

第二阶段：Go常用包和Gin Web框架（上篇）

讲师：阿杜

一、fmt

1.1 常用占位符

动词	功能
%v	按值的本来值输出
%+v	在 %v 的基础上，对结构体字段名和值进行展开
%#v	输出 Go 语言语法格式的值
%T	输出 Go 语言语法格式的类型和值
%%	输出 %% 本体
%b	整型以二进制方式显示
%o	整型以八进制方式显示
%d	整型以十进制方式显示
%x	整型以 十六进制显示
%X	整型以十六进制、字母大写方式显示
%U	Unicode 字符
%f	浮点数
%p	指针，十六进制方式显示
%s	字符串显示

二、time

- **时间对象**：golang中定义的一个对象
- **时间戳**：秒的整数形式
- **格式化时间**：给人看的时间（2022-03-27 09:15:29）

2.1 时间转换

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 1、时间对象
    now := time.Now()
    // 2、格式化时间
    strTime := now.Format("2006-01-02 15:04:05")
    // 3、时间差(从1970年1月1日 开始到现在多少s的时间)
    fmt.Println(now.Unix())
    // 4、格式化时间转时间对象
    loc, _ := time.LoadLocation("Asia/Shanghai")
    timeObj, _ := time.ParseInLocation("2006-01-02 15:04:05", strTime, loc)
    fmt.Println(timeObj.Unix()) // 1647858133
}
```

2.2 时间类型

- 我们可以通过 `time.Now()` 函数获取当前的时间对象，然后获取时间对象的年月日时分秒等信息。
- 注意：`%02d` 中的 `2` 表示宽度，如果整数不够 `2` 列就补上 `0`

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now() // 获取当前时间
    fmt.Printf("current time:%v\n", now)

    year := now.Year() // 年
    month := now.Month() // 月
    day := now.Day() // 日
    hour := now.Hour() // 小时
    minute := now.Minute() // 分钟
    second := now.Second() // 秒

    // 打印结果为: 2021-05-19 09:20:06
    fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute, second)
}
```

2.3 时间戳

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()          // 获取当前时间
    timestamp1 := now.Unix()    // 时间戳
    timestamp2 := now.UnixNano() // 纳秒时间戳
    fmt.Printf("current timestamp1:%v\n", timestamp1) // current timestamp1:1623560753
    fmt.Printf("current timestamp2:%v\n", timestamp2) // current
timestamp2:1623560753965606600
}
```

- 使用 `time.Unix()` 函数可以将时间戳转为时间格式

```
func timestampDemo2(timestamp int64) {
    timeObj := time.Unix(timestamp, 0) // 将时间戳转为时间格式
    fmt.Println(timeObj)
    year := timeObj.Year()    // 年
    month := timeObj.Month()  // 月
    day := timeObj.Day()      // 日
    hour := timeObj.Hour()    // 小时
    minute := timeObj.Minute() // 分钟
    second := timeObj.Second() // 秒
    fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute, second)
}
```

2.4 时间间隔

- `time.Duration` 是 `time` 包定义的一个类型，它代表两个时间点之间经过的时间，以纳秒为单位。
- `time.Duration` 表示一段时间间隔，可表示的最长时间段大约290年。
- `time`包中定义的时间间隔类型的常量如下：

```
const (
    Nanosecond Duration = 1
    Microsecond      = 1000 * Nanosecond
    Millisecond       = 1000 * Microsecond
    Second            = 1000 * Millisecond
    Minute            = 60 * Second
    Hour              = 60 * Minute
)
```

2.5 时间格式化

- 时间类型有一个自带的方法 `Format` 进行格式化
- 需要注意的是Go语言中格式化时间模板不是常见的 `Y-m-d H:M:S`
- 而是使用Go的诞生时间2006年1月2号15点04分（记忆口诀为2006 1 2 3 4）。
- 补充：如果想格式化为12小时方式，需指定 `PM` 。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    // 格式化的模板为Go的出生时间2006年1月2号15点04分 Mon Jan
    fmt.Println(now.Format("2006-01-02 15:04:05.000 Mon Jan")) // 2021-06-13
    13:10:18.143 Sun Jun
}
```

- 字符串转时间类型

```
package main

import (
    "fmt"
    "time"
)

func main() {
    loc, _ := time.LoadLocation("Asia/Shanghai")
    // 按照指定时区和指定格式解析字符串时间
    timeObj, err := time.ParseInLocation("2006-01-02 15:04:05", "2019-08-04
14:15:20", loc)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(timeObj) // 2019-08-04 14:15:20 +0800 CST
}
```

2.6 时间操作函数

查询数据库对时间过滤

- Add
 - 我们在日常的编码过程中可能会遇到要求时间+时间间隔的需求
 - Go 语言的时间对象有提供Add 方法如下

- Sub
 - 求两个时间之间的差值

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now() // 获取当前时间
    // 1分钟前
    m, _ := time.ParseDuration("-1m")
    m1 := now.Add(m)
    fmt.Println(m1)

    // 1分钟后
    mm, _ := time.ParseDuration("1m")
    mm1 := now.Add(mm)
    fmt.Println(mm1)
}
```

三、os、ioutil

3.1 os

- 可以参考: <https://cloud.tencent.com/developer/article/1342799>
- 对操作系统、文件等相关操作

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 1、获取当前目录
    fmt.Println(os.Getwd())
    // 2、修改当前目录
    os.Chdir("/Users/")
    fmt.Println(os.Getwd())
    // 3、创建文件夹
    os.Mkdir("go_demo", 0777)
    // 4、删除文件夹或者文件
    os.Remove("go_demo")
}
```

```

// 5、修改文件夹或者文件的名称
os.Rename("go_demo", "new_demo_go")
// 6、新建文件
os.Create("./file.txt")
// 7、打开文件并写入文件
/*
    O_RDONLY 打开只读文件
    O_WRONLY 打开只写文件
    O_RDWR 打开既可以读取又可以写入文件
    O_APPEND 写入文件时将数据追加到文件尾部
    O_CREATE 如果文件不存在，则创建一个新的文件
*/
file, _ := os.OpenFile("file.txt", os.O_RDWR|os.O_APPEND, 0666)
_, err := file.WriteString("你好222")
fmt.Println(err)
// http://v5blog.cn/pages/d19d5a/#_01-
%E6%89%93%E5%BC%80%E5%92%8C%E5%85%B3%E9%97%AD%E6%96%87%E4%BB%B6
}

```

3.2 ioutil

- 处理文件读写操作

```

package main

import (
    "fmt"
    "io/ioutil"
    "strings"
)

func main() {
    //1. 读取文件中的所有数据
    fileName1 := "./file.txt"
    data, _ := ioutil.ReadFile(fileName1)
    fmt.Println(string(data))

    //2. 写出数据
    fileName2 := "./file.txt"
    s1 := "hello world"
    ioutil.WriteFile(fileName2, []byte(s1), 0777)

    // 3. 读取所有数据
    s2 := "qwertyuiop"
    r1 := strings.NewReader(s2)
    data, _ = ioutil.ReadAll(r1)
    fmt.Println(data)

    // 4. 读取一个目录下内容：子文件和子目录，但是仅有一层

```

```
dirName := "/Users/adoo"
fileInfos, _ := ioutil.ReadDir(dirName)
fmt.Println(len(fileInfos))
}
```

四、Flag

4.1 flag

- Go语言内置的flag包实现了命令行参数的解析，flag包使得开发命令行工具更为简单。
- 支持的命令行参数格式有以下几种：
 - flag xxx （使用空格，一个-符号）
 - flag xxx （使用空格，两个-符号）
 - flag=xxx （使用等号，一个-符号）
 - flag=xxx （使用等号，两个-符号）

其中，布尔类型的参数必须使用等号的方式指定。

Flag解析在第一个非flag参数（单个“-”不是flag参数）之前停止，或者在终止符“-”之后停止。

1. 其他函数

flag.Args() // 返回命令行参数后的其他参数，以[]string类型

flag.NArg() // 返回命令行参数后的其他参数个数

flag.NFlag() // 返回使用的命令行参数个数

4.2 完整示例

1. main.go

```
package main

import (
    "flag"
    "fmt"
    "time"
)

func main() {
    // 定义命令行参数方式1
    var name string
    var age int
    var married bool
    var delay time.Duration
    flag.StringVar(&name, "name", "张三", "姓名")
    flag.IntVar(&age, "age", 18, "年龄")
    flag.BoolVar(&married, "married", false, "婚否")
    flag.DurationVar(&delay, "d", 0, "延迟的时间间隔")
}
```

```

// 解析命令行参数
flag.Parse()
fmt.Println(name, age, married, delay)
// 返回命令行参数后的其他参数
fmt.Println(flag.Args())
// 返回命令行参数后的其他参数个数
fmt.Println(flag.NArg())
// 返回使用的命令行参数个数
fmt.Println(flag.NFlag())
}

```

2. 查看帮助

```

C:\aaa\gin_demo> go run main.go --help
-age int
    年龄 (default 18)
-d duration
    延迟的时间间隔
-married
    婚否
-name string
    姓名 (default "张三")

```

3. flag参数演示

```

C:\aaa\gin_demo> go run main.go -name pprof --age 28 -married=false -d=1h30m
pprof 28 false 1h30m0s
[]
0
4

```

4. 非flag参数演示

```

C:\aaa\gin_demo> go run main.go a b c
张三 18 false 0s
[a b c]
3

```

五、net-http

5.1 介绍

```
.
├── ClientGet
│   └── main.go // 发送get请求
├── ClientPost
│   └── main.go // 发送post请求
└── Server
    └── main.go // web服务
```

- Go语言内置的net/http包十分的优秀，提供了HTTP客户端和服务端的实现。

5.2 web服务

1. Server/main.go

- 客户端 请求信息 封装在 http.Request 对象中
- 服务端返回的响应报文会被保存在http.Response结构体中
- 发送给客户端响应的并不是http.Response，而是通过http.ResponseWriter接口来实现的

方法签名	描述
Header()	用户设置或获取响应头信息
Write()	用于写入数据到响应体
WriteHeader()	用于设置响应状态码，若不调用则默认状态码为200 OK。

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

type Data struct {
    Name string `json:"name"`
}

// 处理GET请求: http://127.0.0.1:8005/req/get?name=root
func dealGetRequestHandler(w http.ResponseWriter, r *http.Request) {
    // 获取请求的参数
    query := r.URL.Query()
    if len(query["name"]) > 0 {
        // 方式1: 通过字典下标取值
        name := query["name"][0]
        fmt.Println("通过字典下标获取: ", name)
    }
}
```

```

// 方式2: 使用Get方法, , 如果没有值会返回空字符串
name2 := query.Get("name")
fmt.Println("通过get方式获取: ", name2)
type data struct {
    Name2 string
}
d := data{
    Name2: name2,
}
//w.Write([]byte(string(d))) // 返回string
json.NewEncoder(w).Encode(d) // 返回json数据
}

// 处理POST请求: http://127.0.0.1:8005/req/post {"name": "root"}
func dealPostRequestHandler(w http.ResponseWriter, r *http.Request) {
    // 请求体数据
    bodyContent, _ := ioutil.ReadAll(r.Body)
    strData := string(bodyContent)
    var d Data
    json.Unmarshal([]byte(strData), &d) // gin.ShouldBind
    fmt.Printf("body content:[%s]\n", string(bodyContent))
    // 返回响应内容
    json.NewEncoder(w).Encode(fmt.Sprintf("收到名字: %s", d.Name))
}

func main() {
    http.HandleFunc("/req/post", dealPostRequestHandler)
    http.HandleFunc("/req/get", dealGetRequestHandler)
    http.ListenAndServe(":8005", nil)
    // 在golang中, 你要构建一个web服务, 必然要用到http.ListenAndServe
    // 第二个参数必须要有一个handler
}

```

2. postman测试

- 测试发送get请求
 - <http://127.0.0.1:8005/req/get?name=root>

GET

http://127.0.0.1:8005/req/get?name=root

Send

ParamsAuthHeaders (8)BodyPre-req. TestsSettingsCookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	name	root			

Body

200 OK3 ms125 BSave Response

PrettyRawPreviewVisualizeText

1 ["root"]
2

- 测试发送post请求
 - http://127.0.0.1:8005/req/post

POST

http://127.0.0.1:8005/req/post

Send

ParamsAuthHeaders (8)BodyPre-req. TestsSettingsCookies

rawJSONBeautify

1 {
2 "name": "root"
3 }

Body

200 OK4 ms139 BSave Response

PrettyRawPreviewVisualizeText

1 "收到名字: root"
2

5.3 请求数据

1. ClientGet/main.go

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
```

```

)

func main() {
    requestGet()
}

func requestGet() {
    apiUrl := "http://127.0.0.1:8005/req/get"
    data := url.Values{}
    data.Set("name", "root")
    u, _ := url.ParseRequestURI(apiUrl)
    u.RawQuery = data.Encode() // URL encode
    fmt.Println("请求路由为: ", u.String())
    resp, _ := http.Get(u.String())
    b, _ := ioutil.ReadAll(resp.Body)
    fmt.Println("返回数据为: ", string(b))
}

/*
请求路由为: http://127.0.0.1:8005/req/get?name=root
返回数据为: ["root"]
*/

```

2. ClientPost/main.go

```

package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "strings"
)

func main() {
    requestPost()
}

func requestPost() {
    url := "http://127.0.0.1:8005/req/post"
    // 表单数据 contentType := "application/x-www-form-urlencoded"
    contentType := "application/json"
    data := `{"name":"rootPort"}`
    resp, _ := http.Post(url, contentType, strings.NewReader(data))
    b, _ := ioutil.ReadAll(resp.Body)
    fmt.Println(string(b))
}

/*
"收到名字: rootPort"
*/

```

六、gRPC

6.1 gRPC是什么

- gRPC是google开源的一个RPC框架和库，支持多语言之间的通信。底层通信采用的是 HTTP2 协议。gRPC在设计上使用了 ProtoBuf 这种接口描述语言。这种IDL语言可以定义各种服务，google还提供了一种工具 protoc 来编译这种IDL语言，生成各种各样的语言来操作服务。

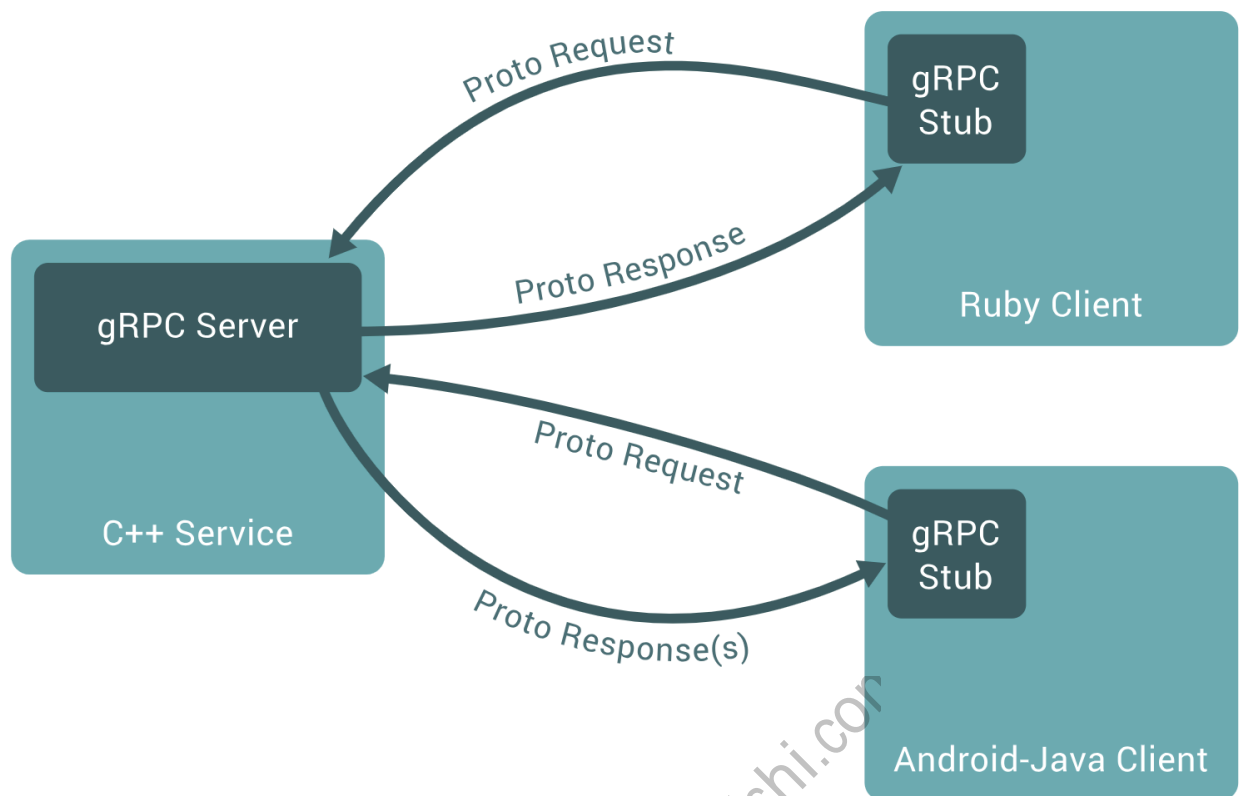
6.2 gRPC特点

- 定义服务简单，可以很快的搭建出一个RPC调度的服务
- gRPC是与语言无关，平台无关的。你定义好了一个protobuf协议，就可以用protoc生成不同语言的协议框架
- 使用HTTP2协议，支持双向流。客户端和服务端可以双向通信

6.3 RPC与RESTful区别是什么

- 在客户端和服务端通信还有一种基于http协议的 RESTful 架构模式，RESTful一般是对资源的操作，它是名词（资源地址），然后添加一些动作对这些资源进行操作。而RPC是基于函数，它是动词。
- RPC一般基于TCP协议，当然gRPC是基于HTTP2，但它也是比HTTP协议更加有效率和更多特性。RESTful一般都是基于HTTP协议。
- 传输方面：自定义的TCP协议或者使用HTTP2协议，报文体积更小，所以传输效率更高，RESTful一般基于http协议，报文体积大。
- gRPC用的是protobuf的IDL语言，会编码为二进制协议的数据，而RESTful一般是用json的数据格式，json格式的编解码更耗时。

6.4 gRPC通信流程



- 客户端 (gRPC Stub) 调用 A 方法, 发起 RPC 调用
- 对请求信息使用 Protobuf 进行对象序列化压缩 (IDL)
- 服务端 (gRPC Server) 接收到请求后, 解码请求体, 进行业务逻辑处理并返回
- 对响应结果使用 Protobuf 进行对象序列化压缩 (IDL)
- 客户端接受到服务端响应, 解码请求体。回调被调用的 A 方法, 唤醒正在等待响应 (阻塞) 的客户端调用并返回响应结果

6.5 gPRC环境安装

- 安装 **protoc**, protoc是protobuf编译器, 可以将源文件 **xxx.proto** 编译成 **开发语言文件**, 例如 **xxx.pb.go**
 - 下载 <https://github.com/protocolbuffers/protobuf/releases/download/v3.20.0/protoc-3.20.0-win64.zip>
 - 解压 **protoc-3.20.0-win64.zip**
 - 将 **protoc-3.20.0-win64.zip/bin** 设置为环境变量
- 验证 **protoc**

```
*** Desktop$ protoc --version
libprotoc 3.20.0
```

- 安装 **protoc-gen-go-grpc** 和 **protoc-gen-go**, 因为protoc没有内置go生成器, 想实现 .proto→.go 的转换的话还需要安装protobuf的golang编译器插件protoc-gen-go, 用于生成go文件

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

- 验证 **protoc-gen-go-grpc** 和 **protoc-gen-go**

```
*** Desktop$ protoc-gen-go-grpc --version
protoc-gen-go-grpc 1.2.0
*** Desktop$ protoc-gen-go --version
protoc-gen-go-grpc 1.28.1
```

6.6 gRPC使用

- 在开发server与client之前，我们需要先定义proto文件， `grpc-practice/proto/message.proto`

```
syntax = "proto3";
// 这部分的内容是关于最后生成的go文件是处在哪个目录哪个包中， ../pb代表在当前目录的上一级pb目录中生成， message代表了生成的go文件的包名是message。
option go_package = "../pb:pb";
// 定义了一个service， 称为MessageSender， 这个服务中有一个rpc方法， 名为Send。这个方法会发送一个MessageRequest， 然后返回一个MessageResponse。
message MessageResponse {
    string responseSomething = 1;
}

message MessageRequest {
    string saySomething = 1;
}

service MessageSender {
    rpc Send(MessageRequest) returns (MessageResponse) {}
}
```

- 根据proto文件生成go代码文件， `grpc-practice/proto/`

```
#生成message.pb.go、message_grpc.pb.go这两个文件。在这两个文件中， 包含了我们定义方法的go语言实现， 也包含了我们定义的请求与响应的go语言实现。
protoc --go_out=. message.proto
protoc --go-grpc_out=. message.proto
```

- server端响应实现 `grpc-practice/serviceImpl/MessageSenderServerImpl.go`

```
package serviceImpl

import (
    "context"
    "grpc-practice/pkg/pb"
    "log"
)

type MessageSenderServerImpl struct {
    *pb.UnimplementedMessageSenderServer
}
```

```
func (MessageSenderServerImpl) Send(context context.Context, request
*pb.MessageRequest) (*pb.MessageResponse, error) {
    log.Println("receive message:", request.GetSaySomething())
    resp := &pb.MessageResponse{}
    resp.ResponseSomething = "roger that!"
    return resp, nil
}
```

- server端启动gRPC服务, `grpc-practice/server/main.go`

```
package main

import (
    "google.golang.org/grpc"
    "grpc-practice/pkg/pb"
    "grpc-practice/pkg/serviceImpl"
    "log"
    "net"
)

func main() {
    //new一个grpc服务
    srv := grpc.NewServer()
    //注册服务
    pb.RegisterMessageSenderServer(srv, serviceImpl.MessageSenderServerImpl{})
    listener, err := net.Listen("tcp", ":8002")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    err = srv.Serve(listener)
    if err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

- client端发起请求, `grpc-practice/client/main.go`

```
package main

import (
    "context"
    "google.golang.org/grpc"
    "grpc-practice/pkg/pb"
    "log"
)

func main() {
    //建立grpc连接
    conn, err := grpc.Dial("127.0.0.1:8002", grpc.WithInsecure())
```



```

if err != nil {
    log.Fatalf("did not connect: %v", err)
}
defer conn.Close()
//new grpc client
client := pb.NewMessageSenderClient(conn)
//发起grpc请求
resp, err := client.Send(context.Background(), &pb.MessageRequest{SaySomething:
"hello world!"})
if err != nil {
    log.Fatalf("could not greet: %v", err)
}
log.Println("receive message:", resp.GetResponseSomething())
}

```

Run: go build main.go x go build main.go (1) x

```

GOROOT=/usr/local/go #gosetup
GOPATH=/Users/apple/go #gosetup
/usr/local/go/bin/go build -o /private/var/folders/1y/myrm4q4s2vq_pz2xtq1lsjpw0000gn/T/GoLand/___go_build_main_go
2022/06/01 19:29:12 receive message: hello world!

```

Run: go build main.go x go build main.go (1) x

```

GOROOT=/usr/local/go #gosetup
GOPATH=/Users/apple/go #gosetup
/usr/local/go/bin/go build -o /private/var/folders/1y/myrm4q4s2vq_pz2xtq1lsjpw0000gn/___go_build_main_go__
2022/06/01 19:29:12 receive message: roger that!
Process finished with the exit code 0

```

七、Gin基本使用-原理篇

扩展知识:

go mod

类似于我们Python的pip做依赖管理的，每个项目可能都会使用一些外部包，外步包有很多版本

- go mod就是帮助我们自动管理你们的包和版本号的
- 如果没有go mod别人如何才能运行你们的代码

外部的包：其他人封装好的，实现特定功能的代码

常用命令：

- `go get github.com/gin-gonic/gin` // 将GitHub上，或者其他仓库里的代码下载到本地，直接导入就可以是用了
- `go mod tidy` // 更新我们的依赖（第一次运行项目，执行这个命令他会将项目中所有的外部包一次性的全部下载到本地

参考资料：http://v5blog.cn/pages/e952db/#_4-1-init-%E5%87%BD%E6%95%B0%E4%BB%8B%E7%BB%8D

6.1 Gin入门

1. 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

2. 安装

要安装Gin软件包，您需要安装Go并首先设置Go工作区。

- 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
```

- 将其导入您的代码中：

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

- （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"
```

3. hello world

```
package main

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

func main() {
    // 1.创建（实例化gin.Engine结构体对象）
    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!")
    })
}
```

```

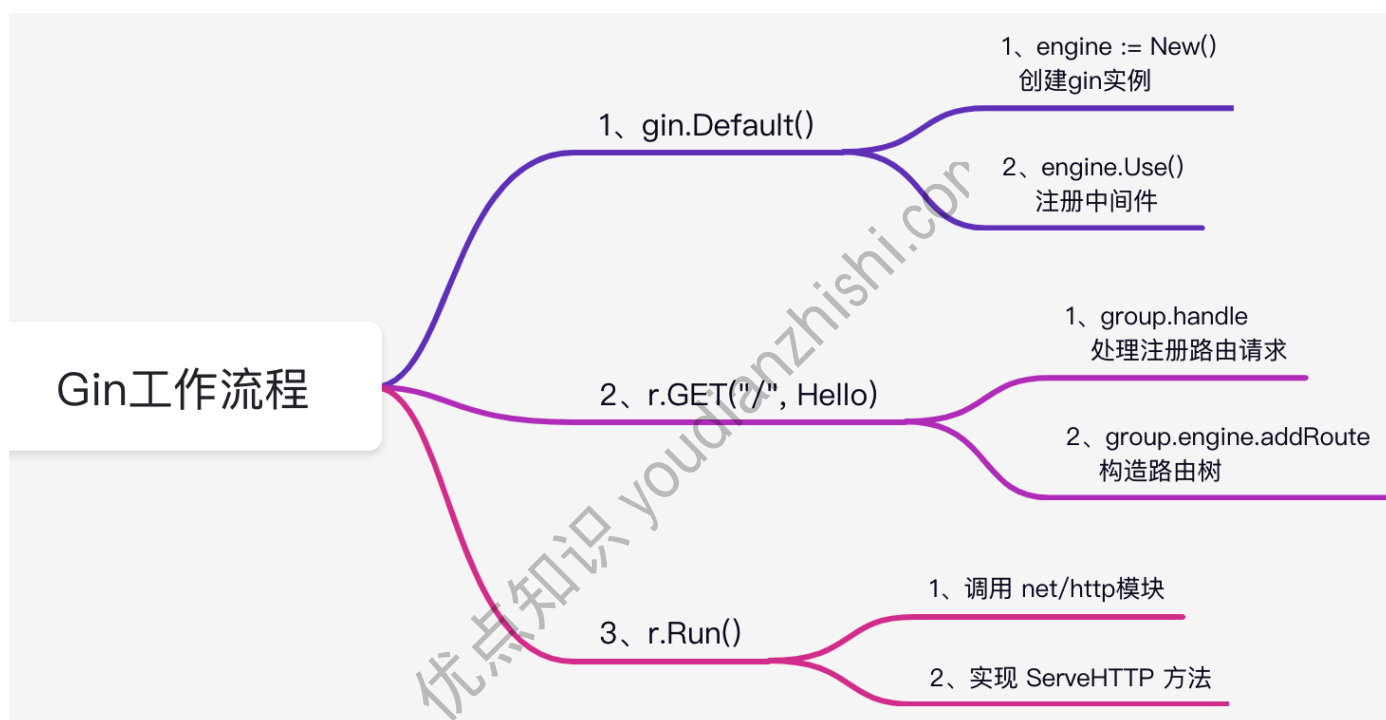
})
// 3. 监听端口, 默认在8080
// Run("里面不指定端口号默认为8080")
r.Run(":8000")
}

```

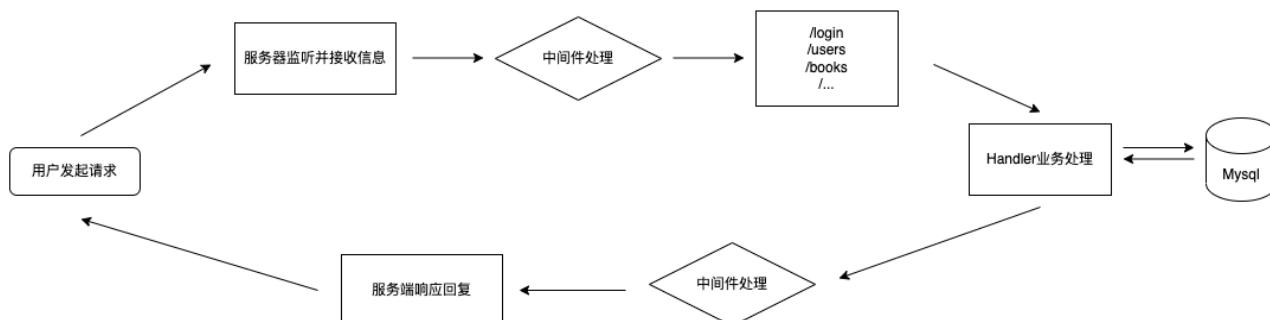
6.2 Gin工作流程

1. 核心概念

- **Engine** 容器对象, 整个框架的基础
- **Engine.trees** 负责存储路由和handle方法的映射, 采用类似字典树的结构
- **Engine.RouterGroup**, 其中的Handlers存储着所有中间件
- **Context** 上下文对象, 负责处理 **请求和回应**, 其中的 **handlers** 是存储处理请求时中间件和处理方法的



2. 请求处理流程



6.3 Gin原理解析

- 参考资料: <http://v5blog.cn/pages/dd7d5a/>

1. 先看 gin.Default()

- Default()跟New()几乎一模一样, 就是调用了gin内置的Logger(), Recovery()中间件

```
// Default返回一个已经附加了Logger和Recovery中间件的Engine实例
func Default() *Engine {
    debugPrintWARNINGDefault()
    engine := New() // 默认实例
    // 注册中间件, 中间件的是一个函数, 最终只要返回一个 type HandlerFunc func(*Context) 就可以
    engine.Use(Logger(), Recovery()) // 默认注册的两个中间件
    return engine
}
```

2. engine := New() 初始化

- 通过调用 gin.New() 方法来实例化 Engine容器 .
 - 1.初始化了Engine
 - 2.将RouterGroup的Handlers(数组)设置成nil, basePath设置成 /
 - 3.为了方便, RouterGroup里面也有一个Engine指针, 这里将刚刚初始化的engine赋值给了RouterGroup的engine指针
 - 4.为了防止频繁的context GC造成效率的降低, 在Engine里使用了sync.Pool, 专门存储gin的Context

```
func New() *Engine {
    debugPrintWARNINGNew()
    // Engine 容器对象, 整个框架的基础
    engine := &Engine{ // 初始化语句
        // Handlers 全局中间件组在注册路由时使用
        RouterGroup: RouterGroup{ // Engine.RouterGroup, 其中的Handlers存储着所有中间件
            Handlers: nil,
            basePath: "/",
            root: true,
        },
        // 树结构, 保存路由和处理方法的映射
        trees: make(methodTrees, 0, 9),
    }
    engine.RouterGroup.engine = engine
    return engine
}
```

3. engine.Use() 注册中间件

- gin框架中的中间件设计很巧妙, 我们可以首先从我们最常用的 r := gin.Default() 的Default 函数开始看
- 它内部构造一个新的 engine 之后就通过 Use() 函数注册了 Logger 中间件和 Recovery 中间件
- Use()就是gin的引入中间件的入口了.
- 仔细分析这个函数, 不难发现Use()其实是在给RouteGroup引入中间件的.
- 具体是如何让中间件在RouteGroup上起到作用的, 等说到RouteGroup再具体说.

```
func Default() *Engine {
    debugPrintWARNINGDefault()
    engine := New()
    engine.Use(Logger(), Recovery()) // 默认注册的两个中间件
    return engine
}
```

gin.use() 调用 RouterGroup.Use() 往 RouterGroup.Handlers 写入记录

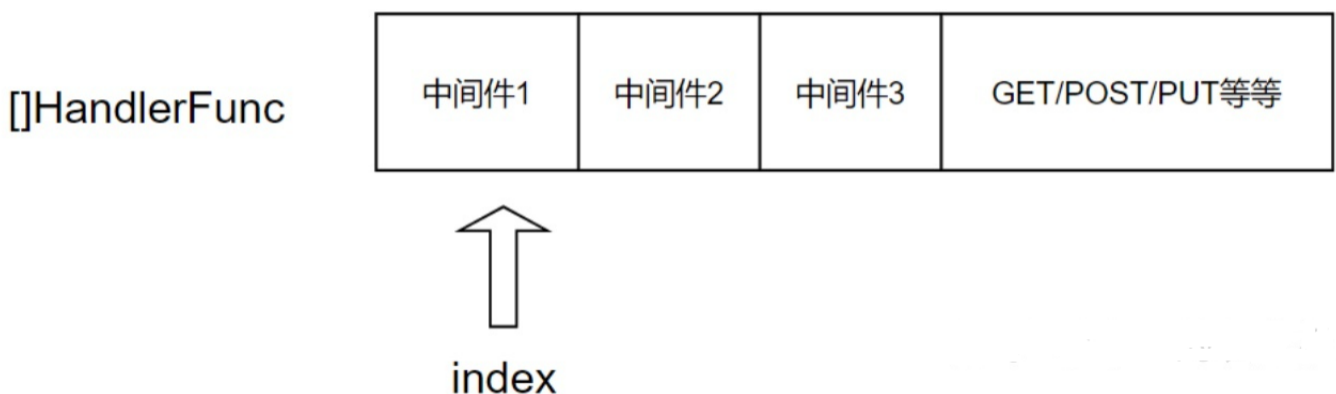
```
func (engine *Engine) Use(middleware ...HandlerFunc) IRoutes {
    engine.RouterGroup.Use(middleware...)
    engine.rebuild404Handlers() //注册404处理方法
    engine.rebuild405Handlers() //注册405处理方法
    return engine
}
// 其中`Handlers`字段就是一个数组,用来存储中间件
func (group *RouterGroup) Use(middleware ...HandlerFunc) IRoutes {
    group.Handlers = append(group.Handlers, middleware...)
    return group.returnObj()
}
```

组成一条处理函数链条 HandlersChain

- 也就是说,我们会将一个路由的中间件函数和处理函数结合到一起组成一条处理函数链条HandlersChain
- 而它本质上就是一个由HandlerFunc组成的切片

```
type HandlersChain []HandlerFunc
```

HandlersChain图解



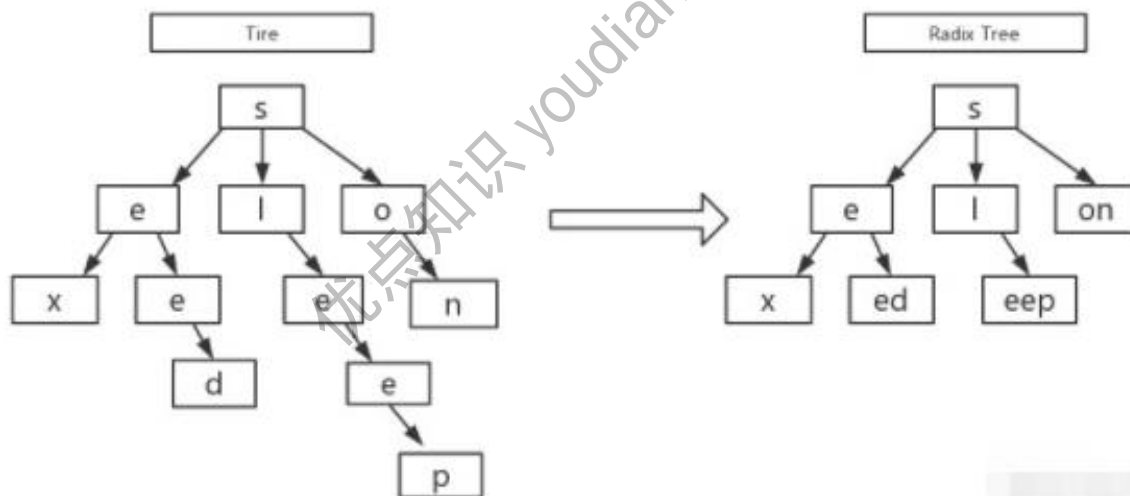
4. r.GET() 注册路由

```
r.GET("/", func(c *gin.Context) {  
    c.String(http.StatusOK, "hello World!")  
})
```

- 通过Get方法将路由和处理视图函数注册

```
func (group *RouterGroup) GET(relativePath string, handlers ...HandlerFunc) IRoutes {  
    return group.handle(http.MethodGet, relativePath, handlers)  
}  
  
func (group *RouterGroup) handle(httpMethod, relativePath string, handlers  
    HandlersChain) IRoutes {  
    absolutePath := group.calculateAbsolutePath(relativePath)  
    handlers = group.combineHandlers(handlers) // 将处理请求的函数与中间件函数结合  
    group.engine.addRoute(httpMethod, absolutePath, handlers) // ---- 调用 addRoute方法注  
    册路由  
    return group.returnObj()  
}
```

- 路由匹配图解



- 这段代码就是利用method, path, 将handlers注册到engine的trees中.
- 注意这里为什么是HandlersChain呢, 可以简单说一下, 就是将中间件和处理函数都注册到method, path的tree中了.

```
// addRoute adds a node with the given handle to the path.  
// Not concurrency-safe! 并发不安全  
func (n *node) addRoute(path string, handlers HandlersChain) {  
    fullPath := path  
    // 添加完成后, 经过此节点的路由条数将会+1  
    n.priority++  
  
    // Empty tree
```

```
// 如果为空树， 即只有一个根节点"/" 则插入一个子节点， 并将当前节点设置为root类型的节点
if len(n.path) == 0 && len(n.children) == 0 {
    n.insertChild(path, fullPath, handlers)
    n.nType = root
    return
}
```

```
parentFullPathIndex := 0
```

```
walk:
```

```
for {
    // Find the longest common prefix.
    // This also implies that the common prefix contains no ':' or '*'
    // since the existing key can't contain those chars.
    // 找到最长的共有前缀的长度 即到i位置 path[i] == n.path[i]
    i := longestCommonPrefix(path, n.path)
```

```
    // Split edge
```

```
    // 假设当前节点存在的前缀信息为 hello
```

```
    // 现有前缀信息为heo的节点进入， 则当前节点需要被拆分
```

```
    // 拆分为 he节点 以及 (llo 和 o 两个子节点)
```

```
    if i < len(n.path) {
```

```
        child := node{
```

```
            // 除去公共前缀部分， 剩余的内容作为子节点
```

```
            path:      n.path[i:],
```

```
            wildChild: n.wildChild,
```

```
            indices:   n.indices,
```

```
            children:  n.children,
```

```
            handlers:  n.handlers,
```

```
            priority:  n.priority - 1,
```

```
            fullPath:  n.fullPath,
```

```
        }
```

```
        // 将现有节点的剩余部分加入子节点
```

```
        n.children = []*node{&child}
```

```
        // []byte for proper unicode char conversion, see #65
```

```
        // 更新当前节点
```

```
        n.indices = bytesconv.BytesToString([]byte{n.path[i]})
```

```
        n.path = path[:i]
```

```
        n.handlers = nil
```

```
        n.wildChild = false
```

```
        n.fullPath = fullPath[:parentFullPathIndex+i]
```

```
    }
```

```
    // Make new node a child of this node
```

```
    // 将新来的节点插入新的parent节点作为子节点
```

```
    if i < len(path) {
```

```
        path = path[i:]
```

```
        c := path[0]
```

```
        // '/' after param
```

```

//如果是参数节点 形如/:i, 那么node节点就是:i
if n.nType == param && c == '/' && len(n.children) == 1 {
    parentFullPathIndex += len(n.path)
    n = n.children[0]
    n.priority++
    continue walk
}

// Check if a child with the next path byte exists
// 检查路径path下一个字节的子节点是否存在,如果存在,就处理公共部分前缀
for i, max := 0, len(n.indices); i < max; i++ {
    if c == n.indices[i] {
        parentFullPathIndex += len(n.path)
        i = n.incrementChildPrio(i)
        n = n.children[i]
        continue walk
    }
}

// Otherwise insert it
// 否则就直接添加到子节点中
if c != ':' && c != '*' && n.nType != catchAll {
    // []byte for proper unicode char conversion, see #65
    // 注意这里是直接拼接第一个字符到n.indices, 缓存下一个节点的第一个字符
    n.indices += bytesconv.BytesToString([]byte{c})
    child := &node{
        fullPath: fullPath,
    }
    n.addChild(child)
    n.incrementChildPrio(len(n.indices) - 1)
    n = child
}
//当children是通配符类型时
} else if n.wildChild {
    // inserting a wildcard node, need to check if it conflicts with the
existing wildcard
    // 添加通配符节点, 需要检查该通配符是否与已有的通配符冲突
    n = n.children[len(n.children)-1]
    n.priority++

    // Check if the wildcard matches
    if len(path) ≥ len(n.path) && n.path == path[:len(n.path)] &&
        // Adding a child to a catchAll is not possible
        n.nType != catchAll &&
        // Check for longer wildcard, e.g. :name and :names
        (len(n.path) ≥ len(path) || path[len(n.path)] == '/') {
        continue walk
    }

    // Wildcard conflict
    // 通配符冲突时, 报错处理

```



```

        pathSeg := path
        if n.nType != catchAll {
            pathSeg = strings.SplitN(pathSeg, "/", 2)[0]
        }
        prefix := fullPath[:strings.Index(fullPath, pathSeg)] + n.path
        panic("'" + pathSeg +
            "' in new path '" + fullPath +
            "' conflicts with existing wildcard '" + n.path +
            "' in existing prefix '" + prefix +
            "'")
    }
    // 添加路径及处理函数到子节点
    n.insertChild(path, fullPath, handlers)
    return
}

// Otherwise add handle to current node
// 设置处理函数, 如果已经存在, 则报错
if n.handlers != nil {
    panic("handlers are already registered for path '" + fullPath + "'")
}
n.handlers = handlers
n.fullPath = fullPath
return
}
}

```

5. r.run()启动服务

- 通过调用 `net/http` 来启动服务, 由于 `engine` 实现了 `ServeHTTP` 方法
- 只需要直接传 `engine` 对象就可以完成初始化并启动

```

func (engine *Engine) Run(addr ...string) (err error) {
    defer func() { debugPrintError(err) }()
    address := resolveAddress(addr)
    debugPrint("Listening and serving HTTP on %s\n", address)
    // 在golang中, 你要构建一个web服务, 必然要用到http.ListenAndServe
    // 第二个参数必须要有一个handler
    err = http.ListenAndServe(address, engine) // gin使用net/http模块
    return
}

func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}

// 来自 net/http 定义的接口, 只要实现了这个接口就可以作为处理请求的函数
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

// 实现了ServeHTTP方法

```

```

//从 sync.pool 里面拿去一块内存
//对这块内存做初始化工作, 防止数据污染
//处理请求 handleHTTPRequest
//请求处理完成后, 把这块内存归还到 sync.pool 中
func (engine *Engine) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    c := engine.pool.Get().(*Context)
    c.writemem.reset(w)
    c.Request = req
    c.reset()
    engine.handleHTTPRequest(c)
    engine.pool.Put(c)
}

```

七、Gin基本使用-基础篇

7.1 路由与传参

1. 无参路由

```

package main

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

func HelloWorldHandler(c *gin.Context) {
    // gin.Context, 封装了request和response
    c.String(http.StatusOK, "hello World!")
}

func main() {
    // 1.创建路由
    r := gin.Default()
    // 2.绑定路由规则, 执行的函数
    // 基本路由 /hello/
    r.GET("/hello", HelloWorldHandler)
    // 3.监听端口, 默认在8080
    fmt.Println("运行地址: http://127.0.0.1:8000")
    r.Run(":8000")
}

```

2. API参数

- 可以通过Context的Param方法来获取API参数

```
package main

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

func GetBookDetailHandler(c *gin.Context) {
    bookId := c.Param("id")
    // gin.Context, 封装了request和response
    c.String(http.StatusOK, fmt.Sprintf("成功获取书籍详情: %s", bookId))
}

func main() {
    // 1.创建路由
    r := gin.Default()
    // 2.绑定路由规则, 执行的函数
    // 基本路由 /book/24/
    r.GET("/book/:id", GetBookDetailHandler)
    // 3.监听端口, 默认在8080
    fmt.Println("运行地址: http://127.0.0.1:8000/book/24/")
    r.Run(":8000")
}
```

3. url参数

- URL参数可以通过DefaultQuery()或Query()方法获取
- DefaultQuery()若参数不存在, 返回默认值, Query()若不存在, 返回空串
- <http://127.0.0.1:8000/user?name=zhangsan>

```
package main

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

func GetUserDetailHandler(c *gin.Context) {
    //username := c.DefaultQuery("name", "xxx")
    username := c.Query("name")
    // gin.Context, 封装了request和response
    c.String(http.StatusOK, fmt.Sprintf("姓名: %s", username))
}
```

```
func main() {
    // 1.创建路由
    r := gin.Default()
    // 2.绑定路由规则, 执行的函数
    // 基本路由 /user?name=root
    r.GET("/user/", GetUserDetailHandler)
    // 3.监听端口, 默认在8080
    fmt.Println("运行地址: http://127.0.0.1:8000/user?name=root")
    r.Run(":8000")
}
```

4. ShouldBind参数绑定

- 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。

```
package main

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

// Binding from JSON
type Login struct {
    Username string `form:"username" json:"username" binding:"required"`
    Password string `form:"password" json:"password" binding:"required"`
}

func LoginHandler(c *gin.Context) {
    var login Login
    if err := c.ShouldBind(&login); err != nil {
        // 如果数据校验不通过直接返回
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    }
    c.String(http.StatusOK, fmt.Sprintf("姓名: %s -- 密码: %s", login.Username,
        login.Password))
}

func main() {
    // 1.创建路由
    r := gin.Default()
    // 2.绑定路由规则, 执行的函数
    // 基本路由 /login/
    r.POST("/login", LoginHandler)
    // 3.监听端口, 默认在8080
    fmt.Println("运行地址: http://127.0.0.1:8000/login/")
    r.Run(":8000")
}
```

}

• postman测试

POST

▼

http://127.0.0.1:8000/login/

Send

▼

ParamsAuthHeaders (8)Body●Pre-req.TestsSettingsCookies

raw▼JSON▼Beautify

1

2

3

4

{

"username": "root",

"password": "123456"

}

Body▼200 OK3 ms149 BSave Response▼

PrettyRawPreviewVisualizeText▼

1 姓名: root -- 密码: 123456

7.2 响应返回

1. 响应String

```
package main

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

func main() {
    r := gin.Default()
    r.GET("/response/", ResponseStringHandler)
    fmt.Println("http://127.0.0.1:8000/response/")
    r.Run(":8000")
}

func ResponseStringHandler(c *gin.Context) {
    c.String(http.StatusOK, "返回简单字符串")
}
```

2. 响应JSON

```
package main

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

func main() {
    r := gin.Default()
    r.GET("/response/", ResponseJsonHandler)
    fmt.Println("http://127.0.0.1:8000/response/")
    r.Run(":8000")
}

func ResponseJsonHandler(c *gin.Context) {
    type Data struct {
        Msg  string `json:"msg"`
        Code int    `json:"code"`
    }
    d := Data{
        Msg:  "Json数据",
        Code: 1001,
    }
    c.JSON(http.StatusOK, d)
    //// 也可以使用 gin.H返回 json数据
    //c.JSON(http.StatusOK, gin.H{
    //    "msg": "success",
    //})
}
```

3. 路由重定向

```
package main

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

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.Redirect(http.StatusMovedPermanently, "http://v5blog.cn")
    })
    fmt.Println("http://127.0.0.1:8000")
    r.Run(":8000")
}
```

```
}
```

7.3 路由分发

- 为什么需要路由分发？
 - 我们一个项目有非常多的模块，如果全部写在一块导致代码结构混乱，不利于后续的扩展
 - 按照大的模块，每个模块有自己独立的路由，主路由可以再main.go中进行注册

1. 项目结构

```
.  
├── go.mod  
├── go.sum  
├── main.go  
└── routers  
    ├── books.go  
    └── users.go
```

2. main.go

```
package main  
  
import (  
    "days/routers"  
    "fmt"  
    "github.com/gin-gonic/gin"  
)  
  
func main() {  
    r := gin.Default()  
    routers.LoadBooks(r)  
    routers.LoadUsers(r)  
    fmt.Println("用户路由: http://127.0.0.1:8000/user")  
    fmt.Println("书籍路由: http://127.0.0.1:8000/book")  
    r.Run(":8000")  
}
```

3. routers/users.go

```
package routers  
  
import (  
    "net/http"  
    "github.com/gin-gonic/gin"  
)  
  
func LoadUsers(e *gin.Engine) {  
    e.GET("/user", UserHandler)
```

```
}

func UserHandler(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "message": "User Router",
    })
}
```

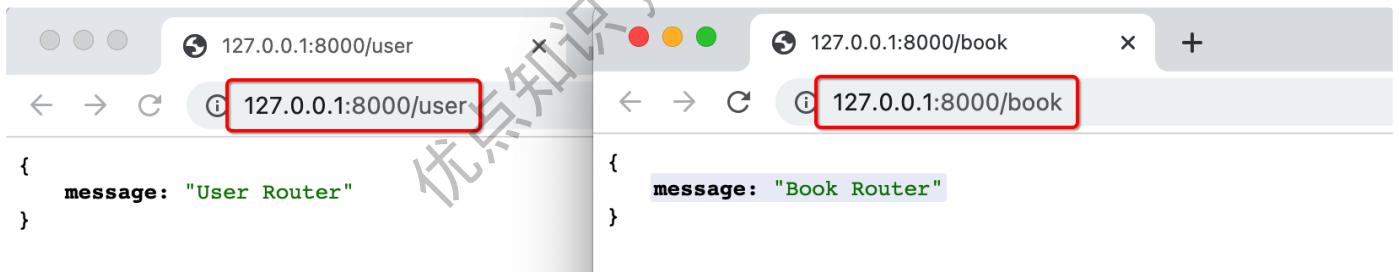
4. routers/books.go

```
package routers

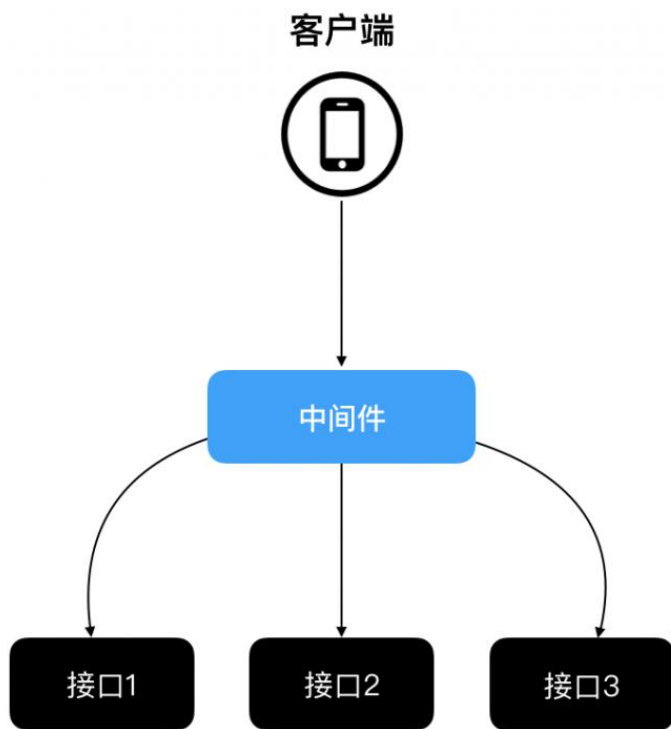
import (
    "net/http"
    "github.com/gin-gonic/gin"
)

func LoadBooks(e *gin.Engine) {
    e.GET("/book", GetBookHandler)
}

func GetBookHandler(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "message": "Book Router",
    })
}
```



7.4 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

1. 全局中间件

```
package main

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

func MiddleWare() gin.HandlerFunc {
    return func(c *gin.Context) {
        fmt.Println("我是一个全局中间件")
    }
}

func main() {
    r := gin.Default()
    r.Use(MiddleWare())
    r.GET("/hello", func(c *gin.Context) {
        fmt.Println("执行了Get方法")
        c.JSON(200, gin.H{"msg": "success"})
    })
    r.Run()
}
```

2. 局部中间件

```
package main

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

func Middleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        fmt.Println("这里可以做一些身份验证等")
    }
}

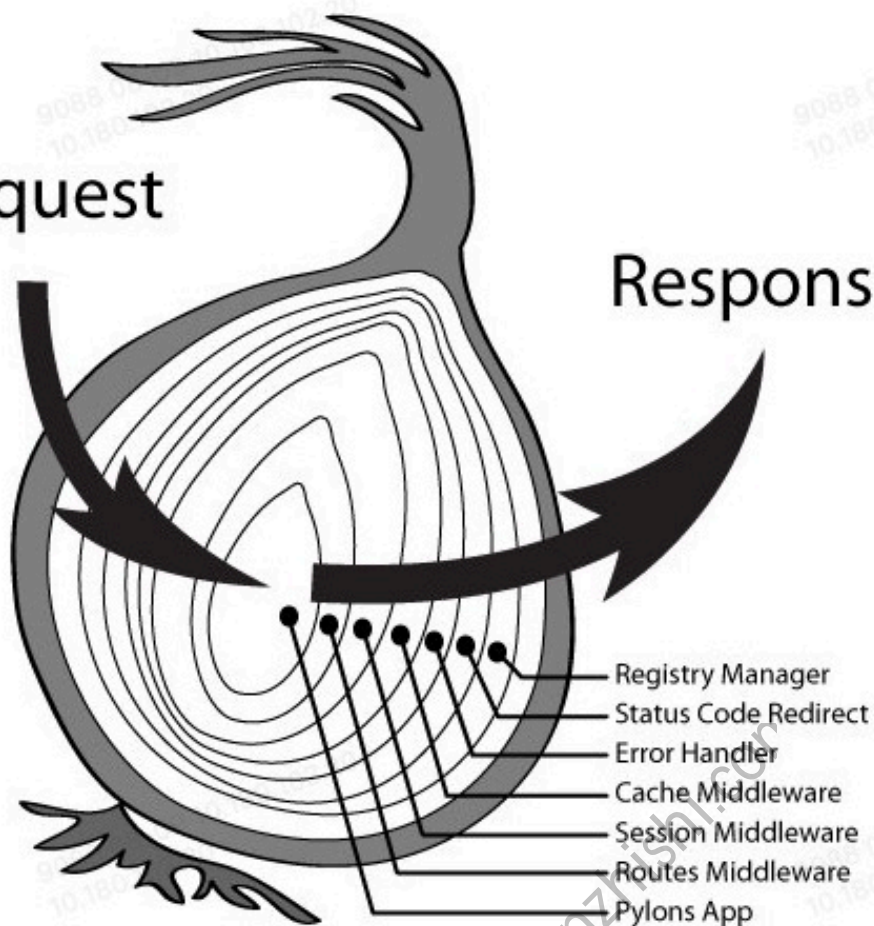
func main() {
    r := gin.Default()
    // 首页无需验证
    r.GET("/index", func(c *gin.Context) {
        c.JSON(200, gin.H{"msg": "index 页面"})
    })
    // home页需要用户登录才能查看
    r.GET("/home", Middleware(), func(c *gin.Context) {
        c.JSON(200, gin.H{"msg": "home 页面"})
    })
    r.Run()
}
```

3. Next()方法

- 在中间件中调用 `next()` 方法，会从`next()`方法调用的地方跳转到Handler函数
- Handler函数执行完成，若中间件还有部分代码未执行（中间件中`next()`之后的代码），则执行该代码

Request

Response



```
package main

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

func main() {
    r := gin.Default()
    r.Use(Log(), RequestID())
    r.GET("/", func(c *gin.Context) {
        fmt.Println("app running...")
        time.Sleep(time.Second * 5)
        c.String(http.StatusOK, "hello World!")
    })
    r.Run()
}

func Log() gin.HandlerFunc {
    return func(c *gin.Context) {
        fmt.Println("log start")
        c.Next()
    }
}
```

```

        fmt.Println("log end")
    }
}

func RequestID() gin.HandlerFunc {
    return func(c *gin.Context) {
        fmt.Println("requestid start")
        c.Next()
        fmt.Println("requestid end")
    }
}

```

4. 实现token认证

- <http://127.0.0.1:8080/index> index首页无需token直接访问
- <http://127.0.0.1:8080/home> home家目录需要对token进行验证，验证通过才可访问

```

package main

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

func AuthMiddleware() func(c *gin.Context) {
    return func(c *gin.Context) {
        // 客户端携带Token有三种方式 1.放在请求头 2.放在请求体 3.放在URI
        // token验证成功，返回c.Next继续，否则返回c.Abort()直接返回
        token := c.Request.Header.Get("token")
        fmt.Println("获取token:", token) // 获取token: xxxxxx
        if token == "" {
            c.String(400, "身份验证不通过")
            c.Abort()
        }
    }
}

func main() {
    r := gin.Default()
    // 首页无需验证
    r.GET("/index", func(c *gin.Context) {
        c.JSON(200, gin.H{"msg": "index 页面"})
    })
    // home页需要用户登录才能查看
    r.GET("/home", AuthMiddleware(), func(c *gin.Context) {
        c.JSON(200, gin.H{"msg": "home 页面"})
    })
    r.Run()
}

```

- 测试验证

GET

▼

http://127.0.0.1:8080/home

Send

▼

ParamsAuthHeaders (7)BodyPre-req.TestsSettingsCookies

<input checked="" type="checkbox"/>	Accept-Encoding	gzip, deflate, br	
<input checked="" type="checkbox"/>	Connection	keep-alive	
<input type="checkbox"/>	token	xxxxxxx	

Body

200 OK4 ms138 B

Save Response

Pretty

Raw

Preview

Visualize

Text

▼

1

身份验证不通过

优点知识 youdianzhishi.com