Gin

Web 本质

本质就是一个请求(request)对应一个响应(responce)

最简单的一个 go 服务：

http_server Go

```go
package main

import (
    "fmt"
    "net/http"
)

func sayHello(w http.ResponseWriter, r *http.Request) {
    _, _ = fmt.Fprintln(w, "hello,http")
}
func main() {
    http.HandleFunc("/hello", sayHello)
    err := http.ListenAndServe(":8000", nil)
    if err != nil {
        fmt.Printf("http serve error ")
        return
    }
}

```

Gin 框架初识

各类请求方法

GET POST DELETE方法 Go

```go

func main() {
    r := gin.Default()
    r.GET("/path/:id", func(c *gin.Context) {
        id := c.Param("id")
        user := c.DefaultQuery("user", "ccc")
        pwd := c.Query("pwd")
        c.JSON(200, gin.H{
            "id":   id,
            "user": user,
            "pwd":  pwd,
        })
    })
    r.POST("/path", func(c *gin.Context) {
        user := c.DefaultPostForm("user", "ccc")
        pwd := c.PostForm("pwd")
        c.JSON(200, gin.H{
            "user": user,
            "pwd":  pwd,
        })
    })
    r.DELETE("/path/:id", func(c *gin.Context) {
        id := c.Param("id")
        c.JSON(200, gin.H{
            "id": id,
        })
    })
    r.PUT("/path", func(c *gin.Context) {
        user := c.DefaultPostForm("user", "ccc")
        pwd := c.PostForm("pwd")
        c.JSON(200, gin.H{
            "user": user,
            "pwd":  pwd,
        })
    })
    r.Run(":1010")
}

```

返回 JSON

Gin 返回 JSON 主要有两种方式：`map` 和 `struct`。

其中，`struct` 的字段如果是不可导出的，无法正常序列化，但是可以通过 tag 指定字段名的方式序列化

```go
    // 1. 使用 map 序列化 json
        data := map[string]interface{}{
            "name":    "max",
            "message": "hello",
            "age":     19,
        }
        c.JSON(http.StatusOK, data)

type msg struct {
        Name    string // 不可导出字段无法序列化 如果首字母一定要小写，可以使用 tag 指定字段名
        message string `json:"message"`
        Age     int
    }
        data := msg{
            Name : "max",
            message : "hello",
            Age : 19,
        }
        c.JSON(http.StatusOK, data)

```

Querystring 参数

用法:

在 `URL` 后加 `?querystring` 即可。

> 比如: `http://localhost:8080/user?name=max` 查询名称为 max 的用户
>
> 在服务端可以使用 `c.Query("name")` 获取到 max 值

当需要查询多个内容时，不同内容之间使用 & 相连接。

除了 `c.Query()` 外，类似的还有 `c.DefaultQuery`:如果没有响应结果，就返回默认的值

`c.GetQuery()` 判断是否有该参数

query_string.go Go

```go
func queryString() {
    e := gin.Default()
    // GET 请求 url? 后是 querystring
    // key=value 格式，多个 key-value 之间用 & 连接
    // eg: /web?query=max&age=ccc
    e.GET("/web", func(c *gin.Context) {
        //获取浏览器请求的 query string 参数
        name := c.Query("query") // 通过 query 获取请求中携带的 querystring 参数 http://127.0.0.1:8080/web?query=max
        age := c.Query("age")    // url 可以通过 & 符号查询两个参数，比如：http://127.0.0.1:8080/web?query=max&age=19 todo: 为什么返回的 json 里 age 在前，name 在后
        //name := c.DefaultQuery("query", "ccc") // 如果没有 query，就返回指定的默认值：http://127.0.0.1:8080/web?aaa=123
        //name, ok := c.GetQuery("query") // 如果没有 query，返回 false
        //if !ok {
        // name = "ccc"
        //}
        c.JSON(http.StatusOK, gin.H{
            "name": name,
            "age":  age,
        })
    })
    e.Run()
}
```

表单参数

获取表单中的数据。与 `c.Query()` 类似

form_parse Go

```go
func formParse() {
    e := gin.Default()
    e.LoadHTMLFiles("./template/login.html", "./template/index.html")
    e.GET("/login", func(c *gin.Context) {
        c.HTML(http.StatusOK, "login.html", nil)
    })
    // 第一种获取 form 表单提交的数据
    // 其余两种参考 querystring
    // login post
    e.POST("/login", func(c *gin.Context) {
        username := c.PostForm("username")
        password := c.PostForm("password")
        c.HTML(http.StatusOK, "index.html", gin.H{
            "Name":     username,
            "password": password,
        })
    })
    e.Run()
}
```

uri 参数

获取在 url 中传递的参数对应的值

path_parse Go

```go
// 获取请求的 path(URI)参数
func main() {
    e := gin.Default()
    // http://127.0.0.1:8080/user/max/12
    // ep :http://127.0.0.1:8080/blog/2023/10
    e.GET("/user/:name/:age", func(c *gin.Context) {
        name := c.Param("name")
        age := c.Param("age")
        c.JSON(http.StatusOK, gin.H{
            "name": name,
            "age":  age,
        })
    })

    e.Run()
}
```

参数绑定

获取 `JSON` 中的数据并绑定到某个 Go 结构体中。

> `ShoudBind` 会按照下面的顺序解析请求中的数据完成绑定：
>
> 1. 如果是 `GET` 请求，只使用 `Form` 绑定引擎(query)
> 2. 如果是 `POST` 请求，首先检查 `content-type` 是否为 `JSON` 或 `xml` ，然后再使用 `Form` (form-data)

`ShoudBind` 部分源码：

shoudBind.go Go

```go
// ShouldBind checks the Method and Content-Type to select a binding engine automatically,
// Depending on the "Content-Type" header different bindings are used, for example:
//
// "application/json" --> JSON binding
// "application/xml"  --> XML binding
//
// It parses the request's body as JSON if Content-Type == "application/json" using JSON or XML as a JSON input.
// It decodes the json payload into the struct specified as a pointer.
// Like c.Bind() but this method does not set the response status code to 400 or abort if input is not valid.
func (c *Context) ShouldBind(obj any) error {
    b := binding.Default(c.Request.Method, c.ContentType())
    return c.ShouldBindWith(obj, b)
}
```

从源码中可以看到，`ShoudBind` 会检查请求方法和 Content-Type 去自动选定一个绑定引擎，根据不同的Content-Type 选择不同的绑定方式。

- 如果是 `"application/json"` 使用 JSON 绑定
- 如果是 `"application/xml"` 使用 xml 绑定

此外，该函数会解析 `json payload` 到指针类型的结构体上。就像 `c.Bind()`，但是如果输入值无效时，`c.ShoudBind` 不会将响应码设置为400或者直接中断。

parm_band Go

```go
type user struct {
    Username string `form:"username"`
    Password string `form:"password"`
}

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

    e.GET("/user", func(c *gin.Context) {
        //username := c.Query("username")
        //password := c.Query("password")
        //u := user{
        // Username: username,
        // Password: password,
        //}
        u := user{}
        err := c.ShouldBind(&u) // 函数传参是值传递，所以如果不加取址符，修改的是 u 的副本，而不是 u 本身
        if err != nil {
            c.JSON(http.StatusBadRequest, gin.H{
                "error": err,
            })
            return
        }
        fmt.Printf("%#v\n", u)
        c.JSON(http.StatusOK, gin.H{
            "message": "ok",
        })
    })
    e.Run()
```

重定向：

主要有两种重定向方式:

1. 站外重定向
2. 站内重定向

站外重定向：

主要是使用 `c.Redirect(http.StatusMovedPermanently, "https://www.baidu.com")`

其中 **http.StatusMovedPermanntly 的编码为301**

这个方法可以让用户重定向到某个外站站点。

Plain Text

```
// 跳转到百度
e.GET("/index", func(c *gin.Context) {
   c.Redirect(http.StatusMovedPermanently, "https://www.baidu.com")
})
```

站内重定向

站内重定向主要是指跳转到站内的其他路由

比如：用户访问路由 "/a" 时，系统自动跳转到路由 "/b"。

Plain Text

```
// 路由跳转
e.GET("/a", func(c *gin.Context) {
   c.Request.URL.Path = "/b" // 把请求的 URL 修改
   e.HandleContext(c)        // 继续后续处理
})
e.GET("/b", func(c *gin.Context) {
   c.JSON(http.StatusOK, gin.H{
      "status": "ok",
   })
})
```

路由

主要是各个路由方法，除了 RESTFUL API 外，Gin 还支持，`Any` 路由，其内部是实现了所有请求方法的相应处理。还有路由组的相关应用：

```
// 路由组
// 视频的首页和详情页
videoG := e.Group("/video")
{
   videoG.GET("/index", func(c *gin.Context) {
      c.JSON(http.StatusOK, gin.H{"message": "/video/index"})
   })
}
```

最后 Gin 支持 `NoRoute` 方法，即如果用户访问了站内没有的路由，同一跳转到某个页面。

```
e.NoRoute(func(c *gin.Context) {
   c.JSON(http.StatusNotFound, gin.H{
      "msg": "https://www.baidu.com",
   })
})
```

中间件

什么是中间件?

> Gin 框架允许开发者在处理请求的工程中，加入用户自己的钩子函数。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

如何实现一个中间件?

在 Gin 中，中间件必须是 `gin.HandlerFunc` 类型，比如下面的代码就是一个中间件

```go
func index(c *gin.Context) {    // 简单地说：只要函数参数是 *gin.Context，即可视为一个中间件
    c.JSON(http.StatusOK, gin.H{
        "msg": "ok",
    })
}
```

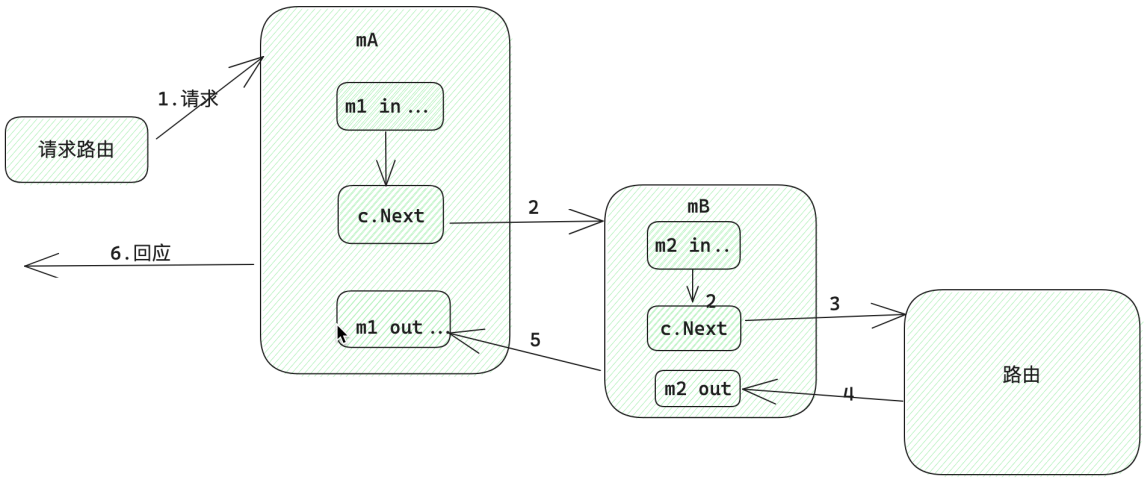
中间件的调用逻辑?

比如：

```go
func middleWareA(c *gin.Context) {
    fmt.Println("a in...")
    c.Next()
    fmt.Println("a out...")
}

func middleWareB(c *gin.Context) {
    fmt.Println("b in ...")
    c.Next()
    fmt.Println("b out ...")
}
// 调用这些中间件
c.Use(A,B)
// 路由
c.GET("/middle"),middle)
```

在上面的代码中，用户访问 `/middle` 路由时，gin 会首先访问中间件 A，B。整个执行顺序为：



其中 `c.Next()` 是指 **调用后续的处理函数**

与之相对， `c.Abort()` 是指**阻止调用后续的处理函数**，即执行到此处时，不再调用后续的逻辑处理，直接执行下一行代码。

gin.Use() 源码：

gin.Use Go

```go
// Use attaches a global middleware to the router. i.e. the middleware attached through Use() will be
// included in the handlers chain for every single request. Even 404, 405, static files...
// For example, this is the right place for a logger or error management middleware.
func (engine *Engine) Use(middleware ...HandlerFunc) IRoutes {
    engine.RouterGroup.Use(middleware...)
    engine.rebuild404Handlers()
    engine.rebuild405Handlers()
    return engine
}
```

从源码中可以看到，`Use()` 函数的参数为 `...HandlerFunc` 类型的函数，所以 `Use()` 可以调用多个中间件。

此外，还可以使用 `Use()` 函数处理 404,405,静态文件等请求。

如何在不同的中间件中传递参数?

`c.Set("key","value")`

`c.Get("key")` 也可以是 `c.MustGet("key")`

如何注册中间件?

1. 在单个路由中注册中间件：

Go

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

2. 在路由组中注册中间件：

```go
// 1.
shopGroup := r.Group("/shop", StatCost())
{
    shopGroup.GET("/index", func(c *gin.Context) {...})
    ...
}
// 2.
shopGroup := r.Group("/shop")
shopGroup.Use(StatCost())
{
    shopGroup.GET("/index", func(c *gin.Context) {...})
    ...
}
```

Gin 其他注意事项:

Gin 默认中间件

> gin.Default() 默认使用了 Logger 和 Recovery 中间件，其中：
>
> - Logger 中间件将日志写入 gin.DefaultWriter，即使配置了 GIN_MODE=release。
> - Recovery 中间件会 recover 任何 panic。如果有 panic 的话，会写入500响应码。
>
> 如果不想使用上面两个默认的中间件，可以使用 gin.New() 新建一个没有任何默认中间件的路由。

Gin 中间件中使用goroutine

> 当在中间件或handler中启动新的goroutine时，**不能使用**原始的上下文（c *gin.Context），必须使用其只读副本（c.Copy()）。