

24-Web服务：Web服务核心功能有哪些，如何实现？

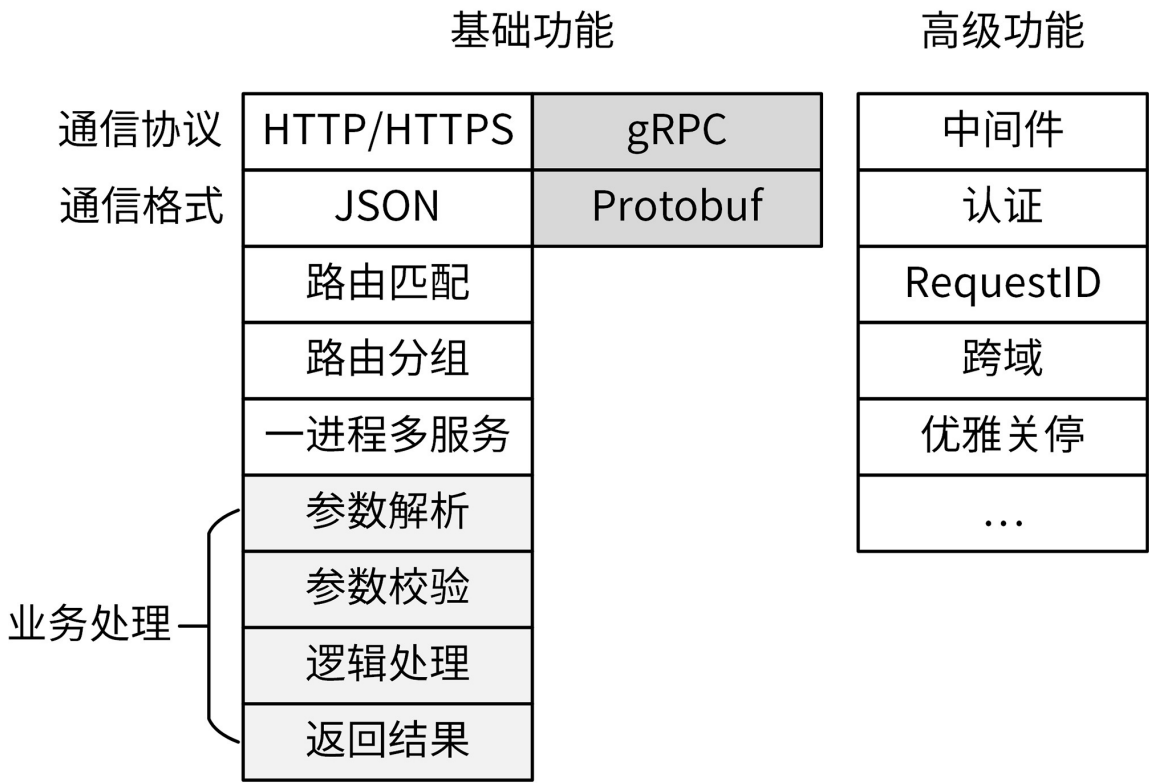
你好，我是孔令飞。从今天开始，我们进入实战第三站：服务开发。在这个部分，我会讲解 IAM项目各个服务的构建方式，帮助你掌握Go 开发阶段的各个技能点。

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

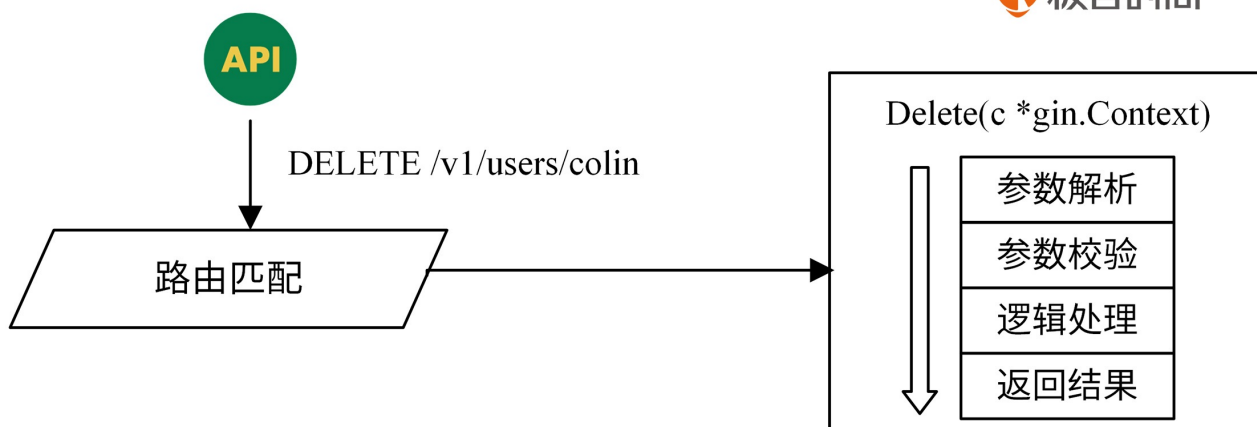
Web服务有很多功能，为了便于你理解，我将这些功能分成了基础功能和高级功能两大类，并总结在了下面这张图中：



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：

```
cat << 'EOF' > ca.pem
-----BEGIN CERTIFICATE-----
MIICSjCCAbOgAwIBAgIJAJHGGR4dG6ioHMA0GCSqGSIb3DQEBCwUAMFYxCzAJBgNV
BAYTAkFVMRMwEQYDVQIEwPTb211LVN0YXR1MSEwHwYDVQQKEWhJbnRlcm5ldCBX
aWRnaXRzIFB0eSBMdGQxDzANBgNVBAMTBnRlc3RjYTAeFw0xNDExMTEyMjMxMjla
Fw0yNDExMDgyMjMxMjlaMFYxCzAJBgNVBAYTAkFVMRMwEQYDVQIEwPTb211LVN0
YXR1MSEwHwYDVQQKEWhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQxDzANBgNVBAMT
BnRlc3RjYTCBnzANBgkqhkiG9w0BAQEFAA0BjQAwYkCgYEAwEDfBV5MYd1HVHJ7
+L4nrxZy7mBfAVXp0c5vMYztssUI7mL2/iYujiiIXM+weZYNTepLdjyJdu7R5gGUu
g1jSVK/EPHfc7407AyZU34PNIP4Sh33N+/A5YexrNgJlPY+E3GdVYi4ldwJjgkAd
Qah2PH5ACLrIIC6tRka9hcaB1IECAwEAAMgMB4wDAYDVR0TBAAUwAwEB/zA0BgNV
HQ8BAf8EBAMCAgQwDQYJKoZIhvcNAQELBQADgYEAHzC7jdY1zAVmddi/gdAekPau
sPBG/C2HCWqHzpCUHcKuvMzDVkY/MP2o6JIW2DBbY64b0/FceExhjcykgaYtCH/m
oIU63+CF0TtR7otyQAWHqXa7q4SbCD1G7DyRFxqG0txPtGvy12lgldA2+RgcigQG
Dfcog5wrJytaQ6UA0wE=
-----END CERTIFICATE-----
EOF

cat << 'EOF' > server.key
-----BEGIN PRIVATE KEY-----
MIICDQIBADANBgkqhkiG9w0BAQEFAASCA18wggJbAgEAAoGBA0HDFScoLCVJpYDD
M4HYtIdV6Ake/sMNaakdODjDmsux/4tDydlumN+fm+AjpEK5GHhGn1BgzkWf+s1f
3Bxhra/8dNsnunstVA7ZBgA/5qQxMfGAq4wHNVX77fBZ0gp9VlSMVfyd9N8YwbBY
Ack0eUQadTi2X1S60gJXgQ0m3MWhAgMBAEECgYAn7qGnM2vbJJBm0VZCk0kTIWm
V10okw7EPJrdL2mkre9NasghNXbE1y5zDshx5Nt3KsazK0xTT8d0Jwh/3KbaN+YY
tTCbKGW0pXDRBhwUHRcuRzScjli8Rih5U0CiZkhefUTcRb6xIhZJuQy71tjaSy0p
dHZRmYyBY02YEQ8xoQJBAPrJPhMBkzmEYFtyIEqAxQ/o/A6E+E4w8i+KM7nQCK7q
K4JXzyXVAjLfYBZWHGM2uro/fjqPggGD6QH1qXCkI4MCQQDmdKeb2TrKRh5BY1LR
81aJGKcJ2XbcDu6wMZK4oqWbTX2KiYn9GB0woM6nSr/Y6iy1u145YzYxEV/iMwff
DJULAKB8B2Mnyz0g0pNFJqBJuH29bKCCHa8gHJzqXhN051A1EbMK95p/P2Wi+4Hd
aiEIAF1BF326QJcvYKmwSmrORp85AkA1SNxRJ500WrfMZnBgZVjDx3xG6KsFQV2
o16Vhql6dFgKUORFUWbvnKSyhjJxur1PEahV6oo6+A+mPhFY8eUvAkAZQyTdupP3
XEFQKctGz+9+gKkemDp7LBBMEMBXRGTLPhefCjv/7KPdnFHYmhYeBTBnuVmTVWe
F98XJ7tIFfJq
-----END PRIVATE KEY-----
EOF

cat << 'EOF' > server.pem
-----BEGIN CERTIFICATE-----
MIICnDCCAagWAwIBAgIBBzANBgkqhkiG9w0BAQsFADBWMQswCQYDVQQGEwJBVTET
MBEGA1UECBMkU29tZS1TdG60ZTEhMB8GA1UEChMYSw50ZXJuZXQv21kZ210cyBQ
dHkgTHRkMQ8wDQYDVQQDEwZ0ZXN0Y2EwHhcNMTEyMTA0MDIyMDI0WbcNMjUxMTAx
MDIyMDI0WjB1MQswCQYDVQQGEwJVUzERMA8GA1UECBMISWxsaw5vaXN0eDA0BgNV
```

```
BACtB0NoawNhZ28xFTATBgNVBAoTDEV4YW1wbGUuIENvLjEaMBGGA1UEAxQRKi50
ZXN0Lmdvb2dsZS5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBA0HDFSco
LCVJpYDDM4HYtIdV6Ake/sMNaaKdODjDMsux/4tDydlumN+fm+AjPEK5GHhGn1Bg
zkWf+s1f3BxhRA/8dNsnunstVA7ZBgA/5qQxMfGAq4wHNVX77fBZ0gp9VlSMVfyd
9N8YwbBYAckOeUqadTi2X1S60gJXgQ0m3MWhAgMBAAGjazBpMAKGA1UdEwQCMAAw
CwYDVROpBAQDAgXgME8GA1UdEQRIMEaCECoudGVzdC5nb29nbGUuZnKCGHdhGvVy
em9vaS50ZXN0Lmdvb2dsZS5iZyISKi50ZXN0Ln1vdXR1YmUuY29thwTAqAEDMA0G
CSqGSIb3DQEBChUAA4GBAJFXVifQNub1LUP4JlN51XN1o8FxZ2a12AFQs+bzoJ6
hM044EDjyqYUqSbVePK0ni3w1fHQB5rY9yYC5f8G7aqqTY1Q0hoUk8ZTSTRpnkTh
y4jjdvTZeLDVBblueZUTDRmy2feY5aZIU18vFDK08dTG0A87pppuv1LNIR3loveU8
-----END CERTIFICATE-----
EOF
```

第二步，创建main.go文件：

```
package main

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

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

type Product struct {
    Username    string `json:"username" binding:"required"`
    Name        string `json:"name" binding:"required"`
    Category    string `json:"category" binding:"required"`
    Price       int    `json:"price" binding:"gte=0"`
    Description  string `json:"description"`
    CreatedAt   time.Time `json:"createdAt"`
}

type productHandler struct {
    sync.RWMutex
    products map[string]Product
}

func newProductHandler() *productHandler {
    return &productHandler{
        products: make(map[string]Product),
    }
}

func (u *productHandler) Create(c *gin.Context) {
    u.Lock()
    defer u.Unlock()

    // 1. 参数解析
    var product Product
    if err := c.ShouldBindJSON(&product); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    // 2. 参数校验
```

```

if _, ok := u.products[product.Name]; ok {
    c.JSON(http.StatusBadRequest, gin.H{"error": fmt.Sprintf("product %s already exist", product.Name)})
    return
}
product.CreatedAt = time.Now()

// 3. 逻辑处理
u.products[product.Name] = product
log.Printf("Register product %s success", product.Name)

// 4. 返回结果
c.JSON(http.StatusOK, product)
}

func (u *productHandler) Get(c *gin.Context) {
    u.Lock()
    defer u.Unlock()

    product, ok := u.products[c.Param("name")]
    if !ok {
        c.JSON(http.StatusNotFound, gin.H{"error": fmt.Errorf("can not found product %s", c.Param("name"))})
        return
    }

    c.JSON(http.StatusOK, product)
}

func router() http.Handler {
    router := gin.Default()
    productHandler := newProductHandler()
    // 路由分组、中间件、认证
    v1 := router.Group("/v1")
    {
        productv1 := v1.Group("/products")
        {
            // 路由匹配
            productv1.POST("", productHandler.Create)
            productv1.GET(":name", productHandler.Get)
        }
    }

    return router
}

func main() {
    var eg errgroup.Group

    // 一进程多端口
    insecureServer := &http.Server{
        Addr:         ":8080",
        Handler:      router(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    secureServer := &http.Server{
        Addr:         ":8443",
        Handler:      router(),
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    eg.Go(func() error {
        err := insecureServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {

```

```

    log.Fatal(err)
}
return err
})

eg.Go(func() error {
    err := secureServer.ListenAndServeTLS("server.pem", "server.key")
    if err != nil && err != http.ErrServerClosed {
        log.Fatal(err)
    }
    return err
})

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

```

运行以上代码：

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

打开另外一个终端，请求HTTP接口：

```

# 创建产品
$ curl -XPOST -H"Content-Type: application/json" -d'{"username":"colin","name":"iphone12","category":"phone"
{"username":"colin","name":"iphone12","category":"phone","price":8000,"description":"cannot afford","create

# 获取产品信息
$ curl -XGET http://127.0.0.1:8080/v1/products/iphone12
{"username":"colin","name":"iphone12","category":"phone","price":8000,"description":"cannot afford","create

```

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：

```

insecureServer := &http.Server{
    Addr:         ":8080",
    Handler:      router(),
    ReadTimeout:  5 * time.Second,

```

```
WriteTimeout: 10 * time.Second,
}
...
err := insecureServer.ListenAndServe()
```

通过以下方式开启一个HTTPS服务：

```
secureServer := &http.Server{
    Addr:         ":8443",
    Handler:      router(),
    ReadTimeout:  5 * time.Second,
    WriteTimeout: 10 * time.Second,
}
...
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
/products/iphone12	匹配
/products/xiaomi8	匹配
/products/xiaomi8/music	匹配
/products/	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：

```
v1 := router.Group("/v1", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"}))
{
    productv1 := v1.Group("/products")
    {
        // 路由匹配
        productv1.POST("", productHandler.Create)
        productv1.GET(":name", productHandler.Get)
    }

    orderv1 := v1.Group("/orders")
    {
        // 路由匹配
        orderv1.POST("", orderHandler.Create)
        orderv1.GET(":name", orderHandler.Get)
    }
}

v2 := router.Group("/v2", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"}))
{
    productv2 := v2.Group("/products")
    {
        // 路由匹配
        productv2.POST("", productHandler.Create)
        productv2.GET(":name", productHandler.Get)
    }
}
```

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

```
v1 := router.Group("/v1", gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"}))
```

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：

```
var eg errgroup.Group
insecureServer := &http.Server{...}
secureServer := &http.Server{...}

eg.Go(func() error {
    err := insecureServer.ListenAndServe()
    if err != nil && err != http.ErrServerClosed {
        log.Fatal(err)
    }
    return err
})
eg.Go(func() error {
    err := secureServer.ListenAndServeTLS("server.pem", "server.key")
    if err != nil && err != http.ErrServerClosed {
        log.Fatal(err)
    }
    return err
})

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

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：

```
func (u *productHandler) Create(c *gin.Context) {
    u.Lock()
    defer u.Unlock()

    // 1. 参数解析
    var product Product
    if err := c.ShouldBindJSON(&product); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    // 2. 参数校验
    if _, ok := u.products[product.Name]; ok {
        c.JSON(http.StatusBadRequest, gin.H{"error": fmt.Sprintf("product %s already exist", product.Name)})
        return
    }
    product.CreatedAt = time.Now()

    // 3. 逻辑处理
    u.products[product.Name] = product
    log.Printf("Register product %s success", product.Name)

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

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，我们先来看一下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：

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

我们可以直接读取每个参数：

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

也可以将所有的路径参数，绑定到结构体中：

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

if err := c.ShouldBindUri(&person); err != nil {
    // normal code
    return
}
```

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](https://github.com/gin-gonic/gin/blob/master/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。

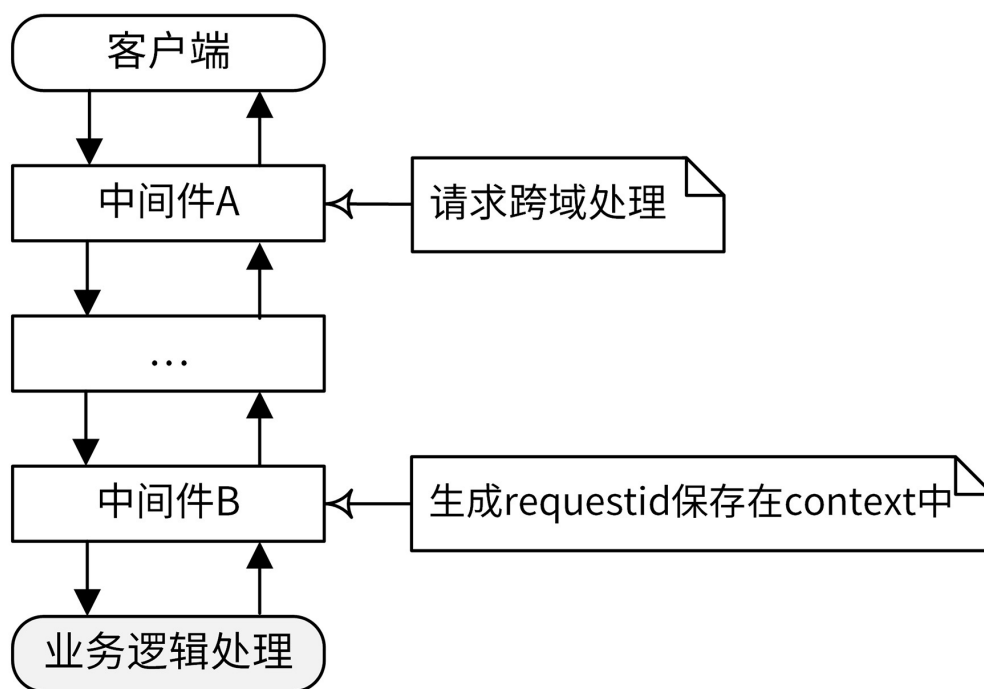
在Go项目开发中，我建议使用ShouldBindXXX，这样可以确保我们设置的HTTP Chain（Chain可以理解为一个HTTP请求的一系列处理插件）能够继续被执行。

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：

```
router := gin.New()
router.Use(gin.Logger(), gin.Recovery()) // 中间件作用于所有的HTTP请求
v1 := router.Group("/v1").Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"})) // 中间件作用于
v1.POST("/login", Login).Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"})) //中间件只作用于.
```

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：

```
package main

import (
    "log"
    "time"

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

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

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

        // 请求之前

        c.Next()

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

        // 访问我们发送的状态
        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)

        // it would print: "12345"
        log.Println(example)
    })

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

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：

```

router := gin.New()

// 认证
router.Use(gin.BasicAuth(gin.Accounts{"foo": "bar", "colin": "colin404"}))

// RequestID
router.Use(requestid.New(requestid.Config{
    Generator: func() string {
        return "test"
    },
}))

// 跨域
// CORS for https://foo.com and https://github.com origins, allowing:
// - PUT and PATCH methods
// - Origin header
// - Credentials share
// - Preflight requests cached for 12 hours
router.Use(cors.New(cors.Config{
    AllowOrigins:    []string{"https://foo.com"},
    AllowMethods:    []string{"PUT", "PATCH"},
    AllowHeaders:    []string{"Origin"},
    ExposeHeaders:   []string{"Content-Length"},
    AllowCredentials: true,
    AllowOriginFunc: func(origin string) bool {

```



```
        return origin == "https://github.com"
    },
    MaxAge: 12 * time.Hour,
})))
```

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[fvbock/endless](https://github.com/fvbock/endless)。我们可以使用fvbock/endless来替换掉net/http的ListenAndServe方法，例如：

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

```
// +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,
}

// Initializing the server in a goroutine so that
// it won't block the graceful shutdown handling below
go func() {
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        log.Fatalf("listen: %s\n", err)
    }
}()

// Wait for interrupt signal to gracefully shutdown the server with
// a timeout of 5 seconds.
quit := make(chan os.Signal)
// kill (no param) default send syscall.SIGTERM
// kill -2 is syscall.SIGINT
// kill -9 is syscall.SIGKILL but can't be catch, so don't need add it
signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
<-quit
log.Println("Shutting down server...")

// The context is used to inform the server it has 5 seconds to finish
// the request it is currently handling
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")
}

```

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，quit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。

精选留言：

- huntersudo 2021-07-20 15:17:18
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！！ [1赞]