用 Golang 快速构建 RESTful API

THE GO WAY

全面简单
Overall Simplicity

正交组合
Orthogonal Composition

偏好并发
Preference in Concurrency

语言决定思考方式

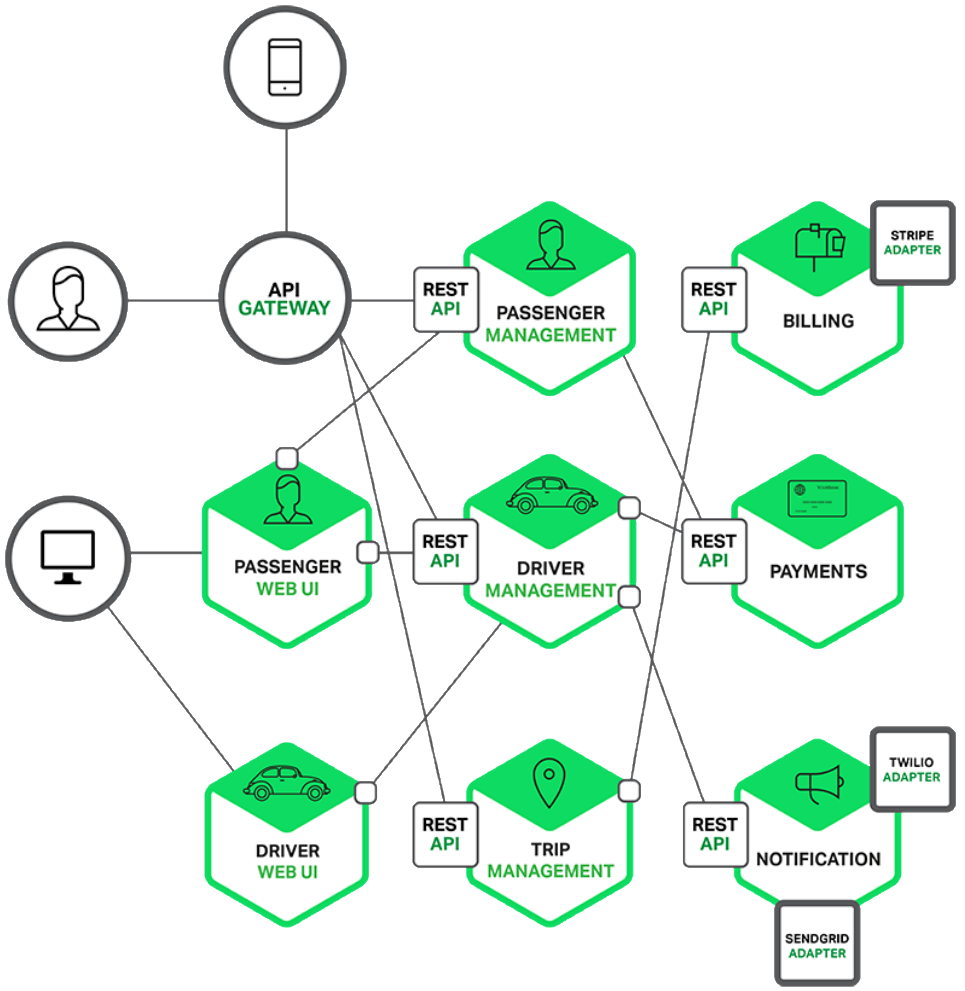
微服务架构

随着移动端的普及，迅速推动了前后端分离的趋势。

只需要一套 API，可以同时供Web前端/移动端/桌面客户端/小程序等所有Client使用。

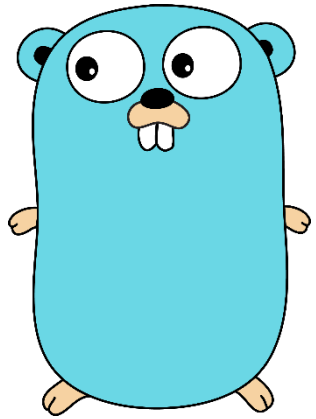
微服务架构又进一步提高了 RESTful API 的地位。

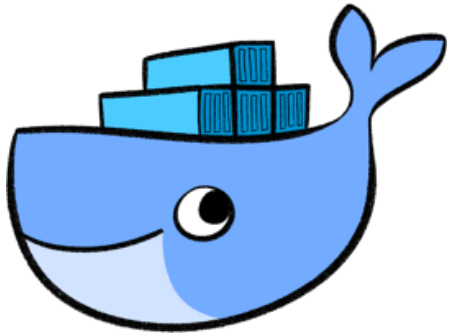
推荐阅读 12-Factor 原则。



RESTful API

- **Endpoint** 我的个人风格建议
  - 常规的增删改查用复数名词
    - 如管理员获取1号用户 GET /users/1
  - 单个资源操作可以接在后边
    - 如管理员封禁1号用户 PUT /users/1/ban
  - 特殊动作用单数名词
    - 如用户获取自身资料 GET /user
    - 如用户想要重置密码 PUT /user/reset
- **Response** 我的个人风格建议
  - 单数返回 json object
  - 列表返回 json array
  - 分页信息在 header 中返回
  - POST 和 PUT 返回新建或者更新后的对象
  - 按照规范返回 Status Code
  - 错误返回统一的错误结构





Golang 1.11

Golang 1.11 可以说是一个很重要的更新

它让 Golang 第一次真正的摆脱了 GOPATH 环境变量

让新手不至于建第一个项目就感到不适

Go Module 也是很有创意的一种依赖管理办法

相信稳定后将会是特别好用的版本管理方案

推荐使用 VS code 当做 IDE

框架选择 Echo

Optimized Router

Highly optimized HTTP router with zero dynamic memory allocation which smartly prioritize routes.

HTTP/2

HTTP/2 support improves speed and provides better user experience.

Data Rendering

API to send variety of HTTP response, including JSON, XML, HTML, File, Attachment, Inline, Stream or Blob.

Scalable

Build robust and scalable RESTful API, easily organized into groups.

Middleware

Many built-in middleware to use, or define your own. Middleware can be set at root, group or route level.

Templates

Template rendering using any template engine.

Automatic TLS

Automatically install TLS certificates from Let's Encrypt.

Data Binding

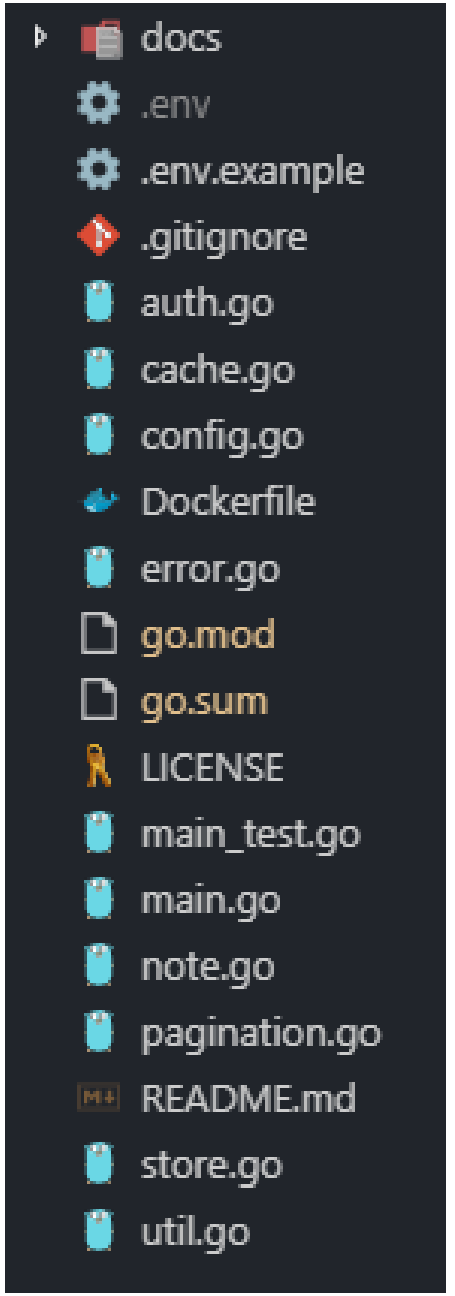
Data binding for HTTP request payload, including JSON, XML or form-data.

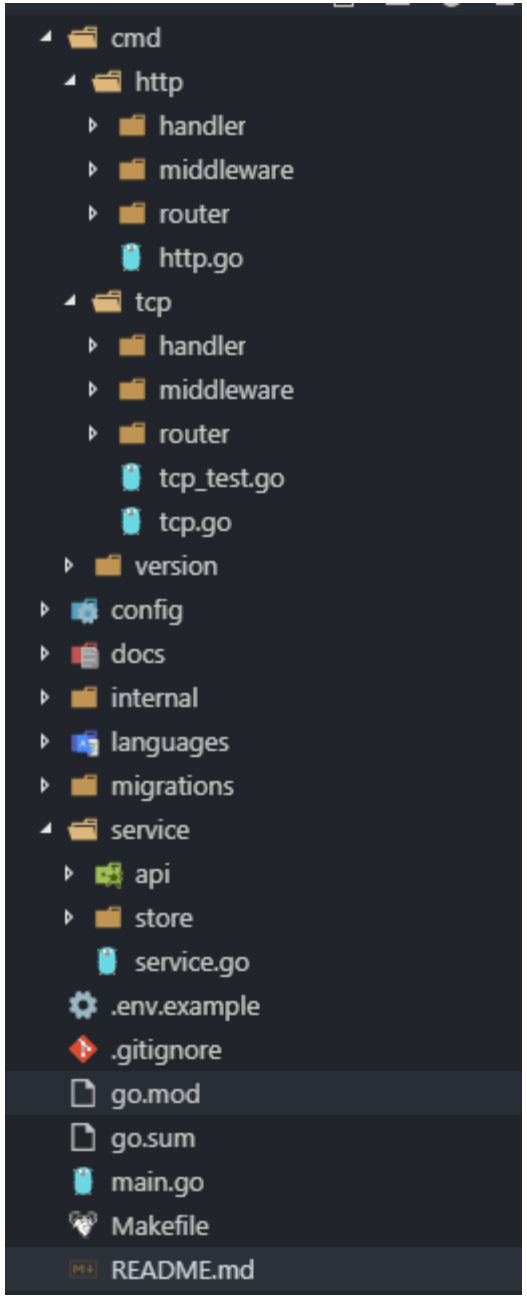
Extensible

Customized central HTTP error handling. Easily extendable API.

项目结构

- 在 Github 去看知名项目学习比看文章更有效
- 注意写 Library 和写 App 是不一样的
- 一个文件夹下，所有文件都是同一个 Package
- Golang 的项目结构有平铺和树状两种流派
- 在微服务架构中，建议使用平铺流
- 微服务做了纵向的分割，项目已经很小，没必要再做横向分层
- 保持一个业务对象在一个文件中，可以显著的提高效率。





新建项目

新接触 Golang 的同学可以参考这个顺序开始

1

在 Github 创建项目并初始化

推荐在任何公开或者私有的 Git 服务创建并初始化项目。
也可以建空项目后续在 PC 关联推送。

2

Clone 项目到本地

需要的是 git remote 的信息。
在本地 git init 并关联远程库也可以。

3

创建 Go module

在项目目录 go mod init
如果没有 git remote ，则需要手动写 go.mod 文件。

4

编写 main.go

先写一个最小化的应用调通。
于是又会写一种 Hello World 啦。

5

运行最小化 go 程序

执行 go run main.go 就可以啦，也可以用 VS code 的调试功能。

6

编写更多的逻辑

现在可以 import 更多的 package，新增别的 go 文件，添加更多的业务逻辑了。

7

记得注释和测试

Golang 对于注释有规范，可以自动生成文档，应该去研究下。养成写单元测试的好习惯。

8

持续集成和部署

Golang 是最适合用 docker 进行持续集成的项目啦。
后边会给大家大概介绍。

```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())

        // Routes
        e.GET("/", hello)

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

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

Echo Example

- Golang 很多库都喜欢用中间件来增加灵活性，与核心耦合较小的部分用中间件实现，用户可以随意选用和替换，也可以自定义中间件在处理流程中增加功能。
- 框架的核心就是把 HTTP Handler 注册到 Router
- Context 是 Golang 的另一个特色，在函数间单向传递的数据集合。
- 这个里的 Context 包含了请求，响应，连接，和 Echo 的很多辅助函数。

https://github.com/hyacinthus/restdemo

下面我会根据这个demo，讲解实际写一个项目要顾及的所有环节。

配置 Config

Golang 最著名的配置库是 viper
它几乎有所有的配置功能
但我们这个 demo 中选用的是另一个更简洁的

配置来源

PC上执行的程序一般用 flag + file 的组合。
分布式应用一般用 Env 或者 集中配置服务。

开发环境

使用 dotenv 来模拟加载环境变量。
注意在 gitignore 中忽略 .env 文件。

生产环境

利用 docker 部署，只用 Env 就足够。
传统部署模式可能需要集成配置服务。

配置 Config

```go
package main

import (
    "github.com/jinzhu/configor"
    "github.com/joho/godotenv"
    log "github.com/sirupsen/logrus"
)

var config = struct {
    APP struct {
        Debug    bool   `default:"false"`
        Host     string `default:"0.0.0.0"`
        Port     string `default:"1324"`
        PageSize int    `default:"10"`
        BaseURL  string `default:"https://api.example.com/"`
        FileURL  string `default:"https://static.example.com/"`
    }

    DB struct {
        Host     string `default:"mysql"`
        Port     string `default:"3306"`
        User     string `default:"root"`
        Password string `default:"root"`
        Name     string `default:"art"`
    }

    Redis struct {
        Host     string `default:"redis"`
        Port     string `default:"6379"`
        Password string
        DB       int `default:"0"`
    }
}{}
```

```go
func init() {
    godotenv.Load()
    configor.Load(&config)
    if config.APP.Debug {
        log.SetFormatter(&log.TextFormatter{
            FullTimestamp:   true,
            TimestampFormat: "06-01-02 15:04:05.00",
        })
        log.SetLevel(log.DebugLevel)
    } else {
        log.SetLevel(log.InfoLevel)
    }
}
```

```
CONFIGOR_ENV_PREFIX=-

APP_DEBUG=true
APP_BASEURL=http://localhost/
APP_FILEURL=http://localhost/

DB_HOST=mysql
DB_PORT=3306
DB_NAME=demo

REDIS_HOST=redis
REDIS_PORT=6379
```

日志 Logging

12-factor：把日志当作事件流

Logrus

输出到STDOUT

服务一律将日志输出到标准输出。

由专职的服务统一收集日志。

日志格式

使用 JSON 格式记录结构化的日志。

Debug模式可使用阅读友好的格式。

后续处理

规模较小时 docker 可以代为管理。

尽早接入 ELK，LogEntries，Loggly 等服务。

存储 ORM

```go
// Note 纸条
type Note struct {
    ID int `json:"id" gorm:"primary_key"`
    // 所属用户
    UserID int `json:"user_id" gorm:"index:idx_user_update"`
    // 标题
    Title string `json:"title"`
    // 内容
    Content string `json:"content" gorm:"size:2000"`
    // 是否公开
    IsPublic bool `json:"is_public"`
    // 创建时间
    CreatedAt time.Time `json:"created_at"`
    // 最后更新时间
    UpdatedAt time.Time `json:"updated_at" gorm:"index:idx_user_update"`
    // 软删除
    DeletedAt *time.Time `json:"-"`
}
```

GORM 是我唯一推荐的 ORM 框架

我也使用过 XORM 和 db.v3

和 GORM 都差得挺远

这还是我们中国人写的框架，提 issue 说不清着急了还可以说中文。

它可以自动管理ID，时间戳，软删除。自动创建和更新表。你可以到文档找你想象的功能，一般都能找到。

推荐研究一下它的 Preload 功能，及其方便。

```go

func initDB() {
    var err error
    // mysql conn
    for {
        db, err = gorm.Open("mysql", config.DB.User+":"+config.DB.Password+
            "@tcp("+config.DB.Host+":"+config.DB.Port+")/"+config.DB.Name+
            "?charset=utf8mb4&parseTime=True&loc=Local&timeout=90s")
        if err != nil {
            log.Warnf("waiting to connect to db: %s", err.Error())
            time.Sleep(time.Second * 2)
            continue
        }
        log.Info("Mysql connect successful.")
        break
    }

    // gorm debug log
    if config.APP.Debug {
        db.LogMode(true)
    }
}

// createTable gorm auto migrate tables
func createTables() {
    db.AutoMigrate(&Note{})
}

```

增删改查 READ

- Golang 建议变量使用短名称
- 函数返回结构体建议使用指针，这样出错时可以返回 nil
- 可以把 Handler 中重用的函数抽象出来，但依然推荐放在一个文件
- GORM 虽然会自动创建对象，但依然推荐提前 make 列表，用以在为空时返回 [ ] 而非 nil

```
39
40  func findNoteByID(id int) (*Note, error) {
41      var n = new(Note)
42      if err := db.First(n, id).Error; err != nil {
43          return nil, err
44      }
45      return n, nil
46  }
47
```

```
234 // @Router /public/notes/{id} [get]
235 func getPublicNote(c echo.Context) error {
236     id, err := strconv.Atoi(c.Param("id"))
237     if err != nil {
238         return newHTTPError(400, "InvalidID", "请在URL中提供合法的ID")
239     }
240     n, err := findNoteByID(id)
241     if err != nil {
242         return err
243     }
244     if !n.IsPublic {
245         return ErrNotFound
246     }
247     return c.JSON(http.StatusOK, n)
248 }
249
```

```
295 // @Router /public/notes [get]
296 func getPublicNotes(c echo.Context) error {
297     // 提前make可以让查询没有结果的时候返回空列表
298     var ns = make([]*Note, 0)
299     // 分页信息
300     limit := c.Get("limit").(int)
301     offset := c.Get("offset").(int)
302     err := db.Where("is_public = true").Order("updated_at desc").
303         Offset(offset).Limit(limit).Find(&ns).Error
304     if err != nil {
305         return err
306     }
307     setPaginationHeader(c, limit > len(ns))
308     return c.JSON(http.StatusOK, ns)
309 }
310
```

增删改查
CREAT & REMOVE

```go
// @Router /notes/{id} [delete]
func deleteNote(c echo.Context) error {
    id, err := strconv.Atoi(c.Param("id"))
    if err != nil {
        return newHTTPError(400, "InvalidID", "请在URL中提供合法的ID")
    }
    // 查询对象
    n, err := findNoteByID(id)
    if err != nil {
        return err
    }
    // 用户权限
    userID, err := parseUser(c)
    if err != nil {
        return err
    }
    if userID != n.UserID {
        return ErrForbidden
    }
    // 删除数据库对象
    if err := db.Delete(&Note{ID: id}).Error; err != nil {
        return err
    }
    return c.NoContent(http.StatusNoContent)
}
```

```go
// @Router /notes [post]
func createNote(c echo.Context) error {
    var a = new(Note)
    if err := c.Bind(a); err != nil {
        return err
    }
    // 校验
    if a.Title == "" {
        return newHTTPError(400, "BadRequest", "Empty title")
    }
    if a.Content == "" {
        return newHTTPError(400, "BadRequest", "Empty content")
    }
    // 用户信息
    userID, err := parseUser(c)
    if err != nil {
        return err
    }
    a.UserID = userID
    // 保存
    if err := db.Create(a).Error; err != nil {
        return err
    }

    return c.JSON(http.StatusCreated, a)
}
```

增删改查 UPDATE

- Golang 中写更新逻辑，无论在数据绑定环节还是数据库操作环节，都会遭受挑战。
- 由于 Golang 是静态语言，无法动态解析 json
- 目前更新有两种流派，一种是使用 map[string]interface{} 做数据绑定和更新，如果不做检查，这样代码极短。
- 这里推荐继续用结构体，不过使用指针类型。
- 后端需要检查前端来源字段，无论是使用第三方库还是自己手写，都是结构体更方便。还考虑了后续文档生成。

```go
// NoteUpdate 更新请求结构体，用指针可以判断是否有请求这个字段
type NoteUpdate struct {
    // 标题
    Title *string `json:"title"`
    // 内容
    Content *string `json:"content"`
    // 是否公开
    IsPublic *bool `json:"is_public"`
}
```

```go
func updateNote(c echo.Context) error {
    // 获取URL中的ID
    id, err := strconv.Atoi(c.Param("id"))
    if err != nil {
        return newHTTPError(400, "InvalidID", "请在URL中提供合法的ID")
    }
    var n = new(NoteUpdate)
    if err := c.Bind(n); err != nil {
        return err
    }
    old, err := findNoteByID(id)
    if err != nil {
        return err
    }
    // 用户权限
    userID, err := parseUser(c)
    if err != nil {
        return err
    }
    if userID != old.UserID {
        return ErrForbidden
    }
    // 利用指针检查是否有请求这个字段
    if n.Title != nil {
        if *n.Title == "" {
            return newHTTPError(400, "BadRequest", "Empty title")
        }
        old.Title = *n.Title
    }
    if n.Content != nil {
        if *n.Content == "" {
            return newHTTPError(400, "BadRequest", "Empty content")
        }
        old.Content = *n.Content
    }
    if n.IsPublic != nil {
        old.IsPublic = *n.IsPublic
    }

    if err := db.Save(old).Error; err != nil {
        return err
    }

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

自定义中间件 MIDDLEWARE

- Echo 的中间件可以选择在 router 前或者后执行，我们写的 Handler 是在中间件之后才执行的。
- 比如我们后面要讲的认证中间件就是在 router 后，解析 token 。
- 我们这个自写的中间件，会解析所有请求是否有分页参数。如果有的话，将它存储在 Context 供后续业务逻辑使用。
- 即使没有传参数，为了后续方便，我们设置默认参数。

```go
// ParsePagination 获得页码，每页条数，Echo中间件。
func ParsePagination(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
        var err error
        var page, pageSize int
        // 获得页码
        if c.QueryParam("page") == "" {
            page = 1
        } else {
            if page, err = strconv.Atoi(c.QueryParam("page")); err != nil {
                return newHTTPError(400, "InvalidPage", "请在URL中提供合法的页码")
            }
        }
        // 获得每页条数
        if c.QueryParam("per_page") == "" {
            pageSize = config.APP.PageSize
        } else {
            if pageSize, err = strconv.Atoi(c.QueryParam("per_page")); err != nil {
                return newHTTPError(400, "InvalidPage", "请在URL中提供合法的每页条数")
            }
        }
        // 设置查询数据时的 offset 和 limit
        c.Set("page", page)
        c.Set("offset", (page-1)*pageSize)
        c.Set("limit", pageSize)
        return next(c)
    }
}
```

分页 PAGINATION

分页有两种流派

核心区别为是否返回总记录条数。当 Web 分页器需要显示总页数时，就需要返回总条数。

而手机 APP 或者新型的 WEB 应用，则更多的使用下拉显示更多的翻页方式，不再需要总条数。我建议采用这种方式减少数据库查询。

分页记录的返回也有两种流派

我们建议不要在结果集中返回分页信息，而将分页信息放在 Header 中。

```go
// setPaginationHeader 设置分页相关 resp header
// 如果要显示页码，还需要返回 X-Total-Count 和 Link 的 last 信息，可以多传入一个记录总数参数进行处理。
// 移动应用一般不用知道总条数，传统的web分页器有时会需要。
func setPaginationHeader(c echo.Context, isLast bool) {
    page := c.Get("page").(int)
    pageSize := c.Get("limit").(int)
    c.Response().Header().Set("X-Page-Num", strconv.Itoa(page))
    c.Response().Header().Set("X-Page-Size", strconv.Itoa(pageSize))
    link := linkheader.Links{
        {URL: config.APP.BaseURL + "?page=" + strconv.Itoa(page) + "&per_page=" + strconv.Itoa(pageSize), Rel: "self"},
    }
    if !isLast {
        link = append(link, linkheader.Link{URL: config.APP.BaseURL + "?page=" + strconv.Itoa(page+1) + "&per_page=" + strconv.Itoa(pageSize), Rel: "next"})
    }
    c.Response().Header().Set("Link", link.String())
    return
}
```

认证 AUTH

```go
// Validator 校验token是否合法，顺便根据token在 context中赋值 user id
func validator(token string, c echo.Context) (bool, error) {
    // 调试后门
    logrus.Debug("token:", token)
    if config.APP.Debug && token == "debug" {
        c.Set("user_id", 1)
        return true, nil
    }
    // 寻找token
    var t *Token
    err := getcc("token:"+token, t)
    if err == cache.ErrCacheMiss {
        return false, nil
    } else if err != nil {
        return false, err
    }
    // 设置用户
    c.Set("user_id", t.UserID)

    return true, nil
}

// 这个函数还有一种设计风格，就是只是返回userid，
// 以支持可选登录，在业务中判断userid如果是0就没有登录
func parseUser(c echo.Context) (userID int, err error) {
    userID, ok := c.Get("user_id").(int)
    if !ok || userID == 0 {
        return 0, ErrUnauthorized
    }
    return userID, nil
}
```

```go
// skipper 这些不需要token
func skipper(c echo.Context) bool {
    method := c.Request().Method
    path := c.Path()
    // 先处理非GET方法，除了登录，现实中还可能有一些 webhooks
    switch path {
    case
        // 登录
        "/login":
        return true
    }
    // 从这里开始必须是GET方法
    if method != "GET" {
        return false
    }
    if path == "" {
        return true
    }
    resource := strings.Split(path, "/")[1]
    switch resource {
    case
        // 公开信息，把需要公开的资源每个一行写这里
        "swagger",
        "public":
        return true
    }
    return false
}
```

SESSION or JWT ?

```go
func login(c echo.Context) error {
    // 判断何种方式登录，小程序为提供code
    var req = new(LoginRequest) // 输入请求
    if err := c.Bind(req); err != nil {
        return err
    }
    var t *Token
    if req.Username == "username" && req.Password == "password" {
        // 发行token
        t = &Token{
            Token:     newUUID(),
            ExpiresAt: time.Now().Add(time.Hour * 96),
            // 这个userid应该是检索出来的，这里为demo写死。
            UserID: 1,
        }
        setcc("token:"+t.Token, t, time.Hour*96)
    } else {
        return ErrAuthFailed
    }
    return c.JSON(http.StatusOK, t)
}
```

```go
// LoginRequest 登录提供内容
type LoginRequest struct {
    // 用户名
    Username string `json:"username"`
    // 密码
    Password string `json:"password"`
}

// Token 以上第四步返回给客户端的token对象
type Token struct {
    Token     string    `json:"token"`
    ExpiresAt time.Time `json:"expires_at"`
    UserID    int       `json:"-"`
}
```

- 取决于用户的权限设计和业务需求
- 如果权限数据量很少，且是微服务架构，建议使用 JWT，在 TOKEN 中存储数据。
- 注意 JWT 的 TOKEN 无法撤销授权。
- 如果需要保存更多的用户状态，建议建立集中的 Session 服务。
- 服务数量少，可以服务间公用 Redis 共享 Session

缓存 CACHE

- 这个 msgpack 库，可以将 golang 的对象序列化和反序列化。
- 为了追求传输效率和节约存储容量时可以选用。
- 缺点是存储后或传输中没有了可读性。
- 所以大部分时候，我们会选用系统的 json 库进行正反序列化。

- 使用缓存时，可以在读取对象的方法里封装先读缓存，没有了再读数据库的逻辑。
- 一定要确定能把控全部的数据修改渠道。
- 在任何数据修改时，删除缓存。

- 这里公用了 Redis 连接，注意 key 的唯一性。

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

    "github.com/go-redis/cache"
    "github.com/labstack/echo"
    "github.com/sirupsen/logrus"
    "github.com/vmihailenco/msgpack"
)

// 初始化缓存
func initCache() {
    cc = &cache.Codec{
        Redis: rdb,
        Marshal: func(v interface{}) ([]byte, error) {
            return msgpack.Marshal(v)
        },
        Unmarshal: func(b []byte, v interface{}) error {
            return msgpack.Unmarshal(b, v)
        },
    }
}

// setcc 写缓存
func setcc(key string, object interface{}, exp time.Duration) {
    cc.Set(&cache.Item{
        Key:        key,
        Object:     object,
        Expiration: exp,
    })
}

// getcc 读缓存
func getcc(key string, pointer interface{}) error {
    return cc.Get(key, pointer)
}

// delcc 清缓存
func delcc(key string) {
    cc.Delete(key)
}
```

```go
// 定义错误
var (
    ErrNotFound     = newHTTPError(404, "NotFound", "没有找到相应记录")
    ErrAuthFailed   = newHTTPError(401, "AuthFailed", "登录失败")
    ErrUnauthorized = newHTTPError(401, "Unauthorized", "本接口只有登录用户才能调用")
    ErrForbidden    = newHTTPError(403, "Forbidden", "权限不足")
)

// httpError 对外输出的错误格式
type httpError struct {
    code int
    // 错误代码，为英文字符串，前端可用此判断大的错误类型。
    Key string `json:"error"`
    // 错误消息，为详细错误描述，前端可选择性的展示此字段。
    Message string `json:"message"`
}

func newHTTPError(code int, key string, msg string) *httpError {
    return &httpError{
        code:    code,
        Key:     key,
        Message: msg,
    }
}

// Error makes it compatible with `error` interface.
func (e *httpError) Error() string {
    return e.Key + ": " + e.Message
}
```

错误 ERROR

- 在 Golang 中 error 是个接口
- 错误和异常是不同的
- 错误要尽早处理
- 一般定义错误是为了后续比较
- 这里定义错误是为了不重复写

集中错误处理

- 集中错误处理是我最喜欢的 Echo 特性
- Gin 在这方面就做的不太顺手
- 它最重要的作用是，可以让你在非 Handler 函数中就返回准确的 HTTP 错误。
- 这样代码就更好把业务逻辑提取出来，Handler 只做接收和校验 Request，拼装 Response
- 它还省去了所有 500 和 404 错误的声明。
- 保证了输出的错误总是期望的格式。

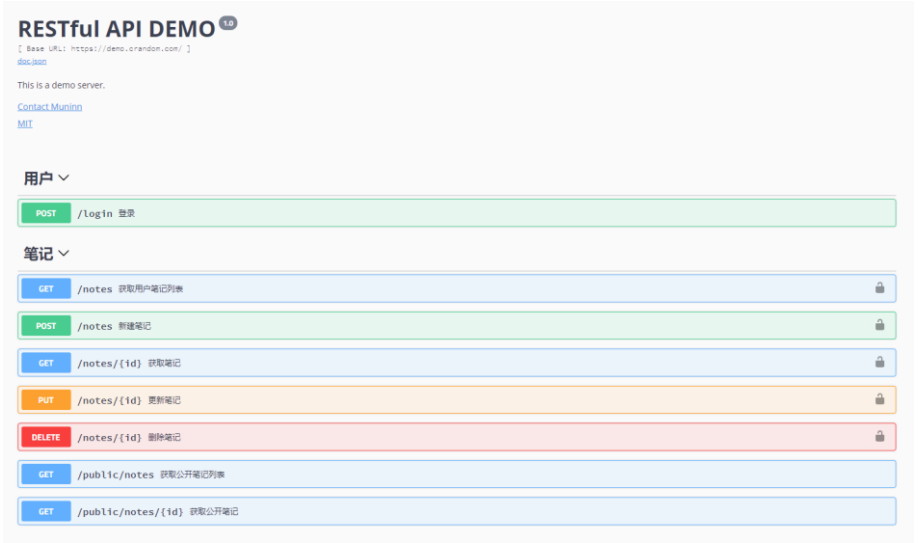
- 这个函数，在 Echo 返回错误 Response 前，根据收到的 error 内容，智能拼装。

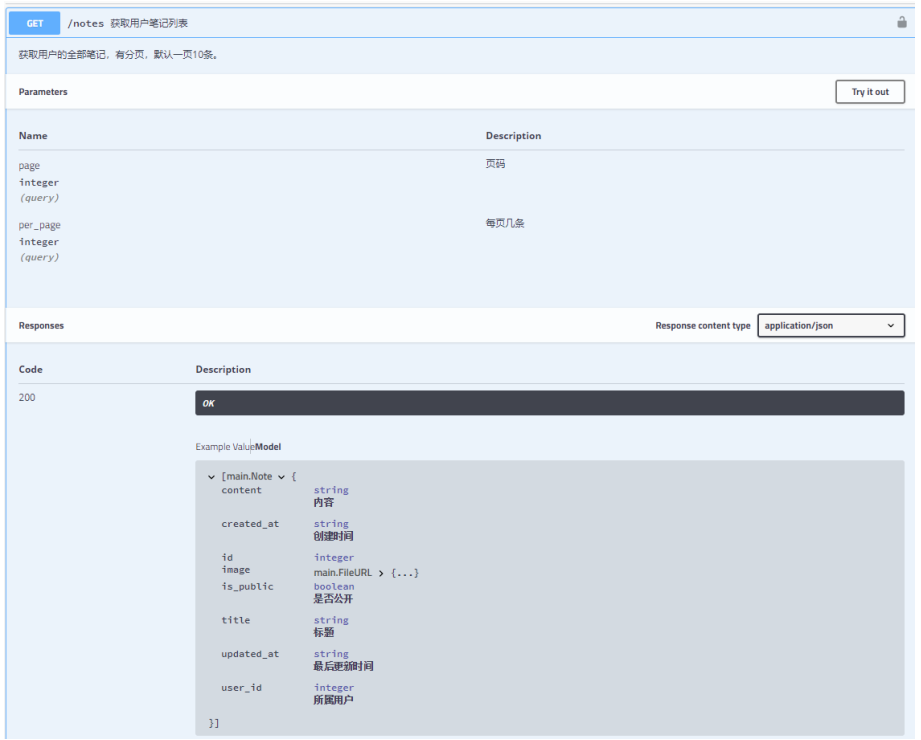
```
39
40  // httpErrorHandler customize echo's HTTP error handler.
41  func httpErrorHandler(err error, c echo.Context) {
42      var (
43          code = http.StatusInternalServerError
44          key  = "ServerError"
45          msg  string
46      )
47      // 二话不说先打日志
48      c.Logger().Error(err.Error())
49
50      if he, ok := err.(*httpError); ok {
51          // 我们自定的错误
52          code = he.code
53          key = he.Key
54          msg = he.Message
55      } else if ee, ok := err.(*echo.HTTPError); ok {
56          // echo 框架的错误
57          code = ee.Code
58          key = http.StatusText(code)
59          msg = key
60      } else if err == gorm.ErrRecordNotFound {
61          // 我们将 gorm 的没有找到直接返回 404
62          code = http.StatusNotFound
63          key = "NotFound"
64          msg = "没有找到相应记录"
65      } else if config.APP.Debug {
66          // 剩下的都是500 开了debug显示详细错误
67          msg = err.Error()
68      } else {
69          // 500 不开debug 用标准错误描述 以防泄漏信息
70          msg = http.StatusText(code)
71      }
72
```

文档 DOC

- 文档驱动开发 or 代码生成文档
- Swagger or API Blueprint
- 云服务推荐 apiary.io
- 但是现在我们自己生成文档更爽
- https://github.com/swaggo/swag

```
// updateNote 更新笔记
// @Tags 笔记
// @Summary 更新笔记
// @Description 更新指定id的笔记
// @Accept  json
// @Produce  json
// @Param data body main.NoteUpdate true "更新内容"
// @Success 200 {object} main.Note
// @Failure 400 {object} main.httpError
// @Failure 401 {object} main.httpError
// @Failure 403 {object} main.httpError
// @Failure 404 {object} main.httpError
// @Failure 500 {object} main.httpError
// @Security ApiKeyAuth
// @Router /notes/{id} [put]
func updateNote(c echo.Context) error {
```





```go

// FileURL 图片链接
type FileURL string

// ToString 转换为string类型
func (f FileURL) ToString() string {
    var s = string(f)
    var url = s
    if !strings.HasPrefix(s, "http") {
        url = config.APP.FileURL + s
    }
    return url
}

// MarshalJSON 转换为json类型 加域名
func (f FileURL) MarshalJSON() ([]byte, error) {
    return json.Marshal(f.ToString())
}

// UnmarshalJSON 不做处理
func (f *FileURL) UnmarshalJSON(data []byte) error {
    var tmp string
    if err := json.Unmarshal(data, &tmp); err != nil {
        return err
    }
    tmp = strings.TrimPrefix(tmp, config.APP.FileURL)
    *f = FileURL(tmp)
    return nil
}

```

自定义类型

Golang 又一个令人惊喜的风格

自己控制 json 序列化和数据库读取

以及更多

```go
// Scan implements the Scanner interface.
func (f *FileURL) Scan(src interface{}) error {
    if src == nil {
        *f = ""
        return nil
    }
    tmp, ok := src.([]byte)
    if !ok {
        return errors.New("Read file url data from DB failed")
    }
    *f = FileURL(tmp)
    return nil
}

// Value implements the driver Valuer interface.
func (f FileURL) Value() (driver.Value, error) {
    return string(f), nil
}

```

THANK YOU

From hyacinthus@gmail.com