

极客时间 Go 初级工程师第二课

type 定义与 Server 抽象

大明

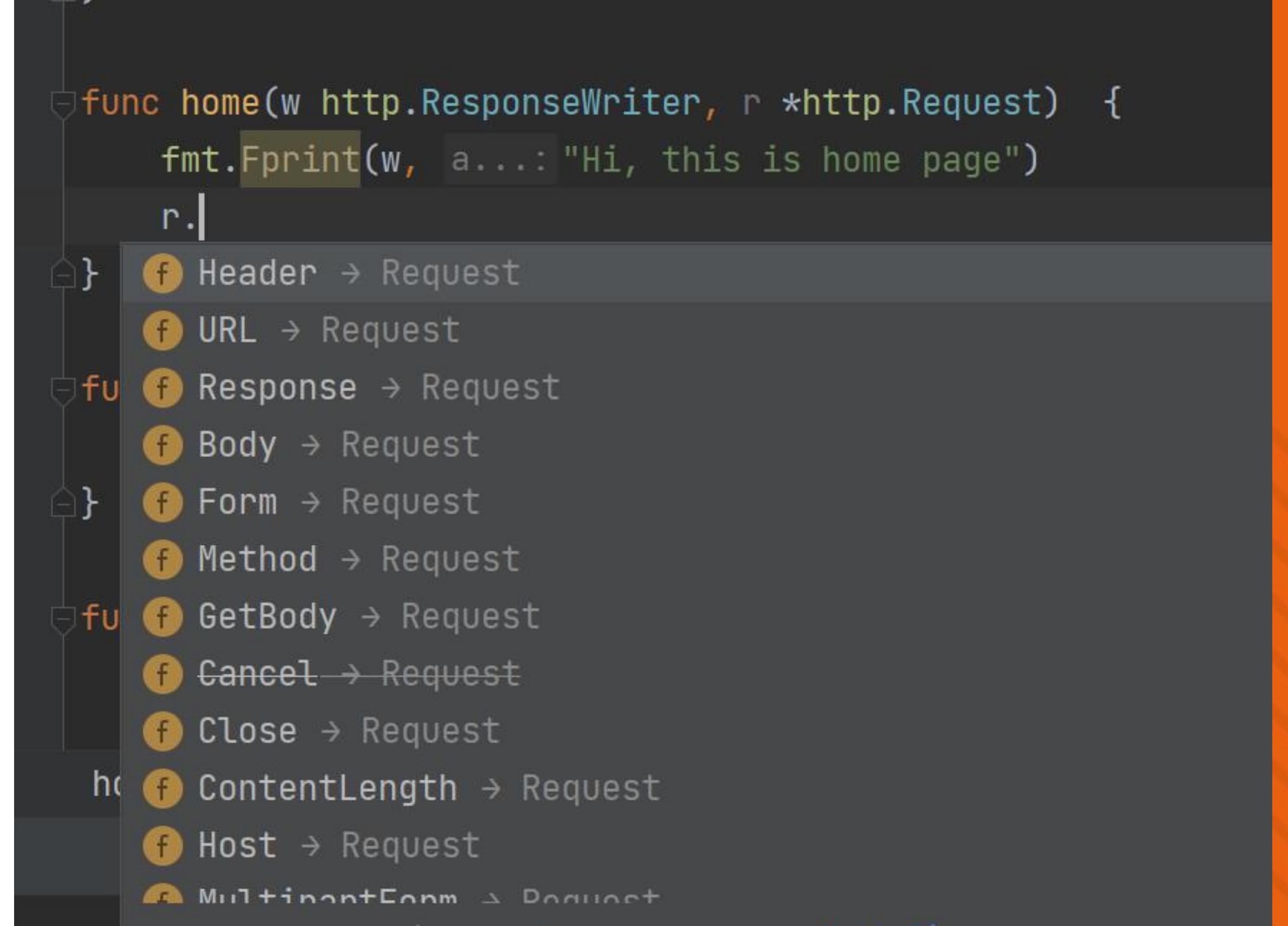


目录

1. http 库
2. 基础语法 type
3. Server 与 Context 抽象
4. 简单支持 RESTFul API

http 库 —— Request 概览

- Body 和 GetBody
- URL
- Method
- Header
- Form
- ...



A screenshot of an IDE showing code completion for the 'r.' field in a Request struct. The code being typed is:

```
func home(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprint(w, a...: "Hi, this is home page")  
    r.|
```

The code completion dropdown shows several methods available for the 'r' parameter:

- f Header → Request
- f URL → Request
- f Response → Request
- f Body → Request
- f Form → Request
- f Method → Request
- f GetBody → Request
- f Cancel → Request
- f Close → Request
- ho f ContentLength → Request
- f Host → Request
- MultipartForm → Request

http 库 —— Request Body

- Body: 只能读取一次, 意味着你读了别人就不能读了; 别人读了你就不能读了;

```
func readBodyOnce(w http.ResponseWriter, r *http.Request) {
    body, err := io.ReadAll(r.Body)
    if err != nil {
        fmt.Fprintf(w, "read body failed: %v", err)
        // 记住要返回, 不然就还会执行后面的代码
        return
    }
    // 类型转换, 将 []byte 转换为 string
    fmt.Fprintf(w, "read the data: %s \n", string(body))

    // 尝试再次读取, 啥也读不到, 但是也不会报错
    body, err = io.ReadAll(r.Body)
    if err != nil {
        // 不会进来这里
        fmt.Fprintf(w, "read the data one more time got error: %v", err)
        return
    }
    fmt.Fprintf(w, "read the data one more time: [%s] and read data length %d \n", string(body), len(body))
}
```

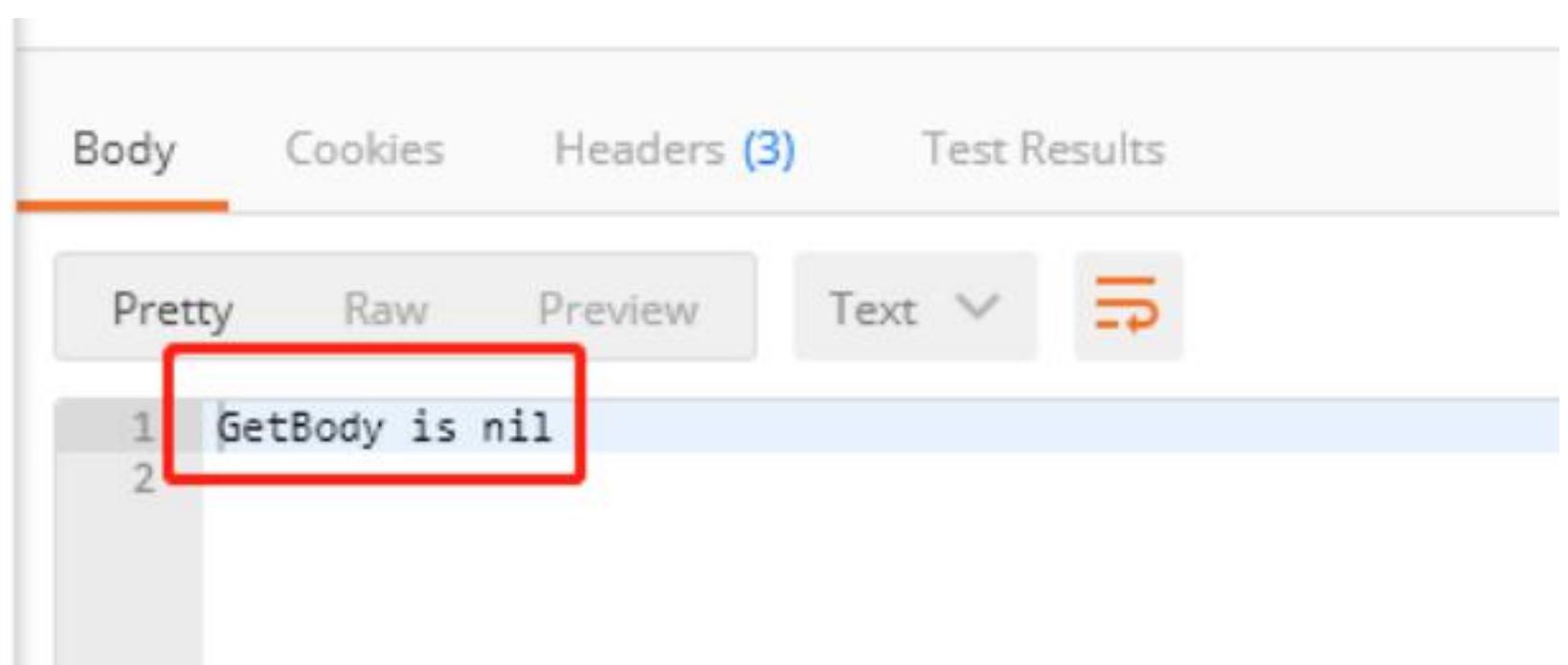
The screenshot shows a debugger interface with the 'Body' tab selected. Below it, there are tabs for 'Pretty', 'Raw', 'Preview', 'Text', and a copy icon. The 'Text' tab is currently active. The content area displays the following text:

```
1 read the data: this is body
2 read the data one more time: [] and read data length 0
3
```

http 库 —— Request Body – GetBody

- Body: 只能读取一次，意味着你读了别人就不能读了；别人读了你就不能读了；
- GetBody: 原则上是可以多次读取，但是在原生的http.Request里面，这个是 nil
- 在读取到 body 之后，我们就可以用于反序列化，比如说将json格式的字符串转化为一个对象等

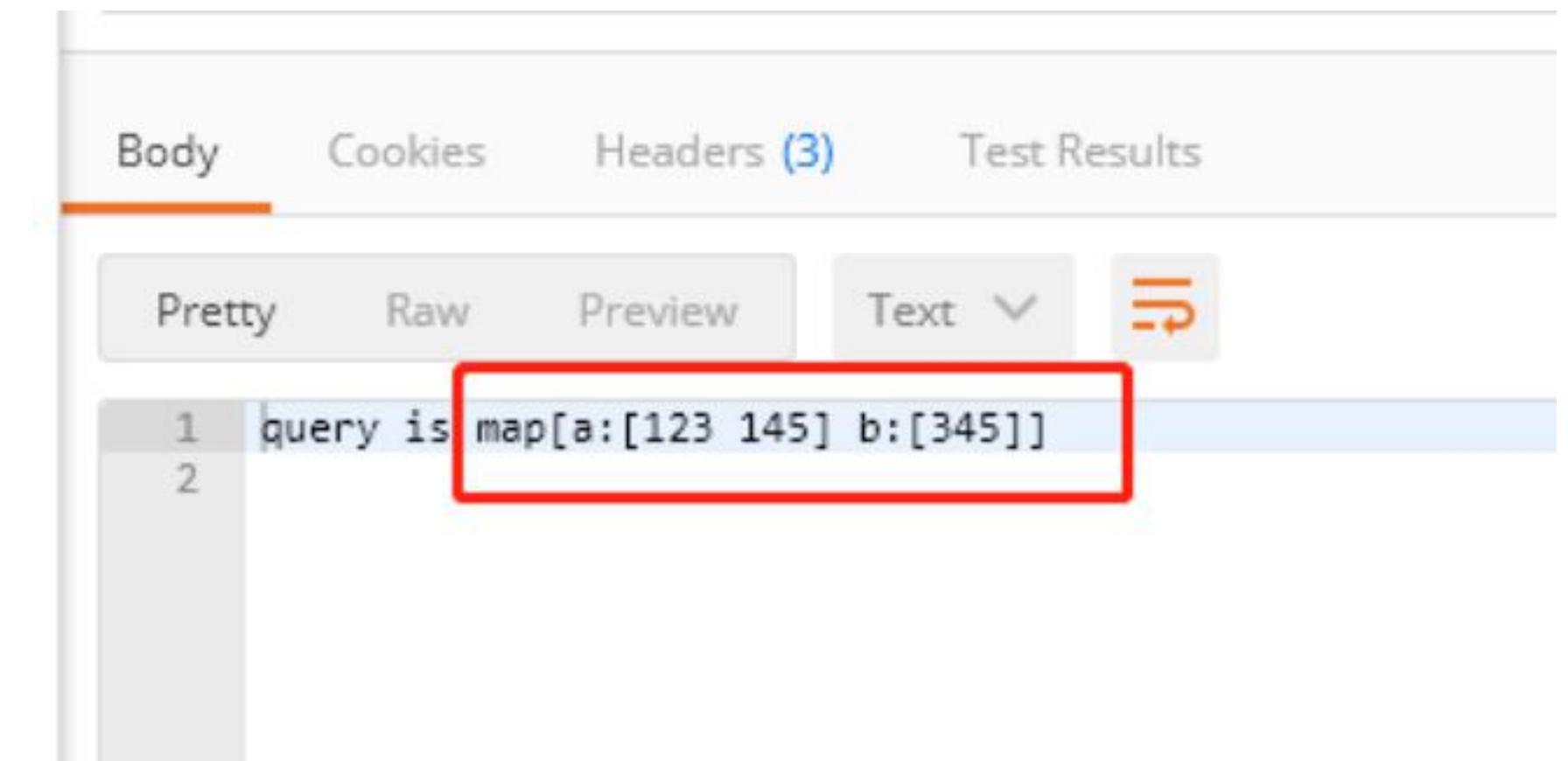
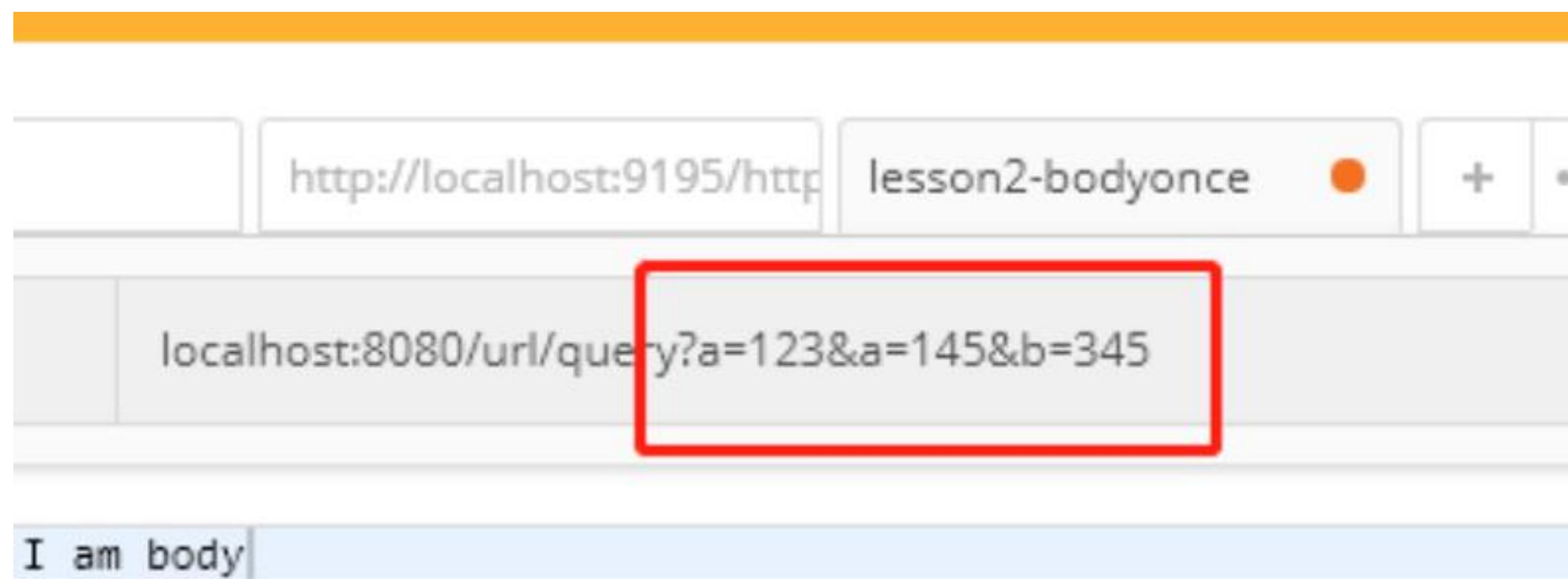
```
func getBodyIsNil(w http.ResponseWriter, r *http.Request) {  
    if r.GetBody == nil {  
        fmt.Fprint(w, a...: "GetBody is nil \n")  
    } else {  
        fmt.Fprintf(w, format: "GetBody not nil \n")  
    }  
}
```



http 库 —— Request Query

- 除了 Body，我们还可能传递参数的地方是 Query
- 也就是
http://xxx.com/your/path?id=123
&b=456
- 所有的值都被解释为字符串，所以需要自己解析为数字等

```
func queryParams(w http.ResponseWriter, r *http.Request) {  
    values := r.URL.Query()  
    fmt.Fprintf(w, "query is %v\n", values)  
}
```



http 库 —— Request URL

- 包含路径方面的所有信息和一些很有用的操作

```
type URL struct {
    Scheme      string
    Opaque      string // encoded
    User        *Userinfo // username
    Host        string // host or IP
    Path        string // path (relative)
    RawPath     string // encoded
    ForceQuery  bool   // append a query
    RawQuery    string // encoded
    Fragment    string // fragment
    RawFragment string // encoded
}
```

```
queryParams(w http.ResponseWriter, r *http.Request) {
    values := r.URL.Query()
    t.Fprintf(w, "URL: %v", values)
}

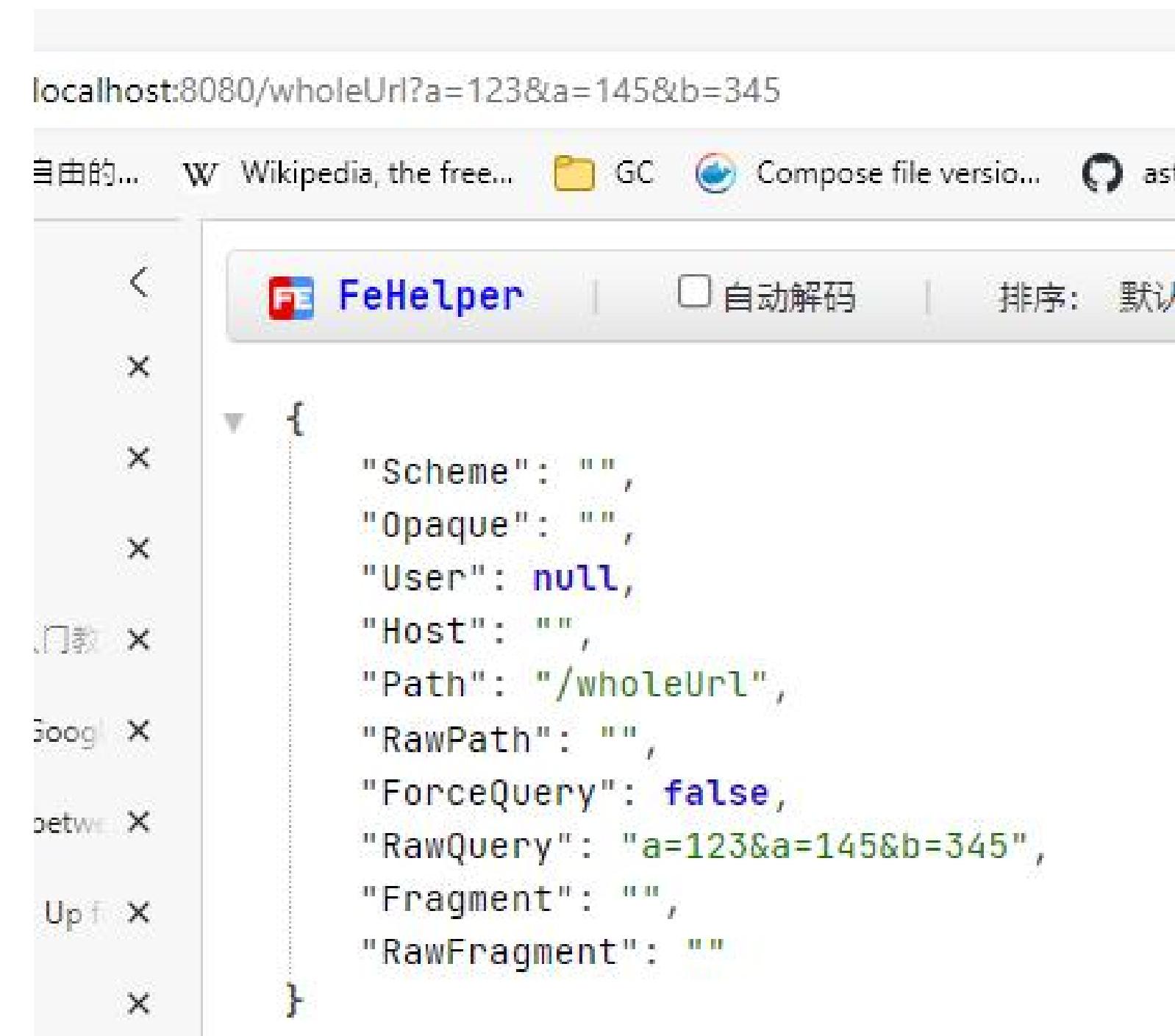
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        queryParams(w, r)
    })
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

http 库 —— Request URL

- URL 里面 Host 不一定有值
- r.Host 一般都有值，是Host这个header的值
- RawPath 也是不一定有
- Path肯定有

Tip: 实际中记得自己输出来看一下，确认有没有

```
func wholeUrl(w http.ResponseWriter, r *http.Request) {  
    data, _ := json.Marshal(r.URL)  
    fmt.Fprintf(w, string(data))  
}
```



http 库 —— Request Header

- header大体上是两类，一类是 http 预定义的；一类是自己定义的
- Go 会自动将 header 名字转为标准名字——其实就是大小写调整
- 一般用 X 开头来表明是自己定义的，比如说 X-mycompany-your=header

The screenshot shows two parts illustrating the handling of custom HTTP headers.

Top Part (Code Editor):

```
func header(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "header is %v\n", r.Header)
```

Bottom Part (Postman Request):

A Postman request configuration for "lesson2-bodyonce" is shown:

- Method: GET
- URL: localhost:8080/header
- Headers tab is selected, showing one header entry:

Key	Value
x-toy-web-app	test-app

- Body tab is selected, showing the response body:

```
1 header is map[Accept:[/*/*] Accept-Encoding:[gzip, deflate, br] Accept-Language:[zh-CN,zh;q=0.9,en;q=0.8] Cache-Control:[no-cache] Connection:[keep-alive] Postman-Token:[48ec8142-434b-3191-f9f9-841174d963d3] Sec-Ch-Ua:[\"Not:A Brand\";v=\"99\", \"Google Chrome\";v=\"91\", \"Chromium\";v=\"100\"] Sec-Ch-Ua-Mobile:[?0] Sec-Fetch-Dest:[empty] Sec-Fetch-Mode:[cors] Sec-Fetch-Site:[none] User-Agent:[Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36] X-Toy-Web-App:[test-app]]
```

http 库 —— Form

- Form 和 ParseForm
- 要先调用 ParseForm
- 建议加上 Content-Type:
application/x-www-form-urlencoded

```
func form(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprintf(w, "before parse form %v\n", r.Form)  
    err := r.ParseForm()  
    if err != nil {  
        fmt.Fprintf(w, "parse form error %v\n", r.Form)  
    }  
    fmt.Fprintf(w, "before parse form %v\n", r.Form)  
}
```

request form

Add a description

GET http://localhost:8080/form?age=23&name=zhaofan

Authorization Headers (1) Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (3) Test Results

Pretty Raw Preview Text

1 before parse form map[]
2 before parse form map[age:[23] name:[zhaofan]]
3

要点总结：http 库使用

- Body 和 GetBody: 重点在于 Body 是一次性的，而 GetBody 默认情况下是没有，一般中间件会考虑帮你注入这个方法
- URL: 注意 URL 里面的字段的含义可能并不如你期望的那样
- Form: 记得调用前先用 ParseForm，别忘了请求里面加上 http 头

如何使用 Golang Debug?



A screenshot of a Go debugger showing a stack trace. The stack consists of two functions:

- queryParams(w http.ResponseWriter, r *http.Request) {
 values := r.URL.Query()
 fmt.Fprintf(w, format: "query is %v\n", values)
}
- header(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, format: "header is %v\n", r.Header)
}

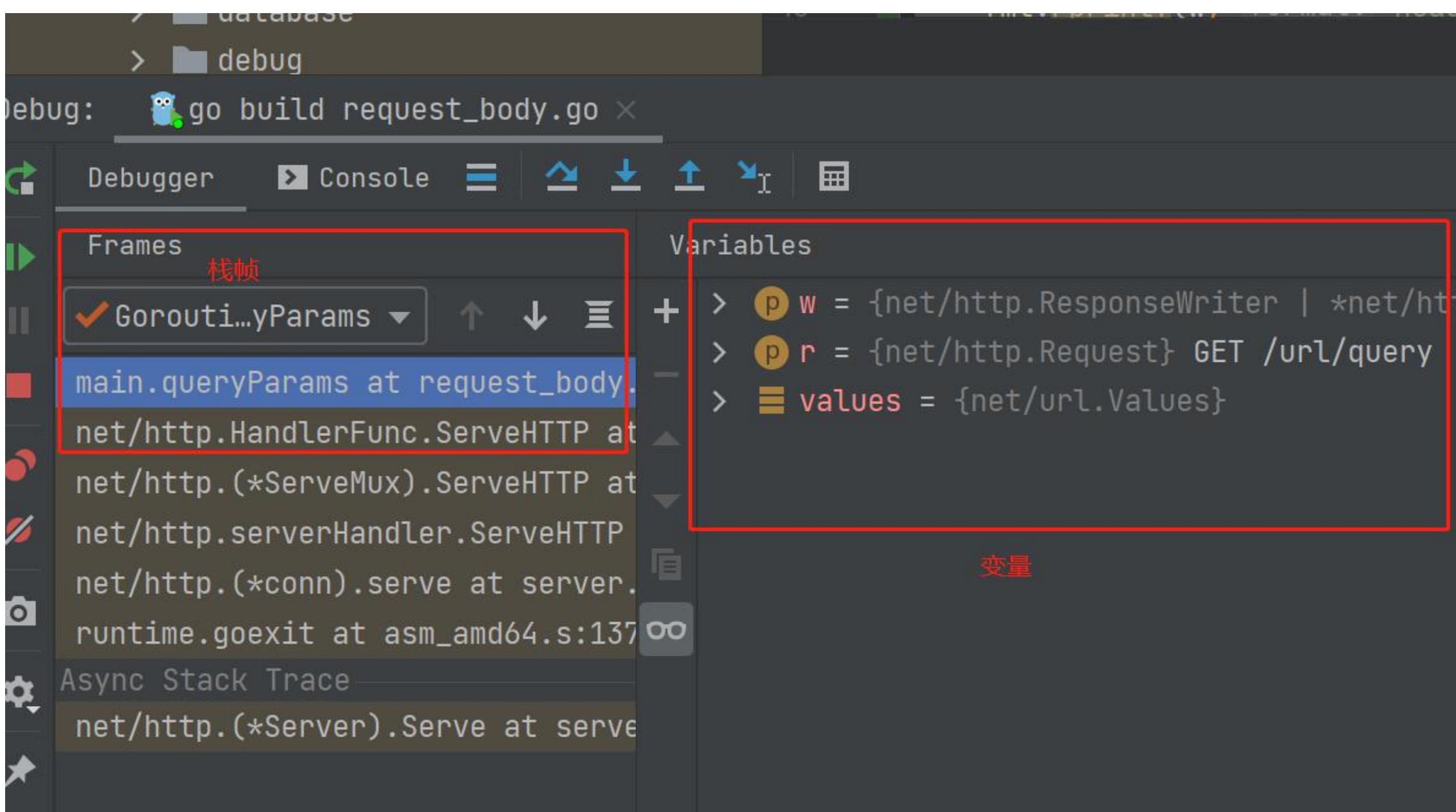
The line `fmt.Fprintf(w, format: "query is %v\n", values)` is highlighted with a red bar and has a red breakpoint icon at the start. The line `fmt.Fprintf(w, format: "header is %v\n", r.Header)` has a yellow lightbulb icon at the start.

如何使用 Golang Debug?

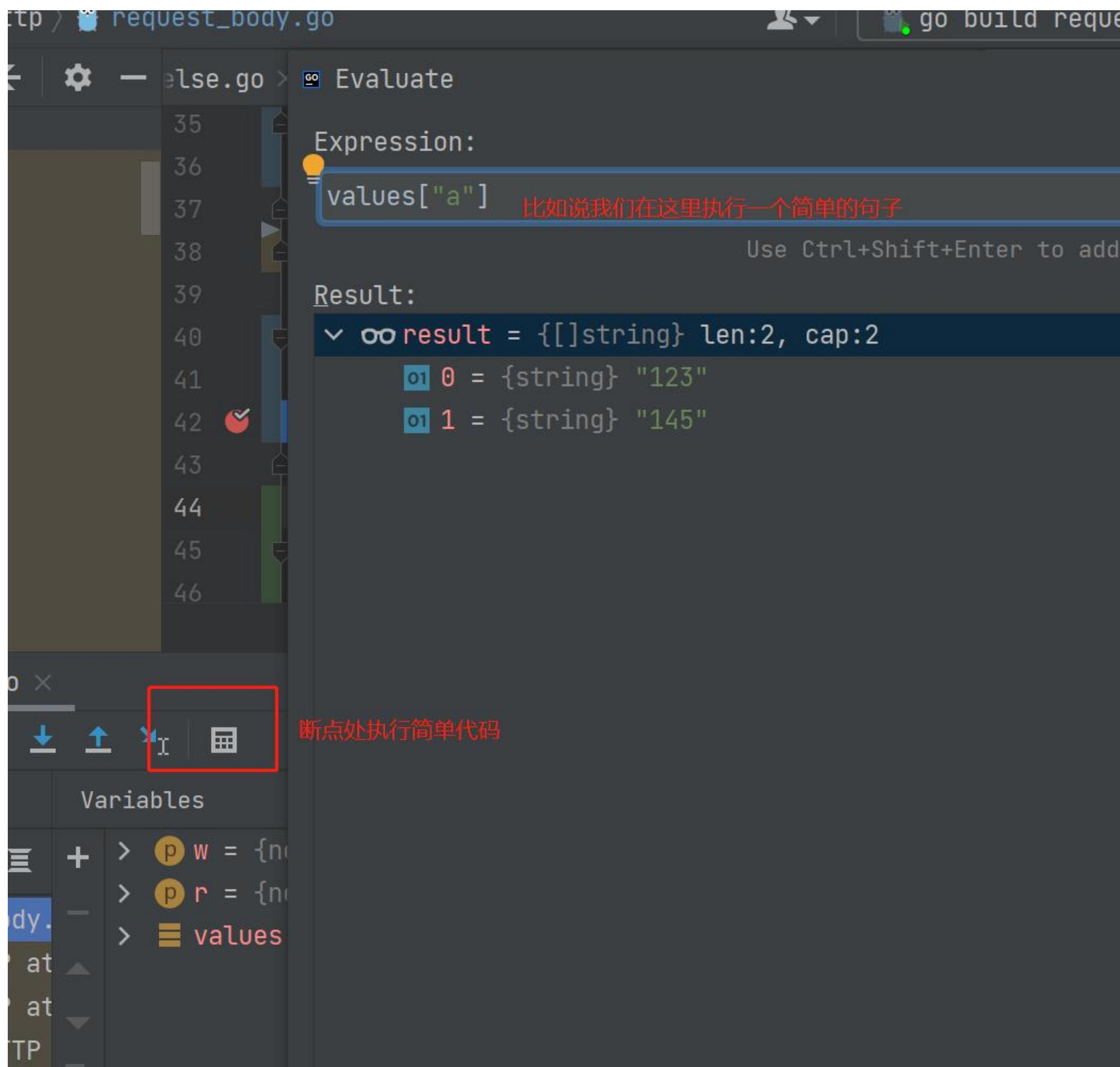


A screenshot of a Go code editor showing a function named `queryParams`. The code prints the query parameters from the URL. A blue bar highlights the line `fmt.Fprintf(w, "query is %v\n", values)`, indicating it is the current line being debugged.

```
func queryParams(w http.ResponseWriter, r *http.Request) {    r: GET /url/query    values := r.URL.Query()    values: net/url.Values    fmt.Fprintf(w, "query is %v\n", values)}
```



如何使用 Golang Debug?



基础语法 —— type 定义

- type 定义
 - type 名字 interface {}
 - type 名字 struct {}
 - type 名字 别的类型
 - type 别名 = 别的类型
- 结构体初始化
- 指针与方法接收器
- 结构体如何实现接口

从 Http Server 开始

```
▶ func main() {
    http.HandleFunc(pattern: "/", home)
    http.HandleFunc(pattern: "/user", user)
    http.HandleFunc(pattern: "/user/create", createUser)
    http.HandleFunc(pattern: "/order", order)
    // 如果我想启动两个服务器，一个监听 8080，一个监听8081，我用8081来作为管理端口
    log.Fatal(http.ListenAndServe(addr: ":8080", handler: nil))
}
```

“这个东西，缺乏一个逻辑上的联系，至少联系不够紧密”

Http Server 抽象



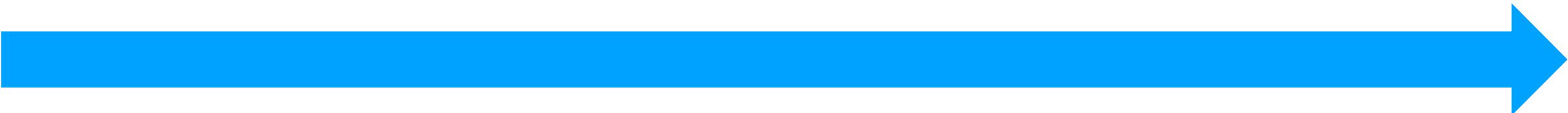
我想要一个 Server 的东西，表达一种逻辑上的抽象，它代表的是对某个端口的进行监听的实体，必要的时候，我可以开启多个 Server，来监听多个端口

Http Server 抽象 —— 接口定义

```
func main() {
    http.HandleFunc(pattern: "/", home)
    http.HandleFunc(pattern: "/user", user)
    http.HandleFunc(pattern: "/user/create", createUser)
    http.HandleFunc(pattern: "/order", order)
    // 如果我想启动两个服务器, 一个监听 8080, 一个监听8081, 我用
    log.Fatal(http.ListenAndServe(addr: ":8080", handler))
}
```

```
type Server interface {
    // Route 设定一个路由, 命中该路由的会执行handlerFunc的代码
    Route(pattern string, handlerFunc http.HandlerFunc)

    // Start 启动我们的服务器
    Start(address string) error
}
```



这两个方法直接来
源于这里

基础语法 —— interface 定义

- 基本语法 type 名字 interface {}
- 里面只能有方法，方法也不需要 func 关键字
- 啥是接口（interface）： 接口是一组行为的抽象
- 尽量用接口，以实现面向接口编程

```
type Server interface {  
    // Route 设定一个路由，命中该路由的会执行handlerFunc的代码  
    Route(pattern string, handlerFunc http.HandlerFunc)  
    // Start 启动我们的服务器  
    Start(address string) error  
}
```

Tip: 当你怀疑要不要用接口的时候，加上去总是很保险的

基础语法 —— struct 定义

- 基本语法：

```
type Name struct {  
    fieldName FieldType  
    // ...  
}
```

- 结构体和结构体的字段都遵循大小写控制访问性的原则

```
// sdkHttpServer 这个是基于 net/http 这个包实现的 http server  
type sdkHttpServer struct {  
    // Name server 的名字，给个标记，日志输出的时候用得上  
    Name string  
}
```

Tip: 其实还有别的第三方 http 库，也可以用来实现一个 server

基础语法 —— type A B

- 基本语法: type TypeA TypeB
- 我一般是，在我使用第三方库又没有办法修改源码的情况下，又想在扩展这个库的结构体的方法，就会用这个

```
type Fish struct {
}

func (f Fish) Swim() {
    fmt.Printf("我是鱼，假装自己是一直鸭子\n")
}
```

```
// 定义了一个新类型，注意是新类型
type FakeFish Fish

func (f FakeFish) FakeSwim() {
    fmt.Printf("我是山寨鱼，嘎嘎嘎\n")
}

// 定义了一个新类型
type StrongFakeFish Fish

func (f StrongFakeFish) Swim() {
    fmt.Printf("我是华强北山寨鱼，嘎嘎嘎\n")
}
```

基础语法 —— type A B

```
fake := FakeFish{}  
// fake 无法调用原来 Fish 的方法  
// 这一句会编译错误  
// fake.Swim()  
fake.FakeSwim()
```

 // 转换为Fish
td := Fish(fake)
// 真的变成了鱼
td.Swim()

```
sFake := StrongFakeFish{}  
// 这里就是调用自己的方法  
sFake.Swim()
```

```
td = Fish(sFake)  
// 真的变成了鱼  
td.Swim()
```

Tip: 这个不用记，属于那种看上去很复杂，但是实际你根本不会这么写的东西。

基础语法 —— type A = B

- 基本语法: type TypeA = TypeB
- 别名, 除了换了一个名字, 没有任何区别

```
▶ ┌ func main() {  
  └ var n News = fakeNews{  
    └ Name: "hello",  
  }  
  └ n.Report()  
} ┌  
  └ type News struct {  
    └ Name string  
} ┌  
  └ func (d News) Report() {  
    └ fmt.Println("I am news: " + d.Name)  
} ┌  
  └ type fakeNews = News
```

基础语法 —— type 定义

- type 定义
 - type 名字 interface {}
 - type 名字 struct {}
 - type 名字 别的类型
 - type 别名 = 别的类型
- 结构体初始化
- 指针与方法接收器
- 结构体如何实现接口

基础语法 —— 初始化

- Go 没有构造函数！！
- 初始化语法: Struct {}
- 获取指针: &Struct {}
- 获取指针2: new(Struct)
- new 可以理解为 Go 会为你的变量分配内存，并且把内存都置为 0

```
duck1 := &ToyDuck{}
```

```
duck1.Swim()
```

```
duck2 := ToyDuck{}
```

```
duck2.Swim()
```

```
duck3 := new(ToyDuck)
```

```
duck3.Swim()
```

基础语法 —— 初始化

```
duck1 := &ToyDuck{}  
duck1.Swim()
```

```
duck2 := ToyDuck{}  
duck2.Swim()
```

```
duck3 := new(ToyDuck)  
duck3.Swim()
```

推荐写法

```
// 当你声明这样的时候，Go 就帮你分配好内存  
// 不用担心空指针的问题，以为它压根就不是指针  
var duck4 ToyDuck  
duck4.Swim()  
  
// duck5 就是一个指针了  
//  
var duck5 *ToyDuck  
// 这边会直接panic 掉  
duck5.Swim()
```

基础语法 —— 字段赋值

```
// 赋值，初始化按字段名字赋值
duck6 := ToyDuck{
    Color: "黄色",
    Price: 100,
}
duck6.Swim()

// 初始化按字段顺序赋值，不建议使用
duck7 := ToyDuck{ Color: "蓝色", Price: 1024}
duck7.Swim()

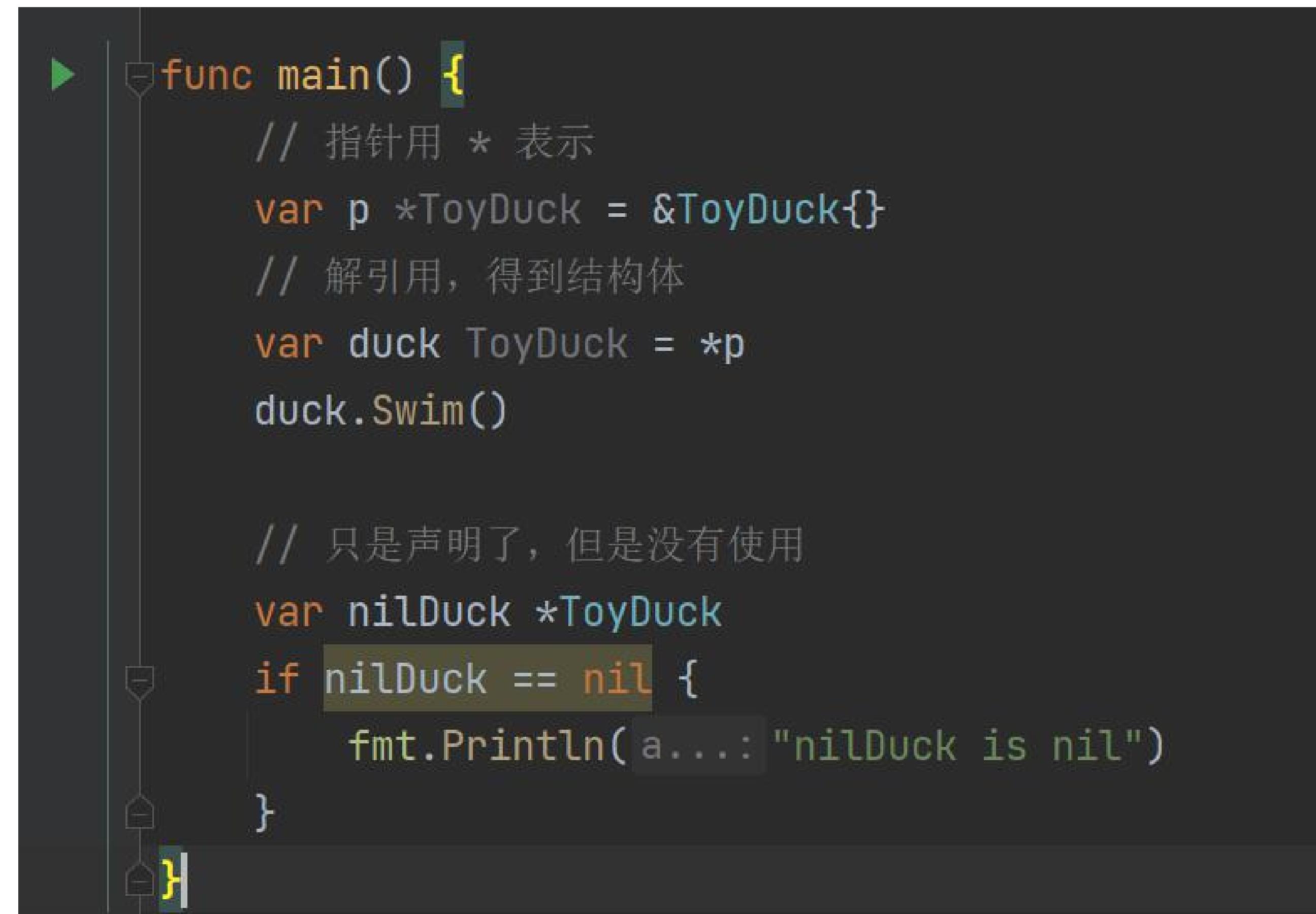
💡 // 后面再单独赋值
duck8 := ToyDuck{}
duck8.Color = "橘色"
```

基础语法 —— type 定义

- type 定义
 - type 名字 interface {}
 - type 名字 struct {}
 - type 名字 别的类型
 - type 别名 = 别的类型
- 结构体初始化
- 指针与方法接收器
- 结构体如何实现接口

基础语法 —— 指针

- 和 C, C++ 一样, *表示指针, &取地址
- 如果声明了一个指针, 但是没有赋值, 那么它是 nil



```
▶ func main() {  
    // 指针用 * 表示  
    var p *ToyDuck = &ToyDuck{}  
    // 解引用, 得到结构体  
    var duck ToyDuck = *p  
    duck.Swim()  
  
    // 只是声明了, 但是没有使用  
    var nilDuck *ToyDuck  
    if nilDuck == nil {  
        fmt.Println("nilDuck is nil")  
    }  
}
```

The screenshot shows a code editor window with a dark theme. It displays a Go program. The first part of the code creates a pointer variable `p` pointing to a `ToyDuck` struct. It then dereferences `p` to get the `duck` variable, which is then used to call the `Swim()` method. The second part of the code declares a pointer variable `nilDuck` but does not assign it to anything. An `if` statement checks if `nilDuck` is equal to `nil`, and if so, it prints the message "nilDuck is nil". The code editor has syntax highlighting and includes icons for file, folder, and search operations.

基础语法 —— 结构体自引用

- 结构体内部引用自己，只能使用指针
- 准确来说，在整个引用链上，如果构成循环，那就只能用指针

```
type Node struct {
    // 自引用只能使用指针
    // left Node
    // right Node
    left *Node
    right *Node
}

// 这个也会报错
// nn NodeNode
}

type NodeNode struct {
    node Node
}
```

基础语法 —— 方法接收器

- 结构体接收器内部永远不要修改字段

```
type User struct {
    Name string
    Age int
}

// 结构体接收器
func (u User) ChangeName(newName string) {
    u.Name = newName
}

// 指针接收器
func (u *User) ChangeAge(newAge int) {
    u.Age = newAge
}
```

基础语法 —— 方法接收器

```
// 因为 u 是结构体，所以方法调用的时候它数据是不会变的
u := User{
    Name: "Tom",
    Age: 10,
}
u.ChangeName(newName: "Tom Changed!")
u.ChangeAge(newAge: 100)
fmt.Printf(format: "%v \n", u)
```

```
// 因为 up 指针，所以内部的数据是可以被改变的
up := &User{
    Name: "Jerry",
    Age: 12,
}
// 因为 ChangeName 的接收器是结构体
// 所以 up 的数据还是会变
up.ChangeName(newName: "Jerry Changed!")
up.ChangeAge(newAge: 120)
fmt.Printf(format: "%v \n", up)
```

Tip: 结构体和指针之间的方法可以互相调用

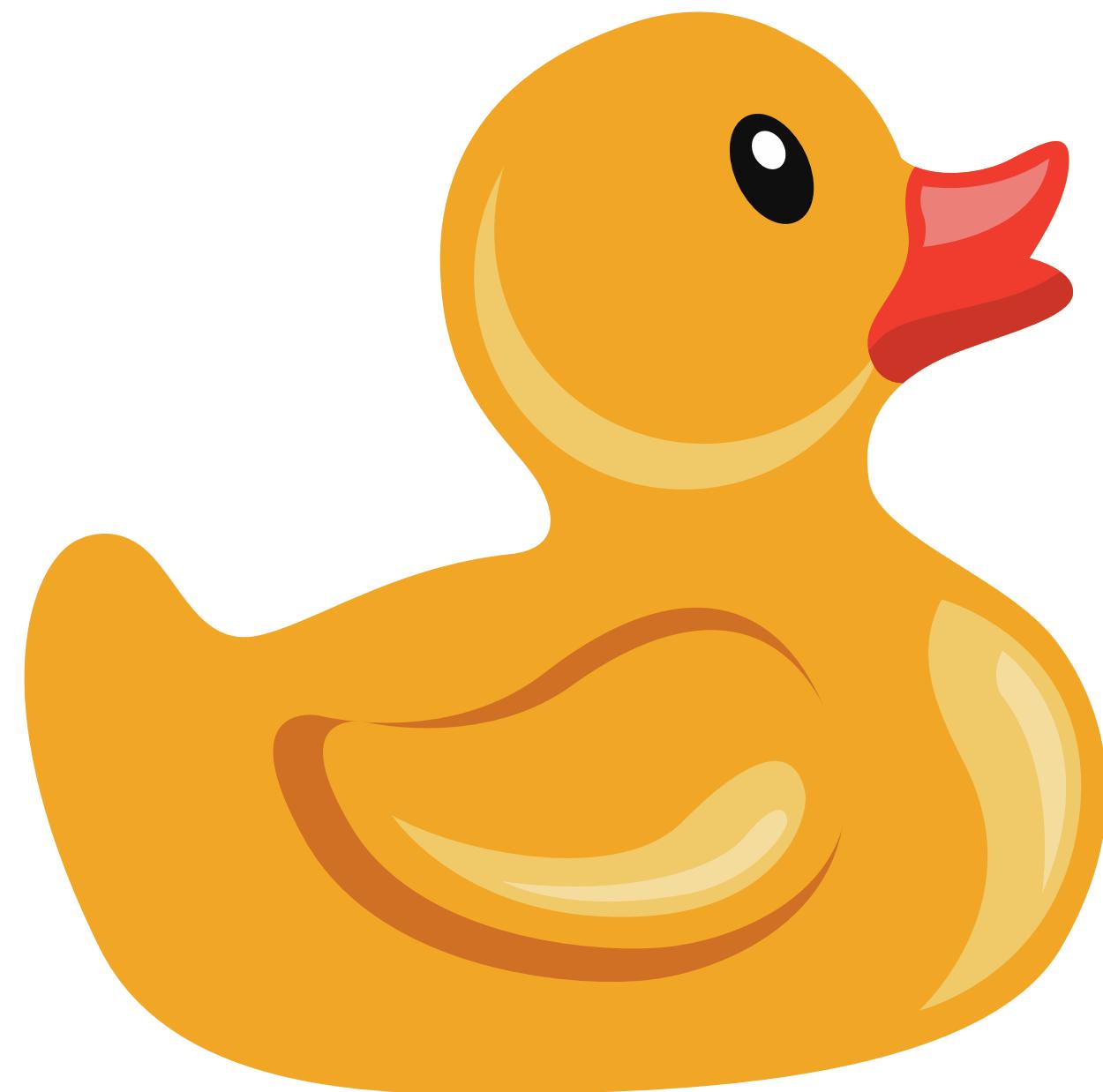
基础语法 —— 方法接收器用哪个？

- 设计不可变对象，用结构体接收器
- 其它用指针
- 总结：遇事不决用指针

基础语法 —— type 定义

- type 定义
 - type 名字 interface {}
 - type 名字 struct {}
 - type 名字 别的类型
 - type 别名 = 别的类型
- 结构体初始化
- 指针与方法接收器
- 结构体如何实现接口

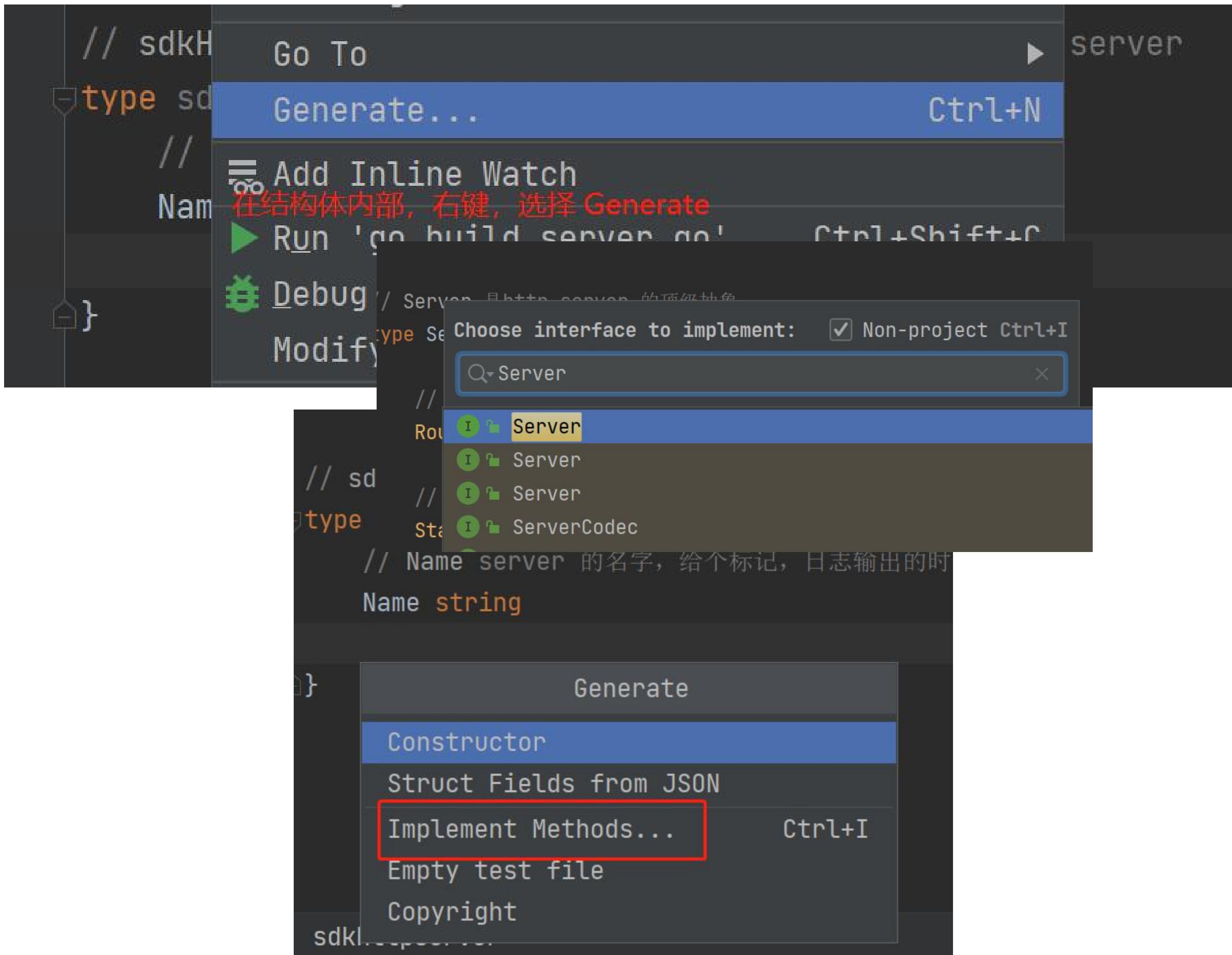
基础语法 —— 结构体如何实现接口？



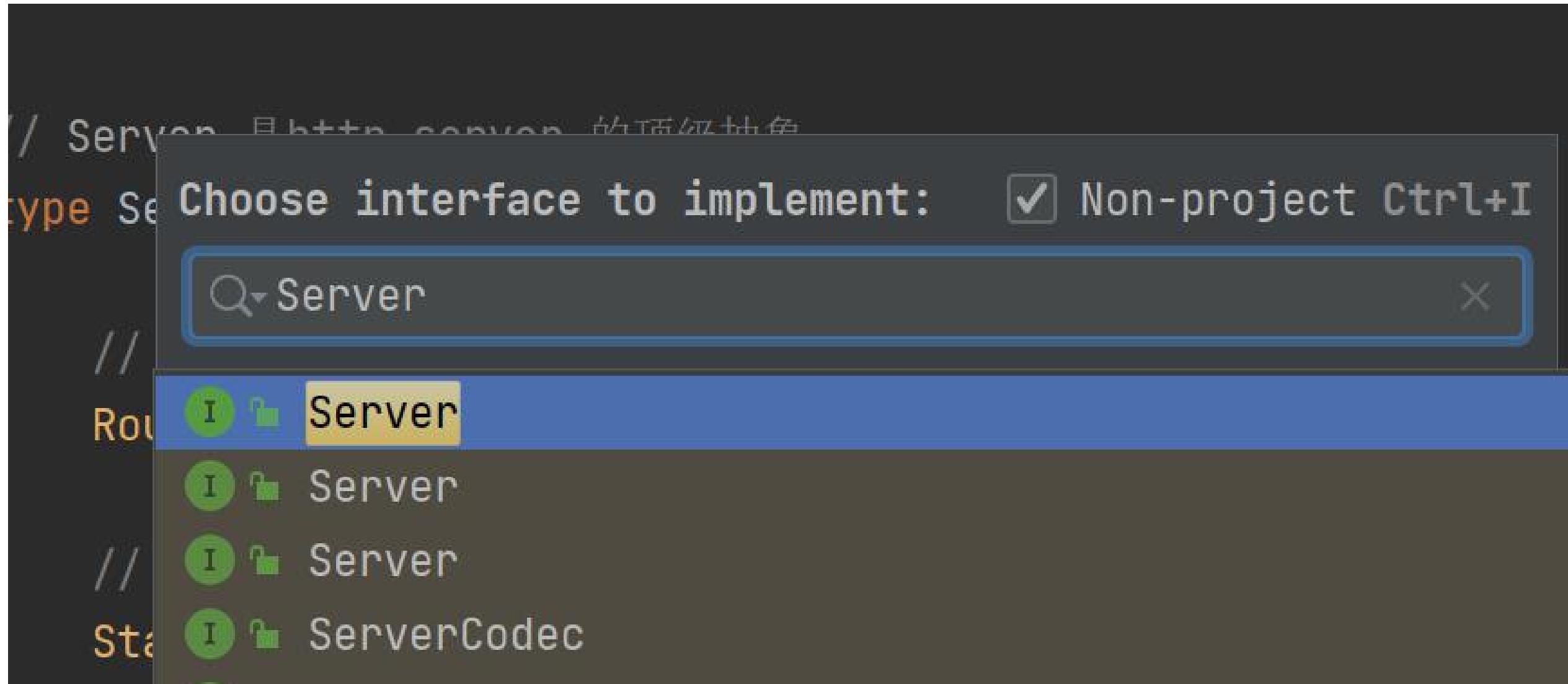
当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。

当一个结构体具备接口的所有方法的时候，它就实现了这个接口

基础语法 —— 结构体如何实现接口？



基础语法 —— 结构体如何实现接口？



```
func (s sdkHttpServer) Route(pattern string) {
    panic(sdkHttpServer) // 结构体接收器
}

func (s *sdkHttpServer) Start(address string) {
    panic("implement me") // 指针接收器
}
```

The image shows a code editor with two function definitions. The first function, "Route", takes a string parameter and returns a value of type "sdkHttpServer". A tooltip above it says "结构体接收器" (Struct Receiver). The second function, "Start", takes a string parameter and returns a value of type "*sdkHttpServer". A tooltip above it says "指针接收器" (Pointer Receiver). Both functions contain a "panic" statement with a placeholder message.

基础语法 —— 结构体如何实现接口？

```
7  
8 // sdkHttpServer 这个是基于 net/http 这个包实现的 http server  
9 type sdkHttpServer struct {  
10    // Name server 的名字，给个标记，日志输出的时候用得上  
11    Name string  
12 }  
13  
14 func (s *sdkHttpServer) Route(pattern string, handlerFunc http.HandlerFunc) {  
15    panic("implement me")  
16 }  
17  
18 func (s *sdkHttpServer) Start(address string) error {  
19    panic("implement me")  
20 }
```

基础语法 —— 注释规范

- 以被注释的开头，后面跟着描述

```
// Server 是http server 的顶级抽象
type Server interface {

    // Route 设定一个路由，命中该路由的会执行handlerFunc的代码
    Route(pattern string, handlerFunc http.HandlerFunc)

    // Start 启动我们的服务器
    Start(address string) error
}

}
```

```
// Route 设定一个路由，命中该路由的会执行handlerFunc的代码
Route(pattern string, handlerFunc http.HandlerFunc)
func (Server) Route(pattern string, handlerFunc http.HandlerFunc)
// Route 设定一个路由，命中该路由的会执行handlerFunc的代码
Start(`Route` on pkg.go.dev`)
:
```

要点总结： type

- type 定义熟记。其中 type A=B 这种别名，一般只用于兼容性处理，所以不需要过多关注；
 - 先有抽象再有实现，所以要先定义接口
- 鸭子类型：一个结构体有某个接口的所有方法，它就实现了这个接口；
- 指针：方法接收器，遇事不决用指针；

Http Server —— Server 和 Context

- Http Server 实现
- Context抽象与实现
 - 读取数据
 - 写入响应
- 创建 Context

Http Server 实现

```
func (s *sdkHttpServer) Route(pattern string, h http.HandlerFunc) {
    s.routes[routeKey(pattern)] = h
}

func (s *sdkHttpServer) Start(address string) error {
    return http.ListenAndServe(address, s)
}

func NewSdkHttpServer(name string) Server {
    return &sdkHttpServer{
        Name: name,
    }
}
```

```
func main() {
    server := web.NewSdkHttpServer(name: "my-test-server")

    // 注册路由
    server.Route(pattern: "/", home)
    server.Route(pattern: "/user", user)
    server.Route(pattern: "/user/create", createUser)
    server.Route(pattern: "/order", order)

    server.Start(address: "8080")
}
```

Http Server —— 用这个实现一下用户注册

```
type signUpReq struct {  
    Email string `json:"email"  
    Password string `json:"password"  
    ConfirmedPassword string `json:"confirmed_password"  
}
```

```
func SignUp(w http.ResponseWriter, r *http.Request) {  
    req := &signUpReq{}  
    body, err := io.ReadAll(r.Body)  
    if err != nil {  
        fmt.Fprintf(w, "read body failed: %v", err)  
        // 要返回掉，不然就会继续执行后面的代码  
        return  
    }  
    err = json.Unmarshal(body, req)  
    if err != nil {  
        fmt.Fprintf(w, "deserialized failed: %v", err)  
        // 要返回掉，不然就会继续执行后面的代码  
        return  
    }  
    // 返回一个虚拟的 user id 表示注册成功了  
    fmt.Fprintf(w, "%d", err)  
}
```

Http Server —— Server 和 Context

- Http Server 实现
- Context抽象与实现
 - 读取数据
 - 写入响应
- 创建 Context

Http Context—— Context 抽象

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    req := &signUpReq{}
    body, err := io.ReadAll(r.Body)
    if err != nil {
        fmt.Fprintf(w, format: "read body failed: %v", err)
        // 要返回掉, 不然就会继续执行后面的代码
        return
    }  
    这一块代码, 但凡你要读json输入, 就得来一遍
    err = json.Unmarshal(body, req)
    if err != nil {
        fmt.Fprintf(w, format: "deserialized failed: %v", err)
        // 要返回掉, 不然就会继续执行后面的代码
        return
    }
    // 返回一个虚拟的 user id 表示注册成功了
    fmt.Fprintf(w, format: "%d", err)
}
```

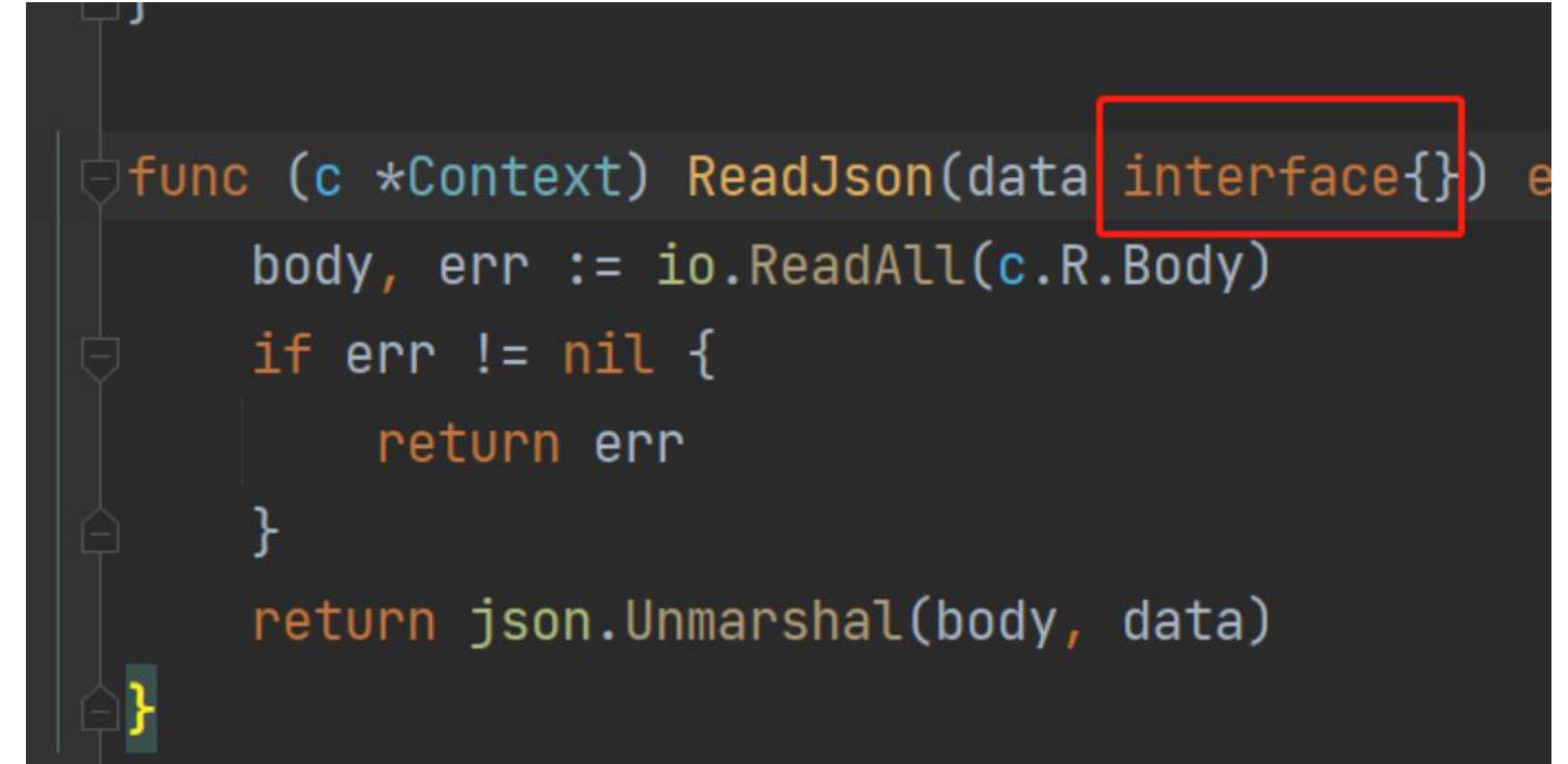
```
type Context struct {
    W http.ResponseWriter
    R *http.Request
}

func (c *Context) ReadJson(data interface{}) error {
    body, err := io.ReadAll(c.R.Body)
    if err != nil {
        return err
    }
    return json.Unmarshal(body, data)
}
```



基础语法 —— 空接口 interface {}

- 空接口 interface {} 不包含任何方法
- 所以任何结构体都实现了该接口
- 类似于 Java 的 Object，即所谓的继承树根节点



```
func (c *Context) ReadJson(data interface{}) error {
    body, err := io.ReadAll(c.R.Body)
    if err != nil {
        return err
    }
    return json.Unmarshal(body, data)
}
```

基础语法 —— json 库

- 用于处理 json 格式的字符串
- 字段后面的内容被称为 Tag，即标签，运行期间可以反射拿到
- json库依据 json Tag 的内容来完成json数据到结构体的映射
- 典型的声明式API设计

Tip: 利用Goland提示来查看 json 库有哪些方法

```
func (c *Context) ReadJson(data interface{}) error {
    body, err := io.ReadAll(c.R.Body)
    if err != nil {
        return err
    }
    return json.Unmarshal(body, data)
}
```

```
type signUpReq struct {
    Email string `json:"email"`
    Password string `json:"password"`
    ConfirmedPassword string `json:"confirmed_password"`
}
```

Http Server —— Server 和 Context

- Http Server 实现
- Context抽象与实现
 - 读取数据
 - 写入响应
- 创建 Context

Http Server —— 写入响应

- 强耦合 fmt 库
- 难以输出格式化数据，比如说返回一个 json 数据给客户端
- 没有处理 http 响应码

```
func SignUp(w http.ResponseWriter, r *http.Request) {  
    c := web.NewContext(w, r)  
    req := &signUpReq{}  
    err := c.ReadJson(req)  
    if err != nil {  
        fmt.Fprintf(w, "invalid request: %v", err)  
        return  
    }  
    fmt.Fprintf(w, "invalid request: %v", err)  
}
```

Http Server —— 写入响应

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    c := web.NewContext(w, r)
    req := &signUpReq{}
    err := c.ReadJson(req)
    if err != nil {
        resp := &commonResponse{
            BizCode: 4, // 假如说我们这个代表输入参数错误
            Msg: fmt.Sprintf(format: "invalid request: %v", err),
        }
        respBytes, _ := json.Marshal(resp)
        fmt.Fprint(w, string(respBytes))
        return
    }
    // 这里又得来一遍 resp 转json
    fmt.Fprintf(w, format: "invalid request: %v", err)
}
```

Http Server —— 写入响应

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    c := web.NewContext(w, r)
    req := &signUpReq{}
    err := c.ReadJson(req)

    if err != nil {
        resp := &commonResponse{
            BizCode: 4, // 假如说我们这个代表输入参数错误
            Msg: fmt.Sprintf(format: "invalid request: %v", err),
        }
        respBytes, _ := json.Marshal(resp)
        fmt.Fprint(w, string(respBytes))
        return
    }

    // 这里又得来一遍 resp 转json
    fmt.Fprintf(w, format: "invalid request: %v", err)
}
```

```
func (c *Context) WriteJson(status int, data interface{}) error {
    bs, err := json.Marshal(data)
    if err != nil : err ↗
    _, err = c.W.Write(bs)
    if err != nil {
        return err
    }
    c.W.WriteHeader(status)
    return nil
}
```

这里有个小差异，是我们不再使用 `fmt`，而是直接使用 `Write` 方法

Http Server —— 进一步封装

- 提供辅助方法
- 注意！它不是 Context 本身必须要提供的方法！
即如果你在设计真实的 web 框架的时候，你需要考虑清楚，究竟要不要提供这种辅助方法

```
func (c *Context) OkJson(data interface{}) error {
    // http 库里面提前定义好了各种响应码
    return c.WriteJson(http.StatusOK, data)
}

func (c *Context) SystemErrJson(data interface{}) error {
    // http 库里面提前定义好了各种响应码
    return c.WriteJson(http.StatusInternalServerError, data)
}

func (c *Context) BadRequestJson(data interface{}) error {
    // http 库里面提前定义好了各种响应码
    return c.WriteJson(http.StatusBadRequest, data)
}
```

Tip: 严格来说, WriteJson也是辅助方法

Http Context—— 对比

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    c := web.NewContext(w, r)
    req := &signUpReq{}
    err := c.ReadJson(req)

    if err != nil {
        resp := &commonResponse{
            BizCode: 4, // 假如说我们这个代表输入参数错误
            Msg: fmt.Sprintf(format: "invalid request: %v", err),
        }
        respBytes, _ := json.Marshal(resp)
        fmt.Fprint(w, string(respBytes))
        return
    }

    // 这里又得来一遍 resp 转json
    fmt.Fprintf(w, format: "invalid request: %v", err)
}
```

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    c := web.NewContext(w, r)
    req := &signUpReq{}
    err := c.ReadJson(req)

    if err != nil {
        _ = c.BadRequestJson(&commonResponse{
            BizCode: 4, // 假如说我们这个代表输入参数错误
            // 注意这里是demo，实际中你应该避免暴露 error
            Msg: fmt.Sprintf(format: "invalid request: %v", err),
        })
        return
    }

    _ = c.BadRequestJson(&commonResponse{
        // 假设这个是新用户的 ID
        Data: 123,
    })
}
```

Http Server —— Server 和 Context

- Http Server 实现
- Context抽象与实现
 - 读取数据
 - 写入响应
- 创建 Context

Http Context—— 让 web 框架来创建 context

```
func SignUp(w http.ResponseWriter, r *http.Request) {
    c := web.NewContext(w, r)
    req := &signUpReq{}
    err := c.ReadJson(req)
    if err != nil {
        _ = c.BadRequestJson(&commonResponse{
            BizCode: 4, // 假如说我们这个值
            // 注意这里是demo，实际中你应该
            Msg: fmt.Sprintf(format: "in
        })
        return
    }
    _ = c.BadRequestJson(&commonResponse{
        // 假设这个是新用户的 ID
        Data: 123,
    })
}
```

```
func SignUp(c *web.Context) {
    req := &signUpReq{}
    err := c.ReadJson(req)
}

func main() {
    server := web.NewSdkHttpServer(name: "my-test-server")
    n(&commonResponse{
        // 假如说我们这个代表输入参数错误
        //，实际中你应该避免暴露 error
        f(format: "invalid request: %v", err),
        commonResponse{
            ID
            Data: 123,
        }
    })
    server.Route(pattern: "/", home)
    server.Route(pattern: "/user", user)
    server.Route(pattern: "/user/create", demo.SignUp)
    server.Route(pattern: "/order", order)
    server.Start(address: ":8080")
}
```

框架来创建context，就可以完全控制什么时候创建，context可以有什么字段。作为设计者，这种东西不能交给用户自由发挥。

Http Context—— 让 web 框架来创建 context

The screenshot shows a code editor with two panes. The left pane contains a Go function named `SignUp`:

```
func SignUp(c *web.Context) {  
    req := &signUpReq{}  
    err := c.ReadJson(req)  
    if err != nil {  
        _ = c.BadRequestJson(&commonRespo  
        BizCode: 4, // 假如说我们这个代  
        // 注意这里是demo，实际中你应该避  
        Msg: fmt.Sprintf(format: "inv  
    })  
    return  
}  
_ = c.BadRequestJson(&commonResponse{  
    // 假设这个是新用户的 ID  
    Data: 123,  
})
```

The right pane shows server routing code:

```
// 注册路由  
//server.Route("/", home)  
//server.Route("/user", user)  
server.Route(pattern: "/user/create", demo.SignUp)  
//server.Route("/order", order)
```

Annotations: The `c` parameter in the `SignUp` function is highlighted with a red box. The `demo.SignUp` method call in the routing code is also highlighted with a red box.

原本的 Router 方法已经不行了，需要改造

Http Server 改造

```
// Server 是http server 的顶级抽象
type Server interface {

    // Route 设定一个路由，命中该路由的会执行handlerFunc的代码
    Route(pattern string, handlerFunc func(c *Context))

}

func (s *sdkHttpServer) Route(pattern string, handlerFunc func(c *Context)) {
    http.HandleFunc(pattern, func(writer http.ResponseWriter, request *http.Request) {
        c := NewContext(writer, request)
        handlerFunc(c)
    })
}
```

要点总结：Server 和 Context

- 从 http.Request 中读取数据并解析
- 往 http.ResponseWriter 中写入数据和响应
- json 数据的序列化与反序列化

设计是一个循序渐进，逐步迭代，螺旋上升的过程。

Http Server —— 支持 RESTFul API

- RESTFul API 定义
- 路由设计 —— Handler 抽象
 - map 语法
 - 基于 map 的 Handler 实现
- 语法：组合
- 重构

Http Server —— RESTful API 定义

应用于Web服务 [编辑]

符合REST设计风格的Web API称为**RESTful API**。它从以下三个方面资源进行定义：

- 直观简短的资源地址：URI，比如：`http://example.com/resources`。
- 传输的资源：Web服务接受与返回的互联网媒体类型，比如：[JSON](#), [XML](#), [YAML](#)等。
- 对资源的操作：Web服务在该资源上所支持的一系列请求方法（比如：[POST](#), [GET](#), [PUT](#)或[DELETE](#)）。

下表列出了在实现RESTful API时HTTP请求方法的典型用途。

HTTP请求方法在RESTful API中的典型应用^[3]

资源	GET	PUT
一组资源的URI，比如 <code>https://example.com/resources</code>	列出URI，以及该资源组中每个资源的详细信息（后者可选）。	使用给定的一组资源替换当前整组资源。
单个资源的URI，比如 <code>https://example.com/resources/142</code>	获取指定的资源的详细信息，格式可以自选一个合适的网络媒体类型（比如：XML、JSON等）	替换/创建指定的资源。并将其追加到相应的资源组中。

PUT和DELETE方法是幂等方法。GET方法是安全方法（不会对服务器端有修改，因此当然也是幂等的）。

不像基于[SOAP](#)的Web服务，RESTful Web服务并没有“正式”的标准^[4]。这是因为REST是一种架构，而SOAP只是一个协议。虽然REST不是一个标准，但大部分RE

简单来说，就是http method 决定了操作，http path 决定了操作对象

Http Server —— 如何支持 RESTful API

```
// PUT /user 创建用户  
// POST /user 更新用户  
// DELETE /user 删除用户  
💡 // GET /user 获取用户
```

http method + http path = http handler

Http Server —— 如何支持 RESTful API

```
// Server 是http server 的顶级抽象
①↓ type Server interface {

    // Route 设定一个路由，命中该路由的会执行handlerFunc的代码
    ②↓ Route(method string, pattern string, handlerFunc func(c *Context))

    // Start 启动我们的服务器
    ③↓ Start(address string) error
}

func (s *sdkHttpServer) Route(method string, pattern string, handlerFunc func(c *Context)) {
    http.HandleFunc(pattern, func(writer http.ResponseWriter, request *http.Request) {
        c := NewContext(writer, request)
        handlerFunc(c)
    })
}
```

④↑ http.HandleFunc 好像不太行，我们得自己做路由了

Http Server —— 支持 RESTFul API

- RESTFul API 定义
- 路由设计 —— Handler 抽象
 - map 语法
 - 基于 map 的 Handler 实现
- 语法：组合
- 重构

Http Server —— Handler 抽象

- 实现一个 Handler，它负责路由
- 如果找到了路由，就执行业务代码
- 找不到就返回 404

```
// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is used
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

Http Server —— 如何路由？

- 尝试用 map 写一个最简单的版本

```
type Handler struct {  
}  
  
func (h *Handler) ServeHTTP(writer http.ResponseWriter, request *http.Request) {  
    // 分发路由  
    if found {  
        do  
    } else {  
        404  
    }  
}
```

Http Server —— 如何路由？

- 尝试用 map 写一个最简单的版本

```
type Handler struct {  
}  
  
func (h *Handler) ServeHTTP(writer http.ResponseWriter, request *http.Request) {  
    // 分发路由  
    if found {  
        do  
    } else {  
        404  
    }  
}
```

基础语法 —— map

- 基本形式: map[KeyType]ValueType
- 创建 make 命令, 或者直接初始化
- 取值: val, ok := m[key]
- 设值: m[key]=val
- key 类型: “可比较” 类型

Tip: 编译器会告诉你能不能做 key

Tip: 尽量用基本类型和string做key, 不要和自己过不去

```
// 创建了一个预估容量是2的 map
m := make(map[string]string, 2)
// 没有指定预估容量
m1 := make(map[string]string)
// 直接初始化
m2 := map[string]string{
    "Tom": "Jerry",
    // 赋值
    m2["hello"] = "world"
    // 取值
    val := m["hello"]
    println(val)
    // 再次取值, 使用两个返回值, 后面的ok会告诉你map有没有这个key
    val, ok := m["invalid_key"]
    if !ok {
        println(args...: "key not found")
    }
}
```

基础语法 —— map 遍历

- for key, val := range m {}
- Go 一个 for 打天下
- Go 的 map 的遍历，顺序是不定的

```
for key, val := range m {
    fmt.Printf(format: "%s => %s \n", key, val)
}
```

Http Server —— 基于 map 的路由

```
type HandlerBasedOnMap struct {
    handlers map[string]func(c *Context)
}

func (h *HandlerBasedOnMap) ServeHTTP(writer http.ResponseWriter,
    request *http.Request) {
    key := h.key(request.Method, request.URL.Path)
    if handler, ok := h.handlers[key]; ok {
        c := NewContext(writer, request)
        handler(c)
    } else {
        writer.WriteHeader(http.StatusNotFound)
        _, _ = writer.Write([]byte("not any router match"))
    }
}

func (h *HandlerBasedOnMap) key(method string,
    path string) string {
    return fmt.Sprintf(format: "%s#%s", method, path)
}
```

Http Server —— 基于 map 的路由

```
// sdkHttpServer 这个是基于 net/http 这个包实现的 http server
type sdkHttpServer struct {
    // Name server 的名字, 给个标记, 日志输出的时候用得上
    Name string
    handler *HandlerBasedOnMap
}

func (s *sdkHttpServer) Route(method string, pattern string,
    handlerFunc func(c *Context)) {
    key := s.handler.key(method, pattern)
    s.handler.handlers[key] = handlerFunc
}

func (s *sdkHttpServer) Start(address string) error {
    return http.ListenAndServe(address, s.handler)
}
```

这种实现有什么缺点？

Http Server —— 基于 map 的路由

- 和实现 HandlerBasedOnMap 强耦合
- Route 方法依赖于知道 HandlerBasedOnMap 的内部细节
- 当我们想要扩展到利用路有树来实现的时候，sdkHttpServer 也要修改

```
// sdkHttpServer 这个是基于 net/http 这个包实现的 http server
type sdkHttpServer struct {
    // Name server 的名字, 给个标记, 日志输出的时候用得上
    Name string
    handler *HandlerBasedOnMap
}

func (s *sdkHttpServer) Route(method string, pattern string,
    handlerFunc func(c *Context)) {
    key := s.handler.key(method, pattern)
    s.handler.handlers[key] = handlerFunc
}

func (s *sdkHttpServer) Start(address string) error {
    return http.ListenAndServe(address, s.handler)
}
```

Http Server —— 支持 RESTFul API

- RESTFul API 定义
- 路由设计 —— Handler 抽象
 - map 语法
 - 基于 map 的 Handler 实现
- 语法：组合
- 重构

Http Server —— Handler 抽象

- 我们给 HandlerBasedOnMap 加一个方法：Route
- 我们希望 sdkHttpServer 依赖于一个接口，所以我们定义一个自己的接口

```
type Handler interface {
    http.Handler
    Route(method string, pattern string, handlerFunc func(c *Context))
}
```

```
// sdkHttpServer 这个是基于 net/http 这个包实现的 http server
type sdkHttpServer struct {
    // Name server 的名字，给个标记，日志输出的时候用得上
    Name string
    handler Handler 依赖于接口
}

func (s *sdkHttpServer) Route(method string, pattern string,
    handlerFunc func(c *Context)) {
    s.handler.Route(method, pattern, handlerFunc)
}
```

Http Server —— 组合

- 组合可以是接口组合，也可以是结构体组合。结构体也可以组合接口

```
// Swimming 会游泳的
type Swimming interface {
    Swim()
}

type Duck interface {
    // 鸭子是会游泳的，所以这里组合了它
    Swimming
}
```

```
type Base struct {
    Name string
}

type Concrete1 struct {
    Base
}

type Concrete2 struct {
    *Base
}

func (b *Base) SayHello() {
    fmt.Printf("I am base and my name is: %s \n", b.Name)
}
```

Http Server —— 组合

- 组合可以是接口组合，也可以是结构体组合。结构体也可以组合接口

```
func (c Concrete1) SayHello() {  
    // c.Name 直接访问了Base的Name字段  
    fmt.Printf(format: "I am base and my name is: %s \n", c.Name)  
    // 这样也是可以的  
    fmt.Printf(format: "I am base and my name is: %s \n", c.Base.Name)  
  
    // 调用了被组合的  
    c.Base.SayHello()  
}
```

Http Server —— 组合与重写

- Go 没有重写
- main 函数会输出 I am Parent
- 而在典型的支特重写的语言，如Java，我们可以期望它输出 I am Son

```
func main() {  
    son := Son{  
        Parent{},  
    }  
    son.SayHello()  
}
```

```
type Parent struct {  
}  
  
func (p Parent) SayHello() {  
    fmt.Println("I am " + p.Name())  
}  
  
func (p Parent) Name() string {  
    return "Parent"  
}  
  
type Son struct {  
    Parent  
}  
  
// 定义了自己的 Name() 方法  
func (s Son) Name() string {  
    return "Son"  
}
```

Tip: 当你写下类似继承的代码的时候，千万要先试试它会调过去哪个方法

Http Server —— 支持 RESTFul API

- RESTFul API 定义
- 路由设计 —— Handler 抽象
 - map 语法
 - 基于 map 的 Handler 实现
- 语法：组合
- 重构

Http Server —— 实现 Handler 接口

```
func (h *HandlerBasedOnMap) ServeHTTP(writer  
request *http.Request) {...}
```

```
func (h *HandlerBasedOnMap) Route(method string,  
handlerFunc func(c *Context)) {...}
```

```
// sdkHttpServer 这个是基于 net/http 这个包实现的 http server  
type sdkHttpServer struct {  
    // Name server 的名字，给个标记，日志输出的时候用得上  
    Name string  
    handler Handler 只依赖于接口  
}  
  
func (s *sdkHttpServer) Route(method string, pattern string,  
handlerFunc func(c *Context)) {  
    s.handler.Route(method, pattern, handlerFunc)  
}  
  
func (s *sdkHttpServer) Start(address string) error {  
    return http.ListenAndServe(address, s.handler)  
}
```

Http Server —— 引入新接口

- Handler 和 Server 都有 Route 方法，就间接说明了我们需要引入一个新的接口

```
type Handler interface {
    http.Handler
    Routable
}
```

```
// Routable 可路由的
type Routable interface {
    // Route 设定一个路由，命中该路由的会执行handlerFunc
    Route(method string, pattern string, handlerFunc func(ResponseWriter, *Request))
}

// Server 是http server 的顶级抽象
type Server interface {
    Routable
    // Start 启动我们的服务器
    Start(address string) error
}
```

Tip: 设计不是凭空而来，而是不断重构而来的

Http Server —— 引入新接口

- Handler 和 Server 都有 Route 方法，就间接说明了我们需要引入一个新的接口

```
type Handler interface {
    http.Handler
    Routable
}
```

```
// Routable 可路由的
type Routable interface {
    // Route 设定一个路由，命中该路由的会执行handlerFunc
    Route(method string, pattern string, handlerFunc func(ResponseWriter, *Request))
}

// Server 是http server 的顶级抽象
type Server interface {
    Routable
    // Start 启动我们的服务器
    Start(address string) error
}
```

Tip: 设计不是凭空而来，而是不断重构而来的

Http Server —— 小技巧

```
// 一种常用的GO设计模式,  
// 用于确保HandlerBasedOnMap肯定实现了这个接口  
var _ Handler = &HandlerBasedOnMap{}
```

Http Server —— 重构效果

```
// 注册路由  
//server.Route("/", home)  
//server.Route("/user", user)  
server.Route(method: "POST", pattern: "/user/create", demo.SignUp)  
//server.Route("/order", order)  
  
server.Start(address: ":8080")  
  
// PUT /user 创建用户
```

签入

GET localhost:8080/user/create

Headers (1)

Key	Value
x-toy-web-app	test-app

Body

1 not any router match

签入

POST localhost:8080/user/create

Authorization Headers (1) Body Pre-request Script

Body (1) Body
form-data x-www-form-urlencoded raw text

1 {"biz_code":0,"msg":"","data":123}

课后练习

- 尝试设计树结构。设计 Tree 的顶级接口，并定义二叉树和多叉树的结构体。不要求实现接口的方法；
- 实现二叉树的深度优先遍历或者广度优先遍历。可以使用递归；
- 利用 map 来实现一个 set
- leetcode 练习题（先看答案，再尝试用 Go 写出来）
 - <https://leetcode-cn.com/problems/binary-tree-preorder-traversal/>
 - <https://leetcode-cn.com/problems/invert-binary-tree/>
 - <https://leetcode-cn.com/problems/maximum-depth-of-n-ary-tree/solution/ncha-shu-de-zui-da-shen-du-by-leetcode/>
- 预习：sync 包和树的知识

THANKS

