

Gin

xbZhong

2025-12-10

[本页PDF](#)

HttpRouter

Gin的路由组件

- **一对一匹配**: 一个请求只能匹配到一个路由
- **路径自动校正**: 路径错误或大小写不匹配会进行重定向
- **路由参数自动解析**: 路由器支持传递动态值
- **错误处理**: 可以设置一个错误处理器来处理请求中的异常，路由器会将其捕获并记录，然后重定向到错误页面
- **高性能**: 内部使用 `Trie` 树存储路由，匹配时间复杂度为 $O(K)$
- **支持Restful API**

Gin

概念

- 一个web框架，底层基于 `HttpRouter`
- 核心对象：
 - **Router**: 管理 URL 路径和处理函数
 - **Handler/ Context**: 处理请求和响应
 - **Params**: 路径参数

基本用法

导入模块

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

创建路由器

- 返回一个带**Logger + Recovery**的默认路由器

```
router := gin.Default()
```

启动服务器

- 如果不写端口， 默认是8080

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

创建路由

- 可以有 `GET`、`POST`、`PUT`、`DELETE`

- **Handler**是一个函数类型

- `type HandlerFunc func(*gin.Context)`

- 函数要想成为 `Handler`，参数列表就**必须要带** `*gin.Context`

```
1 | router.GET("PATH", Login)
2 |
3 | func Login(c *gin.Context){
4 |
5 | }
```

gin.Context

核心：是单次HTTP请求的所有状态、数据和控制权的载体

注意

- 不能保存 `Context` 指针到 `goroutine`
 - 主要是因为 `Context` 指针是复用的。如果当前 `Context` 指针进入到协程中，由于主协程和子协程是并行的，子协程在执行业务的时候主协程已经返回响应，此时的 `Context` 指针已经被回收，当子协程执行完业务准备借助 `Context` 指针进行数据封装等其它操作的时候，很可能会污染到其它HTTP请求的数据

临时数据存储器

- `Context` 内部维护了一个 `map`，可以存放单个HTTP请求的临时数据，同时加入了读写锁保证线程安全

```
1 | type Context struct {
2 |     keys map[string]interface{}
3 |     mu    sync.RWMutex // 读写锁，线程安全
4 | }
```

- 使用 `Context.Set()` 方法设置值
- 使用 `Context.Get()` 方法获取值

生命周期

- 请求进入
- 创建 `gin.Context`
- 经过中间件/ `handler`
- 返回响应
- `Context` 回收，被放入 `sync.Pool`

模式切换

Gin一共有三种运行模式，本质是影响日志输出、调试信息、性能开销

使用 `gin.SetMode()` 进行模式切换

- `Debug`：本地开发和调试
 - `gin.DebugMode`
- `Release`：生产环境
 - `gin.ReleaseMode`
- `Test`：单元测试
 - `gin.TestMode`

响应方法

返回纯文本

- `c.String`
- 用于为前端返回纯文本

```
1 // 函数签名
2 func (c *Context) String(code int, format string, values ...interface{})
3 // 例子
4 c.String(200, "Hello %s", "Jack")
```

返回JSON

- `c.JSON`
- API接口返回结构化数据，Gin会自动设置 `Content-Type: application/json`

```
1 // 函数签名
2 func (c *Context) JSON(code int, obj interface{})
3 // 例子
4 c.JSON(200, gin.H{
5     "id": 42,
6     "name": "Jack",
7 })
```

返回HTML模板

- `c.HTML`
- 用于返回HTML模板，使用前要先加载模板

```
1 // 函数签名
2 func (c *Context) HTML(code int, name string, obj interface{})
3 // 例子
4 r.LoadHTMLGlob("templates/*") // 记载模板
5 c.HTML(200, "index.html", gin.H{"title": "Home"})
```

返回自定义二进制数据

- `c.Data`
- 可用于返回图片，二进制流，文件

```
1 // 函数签名
2 func (c *Context) Data(code int, contentType string, data []byte)
3 // 例子
4 c.Data(200, "text/plain", []byte("hello world"))
```

重定向

- `c.Redirect`

```
1 // 函数签名
2 func (c *Context) Redirect(code int, location string)
3 // 例子
4 c.Redirect(302, "https://www.example.com")
```

路由参数

路由优先级

- 静态路由
- 参数路由
- 通配路由

通配符匹配规则

- 以 `:` 开头的是路径参数（变量），且一个 `:` 只能匹配一个URL段
 - 当方法绑定的路由是 `/user/:user`，下面的几种 URL 的匹配情况

•	1	/user/gordon	match
	2	/user/you	match
	3	/user/gordon/profile	no match
	4	/user/	no match

- 以 `*` 表示捕获全部参数

```

1 | Pattern: /src/*filepath
2 |
3 | /src/           match
4 | /src/somefile.go   match
5 | /src/subdir/somefile.go match

```

参数获取

`c` 的类型是 `*gin.Context`

- 使用 `c.Param()` 获取路径参数，得到的是字符串类型的参数

```

1 | func GetUser(c *gin.Context){
2 |     id := c.Param("id")
3 | }

```

- 使用 `c.Query()` 获取查询参数，若没有则返回空字符串

- 也可以用 `c.DefaultQuery()`，可以设定返回默认值，如果查询参数值为空可以返回默认值

```

1 | func QueryUser(c *gin.Context) {
2 |     id := c.Query("id")           // "42"
3 |     role := c.DefaultQuery("role", "guest") // "admin", 默认值 "guest"
4 | }

```

- 使用 `c.PostForm()` 获取表单参数

- 也可以用 `c.DefaultPostForm()`，可以设定返回默认值，如果对应参数值为空可以返回默认值

```

1 | func FormUser(c *gin.Context) {
2 |     username := c.PostForm("username")      // "jack"
3 |     age := c.DefaultPostForm("age", "0")    // "18"
4 | }

```

- 使用 `c.ShouldBindJSON()` 获取JSON参数，方法里可以传结构体地址，方法会根据**结构体标签**自动把JSON映射为结构体

```

1 type User struct {
2     Username string `json:"username"`
3     Age      int    `json:"age"`
4 }
5
6 func JSONUser(c *gin.Context) {
7     var user User
8     if err := c.ShouldBindJSON(&user); err != nil {
9         c.JSON(400, gin.H{"error": err.Error()})
10        return
11    }
12 }

```

结构体标签

之前在Go中有说过，这里补充一下 `binding` 字段，主要用来告诉Gin绑定器和validator怎么处理字段

常用binding字段

字段/选项	说明
<code>required</code>	必填字段，空值报错
<code>required_with</code>	当另一个字段有值时必须填写
<code>required_without</code>	当另一个字段没有值时必须填写
<code>email</code>	邮箱格式
<code>url</code>	URL格式
<code>ip</code>	IPv4/IPv6
<code>len = x</code>	字符串或数组长度必须为 x
<code>min=x / max=x</code>	数值最小/最大值或长度限制
<code>gte = x / lte = x</code>	大于等于/小于等于
<code>oneof=a b c</code>	值必须在集合里
<code>custom</code>	注册自定义验证器
<code>eqfield=FieldName</code>	值必须等于另一个字
<code>nefield=FieldName</code>	值必须不等于另一个字段

```
1 type User struct {
2     Name string `json:"name" form:"name" binding:"required"`
3     Email string `json:"email" form:"email" binding:"required,email"`
4     Age   int    `json:"age" form:"age" binding:"gte=0,lte=120"`
5 }
```

数据解析

就是把数据自动解析进Go里的结构体

最核心的API: `ShouldBind` , Gin会自动根据**Content-Type + 请求方法**自动选择解析器

- `ShouldBindJSON` 是强制使用JSON解析器，而 `ShouldBind` 会进行自动检测
- 对于URL路径参数的数据绑定，需要用 `ShouldBindUri`
- 对于不同的 `content-type`，需要给结构体打上不同的标签

请求来源	tag
JSON	<code>json:"field"</code>
Query / Form	<code>form:"field"</code>
Path	<code>uri:"field"</code>
Header	<code>header:"field"</code>

```
1 // 例子
2 type CreateUserReq struct {
3     Username string `json:"username"`
4     Age       int    `json:"age"`
5 }
6
7 func CreateUser(c *gin.Context) {
8     var req CreateUserReq
9
10    if err := c.ShouldBindJSON(&req); err != nil {
11        c.JSON(400, gin.H{"error": err.Error()})
12        return
13    }
14
15    c.JSON(200, req)
16 }
```

数据校验

导入模块

```
import "go-playground/validator/v10"
```

工作机制

- 当你给字段加了 `binding:"..."` 标签，Gin会把这个标签交给 `validator` 解析规则并进行校验

```
1 type User struct {
2     Name string `json:"name" binding:"required"`
3     Email string `json:"email" binding:"required,email"`
4     Age   int    `json:"age" binding:"gte=18,lte=100"`
5 }
6
7 func CreateUser(c *gin.Context) {
8     var user User
9     if err := c.ShouldBindJSON(&user); err != nil {
10         c.JSON(400, gin.H{"error": err.Error()})
11         return
12     }
13     c.JSON(200, gin.H{"message": "ok", "user": user})
14 }
```

自定义校验器

后面再学

文件传输

单文件上传

- 使用 `c.FormFile("文件名")` 获取文件

```
1 func Upload(c *gin.Context) {
2     file, err := c.FormFile("file")
3     if err != nil {
4         c.JSON(400, gin.H{"error": err.Error()})
5         return
6     }
7     // 保存文件
8     c.SaveUploadedFile(file, "./uploads/"+file.Filename)
9 }
```

多文件上传

- 使用 `c.MultipartForm()` 获取Gin解析好的表单
- 使用 `.File["值"]` 方法根据键值对获取对应的文件列表

```
1 func MultiUpload(c *gin.Context) {
2     form, _ := c.MultipartForm()
3     files := form.File["files"]
4
5     for _, file := range files {
6         c.SaveUploadedFile(file, "./uploads/"+file.Filename)
7     }
8
9     c.JSON(200, gin.H{"count": len(files)})
10 }
```

文件下载

- 使用 `c.File()` 方法可以直接返回文件给前端

```
1 func Download(c *gin.Context) {
2     c.File("./uploads/test.txt")
3 }
```

- 使用 `c.FileAttachment(filepath,filename)` 设置响应头，返回给前端后让浏览器自动下载

```
1 // 接收前端的文件请求路径并返回文件
2 func download(ctx *gin.Context) {
3     // 获取文件名
4     filename := ctx.Param("filename")
5     // 返回对应文件
6     ctx.FileAttachment(filename, filename)
7 }
```

保存文件

- 使用 `c.SaveUploadedFile(filepath,filename)` 方法保存文件到服务端

```
1 func Upload(c *gin.Context) {
2     file, err := c.FormFile("file")
3     if err != nil {
4         c.JSON(400, gin.H{"error": err.Error()})
5         return
6     }
7
8     // 保存文件
9     c.SaveUploadedFile(file, "./uploads/"+file.Filename)
10 }
```

路由管理

路由分组

- 使用 `c.Group("组名")` 进行路由分组
- 花括号 `{}` 只是为了规范，阅读方便

```
1 func main() {
2     e := gin.Default()
3     v1 := e.Group("v1")
4     {
5         v1.GET("/hello", Hello)
6         v1.GET("/login", Login)
7     }
8     v2 := e.Group("v2")
9     {
10        v2.POST("/update", Update)
11        v2.DELETE("/delete", Delete)
12    }
13 }
```

版本路由（嵌套路由）

```
1 api := r.Group("/api")
2 {
3     v1 := api.Group("/v1")
4     {
5         v1.GET("/user/:id", GetUserV1)
6     }
7
8     v2 := api.Group("/v2")
9     {
10        v2.GET("/user/:id", GetUserV2)
11    }
12 }
```

404路由

- 使用 `c.NoRoute()` 方法设置当访问的 URL 不存在时如何处理

```
1 // 函数签名
2 func (engine *Engine) NoRoute(handlers ...HandlerFunc)
3 // 例子
4 e.NoRoute(func(context *gin.Context) {
5     context.String(http.StatusNotFound, "<h1>404 Page Not Found</h1>")
6 })
```

405路由

- 使用 `c.NoMethod()` 方法设置当请求方法类型不允许时如何处理

```
1 // 函数签名
2 func (engine *Engine) NoMethod(handlers ...HandlerFunc)
3 // 例子
4 e.NoMethod(func(context *gin.Context) {
5     context.String(http.StatusMethodNotAllowed, "method not allowed")
6 })
```

错误机制

错误链机制：Gin拥有内置的 `error` 机制，通过在 `Context` 中维护一个错误列表 `c.Errors`，将错误与响应解耦，允许业务层只记录错误，由中间件统一处理和格式化返回

```
1 // Context
2 type Context struct {
3     Errors errorMsgs
4 }
5
6 type errorMsgs []*Error
7
8 // Error类
9 type Error struct {
10     Err error
11     Type ErrorCode
12     Meta interface{}
13 }
```

Gin引入

```
c.Error(err)
```

来记录错误，而不是立刻返回错误，同时利用全局错误处理中间件来处理，`Handler` 只进行错误的返回

中间件

在Gin中，所有的请求在到达 Handler 之前都要经过中间件 (Middleware)

- 中间件和 Handler 是同一种类型，即

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

中间件操作

- 放行

```
c.Next()
```

- 拦截

```
1 func Auth(c *gin.Context) {
2     if !login {
3         c.JSON(401, gin.H{"msg": "unauthorized"})
4         c.Abort()
5         return
6     }
7     c.Next()
8 }
```

- 拦截并返回响应

```
1 c.AbortWithStatusJSON(403, gin.H{
2     "msg": "forbidden",
3 })
```

中间件生效范围

- 全局中间件：在引擎上挂载

```
1 r := gin.New()
2 r.Use(Logger(), Recovery())
```

- 分组中间件：在路由组上挂载

```
1 admin := r.Group("/admin")
2 admin.Use(AuthMiddleware())
3 {
4     admin.GET("/dashboard", dashboard)
5 }
```

- 单路由中间件：在路由上挂载

```
r.GET("/ping", TimeCost(), ping)
```

原理

Gin中的中间件其实用到了责任链模式，`Context` 中维护着一个 `HandlersChain`，本质上是一个 `[]HandlerFunc` 切片和一个 `index`

```
1 type Context struct {
2     handlers HandlersChain
3     index    int
4 }
```

Next()

- 通过 `index` 的值决定当前执行到第几个中间件

```
1 func (c *Context) Next() {
2     c.index++
3     for c.index < len(c.handlers) {
4         c.handlers[c.index](c)
5         c.index++
6     }
7 }
```

Abort()

- 本质是修改 `index` 为一个特别大的数，致使无法进入循环，从而终止后续中间件的触发

```
1 const abortIndex = math.MaxInt8 >> 1
2 func (c *Context) Abort() {
3     c.index = abortIndex
4 }
```

优雅关闭

优雅关闭是为了防止 `CTRL + C` 致使服务暂停导致正在请求的服务被中断，而优雅关闭可以实现

- 停止接收新请求
- 等待正在处理的请求完成
- 再安全退出

实现核心

- 需要自己创建 `http.Server`，不能使用 `c.Run()`，会失去对服务器的控制权
- 然后需要创建一个新的 `goroutine` 来启动服务，主协程可以继续执行监听代码
- 创建一个 `channel` 来对关闭信号进行监听
- 当进程被 `kill` 或者 `ctrl c` 产生，`channel` 就会获得一个值
- `-<quit` 表示阻塞等待，直到 `quit` 中有值
- 后面那部分以后接着看

```
1 package main
2
3 import (
4     "context"
5     "log"
6     "net/http"
7     "os"
8     "os/signal"
9     "syscall"
10    "time"
11
12    "github.com/gin-gonic/gin"
13 )
14
15 func main() {
16     // 1. 创建 gin 实例
17     r := gin.Default()
18
19     // 2. 注册路由
20     r.GET("/ping", func(c *gin.Context) {
21         time.Sleep(3 * time.Second) // 模拟耗时请求
22         c.JSON(200, gin.H{
23             "msg": "pong",
24         })
25     })
26
27     // 3. 自己创建 http.Server (关键)
28     srv := &http.Server{
29         Addr:    ":8080",
30         Handler: r,
31     }
32
33     // 4. 启动服务 (必须 goroutine)
34     go func() {
35         log.Println("Server running at :8080")
36         if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed
37     {
38             log.Fatalf("listen error: %v\n", err)
39     }
40     }()
41
42     // 5. 监听系统信号
43     quit := make(chan os.Signal, 1)
44     signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
45
46     <-quit // 阻塞, 直到收到信号
```

```
46     log.Println("Shutdown signal received")
47
48     // 6. 设置超时 context
49     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
50     defer cancel()
51
52     // 7. 优雅关闭
53     if err := srv.Shutdown(ctx); err != nil {
54         log.Println("Server forced to shutdown:", err)
55     }
56
57     log.Println("Server exiting")
58 }
```