑起来。

只要点击运行，项目便会启动一个 `8080` 端口，打开浏览器 `localhost:8080` 便可以看到页面上提示出 `404 page not found` ，这是因为根路由上并没有返回任何结果。同时可以在控制台上看到一些打印信息，其中就包括刚刚访问根路由的端口。

基本示例

项目已经启动了，那么如何返回一个接口呢?

通过 `router` 的 `Handle` 进行配置我们返回的参数。

```
package main
import (
 "github.com/gin-gonic/gin"
)
func main() {
 router := gin.Default()

 // 添加 Get 请求路由
 router.GET("/", func(context *gin.Context) {
 context.String(http.StatusOK, "hello gin")
 })

 router.Run()
}
```

同样，我们还可以进行 `POST`,`PUT`,`DELETE`等请求方式。

单元测试

单元测试是项目不能缺少的模块，也是保障项目可以正常运行的重要依赖。下面就对 Gin 进行单元测试。

为了方便单元测试，我们首先要对我们的项目进行一下抽取。

新建立一个文件夹叫做 `initRouter`

建立 `go` 文件 `initRouter.go`

```
package initRouter

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

func SetupRouter() *gin.Engine {
  router := gin.Default()
  // 添加 Get 请求路由
  router.GET("/", func(context *gin.Context) {
    context.String(http.StatusOK, "hello gin")
  })
  return router
}
```

同时修改 `main.go`

```
package main

import (
  "GinHello/initRouter"
)

func main() {
  router := initRouter.SetupRouter()
  _ = router.Run()
}
```

就完成了项目测试的初步建立。

建立 `test` 目录，`golang` 的单元测试都是以 `_test` 结尾，建立 `index_test.go` 文件。

```
package test

import (
  "GinHello/initRouter"
  "github.com/stretchr/testify/assert"
  "net/http"
  "net/http/httptest"
  "testing"
)

func TestIndexGetRouter(t *testing.T) {
  router := initRouter.SetupRouter()
  w := httptest.NewRecorder()
```

```
    req, _ := http.NewRequest(http.MethodGet, "/", nil)
    router.ServeHTTP(w, req)
    assert.Equal(t, http.StatusOK, w.Code)
    assert.Equal(t, "hello gin", w.Body.String())
}
```

通过 `assert` 进行断言，来判断返回状态码和返回值是否与代码中的值一致。

此时的项目目录为：

```
GinHello
|
|-initRouter
|   |-initRouter.go
|
|-test
|   |-index_test.go
|
|-main.go
|-go.mod
|-go.sum
```

运行单元测试，控制台打印出单元测试结果。

> — PASS: TestIndexGetRouter (0.05s) PASS

检查点

> 说明：
>
> 通过简单的搭建一个 Gin 项目，可以看到 Go/Gin 语言搭建一个 http 服务器很简单，也很方便，零配置即可完成项目并运行起来。
>
> 在这个项目中项目初始化和单元测试是很多大型项目的必须项。

2. Gin Router

服务器

默认服务器

默认路由直接使用run启动即可

```
router.Run()
```

http服务器

除了默认服务器中 `router.Run()` 的方式外，还可以用 `http.ListenAndServe()`，比如

```go
func main() {
    router := gin.Default()
    http.ListenAndServe(":8080", router)
}
```

或者自定义HTTP服务器的配置：

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

    s := &http.Server{
        Addr:           ":8080",
        Handler:        router,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    s.ListenAndServe()
}
```

路由

基本路由

基本路由 gin 框架中采用的路由库是 httprouter。

```go
// 创建带有默认中间件的路由：
// 日志与恢复中间件
router := gin.Default()
//创建不带中间件的路由：
//r := gin.New()

router.GET("/someGet", getting)
router.POST("/somePost", posting)
router.PUT("/somePut", putting)
router.DELETE("/someDelete", deleting)
router.PATCH("/somePatch", patching)
router.HEAD("/someHead", head)
router.OPTIONS("/someOptions", options)
```

`Gin` 的路由支持 `GET`，`POST`，`PUT`，`DELETE`，`PATCH`，`HEAD`，`OPTIONS` 请求，同时还有一个 `Any` 函数，可以同时支持以上的所有请求。

```go
router.Any("/some", someHandler)
```

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

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

将上一章节的代码添加其他请求方式的路由,并编写单元测试。

路由参数

gin的路由来自`httprouter`库。因此httprouter具有的功能，gin也具有，不过gin不支持路由正则表达式。

API参数

api 参数通过Context的Param方法来获取。

```
router.GET("/user/:name", func(c *gin.Context) {
    name := c.Param("name")
    c.String(http.StatusOK, name)
})
```

冒号`:`加上一个参数名组成路由参数。可以使用c.Params的方法读取其值。当然这个值是字串string。诸如`/user/ls`，和`/user/hello`都可以匹配，而`/user/`和`/user/ls/`不会被匹配。

```
router.GET("/user/:name/*action", func(c *gin.Context) {
    name := c.Param("name")
    action := c.Param("action")
    message := name + " is " + action
    c.String(http.StatusOK, message)
})
```

除了`:`，gin还提供了`*`号处理参数，`*`号能匹配的规则就更多。

URL参数

web提供的服务通常是client和server的交互。其中客户端向服务器发送请求，除了路由参数，其他的参数无非两种，查询字符串query string和报文体body参数。所谓query string，即路由用，用`?`以后连接的`key1=value2&key2=value2`的形式的参数。当然这个key-value是经过urlencode编码。

URL 参数通过 DefaultQuery 或 Query 方法获取。

对于参数的处理，经常会出现参数不存在的情况，对于是否提供默认值，gin也考虑了，并且给出了一个优雅的方案，使用c.DefaultQuery方法读取参数，其中当参数不存在的时候，提供一个默认值。使用Query方法读取正常参数，当参数不存在的时候，返回空字串。

```
func main() {
    router := gin.Default()
    router.GET("/welcome", func(c *gin.Context) {
        name := c.DefaultQuery("name", "Guest") //可设置默认值
        //nickname := c.Query("nickname") // 是
c.Request.URL.Query().Get("nickname") 的简写
        c.String(http.StatusOK, fmt.Sprintf("Hello %s ", name))
    })
    router.Run(":9527")
}
```

表单参数

http的报文体传输数据就比query string稍微复杂一点，常见的格式就有四种。例如`application/json`，`application/x-www-form-urlencoded`，`application/xml`和`multipart/form-data`。后面一个主要用于图片上传。json格式的很好理解，urlencode其实也不难，无非就是把query string的内容，放到了body体里，同样也需要urlencode。默认情况下，c.PostFROM解析的是`x-www-form-urlencoded`或`from-data`的参数。

表单参数通过 PostForm 方法获取：

```
func main() {

    router := gin.Default()
    //form
    router.POST("/form", func(c *gin.Context) {
        type1 := c.DefaultPostForm("type", "alert") //可设置默认值
        username := c.PostForm("username")
        password := c.PostForm("password")

        //hobbys := c.PostFormMap("hobby")
        //hobbys := c.QueryArray("hobby")
        hobbys := c.PostFormArray("hobby")

        c.String(http.StatusOK, fmt.Sprintf("type is %s, username is %s,
password is %s,hobby is %v", type1, username, password,hobbys))

    })

    router.Run(":9527")
}
```

我们还需要提供一个html页面(login.html)，来进行post请求：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>登录</title>
</head>
<body>
    <form action="http://127.0.0.1:9527/form" method="post"
enctype="application/x-www-form-urlencoded">
        用户名：<input type="text" name="username">
        <br>
        密  &nbsp码：<input type="password" name="password">
        <br>
        兴  &nbsp趣：
        <input type="checkbox" value="girl" name="hobby">女人
        <input type="checkbox" value="game" name="hobby">游戏
        <input type="checkbox" value="money" name="hobby">金钱
        <br>
        <input type="submit" value="登录">
    </form>
</body>
</html>
```

> 使用PostForm形式,注意必须要设置Post的type，同时此方法中忽略URL中带的参数,所有的参数需要从Body中获得。

文件上传

上传单个文件

前面介绍了基本的发送数据，其中`multipart/form-data`转用于文件上传。gin文件上传也很方便，和原生的net/http方法类似，不同在于gin把原生的request封装到c.Request中了。

首先我们创建一个go文件，demo06_file.go：

```
func main() {
    router := gin.Default()
    // Set a lower memory limit for multipart forms (default is 32 MiB)
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // single file
        file, _ := c.FormFile("file")
        log.Println(file.Filename)

        // Upload the file to specific dst.
        c.SaveUploadedFile(file, file.Filename)

        /*
        也可以直接使用io操作，拷贝文件数据。
        out, err := os.Create(filename)
        defer out.Close()
        _, err = io.Copy(out, file)
        */
```

```
        c.String(http.StatusOK, fmt.Sprintf("'%s' uploaded!", file.Filename))
    })
    router.Run(":8080")
}
```

使用`c.Request.FormFile`解析客户端文件name属性。如果不传文件，则会抛错，因此需要处理这个错误。此处我们略写了错误处理。一种是直接用c.SaveUploadedFile()保存文件。另一种方式是使用os的操作，把文件数据复制到硬盘上。

然后我们创建一个html页面，file.html：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>文件</title>
</head>
<body>
    <form action="http://127.0.0.1:8080/upload" method="post"
enctype="multipart/form-data">
        头像:
        <input type="file" name="file">
        <br>
        <input type="submit" value="提交">
    </form>

</body>
</html>
```

运行程序后，打开浏览器传递文件。

也可以使用终端命令访问http，上传文件。打开终端，并输入以下命令：

```
ls:~ tedu$ curl -X POST http://127.0.0.1:8080/upload -F
"file=@~/Documents/pro/momo.mp4" -H "Content-Type: multipart/form-data"
```

上传多个文件

所谓多个文件，无非就是多一次遍历文件，然后一次copy数据存储即可。

```
package main

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

func main() {
```

```
    router := gin.Default()
    // Set a lower memory limit for multipart forms (default is 32 MiB)
    router.MaxMultipartMemory = 8 << 20 // 8 MiB
    //router.Static("/", "./public")
    router.POST("/upload", func(c *gin.Context) {

        // Multipart form
        form, err := c.MultipartForm()
        if err != nil {
            c.String(http.StatusBadRequest, fmt.Sprintf("get form err: %s",
err.Error()))
            return
        }
        files := form.File["files"]

        for _, file := range files {
            if err := c.SaveUploadedFile(file, file.Filename); err != nil {
                c.String(http.StatusBadRequest, fmt.Sprintf("upload file err:
%s", err.Error()))
                return
            }
        }

        c.String(http.StatusOK, fmt.Sprintf("Uploaded successfully %d files ",
len(files)))
    })
    router.Run(":8080")
}
```

然后我们提供一个html页面，当然也可以使用终端命令：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>文件s</title>
</head>
<body>
<h1>上传多个文件</h1>

<form action="http://127.0.0.1:8080/upload" method="post"
enctype="multipart/form-data">
    Files: <input type="file" name="files" multiple><br><br>
    <input type="submit" value="提交">
</form>
</body>
</html>
```

路由组

router group是为了方便一部分相同的URL的管理，新建一个go文件(demo08_group.go)，

```
package main

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

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

    // Simple group: v1
    v1 := router.Group("/v1")
    {
        v1.GET("/login", loginEndpoint)
        v1.GET("/submit", submitEndpoint)
        v1.POST("/read", readEndpoint)
    }

    // Simple group: v2
    v2 := router.Group("/v2")
    {
        v2.POST("/login", loginEndpoint)
        v2.POST("/submit", submitEndpoint)
        v2.POST("/read", readEndpoint)
    }

    router.Run(":8080")
}
func loginEndpoint(c *gin.Context) {
    name := c.DefaultQuery("name", "Guest") //可设置默认值
    c.String(http.StatusOK, fmt.Sprintf("Hello %s \n", name))
}

func submitEndpoint(c *gin.Context) {
    name := c.DefaultQuery("name", "Guest") //可设置默认值
    c.String(http.StatusOK, fmt.Sprintf("Hello %s \n", name))
}

func readEndpoint(c *gin.Context) {
    name := c.DefaultQuery("name", "Guest") //可设置默认值
    c.String(http.StatusOK, fmt.Sprintf("Hello %s \n", name))
}
```

运行程序后，可以通过一个html页面访问，也可以通过终端使用命令直接访问，此处我们使用终端：

```
ls:~ edu$ curl http://127.0.0.1:8080/v1/login?name=ls
```

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")  // r.Group(xx)
    xx.GET("/oo", func(c *gin.Context) {...})
  }
```

通常我们将路由分组用在划分业务逻辑或划分API版本时。

路由原理

Gin框架中的路由使用的是httprouter这个库的原理。

其基本原理就是构造一个路由地址的前缀树。

检查点

> 说明：问题

3. Gin 请求的数据映射出来

数据解析绑定

模型绑定可以将请求体绑定给一个类型。目前Gin支持JSON、XML、YAML和标准表单值的绑定。简单来说，，就是根据Body数据类型，将数据赋值到指定的结构体变量中 (类似于序列化和反序列化) 。

Gin提供了两套绑定方法：

- Mustbind
  - 方法：Bind，BindJSON，BindXML，BindQuery，BindYAML
  - 行为：这些方法使用MustBindWith。如果存在绑定错误，则用c终止请求，使用`c.AbortWithError (400).SetType(ErrorTypeBind)`即可。将响应状态代码设置为400，Content-Type header设置为`text/plain;charset=utf-8`。请注意，如果在此之后设置响应代码，将会受到警告：`[GIN-debug][WARNING] Headers were already written. Wanted to override status code 400 with 422`将导致已经编写了警告[GIN-debug][warning]标头。如果想更好地控制行为，可以考虑使用ShouldBind等效方法。
- Shouldbind
  - 方法：`ShouldBind`,`ShouldBindJSON`,`ShouldBindXML`,`ShouldBindQuery`,`ShouldBindYA`

`ML`

- 行为：这些方法使用ShouldBindWith。如果存在绑定错误，则返回错误，开发人员有责任适当地处理请求和错误。

注意，使用绑定方法时，Gin 会根据请求头中 Content-Type 来自动判断需要解析的类型。如果你明确绑定的类型，你可以不用自动推断，而用 BindWith 方法。 你也可以指定某字段是必需的。如果一个字段被 `binding:"required"` 修饰而值却是空的，请求会失败并返回错误。

JSON绑定

JSON的绑定，其实就是将request中的Body中的数据按照JSON格式进行解析，解析后存储到结构体对象中。新建一个go文件，demo09_bind.go：

```
package main

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

type Login struct {
    User     string `form:"username" json:"user" uri:"user" xml:"user"
binding:"required"`
    Password string `form:"password" json:"password" uri:"password"
xml:"password" binding:"required"`
}

func main() {
    router := gin.Default()
    //1.binding JSON
    // Example for binding JSON ({"user": "ls", "password": "123456"})
    router.POST("/loginJSON", func(c *gin.Context) {
        var json Login
        //其实就是将request中的Body中的数据按照JSON格式解析到json变量中
        if err := c.ShouldBindJSON(&json); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }
        if json.User != "ls" || json.Password != "123456" {
            c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
            return
        }
        c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
    })

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

前面我们使用c.String返回响应，顾名思义则返回string类型。content-type是plain或者text。调用c.JSON则返回json数据。其中gin.H封装了生成json的方式，是一个强大的工具。使用golang可以像动态语言一样写字面量的json，对于嵌套json的实现，嵌套gin.H即可。

然后打开终端输入以下命令：

```
ls:~ edu$ curl -v -X POST http://127.0.0.1:8080/loginJSON -H 'content-
type:application/json' -d '{"user":"ls","password":"123456"}'
```

可以返回正确的结果。

假如我们传递的json中只有user数据：

```
ls:~ edu$ curl -v -X POST http://127.0.0.1:8080/loginJSON -H 'content-
type:application/json' -d '{"user":"ls"}'
```

那么会得到一个错误信息：

Form表单

其实本质是将c中的request中的body数据解析到form中。首先我们先看一下绑定普通表单的例子：

在之前的代码上继续添加：

```
// 3. Form 绑定普通表单的例子
    // Example for binding a HTML form (user=ls&password=123456)
    router.POST("/loginForm", func(c *gin.Context) {
        var form Login
        //方法一：对于FORM数据直接使用Bind函数，默认使用使用form格式解析,if
c.Bind(&form) == nil
        // 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
        }

        if form.User != "ls" || form.Password != "123456" {
            c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
            return
        }

        c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
    })
```

html页面，我们可以使用之前的login.html，但是要记得修改action后的路径：http://127.0.0.1:8080/loginForm

```
    router.POST("/login", func(c *gin.Context) {
        var form Login
```

```
        //方法二：使用BindWith函数,如果你明确知道数据的类型
        // 你可以显式声明来绑定多媒体表单：
        // c.BindWith(&form, binding.Form)
        // 或者使用自动推断：
        if c.BindWith(&form, binding.Form) == nil {
            if form.User == "user" && form.Password == "password" {
                c.JSON(200, gin.H{"status": "you are logged in ..... "})
            } else {
                c.JSON(401, gin.H{"status": "unauthorized"})
            }
        }
    })
```

URI绑定

```
// URI
    router.GET("/:user/:password", func(c *gin.Context) {
        var login Login
        if err := c.ShouldBindUri(&login); err != nil {
            c.JSON(400, gin.H{"msg": err})
            return
        }
        c.JSON(200, gin.H{"username": login.User, "password": login.Password})
    })
```

打开终端输入以下内容：

```
ls:~ edu$ curl -v http://127.0.0.1:8080/ls/123456
```

检查点

> 说明：问题

4. Gin 响应

既然请求可以使用不同的`content-type`，响应也如此。通常响应会有html，text，plain，json和xml等。 Gin提供了很优雅的渲染方法。

JSON/XML/YAML渲染

创建一个go文件(demo10_rendering.go)：

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
    "github.com/gin-gonic/gin/testdata/protoexample"
```

```
)

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

    // gin.H is a shortcut for map[string]interface{}
    r.GET("/someJSON", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/moreJSON", func(c *gin.Context) {
        // You also can use a struct
        var msg struct {
            Name    string `json:"user"`
            Message string
            Number  int
        }
        msg.Name = "ls"
        msg.Message = "hey"
        msg.Number = 123
        // 注意 msg.Name 变成了 "user" 字段
        // 以下方式都会输出 :   {"user": "ls", "Message": "hey", "Number": 123}
        c.JSON(http.StatusOK, msg)
    })

    r.GET("/someXML", func(c *gin.Context) {
        c.XML(http.StatusOK, gin.H{"user":"ls","message": "hey", "status":
http.StatusOK})
    })

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

    r.GET("/someProtoBuf", func(c *gin.Context) {
        reps := []int64{int64(1), int64(2)}
        label := "test"
        // The specific definition of protobuf is written in the
testdata/protoexample file.
        data := &protoexample.Test{
            Label: &label,
            Reps:  reps,
        }
        // Note that data becomes binary data in the response
        // Will output protoexample.Test protobuf serialized data
        c.ProtoBuf(http.StatusOK, data)
    })

    // Listen and serve on 0.0.0.0:8080
```

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

运行项目，打开浏览器输入网址：http://127.0.0.1:8080/moreJSON

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。

先要使用 `LoadHTMLGlob()` 或者 `LoadHTMLFiles()`方法来加载模板文件:

```
func main() {
    router := gin.Default()
    //加载模板
    router.LoadHTMLGlob("templates/*")
    //router.LoadHTMLFiles("templates/template1.html",
"templates/template2.html")
    //定义路由
    router.GET("/index", func(c *gin.Context) {
        //根据完整文件名渲染模板，并传递参数
        c.HTML(http.StatusOK, "index.tmpl", gin.H{
            "title": "Main website",
        })
    })
    router.Run(":8080")
}
```

创建一个目录：templates，然后在该目录下创建一个模板文件：

templates/index.tmpl

```
<html>
    <h1>
        {{ .title }}
    </h1>
</html>
```

运行项目，打开浏览器输入地址：http://localhost:8080/index

不同文件夹下模板名字可以相同，此时需要 `LoadHTMLGlob()` 加载两层模板路径。

```
router.LoadHTMLGlob("templates/**/*")
    router.GET("/posts/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "posts/index.tmpl", gin.H{
            "title": "Posts",
        })
        c.HTML(http.StatusOK, "users/index.tmpl", gin.H{
            "title": "Users",
        })

    })
```

重启项目后，打开浏览器输入以下网址：http://127.0.0.1:8080/posts/index

Gin也可以使用自定义的模板引擎，如下

```
import "html/template"

func main() {
    router := gin.Default()
    html := template.Must(template.ParseFiles("file1", "file2"))
    router.SetHTMLTemplate(html)
    router.Run(":8080")
}
```

自定义模板函数

定义一个不转义相应内容的`safe`模板函数如下：

```
func main() {
  router := gin.Default()
  router.SetFuncMap(template.FuncMap{
    "safe": func(str string) template.HTML{
      return template.HTML(str)
    },
  })
  router.LoadHTMLFiles("./index.tmpl")

  router.GET("/index", func(c *gin.Context) {
    c.HTML(http.StatusOK, "index.tmpl", "<a href='https://lianshiclass.com'>练识课堂</a>")
  })

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

在`index.tmpl`中使用定义好的`safe`模板函数：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>修改模板引擎的标识符</title>
</head>
<body>
<div>{{ . | safe }}</div>
</body>
</html>
```

使用模板继承

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

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

```
templates
├── includes
|   ├── home.tmpl
|   └── index.tmpl
├── layouts
|   └── base.tmpl
└── scripts.tmpl
```

然后我们定义一个`loadTemplates`函数如下：

```
func loadTemplates(templatesDir string) multitemplate.Renderer {
  r := multitemplate.NewRenderer()
  layouts, err := filepath.Glob(templatesDir + "/layouts/*.tmpl")
  if err != nil {
    panic(err.Error())
  }
  includes, err := filepath.Glob(templatesDir + "/includes/*.tmpl")
  if err != nil {
    panic(err.Error())
  }
  // 为layouts/和includes/目录生成 templates map
  for _, include := range includes {
    layoutCopy := make([]string, len(layouts))
    copy(layoutCopy, layouts)
    files := append(layoutCopy, include)
    r.AddFromFiles(filepath.Base(include), files...)
  }
  return r
}
```

我们在`main`函数中

```
func indexFunc(c *gin.Context){
  c.HTML(http.StatusOK, "index.tmpl", nil)
}

func homeFunc(c *gin.Context){
  c.HTML(http.StatusOK, "home.tmpl", nil)
}

func main(){
  r := gin.Default()
  r.HTMLRender = loadTemplates("./templates")
  r.GET("/index", indexFunc)
  r.GET("/home", homeFunc)
  r.Run()
}
```

文件响应

静态文件服务

可以向客户端展示本地的一些文件信息，例如显示某路径下地文件。服务端代码是：

```
package main

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

func main() {
    router := gin.Default()
    // 下面测试静态文件服务
    // 显示当前文件夹下的所有文件/或者指定文件
    router.StaticFS("/showDir", http.Dir("."))
    router.StaticFS("/files", http.Dir("/bin"))
    //Static提供给定文件系统根目录中的文件。
    //router.Static("/files", "/bin")
    router.StaticFile("/image", "./assets/image.jpg")

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

打开浏览器，输入地址：

- http://127.0.0.1:8080/showDir

- http://127.0.0.1:8080/files
- http://127.0.0.1:8080/image

重定向

http重定向

demo13_redirect.go：

```
package main

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

func main() {
    r := gin.Default()
    r.GET("/redirect", func(c *gin.Context) {
        //支持内部和外部的重定向
        c.Redirect(http.StatusMovedPermanently, "http://www.baidu.com/")
    })

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

打开浏览器输入：http://127.0.0.1:8080/redirect

路由重定向

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"})
})
```

同步异步

goroutine 机制可以方便地实现异步处理。当在中间件或处理程序中启动新的Goroutines时，你不应该在原始上下文使用它，你必须使用只读的副本。

新建一个go文件：

```go
package main

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

func main() {
    r := gin.Default()
    //1. 异步
    r.GET("/long_async", func(c *gin.Context) {
        // goroutine 中只能使用只读的上下文 c.Copy()
        cCp := c.Copy()
        go func() {
            time.Sleep(5 * time.Second)

            // 注意使用只读上下文
            log.Println("Done! in path " + cCp.Request.URL.Path)
        }()
    })
    //2. 同步
    r.GET("/long_sync", func(c *gin.Context) {
        time.Sleep(5 * time.Second)

        // 注意可以使用原始上下文
        log.Println("Done! in path " + c.Request.URL.Path)
    })

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


启动程序，打开浏览器并输入网址：

- http://127.0.0.1:8080/long_sync
- http://127.0.0.1:8080/long_async

检查点

> 说明：问题

5. Gin Middleware

中间件middleware

Go的net/http设计的一大特点就是特别容易构建中间件。gin也提供了类似的中间件。需要注意的是中间件只对注册过的路由函数起作用。对于分组路由，嵌套使用中间件，可以限定中间件的作用范围。中间件分为全局中间件，单个路由中间件和群组中间件。

我们之前说过, `Context` 是 `Gin` 的核心, 它的构造如下:

```
type Context struct {
    writermem responseWriter
    Request   *http.Request
    Writer    ResponseWriter

    Params   Params
    handlers HandlersChain
    index    int8

    engine   *Engine
    Keys     map[string]interface{}
    Errors   errorMsgs
    Accepted []string
}
```

其中 `handlers` 我们通过源码可以知道就是 `[]HandlerFunc`. 而它的签名正是:

```
type HandlerFunc func(*Context)
```

所以中间件和我们普通的 `HandlerFunc` 没有任何区别。我们怎么写 `HandlerFunc` 就可以怎么写一个中间件。

全局中间件

先定义一个中间件函数:

```
func MiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()
        fmt.Println("before middleware")
        //设置request变量到Context的Key中,通过Get等函数可以取得
        c.Set("request", "client_request")
        //发送request之前
        c.Next()

        //发送requst之后

        // 这个c.Write是ResponseWriter,我们可以获得状态等信息
        status := c.Writer.Status()
        fmt.Println("after middleware,", status)
        t2 := time.Since(t)
        fmt.Println("time:", t2)
```

```
    }
}
```

该函数很简单，只会给c上下文添加一个属性，并赋值。后面的路由处理器，可以根据被中间件装饰后提取其值。需要注意，虽然名为全局中间件，只要注册中间件的过程之前设置的路由，将不会受注册的中间件所影响。只有注册了中间件一下代码的路由函数规则，才会被中间件装饰。

```
router := gin.Default()

router.Use(MiddleWare())
{
    router.GET("/middleware", func(c *gin.Context) {
        //获取gin上下文中的变量
        request := c.MustGet("request").(string)
        req, _ := c.Get("request")
        fmt.Println("request:",request)
        c.JSON(http.StatusOK, gin.H{
            "middile_request": request,
            "request":         req,
        })
    })
}
router.Run(":8080")
```

使用router装饰中间件，然后在`/middlerware`即可读取request的值，注意在`router.Use(MiddleWare())`代码以上的路由函数，将不会有被中间件装饰的效果。

> 使用花括号包含被装饰的路由函数只是一个代码规范，即使没有被包含在内的路由函数，只要使用router进行路由，都等于被装饰了。想要区分权限范围，可以使用组返回的对象注册中间件。

运行项目，可以在浏览器访问 或者 终端输入命令进行访问，

```
ls:~ edu$ curl http://127.0.0.1:8080/middleware
```

Next()方法

我们怎么解决一个请求和一个响应经过我们的中间件呢？神奇的语句出现了， 没错就是`c.Next()`，所有中间件都有`Request`和`Response`的分水岭， 就是这个`c.Next()`，否则没有办法传递中间件。

服务端使用Use方法导入middleware，当请求/middleware来到的时候，会执行MiddleWare()， 并且我们知道在GET注册的时候，同时注册了匿名函数，所有请看Logger函数中存在一个c.Next()的用法，它是取出所有的注册的函数都执行一遍，然后再回到本函数中，所以，本例中相当于是先执行了c.Next()即注册的匿名函数，然后回到本函数继续执行， 所以本例的Print的输出顺序是：

```
fmt.Println("before middleware")

fmt.Println("request:",request)

fmt.Println("after middleware,", status)

fmt.Println("time:", t2)
```

如果将`c.Next()`放在`fmt.Println("after middleware,", status)`后面，那么`fmt.Println("after middleware,", status)`和`fmt.Println("request:",request)`执行的顺序就调换了。所以一切都取决于`c.Next()`执行的位置。`c.Next()`的核心代码如下：

```
// Next should be used only inside middleware.
// It executes the pending handlers in the chain inside the calling handler.
// See example in GitHub.
func (c *Context) Next() {
    c.index++
    for s := int8(len(c.handlers)); c.index < s; c.index++ {
        c.handlers[c.index](c)
    }
}
```

它其实是执行了后面所有的handlers。

一个请求过来，`Gin`会主动调用`c.Next()`一次。因为`handlers`是`slice`，所以后来者中间件会追加到尾部。这样就形成了形如`m1(m2(f()))`的调用链。正如上面数字① ② 标注的一样, 我们会依次执行如下的调用：

```
m1① -> m2① -> f -> m2② -> m1②
```

另外，如果没有注册就使用`MustGet`方法读取c的值将会抛错，可以使用Get方法取而代之。上面的注册装饰方式，会让所有下面所写的代码都默认使用了router的注册过的中间件。

单个路由中间件

当然，gin也提供了针对指定的路由函数进行注册。

```
router.GET("/before", MiddleWare(), func(c *gin.Context) {
        request := c.MustGet("request").(string)
        c.JSON(http.StatusOK, gin.H{
            "middile_request": request,
        })
    })
```

把上述代码写在 router.Use(Middleware())之前，同样也能看见`/before`被装饰了中间件。

中间件实践(鉴权)

中间件最大的作用，莫过于用于一些记录log，错误handler，还有就是对部分接口的鉴权。下面就实现一个简易的鉴权中间件。

简单认证BasicAuth

关于使用gin.BasicAuth() middleware，可以直接使用一个router group进行处理，本质和上面的一样。

先定义私有数据：

```
// 模拟私有数据
var secrets = gin.H{
    "ls":    gin.H{"email": "ls@lianshiclass.com", "phone": "123456"},
    "yang": gin.H{"email": "yang@lianshiclass.com", "phone": "111111"},
    "edu":   gin.H{"email": "edu@lianshiclass.com", "phone": "666666"},
}
```

然后使用 gin.BasicAuth 中间件，设置授权用户

```
    authorized := r.Group("/admin", gin.BasicAuth(gin.Accounts{
        "ls":    "123",
        "yang":  "111",
        "edu":   "666",
        "lucy":  "4321",
    }))
```

最后定义路由：

```
定义路由
    authorized.GET("/secrets", func(c *gin.Context) {
        // 获取提交的用户名（AuthUserKey）
        user := c.MustGet(gin.AuthUserKey).(string)
        if secret, ok := secrets[user]; ok {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": secret})
        } else {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": "NO SECRET :
("})
        }
    })
```

然后启动项目，打开浏览器输入以下网址：http://127.0.0.1:8080/admin/secrets 然后弹出的登录框输入正确的用户名和密码。

检查点

> 说明：问题

启动多个服务

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)
  }
}
```


五、拓展点（10分钟）

六、总结（5分钟）

> 说明：
>
> 回顾本堂课所有知识点；

微信搜一搜

练识课堂