

# 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RestFulAPI**

## 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

```
router.GET("PATH",Login)

func Login(c *gin.Context){

}
```

## 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

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

- 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

```
// 函数签名
func (c *Context) String(code int, format string, values ...interface{})
// 例子
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

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

### 返回HTML模板

- c.HTML
- 用于返回HTML模板，使用前要先加载模板

```
// 函数签名
func (c *Context) HTML(code int, name string, obj interface{})
// 例子
r.LoadHTMLGlob("templates/*") // 加载模板
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

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

## 重定向

- `c.Redirect`

```
// 函数签名
func (c *Context) Redirect(code int, location string)
// 例子
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

- |                                   |          |
|-----------------------------------|----------|
| <code>/user/gordon</code>         | match    |
| <code>/user/you</code>            | match    |
| <code>/user/gordon/profile</code> | no match |
| <code>/user/</code>               | no match |

- 以 `*` 表示捕获全部参数

- |                                      |       |
|--------------------------------------|-------|
| Pattern: <code>/src/*filepath</code> |       |
| <code>/src/</code>                   | match |
| <code>/src/somefile.go</code>        | match |
| <code>/src/subdir/somefile.go</code> | 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

```
func GetUser(c *gin.Context){
    id := c.Param("id")
}
```

- 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

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

- 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

```
func FormUser(c *gin.Context) {
    username := c.PostForm("username") // "jack"
    age := c.DefaultPostForm("age", "0") // "18"
}
```

- 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

```
type User struct {
    Username string `json:"username"`
    Age      int    `json:"age"`
}

func JSONUser(c *gin.Context) {
    var user User
    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }
}
```

## 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      |

| 字段/选项             | 说明            |
|-------------------|---------------|
| len = x           | 字符串或数组长度必须为 x |
| min=x / max=x     | 数值最小/最大值或长度限制 |
| gte = x / lte = x | 大于等于/小于等于     |
| oneof=a b c       | 值必须在集合里       |
| custom            | 注册自定义验证器      |
| eqfield=FieldName | 值必须等于另一个字     |
| nefield=FieldName | 值必须不等于另一个字段   |

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

## 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         | json:"field"   |
| Query / Form | form:"field"   |
| Path         | uri:"field"    |
| Header       | header:"field" |

```
// 例子
type CreateUserReq struct {
    Username string `json:"username"`
    Age      int    `json:"age"`
}

func CreateUser(c *gin.Context) {
    var req CreateUserReq

    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }

    c.JSON(200, req)
}
```

## 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

```
type User struct {
    Name string `json:"name" binding:"required"`
    Email string `json:"email" binding:"required,email"`
    Age int `json:"age" binding:"gte=18,lte=100"`
}

func CreateUser(c *gin.Context) {
    var user User
    if err := c.ShouldBindJSON(&user); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }
    c.JSON(200, gin.H{"message": "ok", "user": user})
}
```

### 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

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

## 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

```
func MultiUpload(c *gin.Context) {
    form, _ := c.MultipartForm()
    files := form.File["files"]

    for _, file := range files {
        c.SaveUploadedFile(file, "./uploads/"+file.Filename)
    }

    c.JSON(200, gin.H{"count": len(files)})
}
```

## 文件下载

- 使用 `c.File()` 方法可以直接返回文件给前端

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

- 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**

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

## 保存文件

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



```
func Upload(c *gin.Context) {
    file, err := c.FormFile("file")
    if err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }

    // 保存文件
    c.SaveUploadedFile(file, "./uploads/"+file.Filename)
}
```

## 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

```
func main() {
    e := gin.Default()
    v1 := e.Group("v1")
    {
        v1.GET("/hello", Hello)
        v1.GET("/login", Login)
    }
    v2 := e.Group("v2")
    {
        v2.POST("/update", Update)
        v2.DELETE("/delete", Delete)
    }
}
```

### 版本路由（嵌套路由）

```
api := r.Group("/api")
{
    v1 := api.Group("/v1")
    {
        v1.GET("/user/:id", GetUserV1)
    }

    v2 := api.Group("/v2")
    {
        v2.GET("/user/:id", GetUserV2)
    }
}
```

### 404路由

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

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

## 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

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

## 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

```
// Context
type Context struct {
    Errors errorMsgs
}

type errorMsgs []*Error

// Error类
type Error struct {
    Err error
    Type ErrorType
    Meta interface{}
}
```

## 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

```
func Auth(c *gin.Context) {  
    if !login {  
        c.JSON(401, gin.H{"msg": "unauthorized"})  
        c.Abort()  
        return  
    }  
    c.Next()  
}
```

- 拦截并返回响应

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

## 中间件生效范围

- 全局中间件：在引擎上挂载

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

- 分组中间件：在路由组上挂载

```
admin := r.Group("/admin")  
admin.Use(AuthMiddleware())  
{  
    admin.GET("/dashboard", dashboard)  
}
```

- 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`

```
type Context struct {  
    handlers HandlersChain  
    index     int  
}
```

#### Next()

- 通过 `index` 的值决定当前执行到第几个中间件

```
func (c *Context) Next() {  
    c.index++  
    for c.index < len(c.handlers) {  
        c.handlers[c.index](c)  
        c.index++  
    }  
}
```

#### 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

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

## 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

```
package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

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

func main() {
    // 1. 创建 gin 实例
    r := gin.Default()

    // 2. 注册路由
    r.GET("/ping", func(c *gin.Context) {
        time.Sleep(3 * time.Second) // 模拟耗时请求
        c.JSON(200, gin.H{
            "msg": "pong",
        })
    })

    // 3. 自己创建 http.Server (关键)
    srv := &http.Server{
        Addr:    ":8080",
        Handler: r,
    }

    // 4. 启动服务 (必须 goroutine)
    go func() {
        log.Println("Server running at :8080")
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("listen error: %v\n", err)
        }
    }()

    // 5. 监听系统信号
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)

    <-quit // 阻塞，直到收到信号
    log.Println("Shutdown signal received")

    // 6. 设置超时 context
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    // 7. 优雅关闭
    if err := srv.Shutdown(ctx); err != nil {
        log.Println("Server forced to shutdown:", err)
    }
}
```

```
log.Println("Server exiting")  
}
```