目　录

首页

Echo-intro

本项目为 Golang Echo 框架官方文档的汉化文档

go语言中文文档：http://www.topgoer.com/

转自：https://github.com/hilaily/echo-intro

快速开始

快速开始

安装

```
$ go get -u github.com/labstack/echo/...
```

编写 Hello, World!

创建 `server.go` 文件

```
package main

import (
    "net/http"

    "github.com/labstack/echo"
)

func main() {
    e := echo.New()
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Hello, World!")
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

启动服务

```
$ go run server.go
```

用浏览器访问 http://localhost:1323 然后就能在页面上看到 `Hello, World!`

路由

```
e.POST("/users", saveUser)
e.GET("/users/:id", getUser)
```

```
e.PUT("/users/:id", updateUser)
e.DELETE("/users/:id", deleteUser)
```

URL路径参数

```
// e.GET("/users/:id", getUser)
func getUser(c echo.Context) error {
    // User ID 来自于url `users/:id`
    id := c.Param("id")
    return c.String(http.StatusOK, id)
}
```

用浏览器访问 http://localhost:1323/users/Joe 然后就能在页面上看到 `Joe`

请求参数

`/show?team=x-men&member=wolverine`

```
// e.GET("/show", show)
func show(c echo.Context) error {
    // 从请求参数里获取 team 和 member 的值
    team := c.QueryParam("team")
    member := c.QueryParam("member")
    return c.String(http.StatusOK, "team:" + team + ", member:" + member)
}
```

从浏览器访问 `http://localhost:1323/show?team=x-men&member=wolverine` 可以看到页面上显示"team:x-men, member:wolverine"

表单 `application/x-www-form-urlencoded`

`POST` `/save`

name	value
name	Joe Smith
email	joe@labstack.com

```
// e.POST("/save", save)
func save(c echo.Context) error {
    // 获取 name 和 email 的值
```

```
    name := c.FormValue("name")
    email := c.FormValue("email")
    return c.String(http.StatusOK, "name:" + name + ", email:" + email)
}
```

在命令行里执行下面的语句

```
$ curl -F "name=Joe Smith" -F "email=joe@labstack.com" http://localhost:1323/sav
e
```

控制台会输出 `name:Joe Smith, email:joe@labstack.com`

表单 `multipart/form-data`

`POST` `/save`

name	value
name	Joe Smith
avatar	avatar

```
func save(c echo.Context) error {
    // Get name
    name := c.FormValue("name")
    // Get avatar
      avatar, err := c.FormFile("avatar")
      if err != nil {
          return err
      }

      // Source
      src, err := avatar.Open()
      if err != nil {
          return err
      }
      defer src.Close()

      // Destination
      dst, err := os.Create(avatar.Filename)
      if err != nil {
          return err
      }
```

```
    defer dst.Close()

    // Copy
    if _, err = io.Copy(dst, src); err != nil {
        return err
    }

    return c.HTML(http.StatusOK, "<b>Thank you! " + name + "</b>")
}
```

命令行执行下面语句

```
$ curl -F "name=Joe Smith" -F "avatar=@/path/to/your/avatar.png" http://localhost:1323/save
//output => <b>Thank you! Joe Smith</b>
```

使用以下命令查看刚刚上传的图片

```
cd <project directory>
ls avatar.png
// => avatar.png
```

处理请求

- 根据 Content-Type 请求标头将 `json`，`xml`，`form` 或 `query` 负载绑定到 Go 结构中。
- 通过状态码将响应渲染为 `json` 或者 `xml` 格式。

```
type User struct {
    Name  string `json:"name" xml:"name" form:"name" query:"name"`
    Email string `json:"email" xml:"email" form:"email" query:"email"`
}

e.POST("/users", func(c echo.Context) error {
    u := new(User)
    if err := c.Bind(u); err != nil {
        return err
    }
    return c.JSON(http.StatusCreated, u)
    // 或者
    // return c.XML(http.StatusCreated, u)
})
```

静态资源

下面的代码定义 `/static/*` 目录为静态资源文件目录

```
e.Static("/static", "static")
```

了解更多

模板渲染

中间件

```
// Root level middleware
e.Use(middleware.Logger())
e.Use(middleware.Recover())

// Group level middleware
g := e.Group("/admin")
g.Use(middleware.BasicAuth(func(username, password string, c echo.Context) (erro
r, bool) {
  if username == "joe" && password == "secret" {
    return nil, true
  }
  return nil, false
}))

// Route level middleware
track := func(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        println("request to /users")
        return next(c)
    }
}
e.GET("/users", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users")
}, track)
```

了解更多

指南

安装

安装

要求

- 安装 Go
- 设置 GOPATH 环境变量

使用 go get

```
$ cd <PROJECT IN $GOPATH>
$ go get -u github.com/labstack/echo/...
```

使用 dep

```
$ cd <PROJECT IN $GOPATH>
$ dep ensure -add github.com/labstack/echo@^3.1
```

使用 glide

```
$ cd <PROJECT IN $GOPATH>
$ glide get github.com/labstack/echo#^3.1
```

使用 govender

```
$ cd <PROJECT IN $GOPATH>
$ govendor fetch github.com/labstack/echo@v3.1
```

Echo 使用 Go `1.10.x` 开发，并通过了 `1.9.x` 和 `1.10.x` 的测试。

Echo 通过 GitHub releases 进行 语义化版本(semantic versioning) 控制，特定的版本可以使用 package manager 安装。

自定义

自定义

Debug

可使用 `Echo#Debug` 来开启/关闭 debug 模式。Debug 模式下的日志级别是 **DEBUG**。

日志

日志默认使用 JSON 格式，可从通过修改标头进行格式修改。

日志标头

`Echo#Logger.SetHeader(io.Writer)` 用于日志标头的设置，默认值为：

```
{"time":"${time_rfc3339_nano}","level":"${level}","prefix":"${prefix}","file":"${short_file}","line":"${line}"}
```

示例

```
import "github.com/labstack/gommon/log"

/* ... */

if l, ok := e.Logger.(*log.Logger); ok {
  l.SetHeader("${time_rfc3339} ${level}")
}
```

```
2018-05-08T20:30:06-07:00 INFO info
```

可用标签

- `time_rfc3339`
- `time_rfc3339_nano`
- `level`
- `prefix`
- `long_file`

- `short_file`
- `line`

日志输出

`Echo#Logger.SetOutput(io.Writer)` 用于设置日志输出的位置，默认是 `os.Stdout` 。

若需禁用日志，可使用 `Echo#Logger.SetOutput(ioutil.Discard)` 或 `Echo#Logger.SetLevel(log.OFF)` 来完成。

日志级别

`Echo#Logger.SetLevel(log.Lvl)` 用于设置日志级别，默认是 `ERROR` 。可选值包括：

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `OFF`

自定义日志

Echo 的日志实现了 `echo.Logger` 接口，该接口允许使用 `Echo#Logger` 注册自定义日志。

自定义 Server

使用 `Echo#StartServer()` 进行自定义 Server 的启动

示例

```
s := &http.Server{
  Addr:         ":1323",
  ReadTimeout:  20 * time.Minute,
  WriteTimeout: 20 * time.Minute,
}
e.Logger.Fatal(e.StartServer(s))
```

启动横幅

使用 `Echo#HideBanner` 隐藏启动横幅。

自定义监听器

使用 `Echo#*Listener` 启动一个自定义的 listener。
示例

```
l, err := net.Listen("tcp", ":1323")
if err != nil {
  e.Logger.Fatal(l)
}
e.Listener = l
e.Logger.Fatal(e.Start(""))
```

禁用 HTTP/2

使用 `Echo#DisableHTTP2` 关闭 HTTP/2 协议。

读取超时

使用 `Echo#*Server#ReadTimeout` 设置读取请求的最大时间。

写入超时

使用 `Echo#*Server#WriteTimeout` 设置写入响应的最大时间。

验证

使用 `Echo#Validator` 注册一个验证器，从而对请求负载执行数据验证。

查看更多

自定义绑定

使用 `Echo#Binder` 注册一个绑定器，从而绑定请求负载。

查看更多

渲染

使用 `Echo#Renderer` 注册一个渲染引擎，从而进行模板渲染。

HTTP 错误处理

使用 `Echo#HTTPErrorHandler` 注册一个 http 错误处理器。

查看更多

Context

Context

echo.Context 表示当前 HTTP 请求的上下文。通过路径、路径参数、数据、注册处理器和相关 API 进行请求的读取与响应的输出。由于 Context 是一个接口，也可以轻松地使用自定义 API 进行扩展。

扩展 Context

定义一个自定义 context

```
type CustomContext struct {
    echo.Context
}

func (c *CustomContext) Foo() {
    println("foo")
}

func (c *CustomContext) Bar() {
    println("bar")
}
```

创建一个中间件来扩展默认的 context

```
e.Use(func(h echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        cc := &CustomContext{c}
        return h(cc)
    }
})
```

> 此中间件应在任何其他中间件之前注册。

在处理器中使用

```
e.Get("/", func(c echo.Context) error {
    cc := c.(*CustomContext)
    cc.Foo()
    cc.Bar()
```

```
    return cc.String(200, "OK")
})
```

Cookies

Cookies

Cookie 是用户访问网站时浏览器上存储的小型文本文件，由服务器发送而来。每当用户加载网站时，浏览器都会将 cookie 发送回服务器以通知用户之前的活动。
Cookie 作为一个可靠验证凭据，可用来记录状态信息（比如在线商城购物车中的商品）或记录用户的浏览器活动（包括单击特定按钮，登录或记录访问过的页面）。Cookie 还可以存储用户先前输入的密码和表单内容，例如信用卡号或地址。

Cookie 属性

属性	可选
Name	No
Value	No
Path	Yes
Domain	Yes
Expires	Yes
Secure	Yes
HTTPOnly	Yes

Echo 使用 golang 自带的 `http.Cookie` 对象写入 / 读取从上下文中的 cookie。

创建一个 Cookie

```
func writeCookie(c echo.Context) error {
    cookie := new(http.Cookie)
    cookie.Name = "username"
    cookie.Value = "jon"
    cookie.Expires = time.Now().Add(24 * time.Hour)
    c.SetCookie(cookie)
    return c.String(http.StatusOK, "write a cookie")
}
```

- 使用 `new(http.Cookie)` 创建Cookie。
- cookie 的属性值会被赋值给 `http.Cookie` 的可导出属性。
- 最后，使用 `c.SetCookie(cookies)` 来给 HTTP 响应增加 `Set-Cookie` 头。

读取 Cookie

```
func readCookie(c echo.Context) error {
    cookie, err := c.Cookie("username")
    if err != nil {
        return err
    }
    fmt.Println(cookie.Name)
    fmt.Println(cookie.Value)
    return c.String(http.StatusOK, "read a cookie")
}
```

- Cookie 通过名称从 HTTP 请求里读取：`c.Cookie("name")`。
- Cookie 的属性可以使用 `Getter` 方法获取。

读取所有 Cookies

```
func readAllCookies(c echo.Context) error {
    for _, cookie := range c.Cookies() {
        fmt.Println(cookie.Name)
        fmt.Println(cookie.Value)
    }
    return c.String(http.StatusOK, "read all cookie")
}
```

错误处理

错误处理程序

Echo 提倡通过中间件或处理程序 (handler) 返回 HTTP 错误集中处理。集中式错误处理程序允许我们从统一位置将错误记录到外部服务，并向客户端发送自定义 HTTP 响应。

你可以返回一个标准的 `error` 或者 `echo.*HTTPError` 。

例如，当基本身份验证中间件找到无效凭据时，会返回 401未授权错误 (401-Unauthorized)，并终止当前的 HTTP 请求。

```
    e.Use(func(next echo.HandlerFunc) echo.HandlerFunc {
        return func(c echo.Context) error {
            // Extract the credentials from HTTP request header and perform a se
curity
            // check

            // For invalid credentials
            return echo.NewHTTPError(http.StatusUnauthorized)

            // For valid credentials call next
            // return next(c)
        }
    })
}
```

你也可以不带消息内容调用 `echo.NewHTTPError()` ，这种情况下状态文本会被用作错误信息，例如 `Unauthorized` 。

默认 HTTP 错误处理程序

Echo 提供了默认的 HTTP 错误处理程序，它用 JSON 格式发送错误。

```
{
  "message": "error connecting to redis"
}
```

标准错误 `error` 的响应是 `500 - Internal Server Error` 。然而在调试 (debug) 模式模式下，原始的错误信息会被发送。如果错误是 `*HTTPError` ，则使用设置的状态代码和消息发送响应。如果启用了日志记录，则还会记录错误消息。

自定义 HTTP 错误处理程序

通过 `e.HTTPErrorHandler` 可以设置自定义的 HTTP 错误处理程序 (error handler) 。

通常默认的 HTTP 错误处理程序已经够用；然而如果要获取不同类型的错误并采取相应的操作，则可以使用自定义 HTTP 错误处理程序，例如发送通知邮件或记录日志到应用中心的场景。最后，你还可以发送自定义的错误页面或 JSON 响应给客户端。

错误页

利用自定义 HTTP 错误处理程序，可以在显示不同种类的错误页面的同时，记录错误日志。错误页的名称可写作 `<CODE>.html` ，例如 `500.html` 。你可以在 https://github.com/AndiDittrich/HttpErrorPages看到 Echo 内置的错误页。

```
func customHTTPErrorHandler(err error, c echo.Context) {
    code := http.StatusInternalServerError
    if he, ok := err.(*echo.HTTPError); ok {
        code = he.Code
    }
    errorPage := fmt.Sprintf("%d.html", code)
    if err := c.File(errorPage); err != nil {
        c.Logger().Error(err)
    }
    c.Logger().Error(err)
}

e.HTTPErrorHandler = customHTTPErrorHandler
```

> 日志除了记录到 logger，也可以记录到第三方服务，例如 Elasticsearch 或者 Splunk。

迁移

迁移

更新日志

- 通过 Let's Encrypt 自动生成 TLS 证书
- 内置优雅停机 (graceful shutdown)
- 提供用于封装标准处理程序 (standard handler) 和中间件 (middleware) 的功能函数
- `Map` 类型简单表述为 `map[string]interface{}`
- 上下文 (context) 现在封装了标准的 `net/http` 请求与响应
- 新的配置
  - `Echo#ShutdownTimeout`
  - `Echo#DisableHTTP2`
- 新的 API
  - `Echo#Start()`
  - `Echo#StartTLS()`
  - `Echo#StartAutoTLS()`
  - `Echo#StartServer()`
  - `Echo#Shutdown()`
  - `Echo#ShutdownTLS()`
  - `Context#Scheme()`
  - `Context#RealIP()`
  - `Context#IsTLS()`
- Echo 利用以下属性替代 setter/getter 方法
  - Binder
  - Renderer
  - HTTPErrorHandler
  - Debug
  - Logger
- 改善重定向和 CORS 中间件
- 由于 `Echo#Static` 的存在废除了 static 中间件

- 删除 API
  - Echo#Run()
  - Context#P()
- 删除标准 `Context` 支持
- 删除 `fasthttp`
- 删除标记为 deprecated 的 API
- `Logger` 接口移至 root 级别
- 将网站和示例移至主仓库
- 更新文档以及修复了大量问题 (issues)

Cookbook

请求

请求

数据绑定

可以使用 `Context#Bind(i interface{})` 将请求内容体绑定至 go 的结构体。默认绑定器支持基于 Content-Type 标头包含 application/json，application/xml 和 application/x-www-form-urlencoded 的数据。

下面是绑定请求数据到 `User` 结构体的例子

```
// User
User struct {
  Name  string `json:"name" form:"name" query:"name"`
  Email string `json:"email" form:"email" query:"email"`
}
```

```
// Handler
func(c echo.Context) (err error) {
  u := new(User)
  if err = c.Bind(u); err != nil {
    return
  }
  return c.JSON(http.StatusOK, u)
}
```

JSON 数据

```
curl \
  -X POST \
  http://localhost:1323/users \
  -H 'Content-Type: application/json' \
  -d '{"name":"Joe","email":"joe@labstack"}'
```

Form 表单数据

```
curl \
  -X POST \
```

```
  http://localhost:1323/users \
  -d 'name=Joe' \
  -d 'email=joe@labstack.com'
```

查询参数 (Query Parameters)

```
curl \
  -X GET \
  http://localhost:1323/users\?name\=Joe\&email\=joe@labstack.com
```

自定义绑定器

可以通过 `Echo#Binder` 注册自定义绑定器。

示例

```
type CustomBinder struct {}

func (cb *CustomBinder) Bind(i interface{}, c echo.Context) (err error) {
    // 你也许会用到默认的绑定器
    db := new(echo.DefaultBinder)
    if err = db.Bind(i, c); err != echo.ErrUnsupportedMediaType {
        return
    }

    // 做你自己的实现

    return
}
```

检索数据

Form 表单数据

表单数据可以通过名称检索，使用 `Context#FormValue(name string)` 方法。

示例

```
// Handler
func(c echo.Context) error {
    name := c.FormValue("name")
    return c.String(http.StatusOK, name)
}
```

```
curl \
  -X POST \
  http://localhost:1323 \
  -d 'name=Joe'
```

当然，你也可以通过实现 `Echo#BindUnmarshaler` 接口来绑定自定义数据类型。

```
type Timestamp time.Time

func (t *Timestamp) UnmarshalParam(src string) error {
    ts, err := time.Parse(time.RFC3339, src)
    *t = Timestamp(ts)
    return err
}
```

查询参数 (Query Parameters)

查询参数可以通过名称获取，使用 `Context#QueryParam(name string)` 方法。

示例

```
// Handler
func(c echo.Context) error {
    name := c.QueryParam("name")
    return c.String(http.StatusOK, name)
})
```

```
curl \
  -X GET \
  http://localhost:1323\?name\=Joe
```

和表单数据一样，自定义数据类型依然通过 `Context#QueryParam(name string)` 进行绑定。

路径参数 (Path Parameters)

路径参数可以通过 `Context#Param(name string) string` 进行检索。

示例

```
e.GET("/users/:name", func(c echo.Context) error {
    name := c.Param("name")
```

```
    return c.String(http.StatusOK, name)
})
```

```
$ curl http://localhost:1323/users/Joe
```

数据验证

Echo 没有内置的数据验证功能，但是可以通过 `Echo#Validator` 和第三方库来注册一个数据验证器。

下面例子使用 https://github.com/go-playground/validator 所展示的框架来做验证：

```
type (
    User struct {
        Name  string `json:"name" validate:"required"`
        Email string `json:"email" validate:"required,email"`
    }

    CustomValidator struct {
        validator *validator.Validate
    }
)

func (cv *CustomValidator) Validate(i interface{}) error {
    return cv.validator.Struct(i)
}

func main() {
    e := echo.New()
    e.Validator = &CustomValidator{validator: validator.New()}
    e.POST("/users", func(c echo.Context) (err error) {
        u := new(User)
        if err = c.Bind(u); err != nil {
            return
        }
        if err = c.Validate(u); err != nil {
            return
        }
        return c.JSON(http.StatusOK, u)
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

```
curl \
  -X POST \
  http://localhost:1323/users \
  -H 'Content-Type: application/json' \
  -d '{"name":"Joe","email":"joe@invalid-domain"}'
{"message":"Key: 'User.Email' Error:Field validation for 'Email' failed on the
 'email' tag"}
```

响应

响应

发送 string 数据

`Context#String(code int, s string)` 用于发送一个带有状态码的纯文本响应。

示例

```
func(c echo.Context) error {
  return c.String(http.StatusOK, "Hello, World!")
}
```

发送 HTML 响应 (参考模板)

`Context#HTML(code int, html string)` 用于发送一个带有状态码的简单 HTML 响应。如果你需要动态生成 HTML 内容请查看模版。

示例

```
func(c echo.Context) error {
  return c.HTML(http.StatusOK, "<strong>Hello, World!</strong>")
}
```

发送 HTML Blob

`Context#HTMLBlob(code int, b []byte)` 用于发送一个带状态码的 HTML blob（二进制长对象）响应。可以发现，使用输出 `[]byte` 的模版引擎很方便。

模版引擎渲染

查看

发送 JSON 数据

`Context#JSON(code int, i interface{})` 用于发送一个带状态码的 JSON 对象，它会将 Golang 的对象转换成 JSON 字符串。

示例

```
// User
type User struct {
  Name  string `json:"name" xml:"name"`
  Email string `json:"email" xml:"email"`
}

// Handler
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "jon@labstack.com",
  }
  return c.JSON(http.StatusOK, u)
}
```

JSON 流

`Context#JSON()` 内部使用 `json.Marshal` 来转换 JSON 数据，但该方法面对大量的 JSON 数据会显得效率不足，对于这种情况可以直接使用 JSON 流。

示例

```
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "jon@labstack.com",
  }
  c.Response().Header().Set(echo.HeaderContentType, echo.MIMEApplicationJSONCharsetUTF8)
  c.Response().WriteHeader(http.StatusOK)
  return json.NewEncoder(c.Response()).Encode(u)
}
```

JSON 美化 (JSON Pretty)

`Context#JSONPretty(code int, i interface{}, indent string)` 可以发送带有缩进（可以使用空格和 tab）的更为好看的 JSON 数据。

示例

```
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "joe@labstack.com",
```

```
  }
  return c.JSONPretty(http.StatusOK, u, "  ")
}
```

```
{
  "email": "joe@labstack.com",
  "name": "Jon"
}
```

> 通过在请求URL查询字符串中附加 `pretty` ，你也可以使用 `Context#JSON()` 来输出带有缩进的 JSON 数据。

示例

```
curl http://localhost:1323/users/1?pretty
```

JSON Blob

`Context#JSONBlob(code int, b []byte)` 可用来从外部源（例如数据库）直接发送预编码的 JSON 对象。

示例

```
func(c echo.Context) error {
  encodedJSON := []byte{} // Encoded JSON from external source
  return c.JSONBlob(http.StatusOK, encodedJSON)
}
```

发送 JSONP 数据

`Context#JSONP(code int, callback string, i interface{})` 可以将 Golang 的数据类型转换成 JSON 类型，并通过回调以带有状态码的 JSONNP 结构发送。

查看示例

发送 XML 数据

`Context#XML(code int, i interface{})` 可以将 Golang 对象转换成 XML 类型，并带上状态码发送响应。

示例

```
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "jon@labstack.com",
  }
  return c.XML(http.StatusOK, u)
}
```

XML 流

`Context#XML` 内部使用 `xml.Marshal` 来转换 XML 数据，但该方法面对大量的 XML 数据会显得效率不足，对于这种情况可以直接使用 XML 流。

示例

```
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "jon@labstack.com",
  }
  c.Response().Header().Set(echo.HeaderContentType, echo.MIMEApplicationXMLCharsetUTF8)
  c.Response().WriteHeader(http.StatusOK)
  return xml.NewEncoder(c.Response()).Encode(u)
}
```

XML 美化 (XML Pretty)

`Context#XMLPretty(code int, i interface{}, indent string)` 可以发送带有缩进（可以使用空格和 tab）的更为好看的 XML 数据。

示例

```
func(c echo.Context) error {
  u := &User{
    Name:  "Jon",
    Email: "joe@labstack.com",
  }
  return c.XMLPretty(http.StatusOK, u, "  ")
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<User>
```

```
  <Name>Jon</Name>
  <Email>joe@labstack.com</Email>
</User>
```

> 通过在请求URL查询字符串中附加 `pretty` ，你也可以使用 `Context#XML()` 来输出带有缩进的 XML 数据。

示例

```
curl http://localhost:1323/users/1?pretty
```

XML Blob

`Context#XMLBlob(code int, b []byte)` 可用来从外部源（例如数据库）直接发送预编码的 XML 对象。

示例

```
func(c echo.Context) error {
  encodedXML := []byte{} // Encoded XML from external source
  return c.XMLBlob(http.StatusOK, encodedXML)
}
```

发送文件

`Context#File(file string)` 可用来发送内容为文件的响应，并且它能自动设置正确的内容类型、优雅地处理缓存。

示例

```
func(c echo.Context) error {
  return c.File("<文件路径>")
}
```

发送附件

`Context#Attachment(file, name string)` 和发送文件 `File()` 的方法类似，只是它的方法名称不同。

示例

```
func(c echo.Context) error {
  return c.Attachment("<PATH_TO_YOUR_FILE>")
}
```

发送内嵌 (Inline)

`Context#Inline(file, name string)` 和发送文件 `File()` 的方法类似，只是它的方法名称不同。

示例

```
func(c echo.Context) error {
  return c.Inline("<PATH_TO_YOUR_FILE>")
}
```

发送二进制长文件 (Blob)

`Context#Blob(code int, contentType string, b []byte)` 可用于发送带有内容类型 (content type) 和状态代码的任意类型数据。

示例

```
func(c echo.Context) (err error) {
  data := []byte(`0306703,0035866,NO_ACTION,06/19/2006
      0086003,"0005866",UPDATED,06/19/2006`)
    return c.Blob(http.StatusOK, "text/csv", data)
}
```

发送流 (Stream)

`Context#Stream(code int, contentType string, r io.Reader)` 可用于发送带有内容类型 (content type) 、状态代码、 `io.Reader` 的任意类型数据流。

示例

```
func(c echo.Context) error {
  f, err := os.Open("<PATH_TO_IMAGE>")
  if err != nil {
    return err
  }
  return c.Stream(http.StatusOK, "image/png", f)
}
```

发送空内容 (No Content)

`Context#NoContent(code int)` 可用于发送带有状态码的空内容。

示例

```
func(c echo.Context) error {
  return c.NoContent(http.StatusOK)
}
```

重定向

`Context#Redirect(code int, url string)` 可用于重定向至一个带有状态码的 URL。

示例

```
func(c echo.Context) error {
  return c.Redirect(http.StatusMovedPermanently, "<URL>")
}
```

Hooks

响应之前

`Context#Response#Before(func())` 可以用来注册在写入响应之前调用的函数。

响应之后

`Context#Response#After(func())` 可以用来注册在写入响应之后调用的函数。但是如果“Content-Length”是未知状态，则不会有任何方法会被执行。

示例

```
func(c echo.Context) error {
  c.Response().Before(func() {
    println("before response")
  })
  c.Response().After(func() {
    println("after response")
  })
  return c.NoContent(http.StatusNoContent)
}
```

> 可以在响应之前与之后注册多个方法

路由

路由

基于 radix tree ， Echo 的路由查询速度非常快。路由使用 sync pool 来重用内存，实现无 GC 开销下的零动态内存分配。

通过特定的 HTTP 方法，url 路径和一个匹配的处理程序 (handler) 可以注册一个路由。例如，下面的代码则展示了一个注册路由的例子：它包括 `Get` 的访问方式， `/hello` 的访问路径，以及发送 `Hello World` HTTP 响应的处理程序。

```
// 业务处理
func hello(c echo.Context) error {
      return c.String(http.StatusOK, "Hello, World!")
}

// 路由
e.GET("/hello", hello)
```

你可以用 `Echo.Any(path string, h Handler)` 来为所有的 HTTP 方法发送注册 处理程序 (handler)；如果仅需要为某个特定的方法注册路由，可使用 `Echo.Match(methods []string, path string, h Handler)` 。

Echo 通过 `func(echo.Context) error` 定义 handler 方法，其中 `echo.Context` 已经内嵌了 HTTP 请求和响应接口。

匹配所有

匹配零个或多个字符的路径。例如， `/users/*` 将会匹配；

- `/users/`
- `/users/1`
- `/users/1/files/1`
- `/users/anything...`

路径匹配顺序

- Static (固定路径)
- Param (参数路径)

- Match any (匹配所有)

实例

```
e.GET("/users/:id", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users/:id")
})

e.GET("/users/new", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users/new")
})

e.GET("/users/1/files/*", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users/1/files/*")
})
```

上面定义的路由将按下面的优先级顺序匹配:

- `/users/new`
- `/users/:id`
- `/users/1/files/*`

> 路由可以按照任意顺序定义。

组路由

`Echo#Group(prefix string, m ...Middleware) *Group`

可以将具有相同前缀的路由归为一组从而定义具有可选中间件的新子路由。除了一些特殊的中间件外，组路由也会继承父中间件。若要在组路由中添加中间件，则需使用 `Group.Use(m ...Middleware)` 。最后，组路由也可以嵌套。

下面的代码，我们创建了一个 admin 组，它需要对 `/admin/*` 路由进行基本的 HTTP 身份认证。

```
g := e.Group("/admin")
g.Use(middleware.BasicAuth(func(username, password string) bool {
    if username == "joe" && password == "secret" {
        return true
    }
    return false
}))
```

路由命名

每个路由都会返回一个 `Route` 对象，这个对象可以用来给路由命名。比如：

```
routeInfo := e.GET("/users/:id", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users/:id")
})
routeInfo.Name = "user"

// 或者这样写
e.GET("/users/new", func(c echo.Context) error {
    return c.String(http.StatusOK, "/users/new")
}).Name = "newuser"
```

当你需要在模版生成 uri 但是又无法获取路由的引用，或者多个路由使用相同的处理器(handler)的时候，路由命名就会显得更方便。

构造URI

`Echo#URI(handler HandlerFunc, params ...interface{})` 可以用来在任何业务处理代码里生成带有特殊参数的URI。这样在你重构自己的应用程序的时候，可以很方便的集中处理所有的 URI 。

下面的代码中 `e.URI(h, 1)` 将生成 `/users/1` ：

```
// 业务处理
h := func(c echo.Context) error {
    return c.String(http.StatusOK, "OK")
}

// 路由
e.GET("/users/:id", h)
```

除了 `Echo#URI` ，还可以使用 `Echo#Reverse(name string, params ...interface{})` 方法根据路由名生成 uri。比如，当 `foobar` 进行如下设置时，使用 `Echo#Reverse("foobar", 1234)` 就会生成 `/users/1234` ：

```
// Handler
h := func(c echo.Context) error {
    return c.String(http.StatusOK, "OK")
}

// Route
e.GET("/users/:id", h).Name = "foobar"
```

路由列表

`Echo#Routes() []*Route` 会根据路由定义的顺序列出所有已经注册的路由。每一个路由包含 http 方法，路径和对应的处理器(handler)。
示例

```
// Handlers
func createUser(c echo.Context) error {
}

func findUser(c echo.Context) error {
}

func updateUser(c echo.Context) error {
}

func deleteUser(c echo.Context) error {
}

// Routes
e.POST("/users", createUser)
e.GET("/users", findUser)
e.PUT("/users", updateUser)
e.DELETE("/users", deleteUser)
```

用下面的代码你将所有的路由信息输出到 JSON 文件：

```
data, err := json.MarshalIndent(e.Routes(), "", "  ")
if err != nil {
    return err
}
ioutil.WriteFile("routes.json", data, 0644)
```

`routes.json`

```
[
  {
    "method": "POST",
    "path": "/users",
    "handler": "main.createUser"
  },
  {
    "method": "GET",
```

```
    "path": "/users",
    "handler": "main.findUser"
  },
  {
    "method": "PUT",
    "path": "/users",
    "handler": "main.updateUser"
  },
  {
    "method": "DELETE",
    "path": "/users",
    "handler": "main.deleteUser"
  }
]
```

静态文件

静态文件

例如图片，JavaScript，CSS，PDF，字体文件等等...

使用静态中间件

使用 Echo#Static()

`Echo#Static(prefix, root string)` 使用路径前缀注册一个新路由，以便由根目录提供静态文件。

用法 1

```
e := echo.New()
e.Static("/static", "assets")
```

如上所示， assets 目录中 `/static/*` 路径下的任何文件都会被访问。例如，一个访问 `/static/js/main.js` 的请求会匹配到 `assets/js/main.js` 这个文件。

用法 2

```
e := echo.New()
e.Static("/", "assets")
```

如上所示， assets 目录中 `/*` 路径下的任何文件都会被访问。例如，一个访问 `/js/main.js` 的请求将会匹配到 `assets/js/main.js` 文件。

使用 Echo#File()

`Echo#File(path, file string)` 使用路径注册新路由以提供静态文件。

用法 1

使用 `public/index.html` 提供索引页面

```
e.File("/", "public/index.html")
```

用法 2

使用 `images/favicon.ico` 提供一个图标

```
e.File("/favicon.ico", "images/favicon.ico")
```

模板

模板

模板渲染

使用 `Context#Render(code int, name string, data interface{}) error` 命令渲染带有数据的模板，并发送带有状态代码的 text / html 响应。通过 `Echo.Renderer` 的设置我们可以使用任何模板引擎。

下面是使用 Go `html/template` 的示例：

1. 实现 `echo.Renderer` 接口

```
type Template struct {
    templates *template.Template
}

func (t *Template) Render(w io.Writer, name string, data interface{}, c echo.Context) error {
    return t.templates.ExecuteTemplate(w, name, data)
}
```

2. 预编译模板

`public/views/hello.html`

```
{{define "hello"}}Hello, {{.}}!{{end}}
```

```
t := &Template{
    templates: template.Must(template.ParseGlob("public/views/*.html")),
}
```

3. 声明模板

```
e := echo.New()
e.Renderer = t
e.GET("/hello", Hello)
```

4. 在 action 中渲染模板

```
func Hello(c echo.Context) error {
    return c.Render(http.StatusOK, "hello", "World")
}
```

高级 - 在模版中调用 Echo

在某些情况下，从模板生成 uri 可能很有用，为此，您需要从模板本身调用 `Echo#Reverse` 。此时，Golang 的 `html/template` 包并不一定合适这种情况，但我们可以通过两种方法实现它：第一种，给所有的传递到模版的对象提供一个公用的方法；第二种，将 `map[string]interface{}` 作为参数传递并在自定义渲染器中扩充此模版。鉴于后一种方法的灵活性，这里有一个示例程序：

`template.html`

```
<html>
    <body>
        <h1>Hello {{index . "name"}}</h1>

        <p>{{ with $x := index . "reverse" }}
            {{ call $x "foobar" }} &lt;-- this will call the $x with parameter "foobar"
            {{ end }}
        </p>
    </body>
</html>
```

`server.go`

```
package main

import (
    "html/template"
    "io"
    "log"
    "net/http"

    "github.com/labstack/echo"
)

// TemplateRenderer is a custom html/template renderer for Echo framework
type TemplateRenderer struct {
    templates *template.Template
}

// Render renders a template document
```

```
func (t *TemplateRenderer) Render(w io.Writer, name string, data interface{}, c
 echo.Context) error {

    // Add global methods if data is a map
    if viewContext, isMap := data.(map[string]interface{}); isMap {
        viewContext["reverse"] = c.Echo().Reverse
    }

    return t.templates.ExecuteTemplate(w, name, data)
}

func main() {
  e := echo.New()
  renderer := &TemplateRenderer{
      templates: template.Must(template.ParseGlob("*.html")),
  }
  e.Renderer = renderer

  // Named route "foobar"
  e.GET("/something", func(c echo.Context) error {
      return c.Render(http.StatusOK, "something.html", map[string]interface{}{
          "name": "Dolly!",
      })
  }).Name = "foobar"

  log.Fatal(e.Start(":8000"))
}
```

测试

测试

测试处理程序 (Testing handler)

`GET` `/users/:id`

下面的处理程序是根据用户的 id 从数据库取到该用户数据，如果用户不存在则返回 `404` 和提示语句。

创建 User

`POST` `/users`

- 接受 JSON 格式的关键信息
- 创建成功返回 `201 - Created`
- 发生错误返回 `500 - Internal Server Error`

获取 User

`GET` `/users/:email`

- 获取成功返回 `200 - OK`
- 未获取 User 返回 `404 - Not Found`
- 发生其它错误返回 `500 - Internal Server Error`

`handler.go`

```
package handler

import (
    "net/http"

    "github.com/labstack/echo"
)

type (
    User struct {
        Name  string `json:"name" form:"name"`
```

```
        Email string `json:"email" form:"email"`
    }
    handler struct {
        db map[string]*User
    }
)

func (h *handler) createUser(c echo.Context) error {
    u := new(User)
    if err := c.Bind(u); err != nil {
        return err
    }
    return c.JSON(http.StatusCreated, u)
}

func (h *handler) getUser(c echo.Context) error {
    email := c.Param("email")
    user := h.db[email]
    if user == nil {
        return echo.NewHTTPError(http.StatusNotFound, "user not found")
    }
    return c.JSON(http.StatusOK, user)
}
```

handler_test.go

```
package handler

import (
    "net/http"
    "net/http/httptest"
    "strings"
    "testing"

    "github.com/labstack/echo"
    "github.com/stretchr/testify/assert"
)

var (
    mockDB = map[string]*User{
        "jon@labstack.com": &User{"Jon Snow", "jon@labstack.com"},
    }
    userJSON = `{"name":"Jon Snow","email":"jon@labstack.com"}`
)
```

```
func TestCreateUser(t *testing.T) {
    // 设置
    e := echo.New()
    req := httptest.NewRequest(echo.POST, "/", strings.NewReader(userJSON))
    req.Header.Set(echo.HeaderContentType, echo.MIMEApplicationJSON)
    rec := httptest.NewRecorder()
    c := e.NewContext(req, rec)
    h := &handler{mockDB}

    // 断言
    if assert.NoError(t, h.createUser(c)) {
        assert.Equal(t, http.StatusCreated, rec.Code)
        assert.Equal(t, userJSON, rec.Body.String())
    }
}

func TestGetUser(t *testing.T) {
    // 设置
    e := echo.New()
    req := httptest.NewRequest(echo.GET, "/", nil)
    rec := httptest.NewRecorder()
    c := e.NewContext(req, rec)
    c.SetPath("/users/:email")
    c.SetParamNames("email")
    c.SetParamValues("jon@labstack.com")
    h := &handler{mockDB}

    // 断言
    if assert.NoError(t, h.getUser(c)) {
        assert.Equal(t, http.StatusOK, rec.Code)
        assert.Equal(t, userJSON, rec.Body.String())
    }
}
```

使用 Form 表单作为关键信息

```
f := make(url.Values)
f.Set("name", "Jon Snow")
f.Set("email", "jon@labstack.com")
req := httptest.NewRequest(echo.POST, "/", strings.NewReader(f.Encode()))
req.Header.Set(echo.HeaderContentType, echo.MIMEApplicationForm)
```

设置路径 (Path) 参数

```
c.SetParamNames("id", "email")
c.SetParamValues("1", "jon@labstack.com")
```

设置查询 (Query) 参数

```
q := make(url.Values)
q.Set("email", "jon@labstack.com")
req := http.NewRequest(echo.POST, "/?"+q.Encode(), nil)
```

测试中间件

待定
你可以在这里查看框架自带中间件的测试代码。

中间件

尾部斜杠

概述

中间件

中间件是一个函数，嵌入在HTTP 的请求和响应之间。它可以获得 `Echo#Context` 对象用来进行一些特殊的操作，
比如记录每个请求或者统计请求数。

Action的处理在所有的中间件运行完成之后。

中间件级别

Root Level (Before router)

`Echo#Pre()` 用于注册一个在路由执行之前运行的中间件，可以用来修改请求的一些属性。比如在请求路径结尾添加或者删除一个'/'来使之能与路由匹配。

下面的这几个内建中间件应该被注册在这一级别：

- AddTrailingSlash
- RemoveTrailingSlash
- MethodOverride

注意: 由于在这个级别路由还没有执行，所以这个级别的中间件不能调用任何 `echo.Context` 的 API。

Root Level (After router)

大部分时间你将用到 `Echo#Use()` 在这个级别注册中间件。
这个级别的中间件运行在路由处理完请求之后，可以调用所有的 `echo.Context` API。

下面的这几个内建中间件应该被注册在这一级别：

- BodyLimit
- Logger
- Gzip
- Recover
- BasicAuth
- JWTAuth

- Secure
- CORS
- Static

Group Level

当在路由中创建一个组的时候，可以为这个组注册一个中间件。例如，给 admin 这个组注册一个 BasicAuth 中间件。

用法

```
e := echo.New()
admin := e.Group("/admin", middleware.BasicAuth())
```

也可以在创建组之后用 `admin.Use()` 注册该中间件。

Route Level

当你创建了一个新的路由，可以选择性的给这个路由注册一个中间件。

用法

```
e := echo.New()
e.GET("/", <Handler>, <Middleware...>)
```

基本认证

Basic Auth (基本认证) 中间件

Basic Auth 中间件提供了 HTTP 的基本认证方式。

- 对于有效的请求，则继续调用下一个处理程序 (handler) 。
- 对于丢失或无效的请求，则返回 “401 - Unauthorized” 响应。

用法

```
e.Use(middleware.BasicAuth(func(username, password string, c echo.Context) (bool
, error) {
    if username == "joe" && password == "secret" {
        return true, nil
    }
    return false, nil
}))
```

自定义配置

用法

```
e.Use(middleware.BasicAuthWithConfig(middleware.BasicAuthConfig{}))
```

配置

```
BasicAuthConfig struct {
  // Skipper 定义了一个跳过中间件的函数
  Skipper Skipper

  // Validator 是一个用来验证 BasicAuth 是否合法的函数
  // Validator 是必须的.
  Validator BasicAuthValidator

  // Realm 是一个用来定义 BasicAuth 的 Realm 属性的字符串
  // 默认值是 "Restricted"
  Realm string
}
```

默认配置

```
DefaultBasicAuthConfig = BasicAuthConfig{
    Skipper: defaultSkipper,
}
```

请求体转储

Body Dump (请求体转储) 中间件

Body dump 中间件通常在调试 / 记录的情况下被使用，它可以捕获请求并调用已注册的处理程序 (handler) 响应有效负载。然而，当您的请求 / 响应有效负载很大时（例如上传 / 下载文件）需避免使用它；但如果避免不了，可在 skipper 函数中为端点添加异常。

用法

```
e := echo.New()
e.Use(middleware.BodyDump(func(c echo.Context, reqBody, resBody []byte) {
}))
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.BodyDumpWithConfig(middleware.BodyDumpConfig{}))
```

配置

```
BodyDumpConfig struct {
  // Skipper 定义了一个跳过中间件的函数
  Skipper Skipper

  // Handler 接收请求和响应有效负载
  // Required.
  Handler BodyDumpHandler
}
```

默认配置

```
DefaultBodyDumpConfig = BodyDumpConfig{
  Skipper: DefaultSkipper,
}
```

请求体限制

请求体限制

Body Limit (请求体限制) 中间件

Body Limit 中间件用于设置请求体的最大长度，如果请求体的大小超过了限制值，则返回 “413 — Request Entity Too Large” 响应。该限制的判断是根据 `Content-Length` 请求标头和实际内容确定的，这使其尽可能的保证安全。

限制可以指定 `4x` 或者 `4xB`，x是 “K, M, G, T, P” 计算机单位的倍数之一。

用法

```
e := echo.New()
e.Use(middleware.BodyLimit("2M"))
```

自定义配置

使用

```
e := echo.New()
e.Use(middleware.BodyLimitWithConfig(middleware.BodyLimitConfig{}))
```

配置

```
BodyLimitConfig struct {
  // Skipper 定义了一个跳过中间间的函数
  Skipper Skipper

  // 请求体被允许的最大值，可以被指定为类似“4x”和“4xB”这样的值，
  // x 是 K，M，G，T，P 计算机单位的倍数之一。
  Limit string `json:"limit"`
}
```

默认配置

```
DefaultBodyLimitConfig = BodyLimitConfig{
  Skipper: defaultSkipper,
}
```

CORS

CORS (跨域资源共享) 中间件

CORS (Cross-origin resource sharing) 中间件实现了 CORS 的标准。CORS为Web服务器提供跨域访问控制，从而实现安全的跨域数据传输。

用法

```
e.Use(middleware.CORS())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.CORSWithConfig(middleware.CORSConfig{
  AllowOrigins: []string{"https://labstack.com", "https://labstack.net"},
  AllowHeaders: []string{echo.HeaderOrigin, echo.HeaderContentType, echo.HeaderAccept},
}))
```

配置

```
// CORSConfig defines the config for CORS middleware.
CORSConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // AllowOrigin defines a list of origins that may access the resource.
  // Optional. Default value []string{"*"}.
  AllowOrigins []string `json:"allow_origins"`

  // AllowMethods defines a list methods allowed when accessing the resource.
  // This is used in response to a preflight request.
  // Optional. Default value DefaultCORSConfig.AllowMethods.
  AllowMethods []string `json:"allow_methods"`

  // AllowHeaders defines a list of request headers that can be used when
  // making the actual request. This in response to a preflight request.
```

```
  // Optional. Default value []string{}.
  AllowHeaders []string `json:"allow_headers"`

  // AllowCredentials indicates whether or not the response to the request
  // can be exposed when the credentials flag is true. When used as part of
  // a response to a preflight request, this indicates whether or not the
  // actual request can be made using credentials.
  // Optional. Default value false.
  AllowCredentials bool `json:"allow_credentials"`

  // ExposeHeaders defines a whitelist headers that clients are allowed to
  // access.
  // Optional. Default value []string{}.
  ExposeHeaders []string `json:"expose_headers"`

  // MaxAge indicates how long (in seconds) the results of a preflight request
  // can be cached.
  // Optional. Default value 0.
  MaxAge int `json:"max_age"`
}
```

默认配置

```
DefaultCORSConfig = CORSConfig{
  Skipper:      defaultSkipper,
  AllowOrigins: []string{"*"},
  AllowMethods: []string{echo.GET, echo.HEAD, echo.PUT, echo.PATCH, echo.POST, e
cho.DELETE},
}
```

CSRF

CSRF (跨域请求伪造) 中间件

CSRF (Cross-site request forgery) 跨域请求伪造，也被称为 **one-click attack** 或者 **session riding**，通常缩写为 **CSRF** 或者 **XSRF**， 是一种挟制用户在当前已登录的Web应用程序上执行非本意的操作的攻击方法。[1] 跟跨网站脚本 (XSS) 相比，**XSS** 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户网页浏览器的信任。

用法

```
e.Use(middleware.CSRF())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.CSRFWithConfig(middleware.CSRFConfig{
  TokenLookup: "header:X-XSRF-TOKEN",
}))
```

上面的例子使用 `X-XSRF-TOKEN` 请求头取出 CSRF 的 token 值。

获取 CSRF Token

服务器端

服务器端可以使用 `ContextKey` 从 `Echo#Context` 拿到 CSRF token 然后通过模版传给客户端。

客户端

客户端可以通过 CSRF cookie 拿到 token 值。

配置

```
// CSRFConfig defines the config for CSRF middleware.
CSRFConfig struct {
  // Skipper defines a function to skip middleware.
```

```
  Skipper Skipper

  // TokenLength is the length of the generated token.
  TokenLength uint8 `json:"token_length"`
  // Optional. Default value 32.

  // TokenLookup is a string in the form of "<source>:<key>" that is used
  // to extract token from the request.
  // Optional. Default value "header:X-CSRF-Token".
  // Possible values:
  // - "header:<name>"
  // - "form:<name>"
  // - "query:<name>"
  TokenLookup string `json:"token_lookup"`

  // Context key to store generated CSRF token into context.
  // Optional. Default value "csrf".
  ContextKey string `json:"context_key"`

  // Name of the CSRF cookie. This cookie will store CSRF token.
  // Optional. Default value "csrf".
  CookieName string `json:"cookie_name"`

  // Domain of the CSRF cookie.
  // Optional. Default value none.
  CookieDomain string `json:"cookie_domain"`

  // Path of the CSRF cookie.
  // Optional. Default value none.
  CookiePath string `json:"cookie_path"`

  // Max age (in seconds) of the CSRF cookie.
  // Optional. Default value 86400 (24hr).
  CookieMaxAge int `json:"cookie_max_age"`

  // Indicates if CSRF cookie is secure.
  // Optional. Default value false.
  CookieSecure bool `json:"cookie_secure"`

  // Indicates if CSRF cookie is HTTP only.
  // Optional. Default value false.
  CookieHTTPOnly bool `json:"cookie_http_only"`
}
```

默认配置

```
DefaultCSRFConfig = CSRFConfig{
  Skipper:      defaultSkipper,
  TokenLength:  32,
  TokenLookup:  "header:" + echo.HeaderXCSRFToken,
  ContextKey:   "csrf",
  CookieName:   "_csrf",
  CookieMaxAge: 86400,
}
```

Casbin 认证

Casbin Auth 中间件

Casbin 是 Go 下的强大而高效的开源访问控制库，它为基于各种模型的授权提供支持。到目前为止，Casbin 支持的访问控制模型如下：

- ACL (访问控制列表)
- 超级用户下的ACL
- 没有用户的 ACL： 对于没有身份验证或用户登录的系统尤其有用。
- 没有资源的ACL：过使用 write-article , read-log 等权限，某些方案可以应对一类资源而不是单个资源，它不控制对特定文章或日志的访问。
- RBAC (基于角色的访问控制)
- 具有资源角色的 RBAC： 用户和资源可以同时具有角色 (或组)。
- 具有域 / 租户 (tenants) 的 RBAC ：用户可以针对不同的域 / 租户 (tenants) 具有不同的角色集。
- ABAC (基于属性的访问控制)
- RESTful
- Deny-override：支持允许和拒绝授权，否认覆盖允许。

> Echo 社区贡献

依赖

```
import (
  "github.com/casbin/casbin"
  casbin_mw "github.com/labstack/echo-contrib/casbin"
)
```

用法

```
e := echo.New()
e.Use(casbin_mw.Middleware(casbin.NewEnforcer("casbin_auth_model.conf", "casbin_auth_policy.csv")))
```

有关语法，请参阅：Model.md。

自定义配置

用法

```
e := echo.New()
ce := casbin.NewEnforcer("casbin_auth_model.conf", "")
ce.AddRoleForUser("alice", "admin")
ce.AddPolicy(...)
e.Use(casbin_mw.MiddlewareWithConfig(casbin_mw.Config{
  Enforcer: ce,
}))
```

配置

```
// Config defines the config for CasbinAuth middleware.
Config struct {
  // Skipper defines a function to skip middleware.
  Skipper middleware.Skipper

  // Enforcer CasbinAuth main rule.
  // Required.
  Enforcer *casbin.Enforcer
}
```

默认配置

```
// DefaultConfig is the default CasbinAuth middleware config.
DefaultConfig = Config{
  Skipper: middleware.DefaultSkipper,
}
```

Gzip

Gzip 中间件

Gzip 中间件使用 gzip 方案来对 HTTP 响应进行压缩。

用法

```
e.Use(middleware.Gzip())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.GzipWithConfig(middleware.GzipConfig{
  Level: 5,
}))
```

配置

```
GzipConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Gzip compression level.
  // Optional. Default value -1.
  Level int `json:"level"`
}
```

默认配置

```
DefaultGzipConfig = GzipConfig{
  Skipper: defaultSkipper,
  Level:   -1,
}
```

JWT

JWT 中间件

JWT 提供了一个 JSON Web Token (JWT) 认证中间件。

- 对于有效的 token，它将用户置于上下文中并调用下一个处理程序。
- 对于无效的 token，它会发送 “401 - Unauthorized” 响应。
- 对于丢失或无效的 `Authorization` 标头，它会发送 “400 - Bad Request” 。

用法

```
e.Use(middleware.JWT([]byte("secret"))
```

自定义配置

用法

```
e.Use(middleware.JWTWithConfig(middleware.JWTConfig{
  SigningKey: []byte("secret"),
  TokenLookup: "query:token",
}))
```

配置

```
// JWTConfig defines the config for JWT middleware.
JWTConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Signing key to validate token.
  // Required.
  SigningKey interface{}

  // Signing method, used to check token signing method.
  // Optional. Default value HS256.
  SigningMethod string

  // Context key to store user information from the token into context.
```

```
  // Optional. Default value "user".
  ContextKey string

  // Claims are extendable claims data defining token content.
  // Optional. Default value jwt.MapClaims
  Claims jwt.Claims

  // TokenLookup is a string in the form of "<source>:<name>" that is used
  // to extract token from the request.
  // Optional. Default value "header:Authorization".
  // Possible values:
  // - "header:<name>"
  // - "query:<name>"
  // - "cookie:<name>"
  TokenLookup string

  // AuthScheme to be used in the Authorization header.
  // Optional. Default value "Bearer".
  AuthScheme string
}
```

默认配置

```
DefaultJWTConfig = JWTConfig{
  Skipper:       defaultSkipper,
  SigningMethod: AlgorithmHS256,
  ContextKey:    "user",
  TokenLookup:   "header:" + echo.HeaderAuthorization,
  AuthScheme:    "Bearer",
  Claims:        jwt.MapClaims{},
}
```

示例

密钥认证

Key Auth (密钥认证) 中间件

Key Auth 中间件提供了一个基于密钥的验证方式。

- 对于有效密钥，它将调用下一个处理程序。
- 对于无效密钥，它会发送 “401 - Unauthorized” 响应。
- 对于丢失密钥，它发送 “400 - Bad Request” 响应。

用法

```
e.Use(middleware.KeyAuth(func(key string) bool {
  return key == "valid-key"
}))
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.KeyAuthWithConfig(middleware.KeyAuthConfig{
  KeyLookup: "query:api-key",
}))
```

配置

```
// KeyAuthConfig defines the config for KeyAuth middleware.
KeyAuthConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // KeyLookup is a string in the form of "<source>:<name>" that is used
  // to extract key from the request.
  // Optional. Default value "header:Authorization".
  // Possible values:
  // - "header:<name>"
  // - "query:<name>"
  KeyLookup string `json:"key_lookup"`
```

```
  // AuthScheme to be used in the Authorization header.
  // Optional. Default value "Bearer".
  AuthScheme string

  // Validator is a function to validate key.
  // Required.
  Validator KeyAuthValidator
}
```

默认配置

```
DefaultKeyAuthConfig = KeyAuthConfig{
  Skipper:    defaultSkipper,
  KeyLookup:  "header:" + echo.HeaderAuthorization,
  AuthScheme: "Bearer",
}
```

日志

Logger (日志) 中间件

Logger 中间件记录有关每个 HTTP 请求的信息。

用法

```
e.Use(middleware.Logger())
```

输出样例

```
{"time":"2017-01-12T08:58:07.372015644-08:00","remote_ip":"::1","host":"localhost:1323","method":"GET","uri":"/","status":200, "latency":14743,"latency_human":"14.743µs","bytes_in":0,"bytes_out":2}
```

自定义配置

用法

```
e.Use(middleware.LoggerWithConfig(middleware.LoggerConfig{
  Format: "method=${method}, uri=${uri}, status=${status}\n",
}))
```

上面的示例使用 `Format` 来记录请求方法和请求 URI 。

输出样例

```
method=GET, uri=/, status=200
```

配置

```
LoggerConfig struct {
  // Skipper 定义了一个跳过中间件的函数.
  Skipper Skipper

  // 日志的格式可以使用下面的标签定义。:
  //
  // - time_unix
```

```
  // - time_unix_nano
  // - time_rfc3339
  // - time_rfc3339_nano
  // - id (Request ID - Not implemented)
  // - remote_ip
  // - uri
  // - host
  // - method
  // - path
  // - referer
  // - user_agent
  // - status
  // - latency (In microseconds)
  // - latency_human (Human readable)
  // - bytes_in (Bytes received)
  // - bytes_out (Bytes sent)
  // - header:<name>
  // - query:<name>
  // - form:<name>
  //
  // 例如 "${remote_ip} ${status}"
  //
  // 可选。默认值是 DefaultLoggerConfig.Format.
  Format string `json:"format"`

  // Output 是记录日志的位置。
  // 可选。默认值是 os.Stdout.
  Output io.Writer
}
```

默认配置

```
DefaultLoggerConfig = LoggerConfig{
  Skipper: defaultSkipper,
  Format: `{"time":"${time_rfc3339_nano}","remote_ip":"${remote_ip}","host":"${h
ost}",` +
    `"method":"${method}","uri":"${uri}","status":${status}, "latency":${latenc
y},` +
    `"latency_human":"${latency_human}","bytes_in":${bytes_in},` +
    `"bytes_out":${bytes_out}}` + "\n",
  Output: os.Stdout
}
```

更多细节见：golang echo 代码详解之 log 篇

方法重写

(方法重写) 中间件

Method Override 中间件检查从请求中重写的方法，并使用它来代替原来的方法。

出于安全原因，只有 `POST` 方法可以被重写。

用法

```
e.Pre(middleware.MethodOverride())
```

自定义配置

用法

```
e := echo.New()
e.Pre(middleware.MethodOverrideWithConfig(middleware.MethodOverrideConfig{
  Getter: middleware.MethodFromForm("_method"),
}))
```

配置

```
MethodOverrideConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Getter is a function that gets overridden method from the request.
  // Optional. Default values MethodFromHeader(echo.HeaderXHTTPMethodOverride).
  Getter MethodOverrideGetter
}
```

默认配置

```
DefaultMethodOverrideConfig = MethodOverrideConfig{
  Skipper: defaultSkipper,
  Getter:  MethodFromHeader(echo.HeaderXHTTPMethodOverride),
}
```

代理

Proxy (代理) 中间件

Proxy 提供 HTTP / WebSocket 反向代理中间件。它使用已配置的负载平衡技术将请求转发到上游服务器。

用法

```
url1, err := url.Parse("http://localhost:8081")
if err != nil {
  e.Logger.Fatal(err)
}
url2, err := url.Parse("http://localhost:8082")
if err != nil {
  e.Logger.Fatal(err)
}
e.Use(middleware.Proxy(&middleware.RoundRobinBalancer{
  Targets: []*middleware.ProxyTarget{
    {
      URL: url1,
    },
    {
      URL: url2,
    },
  },
}))
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.ProxyWithConfig(middleware.ProxyConfig{}))
```

配置

```
// ProxyConfig defines the config for Proxy middleware.
ProxyConfig struct {
  // Skipper defines a function to skip middleware.
```

```
  Skipper Skipper

  // Balancer defines a load balancing technique.
  // Required.
  // Possible values:
  // - RandomBalancer
  // - RoundRobinBalancer
  Balancer ProxyBalancer
}
```

默认配置

名称	值
Skipper	DefaultSkipper

示例

恢复

Recover (恢复) 中间件

Recover 中间件从 panic 链中的任意位置恢复程序， 打印堆栈的错误信息，并将错误集中交给
HTTPErrorHandler 处理。

用法

```
e.Use(middleware.Recover())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.RecoverWithConfig(middleware.RecoverConfig{
  StackSize:  1 << 10, // 1 KB
}))
```

上面的示例使用了 1 kb 的 `StackSize` 和默认值的 `DisableStackAll` 与 `DisablePrintStack` 。

配置

```
RecoverConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Size of the stack to be printed.
  // Optional. Default value 4KB.
  StackSize int `json:"stack_size"`

  // DisableStackAll disables formatting stack traces of all other goroutines
  // into buffer after the trace for the current goroutine.
  // Optional. Default value false.
  DisableStackAll bool `json:"disable_stack_all"`

  // DisablePrintStack disables printing stack trace.
```

```
  // Optional. Default value as false.
  DisablePrintStack bool `json:"disable_print_stack"`
}
```

默认配置

```
DefaultRecoverConfig = RecoverConfig{
  Skipper:           defaultSkipper,
  StackSize:         4 << 10, // 4 KB
  DisableStackAll:   false,
  DisablePrintStack: false,
}
```

重定向

Redirect (重定向) 中间件

HTTPS 重定向

HTTPS 重定向中间件将 http 请求重定向到 https。例如，http://laily.net 将被重定向到 https://laily.net。

用法

```
e := echo.New()
e.Pre(middleware.HTTPSRedirect())
```

HTTPS WWW 重定向

HTTPS WWW 重定向将 http 请求重定向到带 www 的 https 请求。例如，http://laily.net 将被重定向到 https://www.laily.net。

用法

```
e := echo.New()
e.Pre(middleware.HTTPSWWWRedirect())
```

HTTPS NonWWW 重定向

HTTPS NonWWW 将 http 请求重定向到不带 www 的 https 请求。例如，http://www.laily.net 将被重定向到 https://laily.net。

用法

```
e := echo.New()
e.Pre(middleware.HTTPSNonWWWRedirect())
```

WWW 重定向

WWW 重定向将不带 www 的请求重定向到带 www 的请求。

例如，http://laily.net 重定向到 http://www.laily.net

用法

```
e := echo.New()
e.Pre(middleware.WWWRedirect())
```

NonWWW 重定向

NonWWW 重定向将带 www 的请求重定向到不带 www 的请求。

例如，http://www.laily.net 重定向到 http://laily.net

用法

```
e := echo.New()
e.Pre(middleware.NonWWWRedirect())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.HTTPSRedirectWithConfig(middleware.RedirectConfig{
  Code: http.StatusTemporaryRedirect,
}))
```

上面的示例将 HTTP 的请求重定向到 HTTPS，使用 `307 - StatusTemporaryRedirect` 状态码跳转。

配置

```
RedirectConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Status code to be used when redirecting the request.
  // Optional. Default value http.StatusMovedPermanently.
  Code int `json:"code"`
}
```

默认配置

```
DefaultRedirectConfig = RedirectConfig{
  Skipper: defaultSkipper,
```

```
  Code:    http.StatusMovedPermanently,
}
```

请求ID

Request ID (请求ID) 中间件

Request ID 中间件为请求生成唯一的 ID。

用法

```
e.Use(middleware.RequestID())
```

自定义配置

用法

```
e.Use(middleware.RequestIDWithConfig(middleware.RequestIDConfig{
  Generator: func() string {
    return customGenerator()
  },
}))
```

配置

```
RequestIDConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Generator defines a function to generate an ID.
  // Optional. Default value random.String(32).
  Generator func() string
}
```

默认配置

```
DefaultRequestIDConfig = RequestIDConfig{
  Skipper:   DefaultSkipper,
  Generator: generator,
}
```

重写

Rewrite (重写) 中间件

Rewrite 中间件会根据提供的规则重写URL路径，它更适用于向后兼容与创建更清晰、更具描述性的链接。

用法

```
e.Pre(middleware.Rewrite(map[string]string{
  "/old":              "/new",
  "/api/*":            "/$1",
  "/js/*":             "/public/javascripts/$1",
  "/users/*/orders/*": "/user/$1/order/$2",
}))
```

星号中捕获的值可以通过索引检索，例如 `$1`, `$2` 等等。

Custom Configuration

用法

```
e := echo.New()
e.Pre(middleware.RewriteWithConfig(middleware.RewriteConfig{}))
```

配置

```
// RewriteConfig defines the config for Rewrite middleware.
RewriteConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Rules defines the URL path rewrite rules.
  Rules map[string]string `yaml:"rules"`
}
```

默认配置

名称	值

名称	值
Skipper	DefaultSkipper

> 重写中间件应该在路由之前通过 Echo#Pre() 注册从而触发。

安全

Secure (安全) 中间件

Secure 中间件用于阻止跨站脚本攻击(XSS)，内容嗅探，点击劫持，不安全链接等其他代码注入攻击。

使用

```
e.Use(middleware.Secure())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.SecureWithConfig(middleware.SecureConfig{
    XSSProtection:         "",
    ContentTypeNosniff:    "",
    XFrameOptions:         "",
    HSTSMaxAge:            3600,
    ContentSecurityPolicy: "default-src 'self'",
}))
```

传递空的 `XSSProtection` ， `ContentTypeNosniff` ， `XFrameOptions` 或 `ContentSecurityPolicy` 来禁用这项保护。

配置

```
SecureConfig struct {
  // Skipper 定义了一个跳过该中间件的函数。
  Skipper Skipper

  // XSSProtection 通过设置`X-XSS-Protection`头
  // 来提供XSS攻击的防护。
  // 可选。默认值 "1; mode=block"。
  XSSProtection string `json:"xss_protection"`

  // ContentTypeNosniff 通过设置`X-Content-Type-Options`头
  // 来防止内容嗅探。
```

```
    // 可选。默认值 "nosniff"。
    ContentTypeNosniff string `json:"content_type_nosniff"`

    // XFrameOptions 被用来指示是否允许浏览器在<fram>, <iframe>或者<object>中渲染页面。
    // 网站可以通过这样来避免点击劫持，保证网站的内容不会被其他站点嵌入。
    // 可选。默认值 "SAMEORIGIN".
    // 可使用的值:
    // `SAMEORIGIN` - 页面只能在同域名的页面下被渲染。
    // `DENY` - 页面不允许在 frame 中显示。
    // `ALLOW-FROM uri` - 页面只能在指定域名的 frame 中显示。
    XFrameOptions string `json:"x_frame_options"`

    // HSTSMaxAge 设置 `Strict-Transport-Security` 头来指示浏览器需要记住这个网站只能通过HTTPS来访问的时间(单位秒)。
    // 这样可以减少遭受中间人攻击(HITM)的几率。
    // 可选。默认值 0。
    HSTSMaxAge int `json:"hsts_max_age"`

    // HSTSExcludeSubdomains 不会在`Strict Transport Security`中设置`includeSubdomains`标签。
    // 即从安全规则中排除所有子域名。
    // header, excluding all subdomains from security policy. It has no effect
    // 只有在HSTSMaxAge 被设置为非0值时该参数才有效。
    // 可选。默认值 false。
    HSTSExcludeSubdomains bool `json:"hsts_exclude_subdomains"`

    // ContentSecurityPolicy 用来设置 `Content-Security-Policy` 头。
    // `Content-Security-Policy` 用来定义页面可以加载哪些资源，减少XSS等通过运行不安全代码的注入攻击。
    // 可选。默认值 "".
    ContentSecurityPolicy string `json:"content_security_policy"`
}
```

默认配置

```
DefaultSecureConfig = SecureConfig{
    Skipper:             defaultSkipper,
    XSSProtection:       "1; mode=block",
    ContentTypeNosniff: "nosniff",
    XFrameOptions:       "SAMEORIGIN",
}
```

会话

Session (会话) 中间件

Session 中间件促进了 gorilla/sessions 支持的 HTTP 会话管理。默认提供了基于 cookie 与文件系统的会话存储；然而，你也可以访问 community maintained implementation 来参考其各式各样的后端实现。

> Echo 社区贡献

依赖

```
import (
  "github.com/gorilla/sessions"
  "github.com/labstack/echo-contrib/session"
)
```

用法

```
e := echo.New()
e.Use(session.Middleware(sessions.NewCookieStore([]byte("secret"))))

e.GET("/", func(c echo.Context) error {
  sess, _ := session.Get("session", c)
  sess.Options = &sessions.Options{
    Path:     "/",
    MaxAge:   86400 * 7,
    HttpOnly: true,
  }
  sess.Values["foo"] = "bar"
  sess.Save(c.Request(), c.Response())
  return c.NoContent(http.StatusOK)
})
```

Custom Configuration

用法

```
e := echo.New()
e.Use(session.MiddlewareWithConfig(session.Config{}))
```

配置

```
Config struct {
  // Skipper defines a function to skip middleware.
  Skipper middleware.Skipper

  // Session store.
  // Required.
  Store sessions.Store
}
```

默认配置

```
DefaultConfig = Config{
  Skipper: DefaultSkipper,
}
```

静态

Static (静态) 中间件

Static 中间件可已被用于服务从根目录中提供的静态文件。

用法

```
e := echo.New()
e.Use(middleware.Static("/static"))
```

上例为 `static` 目录下的静态文件提供服务。例如， 一个 `/js/main.js` 的请求将捕获并服务 `static/js/main.js` 文件。

自定义配置

用法

```
e := echo.New()
e.Use(middleware.StaticWithConfig(middleware.StaticConfig{
  Root:   "static",
  Browse: true,
}))
```

上例为 `static` 目录下的静态文件提供服务并启用目录浏览。

配置

```
StaticConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper

  // Root directory from where the static content is served.
  // Required.
  Root string `json:"root"`

  // Index file for serving a directory.
  // Optional. Default value "index.html".
  Index string `json:"index"`

  // Enable HTML5 mode by forwarding all not-found requests to root so that
```

```
  // SPA (single-page application) can handle the routing.
  // Optional. Default value false.
  HTML5 bool `json:"html5"`

  // Enable directory browsing.
  // Optional. Default value false.
  Browse bool `json:"browse"`
}
```

默认配置

```
DefaultStaticConfig = StaticConfig{
  Skipper: DefaultSkipper,
  Index:   "index.html",
}
```

尾部斜杠

Trailing Slash (尾部斜杠) 中间件

添加尾部斜杠

Add trailing slash 中间件会在在请求的 URI 后加上反斜杠

用法

```
e := echo.New()
e.Pre(middleware.AddTrailingSlash())
```

去除尾部斜杠

Remove trailing slash 中间件在请求的 uri 后去除反斜杠

用法

```
e := echo.New()
e.Pre(middleware.RemoveTrailingSlash())
```

自定义配置

用法

```
e := echo.New()
e.Use(middleware.AddTrailingSlashWithConfig(middleware.TrailingSlashConfig{
  RedirectCode: http.StatusMovedPermanently,
}))
```

这个示例将向请求 URI 添加一个尾部斜杠，并使用 `301 - StatusMovedPermanently` 重定向。

配置

```
TrailingSlashConfig struct {
  // Skipper defines a function to skip middleware.
  Skipper Skipper
```

```
  // Status code to be used when redirecting the request.
  // Optional, but when provided the request is redirected using this code.
  RedirectCode int `json:"redirect_code"`
}
```

默认配置

```
DefaultTrailingSlashConfig = TrailingSlashConfig{
  Skipper: defaultSkipper,
}
```

菜谱

Hello World

Hello World 示例

服务器

server.go

```
package main

import (
    "net/http"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func main() {
    // Echo instance
    e := echo.New()

    // Middleware
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    // Route => handler
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Hello, World!\n")
    })

    // Start server
    e.Logger.Fatal(e.Start(":1323"))
}
```

Auto TLS

Auto TLS

这个例子演示如何自动从 Let's Encrypt 获得 TLS 证书。 `Echo#StartAutoTLS` 接受一个接听 443 端口的网络地址。类似 `<DOMAIN>:443` 这样。

如果没有错误，访问 `https://<DOMAIN>` ，可以看到一个 TLS 加密的欢迎界面。

服务器

`server.go`

```
package main

import (
    "net/http"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func main() {
    e := echo.New()
    // e.AutoTLSManager.HostPolicy = autocert.HostWhitelist("<your_domain>")
    // Store the certificate to avoid issues with rate limits (https://letsencrypt.org/docs/rate-limits/)
    // e.AutoTLSManager.Cache = autocert.DirCache("<path to store key and certificate>")
    e.Use(middleware.Recover())
    e.Use(middleware.Logger())
    e.GET("/", func(c echo.Context) error {
        return c.HTML(http.StatusOK, `
            <h1>Welcome to Echo!</h1>
            <h3>TLS certificates automatically installed from Let's Encrypt :)</h3>
        `)
    })
    e.Logger.Fatal(e.StartAutoTLS(":443"))
}
```

CRUD

CRUD 示例

服务端

server.go

```
package main

import (
    "net/http"
    "strconv"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

type (
    user struct {
        ID   int    `json:"id"`
        Name string `json:"name"`
    }
)

var (
    users = map[int]*user{}
    seq   = 1
)

//----------
// Handlers
//----------

func createUser(c echo.Context) error {
    u := &user{
        ID: seq,
    }
    if err := c.Bind(u); err != nil {
        return err
    }
    users[u.ID] = u
```

```
    seq++
    return c.JSON(http.StatusCreated, u)
}

func getUser(c echo.Context) error {
    id, _ := strconv.Atoi(c.Param("id"))
    return c.JSON(http.StatusOK, users[id])
}

func updateUser(c echo.Context) error {
    u := new(user)
    if err := c.Bind(u); err != nil {
        return err
    }
    id, _ := strconv.Atoi(c.Param("id"))
    users[id].Name = u.Name
    return c.JSON(http.StatusOK, users[id])
}

func deleteUser(c echo.Context) error {
    id, _ := strconv.Atoi(c.Param("id"))
    delete(users, id)
    return c.NoContent(http.StatusNoContent)
}

func main() {
    e := echo.New()

    // Middleware
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    // Routes
    e.POST("/users", createUser)
    e.GET("/users/:id", getUser)
    e.PUT("/users/:id", updateUser)
    e.DELETE("/users/:id", deleteUser)

    // Start server
    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

curl

创建 User

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -d '{"name":"Joe Smith"}' \
  localhost:1323/users
```

Response

```
{
  "id": 1,
  "name": "Joe Smith"
}
```

获取 User

```
curl localhost:1323/users/1
```

Response

```
{
  "id": 1,
  "name": "Joe Smith"
}
```

更新 User

```
curl -X PUT \
  -H 'Content-Type: application/json' \
  -d '{"name":"Joe"}' \
  localhost:1323/users/1
```

Response

```
{
  "id": 1,
  "name": "Joe"
}
```

删除 User

```
curl -X DELETE localhost:1323/users/1
```

Response

```
NoContent - 204
```

twitter

类似 Twitter 的 API 服务

这个示例演示如何使用 MongoDB，JWT 和 JSON 创建一个类似 Twitter 的 REST API 服务。

模型

user.go

```
package model

import "gopkg.in/mgo.v2/bson"

type (
    User struct {
        ID        bson.ObjectId `json:"id" bson:"_id,omitempty"`
        Email     string        `json:"email" bson:"email"`
        Password  string        `json:"password,omitempty" bson:"password"`
        Token     string        `json:"token,omitempty" bson:"-"`
        Followers []string      `json:"followers,omitempty" bson:"followers,omitempty"`
    }
)
```

post.go

```
package model

import "gopkg.in/mgo.v2/bson"

type (
    Post struct {
        ID      bson.ObjectId `json:"id" bson:"_id,omitempty"`
        To      string        `json:"to" bson:"to"`
        From    string        `json:"from" bson:"from"`
        Message string        `json:"message" bson:"message"`
    }
)
```

控制器

handler.go

```
package handler

import mgo "gopkg.in/mgo.v2"

type (
    Handler struct {
        DB *mgo.Session
    }
)

const (
    // Key (Should come from somewhere else).
    Key = "secret"
)
```

user.go

```
package handler

import (
    "net/http"
    "time"

    jwt "github.com/dgrijalva/jwt-go"
    "github.com/labstack/echo"
    "github.com/labstack/echo/cookbook/twitter/model"
    mgo "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

func (h *Handler) Signup(c echo.Context) (err error) {
    // Bind
    u := &model.User{ID: bson.NewObjectId()}
    if err = c.Bind(u); err != nil {
        return
    }

    // Validate
    if u.Email == "" || u.Password == "" {
        return &echo.HTTPError{Code: http.StatusBadRequest, Message: "invalid em
ail or password"}
    }
```

```
    // Save user
    db := h.DB.Clone()
    defer db.Close()
    if err = db.DB("twitter").C("users").Insert(u); err != nil {
        return
    }

    return c.JSON(http.StatusCreated, u)
}

func (h *Handler) Login(c echo.Context) (err error) {
    // Bind
    u := new(model.User)
    if err = c.Bind(u); err != nil {
        return
    }

    // Find user
    db := h.DB.Clone()
    defer db.Close()
    if err = db.DB("twitter").C("users").
        Find(bson.M{"email": u.Email, "password": u.Password}).One(u); err != ni
l {
        if err == mgo.ErrNotFound {
            return &echo.HTTPError{Code: http.StatusUnauthorized, Message: "inva
lid email or password"}
        }
        return
    }

    //-----
    // JWT
    //-----

    // Create token
    token := jwt.New(jwt.SigningMethodHS256)

    // Set claims
    claims := token.Claims.(jwt.MapClaims)
    claims["id"] = u.ID
    claims["exp"] = time.Now().Add(time.Hour * 72).Unix()

    // Generate encoded token and send it as response
    u.Token, err = token.SignedString([]byte(Key))
    if err != nil {
        return err
```

```
    }

    u.Password = "" // Don't send password
    return c.JSON(http.StatusOK, u)
}

func (h *Handler) Follow(c echo.Context) (err error) {
    userID := userIDFromToken(c)
    id := c.Param("id")

    // Add a follower to user
    db := h.DB.Clone()
    defer db.Close()
    if err = db.DB("twitter").C("users").
        UpdateId(bson.ObjectIdHex(id), bson.M{"$addToSet": bson.M{"followers": u
serID}}); err != nil {
        if err == mgo.ErrNotFound {
            return echo.ErrNotFound
        }
    }

    return
}

func userIDFromToken(c echo.Context) string {
    user := c.Get("user").(*jwt.Token)
    claims := user.Claims.(jwt.MapClaims)
    return claims["id"].(string)
}
```

post.go

```
package handler

import (
    "net/http"
    "strconv"

    "github.com/labstack/echo"
    "github.com/labstack/echo/cookbook/twitter/model"
    mgo "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

func (h *Handler) CreatePost(c echo.Context) (err error) {
```

```
    u := &model.User{
        ID: bson.ObjectIdHex(userIDFromToken(c)),
    }
    p := &model.Post{
        ID:   bson.NewObjectId(),
        From: u.ID.Hex(),
    }
    if err = c.Bind(p); err != nil {
        return
    }

    // Validation
    if p.To == "" || p.Message == "" {
        return &echo.HTTPError{Code: http.StatusBadRequest, Message: "invalid to
or message fields"}
    }

    // Find user from database
    db := h.DB.Clone()
    defer db.Close()
    if err = db.DB("twitter").C("users").FindId(u.ID).One(u); err != nil {
        if err == mgo.ErrNotFound {
            return echo.ErrNotFound
        }
        return
    }

    // Save post in database
    if err = db.DB("twitter").C("posts").Insert(p); err != nil {
        return
    }
    return c.JSON(http.StatusCreated, p)
}

func (h *Handler) FetchPost(c echo.Context) (err error) {
    userID := userIDFromToken(c)
    page, _ := strconv.Atoi(c.QueryParam("page"))
    limit, _ := strconv.Atoi(c.QueryParam("limit"))

    // Defaults
    if page == 0 {
        page = 1
    }
    if limit == 0 {
        limit = 100
    }
```

```
    // Retrieve posts from database
    posts := []*model.Post{}
    db := h.DB.Clone()
    if err = db.DB("twitter").C("posts").
        Find(bson.M{"to": userID}).
        Skip((page - 1) * limit).
        Limit(limit).
        All(&posts); err != nil {
        return
    }
    defer db.Close()

    return c.JSON(http.StatusOK, posts)
}
```

API

注册

用户注册

- 用请求里取出用户信息验证合法性。
- 不合法的邮箱和密码，返回 `400 - Bad Request` 。
- 合法的邮箱和密码，保存数据到数据库并返回 `201 - Created` 。

请求

```
curl \
  -X POST \
  http://localhost:1323/signup \
  -H "Content-Type: application/json" \
  -d '{"email":"jon@labstack.com","password":"shhh!"}'
```

响应

`201 - Created`

```
{
  "id": "58465b4ea6fe886d3215c6df",
  "email": "jon@labstack.com",
  "password": "shhh!"
}
```

Login

User login

- Retrieve user credentials from the body and validate against database.
- For invalid credentials, send `401 - Unauthorized` response.
- For valid credentials, send `200 - OK` response:
  - Generate JWT for the user and send it as response.
  - Each subsequent request must include JWT in the `Authorization` header.

Method: `POST`

Path: `/login`

Request

```
curl \
  -X POST \
  http://localhost:1323/login \
  -H "Content-Type: application/json" \
  -d '{"email":"jon@labstack.com","password":"shhh!"}'
```

Response

`200 - OK`

```
{
  "id": "58465b4ea6fe886d3215c6df",
  "email": "jon@labstack.com",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0ODEyNjUxMjgsImlkIjoiNTg0NjViNGVhNmZlODg2ZDMyMTVjNmRmIn0.1IsGGxko1qMCsKkJDQ1NfmrZ945XVC9uZpcvDnKwpL0"
}
```

Client should store the token, for browsers, you may use local storage.

Follow

Follow a user

- For invalid token, send `400 - Bad Request` response.

- For valid token:
  - If user is not found, send `404 - Not Found` response.
  - Add a follower to the specified user in the path parameter and send `200 - OK` response.

Method: `POST`

Path: `/follow/:id`

Request

```
curl \
  -X POST \
  http://localhost:1323/follow/58465b4ea6fe886d3215c6df \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0ODEyNjUxMjgsImlkIjoiNTg0NjViNGVhNmZlODg2ZDMyMTVjNmRmIn0.1IsGGxko1qMCsKkJDQ1NfmrZ945XVC9uZpcvDnKwpL0"
```

Response

`200 - OK`

Post

Post a message to specified user

- For invalid request payload, send `400 - Bad Request` response.
- If user is not found, send `404 - Not Found` response.
- Otherwise save post in the database and return it via `201 - Created` response.

Method: `POST`

Path: `/posts`

Request

```
curl \
  -X POST \
  http://localhost:1323/posts \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0ODE
```

```
yNjUxMjgsImlkIjoiNTg0NjViNGVhNmZlODg2ZDMyMTVjNmRmIn0.1IsGGxko1qMCsKkJDQ1NfmrZ945XVC9uZpcvDnKwpL0" \
  -H "Content-Type: application/json" \
  -d '{"to":"58465b4ea6fe886d3215c6df","message":"hello"}'
```

Response

`201 - Created`

```
{
  "id": "584661b9a6fe8871a3804cba",
  "to": "58465b4ea6fe886d3215c6df",
  "from": "58465b4ea6fe886d3215c6df",
  "message": "hello"
}
```

Feed

List most recent messages based on optional `page` and `limit` query parameters

Method: `GET`

Path: `/feed?page=1&limit=5`

Request

```
curl \
  -X GET \
  http://localhost:1323/feed \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0ODEyNjUxMjgsImlkIjoiNTg0NjViNGVhNmZlODg2ZDMyMTVjNmRmIn0.1IsGGxko1qMCsKkJDQ1NfmrZ945XVC9uZpcvDnKwpL0"
```

Response

`200 - OK`

```
[
  {
    "id": "584661b9a6fe8871a3804cba",
    "to": "58465b4ea6fe886d3215c6df",
    "from": "58465b4ea6fe886d3215c6df",
```

```
    "message": "hello"
  }
]
```

HTTP2

HTTP2

HTTP/2 (原本的名字是 HTTP/2.0) 是万维网使用的 HTTP 网络协议的第二个主要版本。
HTTP/2 提供了更快的速度和更好的用户体验。

特性

- 使用二进制格式传输数据，而不是文本。使得在解析和优化扩展上更为方便。
- 多路复用，所有的请求都是通过一个 TCP 连接并发完成。
- 对消息头采用 HPACK 进行压缩传输，能够节省消息头占用的网络的流量。
- Server Push：服务端能够更快的把资源推送给客户端。

怎样运行 HTTP2 和 HTTPS 服务?

生成一个自签名的 X.509 TLS 证书(HTTP/2 需要 TLS 才能运行)

```
go run $GOROOT/src/crypto/tls/generate_cert.go --host localhost
```

上面的命令会生一个 `cert.pem` 和 `key.pem` 文件。

> 这里只是展示使用，所以我们用了自签名的证书，正式环境建议去CA申请证书。

配置服务器引擎 `engine.Config`

`server.go`

```
package main

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

```
    "github.com/labstack/echo"
)

func request(c echo.Context) error {
    req := c.Request()
    format := "<pre><strong>Request Information</strong>\n\n<code>Protocol: %s\n
Host: %s\nRemote Address: %s\nMethod: %s\nPath: %s\n</code></pre>"
    return c.HTML(http.StatusOK, fmt.Sprintf(format, req.Proto, req.Host, req.Re
moteAddr, req.Method, req.URL.Path))
}

func stream(c echo.Context) error {
    res := c.Response()
    gone := res.CloseNotify()
    res.Header().Set(echo.HeaderContentType, echo.MIMETextHTMLCharsetUTF8)
    res.WriteHeader(http.StatusOK)
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    fmt.Fprint(res, "<pre><strong>Clock Stream</strong>\n\n<code>")
    for {
        fmt.Fprintf(res, "%v\n", time.Now())
        res.Flush()
        select {
        case <-ticker.C:
        case <-gone:
            break
        }
    }
}

func main() {
    e := echo.New()
    e.GET("/request", request)
    e.GET("/stream", stream)
    e.Logger.Fatal(e.StartTLS(":1323", "cert.pem", "key.pem"))
}
```

最后

- https://localhost:1323/request (显示 HTTP 请求信息)
- https://localhost:1323/stream (实时展示当前时间)

中间件

中间件

怎样自己写一个中间件？

比如有以下需求

- 通过该中间件去统计请求数目、状态和时间。
- 中间件把写过写回响应。

Server

server.go

```
package main

import (
    "net/http"
    "strconv"
    "sync"
    "time"

    "github.com/labstack/echo"
)

type (
    Stats struct {
        Uptime       time.Time       `json:"uptime"`
        RequestCount uint64          `json:"requestCount"`
        Statuses     map[string]int `json:"statuses"`
        mutex        sync.RWMutex
    }
)

func NewStats() *Stats {
    return &Stats{
        Uptime:   time.Now(),
        Statuses: make(map[string]int),
    }
}
```

```
// Process is the middleware function.
func (s *Stats) Process(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        if err := next(c); err != nil {
            c.Error(err)
        }
        s.mutex.Lock()
        defer s.mutex.Unlock()
        s.RequestCount++
        status := strconv.Itoa(c.Response().Status)
        s.Statuses[status]++
        return nil
    }
}

// Handle is the endpoint to get stats.
func (s *Stats) Handle(c echo.Context) error {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    return c.JSON(http.StatusOK, s)
}

// ServerHeader middleware adds a `Server` header to the response.
func ServerHeader(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        c.Response().Header().Set(echo.HeaderServer, "Echo/3.0")
        return next(c)
    }
}

func main() {
    e := echo.New()

    // Debug mode
    e.Debug = true

    //-------------------
    // Custom middleware
    //-------------------
    // Stats
    s := NewStats()
    e.Use(s.Process)
    e.GET("/stats", s.Handle) // Endpoint to get stats

    // Server header
    e.Use(ServerHeader)
```

```
    // Handler
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Hello, World!")
    })

    // Start server
    e.Logger.Fatal(e.Start(":1323"))
}
```

响应

响应头

```
Content-Length:122
Content-Type:application/json; charset=utf-8
Date:Thu, 14 Apr 2016 20:31:46 GMT
Server:Echo/2.0
```

响应体

```
{
  "uptime": "2016-04-14T13:28:48.486548936-07:00",
  "requestCount": 5,
  "statuses": {
    "200": 4,
    "404": 1
  }
}
```

流式响应

流式响应

- 当数据产生的时候发送数据
- 使用分块传输编码（Chunked transfer encoding）的流式 JOSN 响应。

Server

server.go

```
package main

import (
    "net/http"
    "time"

    "encoding/json"

    "github.com/labstack/echo"
)

type (
    Geolocation struct {
        Altitude  float64
        Latitude  float64
        Longitude float64
    }
)

var (
    locations = []Geolocation{
        {-97, 37.819929, -122.478255},
        {1899, 39.096849, -120.032351},
        {2619, 37.865101, -119.538329},
        {42, 33.812092, -117.918974},
        {15, 37.77493, -122.419416},
    }
)

func main() {
    e := echo.New()
```

```
    e.GET("/", func(c echo.Context) error {
        c.Response().Header().Set(echo.HeaderContentType, echo.MIMEApplicationJSON)
        c.Response().WriteHeader(http.StatusOK)
        for _, l := range locations {
            if err := json.NewEncoder(c.Response()).Encode(l); err != nil {
                return err
            }
            c.Response().Flush()
            time.Sleep(1 * time.Second)
        }
        return nil
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

```
$ curl localhost:1323
```

输出

```
{"Altitude":-97,"Latitude":37.819929,"Longitude":-122.478255}
{"Altitude":1899,"Latitude":39.096849,"Longitude":-120.032351}
{"Altitude":2619,"Latitude":37.865101,"Longitude":-119.538329}
{"Altitude":42,"Latitude":33.812092,"Longitude":-117.918974}
{"Altitude":15,"Latitude":37.77493,"Longitude":-122.419416}
```

WebSocket

WebSocket

使用 `net` 库的 WebSocket

服务端

`server.go`

```
package main

import (
    "fmt"
    "log"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
    "golang.org/x/net/websocket"
)

func hello(c echo.Context) error {
    websocket.Handler(func(ws *websocket.Conn) {
        defer ws.Close()
        for {
            // Write
            err := websocket.Message.Send(ws, "Hello, Client!")
            if err != nil {
                log.Fatal(err)
            }

            // Read
            msg := ""
            err = websocket.Message.Receive(ws, &msg)
            if err != nil {
                log.Fatal(err)
            }
            fmt.Printf("%s\n", msg)
        }
    }).ServeHTTP(c.Response(), c.Request())
    return nil
}
```

```
func main() {
    e := echo.New()
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())
    e.Static("/", "../public")
    e.GET("/ws", hello)
    e.Logger.Fatal(e.Start(":1323"))
}
```

使用 `gorilla` 的 WebSocket

服务端

`server.go`

```
package main

import (
    "fmt"
    "log"

    "github.com/labstack/echo"

    "github.com/gorilla/websocket"
    "github.com/labstack/echo/middleware"
)

var (
    upgrader = websocket.Upgrader{}
)

func hello(c echo.Context) error {
    ws, err := upgrader.Upgrade(c.Response(), c.Request(), nil)
    if err != nil {
        return err
    }
    defer ws.Close()

    for {
        // Write
        err := ws.WriteMessage(websocket.TextMessage, []byte("Hello, Client!"))
        if err != nil {
            log.Fatal(err)
        }
```

```
        // Read
        _, msg, err := ws.ReadMessage()
        if err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%s\n", msg)
    }
}

func main() {
    e := echo.New()
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())
    e.Static("/", "../public")
    e.GET("/ws", hello)
    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

index.html

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>WebSocket</title>
</head>

<body>
  <p id="output"></p>

  <script>
    var loc = window.location;
    var uri = 'ws:';

    if (loc.protocol === 'https:') {
      uri = 'wss:';
    }
    uri += '//' + loc.host;
    uri += loc.pathname + 'ws';

    ws = new WebSocket(uri)
```

```
    ws.onopen = function() {
      console.log('Connected')
    }

    ws.onmessage = function(evt) {
      var out = document.getElementById('output');
      out.innerHTML += evt.data + '<br>';
    }

    setInterval(function() {
      ws.send('Hello, Server!');
    }, 1000);
  </script>
</body>

</html>
```

输出示例

Client

```
Hello, Client!
Hello, Client!
Hello, Client!
Hello, Client!
Hello, Client!
```

Server

```
Hello, Server!
Hello, Server!
Hello, Server!
Hello, Server!
Hello, Server!
```

JSONP

JSONP

JSONP 是一个能够被跨域访问资源的方法。

服务端

server.go

```
package main

import (
    "math/rand"
    "net/http"
    "time"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func main() {
    e := echo.New()
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    e.Static("/", "public")

    // JSONP
    e.GET("/jsonp", func(c echo.Context) error {
        callback := c.QueryParam("callback")
        var content struct {
            Response  string    `json:"response"`
            Timestamp time.Time `json:"timestamp"`
            Random    int       `json:"random"`
        }
        content.Response = "Sent via JSONP"
        content.Timestamp = time.Now().UTC()
        content.Random = rand.Intn(1000)
        return c.JSONP(http.StatusOK, callback, &content)
    })

    // Start server
```

```
    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

index.html

```
<!DOCTYPE html>
<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">
    <title>JSONP</title>
    <script type="text/javascript" src="//ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
    <script type="text/javascript">
        var host_prefix = 'http://localhost:1323';
        $(document).ready(function() {
            // JSONP version - add 'callback=?' to the URL - fetch the JSONP response to the request
            $("#jsonp-button").click(function(e) {
                e.preventDefault();
                // The only difference on the client end is the addition of 'callback=?' to the URL
                var url = host_prefix + '/jsonp?callback=?';
                $.getJSON(url, function(jsonp) {
                    console.log(jsonp);
                    $("#jsonp-response").html(JSON.stringify(jsonp, null, 2));
                });
            });
        });
    </script>

</head>

<body>
    <div class="container" style="margin-top: 50px;">
        <input type="button" class="btn btn-primary btn-lg" id="jsonp-button" value="Get JSONP response">
        <p>
            <pre id="jsonp-response"></pre>
        </p>
    </div>
```

```
</body>

</html>
```

文件上传

文件上传

上传单个文件

服务端

server.go

```
package main

import (
    "fmt"
    "io"
    "os"

    "net/http"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func upload(c echo.Context) error {
    // Read form fields
    name := c.FormValue("name")
    email := c.FormValue("email")

    //-----------
    // Read file
    //-----------

    // Source
    file, err := c.FormFile("file")
    if err != nil {
        return err
    }
    src, err := file.Open()
    if err != nil {
        return err
    }
    defer src.Close()
```

```
    // Destination
    dst, err := os.Create(file.Filename)
    if err != nil {
        return err
    }
    defer dst.Close()

    // Copy
    if _, err = io.Copy(dst, src); err != nil {
        return err
    }

    return c.HTML(http.StatusOK, fmt.Sprintf("<p>File %s uploaded successfully w
ith fields name=%s and email=%s.</p>", file.Filename, name, email))
}

func main() {
    e := echo.New()

    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    e.Static("/", "public")
    e.POST("/upload", upload)

    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

index.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Single file upload</title>
</head>
<body>
<h1>Upload single file 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="file"><br><br>
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

上传多个文件

服务端

server.go

```
package main

import (
    "fmt"
    "io"
    "os"

    "net/http"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func upload(c echo.Context) error {
    // Read form fields
    name := c.FormValue("name")
    email := c.FormValue("email")

    //------------
    // Read files
    //------------

    // Multipart form
    form, err := c.MultipartForm()
    if err != nil {
        return err
    }
    files := form.File["files"]

    for _, file := range files {
        // Source
        src, err := file.Open()
        if err != nil {
```

```
            return err
        }
        defer src.Close()

        // Destination
        dst, err := os.Create(file.Filename)
        if err != nil {
            return err
        }
        defer dst.Close()

        // Copy
        if _, err = io.Copy(dst, src); err != nil {
            return err
        }

    }

    return c.HTML(http.StatusOK, fmt.Sprintf("<p>Uploaded successfully %d files
 with fields name=%s and email=%s.</p>", len(files), name, email))
}

func main() {
    e := echo.New()

    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    e.Static("/", "public")
    e.POST("/upload", upload)

    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

index.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>
```

子域名

子域名

server.go

```
package main

import (
    "net/http"

    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

type (
    Host struct {
        Echo *echo.Echo
    }
)

func main() {
    // Hosts
    hosts := make(map[string]*Host)

    //-----
    // API
    //-----

    api := echo.New()
    api.Use(middleware.Logger())
    api.Use(middleware.Recover())

    hosts["api.localhost:1323"] = &Host{api}

    api.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "API")
    })

    //------
    // Blog
    //------
```

```
    blog := echo.New()
    blog.Use(middleware.Logger())
    blog.Use(middleware.Recover())

    hosts["blog.localhost:1323"] = &Host{blog}

    blog.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Blog")
    })

    //---------
    // Website
    //---------

    site := echo.New()
    site.Use(middleware.Logger())
    site.Use(middleware.Recover())

    hosts["localhost:1323"] = &Host{site}

    site.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Website")
    })

    // Server
    e := echo.New()
    e.Any("/*", func(c echo.Context) (err error) {
        req := c.Request()
        res := c.Response()
        host := hosts[req.Host]

        if host == nil {
            err = echo.ErrNotFound
        } else {
            host.Echo.ServeHTTP(res, req)
        }

        return
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

JWT

JWT

- JWT 使用 HS256 算法认证。
- JWT 从 `Authorization` 请求头取出数据。

服务端（使用 map）

`server.go`

```
package main

import (
    "net/http"
    "time"

    jwt "github.com/dgrijalva/jwt-go"
    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

func login(c echo.Context) error {
    username := c.FormValue("username")
    password := c.FormValue("password")

    if username == "jon" && password == "shhh!" {
        // Create token
        token := jwt.New(jwt.SigningMethodHS256)

        // Set claims
        claims := token.Claims.(jwt.MapClaims)
        claims["name"] = "Jon Snow"
        claims["admin"] = true
        claims["exp"] = time.Now().Add(time.Hour * 72).Unix()

        // Generate encoded token and send it as response.
        t, err := token.SignedString([]byte("secret"))
        if err != nil {
            return err
        }
        return c.JSON(http.StatusOK, map[string]string{
```

```
            "token": t,
        })
    }

    return echo.ErrUnauthorized
}

func accessible(c echo.Context) error {
    return c.String(http.StatusOK, "Accessible")
}

func restricted(c echo.Context) error {
    user := c.Get("user").(*jwt.Token)
    claims := user.Claims.(jwt.MapClaims)
    name := claims["name"].(string)
    return c.String(http.StatusOK, "Welcome "+name+"!")
}

func main() {
    e := echo.New()

    // Middleware
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    // Login route
    e.POST("/login", login)

    // Unauthenticated route
    e.GET("/", accessible)

    // Restricted group
    r := e.Group("/restricted")
    r.Use(middleware.JWT([]byte("secret")))
    r.GET("", restricted)

    e.Logger.Fatal(e.Start(":1323"))
}
```

服务端（使用结构体）

server.go

```
package main
```

```
import (
    "net/http"
    "time"

    jwt "github.com/dgrijalva/jwt-go"
    "github.com/labstack/echo"
    "github.com/labstack/echo/middleware"
)

// jwtCustomClaims are custom claims extending default ones.
type jwtCustomClaims struct {
    Name  string `json:"name"`
    Admin bool   `json:"admin"`
    jwt.StandardClaims
}

func login(c echo.Context) error {
    username := c.FormValue("username")
    password := c.FormValue("password")

    if username == "jon" && password == "shhh!" {

        // Set custom claims
        claims := &jwtCustomClaims{
            "Jon Snow",
            true,
            jwt.StandardClaims{
                ExpiresAt: time.Now().Add(time.Hour * 72).Unix(),
            },
        }

        // Create token with claims
        token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

        // Generate encoded token and send it as response.
        t, err := token.SignedString([]byte("secret"))
        if err != nil {
            return err
        }
        return c.JSON(http.StatusOK, echo.Map{
            "token": t,
        })
    }

    return echo.ErrUnauthorized
}
```

```
func accessible(c echo.Context) error {
    return c.String(http.StatusOK, "Accessible")
}

func restricted(c echo.Context) error {
    user := c.Get("user").(*jwt.Token)
    claims := user.Claims.(*jwtCustomClaims)
    name := claims.Name
    return c.String(http.StatusOK, "Welcome "+name+"!")
}

func main() {
    e := echo.New()

    // Middleware
    e.Use(middleware.Logger())
    e.Use(middleware.Recover())

    // Login route
    e.POST("/login", login)

    // Unauthenticated route
    e.GET("/", accessible)

    // Restricted group
    r := e.Group("/restricted")

    // Configure middleware with the custom claims type
    config := middleware.JWTConfig{
        Claims:     &jwtCustomClaims{},
        SigningKey: []byte("secret"),
    }
    r.Use(middleware.JWTWithConfig(config))
    r.GET("", restricted)

    e.Logger.Fatal(e.Start(":1323"))
}
```

客户端

curl

登录

使用账号和密码登录获取 token。

```
curl -X POST -d 'username=jon' -d 'password=shhh!' localhost:1323/login
```

返回

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0NjE5NTcxMzZ9.RB3arc4-OyzASAaUhC2W3ReWaXAt_z2Fd3BN4aWTgEY"
}
```

请求

在 `Authorization` 请求头设置 token，发送请求获取资源。

```
curl localhost:1323/restricted -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE0NjE5NTcxMzZ9.RB3arc4-OyzASAaUhC2W3ReWaXAt_z2Fd3BN4aWTgEY"
```

返回

```
Welcome Jon Snow!
```

Google App Engine

Google App Engine Recipe

Google App Engine (GAE) provides a range of hosting options from pure PaaS (App Engine Classic)
through Managed VMs to fully self-managed or container-driven Compute Engine instances. Echo
works great with all of these but requires a few changes to the usual examples to run on the
AppEngine Classic and Managed VM options. With a small amount of effort though it's possible
to produce a codebase that will run on these and also non-managed platforms automatically.

We'll walk through the changes needed to support each option.

Standalone

Wait? What? I thought this was about AppEngine! Bear with me - the easiest way to show the changes
required is to start with a setup for standalone and work from there plus there's no reason we
wouldn't want to retain the ability to run our app anywhere, right?

We take advantage of the go build constraints or tags to change
how we create and run the Echo server for each platform while keeping the rest of the application
(e.g. handler wireup) the same across all of them.

First, we have the normal setup based on the examples but we split it into two files - `app.go` will
be common to all variations and holds the Echo instance variable. We initialise it from a function
and because it is a `var` this will happen *before* any `init()` functions run - a feature that we'll
use to connect our handlers later.

`app.go`

{{< embed "google-app-engine/app.go" >}}

A separate source file contains the function to create the Echo instance and add the static
file handlers and middleware. Note the build tag on the first line which says to use this when *not*
bulding with appengine or appenginevm tags (which thoese platforms automatically add for us). We also
have the `main()` function to start serving our app as normal. This should all be very familiar.

`app-standalone.go`

{{< embed “google-app-engine/app-standalone.go” >}}

The handler-wireup that would normally also be a part of this Echo setup moves to separate files which
take advantage of the ability to have multiple `init()` functions which run *after* the `e` Echo var is
initialised but *before* the `main()` function is executed. These allow additional handlers to attach
themselves to the instance - I’ve found the `Group` feature naturally fits into this pattern with a file
per REST endpoint, often with a higher-level `api` group created that they attach to instead of the root
Echo instance directly (so things like CORS middleware can be added at this higher common-level).

`users.go`

{{< embed “google-app-engine/users.go” >}}

If we run our app it should execute as it did before when everything was in one file although we have
at least gained the ability to organize our handlers a little more cleanly.

AppEngine Classic and Managed VMs

So far we’ve seen how to split apart the Echo creation and setup but still have the same app that
still only runs standalone. Now we’ll see hwo those changes allow us to add support for AppEngine
hosting.

Refer to the AppEngine site for full configuration
and deployment information.

app.yaml configuration file

Both of these are Platform as as Service options running on either sandboxed micro-containers
or managed Compute Engine instances. Both require an `app.yaml` file to describe the app to
the service. While the app *could* still serve all it's static files itself, one of the benefits
of the platform is having Google's infrastructure handle that for us so it can be offloaded and
the app only has to deal with dynamic requests. The platform also handles logging and http gzip
compression so these can be removed from the codebase as well.

The yaml file also contains other options to control instance size and auto-scaling so for true
deployment freedom you would likely have separate `app-classic.yaml` and `app-vm.yaml` files and
this can help when making the transition from AppEngine Classic to Managed VMs.

`app-engine.yaml`

{{< embed "google-app-engine/app-engine.yaml" >}}

Router configuration

We'll now use the build constraints again like we did when creating
our `app-standalone.go` instance but this time with the opposite tags to use this file *if* the build has
the appengine or appenginevm tags (added automatically when deploying to these platforms).

This allows us to replace the `createMux()` function to create our Echo server *without* any of the
static file handling and logging + gzip middleware which is no longer required. Also worth nothing is
that GAE classic provides a wrapper to handle serving the app so instead of a `main()` function where
we run the server, we instead wire up the router to the default `http.Handler` instead.

`app-engine.go`

{{< embed "google-app-engine/app-engine.go" >}}

Managed VMs are slightly different. They are expected to respond to requests on port 8080 as well
as special health-check requests used by the service to detect if an instance is still running in
order to provide automated failover and instance replacement. The `google.golang.org/appengine` package provides this for us so we have a slightly different version for Managed VMs:

`app-managed.go`

{{< embed “google-app-engine/app-managed.go” >}}

So now we have three different configurations. We can build and run our app as normal so it can
be executed locally, on a full Compute Engine instance or any other traditional hosting provider
(including EC2, Docker etc…). This build will ignore the code in appengine and appenginevm tagged
files and the `app.yaml` file is meaningless to anything other than the AppEngine platform.

We can also run locally using the Google AppEngine SDK for Go
either emulating AppEngine Classic:

```
goapp serve
```

Or Managed VMs:

```
gcloud config set project [your project id]
gcloud preview app run .
```

And of course we can deploy our app to both of these platforms for easy and inexpensive auto-scaling joy.

Depending on what your app actually does it’s possible you may need to make other changes to allow
switching between AppEngine provided service such as Datastore and alternative storage implementations
such as MongoDB. A combination of go interfaces and build constraints can make this fairly straightforward
but is outside the scope of this recipe.

Maintainers

- CaptainCodeman

[Source Code]({{<source “google-app-engine”>}})

平滑关闭

平滑关闭

使用 grace

server.go

```
package main

import (
    "net/http"

    "github.com/facebookgo/grace/gracehttp"
    "github.com/labstack/echo"
)

func main() {
    // Setup
    e := echo.New()
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Six sick bricks tick")
    })
    e.Server.Addr = ":1323"

    // Serve it like a boss
    e.Logger.Fatal(gracehttp.Serve(e.Server))
}
```

使用 graceful

server.go

```
package main

import (
    "net/http"
    "time"

    "github.com/labstack/echo"
    "github.com/tylerb/graceful"
```

```
)

func main() {
    // Setup
    e := echo.New()
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, "Sue sews rose on slow joe crows nose")
    })
    e.Server.Addr = ":1323"

    // Serve it like a boss
    graceful.ListenAndServe(e.Server, 5*time.Second)
}
```

资源嵌入

资源嵌入

使用 go.rice

server.go

```
package main

import (
    "net/http"

    rice "github.com/GeertJohan/go.rice"
    "github.com/labstack/echo"
)

func main() {
    e := echo.New()
    // the file server for rice. "app" is the folder where the files come from.
    assetHandler := http.FileServer(rice.MustFindBox("app").HTTPBox())
    // serves the index.html from rice
    e.GET("/", echo.WrapHandler(assetHandler))

    // servers other static files
    e.GET("/static/*", echo.WrapHandler(http.StripPrefix("/static/", assetHandler)))

    e.Logger.Fatal(e.Start(":1323"))
}
```