Gin框架 中文文档

翻译者：asong 微信公众号：Golang梦工厂

简介：一名后端开发工程师，热爱技术分享，Golang语言爱好者专注于Golang相关技术：Golang面试、Beego、Gin、Mysql、Linux、网络、操作系统等，致力于Golang开发。

无水印版获取文档方式：直接关注公众号后台回复：Gin，即可获取最新Gin中文文档。作者asong定期维护。

同时文档上传个人github：https://github.com/sunsong2020/Golang_Dream/Gin/Doc，自行下载，能给个Star就更好了！！！

作者会定期维护该文档。 更新时间：2020.06

Gin Web 框架

Gin是用Go（Golang）编写的Web框架。他是一个类似于martini但拥有更好性能的API框架，由于httprouter，速度提高了40倍。如果您追求性能和高效的效率，您将会爱上Gin。

安装

在安装Gin包之前，你需要在你的电脑上安装Go环境并设置你的工作区。

1. 首先需要安装Go(支持版本1.11+)，然后使用以下Go命令安装Gin:

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

2. 在你的代码中导入Gin包:

```
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：

```
import "net/http"
```

快速开始

```
# 假设example.go 文件中包含以下代码
$ cat example.go
```

```
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() // listen and serve on 0.0.0.0:8080 (for windows "localhost:8080")
}
```

```
# 运行example.go文件并在浏览器上访问0.0.0.0:8080/ping（windows访问：
localhost:8080/ping）
$ go run example.go
```

性能测试

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	**43550**	**27364 ns/op**	**0 B/op**	**0 allocs/op**
BenchmarkAce_GithubAll	40543	29670 ns/op	0 B/op	0 allocs/op
BenchmarkAero_GithubAll	57632	20648 ns/op	0 B/op	0 allocs/op
BenchmarkBear_GithubAll	9234	216179 ns/op	86448 B/op	943 allocs/op
BenchmarkBeego_GithubAll	7407	243496 ns/op	71456 B/op	609 allocs/op
BenchmarkBone_GithubAll	420	2922835 ns/op	720160 B/op	8620 allocs/op
BenchmarkChi_GithubAll	7620	238331 ns/op	87696 B/op	609 allocs/op
BenchmarkDenco_GithubAll	18355	64494 ns/op	20224 B/op	167 allocs/op
BenchmarkEcho_GithubAll	31251	38479 ns/op	0 B/op	0 allocs/op
BenchmarkGocraftWeb_GithubAll	4117	300062 ns/op	131656 B/op	1686 allocs/op
BenchmarkGoji_GithubAll	3274	416158 ns/op	56112 B/op	334 allocs/op
BenchmarkGojiv2_GithubAll	1402	870518 ns/op	352720 B/op	4321 allocs/op
BenchmarkGoJsonRest_GithubAll	2976	401507 ns/op	134371 B/op	2737 allocs/op
BenchmarkGoRestful_GithubAll	410	2913158 ns/op	910144 B/op	2938 allocs/op
BenchmarkGorillaMux_GithubAll	346	3384987 ns/op	251650 B/op	1994 allocs/op
BenchmarkGowwwRouter_GithubAll	10000	143025 ns/op	72144 B/op	501 allocs/op
BenchmarkHttpRouter_GithubAll	55938	21360 ns/op	0 B/op	0 allocs/op
BenchmarkHttpTreeMux_GithubAll	10000	153944 ns/op	65856 B/op	671 allocs/op
BenchmarkKocha_GithubAll	10000	106315 ns/op	23304 B/op	843 allocs/op

Benchmark name	(1)	(2)	(3)	(4)
BenchmarkLARS_GithubAll	47779	25084 ns/op	0 B/op	0 allocs/op
BenchmarkMacaron_GithubAll	3266	371907 ns/op	149409 B/op	1624 allocs/op
BenchmarkMartini_GithubAll	331	3444706 ns/op	226551 B/op	2325 allocs/op
BenchmarkPat_GithubAll	273	4381818 ns/op	1483152 B/op	26963 allocs/op
BenchmarkPossum_GithubAll	10000	164367 ns/op	84448 B/op	609 allocs/op
BenchmarkR2router_GithubAll	10000	160220 ns/op	77328 B/op	979 allocs/op
BenchmarkRivet_GithubAll	14625	82453 ns/op	16272 B/op	167 allocs/op
BenchmarkTango_GithubAll	6255	279611 ns/op	63826 B/op	1618 allocs/op
BenchmarkTigerTonic_GithubAll	2008	687874 ns/op	193856 B/op	4474 allocs/op
BenchmarkTraffic_GithubAll	355	3478508 ns/op	820744 B/op	14114 allocs/op
BenchmarkVulcan_GithubAll	6885	193333 ns/op	19894 B/op	609 allocs/op

- (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(alloc/op)，越低越好

G1 v1. 稳定性

- 零分配路由器
- 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，新版本的发布破坏你的代码

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。

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

API 例子

您可以在Gin示例的仓库中找到许多现成的示例。

使用 GET, POST, PUT, PATCH, DELETE and OPTIONS

```
func main() {
    //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
}
```

路由参数

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

    // 这个handler 将会匹配 /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)
    })

    // 但是，这个将匹配 /user/john/ 以及 /user/john/send
    // 如果没有其他路由器匹配 /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)
    })

    // 对于每个匹配的请求，上下文将保留路由定义
    router.POST("/user/:name/*action", func(c *gin.Context) {
        c.FullPath() == "/user/:name/*action" // true
    })

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

查询字符串参数

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

    // 查询字符串参数使用现有的底层 request 对象解析
    // 请求响应匹配的 URL:  /welcome?firstname=Jane&lastname=Doe
    router.GET("/welcome", func(c *gin.Context) {
        firstname := c.DefaultQuery("firstname", "Guest")
        lastname := c.Query("lastname") // 这个是
c.Request.URL.Query().Get("lastname") 快捷写法

        c.String(http.StatusOK, "Hello %s %s", firstname, lastname)
    })
    router.Run(":8080")
}
```

Multipart/Urlencoded 表单

```
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")
}
```

其他示例：query+post 表单

```
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")
}
```

运行结果：

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

Map 作为查询字符串或 post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")
}
```

运行结果：

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

上传文件

单个文件

参考 issue #774 与详细的示例代码： example code.

慎用 file.Filename， 参考 Content-Disposition on MDN 和 #1693

上传文件的文件名可以由用户自定义，所以可能包含非法字符串，为了安全起见，应该由服务端统一文件名规则。

```
func main() {
    router := gin.Default()
    // 给表单限制上传大小 (默认是 32 MiB)
    router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // single file
        file, _ := c.FormFile("file")
        log.Println(file.Filename)

        // 上传文件到指定的路径
        c.SaveUploadedFile(file, dst)
```

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

curl 测试：

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

多个文件

参考详细示例：[example code.]

```
func main() {
    router := gin.Default()
    // 给表单限制上传大小 (default is 32 MiB)
    router.MaxMultipartMemory = 8 << 20  // 8 MiB
    router.POST("/upload", func(c *gin.Context) {
        // 多文件
        form, _ := c.MultipartForm()
        files := form.File["upload[]"]

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

            //上传文件到指定的路径
            c.SaveUploadedFile(file, dst)
        }
        c.String(http.StatusOK, fmt.Sprintf("%d files uploaded!", len(files)))
    })
    router.Run(":8080")
}
```

curl 测试：

```
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"
```

路由分组

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

    // Simple group: v1
    v1 := router.Group("/v1")
    {
        v1.POST("/login", loginEndpoint)
        v1.POST("/submit", submitEndpoint)
        v1.POST("/read", readEndpoint)
    }
```

```
    // Simple group: v2
    v2 := router.Group("/v2")
    {
        v2.POST("/login", loginEndpoint)
        v2.POST("/submit", submitEndpoint)
        v2.POST("/read", readEndpoint)
    }

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

默认的没有中间件的空白 Gin

使用:

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

代替

```
// 默认已经连接了 Logger and Recovery 中间件
r := gin.Default()
```

使用中间件

```
func main() {
    // 创建一个默认的没有任何中间件的路由
    r := gin.New()

    // 全局中间件
    // Logger 中间件将写日志到 gin.DefaultWriter 即使你设置 GIN_MODE=release.
    // 默认设置 gin.DefaultWriter = os.Stdout
    r.Use(gin.Logger())

    // Recovery 中间件从任何 panic 恢复，如果出现 panic，它会写一个 500 错误。
    r.Use(gin.Recovery())

    // 对于每个路由中间件，您可以根据需要添加任意数量
    r.GET("/benchmark", MyBenchLogger(), benchEndpoint)

    // 授权组
    // authorized := r.Group("/", AuthRequired())
    // 也可以这样
    authorized := r.Group("/")
    // 每个组的中间件！ 在这个实例中，我们只需要在 "authorized" 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)

        // 嵌套组
        testing := authorized.Group("testing")
        testing.GET("/analytics", analyticsEndpoint)
    }
```

```
    // 监听并服务于 0.0.0.0:8080
    r.Run(":8080")
}
```

如何写入日志文件

```
func main() {
    // 禁用控制台颜色，当你将日志写入到文件的时候，你不需要控制台颜色
    gin.DisableConsoleColor()

    // 写入日志文件
    f, _ := os.Create("gin.log")
    gin.DefaultWriter = io.MultiWriter(f)

    // 如果你需要同时写入日志文件和控制台上显示，使用下面代码
    // gin.DefaultWriter = io.MultiWriter(f, os.Stdout)

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

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

自定义日志格式

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

    // LoggerWithFormatter 中间件会将日志写入 gin.DefaultWriter
    // 默认 gin.DefaultWriter = os.Stdout
    router.Use(gin.LoggerWithFormatter(func(param gin.LogFormatterParams) string
{

        // 你的自定义格式
        return fmt.Sprintf("%s - [%s] \"%s %s %s %d %s \"%s\" %s\"\n",
                param.ClientIP,
                param.TimeStamp.Format(time.RFC1123),
                param.Method,
                param.Path,
                param.Request.Proto,
                param.StatusCode,
                param.Latency,
                param.Request.UserAgent(),
                param.ErrorMessage,
        )
    }))
    router.Use(gin.Recovery())

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

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

```
}
```

样本输出：

```
::1 - [Fri, 07 Dec 2018 17:04:38 JST] "GET /ping HTTP/1.1 200 122.767µs
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/71.0.3578.80 Safari/537.36" "
```

控制日志输出颜色（Controlling Log output coloring）

默认，控制台上输出的日志应根据检测到的TTY进行着色。

没有为日志着色：

```
func main() {
    // 禁用日志的颜色
    gin.DisableConsoleColor()

    // 使用默认中间件创建一个 gin路由:
    // logger 与 recovery (crash-free) 中间件
    router := gin.Default()

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

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

为日志着色：

```
func main() {
    //记录日志的颜色
    gin.ForceConsoleColor()

    // 使用默认中间件创建一个 gin路由:
    // logger 与 recovery (crash-free) 中间件
    router := gin.Default()

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

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

模型绑定和验证

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

Gin使用go-playground/validator.v8验证参数，点击此处查看完整文档here

需要在绑定的字段上设置tag，比如，绑定格式为json，需要设置为 `json:"fieldname"`

此外，Gin提供了两种绑定方法：

- 类型 - Must bind
  - 方法 - Bind, BindJSON, BindXML, BindQuery, BindYAML, BindHeader
  - 行为 - 这些方法底层使用`MustBindWith`，如果存在绑定错误，请求将被以下指令中止`c.AbortWithError(400, err).SetType(ErrorTypeBind)`，响应状态代码会被设置为400，请求头`Content-Type`被设置为`text/plain; charset=utf-8`。注意，如果你试图在此之后设置响应代码，将会发出一个警告`[GIN-debug] [WARNING] Headers were already written. Wanted to override status code 400 with 422`，如果你希望更好地控制行为，请使用ShouldBind相关的方法。
- 类型 - Should bind
  - 方法 - ShouldBind, ShouldBindJSON, ShouldBindXML, ShouldBindQuery, ShouldBindYAML, ShouldBindHeader。
  - 行为 - 这些方法底层使用`ShouldBindWith`，如果存在绑定错误，则返回错误，开发人员可以正确处理请求和错误。当我们使用绑定方法时，Gin会根据`Content-Type`推断出使用哪种绑定器，如果你确定你绑定的是什么，你可以使用`MustBindWith`或者`BindingWith`。

你还可以给字段指定特定规则的修饰符，如果一个字段用`binding:"required"`修饰，并且在绑定时该字段的值为空，那么将返回一个错误。

```
// 绑定为 JSON
type Login struct {
    User     string `form:"user" json:"user" xml:"user"  binding:"required"`
    Password string `form:"password" json:"password" xml:"password"
binding:"required"`
}

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

    // JSON 绑定示例 ({"user": "manu", "password": "123"})
    router.POST("/loginJSON", func(c *gin.Context) {
        var json Login
        if err := c.ShouldBindJSON(&json); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

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

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

    // XML 绑定示例 (
    //  <?xml version="1.0" encoding="UTF-8"?>
    //  <root>
    //      <user>user</user>
    //      <password>123</password>
    //  </root>)
    router.POST("/loginXML", func(c *gin.Context) {
        var xml Login
        if err := c.ShouldBindXML(&xml); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
```

```
        }

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

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

    // 绑定HTML表单的示例 (user=manu&password=123)
    router.POST("/loginForm", func(c *gin.Context) {
        var form Login
        //这个将通过 content-type 头去推断绑定器使用哪个依赖。
        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"})
    })

    // 监听并服务于 0.0.0.0:8080
    router.Run(":8080")
}
```

请求示例:

```
$ 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"}
```

跳过验证:

当使用上面的`curl`命令运行上面的示例时，返回错误，因为示例中`Password`字段使用了`binding:"required"`，如果我们使用`binding:"-"`，那么它就不会报错。

自定义验证器

也可以注册自定义验证器。请参阅示例代码：[example code]

```
package main

import (
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/binding"
    "gopkg.in/go-playground/validator.v10"
)

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

var bookableDate validator.Func = func(fl validator.FieldLevel) bool {
    date, ok := fl.Field().Interface().(time.Time)
    if ok {
        today := time.Now()
        if today.After(date) {
            return false
        }
    }
    return true
}

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

    if v, ok := binding.Validator.Engine().(*validator.Validate); ok {
        v.RegisterValidation("bookabledate", bookableDate)
    }

    route.GET("/bookable", getBookable)
    route.Run(":8085")
}

func getBookable(c *gin.Context) {
    var b Booking
    if err := c.ShouldBindWith(&b, binding.Query); err == nil {
        c.JSON(http.StatusOK, gin.H{"message": "Booking dates are valid!"})
    } else {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    }
}
```

```
$ curl "localhost:8085/bookable?check_in=2018-04-16&check_out=2018-04-17"
{"message":"Booking dates are valid!"}

$ curl "localhost:8085/bookable?check_in=2018-03-10&check_out=2018-03-09"
{"error":"Key: 'Booking.CheckOut' Error:Field validation for 'CheckOut' failed
on the 'gtfield' tag"}
```

Struct level validations 也可以以这种方式被注册。请参阅struct-lvl-validation示例以了解更多信息。

只绑定查询字符串（Only Bind Query String）

`ShouldBindQuery`函数只绑定Get参数，不绑定post数据，查看详细信息

```go
package main

import (
    "log"

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

type Person struct {
    Name    string `form:"name"`
    Address string `form:"address"`
}

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

func startPage(c *gin.Context) {
    var person Person
    if c.ShouldBindQuery(&person) == nil {
        log.Println("====== Only Bind By Query String ======")
        log.Println(person.Name)
        log.Println(person.Address)
    }
    c.String(200, "Success")
}
```

绑定Get参数或者Post参数

查看详细信息

```go
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"`
        CreateTime time.Time `form:"createTime" time_format:"unixNano"`
        UnixTime   time.Time `form:"unixTime" time_format:"unix"`
}

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

func startPage(c *gin.Context) {
    var person Person
    // If `GET`, only `Form` binding engine (`query`) used.
    // If `POST`, first checks the `content-type` for `JSON` or `XML`, then uses
`Form` (`form-data`).
    // See more at 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)
                log.Println(person.CreateTime)
                log.Println(person.UnixTime)
        }

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

测试示例:

```
$ curl -X GET "localhost:8085/testing?name=appleboy&address=xyz&birthday=1992-
03-15&createTime=1562400033000000123&unixTime=1562400033"
```

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")
}
```

测试示例：

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

绑定Header

```
package main

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

type testHeader struct {
    Rate   int    `header:"Rate"`
    Domain string `header:"Domain"`
}

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        h := testHeader{}

        if err := c.ShouldBindHeader(&h); err != nil {
            c.JSON(200, err)
        }

        fmt.Printf("%#v\n", h)
        c.JSON(200, gin.H{"Rate": h.Rate, "Domain": h.Domain})
    })

    r.Run()

// client
// curl -H "rate:300" -H "domain:music" 127.0.0.1:8080/
// output
// {"Domain":"music","Rate":300}
}
```

绑定HTML复选框

查看详细信息

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>
```

result:

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

Multipart/Urlencoded绑定

```
type ProfileForm struct {
    Name   string                `form:"name" binding:"required"`
    Avatar *multipart.FileHeader `form:"avatar" binding:"required"`

    // 或多个文件
    // Avatars []*multipart.FileHeader `form:"avatar" binding:"required"`
}

func main() {
    router := gin.Default()
    router.POST("/profile", func(c *gin.Context) {
        // 你可以使用显示绑定声明来绑定多部分表单:
        // c.ShouldBindWith(&form, binding.Form)
        // 或者你可以简单地将自动绑定与ShouldBind方法一起使用:
        var form ProfileForm
        // 在这种情况下，将自动选择适当的绑定
        if err := c.ShouldBind(&form); err != nil {
            c.String(http.StatusBadRequest, "bad request")
            return
        }
```

```
        err := c.SaveUploadedFile(form.Avatar, form.Avatar.Filename)
        if err != nil {
            c.String(http.StatusInternalServerError, "unknown error")
            return
        }

        // db.Save(&form)

        c.String(http.StatusOK, "ok")
    })
    router.Run(":8080")
}
```

测试示例：

```
$ curl -X POST -v --form name=user --form "avatar=@./avatar.png"
http://localhost:8080/profile
```

XML、JSON、YAML和ProtoBuf 渲染

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

    // gin.H 是 map[string]interface{} 的快捷写法
    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 struct
        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中会变成 "user"
        // 将会输出  :   {"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 的特定定义写在testdata/protoexample文件中
```

```
        data := &protoexample.Test{
            Label: &label,
            Reps:  reps,
        }
        // 注意数据在响应中变为二进制数据
        // 将会输出protoexample.test protobuf序列化的数据
        c.ProtoBuf(http.StatusOK, data)
    })

    // 监听并服务于 0.0.0.0:8080
    r.Run(":8080")
}
```

SecureJSON

使用SecureJSON可以防止json劫持，如果返回的数据是数组，则会默认在返回值前加上"while(1)"。

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

    // 你也可以使用自己的secure json前缀
    // r.SecureJsonPrefix(")]}',\n")

    r.GET("/someJSON", func(c *gin.Context) {
        names := []string{"lena", "austin", "foo"}

        // 将会输出 :   while(1);["lena","austin","foo"]
        c.SecureJSON(http.StatusOK, names)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

JSONP

在不同的域中使用 JSONP 从一个服务器请求数据。如果请求参数中存在 callback，添加 callback 到 response body 。

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

    r.GET("/JSONP", func(c *gin.Context) {
        data := gin.H{
            "foo": "bar",
        }

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

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")

        // client
```

```
        // curl http://127.0.0.1:8080/JSONP?callback=x
}
```

AsciiJSON

使用 AsciiJSON 生成仅有 ASCII 字符的 JSON，非 ASCII 字符将会被转义 。

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

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

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

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

PureJSON

通常情况下，JSON会将特殊的HTML字符替换为对应的unicode字符，比如<替换为\u003c，如果想原样输出html，则使用PureJSON，这个特性在Go 1.6及以下版本中无法使用。

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

    // Serves unicode entities
    r.GET("/json", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "html": "<b>Hello, world!</b>",
        })
    })

    // Serves literal characters
    r.GET("/purejson", func(c *gin.Context) {
        c.PureJSON(200, gin.H{
            "html": "<b>Hello, world!</b>",
        })
    })

    // listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

提供静态文件

```
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")

    // Listen and serve on 0.0.0.0:8080
    router.Run(":8080")
}
```

从文件提供数据

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

    router.GET("/local/file", func(c *gin.Context) {
        c.File("local/file.go")
    })

    var fs http.FileSystem = // ...
    router.GET("/fs/file", func(c *gin.Context) {
        c.FileFromFS("fs/file.go", fs)
    })
}
```

从reader提供数据

```
func main() {
    router := gin.Default()
    router.GET("/someDataFromReader", func(c *gin.Context) {
        response, err := http.Get("https://raw.githubusercontent.com/gin-
gonic/logo/master/color.png")
        if err != nil || response.StatusCode != http.StatusOK {
            c.Status(http.StatusServiceUnavailable)
            return
        }

        reader := response.Body
        contentLength := response.ContentLength
        contentType := response.Header.Get("Content-Type")

        extraHeaders := map[string]string{
            "Content-Disposition": `attachment; filename="gopher.png"`,
        }

        c.DataFromReader(http.StatusOK, contentLength, contentType, reader,
extraHeaders)
    })
    router.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>
```

在不同目录中使用具有相同名称的模板

```
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

你可以使用自己的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符

```
    r := gin.Default()
    r.Delims("{[{", "}]}")
    r.LoadHTMLGlob("/path/to/templates")
```

自定义模板函数

查看详细示例：[example code](#)。

main.go

```
import (
    "fmt"
    "html/template"
    "net/http"
    "time"

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

func formatAsDate(t time.Time) string {
    year, month, day := t.Date()
    return fmt.Sprintf("%d%02d/%02d", year, month, day)
}

func main() {
    router := gin.Default()
    router.Delims("{[{", "}]}")
    router.SetFuncMap(template.FuncMap{
        "formatAsDate": formatAsDate,
    })
    router.LoadHTMLFiles("./testdata/template/raw.tmpl")
```

```
    router.GET("/raw", func(c *gin.Context) {
        c.HTML(http.StatusOK, "raw.tmpl", gin.H{
            "now": time.Date(2017, 07, 01, 0, 0, 0, 0, time.UTC),
        })
    })

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

raw.tmpl

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

Result:

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

多模板

Gin允许默认只使用一个 `html.Template` 。查看 多模板渲染 的使用详情，类似 go 1.6 `block template`。

重定向

发布HTTP重定向很容易，支持内部和外部链接

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

从POST发出HTTP重定向。请参阅问题：#444

```
r.POST("/test", func(c *gin.Context) {
    c.Redirect(http.StatusFound, "/foo")
})
```

发出路由器重定向，使用HandleContext如下：

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

自定义中间件

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

```
        // 设置简单的变量
        c.Set("example", "12345")

        // 在请求之前

        c.Next()

        // 在请求之后
        latency := time.Since(t)
        log.Print(latency)

        // 记录发送状态
        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)

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

    // 监听并服务于 0.0.0.0:8080
    r.Run(":8080")
}
```

使用Using BasicAuth() 中间件

```
// 模拟一些私有的数据
var secrets = gin.H{
    "foo":    gin.H{"email": "foo@bar.com", "phone": "123433"},
    "austin": gin.H{"email": "austin@example.com", "phone": "666"},
    "lena":   gin.H{"email": "lena@guapa.com", "phone": "523443"},
}

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

    // 在组中使用 gin.BasicAuth() 中间件
    // gin.Accounts 是 map[string]string 的快捷写法
    authorized := r.Group("/admin", gin.BasicAuth(gin.Accounts{
        "foo":    "bar",
        "austin": "1234",
        "lena":   "hello2",
        "manu":   "4321",
    }))

    // /admin/secrets 结尾
    // 点击 "localhost:8080/admin/secrets
    authorized.GET("/secrets", func(c *gin.Context) {
        // 获取 user, 它是由 BasicAuth 中间件设置的
```

```
        user := c.MustGet(gin.AuthUserKey).(string)
        if secret, ok := secrets[user]; ok {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": secret})
        } else {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": "NO SECRET :("})
        }
    })

    // 监听并服务于 0.0.0.0:8080
    r.Run(":8080")
}
```

中间件中使用Goroutines

在中间件或处理程序中启动新的Goroutines时，你不应该使用其中的原始上下文，你必须使用只读副本（c.Copy()）

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

    r.GET("/long_async", func(c *gin.Context) {
        // 创建在goroutine中使用的副本
        cCp := c.Copy()
        go func() {
            // 使用time.sleep()休眠5秒，模拟一个用时长的任务
            time.Sleep(5 * time.Second)

            // 注意，你使用的是复制的 context "cCp" ，重要
            log.Println("Done! in path " + cCp.Request.URL.Path)
        }()
    })

    r.GET("/long_sync", func(c *gin.Context) {
        // 使用 time.Sleep() 休眠 5 秒，模拟一个用时长的任务。
        time.Sleep(5 * time.Second)

        // 因为我们没有使用协程，我们不需要复制 context
        log.Println("Done! in path " + c.Request.URL.Path)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

自定义HTTP配置

直接使用 http.ListenAndServe()，像这样：

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

或者

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

    s := &http.Server{
        Addr:           ":8080",
        Handler:        router,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    s.ListenAndServe()
}
```

支持 Let's Encrypt

一个 LetsEncrypt HTTPS 服务器的示例。

```
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 管理器示例。

```
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 处理器
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })
```

```
    m := autocert.Manager{
        Prompt:     autocert.AcceptTOS,
        HostPolicy: autocert.HostWhitelist("example1.com", "example2.com"),
        Cache:      autocert.DirCache("/var/www/.cache"),
    }

    log.Fatal(autotls.RunWithManager(r, &m))
}
```

使用Gin运行多种服务

查看 问题 并尝试下面示例:

```
package main

import (
    "log"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "golang.org/x/sync/errgroup"
)

var (
    g errgroup.Group
)

func router01() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 01",
            },
        )
    })

    return e
}

func router02() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            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 {
        err := server01.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal(err)
        }
        return err
    })

    g.Go(func() error {
        err := server02.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal(err)
        }
        return err
    })

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

正常的重启或停止

你可以使用几种方法来正常的重启或停止。您可以使用专门为此目的构建的第三方程序包，也可以使用内置程序包中的功能和方法手动执行相同的操作。

第三方程序包

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

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

备选方案：

- manners：一个有礼貌的 Go HTTP 服务器，它可以正常的关闭。

- graceful ： Graceful 是一个 Go 包，它可以正常的关闭一个 http.Handler 服务器。
- grace ： 正常的重启 & Go 服务器零停机部署。

手动配置：

如果你使用的是Go 1.8或更高的版本，则可能不需要使用这些库。考虑使用http.Server的内置Shutdown()方法进行正常关闭。下面的示例描述了它的用法，我们在这里有更多使用gin的示例。

```
// +build go1.8

package main

import (
    "context"
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "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")
    })

    srv := &http.Server{
        Addr:    ":8080",
        Handler: router,
    }

    // 在goroutine中初始化服务器，以使其不会阻止下面的正常关闭处理
    go func() {
        if err := srv.ListenAndServe(); err != nil && err !=
http.ErrServerClosed {
            log.Fatalf("listen: %s\n", err)
        }
    }()

    //等待中断信号超时5秒 正常关闭服务器
    quit := make(chan os.Signal)
    // kill (没有参数) 默认发送 syscall.SIGTERM
    // kill -2 is syscall.SIGINT
    // kill -9 is syscall.SIGKILL 但是不能被捕获，所以不需要添加他
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
    <-quit
    log.Println("Shutting down server...")

    // 上下文用于通知服务器它有5秒的时间完成
    // 当前正在处理的请求
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
```

```
        log.Fatal("Server forced to shutdown:", err)
    }

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

使用模板构建单个二进制文件

你可以使用 go-assets 将服务器构建到一个包含模板的单独的二进制文件中。

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

    t, err := loadTemplate()
    if err != nil {
        panic(err)
    }
    r.SetHTMLTemplate(t)

    r.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "/html/index.tmpl",nil)
    })
    r.Run(":8080")
}

// loadTemplate加载go-assets-builder嵌入的模板
func loadTemplate() (*template.Template, error) {
    t := template.New("")
    for name, file := range Assets.Files {
        defer file.Close()
        if file.IsDir() || !strings.HasSuffix(name, ".tmpl") {
            continue
        }
        h, err := ioutil.ReadAll(file)
        if err != nil {
            return nil, err
        }
        t, err = t.New(name).Parse(string(h))
        if err != nil {
            return nil, err
        }
    }
    return t, nil
}
```

在该目录下查看一个完整示例：https://github.com/gin-gonic/examples/tree/master/assets-in-binary

使用自定义结构绑定表单数据

下面示例使用自定义结构：

```
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()
}
```

命令行中使用 curl　命令的结果：

```
$ 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"}}
```

尝试将 body 绑定到不同的结构中

绑定 request body 的常规方法是使用 `c.Request.Body` 并且不能多次调用它们。

```
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{}
  // This c.ShouldBind consumes c.Request.Body and it cannot be reused.
  if errA := c.ShouldBind(&objA); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
  // Always an error is occurred by this because c.Request.Body is EOF now.
  } else if errB := c.ShouldBind(&objB); errB == nil {
    c.String(http.StatusOK, `the body should be formB`)
  } else {
    ...
  }
}
```

对于这一点，你可以使用 `c.ShouldBindBodyWith`。

```
func SomeHandler(c *gin.Context) {
  objA := formA{}
  objB := formB{}
  // 这里读取 c.Request.Body 并将结果存储到 context 中。
  if errA := c.ShouldBindBodyWith(&objA, binding.JSON); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
  // At this time, it reuses body stored in the context.
  } else if errB := c.ShouldBindBodyWith(&objB, binding.JSON); errB == nil {
    c.String(http.StatusOK, `the body should be formB JSON`)
  // And it can accepts other formats
  } else if errB2 := c.ShouldBindBodyWith(&objB, binding.XML); errB2 == nil {
    c.String(http.StatusOK, `the body should be formB XML`)
  } else {
    ...
  }
}
```

- `c.ShouldBindBodyWith` 在绑定前存储 body 到 context 中。这对性能会有轻微的影响，所以如果你可以通过立即调用绑定，不应该使用这个方法。

- 只有一些格式需要这个功能 -- JSON 、 XML 、 MsgPack、 ProtoBuf 。 对于其他格式，Query、 Form、 FormPost、 FormMultipart， 能被 c.ShouldBind() 多次调用，而不会对性能造成任何损害 （参见 #1341）。

http2 服务器推送

http.Pusher 仅仅被 go1.8+ 支持。 在 golang 官方博客 中查看详细信息。

```
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 {
            // use pusher.Push() to do server push
            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",
        })
    })

    // Listen and Server in https://127.0.0.1:8080
    r.RunTLS(":8080", "./testdata/server.pem", "./testdata/server.key")
}
```

自定义路由日志的格式

默认的路由日志是：

```
[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)
```

如果你想以给定的格式记录这些信息（例如 JSON，键值对或其他格式），你可以使用 gin.DebugPrintRouteFunc来定义格式，在下面的示例中，我们使用标准日志包记录路由日志，你可以使用其他适合你需求的日志工具。

```
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")
    })

    // Listen and Server in http://0.0.0.0:8080
    r.Run()
}
```

设置并获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()
}
```

测试

net/http/httptest包是http测试的首选方式。

```
package main

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

func main() {
    r := setupRouter()
    r.Run(":8080")
}
Test for code example above:

package main

import (
    "net/http"
    "net/http/httptest"
    "testing"

    "github.com/stretchr/testify/assert"
)

func TestPingRoute(t *testing.T) {
    router := setupRouter()

    w := httptest.NewRecorder()
    req, _ := http.NewRequest("GET", "/ping", nil)
    router.ServeHTTP(w, req)

    assert.Equal(t, 200, w.Code)
    assert.Equal(t, "pong", w.Body.String())
}
```

优秀开源项目

使用Gin Web框架的出色项目列表。

- gorush: 用Go编写的推送通知服务器。
- fnproject：容器本地的，与云无关的无服务器平台。
- photoprism: 由Go和Google TensorFlow编写的个人照片管理。
- krakend: 具有中间件的超高性能API网关。
- picfit：用Go编写的图像大小调整服务器。
- brigade: Kubernetes的基于事件的脚本。
- dkron: 分布式容错调度系统。

公众号：Golang梦工厂

Asong是一名Golang开发工程师，专注于Golang相关技术：Golang面试、Beego、Gin、Mysql、Linux、网络、操作系统等，致力于Golang开发。欢迎关注公众号：Golang梦工厂。一起学习，一起进步。

获取文档方式：直接公众号后台回复：Gin，即可获取最新Gin中文文档。作者asong定期维护。

同时文档上传个人github：https://github.com/sunsong2020/Golang_Dream/Gin/Doc，自行下载，能给个Star就更好了！！！