

# Go Gin 简明教程

## Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)



关键字：**Gin教程** **Gin中文文档** **Go语言Web框架** **Go环境搭建**

## Gin 简介

Gin is a HTTP web framework written in Go (Golang). It features a Martini-like API with much better performance – up to 40 times faster. If you need smashing performance, get yourself some Gin.

Gin 是使用 Go/golang 语言实现的 HTTP Web 框架。接口简洁，性能极高。截止 1.4.0 版本，包含测试代码，仅14K，其中测试代码 9K 左右，也就是说框架源码仅 5K 左右。

```
1 $ find . -name "*_test.go" | xargs cat | wc -l
2 8657
3 $ find . -name "*.go" | xargs cat | wc -l
4 14115
```

### Gin 特性

- **快速**：路由不使用反射，基于Radix树，内存占用少。
- **中间件**：HTTP请求，可先经过一系列中间件处理，例如：Logger，Authorization，GZIP等。这个特性和 NodeJs 的 koa 框架很像。中间件机制也极大地提高了框架的可扩展性。
- **异常处理**：服务始终可用，不会宕机。Gin 可以捕获 panic，并恢复。而且有极为便利的机制处理HTTP请求过程中发生的错误。
- **JSON**：Gin可以解析并验证请求的JSON。这个特性对 Restful API 的开发尤其有用。
- **路由分组**：例如将需要授权和不需要授权的API分组，不同版本的API分组。而且分组可嵌套，且性能不受影响。
- **渲染内置**：原生支持JSON，XML和HTML的渲染。

## 安装Go & Gin

初学者建议先阅读 [Go 语言简明教程](#)。

一篇文章介绍了 Go 基本类型，结构体，单元测试，并发编程，依赖管理等内容。Go 1.13 以上版本的安装推荐该教程的方式。

- 安装 Go (Ubuntu)

```
1 $ sudo apt-get install golang-go
2 $ go version
3 # go version go1.6.2 linux/amd64
```

Ubuntu自带版本太老了，安装新版可以使用如下命令。

```
1 $ sudo add-apt-repository ppa:gophers/archive
2 $ sudo apt-get update
3 $ sudo apt-get install golang-1.11-go
```

默认安装在/usr/lib/go-1.11，需要将 /usr/lib/go-1.11/bin 手动加入环境变量。在 .bashrc 中添加下面的配置，并 source ~/.bashrc

```
1 export PATH=$PATH:/usr/lib/go-1.11/bin
```

参考: [Golang Ubuntu - Github](#)

#### ■ 安装 Go (Mac)

```
1 $ brew install go
2 $ go version
3 # go version go1.12.5 darwin/amd64
```

#### ■ 设置环境变量

在 ~/.bashrc 中添加 GOPATH 变量

```
1 export GOPATH=~/.go
2 export PATH=$PATH:$GOPATH/bin
```

添加完后，source ~/.bashrc

#### ■ 安装一些辅助的工具库

由于网络原因，不能够直接访问 golang.org，但相关的库已经镜像到 [Golang - Github](#)

例如，直接安装 go-outline 时会报网络错误，因为 golang.org/x/tools 是 go-outline 的依赖库。

```
1 $ go get -u -v github.com/ramya-rao-a/go-outline
2 github.com/ramya-rao-a/go-outline (download)
3 Fetching https://golang.org/x/tools/go/buildutil?go-get=1
4 https fetch failed: Get https://golang.org/x/tools/go/buildutil?go-get=1:
5 dial tcp 216.239.37.1:443: i/o timeout
```

因此，可以先从 Github 手动安装好，再安装 go-outline 和 goreturns。

```
1 git clone https://github.com/golang/tools.git $GOPATH/src/golang.org/x/tools
```

```
2 go get -v github.com/ramya-rao-a/go-outline
3 go get -v github.com/sqs/goreturns
4 go get -v github.com/rogppe/godef
```

Go语言有大量的辅助工具，如果你使用 VSCode，将会提示你将必要的工具，例如静态检查、自动补全等工具依次安装完毕。

#### ■ 安装 Gin

```
1 go get -u -v github.com/gin-gonic/gin
```

-v：打印出被构建的代码包的名字

-u：已存在相关的代码包，强行更新代码包及其依赖包

## 第一个Gin程序

在一个空文件夹里新建文件 main.go。

```
1 // geektutu.com
2 // main.go
3 package main
4
5 import "github.com/gin-gonic/gin"
6
7 func main() {
8     r := gin.Default()
9     r.GET("/", func(c *gin.Context) {
10         c.String(200, "Hello, Geektutu")
11     })
12     r.Run() // listen and serve on 0.0.0.0:8080
13 }
```

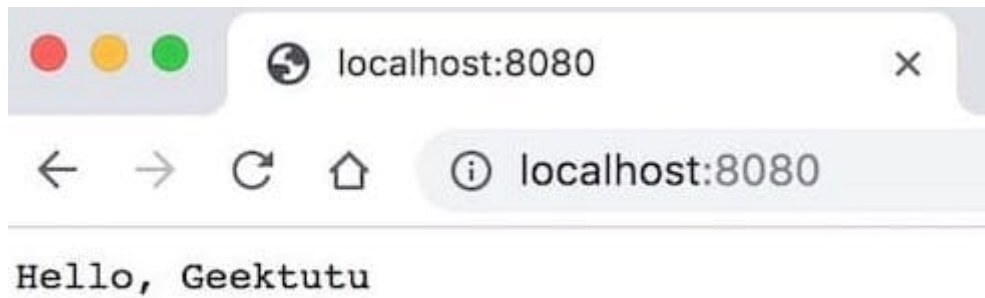
1. 首先，我们使用了 `gin.Default()` 生成了一个实例，这个实例即 WSGI 应用程序。
2. 接下来，我们使用 `r.Get("/", ...)` 声明了一个路由，告诉 Gin 什么样的URL 能触发传入的函数，这个函数返回我们想要显示在用户浏览器中的信息。
3. 最后用 `r.Run()` 函数来让应用运行在本地服务器上，默认监听端口是 `_8080_`，可以传入参数设置端口，例如 `r.Run(":9999")` 即运行在 `_9999_` 端口。

#### ■ 运行

```
1 $ go run main.go
2 [GIN-debug] GET    /                  --> main.main.func1 (3 handlers)
```

```
3 [GIN-debug] Environment variable PORT is undefined. Using port :8080 by default
4 [GIN-debug] Listening and serving HTTP on :8080
```

- 浏览器访问 `http://localhost:8080`



## 路由(Route)

路由方法有 **GET, POST, PUT, PATCH, DELETE** 和 **OPTIONS**, 还有 **Any**, 可匹配以上任意类型的请求。

无参数

```
1 // 无参数
2 r.GET("/", func(c *gin.Context) {
3     c.String(http.StatusOK, "Who are you?")
4 })
```

```
1 $ curl http://localhost:9999/
2 Who are you?
```

curl 参数可参考 <https://man.linuxde.net/curl>

解析路径参数

有时候我们需要动态的路由, 如 `/user/:name`, 通过调用不同的 url 来传入不同的 name。 `/user/:name/*role`, `*` 代表可选。

```
1 // 匹配 /user/geektutu
2 r.GET("/user/:name", func(c *gin.Context) {
3     name := c.Param("name")
4     c.String(http.StatusOK, "Hello %s", name)
5 })
```

```
1 $ curl http://localhost:9999/user/geektutu
2 Hello geektutu
```

## 获取Query参数

```
1 // 匹配users?name=xxx&role=xxx, role可选
2 r.GET("/users", func(c *gin.Context) {
3     name := c.Query("name")
4     role := c.DefaultQuery("role", "teacher")
5     c.String(http.StatusOK, "%s is a %s", name, role)
6 })

1 $ curl "http://localhost:9999/users?name=Tom&role=student"
2 Tom is a student
```

## 获取POST参数

```
1 // POST
2 r.POST("/form", func(c *gin.Context) {
3     username := c.PostForm("username")
4     password := c.DefaultPostForm("password", "000000") // 可设置默认值
5
6     c.JSON(http.StatusOK, gin.H{
7         "username": username,
8         "password": password,
9     })
10 })

1 $ curl http://localhost:9999/form -X POST -d 'username=geektutu&password=1234'
2 {"password":"1234","username":"geektutu"}
```

## Query和POST混合参数

```
1 // GET 和 POST 混合
2 r.POST("/posts", func(c *gin.Context) {
3     id := c.Query("id")
4     page := c.DefaultQuery("page", "0")
5     username := c.PostForm("username")
6     password := c.DefaultPostForm("password", "000000") // 可设置默认值
7
8     c.JSON(http.StatusOK, gin.H{
9         "id": id,
```

```
10         "page":    page,
11         "username": username,
12         "password": password,
13     })
14 })
```

```
1 $ curl "http://localhost:9999/posts?id=9876&page=7" -X POST -d 'username=geektutu&passw
2 {"id":"9876","page":"7","password":"1234","username":"geektutu"}
```

## Map参数(字典参数)

```
1 r.POST("/post", func(c *gin.Context) {
2     ids := c.QueryMap("ids")
3     names := c.PostFormMap("names")
4
5     c.JSON(http.StatusOK, gin.H{
6         "ids":  ids,
7         "names": names,
8     })
9 })
```

```
1 $ curl -g "http://localhost:9999/post?ids[Jack]=001&ids[Tom]=002" -X POST -d 'names[a]=S
2 {"ids":{"Jack":"001","Tom":"002"},"names":{"a":"Sam","b":"David"}}}
```

## 重定向(Redirect)

```
1 r.GET("/redirect", func(c *gin.Context) {
2     c.Redirect(http.StatusMovedPermanently, "/index")
3 })
4
5 r.GET("/goindex", func(c *gin.Context) {
6     c.Request.URL.Path = "/"
7     r.HandleContext(c)
8 })
```

```
1 $ curl -i http://localhost:9999/redirect
2 HTTP/1.1 301 Moved Permanently
3 Content-Type: text/html; charset=utf-8
```



```
4   Location: /
5   Date: Thu, 08 Aug 2019 17:22:14 GMT
6   Content-Length: 36
7
8   <a href="/">Moved Permanently</a>.
9
10  $ curl "http://localhost:9999/goindex"
11  Who are you?
```

## 分组路由(Grouping Routes)

如果有一组路由，前缀都是 `/api/v1` 开头，是否每个路由都需要加上 `/api/v1` 这个前缀呢？答案是不需要，分组路由可以解决这个问题。利用分组路由还可以更好地实现权限控制，例如将需要登录鉴权的路由放到同一分组中去，简化权限控制。

```
1  // group routes 分组路由
2  defaultHandler := func(c *gin.Context) {
3      c.JSON(http.StatusOK, gin.H{
4          "path": c.FullPath(),
5      })
6  }
7  // group: v1
8  v1 := r.Group("/v1")
9  {
10     v1.GET("/posts", defaultHandler)
11     v1.GET("/series", defaultHandler)
12 }
13 // group: v2
14 v2 := r.Group("/v2")
15 {
16     v2.GET("/posts", defaultHandler)
17     v2.GET("/series", defaultHandler)
18 }

1  $ curl http://localhost:9999/v1/posts
2  {"path":"/v1/posts"}
3  $ curl http://localhost:9999/v2/posts
4  {"path":"/v2/posts"}
```

## 上传文件

### 单个文件

```
1  r.POST("/upload1", func(c *gin.Context) {
2      file, _ := c.FormFile("file")
3      // c.SaveUploadedFile(file, dst)
4      c.String(http.StatusOK, "%s uploaded!", file.Filename)
5  })
```

## 多个文件

```
1  r.POST("/upload2", func(c *gin.Context) {
2      // Multipart form
3      form, _ := c.MultipartForm()
4      files := form.File["upload[]"]
5
6      for _, file := range files {
7          log.Println(file.Filename)
8          // c.SaveUploadedFile(file, dst)
9      }
10     c.String(http.StatusOK, "%d files uploaded!", len(files))
11 })
```

## HTML模板(Template)

```
1  type student struct {
2      Name string
3      Age  int8
4  }
5
6  r.LoadHTMLGlob("templates/*")
7
8  stu1 := &student{Name: "Geektutu", Age: 20}
9  stu2 := &student{Name: "Jack", Age: 22}
10 r.GET("/arr", func(c *gin.Context) {
11     c.HTML(http.StatusOK, "arr.tpl", gin.H{
12         "title": "Gin",
13         "stuArr": [2]*student{stu1, stu2},
14     })
15 })
```

```
1  <!-- templates/arr.tpl -->
2  <html>
3  <body>
```

```
4      <p>hello, {{.title}}</p>
5      {{range $index, $ele := .stuArr }}
6      <p>{{ $index }}: {{ $ele.Name }} is {{ $ele.Age }} years old</p>
7      {{ end }}
8  </body>
9  </html>
```

```
1  $ curl http://localhost:9999/arr
2
3  <html>
4  <body>
5      <p>hello, Gin</p>
6      <p>0: Geektutu is 20 years old</p>
7      <p>1: Jack is 22 years old</p>
8  </body>
9  </html>
```

- Gin默认使用模板Go语言标准库的模板 `text/template` 和 `html/template` , 语法与标准库一致, 支持各种复杂场景的渲染。
- 参考官方文档[text/template](#), [html/template](#)

## 中间件(Middleware)

```
1  // 作用于全局
2  r.Use(gin.Logger())
3  r.Use(gin.Recovery())
4
5  // 作用于单个路由
6  r.GET("/benchmark", MyBenchLogger(), benchEndpoint)
7
8  // 作用于某个组
9  authorized := r.Group("/")
10 authorized.Use(AuthRequired())
11 {
12     authorized.POST("/login", loginEndpoint)
13     authorized.POST("/submit", submitEndpoint)
14 }
```

如何自定义中间件呢?

```
1  func Logger() gin.HandlerFunc {
2      return func(c *gin.Context) {
```

```
3         t := time.Now()
4         // 给Context实例设置一个值
5         c.Set("geektutu", "1111")
6         // 请求前
7         c.Next()
8         // 请求后
9         latency := time.Since(t)
10        log.Print(latency)
11    }
12 }
```

## 热加载调试 Hot Reload

Python 的 Flask 框架，有 debug 模式，启动时传入 debug=True 就可以热加载(Hot Reload, Live Reload)了。即更改源码，保存后，自动触发更新，浏览器上刷新即可。免去了杀进程、重新启动之苦。

Gin 原生不支持，但有很多额外的库可以支持。例如

- [github.com/codegangsta/gin](https://github.com/codegangsta/gin)
- [github.com/pilu/fresh](https://github.com/pilu/fresh)

这次，我们采用 [github.com/pilu/fresh](https://github.com/pilu/fresh)。

```
1 go get -v -u github.com/pilu/fresh
```

安装好后，只需要将 `go run main.go` 命令换成 `fresh` 即可。每次更改源文件，代码将自动重新编译(Auto Compile)。

参考 [github.com/pilu/fresh](https://github.com/pilu/fresh) - Github

## 相关链接

- [Golang Gin - Github](#)
- [Gin Web Framework - 英文官方网站](#)

---

专题: [Go 简明教程](#)

本文发表于 2019-08-07，最后修改于 2021-02-06。

本站永久域名「[geektutu.com](https://geektutu.com)」，也可搜索「极客兔兔」找到我。

---

[上一篇](#) « [Go 语言简明教程](#)

[下一篇](#) » [7天用Go从零实现Web框架Gee教程](#)

赞赏支持



## 推荐阅读

### 切片(slice)性能及陷阱

发表于2020-11-30, 阅读约27分钟

### 动手写RPC框架 - GeeRPC第四天 超时处理(timeout)

发表于2020-10-07, 阅读约25分钟

### 动手写ORM框架 - GeeORM第一天 database/sql 基础

发表于2020-03-07, 阅读约32分钟

#关于我 (9)   #Go (48)   #百宝箱 (2)   #Cheat Sheet (1)   #Go语言高性能编程 (20)   #友链 (1)   #Pandas (3)

#机器学习 (9)   #TensorFlow (9)   #mnist (5)   #Python (10)   #强化学习 (3)   #OpenAI gym (4)   #DQN (1)


#Q-Learning (1)   #CNN (1)   #TensorFlow 2 (10)   #官方文档 (10)   #Rust (1)



Gitalk 加载中 ...


### 如何退出协程 goroutine (其他场景)

2 评论 • 13小时前

 **DasyDong** —— `ch := make(chan int) // 带缓冲区`  
`ch := make(chan int, 10) // 不带缓冲区`, 缓冲区满之前, 即使没有接收方, 发送方

### 动手写ORM框架 - GeeORM第三天 记录新增和查询

10 评论 • 26天前

 **xiezhenyu19970913** —— `type generator`  
`func(values ...interface{}) (string,`


### 动手写RPC框架 - GeeRPC第五天 支持HTTP协议

10 评论 • 17小时前

 **andcarefree** —— 请问XDial测试里面,  
`addr := "/tmp/geerpc.sock"`。之后remove

### 切片(slice)性能及陷阱


4 评论 • 15天前

 **bestgopher** —— 大佬爱你, 上班看得停不下来

Gitalk Plus



© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#)  [Star](#)

 996646  48289