bobby · 更新于 2022-11-08

上一节 4. JSON、Prot... 6. 中间件和next... 下一节

validator库参数校验若干实用技巧

1. 表单的基本验证

若要将请求主体绑定到结构体中，请使用模型绑定，目前支持JSON、XML、YAML和标准表单值(foo=bar&boo=baz)的绑定。
Gin使用 go-playground/validator 验证参数，查看完整文档。

需要在绑定的字段上设置tag，比如，绑定格式为json，需要这样设置 `json:"fieldname"` 。
此外，Gin还提供了两套绑定方法：

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

当我们使用绑定方法时，Gin会根据Content-Type推断出使用哪种绑定器，如果你确定你绑定的是什么，你可以使用 `MustBindWith` 或者 `BindingWith` 。
你还可以给字段指定特定规则的修饰符，如果一个字段用 `binding:"required"` 修饰，并且在绑定时该字段的值为空，那么将返回一个错误。

代码块

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

```go
type SignUpParam struct {
    Age        uint8  `json:"age" binding:"gte=1,lte=130"`
    Name       string `json:"name" binding:"required"`
    Email      string `json:"email" binding:"required,email"`
    Password   string `json:"password" binding:"required"`
    RePassword string `json:"re_password" binding:"required,eqfield=Password"`
}

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

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

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

    // Example for binding a HTML form (user=manu&password=123)
    router.POST("/loginForm", func(c *gin.Context) {
        var form Login
        // This will infer what binder to use depending on the content-type heade
        if err := c.ShouldBind(&form); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

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

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

    r.POST("/signup", func(c *gin.Context) {
        var u SignUpParam
        if err := c.ShouldBind(&u); err != nil {
            c.JSON(http.StatusOK, gin.H{
                "msg": err.Error(),
            })
            return
        }
        // 保存入库等业务逻辑代码...

        c.JSON(http.StatusOK, "success")
    })

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

意见反馈 收藏教程 标记书签

代码块

```go
package main

import (
    "fmt"
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/binding"
    "github.com/go-playground/locales/en"
    "github.com/go-playground/locales/zh"
    ut "github.com/go-playground/universal-translator"
    "github.com/go-playground/validator/v10"
    enTranslations "github.com/go-playground/validator/v10/translations/en"
    zhTranslations "github.com/go-playground/validator/v10/translations/zh"
)

// 定义一个全局翻译器T
var trans ut.Translator

// InitTrans 初始化翻译器
func InitTrans(locale string) (err error) {
    // 修改gin框架中的Validator引擎属性，实现自定制
    if v, ok := binding.Validator.Engine().(*validator.Validate); ok {

        zhT := zh.New() // 中文翻译器
        enT := en.New() // 英文翻译器

        // 第一个参数是备用（fallback）的语言环境
        // 后面的参数是应该支持的语言环境（支持多个）
        // uni := ut.New(zhT, zhT) 也是可以的
        uni := ut.New(enT, zhT, enT)

        // locale 通常取决于 http 请求头的 'Accept-Language'
        var ok bool
        // 也可以使用 uni.FindTranslator(...) 传入多个locale进行查找
        trans, ok = uni.GetTranslator(locale)
        if !ok {
            return fmt.Errorf("uni.GetTranslator(%s) failed", locale)
        }

        // 注册翻译器
        switch locale {
        case "en":
            err = enTranslations.RegisterDefaultTranslations(v, trans)
        case "zh":
            err = zhTranslations.RegisterDefaultTranslations(v, trans)
        default:
            err = enTranslations.RegisterDefaultTranslations(v, trans)
        }
        return
    }
    return
}

type SignUpParam struct {
    Age        uint8  `json:"age" binding:"gte=1,lte=130"`
    Name       string `json:"name" binding:"required"`
    Email      string `json:"email" binding:"required,email"`
    Password   string `json:"password" binding:"required"`
    RePassword string `json:"re_password" binding:"required,eqfield=Password"`
}

func main() {
```

意见反馈 收藏教程 标记书签

```
65          return
66      }
67
68      r := gin.Default()
69
70      r.POST("/signup", func(c *gin.Context) {
71          var u SignUpParam
72          if err := c.ShouldBind(&u); err != nil {
73              // 获取validator.ValidationErrors类型的errors
74              errs, ok := err.(validator.ValidationErrors)
75              if !ok {
76                  // 非validator.ValidationErrors类型错误直接返回
77                  c.JSON(http.StatusOK, gin.H{
78                      "msg": err.Error(),
79                  })
80                  return
81              }
82              // validator.ValidationErrors类型错误则进行翻译
83              c.JSON(http.StatusOK, gin.H{
84                  "msg":errs.Translate(trans),
85              })
86              return
87          }
88          // 保存入库等具体业务逻辑代码...
89
90          c.JSON(http.StatusOK, "success")
91      })
92
93      _ = r.Run(":8999")
94  }
```

3. 进一步改进校验方法

上面的错误提示看起来是可以了，但是还是差点意思，首先是错误提示中的字段并不是请求中使用的字段，例如： `RePassword` 是我们后端定义的结构体中的字段名，而请求中使用的是 `re_password` 字段。
如何是错误提示中的字段使用自定义的名称，例如 `json` tag指定的值呢?
只需要在初始化翻译器的时候像下面一样添加一个获取 `json` tag的自定义方法即可。

代码块

```
1   // InitTrans 初始化翻译器
2   func InitTrans(locale string) (err error) {
3       // 修改gin框架中的Validator引擎属性，实现自定制
4       if v, ok := binding.Validator.Engine().(*validator.Validate); ok {
5
6           // 注册一个获取json tag的自定义方法
7           v.RegisterTagNameFunc(func(fld reflect.StructField) string {
8               name := strings.SplitN(fld.Tag.Get("json"), ",", 2)[0]
9               if name == "-" {
10                  return ""
11              }
12              return name
13          })
14
15          zhT := zh.New() // 中文翻译器
16          enT := en.New() // 英文翻译器
17
18          // 第一个参数是备用（fallback）的语言环境
19          // 后面的参数是应该支持的语言环境（支持多个）
20          // uni := ut.New(zhT, zhT) 也是可以的
21          uni := ut.New(enT, zhT, enT)
```

意见反馈 收藏教程 标记书签

但是还是有点瑕疵，那就是最终的错误提示信息中心还是有我们后端定义的结构体名称——`SignUpParam`，这个名称其实是不需要随错误提示返回给前端的，前端并不需要这个值。我们需要想办法把它去掉。定义一个去掉结构体名称前缀的自定义方法：

代码块

```
func removeTopStruct(fields map[string]string) map[string]string {
    res := map[string]string{}
    for field, err := range fields {
        res[field[strings.Index(field, ".")+1:]] = err
    }
    return res
}
...

if err := c.ShouldBind(&u); err != nil {
    // 获取validator.ValidationErrors类型的errors
    errs, ok := err.(validator.ValidationErrors)
    if !ok {
        // 非validator.ValidationErrors类型错误直接返回
        c.JSON(http.StatusOK, gin.H{
            "msg": err.Error(),
        })
        return
    }
    // validator.ValidationErrors类型错误则进行翻译
    // 并使用removeTopStruct函数去除字段名中的结构体名称标识
    c.JSON(http.StatusOK, gin.H{
        "msg": removeTopStruct(errs.Translate(trans)),
    })
    return
}
```

4. JSON、ProtoBuf 渲染 ‹ 上一节　　下一节 › 6. 中间件和next函数

我要提出意见反馈

企业服务　网站地图　网站首页　关于我们　联系我们　讲师招募　帮助中心　意见反馈　代码托管

意见反馈　收藏教程　标记书签