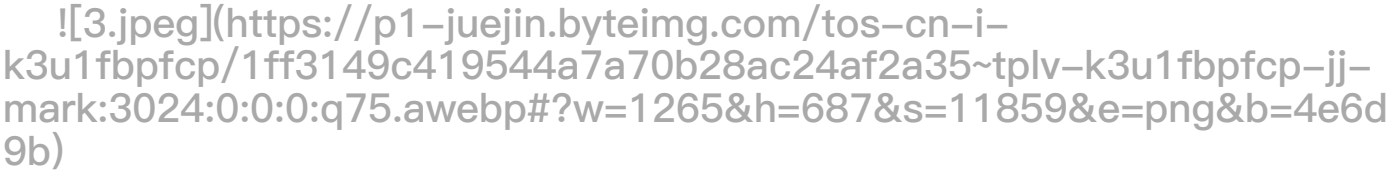
精通Gin服务器(一)

==========



精通Gin服务器(一)

目录

⊙Gin框架简介

⊙Gin框架的优势

⊙实现原理

⊙使用Gin框架的步骤

⊙使用Gin可能遇到的问题

⊙代码示例

⊙ 深入理解Gin框架核心用法

Gin框架简介

Gin是一个用Go语言实现的Web框架，它小巧而灵活，便于使用。Gin框架被广泛应用于构建大型web应用程序和API的开发

在没有Gin之前，Go开发web服务常用的方式是使用net/http库，但编写大规模的web服务可能会面临一些问题，如路由配置繁琐，错误处理不方便等

引入Gin后，开发者可以更方便的进行web服务的开发，简化了路由的配置，并提供了丰富的中间件支持

Gin框架的优势

1. 快速路由：Gin框架由于采用了httprouter路由器，因此在处理请求上表现非常快速
2. 中间件支持：中间件是在处理HTTP请求时在路由处理函数之前执行的函数，Gin有非常全面的中间件支持，例如生成日志，处理错误和验证请求等
3. 易于扩展：Gin框架支持创建中间件，方便进行功能扩展，并且有很多第三方中间件可以直接使用
4. 数据验证：Gin已内建绑定和验证JSON，XML和其他格式的请求体。只需要定义结构体，就可以自动处理复杂的数据验证

技术原理

作为一个HTTP web框架，Gin框架在处理每一个HTTP请求时，都会新开一个goroutine，在这个goroutine中完成整个HTTP请求的处理过程

Gin将HTTP请求的处理过程分解为多个处理器函数，然后按照请求的到来的顺序，依次调用这些处理器函数，逐步完成HTTP请求的完整处理请求。Gin框架的核心是Engine结构体，Engine结构体包含了路由、中间件、处理器函数等信息，负责整个HTTP请求的处理流程

1. 路由匹配：Gin框架内部使用httprouter作为路由，使用的是一种叫做Radix tree的数据结构，无论路由数量的大小，路由查找的时间复杂度都是O(1)级别的，所以性能非常好
2. 中间件：Gin的中间件的机制十分强大，用户可以自定义或使用已有的中间件，可以自由地在请求处理前、处理后或全局范围内使用这些中间件，以实现日志记录、错误处理、状态检查等功能
3. Gin HTTP处理流程：当Gin接收到一个HTTP请求后，会将其交给Engine的ServeHTTP方法，Engine会根据Context的路由信息找到对应的Handler Chain（也就是一系列的中间件和处理函数），最后将结果通过HTTP返回

使用Gin框架的步骤

1. 导入Gin框架：首先在项目中导入Gin包
2. 实例化一个Gin引擎：在Go中，gin.Default函数返回一个默认的框架引擎

3. 绑定路由规则函数：利用Gin固化路由规则和对应的处理函数
4. 运行服务：通过Run()函数，让服务在一个http的端口上监听并提供服务

使用Gin可能遇到的问题

1. 效率问题：虽然Gin非常方便且功能强大，但需要注意的是，过多地使用Gin提供的各种功能可能会降低服务的性能

2. 路由规则限制：Gin的路由匹配规则有一些限制，例如，同一个路径，使用不同的请求方法是可以注册的，但不能注册相同的路径

3. 错误处理：Gin自身不提供应用级错误处理，而是将错误放到上下文Context对象中，不会反悔，需要开发者自己实现错误处理的逻辑

代码实例

使用Gin创建一个简单的HTTP服务

```
package main

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

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

    // 创建路由组
    api := r.Group("/api")
    {
        api.GET("/ping", func(c *gin.Context) {
            c.JSON(200, gin.H{
                "message": "pong",
            })
        })
        api.POST("/post", func(c *gin.Context) {
            c.JSON(200, gin.H{
                "message": "POST request",
            })
        })
    }
```

```
    r.Run() // 在 0.0.0.0:8080 上监听并服务
}

```

在该示例中,首先创建了一个Gin引擎，然后定义了一个GET请求的处理，通过/ping路径就可以访问这个请求处理，返回JSON数据。最后我们通过Run启动并运行Gin引擎，开始监听服务

使用Gin创建一个带有中间件的HTTP服务

```
package main

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

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

    // 使用中间件
    r.Use(func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{
            "message": "This is a middleware",
        })
    })

    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })

    r.Run() // 在 0.0.0.0:8080 上监听并服务
}

```

在该示例中，我们使用了一个中间件，中间件是在处理HTTP请求时在路由处理函数之前执行的函数。在这个中间件中，我们返回了一个JSON数据，然后定义了一个GET请求的处理，通过/ping路径就可以访问这个请求处理，返回JSON数据。最后我们通过Run启动并运行Gin引擎，开始监听服务

使用路由组创建多个路由

```
package main

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

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

    // 创建路由组
    api := r.Group("/api")
    {
        api.GET("/ping", func(c *gin.Context) {
            c.JSON(200, gin.H{
                "message": "pong",
            })
        })
        api.POST("/post", func(c *gin.Context) {
            c.JSON(200, gin.H{
                "message": "POST request",
            })
        })
    }

    r.Run() // 在 0.0.0.0:8080 上监听并服务
}
```

在这个例子中，我们创建了一个/api的路由组，然后在这个路由组中定义了两个路由，一个是GET请求的处理，一个是POST请求的处理。最后我们通过Run启动并运行Gin引擎，开始监听服务

深入理解Gin框架核心用法

中间件执行顺序解析

Gin框架中的中间件执行顺序是按照注册的顺序执行的，即先注册的中间件先执行，后注册的中间件后执行

c.Next()：如果在中间件中调用了c.Next()方法，则会继续执行后续的中间件，如果不调用c.Next()方法，则不会执行后续的中间件。需要注意的是，c.Next() 之前的代码执行顺序是”进入”的顺序，也就是注册的顺序；而 c.Next() 之后的代码则是以 ”返回”的顺序执行的，也就是注册的反序

```
router.Use(func(c *gin.Context) {
    // 部分代码1
    c.Next()
    // 部分代码2
})
router.Use(func(c *gin.Context) {
    // 部分代码3
    c.Next()
    // 部分代码4
})
```

执行顺序会是 ： 部分代码1 –> 部分代码3 –> 部分代码4 –> 部分代码2

c.Abort()：如果在中间件中调用了c.Abort()方法，则会终止后续的中间件的执行，直接返回响应。但需要注意的是，如果在 c.Abort() 调用前有中间件已经调用过 c.Next()，那么这些中间件在 c.Next() 之后的部分会正常执行

```
router.Use(func(c *gin.Context) {
    // (mw1.1)部分代码...
    c.Next()
    // (mw1.2)部分代码...
})
router.Use(func(c *gin.Context) {
    // (mw2.1)部分代码...
    c.Abort()
    // (mw2.2)部分代码...
})
router.Use(func(c *gin.Context) {
    // (mw3)部分代码...
    c.Next()
})
```

执行顺序会是：mw1.1 -> mw2.1 -> mw1.2

c.JSON()：如果在中间件中调用了c.JSON()方法，相当于在响应体中设置了内容，但不会终止后续的中间件的执行

```
router.GET("/test", func(c *gin.Context) {
    // 中间件1
    c.JSON(200, gin.H{
        "message": "Hello from middleware1",
    })
    c.Next()

    // 中间件2
    networkData := fetchSomeData()
    c.JSON(200, gin.H{
        "message": "Hello from middleware2",
        "data": networkData,
    })
    c.Next()
//...
})
```

c.JSON() 被调用两次，但是因为HTTP规定一个请求只能有一个响应，所以最终实际生效的只有最后一个调用的 c.JSON()。一般在调用 c.JSON 方法后，如果不希望后续的逻辑继续执行，可以调用 c.Abort() 方法来停止处理过程

跨域处理中间件

跨域资源共享（CORS）是一种机制，它使用额外的HTTP头来告诉浏览器允许一个Web应用运行在一个不同于自身的源。在Gin框架中，我们可以使用cors中间件来处理跨域问题

```
package main

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

```
    "github.com/gin-contrib/cors"
)

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

    // 使用CORS中间件
    r.Use(cors.Default())

    ...
}
```

上述利用开源的cors中间件实现，也可以自定义CORS中间件，如下：

```
func Cors() gin.HandlerFunc {
  return func(c *gin.Context) {
    // 添加请求头 Access-Control-Allow-Origin ， 值设为 * ， 意味着允许所有源访问
    c.Writer.Header().Set("Access-Control-Allow-Origin", "*")
    // 添加请求头 Access-Control-Allow-Credentials ， 值设置为 true ， 意味着允许携带凭证信息（cookie等）访问
    c.Header("Access-Control-Allow-Credentials", "true")
    // 添加请求头 Access-Control-Allow-Headers ， 允许接收的请求头类型 ，常见的例如"Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization, accept, origin, Cache-Control, X-Requested-With"
    c.Header("Access-Control-Allow-Headers", "Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization, accept, origin, Cache-Control, X-Requested-With")
    // 添加请求头 Access-Control-Allow-Methods ， 允许接收的请求方法类型，如"POST, OPTIONS, GET, PUT, DELETE"
    c.Header("Access-Control-Allow-Methods", "POST, OPTIONS, GET, PUT, DELETE")
    // 如果请求方法为 OPTIONS 则说明是预检请求，此时我们返回 HTTP 204 状态，并阻止之后的处理函数执行
    if c.Request.Method == "OPTIONS" {
      c.AbortWithStatus(204)
      return
    }
    // 使用 defer 和 recover 来捕获和处理可能出现的 panic
    defer func() {
      if err := recover(); err != nil {
        log.Printf("CORS middleware panic: %v", err)
```

```
        // 如果出现 panic，返回 HTTP 500 错误
        c.JSON(http.StatusInternalServerError, gin.H{
          "message": "Internal Server Error",
        })
      }
    }()
    c.Next()
  }
}

```

自定义全局处理异常中间件

在Gin框架中，我们可以使用中间件来处理异常，然后返回一个统一的错误信息

```
package main

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

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

    // recover()函数会捕获panic，如果没有发生panic，recover()会返回nil
    r.Use(func(c *gin.Context) {
        defer func() {
            if err := recover(); err != nil {
                // 自定义错误处理逻辑
                c.JSON(500, gin.H{
                    "message": "Internal Server Error",
                })
            }
        }()
        c.Next()
    })

    r.GET("/ping", func(c *gin.Context) {
        panic("An unexpected error occurred")
    })

    r.Run() // 在Gin框架中监听并服务
}
```

```

在上述代码中，我们定义了一个异常处理中间件，当路由处理函数中出现异常时，会被捕获并返回一个统一的错误信息。这样可以避免在路由处理函数中处理异常，提高代码的可读性和可维护性

这里需要注意的是defer的使用，defer语句会在函数执行完毕后执行，而这里的函数执行完毕，包含了/ping路由处理函数的执行（由函数中的c.Next()执行），所以在路由处理函数中出现异常时，会被这里的defer捕获到

原文链接: https://juejin.cn/post/7361268292801789988
```