gin框架

go get github.com/gin-gonic/gin

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

1. gin路由

例子 结构

**type User struct {**

**Account string `json:"account"`**

**Passwd string `json:"passwd"`**

**}**

app= gin.Engine

app.Post(“luyou”,func(){})增

例子：

// 这个函数就是POST的一个handler

**func CreateUser(c \*gin.Context) {**

**var user User**

**err := c.BindJSON(&user)**

**if err != nil {**

**fmt.Println(err.Error())**

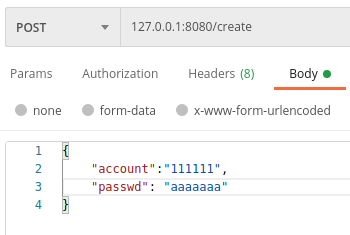
**}**

**fmt.Println(user)**

**c.JSON(200,"create success")**

**}**

**app.POST("/create", CreateUser)**



app.Get()查

例子：

// 这个函数就是GET的一个handler

**func GetUser(c \*gin.Context) {**

**useracc := c.Query("account")**

**// 执行查询操作**

**//there is select but now i do not**

**// hava database so let it go**

**// and i just print it**

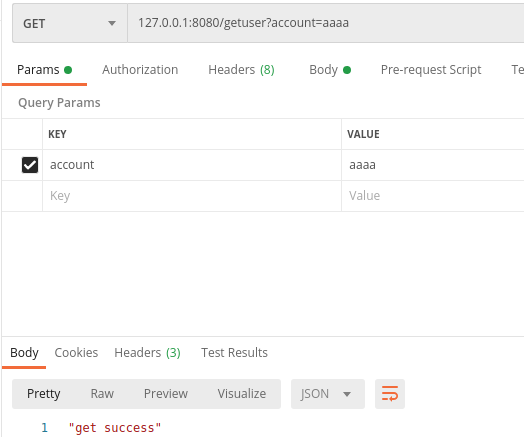
**fmt.Println(useracc)**

**//查询操作结束**

**c.JSON(200,"get success")**

**}**

**app.GET("/getuser", GetUser)**



app.Delete() 删

例子：

// 这个函数就是DELETE的一个handler

**func DeleteUser(c \*gin.Context) {**

**useracc := c.Query("account")**

**// 执行delete操作**

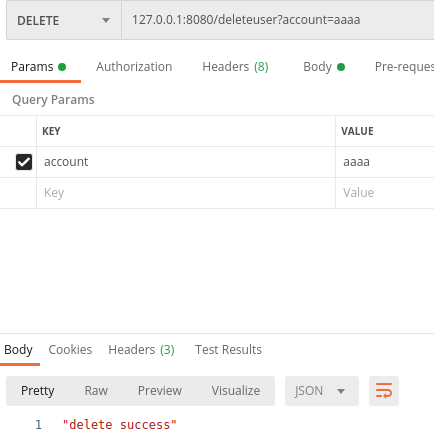
**fmt.Println(useracc)**

**//查询操作结束**

**c.JSON(200,"delete success")**

**}**

**app.DELETE("/deleteuser", DeleteUser)**



app.Put()改

例子：

// PUT 的 handler

**func UpdateUser(c \*gin.Context) {**

**var user User**

**err := c.BindJSON(&user)**

**if err != nil {**

**fmt.Println(err.Error())**

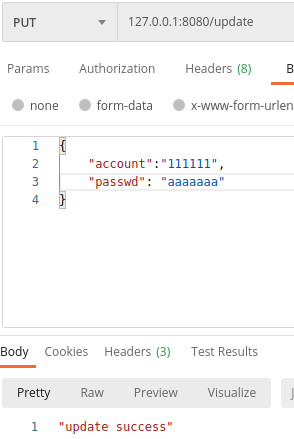
**}**

**fmt.Println(user)**

**c.JSON(200,"update success")**

**}**

**app.PUT("/update", UpdateUser)**



使用Group

group := engine.Group("user") // 作为一个Group组，使用

{

group.POST("/get/parameter",Getparameter)

}

进行管理，（更加整洁吧），整体的路由就变为

x.x.x.x:xxxx/user/get/parameter

2、静态资源加载

①Engine. Static(relativePath, root string)

②Engine.StaticFS(relativePath string,fs http.FileSystem)

③Engine. StaticFile(relativePath, filepath string)

①中两个参数均为string类型，前一个参数为想要暴露在网络上的路由参数，后一个参数代表服务器上的具体实质性路径

// Static serves files from the given file system root.

// Internally a http.FileServer is used, therefore http.NotFound is used instead

// of the Router's NotFound handler.

// To use the operating system's file system implementation,

// use :

// router.Static("/static", "/var/www")

②中两个参数第一个为暴露在外的路由，第二个则是需要http.Dir()包装的一个路径

经过测试，这两个函数并无明显区别，仅是参数不同。介绍里也是这么说。

③//StaticFile注册一个路由，以便为本地文件系统的单个文件提供服务

3、中间件的使用：

Middleware：

全局中间件 和 局部中间件

全局中间件 使用

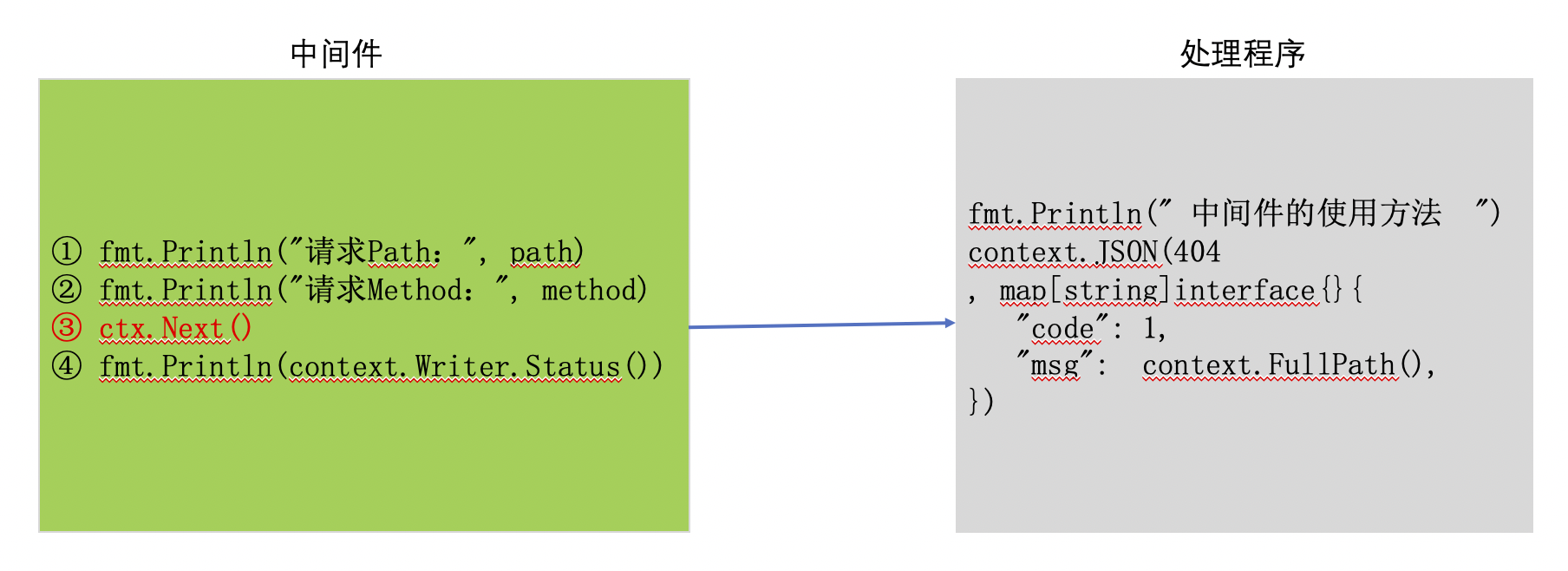
Engine.use(middleware)加载

局部中间件 直接在engine.Post(‘路由’，中间件，处理函数) 这样搭配使用

关于中间件 有Next（）方法和 Abort（）方法

（两个方法都只能在中间件内部使用）

使用Next（）方法后 会执行后续的处理函数，然后再返回来执行中间件内的剩余部分



* 1、程序先执行①和②。
* 2、执行到③时，转而去执行业务处理程序。
* 3、返回到中间件中，执行④。

使用Abort（）方法后，将不执行后续的处理函数，仅执行本函数。Abort不会返回任何消息给前端，

若要返回消息给前端，需要使用AbortWithStatusJSON

例子：

// 创建一个get请求的token验证

**func Middlewareforusetoken(c \*gin.Context) {**

**token := c.GetHeader("token")**

**if token == "admin" {**

**fmt.Println("token verifies success")**

**c.Next()**

**} else {**

**fmt.Println("token verifies failed")**

**c.AbortWithStatusJSON(404, "token wrong")**

**}**

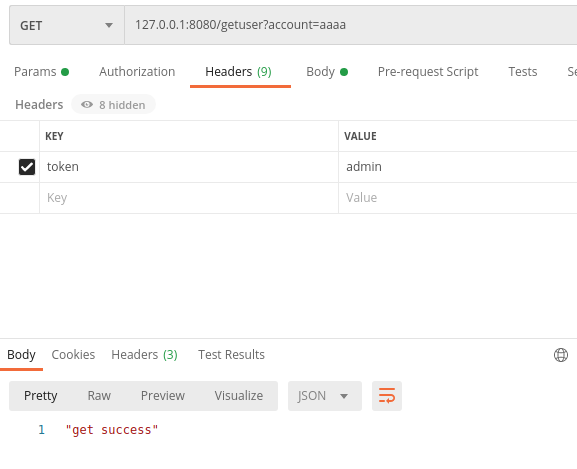
**}**

**// 仅在getuser中使用**

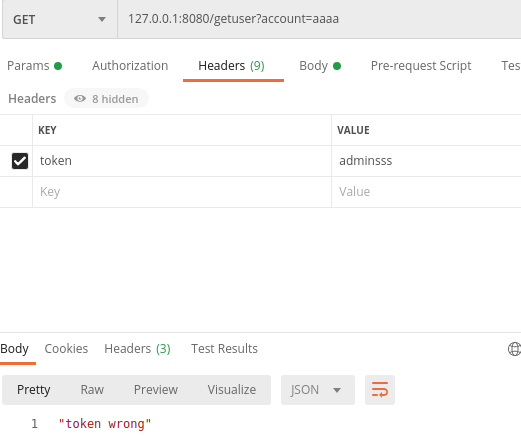
**app.GET("/getuser", Middlewareforusetoken, GetUser)**

**首先成功尝试：**





**失败尝试：**





4、gin关于数据读取

### ①Param（路由变量）

### 例子：/usr/:name

### name := context.Param("name")

### ②Query

### 例子：/welcome?firstname=Jane&lastname=Doe

### Firstname := context.Query（”firstname”）

### Lastname:= context.Query（” lastname”）

### ③ PostForm （不知道是不是对格式没有要求）验证后发现 返回类型全是string类型

### 例子：POST /user/1

### {

### "name":manu,

### "message":this\_is\_great

### }

name := c.PostForm("name")

message := c.PostForm("message")

④ Bind

* context.BindJSON() 支持MIME为application/json的解析
* context.BindXML() 支持MIME为application/xml的解析
* context.BindYAML() 支持MIME为application/x-yaml的解析
* context.BindQuery() 只支持QueryString的解析, 和Query()函数一样
* context.BindUri() 只支持路由变量的解析
* Context.Bind() 支持所有的类型的解析, 这个函数尽量还是少用(当QueryString, PostForm, 路由变量在一块同时使用时会产生意想不到的效果), 目前测试Bind不支持路由变量的解析, Bind()函数的解析比较复杂, 这部分代码后面再看

5、服务启动

Engine.run（“x.x.x.x:xxxx”）

x.x.x.x: 限制访问ip

xxxx: 端口号

File文件上传：

Gin中使用formFile来读取上传的文件

读取得到的文件类型为\*multipart.FileHeader类型，需要将此类型转为file类型，使用.Open()方法读取此文件，得到的类型为File类型。此时，可以将此File类型文件读取到缓存中，然后使用os.Write方法写入文件。