



下载APP



24 | Web 服务：Web 服务核心功能有哪些，如何实现？

2021-07-20 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 17:37 大小 16.15M



你好，我是孔令飞。从今天开始，我们进入实战第三站：服务开发。在这个部分，我会讲解 IAM 项目各个服务的构建方式，帮助你掌握 Go 开发阶段的各个技能点。

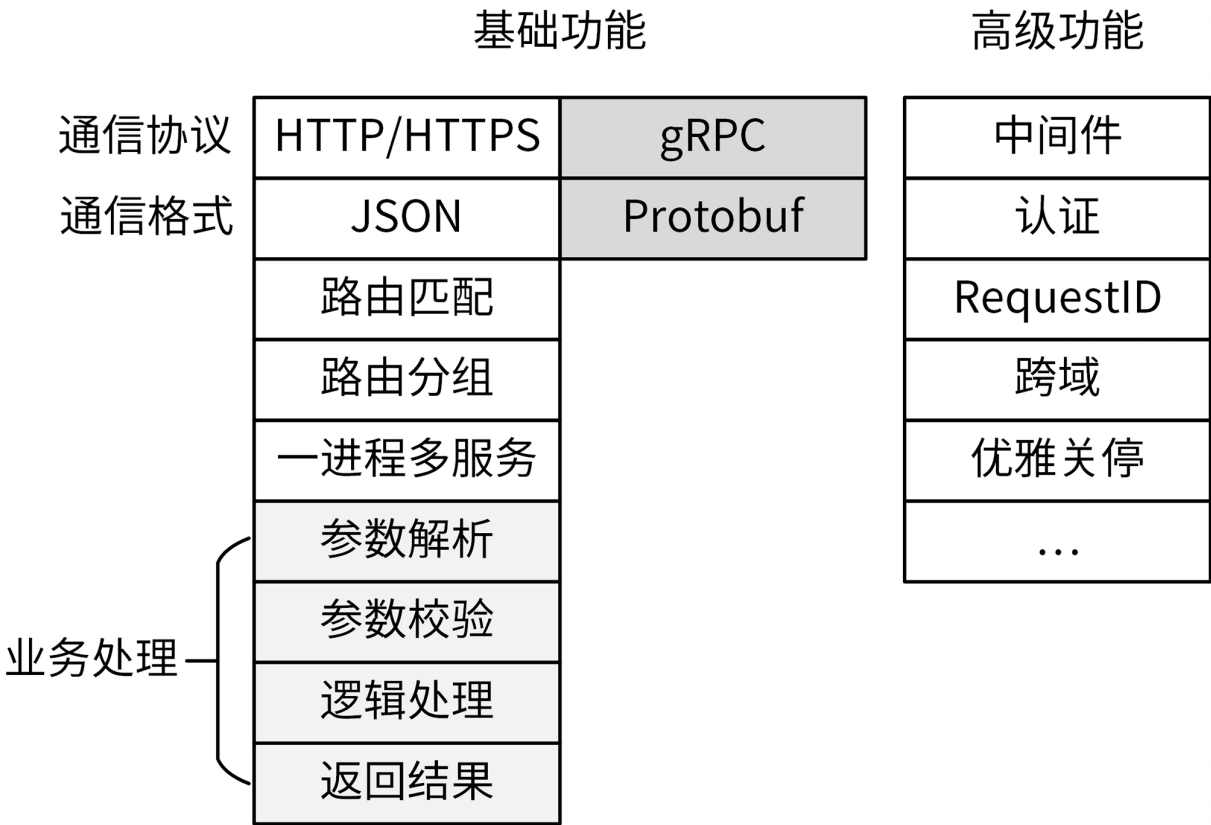
在 Go 项目开发中，绝大部分情况下，我们是在写能提供某种功能的后端服务，这些功能以 RPC API 接口或者 RESTful API 接口的形式对外提供，能提供这两种 API 接口的服务也统称为 Web 服务。今天这一讲，我就通过介绍 RESTful API 风格的 Web 服务，来给你介绍下如何实现 Web 服务的核心功能。

那今天我们就来看下，Web 服务的核心功能有哪些，以及如何开发这些功能。



Web 服务的核心功能

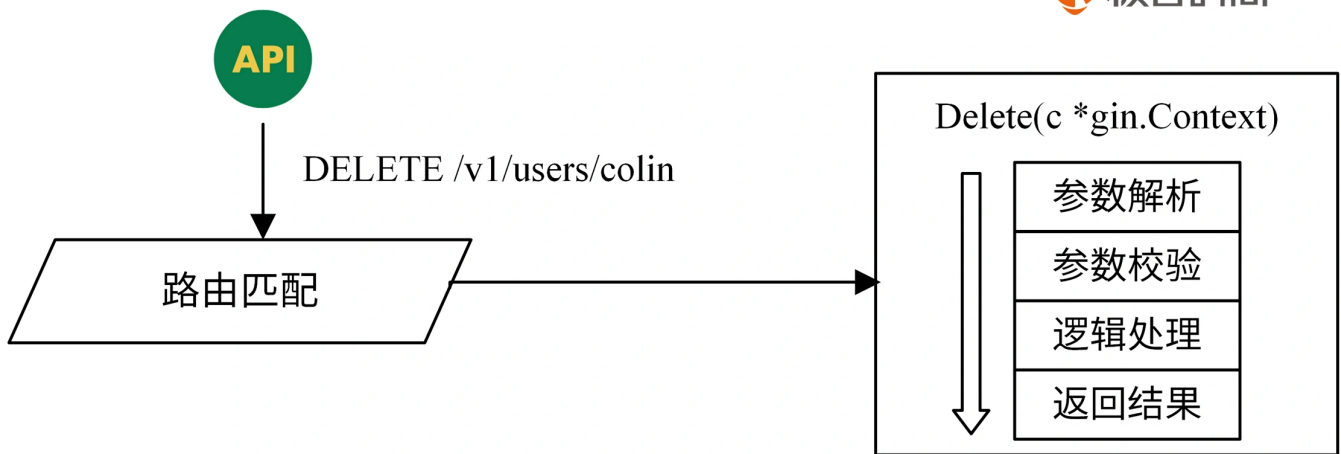
Web 服务有很多功能，为了便于你理解，我将这些功能分成了基础功能和高级功能两大类，并总结在了下面这张图中：



下面，我就按图中的顺序，来串讲下这些功能。

要实现一个 Web 服务，首先我们要选择通信协议和通信格式。在 Go 项目开发中，有 HTTP+JSON 和 gRPC+Protobuf 两种组合可选。因为 iam-apiserver 主要提供的是 REST 风格的 API 接口，所以选择的是 HTTP+JSON 组合。

Web 服务最核心的功能是路由匹配。路由匹配其实就是根据(HTTP方法，请求路径)匹配到处理这个请求的函数，最终由该函数处理这次请求，并返回结果，过程如下图所示：



一次 HTTP 请求经过路由匹配，最终将请求交由 `Delete(c *gin.Context)` 函数来处理。变量 `c` 中存放了这次请求的参数，在 `Delete` 函数中，我们可以进行参数解析、参数校验、逻辑处理，最终返回结果。

对于大型系统，可能会有很多个 API 接口，API 接口随着需求的更新迭代，可能会有多个版本，为了便于管理，我们需要**对路由进行分组**。

有时候，我们需要在一个服务进程中，同时开启 HTTP 服务的 80 端口和 HTTPS 的 443 端口，这样我们就可以做到：对内的服务，访问 80 端口，简化服务访问复杂度；对外的服务，访问更为安全的 HTTPS 服务。显然，我们没必要为相同功能启动多个服务进程，所以这时候就需要 Web 服务能够支持**一进程多服务**的功能。

我们开发 Web 服务最核心的诉求是：输入一些参数，校验通过后，进行业务逻辑处理，然后返回结果。所以 Web 服务还应该能够进行**参数解析、参数校验、逻辑处理、返回结果**。这些都是 Web 服务的业务处理功能。

上面这些是 Web 服务的基本功能，此外，我们还需要支持一些高级功能。

在进行 HTTP 请求时，经常需要针对每一次请求都设置一些通用的操作，比如添加 Header、添加 RequestID、统计请求次数等，这就要求我们的 Web 服务能够支持**中间件**特性。

为了保证系统安全，对于每一个请求，我们都需要进行**认证**。Web 服务中，通常有两种认证方式，一种是基于用户名和密码，一种是基于 Token。认证通过之后，就可以继续处理

请求了。

为了方便定位和跟踪某一次请求，需要支持 **RequestID**，定位和跟踪 RequestID 主要是为了排障。

最后，当前的软件架构中，很多采用了前后端分离的架构。在前后端分离的架构中，前端访问地址和后端访问地址往往是不同的，浏览器为了安全，会针对这种情况设置跨域请求，所以 Web 服务需要能够处理浏览器的**跨域**请求。

到这里，我就把 Web 服务的基础功能和高级功能串讲了一遍。当然，上面只介绍了 Web 服务的核心功能，还有很多其他的功能，你可以通过学习 [Gin 的官方文档](#) 来了解。

你可以看到，Web 服务有很多核心功能，这些功能我们可以基于 net/http 包自己封装。但在实际的项目开发中，我们更多会选择使用基于 net/http 包进行封装的优秀开源 Web 框架。本实战项目选择了 Gin 框架。

接下来，我们主要看下 Gin 框架是如何实现以上核心功能的，这些功能我们在实际的开发中可以直接拿来使用。

为什么选择 Gin 框架？

优秀的 Web 框架有很多，我们为什么要选择 Gin 呢？在回答这个问题之前，我们先来看下选择 Web 框架时的关注点。

在选择 Web 框架时，我们可以关注如下几点：

路由功能；

是否具备 middleware/filter 能力；

HTTP 参数 (path、query、form、header、body) 解析和返回；

性能和稳定性；

使用复杂度；

社区活跃度。

按 GitHub Star 数来排名，当前比较火的 Go Web 框架有 Gin、Beego、Echo、Revel、Martini。经过调研，我从中选择了 Gin 框架，原因是 Gin 具有如下特性：

轻量级，代码质量高，性能比较高；

项目目前很活跃，并有很多可用的 Middleware；

作为一个 Web 框架，功能齐全，使用起来简单。

那接下来，我就先详细介绍下 Gin 框架。

🔗Gin是用 Go 语言编写的 Web 框架，功能完善，使用简单，性能很高。Gin 核心的路由功能是通过一个定制版的 🔗HttpRouter来实现的，具有很高的路由性能。

Gin 有很多功能，这里我给你列出了它的一些核心功能：

支持 HTTP 方法：GET、POST、PUT、PATCH、DELETE、OPTIONS。

支持不同位置的 HTTP 参数：路径参数（path）、查询字符串参数（query）、表单参数（form）、HTTP 头参数（header）、消息体参数（body）。

支持 HTTP 路由和路由分组。

支持 middleware 和自定义 middleware。

支持自定义 Log。

支持 binding 和 validation，支持自定义 validator。可以 bind 如下参数：query、path、body、header、form。

支持重定向。

支持 basic auth middleware。

支持自定义 HTTP 配置。

支持优雅关闭。

支持 HTTP2。


支持设置和获取 cookie。

Gin 是如何支持 Web 服务基础功能的？

接下来, 我们先通过一个具体的例子, 看下 Gin 是如何支持 Web 服务基础功能的, 后面再详细介绍这些功能的用法。

我们创建一个 `allinone` 目录, 用来存放示例代码。因为要演示 HTTPS 的用法, 所以需要创建证书文件。具体可以分为两步。

第一步, 执行以下命令创建证书:

 复制代码

```
1 cat << 'EOF' > ca.pem
2 -----BEGIN CERTIFICATE-----
3 MIICSjCCAb0gAwIBAgIJAJHGGR4dG1oHMA0GCSqGSIb3DQEBCwUAMFYxCzAJBgNV
4 BAYTAKFVMRMwEQYDVQKIEWpTb21lLVN0YXRlMSEwHwYDVQKEXhJbnRlcmlldCBX
5 aWRnaXRzIFB0eSBMdGQxDzANBgNVBAMTBnRlc3RjYTAeFw0xNDExMTEyMjMxMjla
6 Fw0yNDExMDgyMjMxMjlaMFYxCzAJBgNVBAYTAKFVMRMwEQYDVQKIEWpTb21lLVN0
7 YXRlMSEwHwYDVQKEXhJbnRlcmlldCBXaWRnaXRzIFB0eSBMdGQxDzANBgNVBAMT
8 BnRlc3RjYTCBnzANBgkqhkiG9w0BAQEFAA0BjQAwYkCgYEAwEDfBV5MYdlHVHJ7
9 +L4nrxZy7mBfAVXpOc5vMYztssUI7mL2/iYujiIXM+weZYNTepLdjyJdu7R5gGUu
10 g1jSVK/EPHfc7407AyZU34PNIP4Sh33N+/A5YexrNgJlPY+E3GdVYi4ldWJjgkAd
11 Qah2PH5ACLRiIC6tRka9hcaBlIECAwEAAAMgMB4wDAYDVR0TBAAUwAwEB/zA0BgNV
12 HQ8BAf8EBAMCAgQwDQYJKoZIhvcNAQELBQADgYEAHzC7jdYlZAVmddi/gdAeKPau
13 sPBG/C2HCWqHzpCUHcKuvMzDVkY/MP2o6JIW2DBbY64b0/FceExhjcykgaYtCH/m
14 oIU63+CF0TtR7otyQAWHqXa7q4SbCDlG7DyRFxqG0txPtGvy12lgldA2+RgcigQG
15 Dfcog5wrJytaQ6UA0wE=
16 -----END CERTIFICATE-----
17 EOF
18
19 cat << 'EOF' > server.key
20 -----BEGIN PRIVATE KEY-----
21 MIICdQIBADANBgkqhkiG9w0BAQEFAASCA18wggJbAgEAAoGBA0HDFScoLCVJpYDD
22 M4HYtIdV6Ake/sMNaakdODjDMsux/4tDydLumN+fm+AjPEK5GHhGn1BgzkWF+sLf
23 3BxhRA/8dNsnunstVA7ZBgA/5qQxMfGAq4wHNVX77fBZ0gp9VLSMVfyd9N8YwbBY
24 Ack0eUQadTi2X1S60gJXgQ0m3MWhAgMBAAECgYAn7qGnM2vbJJBm0VZCk0kTIWm
25 V10okw7EPJrdL2mkre9NasghNXbE1y5zDshx5Nt3KsazK0xTT8d0Jwh/3KbaN+YY
26 tTCbKGW0pXDRBhwUHRcuRzScjli8Rih5U0CiZkhefUTcRb6xIhZJuQy71tjaSy0p
27 dHZRmYyBY02YEQ8xoQJBAPrJPhMBkzmEYFtyIEqAxQ/o/A6E+E4w8i+KM7nQCK7q
28 K4JXzyXVAjLfyBZWHGM2uro/fjqPggGD6QH1qXCKI4MCQQDmdKeb2TrKRh5BY1LR
29 81aJGKcJ2XbcDu6wMZK4oqWbTX2KiYn9GB0woM6nSr/Y6iy1u145YzYxEV/iMwff
30 DJULAKB8B2Mnyz0g0pNFJqBJuH29bKCCa8gHJzqXhN05lAlEbMK95p/P2Wi+4Hd
31 aiEIAF1BF326QJcvYKmwSmrORp85AkAlSNxRJ500WrfmZnBgZVjDx3xG6KsFQV2
32 ol6Vhql6dFgKUORFUWbvnKSyhjJxurLPEahV6oo6+A+mPhFY8eUvAkAZQyTdupP3
33 XEFQKctGz+9+gKkemDp7LBBMEMBXRgtLPhPEfcjv/7KPdnFHYmhYeBTBnuVmTVWe
34 F98XJ7tIFfJq
35 -----END PRIVATE KEY-----
36 EOF
37
38 cat << 'EOF' > server.pem
```




```

39 -----BEGIN CERTIFICATE-----
40 MIIcNDCCAgWgAwIBAgIBBzANBgkqhkiG9w0BAQsFADBWMQswCQYDVQQGEwJBVTET
41 MBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV2lkZ2l0cyBQ
42 dHkgTHRkMQ8wDQYDVQQDEwZ0ZXN0Y2EwHhcNMTUxMTA0MDIyMDI0WhcNMjUxMTAx
43 MDIyMDI0WjBlMQswCQYDVQQGEwJVUzERMA8GA1UECBMISWxsaw5vaXMxEDA0BgNV
44 BAQTB0NoaWNhZ28xFTATBgNVBAoTDEV4YW1wbGUzIENvLjEaMBGGA1UEAxQRKj50
45 ZXN0Lmdvb2dsZS5jb20wgZ8wDQYJYKozIhvcNAQEBBQADgY0AMIGJAoGBA0HDFSco
46 LCVJpYDDM4HYtIdV6Ake/sMNaaKdODjDmsux/4tDydlumN+fm+AjPEK5GHhGn1Bg
47 zkWF+sLf3BxhRA/8dNsnunstVA7ZBgA/5qQxMfGAq4wHNVX77fBZ0gp9VlSMVfyd
48 9N8YwbBYAckOeUQadTi2X1S60gJXgQ0m3MWhAgMBAAGjazBpMAkGA1UdEwQCAAw
49 CwYDVIR0PBAQDAgXgME8GA1UdEQRIMEaCECoudGVzdC5nb29nbGUuZnKCGHdhhdGVy
50 em9vaS50ZXN0Lmdvb2dsZS5iZyISKi50ZXN0LnldXR1YmUuY29thwTAQAEADMA0G
51 CSqGSIB3DQEBcWUAA4GBAJFXVifQNub1LUP4JlnX5lXNlo8FxZ2a12AFQs+bzoJ6
52 hM044EDJqyxUqSbVePK0ni3w1fHQB5rY9yYC5f8G7aqqTY1Q0hoUk8ZTSTRpnkTh
53 y4jjdvTZelDVBblueZUTDRmy2feY5aZIU18vFDK08dTG0A87pppuv1LNIR3loveU8
54 -----END CERTIFICATE-----
55 EOF

```

第二步, 创建 main.go 文件:

 复制代码

```

1 package main
2
3 import (
4     "fmt"
5     "log"
6     "net/http"
7     "sync"
8     "time"
9
10    "github.com/gin-gonic/gin"
11    "golang.org/x/sync/errgroup"
12 )
13
14 type Product struct {
15     Username    string    `json:"username" binding:"required"`
16     Name        string    `json:"name" binding:"required"`
17     Category    string    `json:"category" binding:"required"`
18     Price       int       `json:"price" binding:"gte=0"`
19     Description  string    `json:"description"`
20     CreatedAt   time.Time `json:"createdAt"`
21 }
22
23 type productHandler struct {
24     sync.RWMutex
25     products map[string]Product
26 }
27
28 func newProductHandler() *productHandler {


```

```
29     return &productHandler{
30         products: make(map[string]Product),
31     }
32 }
33
34 func (u *productHandler) Create(c *gin.Context) {
35     u.Lock()
36     defer u.Unlock()
37
38     // 1. 参数解析
39     var product Product
40     if err := c.ShouldBindJSON(&product); err != nil {
41         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
42         return
43     }
44
45     // 2. 参数校验
46     if _, ok := u.products[product.Name]; ok {
47         c.JSON(http.StatusBadRequest, gin.H{"error": fmt.Sprintf("product %s already exists", product.Name)})
48         return
49     }
50     product.CreatedAt = time.Now()
51
52     // 3. 逻辑处理
53     u.products[product.Name] = product
54     log.Printf("Register product %s success", product.Name)
55
56     // 4. 返回结果
57     c.JSON(http.StatusOK, product)
58 }
59
60 func (u *productHandler) Get(c *gin.Context) {
61     u.Lock()
62     defer u.Unlock()
63
64     product, ok := u.products[c.Param("name")]
65     if !ok {
66         c.JSON(http.StatusNotFound, gin.H{"error": fmt.Errorf("can not found product %s", c.Param("name"))})
67         return
68     }
69
70     c.JSON(http.StatusOK, product)
71 }
72
73 func router() http.Handler {
74     router := gin.Default()
75     productHandler := newProductHandler()
76     // 路由分组、中间件、认证
77     v1 := router.Group("/v1")
78     {
79         productv1 := v1.Group("/products")
80         {
```




```
81     // 路由匹配
82     productv1.POST("", productHandler.Create)
83     productv1.GET(":name", productHandler.Get)
84 }
85 }
86
87 return router
88 }
89
90 func main() {
91     var eg errgroup.Group
92
93     // 一进程多端口
94     insecureServer := &http.Server{
95         Addr:         ":8080",
96         Handler:        router(),
97         ReadTimeout:    5 * time.Second,
98         WriteTimeout:   10 * time.Second,
99     }
100
101     secureServer := &http.Server{
102         Addr:         ":8443",
103         Handler:        router(),
104         ReadTimeout:    5 * time.Second,
105         WriteTimeout:   10 * time.Second,
106     }
107
108     eg.Go(func() error {
109         err := insecureServer.ListenAndServe()
110         if err != nil && err != http.ErrServerClosed {
111             log.Fatal(err)
112         }
113         return err
114     })
115
116     eg.Go(func() error {
117         err := secureServer.ListenAndServeTLS("server.pem", "server.key")
118         if err != nil && err != http.ErrServerClosed {
119             log.Fatal(err)
120         }
121         return err
122     })
123
124     if err := eg.Wait(); err != nil {
125         log.Fatal(err)
126     }
127 }
```

运行以上代码：

 复制代码

```
1 $ go run main.go
```

打开另外一个终端, 请求 HTTP 接口:

 复制代码

```
1 # 创建产品
2 $ curl -XPOST -H"Content-Type: application/json" -d'{"username":"colin","name"
3 {"username":"colin","name":"iphone12","category":"phone","price":8000,"descrip
4
5 # 获取产品信息
6 $ curl -XGET http://127.0.0.1:8080/v1/products/iphone12
7 {"username":"colin","name":"iphone12","category":"phone","price":8000,"descrip
```


示例代码存放地址为 [@webfeature](#)。

另外, Gin 项目仓库中也包含了很多使用示例, 如果你想详细了解, 可以参考 [@gin examples](#)。

下面, 我来详细介绍下 Gin 是如何支持 Web 服务基础功能的。

HTTP/HTTPS 支持

因为 Gin 是基于 net/http 包封装的一个 Web 框架, 所以它天然就支持 HTTP/HTTPS。在上述代码中, 通过以下方式开启一个 HTTP 服务:

 复制代码

```
1 insecureServer := &http.Server{
2     Addr:         ":8080",
3     Handler:      router(),
4     ReadTimeout:  5 * time.Second,
5     WriteTimeout: 10 * time.Second,
6 }
7 ...
8 err := insecureServer.ListenAndServe()
```

通过以下方式开启一个 HTTPS 服务:

```
1 secureServer := &http.Server{
2     Addr:         ":8443",
3     Handler:      router(),
4     ReadTimeout:  5 * time.Second,
5     WriteTimeout: 10 * time.Second,
6 }
7 ...
8 err := secureServer.ListenAndServeTLS("server.pem", "server.key")
```

JSON 数据格式支持

Gin 支持多种数据通信格式, 例如 application/json、application/xml。可以通过 `c.ShouldBindJSON` 函数, 将 Body 中的 JSON 格式数据解析到指定的 Struct 中, 通过 `c.JSON` 函数返回 JSON 格式的数据。

路由匹配

Gin 支持两种路由匹配规则。

第一种匹配规则是精确匹配。例如, 路由为 `/products/:name`, 匹配情况如下表所示:


路径	匹配情况
/products/iphone12	匹配
/products/xiaomi8	匹配
/products/xiaomi8/music	不匹配
/products/	不匹配

第二种匹配规则是模糊匹配。例如, 路由为 `/products/*name`, 匹配情况如下表所示:

路径	匹配情况
<code>/products/iphone12</code>	匹配
<code>/products/xiaomi8</code>	匹配
<code>/products/xiaomi8/music</code>	匹配
<code>/products/</code>	匹配

路由分组

Gin 通过 `Group` 函数实现了路由分组的功能。路由分组是一个非常常用的功能, 可以将相同版本的路由分为一组, 也可以将相同 RESTful 资源的路由分为一组。例如:

 复制代码

```
1 v1 := router.Group("/v1", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "c
2 {
3     productv1 := v1.Group("/products")
4     {
5         // 路由匹配
6         productv1.POST("", productHandler.Create)
7         productv1.GET(":name", productHandler.Get)
8     }
9
10    orderv1 := v1.Group("/orders")
11    {
12        // 路由匹配
13        orderv1.POST("", orderHandler.Create)
14        orderv1.GET(":name", orderHandler.Get)
```

```
15     }
16 }
17
18 v2 := router.Group("/v2", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "c
19 {
20     productv2 := v2.Group("/products")
21     {
22         // 路由匹配
23         productv2.POST("", productHandler.Create)
24         productv2.GET(":name", productHandler.Get)
25     }
26 }
```

通过将路由分组, 可以对相同分组的路由做统一处理。比如上面那个例子, 我们可以通过代码

[复制代码](#)

```
1 v1 := router.Group("/v1", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "c
```

给所有属于 v1 分组的路由都添加 gin.BasicAuth 中间件, 以实现认证功能。中间件和认证, 这里你先不用深究, 下面讲高级功能的时候会介绍到。

一进程多服务

我们可以通过以下方式实现一进程多服务:

[复制代码](#)

```
1 var eg errgroup.Group
2 insecureServer := &http.Server{...}
3 secureServer := &http.Server{...}
4
5 eg.Go(func() error {
6     err := insecureServer.ListenAndServe()
7     if err != nil && err != http.ErrServerClosed {
8         log.Fatal(err)
9     }
10    return err
11 })
12 eg.Go(func() error {
13     err := secureServer.ListenAndServeTLS("server.pem", "server.key")
14     if err != nil && err != http.ErrServerClosed {
15         log.Fatal(err)
16     }
17    return err
```

```
18 }
19
20 if err := eg.Wait(); err != nil {
21     log.Fatal(err)
22 }
```

上述代码实现了两个相同的服务, 分别监听在不同的端口。这里需要注意的是, 为了不阻塞启动第二个服务, 我们需要把 ListenAndServe 函数放在 goroutine 中执行, 并且调用 eg.Wait() 来阻塞程序进程, 从而让两个 HTTP 服务在 goroutine 中持续监听端口, 并提供服务。

参数解析、参数校验、逻辑处理、返回结果

此外, Web 服务还应该具有参数解析、参数校验、逻辑处理、返回结果 4 类功能, 因为这些功能联系紧密, 我们放在一起来说。

在 productHandler 的 Create 方法中, 我们通过 c.ShouldBindJSON 来解析参数, 接下来自己编写校验代码, 然后将 product 信息保存在内存中 (也就是业务逻辑处理), 最后通过 c.JSON 返回创建的 product 信息。代码如下:

[复制代码](#)

```
1 func (u *productHandler) Create(c *gin.Context) {
2     u.Lock()
3     defer u.Unlock()
4
5     // 1. 参数解析
6     var product Product
7     if err := c.ShouldBindJSON(&product); err != nil {
8         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
9         return
10    }
11
12    // 2. 参数校验
13    if _, ok := u.products[product.Name]; ok {
14        c.JSON(http.StatusBadRequest, gin.H{"error": fmt.Sprintf("product %s already exists", product.Name)})
15        return
16    }
17    product.CreatedAt = time.Now()
18
19    // 3. 逻辑处理
20    u.products[product.Name] = product
21    log.Printf("Register product %s success", product.Name)
22 }
```



```
23 // 4. 返回结果
24 c.JSON(http.StatusOK, product)
25 }
```

那这个时候, 你可能会问: HTTP 的请求参数可以存在不同的位置, Gin 是如何解析的呢? 这里, 我们先来看下 HTTP 有哪些参数类型。HTTP 具有以下 5 种参数类型:

路径参数 (path)。例如 `gin.Default().GET("/user/:name", nil)`, `name` 就是路径参数。

查询字符串参数 (query)。例如 `/welcome?`

`firstname=Lingfei&lastname=Kong`, `firstname` 和 `lastname` 就是查询字符串参数。

表单参数 (form)。例如 `curl -X POST -F 'username=colin' -F 'password=colin1234' http://mydomain.com/login`, `username` 和 `password` 就是表单参数。

HTTP 头参数 (header)。例如 `curl -X POST -H 'Content-Type: application/json' -d '{"username":"colin","password":"colin1234"}' http://mydomain.com/login`, `Content-Type` 就是 HTTP 头参数。


消息体参数 (body)。例如 `curl -X POST -H 'Content-Type: application/json' -d '{"username":"colin","password":"colin1234"}' http://mydomain.com/login`, `username` 和 `password` 就是消息体参数。

Gin 提供了一些函数, 来分别读取这些 HTTP 参数, 每种类别会提供两种函数, 一种函数可以直接读取某个参数的值, 另外一种函数会把同类 HTTP 参数绑定到一个 Go 结构体中。比如, 有如下路径参数:

```
1 gin.Default().GET("/:name/:id", nil)
```


[复制代码](#)

我们可以直接读取每个参数:

 复制代码

```
1 name := c.Param("name")
2 action := c.Param("action")
```

也可以将所有的路径参数, 绑定到结构体中:

 复制代码

```
1 type Person struct {
2     ID string `uri:"id" binding:"required,uuid"`
3     Name string `uri:"name" binding:"required"`
4 }
5
6 if err := c.ShouldBindUri(&person); err != nil {
7     // normal code
8     return
9 }
```

Gin 在绑定参数时, 是通过结构体的 tag 来判断要绑定哪类参数到结构体中的。这里要注意, 不同的 HTTP 参数有不同的结构体 tag。

路径参数: uri。

查询字符串参数: form。

表单参数: form。

HTTP 头参数: header。

消息体参数: 会根据 Content-Type, 自动选择使用 json 或者 xml, 也可以调用 ShouldBindJSON 或者 ShouldBindXML 直接指定使用哪个 tag。

针对每种参数类型, Gin 都有对应的函数来获取和绑定这些参数。这些函数都是基于如下两个函数进行封装的:

1. ShouldBindWith(obj interface{}, b binding.Binding) error

非常重要的一个函数, 很多 ShouldBindXXX 函数底层都是调用 ShouldBindWith 函数来完成参数绑定的。该函数会根据传入的绑定引擎, 将参数绑定到传入的结构体指针中, **如果绑定失败, 只返回错误内容, 但不终止 HTTP 请求**。ShouldBindWith 支持多种绑定引

擎，例如 `binding.JSON`、`binding.Query`、`binding.Uri`、`binding.Header` 等，更详细的信息你可以参考 binding.go。

2. `MustBindWith(obj interface{}, b binding.Binding)` error

这是另一个非常重要的函数，很多 `BindXXX` 函数底层都是调用 `MustBindWith` 函数来完成参数绑定的。该函数会根据传入的绑定引擎，将参数绑定到传入的结构体指针中，**如果绑定失败，返回错误并终止请求，返回 HTTP 400 错误**。`MustBindWith` 所支持的绑定引擎跟 `ShouldBindWith` 函数一样。

Gin 基于 `ShouldBindWith` 和 `MustBindWith` 这两个函数，又衍生出很多新的 `Bind` 函数。这些函数可以满足不同场景下获取 HTTP 参数的需求。Gin 提供的函数可以获取 5 个类别的 HTTP 参数。

路径参数：`ShouldBindUri`、`BindUri`；

查询字符串参数：`ShouldBindQuery`、`BindQuery`；

表单参数：`ShouldBind`；

HTTP 头参数：`ShouldBindHeader`、`BindHeader`；

消息体参数：`ShouldBindJSON`、`BindJSON` 等。

每个类别的 `Bind` 函数，详细信息你可以参考 [Gin 提供的 Bind 函数](#)。

这里要注意，Gin 并没有提供类似 `ShouldBindForm`、`BindForm` 这类函数来绑定表单参数，但我们可以通过 `ShouldBind` 来绑定表单参数。当 HTTP 方法为 GET 时，`ShouldBind` 只绑定 Query 类型的参数；当 HTTP 方法为 POST 时，会先检查 `content-type` 是否是 json 或者 xml，如果不是，则绑定 Form 类型的参数。

所以，`ShouldBind` 可以绑定 Form 类型的参数，但前提是 HTTP 方法是 POST，并且 `content-type` 不是 `application/json`、`application/xml`。

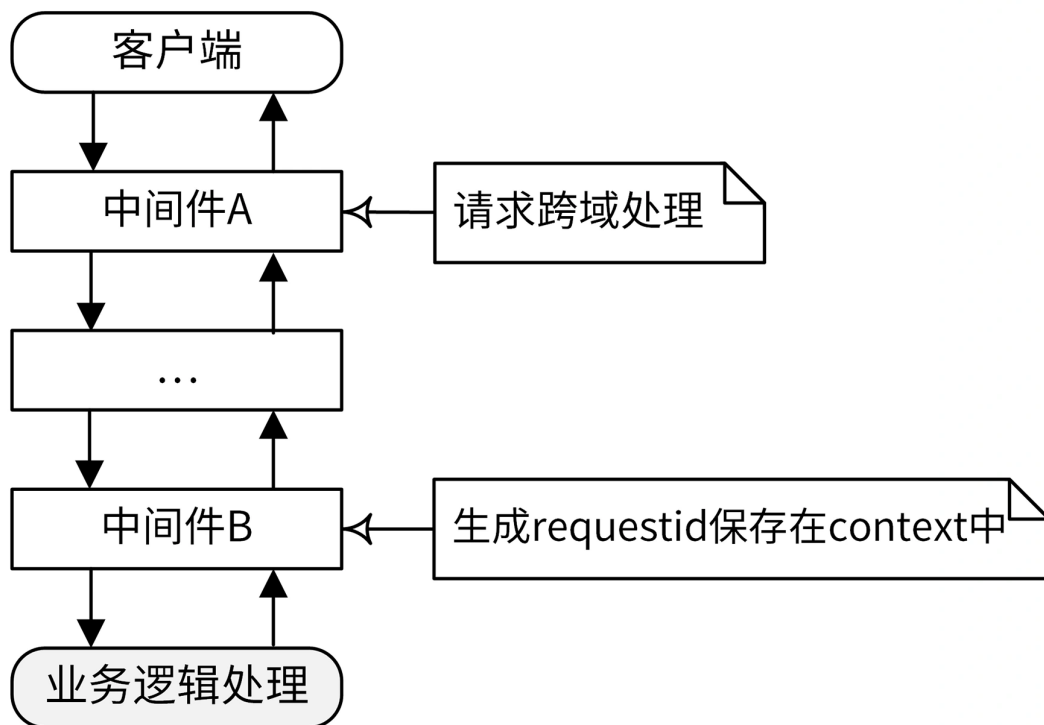
在 Go 项目开发中，我建议使用 `ShouldBindXXX`，这样可以确保我们设置的 HTTP Chain（Chain 可以理解为一个 HTTP 请求的一系列处理插件）能够继续被执行。

Gin 是如何支持 Web 服务高级功能的？

上面介绍了 Web 服务的基础功能，这里我再来介绍下高级功能。Web 服务可以具备多个高级功能，但比较核心的高级功能是中间件、认证、RequestID、跨域和优雅关停。

中间件

Gin 支持中间件，HTTP 请求在转发到实际的处理函数之前，会被一系列加载的中间件进行处理。在中间件中，可以解析 HTTP 请求做一些逻辑处理，例如：跨域处理或者生成 X-Request-ID 并保存在 context 中，以便追踪某个请求。处理完之后，可以选择中断并返回这次请求，也可以选择将请求继续转交给下一个中间件处理。当所有的中间件都处理完之后，请求才会转给路由函数进行处理。具体流程如下图：



通过中间件，可以实现对所有请求都做统一的处理，提高开发效率，并使我们的代码更简洁。但是，因为所有的请求都需要经过中间件的处理，可能会增加请求延时。对于中间件特性，我有如下建议：

中间件做成可加载的，通过配置文件指定程序启动时加载哪些中间件。

只将一些通用的、必要的功能做成中间件。

在编写中间件时，一定要保证中间件的代码质量和性能。

在 Gin 中，可以通过 gin.Engine 的 Use 方法来加载中间件。中间件可以加载到不同的位置上，而且不同的位置作用范围也不同，例如：

[复制代码](#)

```
1 router := gin.New()
2 router.Use(gin.Logger(), gin.Recovery()) // 中间件作用于所有的HTTP请求
3 v1 := router.Group("/v1").Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin":
4 v1.POST("/login", Login).Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin":
```

Gin 框架本身支持了一些中间件。

gin.Logger()：Logger 中间件会将日志写到 gin.DefaultWriter，gin.DefaultWriter 默认为 os.Stdout。

gin.Recovery()：Recovery 中间件可以从任何 panic 恢复，并且写入一个 500 状态码。

gin.CustomRecovery(handle gin.RecoveryFunc)：类似 Recovery 中间件，但是在恢复时还会调用传入的 handle 方法进行处理。

gin.BasicAuth()：HTTP 请求基本认证（使用用户名和密码进行认证）。

另外，Gin 还支持自定义中间件。中间件其实是一个函数，函数类型为 gin.HandlerFunc，HandlerFunc 底层类型为 func(*Context)。如下是一个 Logger 中间件的实现：

[复制代码](#)

```
1 package main
2
3 import (
4     "log"
5     "time"
6
7     "github.com/gin-gonic/gin"
8 )
9
10 func Logger() gin.HandlerFunc {
```

```
11  return func(c *gin.Context) {
12      t := time.Now()
13
14      // 设置变量example
15      c.Set("example", "12345")
16
17      // 请求之前
18
19      c.Next()
20
21      // 请求之后
22      latency := time.Since(t)
23      log.Print(latency)
24
25      // 访问我们发送的状态
26      status := c.Writer.Status()
27      log.Println(status)
28  }
29  }
30
31  func main() {
32      r := gin.New()
33      r.Use(Logger())
34
35      r.GET("/test", func(c *gin.Context) {
36          example := c.MustGet("example").(string)
37
38          // it would print: "12345"
39          log.Println(example)
40      })
41
42      // Listen and serve on 0.0.0.0:8080
43      r.Run(":8080")
44  }
```

另外, 还有很多开源的中间件可供我们选择, 我把一些常用的总结在了表格里:

中间件	功能
gin-jwt	JWT中间件，实现JWT认证
gin-swagger	自动生成Swagger 2.0格式的RESTful API文档
cors	实现HTTP请求跨域
sessions	会话管理中间件
authz	基于casbin的授权中间件
pprof	gin pprof中间件
go-gin-prometheus	Prometheus metrics exporter
gzip	支持HTTP请求和响应的gzip压缩
gin-limit	HTTP请求并发控制中间件
requestid	给每个Request生成uuid，并添加在返回的X-Request-ID Header中

认证、RequestID、跨域

认证、RequestID、跨域这三个高级功能，都可以通过 Gin 的中间件来实现，例如：

复制代码

```
1 router := gin.New()
2
3 // 认证
4 router.Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"}))
5
6 // RequestID
7 router.Use(requestid.New(requestid.Config{
8     Generator: func() string {
9         return "test"
10    },
11 })))
12
13 // 跨域
14 // CORS for https://foo.com and https://github.com origins, allowing:
15 // - PUT and PATCH methods
```

```
16 // - Origin header
17 // - Credentials share
18 // - Preflight requests cached for 12 hours
19 router.Use(cors.New(cors.Config{
20     AllowOrigins:    []string{"https://foo.com"},
21     AllowMethods:    []string{"PUT", "PATCH"},
22     AllowHeaders:    []string{"Origin"},
23     ExposeHeaders:   []string{"Content-Length"},
24     AllowCredentials: true,
25     AllowOriginFunc: func(origin string) bool {
26         return origin == "https://github.com"
27     },
28     MaxAge: 12 * time.Hour,
29 })})
```

优雅关停

Go 项目上线后，我们还需要不断迭代来丰富项目功能、修复 Bug 等，这也就意味着，我们要不断地重启 Go 服务。对于 HTTP 服务来说，如果访问量大，重启服务的时候可能还有很多连接没有断开，请求没有完成。如果这时候直接关闭服务，这些连接会直接断掉，请求异常终止，这就会对用户体验和产品口碑造成很大影响。因此，这种关闭方式不是一种优雅的关闭方式。

这时候，我们期望 HTTP 服务可以在处理完所有请求后，正常地关闭这些连接，也就是优雅地关闭服务。我们有两种方法来优雅关闭 HTTP 服务，分别是借助第三方的 Go 包和自己编码实现。

方法一：借助第三方的 Go 包


如果使用第三方的 Go 包来实现优雅关闭，目前用得比较多的包是 [fvbck/endless](#)。我们可以使用 `fvbck/endless` 来替换掉 `net/http` 的 `ListenAndServe` 方法，例如：

 复制代码

```
1 router := gin.Default()
2 router.GET("/", handler)
3 // [...]
4 endless.ListenAndServe(":4242", router)
```

方法二：编码实现

借助第三方包的好处是可以稍微减少一些编码工作量, 但缺点是引入了一个新的依赖包, 因此我更倾向于自己编码实现。Go 1.8 版本或者更新的版本, `http.Server` 内置的 `Shutdown` 方法, 已经实现了优雅关闭。下面是一个示例:

 复制代码

```
1 // +build go1.8
2
3 package main
4
5 import (
6     "context"
7     "log"
8     "net/http"
9     "os"
10    "os/signal"
11    "syscall"
12    "time"
13
14    "github.com/gin-gonic/gin"
15 )
16
17 func main() {
18     router := gin.Default()
19     router.GET("/", func(c *gin.Context) {
20         time.Sleep(5 * time.Second)
21         c.String(http.StatusOK, "Welcome Gin Server")
22     })
23
24     srv := &http.Server{
25         Addr:    ":8080",
26         Handler: router,
27     }
28
29     // Initializing the server in a goroutine so that
30     // it won't block the graceful shutdown handling below
31     go func() {
32         if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
33             log.Fatalf("listen: %s\n", err)
34         }
35     }()
36
37     // Wait for interrupt signal to gracefully shutdown the server with
38     // a timeout of 5 seconds.
39     quit := make(chan os.Signal)
40     // kill (no param) default send syscall.SIGTERM
41     // kill -2 is syscall.SIGINT
42     // kill -9 is syscall.SIGKILL but can't be catch, so don't need add it
43     signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
44     <-quit
```

```
45     log.Println("Shutting down server...")
46
47     // The context is used to inform the server it has 5 seconds to finish
48     // the request it is currently handling
49     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
50     defer cancel()
51     if err := srv.Shutdown(ctx); err != nil {
52         log.Fatal("Server forced to shutdown:", err)
53     }
54
55     log.Println("Server exiting")
56 }
```

上面的示例中, 需要把 `srv.ListenAndServe` 放在 goroutine 中执行, 这样才不会阻塞到 `srv.Shutdown` 函数。因为我们把 `srv.ListenAndServe` 放在了 goroutine 中, 所以需要一种可以让整个进程常驻的机制。

这里, 我们借助了无缓冲 channel, 并且调用 `signal.Notify` 函数将该 channel 绑定到 `SIGINT`、`SIGTERM` 信号上。这样, 收到 `SIGINT`、`SIGTERM` 信号后, `quilt` 通道会被写入值, 从而结束阻塞状态, 程序继续运行, 执行 `srv.Shutdown(ctx)`, 优雅关停 HTTP 服务。

总结

今天我们主要学习了 Web 服务的核心功能, 以及如何开发这些功能。在实际的项目开发中, 我们一般会使用基于 `net/http` 包进行封装的优秀开源 Web 框架。

当前比较火的 Go Web 框架有 `Gin`、`Beego`、`Echo`、`Revel`、`Martini`。你可以根据需要进行选择。我比较推荐 `Gin`, `Gin` 也是目前比较受欢迎的 Web 框架。`Gin` Web 框架支持 Web 服务的很多基础功能, 例如 `HTTP/HTTPS`、`JSON` 格式的数据、路由分组和匹配、一进程多服务等。

另外, `Gin` 还支持 Web 服务的一些高级功能, 例如 中间件、认证、`RequestID`、跨域和优雅关停等。

课后练习

1. 使用 `Gin` 框架编写一个简单的 Web 服务, 要求该 Web 服务可以解析参数、校验参数, 并进行一些简单的业务逻辑处理, 最终返回处理结果。欢迎在留言区分享你的成

果, 或者遇到的问题。

2. 思考下, 如何给 iam-apiserver 的 /healthz 接口添加一个限流中间件, 用来限制请求 /healthz 的频率。

欢迎你在留言区与我交流讨论, 我们下一讲见。

分享给需要的人, Ta订阅后你可得 **24 元现金奖励**

👍 赞 0 💡 提建议

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 应用构建实战: 如何构建一个优秀的企业应用框架?

下一篇 特别放送 | 给你一份清晰、可直接套用的Go编码规范

更多课程推荐

容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师
云平台架构师



涨价倒计时 🕒

今日订阅 **¥69**, 7月20日涨价至 **¥129**

精选留言 (1)

写留言



huntersudo

2021-07-20

Gin的示例和代码看了很多，知道这样写，有时候就不知道为啥这样写，老师的文章不少地方给了解释，给力给力！！

