# Gin Web 框架官方文档中文版

书栈(BookStack.CN)

# 目　录

# 致谢

当前文档 《Gin Web 框架官方文档中文版》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-09-28。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：Gin  https://gin-gonic.com/zh-cn/docs/

文档地址：http://www.bookstack.cn/books/gin-docs-zh

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# 文档

## Gin 是什么？

Gin 是一个用 Go (Golang) 编写的 HTTP web 框架。 它是一个类似于 martini 但拥有更好性能的 API 框架, 由于 httprouter，速度提高了近 40 倍。如果你需要极好的性能，使用 Gin 吧。

## 如何使用 Gin？

我们提供了一些 API 的使用示例 并罗列了一些众所周知的 Gin 用户.

## 如何为 Gin 做贡献？

- 在社区帮助其他人
- 告诉我们你使用 Gin 的成功案例
- 告诉我们如何改善 Gin 并帮助我们实现它
- 为已有的库做贡献

# 介绍

Gin 是一个用 Go (Golang) 编写的 web 框架。 它是一个类似于 martini 但拥有更好性能的 API 框架, 由于 httprouter，速度提高了近 40 倍。 如果你是性能和高效的追求者, 你会爱上 Gin.

在本节中，我们将介绍 Gin 是什么，它解决了哪些问题，以及它如何帮助你的项目。

或者, 如果你已经准备在项目中使用 Gin，请访问快速入门.

# 特性

## 快速

基于 Radix 树的路由，小内存占用。没有反射。可预测的 API 性能。

## 支持中间件

传入的 HTTP 请求可以由一系列中间件和最终操作来处理。例如: Logger，Authorization，GZIP，最终操作 DB。

## Crash 处理

Gin 可以 catch 一个发生在 HTTP 请求中的 panic 并 recover 它。这样，你的服务器将始终可用。例如，你可以向 Sentry 报告这个 panic！

## JSON 验证

Gin 可以解析并验证请求的 JSON，例如检查所需值的存在。

## 路由组

更好地组织路由。是否需要授权，不同的 API 版本…… 此外，这些组可以无限制地嵌套而不会降低性能。

## 错误管理

Gin 提供了一种方便的方法来收集 HTTP 请求期间发生的所有错误。最终，中间件可以将它们写入日志文件，数据库并通过网络发送。

## 内置渲染

Gin 为 JSON，XML 和 HTML 渲染提供了易于使用的 API。

## 可扩展性

新建一个中间件非常简单，去查看示例代码吧。

# 快速入门



## 要求

- Go 1.6 及以上版本

> 很快会需要 Go 1.8 版本.

## 安装

要安装 Gin 软件包，需要先安装 Go 并设置 Go 工作区。

1.下载并安装 gin:

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

2.将 gin 引入到代码中:

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

3.（可选）如果使用诸如 `http.StatusOK` 之类的常量，则需要引入 `net/http` 包:

```
1.  import "net/http"
```

### 使用 Govendor 工具创建项目

1. `go get` govendor

```
1.  $ go get github.com/kardianos/govendor
```

2.创建项目并且 `cd` 到项目目录中

```
1.  $ mkdir -p $GOPATH/src/github.com/myusername/project && cd "$_"
```

3.使用 govendor 初始化项目，并且引入 gin

```
$ govendor init
$ govendor fetch github.com/gin-gonic/gin@v1.3
```

4.复制启动文件模板到项目目录中

```
$ curl https://raw.githubusercontent.com/gin-gonic/examples/master/basic/main.go > main.go
```

5.启动项目

```
$ go run main.go
```

# 开始

> 不确定如何编写和执行 Go 代码? 点击这里.

首先，创建一个名为 `example.go` 的文件

```
$ touch example.go
```

接下来，将如下的代码写入 `example.go` 中:

```
package main

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

func main() {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    r.Run() // 监听并在 0.0.0.0:8080 上启动服务
}
```

然后，执行 `go run example.go` 命令来运行代码:

```
# 运行 example.go 并且在浏览器中访问 0.0.0.0:8080/ping
$ go run example.go
```

# 基准测试

Gin 使用了自定义版本的 HttpRouter

查看所有基准测试

| Benchmark name | (1) | (2) | (3) | (4) |
|---|---|---|---|---|
| **BenchmarkGin_GithubAll** | **30000** | **48375** | **0** | **0** |
| BenchmarkAce_GithubAll | 10000 | 134059 | 13792 | 167 |
| BenchmarkBear_GithubAll | 5000 | 534445 | 86448 | 943 |
| BenchmarkBeego_GithubAll | 3000 | 592444 | 74705 | 812 |
| BenchmarkBone_GithubAll | 200 | 6957308 | 698784 | 8453 |
| BenchmarkDenco_GithubAll | 10000 | 158819 | 20224 | 167 |
| BenchmarkEcho_GithubAll | 10000 | 154700 | 6496 | 203 |
| BenchmarkGocraftWeb_GithubAll | 3000 | 570806 | 131656 | 1686 |
| BenchmarkGoji_GithubAll | 2000 | 818034 | 56112 | 334 |
| BenchmarkGojiv2_GithubAll | 2000 | 1213973 | 274768 | 3712 |
| BenchmarkGoJsonRest_GithubAll | 2000 | 785796 | 134371 | 2737 |
| BenchmarkGoRestful_GithubAll | 300 | 5238188 | 689672 | 4519 |
| BenchmarkGorillaMux_GithubAll | 100 | 10257726 | 211840 | 2272 |
| BenchmarkHttpRouter_GithubAll | 20000 | 105414 | 13792 | 167 |
| BenchmarkHttpTreeMux_GithubAll | 10000 | 319934 | 65856 | 671 |
| BenchmarkKocha_GithubAll | 10000 | 209442 | 23304 | 843 |
| BenchmarkLARS_GithubAll | 20000 | 62565 | 0 | 0 |
| BenchmarkMacaron_GithubAll | 2000 | 1161270 | 204194 | 2000 |
| BenchmarkMartini_GithubAll | 200 | 9991713 | 226549 | 2325 |
| BenchmarkPat_GithubAll | 200 | 5590793 | 1499568 | 27435 |
| BenchmarkPossum_GithubAll | 10000 | 319768 | 84448 | 609 |
| BenchmarkR2router_GithubAll | 10000 | 305134 | 77328 | 979 |
| BenchmarkRivet_GithubAll | 10000 | 132134 | 16272 | 167 |
| BenchmarkTango_GithubAll | 3000 | 552754 | 63826 | 1618 |
| BenchmarkTigerTonic_GithubAll | 1000 | 1439483 | 239104 | 5374 |
| BenchmarkTraffic_GithubAll | 100 | 11383067 | 2659329 | 21848 |
| BenchmarkVulcan_GithubAll | 5000 | 394253 | 19894 | 609 |

- (1): 在一定的时间内实现的总调用数，越高越好
- (2): 单次操作耗时（ns/op），越低越好

- (3): 堆内存分配 （B/op）, 越低越好
- (4): 每次操作的平均内存分配次数（allocs/op）, 越低越好

# 特性

Gin v1 稳定的特性：

- 零分配路由。
- 仍然是最快的 http 路由器和框架。
- 完整的单元测试支持。
- 实战考验。
- API 冻结，新版本的发布不会破坏你的代码。

# Jsoniter

## 使用 jsoniter 编译

Gin 使用 `encoding/json` 作为默认的 json 包，但是你可以在编译中使用标签将其修改为 jsoniter。

```
1.  $ go build -tags=jsoniter .
```

# 示例

该节列出了 `api` 的用法。

# AsciiJSON

使用 AsciiJSON 生成具有转义的非 ASCII 字符的 ASCII-only JSON。

```
func main() {
    r := gin.Default()

    r.GET("/someJSON", func(c *gin.Context) {
        data := map[string]interface{}{
            "lang": "GO语言",
            "tag":  "<br>",
        }

        // 输出 : {"lang":"GO\u8bed\u8a00","tag":"\u003cbr\u003e"}
        c.AsciiJSON(http.StatusOK, data)
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

# HTML 渲染

使用 LoadHTMLGlob() 或者 LoadHTMLFiles()

```
func main() {
    router := gin.Default()
    router.LoadHTMLGlob("templates/*")
    //router.LoadHTMLFiles("templates/template1.html",
"templates/template2.html")
    router.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.tmpl", gin.H{
            "title": "Main website",
        })
    })
    router.Run(":8080")
}
```

templates/index.tmpl

```
<html>
    <h1>
        {{ .title }}
    </h1>
</html>
```

使用不同目录下名称相同的模板

```
func main() {
    router := gin.Default()
    router.LoadHTMLGlob("templates/**/*")
    router.GET("/posts/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "posts/index.tmpl", gin.H{
            "title": "Posts",
        })
    })
    router.GET("/users/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/index.tmpl", gin.H{
            "title": "Users",
        })
    })
```

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

templates/posts/index.tmpl

```
{{ define "posts/index.tmpl" }}
<html><h1>
    {{ .title }}
</h1>
<p>Using posts/index.tmpl</p>
</html>
{{ end }}
```

templates/users/index.tmpl

```
{{ define "users/index.tmpl" }}
<html><h1>
    {{ .title }}
</h1>
<p>Using users/index.tmpl</p>
</html>
{{ end }}
```

## 自定义模板渲染器

你可以使用自定义的 `html` 模板渲染

```
import "html/template"

func main() {
    router := gin.Default()
    html := template.Must(template.ParseFiles("file1", "file2"))
    router.SetHTMLTemplate(html)
    router.Run(":8080")
}
```

## 自定义分隔符

你可以使用自定义分隔

```
    r := gin.Default()
    r.Delims("{[{", "}]}")
```

```
3.      r.LoadHTMLGlob("/path/to/templates")
```

## 自定义模板功能

查看详细示例代码。

main.go

```
1.  import (
2.      "fmt"
3.      "html/template"
4.      "net/http"
5.      "time"
6.
7.      "github.com/gin-gonic/gin"
8.  )
9.
10. func formatAsDate(t time.Time) string {
11.     year, month, day := t.Date()
12.     return fmt.Sprintf("%d/%02d/%02d", year, month, day)
13. }
14.
15. func main() {
16.     router := gin.Default()
17.     router.Delims("{[{", "}]}")
18.     router.SetFuncMap(template.FuncMap{
19.         "formatAsDate": formatAsDate,
20.     })
21.     router.LoadHTMLFiles("./testdata/template/raw.tmpl")
22.
23.     router.GET("/raw", func(c *gin.Context) {
24.         c.HTML(http.StatusOK, "raw.tmpl", map[string]interface{}{
25.             "now": time.Date(2017, 07, 01, 0, 0, 0, 0, time.UTC),
26.         })
27.     })
28.
29.     router.Run(":8080")
30. }
```

raw.tmpl

```
1.  Date: {[{.now | formatAsDate}]}
```

**结果：**

```
Date: 2017/07/01
```

# HTTP2 server 推送

http.Pusher 仅支持 **go1.8+**。 更多信息，请查阅 golang blog。

```
package main

import (
    "html/template"
    "log"

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

var html = template.Must(template.New("https").Parse(`
<html>
<head>
  <title>Https Test</title>
  <script src="/assets/app.js"></script>
</head>
<body>
  <h1 style="color:red;">Welcome, Ginner!</h1>
</body>
</html>
`))

func main() {
    r := gin.Default()
    r.Static("/assets", "./assets")
    r.SetHTMLTemplate(html)

    r.GET("/", func(c *gin.Context) {
        if pusher := c.Writer.Pusher(); pusher != nil {
            // 使用 pusher.Push() 做服务器推送
            if err := pusher.Push("/assets/app.js", nil); err != nil {
                log.Printf("Failed to push: %v", err)
            }
        }
        c.HTML(200, "https", gin.H{
            "status": "success",
        })
    })
```

```
38.
39.     // 监听并在 https://127.0.0.1:8080 上启动服务
40.     r.RunTLS(":8080", "./testdata/server.pem", "./testdata/server.key")
41. }
```

# JSONP

使用 JSONP 向不同域的服务器请求数据。如果查询参数存在回调，则将回调添加到响应体中。

```
func main() {
    r := gin.Default()

    r.GET("/JSONP?callback=x", func(c *gin.Context) {
        data := map[string]interface{}{
            "foo": "bar",
        }

        // callback 是 x
        // 将输出：x({\"foo\":\"bar\"})
        c.JSONP(http.StatusOK, data)
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

# Multipart/Urlencoded 绑定

```go
package main

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

type LoginForm struct {
    User     string `form:"user" binding:"required"`
    Password string `form:"password" binding:"required"`
}

func main() {
    router := gin.Default()
    router.POST("/login", func(c *gin.Context) {
        // 你可以使用显式绑定声明绑定 multipart form：
        // c.ShouldBindWith(&form, binding.Form)
        // 或者简单地使用 ShouldBind 方法自动绑定：
        var form LoginForm
        // 在这种情况下，将自动选择合适的绑定
        if c.ShouldBind(&form) == nil {
            if form.User == "user" && form.Password == "password" {
                c.JSON(200, gin.H{"status": "you are logged in"})
            } else {
                c.JSON(401, gin.H{"status": "unauthorized"})
            }
        }
    })
    router.Run(":8080")
}
```

**测试：**

```
$ curl -v --form user=user --form password=password http://localhost:8080/login
```

# Multipart/Urlencoded 表单

```go
func main() {
    router := gin.Default()

    router.POST("/form_post", func(c *gin.Context) {
        message := c.PostForm("message")
        nick := c.DefaultPostForm("nick", "anonymous")

        c.JSON(200, gin.H{
            "status":  "posted",
            "message": message,
            "nick":    nick,
        })
    })
    router.Run(":8080")
}
```

# PureJSON

通常，JSON 使用 unicode 替换特殊 HTML 字符，例如 < 变为 \ u003c。如果要按字面对这些字符进行编码，则可以使用 PureJSON。Go 1.6 及更低版本无法使用此功能。

```
1.  func main() {
2.      r := gin.Default()
3.
4.      // 提供 unicode 实体
5.      r.GET("/json", func(c *gin.Context) {
6.          c.JSON(200, gin.H{
7.              "html": "<b>Hello, world!</b>",
8.          })
9.      })
10.
11.     // 提供字面字符
12.     r.GET("/purejson", func(c *gin.Context) {
13.         c.PureJSON(200, gin.H{
14.             "html": "<b>Hello, world!</b>",
15.         })
16.     })
17.
18.     // 监听并在 0.0.0.0:8080 上启动服务
19.     r.Run(":8080")
20. }
```

# Query 和 post form

```
POST /post?id=1234&page=1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

name=manu&message=this_is_great
```

```
func main() {
    router := gin.Default()

    router.POST("/post", func(c *gin.Context) {

        id := c.Query("id")
        page := c.DefaultQuery("page", "0")
        name := c.PostForm("name")
        message := c.PostForm("message")

        fmt.Printf("id: %s; page: %s; name: %s; message: %s", id, page, name,
message)
    })
    router.Run(":8080")
}
```

```
id: 1234; page: 1; name: manu; message: this_is_great
```

# SecureJSON

使用 SecureJSON 防止 json 劫持。如果给定的结构是数组值，则默认预置 `"while(1),"` 到响应体。

```
1.  func main() {
2.      r := gin.Default()
3.
4.      // 你也可以使用自己的 SecureJSON 前缀
5.      // r.SecureJsonPrefix(")]}',\n")
6.
7.      r.GET("/someJSON", func(c *gin.Context) {
8.          names := []string{"lena", "austin", "foo"}
9.
10.         // 将输出：while(1);["lena","austin","foo"]
11.         c.SecureJSON(http.StatusOK, names)
12.     })
13.
14.     // 监听并在 0.0.0.0:8080 上启动服务
15.     r.Run(":8080")
16. }
```

# XML/JSON/YAML/ProtoBuf 渲染

```
func main() {
    r := gin.Default()

    // gin.H 是 map[string]interface{} 的一种快捷方式
    r.GET("/someJSON", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/moreJSON", func(c *gin.Context) {
        // 你也可以使用一个结构体
        var msg struct {
            Name    string `json:"user"`
            Message string
            Number  int
        }
        msg.Name = "Lena"
        msg.Message = "hey"
        msg.Number = 123
        // 注意 msg.Name 在 JSON 中变成了 "user"
        // 将输出：{"user": "Lena", "Message": "hey", "Number": 123}
        c.JSON(http.StatusOK, msg)
    })

    r.GET("/someXML", func(c *gin.Context) {
        c.XML(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/someYAML", func(c *gin.Context) {
        c.YAML(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/someProtoBuf", func(c *gin.Context) {
        reps := []int64{int64(1), int64(2)}
        label := "test"
        // protobuf 的具体定义写在 testdata/protoexample 文件中。
        data := &protoexample.Test{
            Label: &label,
            Reps:  reps,
```

```
        }
        // 请注意，数据在响应中变为二进制数据
        // 将输出被 protoexample.Test protobuf 序列化了的数据
        c.ProtoBuf(http.StatusOK, data)
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

# 上传文件

本节列出了上传图片的 `api` 用法。

# 单文件

参考 issue #774 和详细示例代码.

```go
func main() {
    router := gin.Default()
    // 为 multipart forms 设置较低的内存限制 (默认是 32 MiB)
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // 单文件
        file, _ := c.FormFile("file")
        log.Println(file.Filename)

        // 上传文件至指定目录
        // c.SaveUploadedFile(file, dst)

        c.String(http.StatusOK, fmt.Sprintf("'%s' uploaded!", file.Filename))
    })
    router.Run(":8080")
}
```

如何使用 `curl`:

```
curl -X POST http://localhost:8080/upload \
  -F "file=@/Users/appleboy/test.zip" \
  -H "Content-Type: multipart/form-data"
```

# 多文件

查看详细示例代码.

```
func main() {
    router := gin.Default()
    // 为 multipart forms 设置较低的内存限制 (默认是 32 MiB)
    // router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // Multipart form
        form, _ := c.MultipartForm()
        files := form.File["upload[]"]

        for _, file := range files {
            log.Println(file.Filename)

            // 上传文件至指定目录
            // c.SaveUploadedFile(file, dst)
        }
        c.String(http.StatusOK, fmt.Sprintf("%d files uploaded!", len(files)))
    })
    router.Run(":8080")
}
```

如何使用 `curl` :

```
curl -X POST http://localhost:8080/upload \
  -F "upload[]=@/Users/appleboy/test1.zip" \
  -F "upload[]=@/Users/appleboy/test2.zip" \
  -H "Content-Type: multipart/form-data"
```

# 不使用默认的中间件

使用

```
r := gin.New()
```

代替

```
// Default 使用 Logger 和 Recovery 中间件
r := gin.Default()
```

# 从 reader 读取数据

```
1.  func main() {
2.      router := gin.Default()
3.      router.GET("/someDataFromReader", func(c *gin.Context) {
            response, err := http.Get("https://raw.githubusercontent.com/gin-
4.  gonic/logo/master/color.png")
5.          if err != nil || response.StatusCode != http.StatusOK {
6.              c.Status(http.StatusServiceUnavailable)
7.              return
8.          }
9.
10.         reader := response.Body
11.         contentLength := response.ContentLength
12.         contentType := response.Header.Get("Content-Type")
13.
14.         extraHeaders := map[string]string{
15.             "Content-Disposition": `attachment; filename="gopher.png"`,
16.         }
17.
            c.DataFromReader(http.StatusOK, contentLength, contentType, reader,
18.  extraHeaders)
19.     })
20.     router.Run(":8080")
21. }
```

# 优雅地重启或停止

你想优雅地重启或停止 web 服务器吗？有一些方法可以做到这一点。

我们可以使用 fvbock/endless 来替换默认的 `ListenAndServe` 。更多详细信息，请参阅 issue #296。

```
router := gin.Default()
router.GET("/", handler)
// [...]
endless.ListenAndServe(":4242", router)
```

替代方案：

- manners：可以优雅关机的 Go Http 服务器。
- graceful：Graceful 是一个 Go 扩展包，可以优雅地关闭 http.Handler 服务器。
- grace：Go 服务器平滑重启和零停机时间部署。如果你使用的是 Go 1.8，可以不需要这些库！考虑使用 http.Server 内置的 Shutdown() 方法优雅地关机. 请参阅 gin 完整的 graceful-shutdown 示例。

```
// +build go1.8

package main

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

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

func main() {
    router := gin.Default()
    router.GET("/", func(c *gin.Context) {
        time.Sleep(5 * time.Second)
        c.String(http.StatusOK, "Welcome Gin Server")
    })
```

```
22.
23.     srv := &http.Server{
24.         Addr:    ":8080",
25.         Handler: router,
26.     }
27.
28.     go func() {
29.         // 服务连接
            if err := srv.ListenAndServe(); err != nil && err !=
30. http.ErrServerClosed {
31.             log.Fatalf("listen: %s\n", err)
32.         }
33.     }()
34.
35.     // 等待中断信号以优雅地关闭服务器（设置 5 秒的超时时间）
36.     quit := make(chan os.Signal)
37.     signal.Notify(quit, os.Interrupt)
38.     <-quit
39.     log.Println("Shutdown Server ...")
40.
41.     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
42.     defer cancel()
43.     if err := srv.Shutdown(ctx); err != nil {
44.         log.Fatal("Server Shutdown:", err)
45.     }
46.     log.Println("Server exiting")
47. }
```

# 使用 BasicAuth 中间件

```
1.  // 模拟一些私人数据
2.  var secrets = gin.H{
3.      "foo":    gin.H{"email": "foo@bar.com", "phone": "123433"},
4.      "austin": gin.H{"email": "austin@example.com", "phone": "666"},
5.      "lena":   gin.H{"email": "lena@guapa.com", "phone": "523443"},
6.  }
7.
8.  func main() {
9.      r := gin.Default()
10.
11.     // 路由组使用 gin.BasicAuth() 中间件
12.     // gin.Accounts 是 map[string]string 的一种快捷方式
13.     authorized := r.Group("/admin", gin.BasicAuth(gin.Accounts{
14.         "foo":    "bar",
15.         "austin": "1234",
16.         "lena":   "hello2",
17.         "manu":   "4321",
18.     }))
19.
20.     // /admin/secrets 端点
21.     // 触发 "localhost:8080/admin/secrets
22.     authorized.GET("/secrets", func(c *gin.Context) {
23.         // 获取用户，它是由 BasicAuth 中间件设置的
24.         user := c.MustGet(gin.AuthUserKey).(string)
25.         if secret, ok := secrets[user]; ok {
26.             c.JSON(http.StatusOK, gin.H{"user": user, "secret": secret})
27.         } else {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": "NO SECRET :
28. ("})
29.         }
30.     })
31.
32.     // 监听并在 0.0.0.0:8080 上启动服务
33.     r.Run(":8080")
34. }
```

# 使用 HTTP 方法

```go
func main() {
    // 禁用控制台颜色
    // gin.DisableConsoleColor()

    // 使用默认中间件（logger 和 recovery 中间件）创建 gin 路由
    router := gin.Default()

    router.GET("/someGet", getting)
    router.POST("/somePost", posting)
    router.PUT("/somePut", putting)
    router.DELETE("/someDelete", deleting)
    router.PATCH("/somePatch", patching)
    router.HEAD("/someHead", head)
    router.OPTIONS("/someOptions", options)

    // 默认在 8080 端口启动服务，除非定义了一个 PORT 的环境变量。
    router.Run()
    // router.Run(":3000") hardcode 端口号
}
```

# 使用中间件

```go
func main() {
    // 新建一个没有任何默认中间件的路由
    r := gin.New()

    // 全局中间件
    // Logger 中间件将日志写入 gin.DefaultWriter，即使你将 GIN_MODE 设置为 release。
    // By default gin.DefaultWriter = os.Stdout
    r.Use(gin.Logger())

    // Recovery 中间件会 recover 任何 panic。如果有 panic 的话，会写入 500。
    r.Use(gin.Recovery())

    // 你可以为每个路由添加任意数量的中间件。
    r.GET("/benchmark", MyBenchLogger(), benchEndpoint)

    // 认证路由组
    // authorized := r.Group("/", AuthRequired())
    // 和使用以下两行代码的效果完全一样:
    authorized := r.Group("/")
    // 路由组中间件! 在此例中，我们在 "authorized" 路由组中使用自定义创建的
    // AuthRequired() 中间件
    authorized.Use(AuthRequired())
    {
        authorized.POST("/login", loginEndpoint)
        authorized.POST("/submit", submitEndpoint)
        authorized.POST("/read", readEndpoint)

        // 嵌套路由组
        testing := authorized.Group("testing")
        testing.GET("/analytics", analyticsEndpoint)
    }

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

# 只绑定 url 查询字符串

`ShouldBindQuery` 函数只绑定 url 查询参数而忽略 post 数据。参阅详细信息.

```
1.  package main
2.
3.  import (
4.      "log"
5.
6.      "github.com/gin-gonic/gin"
7.  )
8.
9.  type Person struct {
10.     Name    string `form:"name"`
11.     Address string `form:"address"`
12. }
13.
14. func main() {
15.     route := gin.Default()
16.     route.Any("/testing", startPage)
17.     route.Run(":8085")
18. }
19.
20. func startPage(c *gin.Context) {
21.     var person Person
22.     if c.ShouldBindQuery(&person) == nil {
23.         log.Println("====== Only Bind By Query String ======")
24.         log.Println(person.Name)
25.         log.Println(person.Address)
26.     }
27.     c.String(200, "Success")
28. }
```

# 在中间件中使用 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，必须使用只读副本。

```
1.  func main() {
2.      r := gin.Default()
3.
4.      r.GET("/long_async", func(c *gin.Context) {
5.          // 创建在 goroutine 中使用的副本
6.          cCp := c.Copy()
7.          go func() {
8.              // 用 time.Sleep() 模拟一个长任务。
9.              time.Sleep(5 * time.Second)
10.
11.             // 请注意您使用的是复制的上下文 "cCp"，这一点很重要
12.             log.Println("Done! in path " + cCp.Request.URL.Path)
13.         }()
14.     })
15.
16.     r.GET("/long_sync", func(c *gin.Context) {
17.         // 用 time.Sleep() 模拟一个长任务。
18.         time.Sleep(5 * time.Second)
19.
20.         // 因为没有使用 goroutine，不需要拷贝上下文
21.         log.Println("Done! in path " + c.Request.URL.Path)
22.     })
23.
24.     // 监听并在 0.0.0.0:8080 上启动服务
25.     r.Run(":8080")
26. }
```

# 多模板

Gin 默认允许只使用一个 html 模板。 查看多模板渲染 以使用 go 1.6 `block template` 等功能。

# 如何记录日志

```go
func main() {
    // 禁用控制台颜色，将日志写入文件时不需要控制台颜色。
    gin.DisableConsoleColor()

    // 记录到文件。
    f, _ := os.Create("gin.log")
    gin.DefaultWriter = io.MultiWriter(f)

    // 如果需要同时将日志写入文件和控制台，请使用以下代码。
    // gin.DefaultWriter = io.MultiWriter(f, os.Stdout)

    router := gin.Default()
    router.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

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

# 定义路由日志的格式

默认的路由日志格式：

```
[GIN-debug] POST   /foo                      --> main.main.func1 (3 handlers)
[GIN-debug] GET    /bar                      --> main.main.func2 (3 handlers)
[GIN-debug] GET    /status                   --> main.main.func3 (3 handlers)
```

如果你想要以指定的格式（例如 JSON，key values 或其他格式）记录信息，则可以使用 `gin.DebugPrintRouteFunc` 指定格式。在下面的示例中，我们使用标准日志包记录所有路由，但你可以使用其他满足你需求的日志工具。

```
import (
    "log"
    "net/http"

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

func main() {
    r := gin.Default()
    gin.DebugPrintRouteFunc = func(httpMethod, absolutePath, handlerName
string, nuHandlers int) {
        log.Printf("endpoint %v %v %v %v\n", httpMethod, absolutePath,
handlerName, nuHandlers)
    }

    r.POST("/foo", func(c *gin.Context) {
        c.JSON(http.StatusOK, "foo")
    })

    r.GET("/bar", func(c *gin.Context) {
        c.JSON(http.StatusOK, "bar")
    })

    r.GET("/status", func(c *gin.Context) {
        c.JSON(http.StatusOK, "ok")
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run()
```

```
28. }
```

# 将 request body 绑定到不同的结构体中

一般通过调用 `c.Request.Body` 方法绑定数据，但不能多次调用这个方法。

```
type formA struct {
  Foo string `json:"foo" xml:"foo" binding:"required"`
}

type formB struct {
  Bar string `json:"bar" xml:"bar" binding:"required"`
}

func SomeHandler(c *gin.Context) {
  objA := formA{}
  objB := formB{}
  // c.ShouldBind 使用了 c.Request.Body, 不可重用。
  if errA := c.ShouldBind(&objA); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
  // 因为现在 c.Request.Body 是 EOF, 所以这里会报错。
  } else if errB := c.ShouldBind(&objB); errB == nil {
    c.String(http.StatusOK, `the body should be formB`)
  } else {
    ...
  }
}
```

要想多次绑定，可以使用 `c.ShouldBindBodyWith` .

```
func SomeHandler(c *gin.Context) {
  objA := formA{}
  objB := formB{}
  // 读取 c.Request.Body 并将结果存入上下文。
  if errA := c.ShouldBindBodyWith(&objA, binding.JSON); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
  // 这时, 复用存储在上下文中的 body。
  } else if errB := c.ShouldBindBodyWith(&objB, binding.JSON); errB == nil {
    c.String(http.StatusOK, `the body should be formB JSON`)
  // 可以接受其他格式
  } else if errB2 := c.ShouldBindBodyWith(&objB, binding.XML); errB2 == nil {
    c.String(http.StatusOK, `the body should be formB XML`)
```

```
13.     } else {
14.       ...
15.     }
16. }
```

- `c.ShouldBindBodyWith` 会在绑定之前将 body 存储到上下文中。 这会对性能造成轻微影响，如果调用一次就能完成绑定的话，那就不要用这个方法。
- 只有某些格式需要此功能，如 `JSON`， `XML`， `MsgPack`，`ProtoBuf` 。 对于其他格式， 如 `Query`， `Form`， `FormPost`， `FormMultipart`可以多次调用 `c.ShouldBind()` 而不会造成任任何性能损失 (详见 #1341)。

# 支持 Let's Encrypt

一行代码支持 LetsEncrypt HTTPS servers 示例。

```go
package main

import (
    "log"

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

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

    // Ping handler
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    log.Fatal(autotls.Run(r, "example1.com", "example2.com"))
}
```

自定义 autocert manager 示例。

```go
package main

import (
    "log"

    "github.com/gin-gonic/autotls"
    "github.com/gin-gonic/gin"
    "golang.org/x/crypto/acme/autocert"
)

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

    // Ping handler
```

```
15.     r.GET("/ping", func(c *gin.Context) {
16.         c.String(200, "pong")
17.     })
18.
19.     m := autocert.Manager{
20.         Prompt:     autocert.AcceptTOS,
21.         HostPolicy: autocert.HostWhitelist("example1.com", "example2.com"),
22.         Cache:      autocert.DirCache("/var/www/.cache"),
23.     }
24.
25.     log.Fatal(autotls.RunWithManager(r, &m))
26. }
```

# 映射查询字符串或表单参数

```
POST /post?ids[a]=1234&ids[b]=hello HTTP/1.1
Content-Type: application/x-www-form-urlencoded

names[first]=thinkerou&names[second]=tianou
```

```
func main() {
    router := gin.Default()

    router.POST("/post", func(c *gin.Context) {

        ids := c.QueryMap("ids")
        names := c.PostFormMap("names")

        fmt.Printf("ids: %v; names: %v", ids, names)
    })
    router.Run(":8080")
}
```

```
ids: map[b:hello a:1234], names: map[second:tianou first:thinkerou]
```

# 查询字符串参数

```
1.  func main() {
2.      router := gin.Default()
3.
4.      // 使用现有的基础请求对象解析查询字符串参数。
5.      // 示例 URL： /welcome?firstname=Jane&lastname=Doe
6.      router.GET("/welcome", func(c *gin.Context) {
7.          firstname := c.DefaultQuery("firstname", "Guest")
            lastname := c.Query("lastname") //
8.  c.Request.URL.Query().Get("lastname") 的一种快捷方式
9.
10.         c.String(http.StatusOK, "Hello %s %s", firstname, lastname)
11.     })
12.     router.Run(":8080")
13. }
```

# 模型绑定和验证

要将请求体绑定到结构体中，使用模型绑定。 Gin目前支持JSON、XML、YAML和标准表单值的绑定（foo=bar&boo=baz）。

Gin使用 **go-playground/validator.v8** 进行验证。 查看标签用法的全部文档.

使用时，需要在要绑定的所有字段上，设置相应的tag。 例如，使用 JSON 绑定时，设置字段标签为 `json:"fieldname"` 。

Gin提供了两类绑定方法:

- **Type** - Must bind
  - **Methods** - `Bind` , `BindJSON` , `BindXML` , `BindQuery` , `BindYAML`
  - **Behavior** - 这些方法属于 `MustBindWith` 的具体调用。 如果发生绑定错误，则请求终止，并触发 `c.AbortWithError(400, err).SetType(ErrorTypeBind)` 。响应状态码被设置为 400 并且 `Content-Type` 被设置为 `text/plain; charset=utf-8` 。 如果您在此之后尝试设置响应状态码，Gin会输出日志 `[GIN-debug] [WARNING] Headers were already written. Wanted to override status code 400 with 422` 。 如果您希望更好地控制绑定，考虑使用 `ShouldBind` 等效方法。
- **Type** - Should bind
  - **Methods** - `ShouldBind` , `ShouldBindJSON` , `ShouldBindXML` , `ShouldBindQuery` , `ShouldBindYAML`
  - **Behavior** - 这些方法属于 `ShouldBindWith` 的具体调用。 如果发生绑定错误，Gin会返回错误并由开发者处理错误和请求。使用 Bind 方法时，Gin 会尝试根据 Content-Type 推断如何绑定。 如果你明确知道要绑定什么，可以使用 `MustBindWith` 或 `ShouldBindWith` 。

你也可以指定必须绑定的字段。 如果一个字段的 tag 加上了 `binding:"required"` ，但绑定时是空值, Gin 会报错。

```
1. // 绑定 JSON
2. type Login struct {
3.     User     string `form:"user" json:"user" xml:"user"  binding:"required"`
       Password string `form:"password" json:"password" xml:"password"
4. binding:"required"`
5. }
6.
7. func main() {
8.     router := gin.Default()
9.
```

```
10.     // 绑定 JSON ({"user": "manu", "password": "123"})
11.     router.POST("/loginJSON", func(c *gin.Context) {
12.         var json Login
13.         if err := c.ShouldBindJSON(&json); err != nil {
14.             c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
15.             return
16.         }
17.
18.         if json.User != "manu" || json.Password != "123" {
19.             c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
20.             return
21.         }
22.
23.         c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
24.     })
25.
26.     // 绑定 XML (
27.     //    <?xml version="1.0" encoding="UTF-8"?>
28.     //    <root>
29.     //        <user>user</user>
30.     //        <password>123</password>
31.     //    </root>)
32.     router.POST("/loginXML", func(c *gin.Context) {
33.         var xml Login
34.         if err := c.ShouldBindXML(&xml); err != nil {
35.             c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
36.             return
37.         }
38.
39.         if xml.User != "manu" || xml.Password != "123" {
40.             c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
41.             return
42.         }
43.
44.         c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
45.     })
46.
47.     // 绑定 HTML 表单 (user=manu&password=123)
48.     router.POST("/loginForm", func(c *gin.Context) {
49.         var form Login
50.         // 根据 Content-Type Header 推断使用哪个绑定器。
51.         if err := c.ShouldBind(&form); err != nil {
```

```
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

        if form.User != "manu" || form.Password != "123" {
            c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
            return
        }

        c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    router.Run(":8080")
}
```

## 示例请求

```
$ curl -v -X POST \
  http://localhost:8080/loginJSON \
  -H 'content-type: application/json' \
  -d '{ "user": "manu" }'
> POST /loginJSON HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.51.0
> Accept: */*
> content-type: application/json
> Content-Length: 18
>
* upload completely sent off: 18 out of 18 bytes
< HTTP/1.1 400 Bad Request
< Content-Type: application/json; charset=utf-8
< Date: Fri, 04 Aug 2017 03:51:31 GMT
< Content-Length: 100
<
{"error":"Key: 'Login.Password' Error:Field validation for 'Password' failed on the 'required' tag"}
```

## 忽略验证

使用上述的 `curl` 命令运行上面的示例时会返回错误。 因为示例中 `Password` 使用了 `binding:"required"` 。 如果 `Password` 使用 `binding:"-"` ， 再次运行上面的示例就不

会返回错误。

# 绑定 HTML 复选框

参见详细信息

main.go

```
...

type myForm struct {
    Colors []string `form:"colors[]"`
}

...

func formHandler(c *gin.Context) {
    var fakeForm myForm
    c.ShouldBind(&fakeForm)
    c.JSON(200, gin.H{"color": fakeForm.Colors})
}

...
```

form.html

```
<form action="/" method="POST">
    <p>Check some colors</p>
    <label for="red">Red</label>
    <input type="checkbox" name="colors[]" value="red" id="red">
    <label for="green">Green</label>
    <input type="checkbox" name="colors[]" value="green" id="green">
    <label for="blue">Blue</label>
    <input type="checkbox" name="colors[]" value="blue" id="blue">
    <input type="submit">
</form>
```

结果：

```
{"color":["red","green","blue"]}
```

# 绑定 Uri

查看详细信息.

```
package main

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

type Person struct {
    ID string `uri:"id" binding:"required,uuid"`
    Name string `uri:"name" binding:"required"`
}

func main() {
    route := gin.Default()
    route.GET("/:name/:id", func(c *gin.Context) {
        var person Person
        if err := c.ShouldBindUri(&person); err != nil {
            c.JSON(400, gin.H{"msg": err})
            return
        }
        c.JSON(200, gin.H{"name": person.Name, "uuid": person.ID})
    })
    route.Run(":8088")
}
```

测试:

```
$ curl -v localhost:8088/thinkerou/987fbc97-4bed-5078-9f07-9141ba07c9f3
$ curl -v localhost:8088/thinkerou/not-uuid
```

# 绑定查询字符串或表单数据

查看详细信息。

```
package main

import (
    "log"
    "time"

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

type Person struct {
    Name     string    `form:"name"`
    Address  string    `form:"address"`
    Birthday time.Time `form:"birthday" time_format:"2006-01-02" time_utc:"1"`
}

func main() {
    route := gin.Default()
    route.GET("/testing", startPage)
    route.Run(":8085")
}

func startPage(c *gin.Context) {
    var person Person
    // 如果是 `GET` 请求，只使用 `Form` 绑定引擎（`query`）。
    // 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`）。
    // 查看更多：https://github.com/gin-gonic/gin/blob/master/binding/binding.go#L48
    if c.ShouldBind(&person) == nil {
        log.Println(person.Name)
        log.Println(person.Address)
        log.Println(person.Birthday)
    }

    c.String(200, "Success")
}
```

**测试：**

```
$ curl -X GET "localhost:8085/testing?name=appleboy&address=xyz&birthday=1992-03-15"
```

# 绑定表单数据至自定义结构体

以下示例使用自定义结构体：

```
type StructA struct {
    FieldA string `form:"field_a"`
}

type StructB struct {
    NestedStruct StructA
    FieldB string `form:"field_b"`
}

type StructC struct {
    NestedStructPointer *StructA
    FieldC string `form:"field_c"`
}

type StructD struct {
    NestedAnonyStruct struct {
        FieldX string `form:"field_x"`
    }
    FieldD string `form:"field_d"`
}

func GetDataB(c *gin.Context) {
    var b StructB
    c.Bind(&b)
    c.JSON(200, gin.H{
        "a": b.NestedStruct,
        "b": b.FieldB,
    })
}

func GetDataC(c *gin.Context) {
    var b StructC
    c.Bind(&b)
    c.JSON(200, gin.H{
        "a": b.NestedStructPointer,
        "c": b.FieldC,
    })
```

```
}

func GetDataD(c *gin.Context) {
    var b StructD
    c.Bind(&b)
    c.JSON(200, gin.H{
        "x": b.NestedAnonyStruct,
        "d": b.FieldD,
    })
}

func main() {
    r := gin.Default()
    r.GET("/getb", GetDataB)
    r.GET("/getc", GetDataC)
    r.GET("/getd", GetDataD)

    r.Run()
}
```

使用 `curl` 命令结果：

```
$ curl "http://localhost:8080/getb?field_a=hello&field_b=world"
{"a":{"FieldA":"hello"},"b":"world"}
$ curl "http://localhost:8080/getc?field_a=hello&field_c=world"
{"a":{"FieldA":"hello"},"c":"world"}
$ curl "http://localhost:8080/getd?field_x=hello&field_d=world"
{"d":"world","x":{"FieldX":"hello"}}
```

注意：不支持以下格式结构体：

```
type StructX struct {
    X struct {} `form:"name_x"` // 有 form
}

type StructY struct {
    Y StructX `form:"name_y"` // 有 form
}

type StructZ struct {
    Z *StructZ `form:"name_z"` // 有 form
}
```

总之，目前仅支持没有 `form` 的嵌套结构体。

# 自定义 HTTP 配置

直接使用 `http.ListenAndServe()`，如下所示：

```
1. func main() {
2.     router := gin.Default()
3.     http.ListenAndServe(":8080", router)
4. }
```

或

```
1. func main() {
2.     router := gin.Default()
3.
4.     s := &http.Server{
5.         Addr:           ":8080",
6.         Handler:        router,
7.         ReadTimeout:    10 * time.Second,
8.         WriteTimeout:   10 * time.Second,
9.         MaxHeaderBytes: 1 << 20,
10.    }
11.    s.ListenAndServe()
12. }
```

# 自定义中间件

```go
func Logger() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()

        // 设置 example 变量
        c.Set("example", "12345")

        // 请求前

        c.Next()

        // 请求后
        latency := time.Since(t)
        log.Print(latency)

        // 获取发送的 status
        status := c.Writer.Status()
        log.Println(status)
    }
}

func main() {
    r := gin.New()
    r.Use(Logger())

    r.GET("/test", func(c *gin.Context) {
        example := c.MustGet("example").(string)

        // 打印："12345"
        log.Println(example)
    })

    // 监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

# 自定义验证器

注册自定义验证器，查看示例代码.

```
package main

import (
    "net/http"
    "reflect"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/binding"
    "gopkg.in/go-playground/validator.v8"
)

// Booking 包含绑定和验证的数据。
type Booking struct {
    CheckIn  time.Time `form:"check_in" binding:"required,bookabledate" time_format:"2006-01-02"`
    CheckOut time.Time `form:"check_out" binding:"required,gtfield=CheckIn" time_format:"2006-01-02"`
}

func bookableDate(
    v *validator.Validate, topStruct reflect.Value, currentStructOrField reflect.Value,
    field reflect.Value, fieldType reflect.Type, fieldKind reflect.Kind, param string,
) bool {
    if date, ok := field.Interface().(time.Time); ok {
        today := time.Now()
        if today.Year() > date.Year() || today.YearDay() > date.YearDay() {
            return false
        }
    }
    return true
}

func main() {
    route := gin.Default()
```

```
34.
35.     if v, ok := binding.Validator.Engine().(*validator.Validate); ok {
36.         v.RegisterValidation("bookabledate", bookableDate)
37.     }
38.
39.     route.GET("/bookable", getBookable)
40.     route.Run(":8085")
41. }
42.
43. func getBookable(c *gin.Context) {
44.     var b Booking
45.     if err := c.ShouldBindWith(&b, binding.Query); err == nil {
46.         c.JSON(http.StatusOK, gin.H{"message": "Booking dates are valid!"})
47.     } else {
48.         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
49.     }
50. }
```

```
1. $ curl "localhost:8085/bookable?check_in=2018-04-16&check_out=2018-04-17"
2. {"message":"Booking dates are valid!"}
3.
4. $ curl "localhost:8085/bookable?check_in=2018-03-08&check_out=2018-03-09"
   {"error":"Key: 'Booking.CheckIn' Error:Field validation for 'CheckIn' failed on
5. the 'bookabledate' tag"}
```

结构体级别的验证器 也可以通过其他的方式注册。更多信息请参阅 struct-lvl-validation 示例。

# 设置和获取 Cookie

```
import (
    "fmt"

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

func main() {

    router := gin.Default()

    router.GET("/cookie", func(c *gin.Context) {

        cookie, err := c.Cookie("gin_cookie")

        if err != nil {
            cookie = "NotSet"
            c.SetCookie("gin_cookie", "test", 3600, "/", "localhost", false,
true)
        }

        fmt.Printf("Cookie value: %s \n", cookie)
    })

    router.Run()
}
```

# 路由参数

```go
func main() {
    router := gin.Default()

    // 此 handler 将匹配 /user/john 但不会匹配 /user/ 或者 /user
    router.GET("/user/:name", func(c *gin.Context) {
        name := c.Param("name")
        c.String(http.StatusOK, "Hello %s", name)
    })

    // 此 handler 将匹配 /user/john/ 和 /user/john/send
    // 如果没有其他路由匹配 /user/john, 它将重定向到 /user/john/
    router.GET("/user/:name/*action", func(c *gin.Context) {
        name := c.Param("name")
        action := c.Param("action")
        message := name + " is " + action
        c.String(http.StatusOK, message)
    })

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

# 路由组

```
1.  func main() {
2.      router := gin.Default()
3.
4.      // 简单的路由组: v1
5.      v1 := router.Group("/v1")
6.      {
7.          v1.POST("/login", loginEndpoint)
8.          v1.POST("/submit", submitEndpoint)
9.          v1.POST("/read", readEndpoint)
10.     }
11.
12.     // 简单的路由组: v2
13.     v2 := router.Group("/v2")
14.     {
15.         v2.POST("/login", loginEndpoint)
16.         v2.POST("/submit", submitEndpoint)
17.         v2.POST("/read", readEndpoint)
18.     }
19.
20.     router.Run(":8080")
21. }
```

# 运行多个服务

请参阅 `issues` 并尝试以下示例:

```
1.  package main
2.
3.  import (
4.      "log"
5.      "net/http"
6.      "time"
7.
8.      "github.com/gin-gonic/gin"
9.      "golang.org/x/sync/errgroup"
10. )
11.
12. var (
13.     g errgroup.Group
14. )
15.
16. func router01() http.Handler {
17.     e := gin.New()
18.     e.Use(gin.Recovery())
19.     e.GET("/", func(c *gin.Context) {
20.         c.JSON(
21.             http.StatusOK,
22.             gin.H{
23.                 "code":  http.StatusOK,
24.                 "error": "Welcome server 01",
25.             },
26.         )
27.     })
28.
29.     return e
30. }
31.
32. func router02() http.Handler {
33.     e := gin.New()
34.     e.Use(gin.Recovery())
35.     e.GET("/", func(c *gin.Context) {
36.         c.JSON(
37.             http.StatusOK,
```

```
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 02",
            },
        )
    })

    return e
}

func main() {
    server01 := &http.Server{
        Addr:         ":8080",
        Handler:      router01(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    server02 := &http.Server{
        Addr:         ":8081",
        Handler:      router02(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    g.Go(func() error {
        return server01.ListenAndServe()
    })

    g.Go(func() error {
        return server02.ListenAndServe()
    })

    if err := g.Wait(); err != nil {
        log.Fatal(err)
    }
}
```

# 重定向

HTTP 重定向很容易。 内部、外部重定向均支持。

```
1. r.GET("/test", func(c *gin.Context) {
2.     c.Redirect(http.StatusMovedPermanently, "http://www.google.com/")
3. })
```

路由重定向，使用 `HandleContext` ：

```
1. r.GET("/test", func(c *gin.Context) {
2.     c.Request.URL.Path = "/test2"
3.     r.HandleContext(c)
4. })
5. r.GET("/test2", func(c *gin.Context) {
6.     c.JSON(200, gin.H{"hello": "world"})
7. })
```

# 静态文件服务

```go
func main() {
    router := gin.Default()
    router.Static("/assets", "./assets")
    router.StaticFS("/more_static", http.Dir("my_file_system"))
    router.StaticFile("/favicon.ico", "./resources/favicon.ico")

    // 监听并在 0.0.0.0:8080 上启动服务
    router.Run(":8080")
}
```

# 静态资源嵌入

你可以使用 `go-assets` 将静态资源打包到可执行文件中。

```
1.  func main() {
2.      r := gin.New()
3.
4.      t, err := loadTemplate()
5.      if err != nil {
6.          panic(err)
7.      }
8.      r.SetHTMLTemplate(t)
9.
10.     r.GET("/", func(c *gin.Context) {
11.         c.HTML(http.StatusOK, "/html/index.tmpl",nil)
12.     })
13.     r.Run(":8080")
14. }
15.
16. // loadTemplate 加载由 go-assets-builder 嵌入的模板
17. func loadTemplate() (*template.Template, error) {
18.     t := template.New("")
19.     for name, file := range Assets.Files {
20.         if file.IsDir() || !strings.HasSuffix(name, ".tmpl") {
21.             continue
22.         }
23.         h, err := ioutil.ReadAll(file)
24.         if err != nil {
25.             return nil, err
26.         }
27.         t, err = t.New(name).Parse(string(h))
28.         if err != nil {
29.             return nil, err
30.         }
31.     }
32.     return t, nil
33. }
```

请参阅 `examples/assets-in-binary` 目录中的完整示例。

# 测试

怎样编写 Gin 的测试用例

HTTP 测试首选 `net/http/httptest` 包。

```
1.  package main
2.
3.  func setupRouter() *gin.Engine {
4.      r := gin.Default()
5.      r.GET("/ping", func(c *gin.Context) {
6.          c.String(200, "pong")
7.      })
8.      return r
9.  }
10.
11. func main() {
12.     r := setupRouter()
13.     r.Run(":8080")
14. }
```

上面这段代码的测试用例：

```
1.  package main
2.
3.  import (
4.      "net/http"
5.      "net/http/httptest"
6.      "testing"
7.
8.      "github.com/stretchr/testify/assert"
9.  )
10.
11. func TestPingRoute(t *testing.T) {
12.     router := setupRouter()
13.
14.     w := httptest.NewRecorder()
15.     req, _ := http.NewRequest("GET", "/ping", nil)
16.     router.ServeHTTP(w, req)
17.
18.     assert.Equal(t, 200, w.Code)
```

```
19.     assert.Equal(t, "pong", w.Body.String())
20. }
```

# 用户

使用 Gin web 框架的知名项目：

- gorush：Go 编写的通知推送服务器。
- fnproject：原生容器，云 serverless 平台。
- photoprism：由 Go 和 Google TensorFlow 提供支持的个人照片管理工具。
- krakend：拥有中间件的超高性能 API 网关。
- picfit：Go 编写的图像尺寸调整服务器。
- gotify：使用实时 web socket 做消息收发的简单服务器。
- cds：企业级持续交付和 DevOps 自动化开源平台。

# FAQ

TODO: 记录 GitHub Issue 中的一些常见问题。