gin实战

官方网站
https://gin-gonic.com/
工程代码
https://github.com/gin-gonic/gin.git
测试范例
https://github.com/gin-gonic/examples.git
中间件
https://github.com/gin-gonic/contrib.git

1 入门

第一个gin程序

```
1  // 1-basic
2  package main
3
4  import "github.com/gin-gonic/gin"
5
6  func main() {
7      // Default方法的主要作用是实例化一个带有日志、故障恢复中间件的引擎。
8      r := gin.Default() //实例化一个gin对象
9      // 定义请求
10     //定义一个GET请求的路由，参数一是路由地址，也就是在浏览器访问的相对路径，
11     //                        参数二是一个匿名函数，函数内部用于业务逻辑处
```

```
理。
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{ //JSON内容可以通过gin提供的H方法来构建，非常方便。
            "msg": "Hello world!", //调用JSON方法返回数据。JSON的操作非常简单，参数一是状态码，参数二是JSON的内容。
        })
    })
    // Run方法最终会调用内置http库的ListenAndServe方法来监听端口，如果不传参数默认监听80端口，
    // 也可以通过参数来变更地址和端口。
    r.Run(":8081")
}
```

2 RESTful API

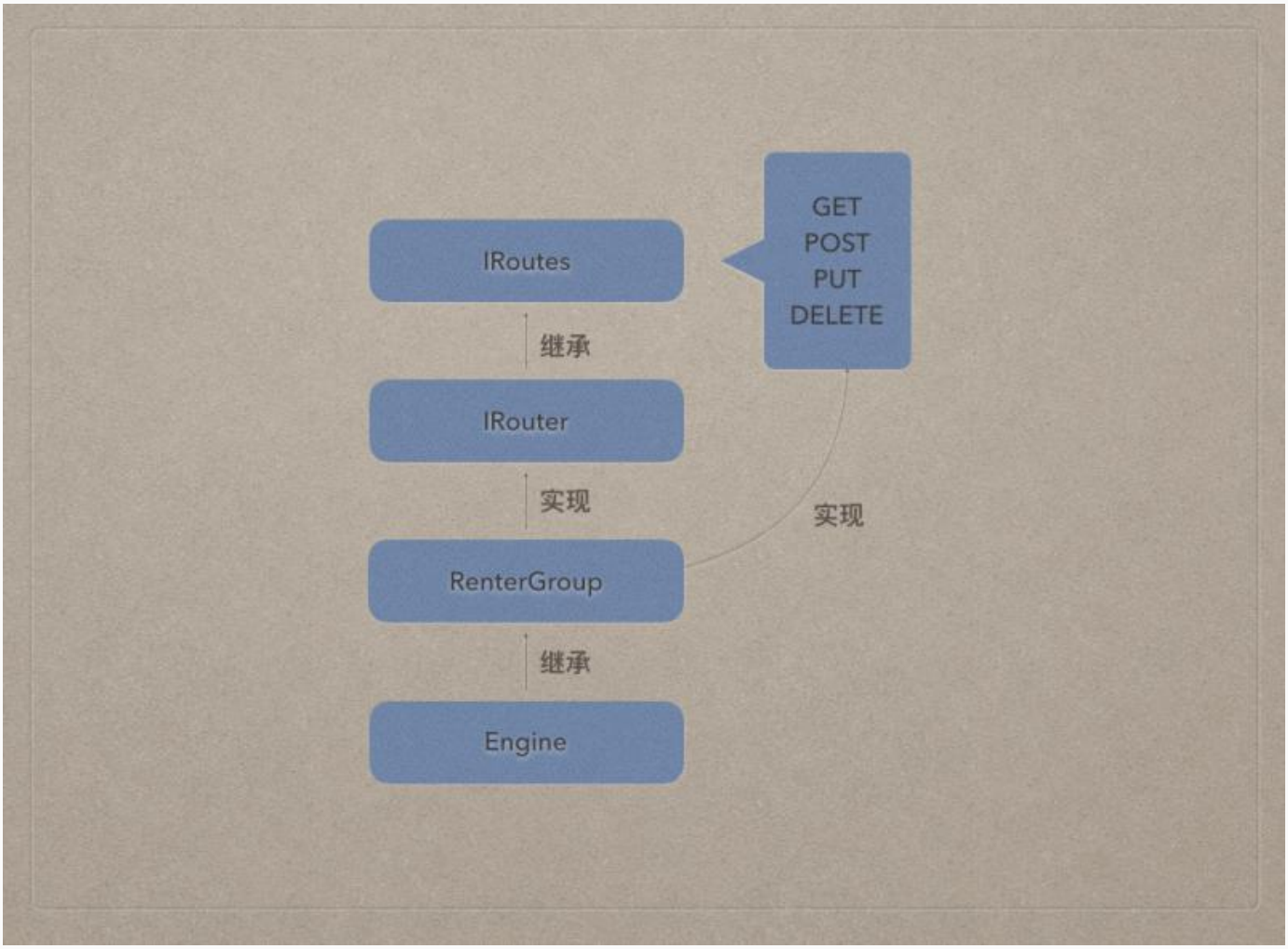
结构体

`RESTful`是网络应用程序的一种设计风格和开发方式，每一个URI代表一种资源，客户端通过`POST`、`DELETE`、`PUT`、`GET`四种请求方式来对资源做增删改查的操作。
同样的，Gin框架给我们提供的除这4种动词外，还有`PATCH`、`OPTION`、`HEAD`等，详细内容可以查看`rentergroup.go`文件的`IRoutes`接口。

```
type IRoutes interface {
    Use(...HandlerFunc) IRoutes

    Handle(string, string, ...HandlerFunc) IRoutes
    Any(string, ...HandlerFunc) IRoutes
    GET(string, ...HandlerFunc) IRoutes
    POST(string, ...HandlerFunc) IRoutes
    DELETE(string, ...HandlerFunc) IRoutes
    PATCH(string, ...HandlerFunc) IRoutes
    PUT(string, ...HandlerFunc) IRoutes
    OPTIONS(string, ...HandlerFunc) IRoutes
    HEAD(string, ...HandlerFunc) IRoutes

```

```
    StaticFile(string, string) IRoutes
    Static(string, string) IRoutes
    StaticFS(string, http.FileSystem) IRoutes
}
```

因为`RenterGroup`实现了`IRoutes`定义的所有请求动词，而且`gin.Default`返回的`Engine`类型继承了`RenterGroup`，所以使用起来非常简单，只需要通过`gin.Default`实例化对象，接下来所有的路由操作都通过该对象使用即可。



基本的REST ful范例

```
// 2-1-restful.go  基本restful api
package main

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

// 测试方法
```

```go
// curl -X PUT http://localhost:8080/article
// curl -X POST http://localhost:8080/article
// curl -X GET http://localhost:8080/article
// curl -X DELETE http://localhost:8080/article

func main() {
    router := gin.Default()
    // 请求动词的第一个参数是请求路径，第二个参数是用于逻辑处理的函数
    router.POST("/article", func(c *gin.Context) {
        c.String(200, "article post")
    })
    router.DELETE("/article", func(c *gin.Context) {
        c.String(200, "article delete")
    })
    router.PUT("/article", func(c *gin.Context) {
        c.String(200, "article put")
    })
    router.GET("/article", func(c *gin.Context) {
        c.String(200, "article get")
    })
    router.GET("/article/:id", func(c *gin.Context) {
        id := c.Param("id")
        c.String(200, id)
    })

    router.GET("/article/:id/*action", func(c *gin.Context) {
        id := c.Param("id")
        action := c.Param("action")
        c.String(200, id+" "+action)
    })

    router.Run()
}
```

请求动词的第一个参数是请求路径，第二个参数是用于逻辑处理的函数，可以是匿名的或是其他地方定义的函数名。不同的请求动词可以定义相同的路径，只需要切换动词就可以进入对应的处理逻辑。

```
curl -X PUT http://localhost:8080/article
```

```
curl -X POST http://localhost:8080/article
curl -X GET http://localhost:8080/article
curl -X DELETE http://localhost:8080/article
```

路由参数

如下URL：

```
/users/123
/users/456
/users/23456
```

以上等等，我们有很多用户，如果我们都一个个为这些用户注册这些路由（URL），那么我们是很难注册完的，而且我们还会有新注册的用户，可见这种办法不行。
我们观察这些路由（URL），发现它们具备一定的规则：前面都是`users`，后面是`users`的`id`。这样我们就可以把这些路由归纳为：

```
/users/id
```

这样我们就知道只有`id`这部分是可以变的，前面的`users`是不变的。可变的`id`可以当成我们API服务输入的参数，这样我们就可以通过这个`id`参数，获取对应的用户信息，这种URL匹配的模式，我们称之为路由参数。

:路由

在`Gin`中，要实现以上路由参数非常简单：

```go
// 2-2-route-param.go :路由携带参数
package main

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

func main() {
    r := gin.Default()
    r.GET("/users/:id", func(c *gin.Context) {
        id := c.Param("id")
        c.String(200, "The user id is  %s", id)
    })
    r.Run(":8080")
```

```
13 }
```

我们运行如上代码，打开浏览器，输入`http://localhost:8080/users/123`，就可以看到如下信息：

```
1 The user id is  123
```

我们可以更换`http://localhost:8080/users/123`中的id `123` 为其他字符串，会发现都可以正常打印，这就是路由匹配、路由正则，或者路由参数。
`Gin`的路由采用的是`httprouter`，所以它的路由参数的定义和`httprouter`也是一样的。
`/users/:id` 就是一种路由匹配模式，也是一个通配符，其中`:id`就是一个路由参数，我们可以通过`c.Param("id")`获取定义的路由参数的值，然后用来做事情，比如打印出来。
`/users/:id`这种匹配模式是精确匹配的，只能匹配一个，我们举几个例子说明：

```
1 Pattern: /users/:id
2 /users/123           匹配
3 /users/哈哈         匹配
4 /users/123/go       不匹配
5 /users/              不匹配
```

这里我故意写了`/users/哈哈`，并且是匹配的，意思就是对于`Gin`路径中的匹配都是字符串，它是不区分数字、字母和汉字的，都匹配。
这里还需要说明的是，`Gin`的路由是单一的，不能有重复。比如这里我们注册了`/users/:id`,那么我们就不能再注册匹配`/users/:id`模式的路由，比如：

```
1 r.GET("/users/list", func(c *gin.Context) {
2     //省略无关代码
3 })
```

这时候我们运行程序的话，会出现如下提示：

```
1 panic: 'list' in new path '/users/list' conflicts with existing wi
  ldcard ':id' in existing prefix '/users/:id'
```

通配符重复了，路由必须要唯一。

*路由

`:`号的路由参数，这种路由参数最常用。还有一种不常用的就是`*`号类型的参数，表示匹配所有。
以`/users/*id`为例：

```
1 Pattern: /users/*id
```

```
/users/123          匹配
/users/哈哈         匹配
/users/123/go       匹配
/users/             匹配
```

我们把上面的例子改下：

```go
// 2-3-route-param.go *路由携带参数
package main

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

// 浏览器里访问http://localhost:8080/users/123，会看到如下信息：
// The user id is  /123
// 我们获取到的id不是123了，而是/123，多了一个/
func main() {
    r := gin.Default()
    r.GET("/users/*id", func(c *gin.Context) {
        id := c.Param("id")
        c.String(200, "The user id is  %s", id)
    })
    r.Run(":8080")
}
```

现在我们运行，浏览器里访问`http://localhost:8080/users/123`，会看到如下信息：

```
The user id is  /123
```

是否发现区别了，我们获取到的`id`不是`123`了，而是`/123`，多了一个`/`
同样的你试试`http://localhost:8080/users/123/go`会发现显示的信息是：

```
The user id is  /123/go
```

是一个`/`开头的路径字符串。
这里要特别说明一点的是，如果你用浏览器访问`http://localhost:8080/users`，会被重定向到`http://localhost:8080/users/`，然后显示的信息如下：

```
The user id is  /
```

重定向的根本原因在于`/users`没有匹配的路由，但是有匹配`/users/`的路由，所以就会被重定向到`/users/`。现在我们注册一个`/users`来验证下这个猜测：

```go
// 2-4-route-param.go *路由携带参数, 匹配路由
package main

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

func main() {
    r := gin.Default()
    r.GET("/users/*id", func(c *gin.Context) {
        id := c.Param("id")
        c.String(200, "The user id is  %s", id)
    })
    r.GET("/users", func(c *gin.Context) {
        c.String(200, "这是真正的/users")
    })
    r.Run(":8080")
}
```

现在再访问`http://localhost:8080/users`，会看到显示的信息变成了：

```
这是真正的/users
```

这也间接证明了`/users/*id`和`/users`这两个路由是不冲突的，可以被`Gin`注册。
以上自动重定向的原理，得益于`gin.RedirectTrailingSlash` 等于`true`的配置。如果我们把它改为`false`就不会自动重定向了。

```go
// 2-5-route-param.go *路由携带参数,禁止重定向
package main

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

func main() {
    r := gin.Default()
    r.RedirectTrailingSlash = false
    r.GET("/users/*id", func(c *gin.Context) {
      id := c.Param("id")
      c.String(200, "The user id is  %s", id)
```

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

现在我们运行程序，访问`http://localhost:8080/users`发现显示的信息是`404 page not found`

3 URL查询参数

Querystring parameters ，比如：

```
https://www.baidu.com/search?q=golang&sitesearch=https%3A%2F%2Fwww.baidu.com
```

URL查询参数，或者也可以简称为URL参数，是存在于我们请求的URL中，以`?`为起点，后面的`k=v&k1=v1&k2=v2`这样的字符串就是查询参数，比如上面示例中的：

```
?q=golang&sitesearch=https%3A%2F%2Fwww.baidu.com
```

这个示例中有两个查询参数键值对：

```
q=golang
sitesearch=https%3A%2F%2Fwww.baidu.com
```

第一个key是`q`，对应的值是`golang`。第二个key是`sitesearch`,对应的值是`https%3A%2F%2F`www.baidu.com，它们通过`&`相连。在URL中，多个查询参数键值对通过`&`相连。

Gin获取查询参数

在`Gin`中，为我们提供了简便的方法来获取查询参数的值，我们只需要知道查询参数的key（参数名）就可以了。

```go
// 3-1-url-param.go url参数获取
package main

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

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

```
    r.GET("/", func(c *gin.Context) {
        c.String(200, c.Query("wechat"))
    })
    r.Run(":8080")
}
```

我们运行这段代码，打开浏览器访问`http://localhost:8080/?wechat=baidu_org`,就可以看到`baidu_org`文字。这表示我们通过`c.Query("wechat")`获取到了查询参数`wechat`的值是`baidu_org`。

`Query`方法为我们提供了获取对应`key`的值的能力，如果该`key`不存在，则返回`""`字符串。如果对于一些数字参数，比如`id`如果返回为空的话，我们进行字符串转数字的时候会报错，这时候，我们就可以通过`DefaultQuery`方法指定一个默认值：

```
// 3-2-url-param.go url参数获取
package main

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

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.DefaultQuery("id", "0")
        c.String(200, c.DefaultQuery("wechat", "default baidu_or
g"))
    })
    r.Run(":8080")
}
```

比如这样，尤其是第二个例子，默认为0，让我们字符串转数字很方便。

```
func (c *Context) Query(key string) string {
    value, _ := c.GetQuery(key)
    return value
}

func (c *Context) DefaultQuery(key, defaultValue string) string {
    if value, ok := c.GetQuery(key); ok {
        return value
```

```
    }
    return defaultValue
}
```

看下这两个函数的源代码实现，它们都是调用的`GetQuery`方法获取对应的值，唯一不同的是`DefaultQuery`会判断对应的`key`是否存在，如果不存在的话，则返回默认`defaultValue`值。

原理解析

从以上两个获取查询参数值的方法可以看到，他们调用的都是`GetQuery`，这也是`gin.Context`的一个方法，它和`Query`唯一不同的是，它返回两个值，可以告诉我们要获取的`key`是否存在。

```
value, ok := c.GetQuery("wechat")
```

如果我们自己的业务中，需要这类功能，可以用`GetQuery`来代替`Query`方法。
`GetQuery`方法的底层实现其实是`c.Request.URL.Query().Get(key)`，通过`url.URL.Query()`来获取所有的参数键值对。

```
本质上是调用的GetQueryArray，取的数组中第一个值
func (c *Context) GetQuery(key string) (string, bool) {
  if values, ok := c.GetQueryArray(key); ok {
    return values[0], ok
  }
  return "", false
}
func (c *Context) GetQueryArray(key string) ([]string, bool) {
  c.getQueryCache() //缓存所有的键值对
  if values, ok := c.queryCache[key]; ok && len(values) > 0 {
    return values, true
  }
  return []string{}, false
}
func (c *Context) getQueryCache() {
  if c.queryCache == nil {
    c.queryCache = c.Request.URL.Query()
  }
}
```

从以上的实现代码中，可以看到最终的实现都在`GetQueryArray`方法中，找到对应的`key`就返回对应的`[]string`，返回就返回空数组。
这里`Gin`进行了优化，通过缓存所有的键值对，提升代码的查询效率。这里缓存的`queryCache`本质上是`url.Values`，也是一个`map[string][]string`。

```
type Values map[string][]string
```

其中`c.Request.URL.Query()`这个方法就是把`?k=v&k1=v1&k2=v2`这类查询键值对转换为`map[string][]string`，所以还是很耗性能的，这里`Gin`采用了缓存的做法提高了性能挺好，这也是`Gin`成为性能最快的Golang Web 框架的原因之一吧。

4 接收数组和 Map

QueryArray

在实际的业务开发中，我们有些业务多选的，比如一个活动有多个人参加，一个问题有多个答案等等，对于这类业务功能来说，如果是通过查询参数提交的，它们的URL大概这样`?a=b&a=c&a=d`,`key`值都一样，但是对应的`value`不一样。
这类URL查询参数，就是一个数组，那么在Gin中我们如何获取它们呢?
这里举个例子，比如有一份调查问卷，问我有哪些自媒体，我选择个人博客和微信公众号

```
// 4-1-query-array.go
package main

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

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.JSON(200, c.QueryArray("media"))
    })
    r.Run(":8080")
}
```

运行代码，在浏览器里访问`http://localhost:8080/?media=blog&media=wechat`，会看到如下信息：

```
["blog","wechat"]
```

我们的自媒体信息，已经作为一个数组被输出了，非常简单，这样我们就可以很方便的处理多选的业务。`QueryArray`方法也有对应的`GetQueryArray`方法，区别在于返回对应的`key`是否存在，这里不再举例。

`QueryArray`和`GetQueryArray`源代码实现已经在上一篇Golang Gin 实战（四）| URL查询参数的获取和原理分析分析了，这里不再赘述，大家可以再看下上一篇文章。

QueryMap

`QueryMap`其实就是把满足一定格式的URL查询参数，转换为一个`map`，假设有a,b,c三个人，他们对应的id是123,456,789.那么用map的方式表示，这种格式类似于：

```
?ids[a]=123&ids[b]=456&ids[c]=789
```

从以上URL看，关键在于`key`，这个key必须符合`map`的定义，`[]`外面的必须相同，也就是`ids`这个`map`变量名，`[]`里面的,也就是`map`的key不能相同，这样就满足了`Gin`定义的把URL查询参数转换为`map`的格式定义。

```go
// 4-2-query-map.go
package main

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

// 现在运行代码，访问http://localhost:8080/map?ids[a]=123&ids[b]=456&ids[c]=789，就会看到如下信息：
// {"a":"123","b":"456","c":"789"}
func main() {
    r := gin.Default()
    r.GET("/map", func(c *gin.Context) {
        c.JSON(200, c.QueryMap("ids"))
    })
    r.Run(":8080")
}
```

获取`map`的方法很简单，把`ids`作为`key`即可。现在运行代码，访问`http://localhost:8080/map?ids[a]=123&ids[b]=456&ids[c]=789`，就会看到如下信息：

```
{"a":"123","b":"456","c":"789"}
```

我们输入的信息，正好被我们打印出来了。

`GetQueryMap`和`QueryMap`是一样的，只是返回了对应的`key`是否存在。

QueryMap 的原理

```
func (c *Context) QueryMap(key string) map[string]string {
  dicts, _ := c.GetQueryMap(key)
  return dicts
}
func (c *Context) GetQueryMap(key string) (map[string]string, bool
) {
  c.getQueryCache()
  return c.get(c.queryCache, key)
}
```

`QueryMap`是通过`GetQueryMap`，最终都是`c.get`这个方法实现，我们只需要分析`c.get`就可以了。
注意这里同样用到了`getQueryCache`进行缓存提高性能。

```
func (c *Context) get(m map[string][]string, key string) (map[str
ing]string, bool) {
  dicts := make(map[string]string)
  exist := false
  for k, v := range m {
    if i := strings.IndexByte(k, '['); i >= 1 && k[0:i] == key {
      if j := strings.IndexByte(k[i+1:], ']'); j >= 1 {
        exist = true
        dicts[k[i+1:][:j]] = v[0]
      }
    }
  }
```

这段实现代码看着比较绕，其实挺简单，它有两个参数，一个`m`其实就是缓存的所有查询参数键值对`queryCache`,另外一个就是我们要找的`key`
因为`Gin`定义的map的URL特殊格式化，所以这里需要判断是否有`[]`，如果有的话，并且`key`匹配，那么这个键值对就是我们需要找的，把它存在`dicts`即可，最终返回的是这个`dicts`。

6 表单参数

除了通过URL查询参数提交数据到服务器外，常用的还有通过Form表单的方式。Form表单相比URL查询参数，用户体验好，可以承载更多的数据，尤其是文件上传，所以也更为方便。

Form 表单

对于Form表单，我们不会陌生，比如`input`文本框、密码框等等，可以让我们输入一些数据，然后点击「保存」、「提交」等按钮，把数据提交到服务器的。
对于Form表单来说，有两种提交方式`GET`和`POST`。其中`GET`方式就是我们前面讲的URL查询参数的方式，参考即可获得对应的参数键值对，这篇文章主要介绍`POST`的方式的表单，而`Gin`处理的也是这种表单。

Gin 接收表单数据

Gin 对于表单数据的获取也非常简单，为我们提供了和获取URL查询参数一样的系列方法。

```go
// 5-1-form-param.go
package main

import (
    "fmt"
    "io/ioutil"

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

// curl -d "message=pingye" http://localhost:8080/form_post
// {"message":"pingye","nick":"anonymous","status":"posted"}
func main() {
    r := gin.Default()
    r.POST("/form_post", func(c *gin.Context) {
        body, _ := ioutil.ReadAll(c.Request.Body)
        fmt.Println("---body--- \r\n " + string(body))
        fmt.Println("---header--- \r\n")
        for k, v := range c.Request.Header {
            fmt.Println(k, v)
        }
        message := c.PostForm("message")
        nick := c.DefaultPostForm("nick", "anonymous")

        c.JSON(200, gin.H{
            "status":  "posted",
```

```
            "message": message,
            "nick":    nick,
        })
    })
    r.Run(":8080")
}
```

运行这段代码，然后打开终端输入`curl -d  "message=pingye" http://localhost:8080/form_post` 回车，就会看到打印的如下信息：

```
{"message":"pingye","nick":"anonymous","status":"posted"}baidu_org
```

这里我们通过`curl`这个工具来模拟POST请求，当然你可以可以使用`Postman`比较容易操作的可视化工具。
在这个`Gin`示例中，使用`PostForm`方法来获取相应的键值对，它接收一个`key`，也就是我们`html`中`input`这类表单标签的`name`属性值。
`PostForm`方法和查询参数的`Query`是一样的，如果对应的`key`不存在则返回空字符串。

PostFormArray()方法获取表单参数

首先创建一个html：

```
<!-- 5-2-form-array.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>提交</title>
</head>
<body>
    <form action="http://127.0.0.1:8080/form" method="post" encty
pe="application/x-www-form-urlencoded">
        用户名:<input type="text" name="username">
        <br>
        密 &nbsp码:<input type="password" name="password">
        兴 &nbsp趣:
        <input type="checkbox" value="run" name="hobby">跑步
        <input type="checkbox" value="game" name="hobby">游戏
        <input type="checkbox" value="money" name="hobby">金钱
```

```
        <br>
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

通过PostForm()方法获取表单参数：

```
// 5-2-form-array.go
package main

import (
    "fmt"
    "net/http"

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

func getting(c *gin.Context) {

}

func main() {
    // 1创建路由,默认使用了两个中间件Logger(),Recovery()
    r := gin.Default()
    // 指明html加载文件目录
    r.LoadHTMLGlob("./*")
    r.Handle("GET", "/", func(context *gin.Context) {
        fmt.Println("/访问")
        // 返回HTML文件，响应状态码200，html文件名为index.html，模板参数为nil
        context.HTML(http.StatusOK, "5-2-form-array.html", nil)
    })
    // 2绑定路由规则,
    // gin.Context,封装了request和respose
    r.POST("/form", func(c *gin.Context) {
        fmt.Println("/form访问")
        typeStr := c.DefaultQuery("type", "alert")
        username := c.PostForm("username")
```

```
31          password := c.PostForm("password")
32          // 多选框
33          hobbys := c.PostFormArray("hobby")
34          c.String(http.StatusOK, fmt.Sprintf("type is %s, usernam
    e:%s, password:%s, hooby:%v",
35              typeStr, username, password, hobbys))
36      })
37      // 3监听端口，默认8080
38      r.Run(":8080")
39  }
```

然后浏览器访问http://127.0.0.1:8080

提交后

type is alert, username:darren, password:2323, hooby:[run game]

Gin PostForm系列方法

和查询参数方法一样，对于表单的参数接收，`Gin`也提供了一系列的方法，他们的用法和查询参数的一样。

查询参数	Form表单	说明
Query	PostForm	获取key对应的值，不存在为空字符串
GetQuery	GetPostForm	多返回一个key是否存在的结果
QueryArray	PostFormArray	获取key对应的数组，不存在返回一个空数组
GetQueryArray	GetPostFormArray	多返回一个key是否存在的结果
QueryMap	PostFormMap	获取key对应的map，不存在返回空map
GetQueryMap	GetPostFormMap	多返回一个key是否存在的结果
DefaultQuery	DefaultPostForm	key不存在的话，可以指定返回的默认值

通过这个表格以及对应和说明，可以更好记一些。

实现原理

关于`PostForm`系列方法的实现原理和`Query`系列类似，并且遵循`Query-GetQuery-GetQueryArray`这么一个内部调用顺序，所以我们直接看`GetPostFormArray`的源代码即可。

```
func (c *Context) GetPostFormArray(key string) ([]string, bool) {
  c.getFormCache()
  if values := c.formCache[key]; len(values) > 0 {
    return values, true
  }
  return []string{}, false
}
```

这里关键点在于`getFormCache`缓存Form表单的数据，接下来就是根据`key`获取对应的值了。

```
func (c *Context) getFormCache() {
  if c.formCache == nil {
    c.formCache = make(url.Values)
    req := c.Request
    if err := req.ParseMultipartForm(c.engine.MaxMultipartMemory)
; err != nil {
      if err != http.ErrNotMultipart {
        debugPrint("error on parse multipart form array: %v", err
)
      }
    }
    c.formCache = req.PostForm
  }
}
```

从以上实现代码可以看到，`formCache`表单缓存其实也是一个`url.Values`，通过调用`http.Request`的`ParseMultipartForm`对提交的表单解析，获得里面的数据保存在`http.Request`的`PostForm`字段中，最后从`req.PostForm`获取表单数据，赋值给`c.formCache`表单缓存即可。

这里需要注意的是保存表单缓存的内存大小，`Gin`默认给的是32M,通过`const defaultMultipartMemory = 32 << 20 // 32 MB`可以看出。

如果你觉得不够，可以提前通过修改`MaxMultipartMemory`的值增加，比如：

```go
r := gin.Default()
r.MaxMultipartMemory = 100 << 20
```

最后的`GetPostFormMap`方法，其实现原理和`GetQueryMap`一模一样，这里不再赘述，大家可以看下源代码。

小结

不管是查询参数还是表单提交，`Gin`都为我们做了很好的封装，并且通过缓存提升性能，让我们不再去关注这些具体的细节，可以专注于我们的业务实现，这也是框架的魅力所在。
所以，在我们日常的开发中，不管你是做什么业务，什么语言，还是要尽可能的复用、性能提升等，这样才能逐步的成长。

6 上传文件

上传单个文件FormFile

```html
<!-- 6-1-upload-file.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登录</title>
</head>
<body>
    <form action="http://127.0.0.1:8080/upload" method="post" enctype="multipart/form-data">
        头像:
        <input type="file" name="file">
        <br>
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

```go
// 6-1-upload-file.go
```

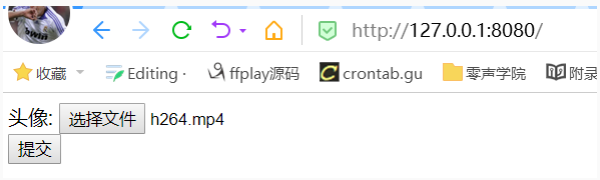
```go
package main

import (
    "fmt"
    "log"
    "net/http"

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

func main() {
    // 1创建路由,默认使用了两个中间件Logger(),Recovery()
    r := gin.Default()
    // 指明html加载文件目录
    r.LoadHTMLGlob("./*")
    r.Handle("GET", "/", func(context *gin.Context) {
        fmt.Println("/访问")
        // 返回HTML文件，响应状态码200，html文件名为index.html，模板参数为nil
        context.HTML(http.StatusOK, "6-1-upload-file.html", nil)
    })
    // 2绑定路由规则,
    // gin.Context,封装了request和respose
    r.POST("/upload", func(c *gin.Context) {
        file, _ := c.FormFile("file")
        log.Println("file:", file.Filename)
        c.SaveUploadedFile(file, file.Filename)
        c.String(200, fmt.Sprintf("%s upload file!", file.Filename))
    })
    // 3监听端口，默认8080
    r.Run(":8080")
}

```

`curl` 测试：

```
curl -X POST http://localhost:8080/upload \
```

```
  -F "file=@/Users/appleboy/test.zip" \
  -H "Content-Type: multipart/form-data"
```

浏览器测试



上传多个文件MultipartForm

```
<!-- 6-2-upload-multi-files.html -->
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Multiple file upload</title>
</head>
<body>
<h1>Upload multiple files with fields</h1>

<form action="/upload" method="post" enctype="multipart/form-data">
    Name: <input type="text" name="name"><br>
    Email: <input type="email" name="email"><br>
    Files: <input type="file" name="files" multiple><br><br>
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

```
package main

import (
```

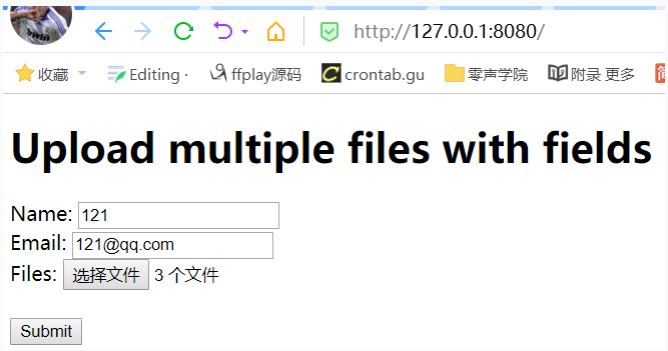
```
    "fmt"
    "log"
    "net/http"
    "path/filepath"

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

func main() {
    router := gin.Default()
    // Set a lower memory limit for multipart forms (default is 32 MiB)
    router.MaxMultipartMemory = 8 << 20 // 8 MiB
    router.Static("/", "./public")
    router.POST("/upload", func(c *gin.Context) {
        name := c.PostForm("name")
        email := c.PostForm("email")

        // Multipart form
        form, err := c.MultipartForm()
        if err != nil {
            c.String(http.StatusBadRequest, fmt.Sprintf("get form err: %s", err.Error()))
            return
        }
        files := form.File["files"]

        for _, file := range files {
            log.Println("file:", file.Filename)
            filename := filepath.Base(file.Filename)
            if err := c.SaveUploadedFile(file, filename); err != nil {
                c.String(http.StatusBadRequest, fmt.Sprintf("upload file err: %s", err.Error()))
                return
            }
        }

        c.String(http.StatusOK, fmt.Sprintf("Uploaded successfully %d files with fields name=%s and email=%s.", len(files), name,
```

```
    email))
39      })
40      router.Run(":8080")
41  }
```

`curl` 测试：

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

浏览器



7 分组路由

在我们开发定义路由的时候，可能会遇到很多部分重复的路由：

```
1 /admin/users
2 /admin/manager
3 /admin/photo
```

以上等等，这些路由最前面的部分`/admin/`是相同的，如果我们一个个写也没问题，但是不免会觉得琐碎、重复，无用劳动，那么有没有一种更好的办法来解决呢？`Gin`为我们提供的解决方案就是分组路由

分组路由

类似以上示例，就是分好组的路由，分组的原因有很多，比如基于模块化，把同样模块的放在一起，比如基于版本，把相同版本的API放一起，便于使用。在有的框架中，分组路由也被称之为命名空间。

假如我们现在要升级新版本APi，但是旧的版本我们又要保留以兼容老用户。那么我们使用`Gin`就可以这么做

```go
// 7-1-route-group.go
package main

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

func main() {
    r := gin.Default()
    //V1版本的API
    v1Group := r.Group("/v1")
    v1Group.GET("/users", func(c *gin.Context) {
        c.String(200, "/v1/users")
    })
    v1Group.GET("/products", func(c *gin.Context) {
        c.String(200, "/v1/products")
    })
    //V2版本的API
    v2Group := r.Group("/v2")
    v2Group.GET("/users", func(c *gin.Context) {
        c.String(200, "/v2/users")
    })
    v2Group.GET("/products", func(c *gin.Context) {
        c.String(200, "/v2/products")
    })
    r.Run(":8080")
}
```

只需要通过`Group`方法就可以生成一个分组，然后用这个分组来注册不同路由，用法和我们直接使用`r`变量一样，非常简单。这里为了便于阅读，一般都是把不同分组的，用`{}`括起来。

```go
// 7-2-route-group.go
package main

import (
```

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

func main() {
    r := gin.Default()
    v1Group := r.Group("/v1")
    {
        v1Group.GET("/users", func(c *gin.Context) {
            c.String(200, "/v1/users")
        })
        v1Group.GET("/products", func(c *gin.Context) {
            c.String(200, "/v1/products")
        })
    }
    v2Group := r.Group("/v2")
    {
        v2Group.GET("/users", func(c *gin.Context) {
            c.String(200, "/v2/users")
        })
        v2Group.GET("/products", func(c *gin.Context) {
            c.String(200, "/v2/products")
        })
    }
    r.Run(":8080")
}
```

路由中间件

通过`Group`方法的定义，我们可以看到，它是可以接收两个参数的：

func (group *RouterGroup) Group(relativePath string, handlers ...HandlerFunc) *RouterGroup

第一个就是我们注册的分组路由（命名空间）；第二个是一个`...HandlerFunc`，可以把它理解为这个分组路由的中间件，所以这个分组路由下的子路由在执行的时候，都会调用它。

这样就给我们带来很多的便利，比如请求的统一处理，比如`/admin`分组路由下的授权校验处理。比如刚刚上面的例子：

```
v1Group := r.Group("/v1", func(c *gin.Context) {
    fmt.Println("/v1中间件")
 })
```

这样不管你是访问`/v1/users`，还是访问`/v1/products`,控制台都会打印出`/v1`中间件。
从`Group`的方法定义，还可以看到，我们可以注册多个`HandlerFunc`，对分组路由进行多次处理。

分组路由嵌套

我们不光可以定义一个分组路由，还可以在这个分组路由中再添加一个分组路由，达到分组路由嵌套的目的，这种业务场景也不少，比如：

```
/v1/admin/users
/v1/admin/manager
/v1/admin/photo
```

V1版本下的`admin`模块，我们使用`Gin`可以这么实现。

```
v1AdminGroup := v1Group.Group("/admin")
  {
    v1AdminGroup.GET("/users", func(c *gin.Context) {
      c.String(200, "/v1/admin/users")
    })
    v1AdminGroup.GET("/manager", func(c *gin.Context) {
      c.String(200, "/v1/admin/manager")
    })
    v1AdminGroup.GET("/photo", func(c *gin.Context) {
      c.String(200, "/v1/admin/photo")
    })
  }
```

如上代码，再调用一次`Group`生成一个分组路由即可，就是这么简单，通过这种方式你还可以继续嵌套。

原理解析

那么以前这种分组路由这么方便，实现会不会很复杂呢？我们来看看源代码，分析一下它的实现方式。
在分析之前，我们先来看看我们最开始用的`GET`方法签名。

```
func (group *RouterGroup) GET(relativePath string, handlers ...HandlerFunc) IRoutes {
  return group.handle("GET", relativePath, handlers)
}
```

注意第一个参数`relativePath`，这是一个相对路径，也就是我们传给`Gin`的是一个相对路径，那么是相对谁的呢?

```
func (group *RouterGroup) handle(httpMethod, relativePath string, handlers HandlersChain) IRoutes {
  absolutePath := group.calculateAbsolutePath(relativePath)
  handlers = group.combineHandlers(handlers)
  group.engine.addRoute(httpMethod, absolutePath, handlers)
  return group.returnObj()
}
```

通过这句`absolutePath := group.calculateAbsolutePath(relativePath)`代码，我们可以看出是相对当前的这个`group`(方法接收者)的。
现在`calculateAbsolutePath`方法的源代码我们暂时不看，回过头来看`Group`这个生成分组路由的方法。

```
func (group *RouterGroup) Group(relativePath string, handlers ...HandlerFunc) *RouterGroup {
  return &RouterGroup{
    Handlers: group.combineHandlers(handlers),
    basePath: group.calculateAbsolutePath(relativePath),
    engine:   group.engine,
  }
}
```

这里要注意的是,我们通过`gin.Default()`生成的`gin.Engine`其实包含一个`RouterGroup`(嵌套组合),所以它可以用`RouterGroup`的方法。
`Group`方法又生成了一个`*RouterGroup`，这里最重要的就是`basePath`,它的值是`group.calculateAbsolutePath(relativePath)`，和我们刚刚暂停的分析的方法一样，既然这样，就来看看这个方法吧。

```
func (group *RouterGroup) calculateAbsolutePath(relativePath string) string {
  return joinPaths(group.basePath, relativePath)
}
```

就是一个基于当前`RouterGroup`的`basePath`的路径拼接，所以我们通过`Group`方法改变新生成`RouterGroup`中的`basePath`，就达到了路由分组的目的。
同时因为多次调用`Group`方法，都是基于上一个`RouterGroup`的`basePath`拼接成下一个`RouterGroup`的`basePath`，也就达到了路由分组嵌套的目的。
而我们通过`gin.Default()`生成的最初的`gin.Engine`，对应的`basePath`是`/`，根节点。
这是一种非常棒的代码实现方式，简单的代码，是强大的功能。

GIn中间件

Gin框架允许开发者在处理请求的过程中，加入用户自己的钩子（Hook）函数。这个钩子函数就叫中间件，中间件适合处理一些公共的业务逻辑，比如登录认证、权限校验、数据分页、记录日志、耗时统计等

Gin默认中间件

在Gin中，我们可以通过Gin提供的默认函数，来构建一个自带默认中间件的`*Engine`。

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

`Default`函数会默认绑定两个已经准备好的中间件，它们就是Logger 和 Recovery，帮助我们打印日志输出和`painc`处理。

```
func Default() *Engine {
  debugPrintWARNINGDefault()
  engine := New()
  engine.Use(Logger(), Recovery())
  return engine
}
```

从中我们可以看到，Gin的中间件是通过`Use`方法设置的，它接收一个可变参数，所以我们同时可以设置多个中间件。

```
func (engine *Engine) Use(middleware ...HandlerFunc) IRoutes
```

到了这里其实我们应该更加明白了，一个Gin的中间件，其实就是Gin定义的一个`HandlerFunc`,而它在我们Gin中经常使用，比如：

```
r.GET("/", func(c *gin.Context) {
    fmt.Println("首页")
    c.JSON(200, "")
 })
```

后面的`func(c *gin.Context)`这部分其实就是一个`HandlerFunc`。

中间件实现HTTP Basic Authorization

HTTP Basic Authorization 是HTTP常用的认证方案，它通过Authorization 请求消息头含有服务器用于验证用户代理身份的凭证，格式为：

```
Authorization: Basic <credentials>
```

如果认证不成功，服务器返回401 Unauthorized 状态码以及WWW-Authenticate 消息头，让客户端输入用户名和密码进一步认证。
在Gin中，为我们提供了`gin.BasicAuth`帮我们生成基本认证的中间件，方便我们的开发。

```
// 7-3-basic-auth.go
package main

import (
    "fmt"

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

func main() {
    r := gin.Default()
    r.Use(gin.BasicAuth(gin.Accounts{
        "admin": "123456",
    }))

    r.GET("/", func(c *gin.Context) {
        fmt.Println("进入主页")
        c.JSON(200, "首页")
    })

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

我们添加一个用户名为`admin`,密码是`123456`的账户，用于HTTP 基本认证。现在我们运行启动，访问`http://localhost:8080/`，这时候只有我们输入正确的用户名和密码，才能看到`首页`，否则是看不到的，这样我们就达到了授权的目的,就是这么简单。

针对特定URL的Basic Authorization

其实在实际的项目开发中，我们基本上不太可能对所有的URL都进行认证的，一般只有一些需要认证访问的数据才需要认证，比如网站的后台，那么这时候我们就可以用分组路由来处理。

```go
// 7-4-url-auth.go  针对特定的url授权
package main

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

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.JSON(200, "首页")
    })
    adminGroup := r.Group("/admin")
    adminGroup.Use(gin.BasicAuth(gin.Accounts{
        "admin": "123456",
    }))
    adminGroup.GET("/index", func(c *gin.Context) {
        c.JSON(200, "后台首页")
    })
```

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

现在我们运行访问`/`首页是可以正常显示的，但是我们访问`/admin/index`会提示输入密码，其实所有`/admin/*`下的URL都会让输入密码才能访问，这就是我们分组路由的好处，我们通过把中间件加到`/admin`这个分组路由上，就可以达到我们的目的。
通过分组路由的控制，我们可以比较灵活的设置HTTP认证，粒度可以自己随意控制。

自定义中间件

我们已经知道，Gin的中间件其实就是一个`HandlerFunc`,那么只要我们自己实现一个`HandlerFunc`，就可以自定义一个自己的中间件。现在我们以统计每次请求的执行时间为例，来演示如何自定义一个中间件。

```
func costTime() gin.HandlerFunc {
    return func(c *gin.Context) {
        //请求前获取当前时间
        nowTime := time.Now()
        fmt.Println("获取起始时间")
        //请求处理
        c.Next() // 处理handler
        //处理后获取消耗时间
        fmt.Println("获取结束时间")
        costTime := time.Since(nowTime)
        url := c.Request.URL.String()
        fmt.Printf("the request URL %s cost %v\n", url, costTime)
    }
}
```

以上我们就实现了一个Gin中间件，比较简单，而且有注释加以说明，这里要注意的是`c.Next`方法，这个是执行后续中间件请求处理的意思（含没有执行的中间件和我们定义的GET方法处理），这样我们才能获取执行的耗时。也就是在`c.Next`方法前后分别记录时间，就可以得出耗时。
有了自定义的中间件，我们就可以这么使用。

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

```
    r.GET("/", func(c *gin.Context) {
        fmt.Println("处理过程")
        c.JSON(200, "首页")
    })
    r.Run(":8080")
}
```

现在启动程序，在浏览器里打开就可以看到如下日志信息了。
the request URL / cost 26.533µs

通过自定义中间件,我们可以很方便的拦截请求，来做一些我们需要做的事情，比如**日志记录、授权校验、各种过滤**等等。

7 再谈中间件

Gin框架允许开发者在处理请求的过程中，加入用户自己的钩子（Hook）函数。这个钩子函数就叫中间件，中间件适合处理一些公共的业务逻辑，比如登录认证、权限校验、数据分页、记录日志、耗时统计等

定义中间件

Gin中的中间件必须是一个gin.HandlerFunc类型

入门案例

```
// 7-1-basic.go

package main

import (
    "fmt"
    "net/http"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    c.JSON(http.StatusOK, gin.H{
        "msg": "indx",
```

```
    })
}

//定义一个中间件
func m1(c *gin.Context) {
    fmt.Println("m1 in ....")
}

func main() {
    r := gin.Default()
    //GET(relativePath string, handlers ...HandlerFunc) IRoutes
    r.GET("/index", m1, indexHandler)

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

```
package main

import (
    "fmt"
    "net/http"
    "time"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    c.JSON(http.StatusOK, gin.H{
        "msg": "indx",
    })
}

//定义一个中间件:统计耗时
func m1(c *gin.Context) {
    fmt.Println("m1 in ....")
    //计时
```

```
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    //输出
    // m1 in ....
    //index in ...
    //cost:%v
    // 996.8µs
}
func main() {
    r := gin.Default()
    //GET(relativePath string, handlers ...HandlerFunc) IRoutes
    r.GET("/index", m1, indexHandler) //先执行m1函数再执行indexHandler函数

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

注册中间件

在gin框架中，可以为每个路由添加任意数量的中间件。

为全局路由注册

```
// 7-3-global.go
package main

import (
    "fmt"
    "net/http"
    "time"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    c.JSON(http.StatusOK, gin.H{
```

```
        "msg": "indx",
    })
}

//定义一个中间件:统计耗时
func m1(c *gin.Context) {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
func m2(c *gin.Context) {
    fmt.Println("m2 in ....")
    c.Next() //调用后续的处理函数
    fmt.Println("m2 out")
}
func main() {
    r := gin.Default()
    r.Use(m1, m2) //全局注册中间件函数m1,m2    洋葱模型    类似递归调用
    //GET(relativePath string, handlers ...HandlerFunc) IRoutes
    //r.GET("/index",m1,indexHandler)  //先执行m1函数再执行indexHand
ler函数
    r.GET("/index", indexHandler)
    r.GET("/shop", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
        })
    })
    r.GET("/user", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
        })
    })
    r.Run(":8080")
}
```

```go
// 7-4-global.go 全局中间件
package main

import (
    "fmt"
    "net/http"
    "time"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    c.JSON(http.StatusOK, gin.H{
        "msg": "indx",
    })
}

//定义一个中间件:统计耗时
func m1(c *gin.Context) {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
func m2(c *gin.Context) {
    fmt.Println("m2 in ....")
    //c.Next() //调用后续的处理函数
    c.Abort() //阻止后续调用
    //return   //return 立即结束m2函数
    //m1 in ....
    //m2 in ....
    //cost:%v
    // 0s
    //m1 out
```

```
40      fmt.Println("m2 out")
41  }
42
43  //func authMiddleware(c *gin.Context)  {   //通常写成闭包
44  //  //是否登陆的判断
45  //  //if 是登陆用户
46  //  //c.Next()
47  //  //else
48  //  //c.Abort()
49  //}
50  func authMiddleware(doCheck bool) gin.HandlerFunc { //开关注册
51      //连接数据库
52      //或着其他准备工作
53      return func(c *gin.Context) {
54          if doCheck {
55              //是否登陆的判断
56              //if 是登陆用户
57              //c.Next()
58              //else
59              //c.Abort()
60          } else {
61              c.Next()
62          }
63      }
64
65  }
66  func main() {
67      r := gin.Default()
68      r.Use(m1, m2, authMiddleware(true)) //全局注册中间件函数m1,m2
    洋葱模型    类似递归调用
69      //GET(relativePath string, handlers ...HandlerFunc) IRoutes
70      //r.GET("/index",m1,indexHandler)  //先执行m1函数再执行indexHand
    ler函数
71      r.GET("/index", indexHandler)
72      r.GET("/shop", func(c *gin.Context) {
73          c.JSON(http.StatusOK, gin.H{
74              "msg": "index",
75          })
76      })
77      r.GET("/user", func(c *gin.Context) {
```

```
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
        })
    })
    r.Run(":8080")
}
```

为某个路由单独注册

```
// 7-5-route.go
import (
    "fmt"
    "github.com/gin-gonic/gin"
    "net/http"
    "time"
)
//定义一个中间件:统计耗时
func m1(c *gin.Context)  {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
func main() {
    r := gin.Default()
    r.GET("/user", m1, func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
        })
    })

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

```go
// 7-6-route.go
import (
    "fmt"
    "github.com/gin-gonic/gin"
    "net/http"
    "time"
)
func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    c.JSON(http.StatusOK, gin.H{
        "msg": "indx",
    })
}
//定义一个中间件:统计耗时
func m1(c *gin.Context)  {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
func m2(c *gin.Context)  {
    fmt.Println("m2 in ....")
    //c.Next() //调用后续的处理函数
    c.Abort() //阻止后续调用
    //return   //return 立即结束m2函数
    //m1 in ....
    //m2 in ....
    //cost:%v
    // 0s
    //m1 out
    fmt.Println("m2 out")
}
func main() {
    r := gin.Default()
    r.GET("/user", m1,m2, func(c *gin.Context) {  //可以单独多个路由
        c.JSON(http.StatusOK, gin.H{
```

```
            "msg": "index",
        })
    })
    //[GIN-debug] Listening and serving HTTP on :8080
    //m1 in ....
    //m2 in ....
    //m2 out
    //cost:%v
    // 0s
    //m1 out
    r.Run(":8080")
}
```

为路由组注册中间件

```
// 7-7-group.go
package main

import (
    "net/http"

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

func authMiddleware(doCheck bool) gin.HandlerFunc { //开关注册
    //连接数据库
    //或着其他准备工作
    return func(c *gin.Context) {
        if doCheck {
            //是否登陆的判断
            //if 是登陆用户
            //c.Next()
            //else
            //c.Abort()
        } else {
            c.Next()
        }
    }

```

```
}

func main() {
    r := gin.Default()
    //路由组注册中间件方法1:
    xx1Group := r.Group("/xx1", authMiddleware(true))
    {
        xx1Group.GET("/index", func(c *gin.Context) {
            c.JSON(http.StatusOK, gin.H{"msg": "xx1Group"})
        })
    }
    //路由组注册中间件方法2:
    xx2Group := r.Group("/xx2")
    xx2Group.Use(authMiddleware(true))
    {
        xx2Group.GET("/index", func(c *gin.Context) {
            c.JSON(http.StatusOK, gin.H{"msg": "xx2Group"})
        })
    }
    r.Run(":8080")
}
```

跨中间件存取值

```
// 7-8-meddle-ctx.go 跨中间件取值
package main

import (
    "fmt"
    "net/http"
    "time"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    name, ok := c.Get("name") //从上下文中取值，跨中间件存取值
    if !ok {
```

```
16          name = "匿名用户"
17      }
18      c.JSON(http.StatusOK, gin.H{
19          "msg": name,
20      })
21  }
22
23  //定义一个中间件:统计耗时
24  func m1(c *gin.Context) {
25      fmt.Println("m1 in ....")
26      //计时
27      start := time.Now()
28      c.Next() //调用后续的处理函数 执行indexHandler函数
29      //c.Abort() //阻止调用后续的处理函数
30      cost := time.Since(start)
31      fmt.Println("cost:%v\n", cost)
32      fmt.Println("m1 out")
33  }
34  func m2(c *gin.Context) {
35      fmt.Println("m2 in ....")
36      c.Set("name", "0voice") //在上下文中设置c的值
37      fmt.Println("m2 out")
38  }
39  func authMiddleware(doCheck bool) gin.HandlerFunc { //开关注册
40      //连接数据库
41      //或着其他准备工作
42      return func(c *gin.Context) {
43          if doCheck {
44              //是否登陆的判断
45              //if 是登陆用户
46              c.Next()
47              //else
48              //c.Abort()
49          } else {
50              c.Next()
51          }
52      }
53  }
54  func main() {
55      r := gin.Default()
```

```
    r.Use(m1, m2, authMiddleware(true)) //全局注册中间件函数m1,m2 洋葱模型    类似递归调用
    //GET(relativePath string, handlers ...HandlerFunc) IRoutes
    //r.GET("/index",m1,indexHandler)  //先执行m1函数再执行indexHandler函数
    r.GET("/index", indexHandler)
    r.Run(":8080")
}
```

中间件注意事项

gin.Default()

gin.Default()默认使用了Logger和Recovery中间件，其中：Logger中间件将日志写入gin.DefaultWriter，即使配置了GIN_MODE=release。**Recovery中间件会recover任何panic**。如果有panic的话，会写入500响应码。如果不想使用上面两个默认的中间件，可以使用gin.New()新建一个没有任何默认中间件的路由。

```go
// 7-9-self.go 自定义中间件
package main

import (
    "fmt"
    "net/http"
    "time"

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

func indexHandler(c *gin.Context) {
    fmt.Println("index in ...")
    name, ok := c.Get("name") //从上下文中取值，跨中间件存取值
    if !ok {
        name = "匿名用户"
    }
    c.JSON(http.StatusOK, gin.H{
        "msg": name,
    })
}

```

```go
//定义一个中间件:统计耗时
func m1(c *gin.Context) {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
func m2(c *gin.Context) {
    fmt.Println("m2 in ....")
    c.Set("name", "0voice") //在上下文中设置c的值
    fmt.Println("m2 out")
}
func authMiddleware(doCheck bool) gin.HandlerFunc { //开关注册
    //连接数据库
    //或着其他准备工作
    return func(c *gin.Context) {
        if doCheck {
            //是否登陆的判断
            //if 是登陆用户
            c.Next()
            //else
            //c.Abort()
        } else {
            c.Next()
        }
    }
}
func main() {
    //r := gin.Default()  //默认使用Logger()和Recovery()中间件
    r := gin.New()
    r.Use(m1, m2, authMiddleware(true)) //全局注册中间件函数m1,m2
洋葱模型    类似递归调用
    r.GET("/index", indexHandler)
    r.GET("/shop", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
```

```
        })
    })
    r.GET("/user", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "msg": "index",
        })
    })
    r.Run(":8080")
}
```

gin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（c *gin.Context），必须使用其只读副本（c.Copy()）。

```
//定义一个中间件:统计耗时
func m1(c *gin.Context)  {
    fmt.Println("m1 in ....")
    //计时
    start := time.Now()
    go funcXX(c.Copy()) //在funcXX中只能使用c的拷贝
    c.Next() //调用后续的处理函数 执行indexHandler函数
    //c.Abort() //阻止调用后续的处理函数
    cost := time.Since(start)
    fmt.Println("cost:%v\n", cost)
    fmt.Println("m1 out")
}
```

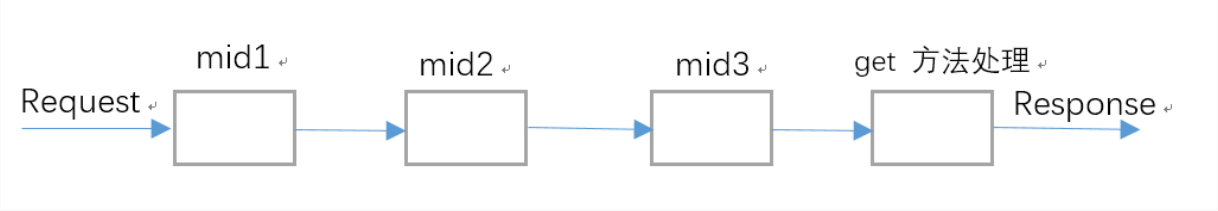
gin框架中间件c.Next()理解

```
package main

import (
    "fmt"
    "net/http"

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

```go

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

    mid1 := func(c *gin.Context) {
        fmt.Println("mid1 start")
        c.Next()
        // fmt.Println("mid1 Next1")
        // c.Next()
        // fmt.Println("mid1 Next2")
        // c.Next()
        // fmt.Println("mid1 Next3")
        // c.Next()
        // fmt.Println("mid1 Next4")
        fmt.Println("mid1 end")
    }
    mid2 := func(c *gin.Context) {
        fmt.Println("mid2 start")
        //c.Abort()
        c.Next()
        fmt.Println("mid2 end")
    }
    mid3 := func(c *gin.Context) {
        fmt.Println("mid3 start")
        c.Next()
        fmt.Println("mid3 end")
    }
    router.Use(mid1, mid2, mid3)
    router.GET("/index", func(c *gin.Context) {
        fmt.Println("process get request")
        c.JSON(http.StatusOK, "hello")
    })
    router.Run(":8080")
}
```

上述代码中使用了3个中间件（mid1,mid2,mid3），加上最后的路由处理即返回hello部分，共4个handles。

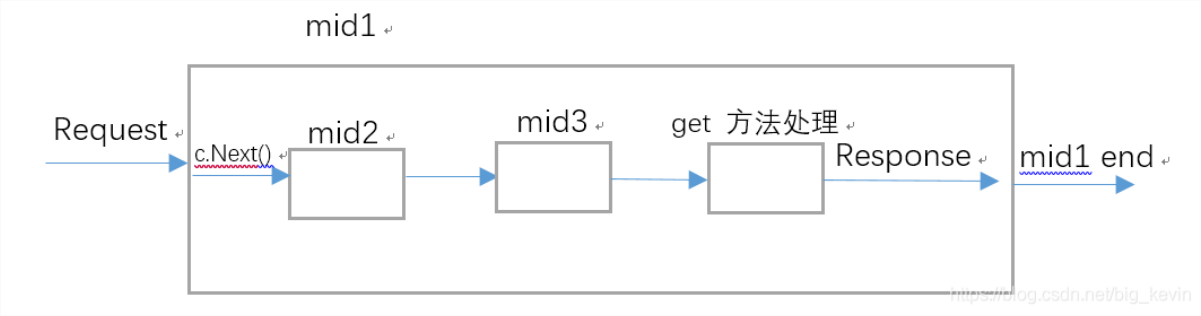
1. 如果注释掉3个中间件中的c.Next()，则执行情况如下：



程序会依次调用中间件函数，最终调用get 路由处理函数并返回，打印日志如下：

```
mid1 start
mid1 end
mid2 start
mid2 end
mid3 start
mid3 end
process get request
```

2. 如果仅在mid1中间件中使用c.Next()，则执行流程如下：



如果在mid1中调用c.Next(),则程序会从当前节点调用下一个中间件即mid2，打印日志如下：

```
mid1 start
mid2 start
mid2 end
mid3 start
mid3 end
process get request
mid1 end
```

总结：最后的get路由处理函数可以理解为最后的中间件，在不是调用c.Abort()的情况下，所有的中间件都会被执行到。当某个中间件调用了c.Next(),则整个过程会产生嵌套关系。如果某个中间件调用了c.Abort(),则此中间件结束后会直接返回，后面的中间件均不会调用。

8 json、struct、xml、yaml、protobuf渲染

各种数据格式的响应

- json、结构体、XML、YAML类似于java的properties、ProtoBuf

分别对所有数据格式举个列子

范例

```go
// 8-1-json-xml-protobuf.go
package main

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

// 分别在浏览器中输入以下内容，会得到具体的响应信息。
// http://127.0.0.1:8080/someJSON
// http://127.0.0.1:8080/someStruct
// http://127.0.0.1:8080/someXML
// http://127.0.0.1:8080/someYAML
// http://127.0.0.1:8080/someProtoBuf

func main() {
    r := gin.Default()
    //1. json响应
    r.GET("/someJSON", func(c *gin.Context) {
        c.JSON(200, gin.H{"message": "someJSON", "status": 200})
    })
    //2. 结构体响应
    r.GET("/someStruct", func(c *gin.Context) {
        var msg struct {
            Name    string
            Message string
            Number  int
        }
        msg.Name = "root"
        msg.Message = "message"
        msg.Number = 123
        c.JSON(200, msg)
    })

    //3. XML
    r.GET("/someXML", func(c *gin.Context) {
        c.XML(200, gin.H{"message": "abc"})
    })
```

```
39
40      //4. YAML响应
41      r.GET("/someYAML", func(c *gin.Context) {
42          c.YAML(200, gin.H{"name": "you"})
43      })
44
45      //5.Protobuf格式，谷歌开发的高效存储读取的工具
46      r.GET("/someProtoBuf", func(c *gin.Context) {
47          reps := []int64{int64(1), int64(2)}
48          //定义数据
49          label := "label"
50          //传protobuf格式数据
51          data := &protoexample.Test{
52              Label: &label,
53              Reps:  reps,
54          }
55          c.ProtoBuf(200, data)
56      })
57
58      r.Run(":8080")
59  }
```

分别在浏览器中输入以下内容，会得到具体的响应信息。

http://127.0.0.1:8080/someJSON

http://127.0.0.1:8080/someStruct

http://127.0.0.1:8080/someXML

http://127.0.0.1:8080/someYAML

http://127.0.0.1:8080/someProtoBuf

9 HTML模板渲染

- gin支持加载HTML模板，然后根据模板参数进行配置并返回响应的数据，本质上就是字符串替换
- LoadHTMLGlob()方法可以加载模板文件

最简单的例子

把index.html放在view目录下，index.html代码如下：

```
1 <!DOCTYPE html>
```

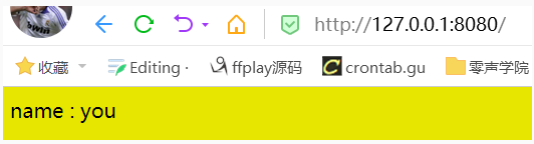
```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{.title}}</title>
</head>
<body bgcolor="#E6E600">
name : {{.name}}
</body>
</html>
```

代码

```
// 9-1-html.go
package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.LoadHTMLGlob("view/*")
    r.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", gin.H{"title": "我是gi
n", "name": "you"})
    })
    r.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", gin.H{"title": "我是gi
n", "name": "you"})
    })
    r.Run(":8080")
}
```

测试 http://127.0.0.1:8080/

效果如下:

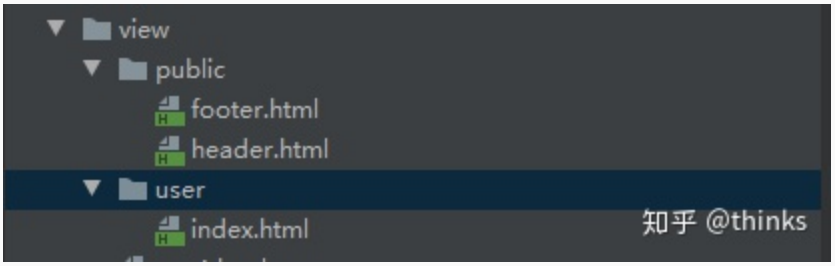


复杂点的例子

如果view目录还有子文件夹，使用 `r.LoadHTMLGlob("tem/**/*")` 去加载html文件夹，然后使用c.HTML(xxx, "user/index.html",xx)，渲染html模板。

- 如果你想进行头尾分离就是下面这种写法了：

文件结构：



header.html代码：

```
{{define "public/header"}}
  <!DOCTYPE html>
  <html lang="en">
  <head>
      <meta charset="UTF-8">
      <title>{{.title}}</title>
  </head>
  <body>
  {{end}}
```

footer.html代码：

```
{{define "public/footer"}}
      </body>
      </html>
  {{end}}
```

index.html代码：

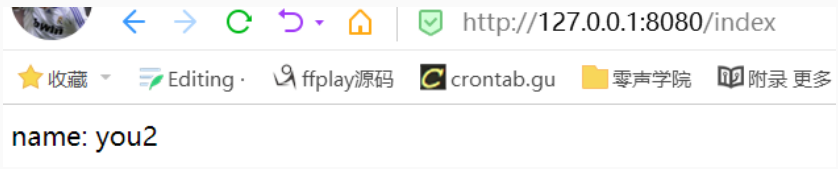
```
{{ define "user/index.html" }}
      {{template "public/header" .}}
      name: {{.name}}
      {{template "public/footer" .}}
```

```
5   {{ end }}
```

go代码：

```
1 // 9-2-html.go 复杂一些的例子
2 package main
3
4 import (
5     "net/http"
6
7     "github.com/gin-gonic/gin"
8 )
9
10 func main() {
11     r := gin.Default()
12     r.LoadHTMLGlob("view2/**/*")
13     r.GET("/index", func(c *gin.Context) {
14         c.HTML(http.StatusOK, "user/index.html", gin.H{"title":
   "我是gin", "name": "you2"})
15     })
16     r.Run()
17 }
```

效果演示：



静态文件目录

- 如果你需要引入静态文件需要定义一个静态文件目录

```
1 r.Static("/assets", "./assets")
```

更多参考：realtime-advanced目录

重定向

```go
// 9-3-redirect.go

package main

import (
    "net/http"

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

func main() {
    r := gin.Default()
    r.LoadHTMLGlob("view/*")
    r.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", gin.H{"title": "我是gi
n", "name": "you"})
    })
    r.GET("/", func(c *gin.Context) {
        c.Redirect(http.StatusMovedPermanently, "/index")   // 重
定向
    })
    r.Run(":8080")
}
```

10 异步协程

- goroutine机制可以方便地实现异步处理
- 另外，在启动新的goroutine时，不应该使用原始上下文，必须使用它的只读副本。

```go
// 10-1-sync-async.go
package main

import (
    "log"
    "time"

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

```
)

func main() {
    r := gin.Default()
    //1. 异步
    r.GET("/long_async", func(c *gin.Context) {
        //需要搞一个副本
        copyContext := c.Copy()
        //异步处理
        go func() {
            time.Sleep(3 * time.Second)
            log.Println("异步执行：" + copyContext.Request.URL.Path
)
        }()
    })
    //2. 同步
    r.GET("/long_sync", func(c *gin.Context) {
        time.Sleep(3 * time.Second)
        log.Println("同步执行：" + c.Request.URL.Path)
    })
    r.Run()
}
```

测试

http://127.0.0.1:8080/long_async

http://127.0.0.1:8080/long_sync

11 Gin源码简要分析

简单介绍gin源码，主要是路由和中间件的相关绑定使用流程，以及Context设计，但是不包括render

概述

通过日常对gin场景出发，深入源码，总结介绍gin的核心设计。包含：Engine / HandlerFunc / RouterGroup(Router) / Context。在日常使用中常见的就以上概念，汇总如下：

概念	解释	应用意义
Engine	引擎	web server的基础支持，也是服务的入口 和 根级数据结构

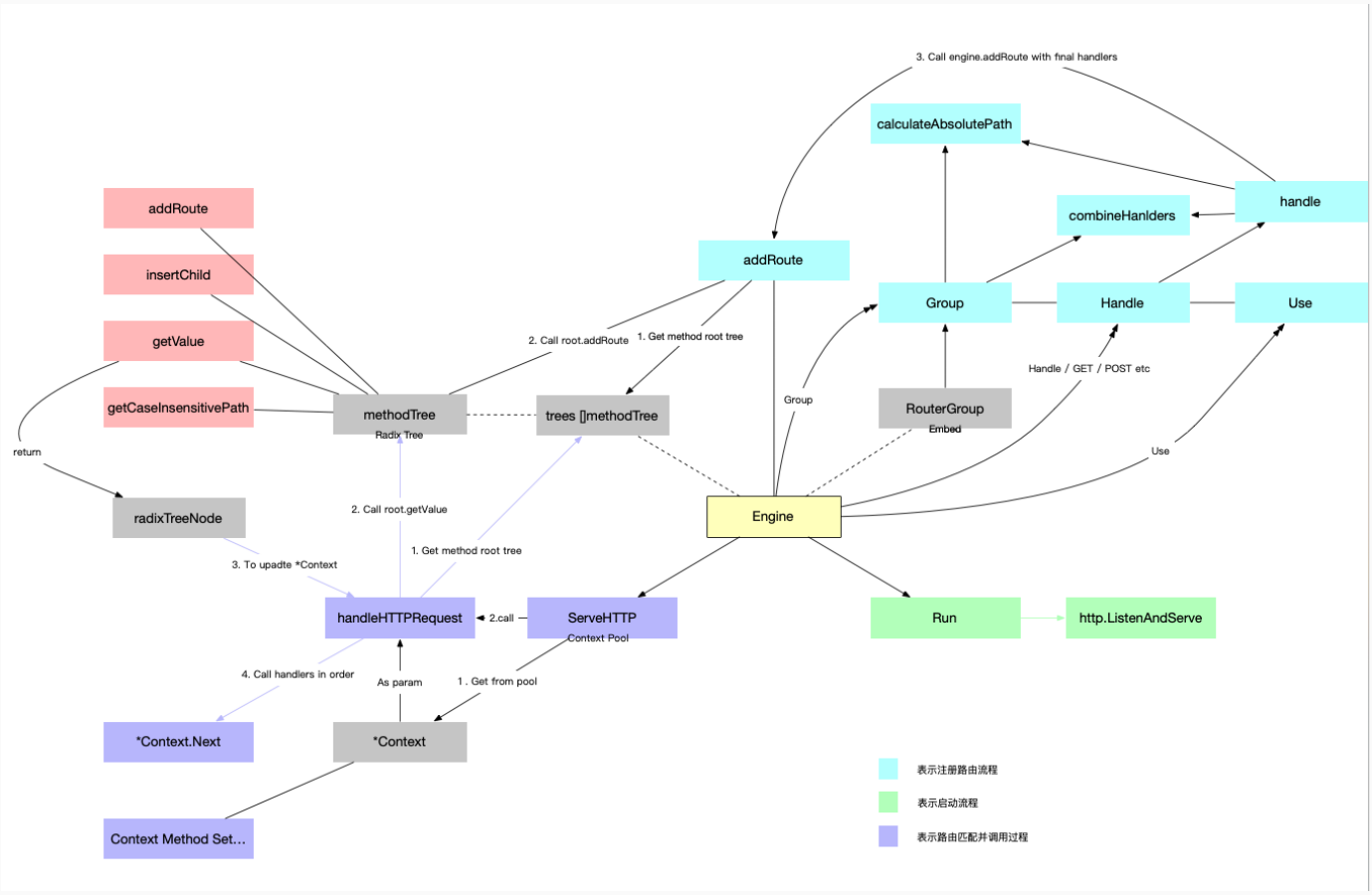
概念	解释	应用意义
RouterGroup(Router)	路由	用于支持gin,REST路由绑定和路由匹配的基础，源于radix-tree数据结构支持
HandlerFunc	处理函数	逻辑处理器和中间件实现的函数签名
Context	上下文	封装了请求和响应的操作，为HandlerFunc的定义和中间件模式提供支持

从DEMO开始

```
type barForm struct {
    Foo string  `form:"foo" binding:"required"`
    Bar int     `form:"bar" binding:"required"`
}
func (fooHdl FooHdl) Bar(c *gin.Context) {
    var bform = new(barForm)
    if err := c.ShouldBind(bform); err != nil {
        // true: parse form error
        return
    }
    // handle biz logic and generate response structure
    // c (gin.Context) methods could called to support process-co
ntroling
    c.JSON(http.StatusOK, resp)
    // c.String() alse repsonse to client
}
// mountRouters .
func mountRouters(engi *gin.Engine) {
    // use middlewares
    engi.Use(gin.Logger())
    engi.Use(gin.Recovery())

    // mount routers
    group := engi.Group("/v1")
    {
        fooHdl := demohtp.New()
        group.GET("/foo", fooHdl.Bar)
        group.GET("/echo", fooHdl.Echo)
        // subGroup := group.Group("/subg")
```

```
29          // subGroup.GET("/hdl1", fooHdl.SubGroupHdl1) // 最终路
    由: "targetURI = /v1/subg/hdl1"
30      }
31  }
32  func main() {
33      engi := gin.New()
34      mountRouters(engi)
35      if err := engi.Run(":8080"); err != nil {
36          log.Fatalf("engi exit with err=%v", err)
37      }
38  }
```

通过上述的代码就简单开启了一个gin server，其中就包括了常见的：路由注册，中间件注册，路由分组，服务启动。核心概念也就是刚刚在上文提到的那四个概念。概览流程如下图：



可以参照DEMO中的方法名在图中检索改流程。

ENGINE

```
1  // Engine is the framework's instance, it contains the muxer, mid
   dleware and configuration settings.
2  type Engine struct {
3      // 路由管理
```

```
    RouterGroup
    // 省略非关心属性
    // ...
    // context poo，支持context复用，减少对象创建提高性能。
    pool             sync.Pool

    // []methodTree方法树根节点集合 这部分在路由部分会详细介绍
    trees            methodTrees
}
```

Engine是根入口，它把RouterGroup结构图体嵌入自身，以获得了Group，GET，POST等路由管理方法。从 `Run` 方法的代码：

> 因为直接复制代码太多了，推荐下载源码来跳转，更便捷高效。

```
func (engine *Engine) Run(addr ...string) (err error) {
    defer func() { debugPrintError(err) }()
    address := resolveAddress(addr)
    debugPrint("Listening and serving HTTP on %s\n", address)
    // 使用的是标准库的http服务启动方法
    // 如果看过http.ListenAndServe函数签名，就知道engine实现了http.Handler方法，
    // 也就是它一定有一个ServeHTTP方法
    err = http.ListenAndServe(address, engine)
    return
}
```

通过层层跳转我们可以找到图中淡紫色部分中的 `handlerHTTPRequest` 方法，在ServeHTTP函数时，engine会从pool中取得一个Context对象，并准备好Request和ResponseWriter的相关数据设置到Context中去，提供给方法链调用。

```
// 调用中间件和请求逻辑函数
func (engine *Engine) handleHTTPRequest(c *Context) {
    httpMethod := c.Request.Method
    rPath := c.Request.URL.Path
    unescape := false
    if engine.UseRawPath && len(c.Request.URL.RawPath) > 0 {
        rPath = c.Request.URL.RawPath
        unescape = engine.UnescapePathValues
    }
    if engine.RemoveExtraSlash {
```

```
        rPath = cleanPath(rPath)
    }
    // 这里必须知道gin使用的是 “基数树（Radix Tree）” 用于存储路由
    // https://michaelyou.github.io/2018/02/10/%E8%B7%AF%E7%94%B1%E6%9F%A5%E6%89%BE%E4%B9%8BRadix-Tree/
    t := engine.trees
    for i, tl := 0, len(t); i < tl; i++ {
        if t[i].method != httpMethod {
            // 从这里可以知道，engine中的methodTrees []methodTree是请求方法分组的路由匹配树。
            continue
        }
        root := t[i].root
        // root数据结构（radix-tree）？如何查找?
        value := root.getValue(rPath, c.Params, unescape) // 找到路由对应的处理函数
        if value.handlers != nil {
            c.handlers = value.handlers // 复制节点的所有handlers
            c.Params = value.params     // 路由参数
            c.fullPath = value.fullPath // 全路径
            c.Next()                    // 依次调用HandlerChain中的函数
            c.writermem.WriteHeaderNow()
            return
        }
        // 省略，这部分流程控制也自行阅读
    }

    // 省略...
}
```

从上述流程我们可以总结得到图中紫色部分调用链：

engine.Run -> http.ListenAndServe -> engine.handleHTTPRequest -> c.Next

但是其中还有不清楚的问题：

- 路由是通过方法确定到该方法树的根节点，再查询到路由对应的节点，那么在radix-tree中是如何进行路由匹配的呢?
- c.handlers 直接从匹配到的路由节点的handlers复制得到，那么中间件是如何挂在到其中的呢?
- 获取到c.handlers之后，只调用了一次c.Next，那么该链路是如何继续调用下去的?
- Path Param是如何获取到？**root.getValue**

ROUTERGROUP & METHODTREE

这部分是除了Context概念之外理解gin的第二核心部分，提供路由注册，调用链路函数链处理，路由分组，路由匹配功能。在开始之前，必须知道gin的路由是由httprouter提供，而httprouter包是使用了radix-tree来实现路由管理功能。

```
// IRouter defines all router handle interface includes single and group router.
type IRouter interface {
    IRoutes
    Group(string, ...HandlerFunc) *RouterGroup
}
// IRoutes defines all router handle interface.
type IRoutes interface {
    Use(...HandlerFunc) IRoutes
    Handle(string, string, ...HandlerFunc) IRoutes
    Any(string, ...HandlerFunc) IRoutes
    GET(string, ...HandlerFunc) IRoutes
    POST(string, ...HandlerFunc) IRoutes
    DELETE(string, ...HandlerFunc) IRoutes
    PATCH(string, ...HandlerFunc) IRoutes
    PUT(string, ...HandlerFunc) IRoutes
    OPTIONS(string, ...HandlerFunc) IRoutes
    HEAD(string, ...HandlerFunc) IRoutes
    StaticFile(string, string) IRoutes
    Static(string, string) IRoutes
    StaticFS(string, http.FileSystem) IRoutes
}
// RouterGroup is used internally to configure router, a RouterGroup is associated with
// a prefix and an array of handlers (middleware).
type RouterGroup struct {
    Handlers HandlersChain // 中间件
    basePath string        // 路由前缀
    engine   *Engine       // 指向engine入口的指针
    root     bool          // 是否是root
}
```

.路由注册

通过任意一个路由注册方法，都可以进入到 `func (group *RouterGroup) handle(httpMethod, relativePath string, handlers HandlersChain) IRoutes` 这个方法。

```
// 挂载路由的实际处理函数
func (group *RouterGroup) handle(httpMethod, relativePath string, handlers HandlersChain) IRoutes {
    // 计算最终的绝对路径 base + relativePath
    // 注意base也是绝对路径
    absolutePath := group.calculateAbsolutePath(relativePath)
    // 合并处理函数，将中间件和逻辑函数结合在一起
    // 一般来说这里传入的handlder是逻辑函数 len(handlers) = 1
    // 只有少数的handler会有自己的中间件处理函数 len(handlers) > 1
    handlers = group.combineHandlers(handlers)
    // 将处理好的 HandlersChain 加载到Radix Tree中去
    // 这也表明，这里的RouterGroup只会载处理路由时发挥作用
    group.engine.addRoute(httpMethod, absolutePath, handlers)
    return group.returnObj()
}
```

> engine在初始化的时候会设置RouterGroup.basePath = “/” engine是将相同请求方法挂载到同一棵树下

看到这里回到engine.addRoute中去，通过深入发现如下调用链：

engine.GET -> routergroup.GET -> routergroup.handle -> engine.addRoute -> methodTree.addRoute -> node(radix-tree’s node).insertChild

.路由分组

RouterGroup通过Group函数来衍生下一级别的RouterGroup，会拷贝父级RouterGroup的中间件，重新计算basePath。

```
func (group *RouterGroup) Group(relativePath string, handlers ...HandlerFunc) *RouterGroup {
    return &RouterGroup{
        Handlers: group.combineHandlers(handlers),           // 初始化时，拷贝父亲节点的中间件
        basePath: group.calculateAbsolutePath(relativePath), // 初始化时，计算孩子group的绝对路径
        engine:   group.engine,
    }
}
```

.中间件挂载

有两个地方提供给中间件挂载，一个是RouterGroup.Use集中挂载中间件；另一个地方是挂载特定路由的时候(RouterGroup.GET / POST 等等)，将中间件和逻辑处理函数一起挂载。第二种方式在路由注册的时候已经见过了，这里再说下Use方法，`RouterGroup.Handlers = append(RouterGroup.Handlers, middlewareHandlers)`，这里只是简单的将后来的中间件复制到RouterGroup的中间件上，没有其他的逻辑。

.路由匹配

这部分全都是交由radix-tree来负责的部分。

HANDLERFUNC

```
type HandlerFunc func(*Context)  // 处理函数签名
type HandlersChain []HandlerFunc // 处理函数链
```

这部分没什么好说的，就是定义了函数签名，指定Context作为上下文传递的关键数据。

CONTEXT

Context是上下文传递的核心，它包括了请求处理，响应处理，表单解析等重要工作。

```
// Context is the most important part of gin. It allows us to pass variables between middleware,
// manage the flow, validate the JSON of a request and render a JSON response for example.
type Context struct {
    writermem responseWriter // 实现了http.ResponseWriter 和 gin.ResponseWriter
    Request   *http.Request  // http.Request, 暴露给handler
    // gin.ResponseWriter 包含了:
    // http.ResponseWriter, http.Hijacker, http.Flusher, http.CloseNotifier和额外方法
    // 暴露给handler, 是writermm的复制
    Writer    ResponseWriter
    Params   Params        // 路径参数
    handlers HandlersChain // 调用链
    index    int8          // 当前handler的索引
    fullPath string
    engine *Engine // 对engine的引用
    Keys map[string]interface{} // c.GET / c.SET 的支持, 常用于session传递。
```

```
    // Errors is a list of errors attached to all the handlers/middlewares who used this context.
    Errors errorMsgs
    // Accepted defines a list of manually accepted formats for content negotiation.
    Accepted []string
    // query 参数缓存
    queryCache url.Values
    // 表单参数缓存, 跟queryCache作用类似
    formCache url.Values
}
```

.调用链流转和控制

从Engine部分已经知道，当路由被匹配到之后会执行一次c.Next，Next逻辑非常简单如下：

```
func (c *Context) Next() {
    // 从c.reset 可以知道 c.index == -1
    c.index++
    for c.index < int8(len(c.handlers)) {
        c.handlers[c.index](c) // 发生调用
        c.index++
    }
}
// 63，这意味这如果最大链路超过了63，那么通过index的流程控制就出问题了。
const abortIndex int8 = math.MaxInt8 / 2
func (c *Context) Abort() {
    c.index = abortIndex
}
```

要注意的是，当你在调用链中的某一个handler中调用了c.Abort之类的函数，调用链会直接退出也是通过c.index来控制的。除了调用链流转，在Context这一部分还比较重要的是，参数的解析，响应处理。

.参数解析

参数传递一共有一下几种方式：

序号	参数类型	解释	CONTEXT支持
1	path param	在URI中将参数作为路径的一部分	c.Param(“key”) string
2	query param	在URI中以”?“开始的，”key=value”形式的部分	c.Query(“key”) string

序号	参数类型	解释	CONTEXT支持
3	body [form; json; xml 等等]	根据请求头Content-Type判定或指定	c.Bind类似函数

```
// Content-Type MIME of the most common data formats.
const (
    MIMEJSON              = "application/json"
    MIMEHTML              = "text/html"
    MIMEXML               = "application/xml"
    MIMEXML2              = "text/xml"
    MIMEPlain             = "text/plain"
    MIMEPOSTForm          = "application/x-www-form-urlencoded"
    MIMEMultipartPOSTForm = "multipart/form-data"
    MIMEPROTOBUF          = "application/x-protobuf"
    MIMEMSGPACK           = "application/x-msgpack"
    MIMEMSGPACK2          = "application/msgpack"
    MIMEYAML              = "application/x-yaml"
)
```

> 关于第三种参数解析是最常用的，也是最复杂的部分包含的内容比较多，这里就不展开了。别忘了，binding的同时还有参数校验是基于“github.com/go-playground/validator”实现的。

.响应处理

常用的响应方法有 `c.String(http.Status, string)`, `c.JSON(http.Status, interface{})`

c.Render(code http.Status, r gin.Render) -> r.Render

```
// Render interface is to be implemented by JSON, XML, HTML, YAML
   and so on.
type Render interface {
    // Render writes data with custom ContentType.
    Render(http.ResponseWriter) error
    // WriteContentType writes custom ContentType.
    WriteContentType(w http.ResponseWriter)
}
```

这里以gin.render.JSON为例：

> 这里json包并不是直接使用的标准库json，而是经过了一层包装用于支持jsoniter替换json。 这一支持是基于golang选择性编译实现的。

```
// Render (JSON) writes data with custom ContentType.
func (r JSON) Render(w http.ResponseWriter) (err error) {
    if err = WriteJSON(w, r.Data); err != nil {
        panic(err)
    }
    return
}
// WriteContentType (JSON) writes JSON ContentType.
func (r JSON) WriteContentType(w http.ResponseWriter) {
    writeContentType(w, jsonContentType)
}
// WriteJSON marshals the given interface object and writes it wi
th custom ContentType.
func WriteJSON(w http.ResponseWriter, obj interface{}) error {
    writeContentType(w, jsonContentType)
    encoder := json.NewEncoder(w)
    err := encoder.Encode(&obj)
    return err
}
```

总结

这里知识讲了把gin作为API Server开发时常用到的主要流程和数据结构，其中更多的流程控制部分被省略了，可以自行结合代码和源码阅读。这里主要解析了gin中函数调用链路和handlers流程控制，***RadixTree***的部分都被省略了，算法菜鸡不敢乱说，掌握后再结合gin的源码重读。
文中代码偏多，最好是结合源码和图片食用。

> 水平有限，如有错误，欢迎勘误指正🙏 。

参考文献

- https://michaelyou.github.io/2018/02/10/%E8%B7%AF%E7%94%B1%E6%9F%A5%E6%89%BE%E4%B9%8BRadix-Tree/