# Why We Use Go Web Frameworks

Web Service 框架解决的核心问题 —— 严清

**teambition**

团队协作工具创导者

# 关于我

- **五年 JS，一年 Go，也玩 Rust**

  Github 满满绿格子见证我开发生涯的一面

- **16 年底组建 Go 团队，重构后端服务体系，为此造了一些轮子，如 Gear 框架**

  基于 kubernetes 和 SOA，部分已上线，如 TCM 消息推送服务直接提供 HTTP/2 和 gRPC 接口

- **前端架构师 → 后端架构师 ⇢ 技术负责人**

  技术学习无止境，逼迫个人成长，推进团队成长

本想网上搜个 PPT
交作业给谢大~ 结果：

Why I Don't Use Go Web Frameworks
——Joe Chasinga

- http package 能力强大，已是一个 web 框架 HTTP能力确实相对完整
- 即使有更复杂的需求，即插即用的包管理机制也能轻易实现 Go 的大糟点啊，学学 Rust
- 第三方框架都有学习成本，踩上坑就得潜入源码求解决 其实就两三千行代码，都是精华，值得看

如果你只写 Hello World 或 Todolist，或者是个人开发者、爱折腾，没问题！
否则，还是使用一款框架吧！

# Web 框架要解决三个核心问题

- 定义灵活、一致的开发模式

  简单易上手，支撑大规模复杂应用，支撑团队开发

- 集成简洁、完善的异常处理能力

  不被 `if err != nil { }` 羁绊，不放过任何异常，优雅漂亮地处理错误和异常

- 提供强大、实用的 HTTP 操作方法语法糖

  写 web 服务就是操作 HTTP，实用语法糖极大提升开发人员的幸福指数

# Middleware 模式及其控制

```
app := gear.New()

app.Set(gear.SetLogger, log.New(gear.DefaultFilterWriter(), "", log.LstdFlags))
app.Use(func(ctx *gear.Context) error {
  if ctx.Path != "/" {
    return nil
  }
  return ctx.JSON(http.StatusOK, map[string]string{
    "name":    "Teambition KBS Service",
    "version": Version,
  })
})
app.UseHandler(util.Logger)
app.UseHandler(authService.Client)
app.UseHandler(middleware.Ratelimit(&cfg, redisService.Client, authService))
app.Use(zipkinService.New())
app.Use(middleware.Pagination)
app.UseHandler(initRouterV1(mongoService, zipkinService, authService))
```

**Express、koa、toa、Gear、Echo、Gin、Iris... 大家都选择了中间件模式**

# Middleware 模式及其控制

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

myServer := grpc.NewServer(
    grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
        grpc_ctxtags.StreamServerInterceptor(),
        grpc_opentracing.StreamServerInterceptor(),
        grpc_prometheus.StreamServerInterceptor,
        grpc_zap.StreamServerInterceptor(zapLogger),
        grpc_auth.StreamServerInterceptor(myAuthFunction),
        grpc_recovery.StreamServerInterceptor(),
    )),
    grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
        grpc_ctxtags.UnaryServerInterceptor(),
        grpc_opentracing.UnaryServerInterceptor(),
        grpc_prometheus.UnaryServerInterceptor,
        grpc_zap.UnaryServerInterceptor(zapLogger),
        grpc_auth.UnaryServerInterceptor(myAuthFunction),
        grpc_recovery.UnaryServerInterceptor(),
    )),
)
```

**gRPC 生态也玩起了中间件模式。。。**

## Middleware 模式及其控制

- 简单标准的接口，通过插拔式组合能力构建复杂应用
- 专注于单一功能的实现，逻辑解耦，精益求精

**Gear 定义了两种形式中间件：**

```go
// Middleware defines a function to process as middleware.
type Middleware func(ctx *Context) error

// Handler interface is used by app.UseHandler as a middleware.
type Handler interface {
  Serve(ctx *Context) error
}
```

**但本质是：func(ctx *gear.Context) *error***

**非常简洁，却集成了 Web 框架核心能力三要素**

**Router, Logging, CORS, Favicon, Secure, Static, JWT-Auth, Ratelimiter, Tracing…**

# 几种 Middleware 形态

## Gin中间件，error 处理?

```
type HandlerFunc func(*Context)
```

## Echo 中间件，MiddlewareFunc?

```
type HandlerFunc func(Context) error
type MiddlewareFunc func(HandlerFunc) HandlerFunc
```

## Go-kit中间件，http 对象?

```
type Endpoint func(ctx context.Context, request interface{}) (response interface{}, err error)
type Middleware func(Endpoint) Endpoint
```

## Go 原生接口，流程控制?

```
type HandlerFunc func(ResponseWriter, *Request)
type Handler interface {
  ServeHTTP(ResponseWriter, *Request)
}
```

# Middlewares 的组合：流程之级联控制

```
app.Use(func(ctx *gin.Context) {
  fmt.Println("A")
  ctx.Next()
  fmt.Println("B")
})

app.Use(func(ctx *gin.Context) {
  fmt.Println("C")
  ctx.JSON(200, someBody)
})
// ACB
```

**Gin 的中间件流程控制，koa 的洋葱头级联模型**

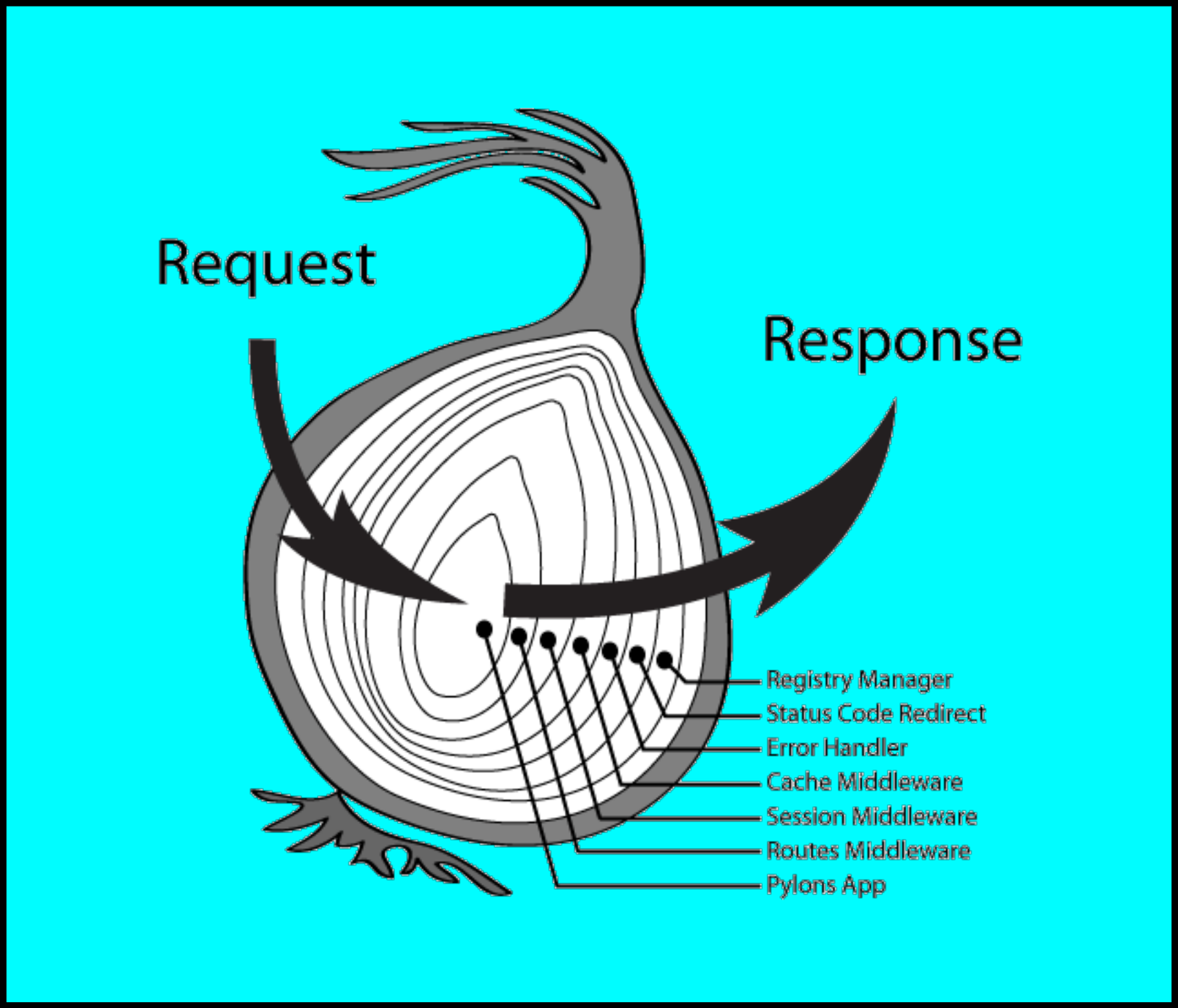
**思考：如何进行异常和错误控制？比如 Auth 中间件用户身份验证失败**

# Middlewares 的组合：流程之级联控制

```
app.Use(func(next echo.HandlerFunc) echo.HandlerFunc {
  return func(ctx echo.Context) error {
    fmt.Println("A")
    e := next(ctx)
    fmt.Println("B")
    return e
  }
})

app.Use(func(next echo.HandlerFunc) echo.HandlerFunc {
  return func(ctx echo.Context) error {
    fmt.Println("C")
    return ctx.JSON(200, someBody)
  }
})
// ACB
```

**Echo 的中间件流程控制，通过 next 控制，依然是葱头级联模型**

# Middlewares 的组合：流程之级联控制



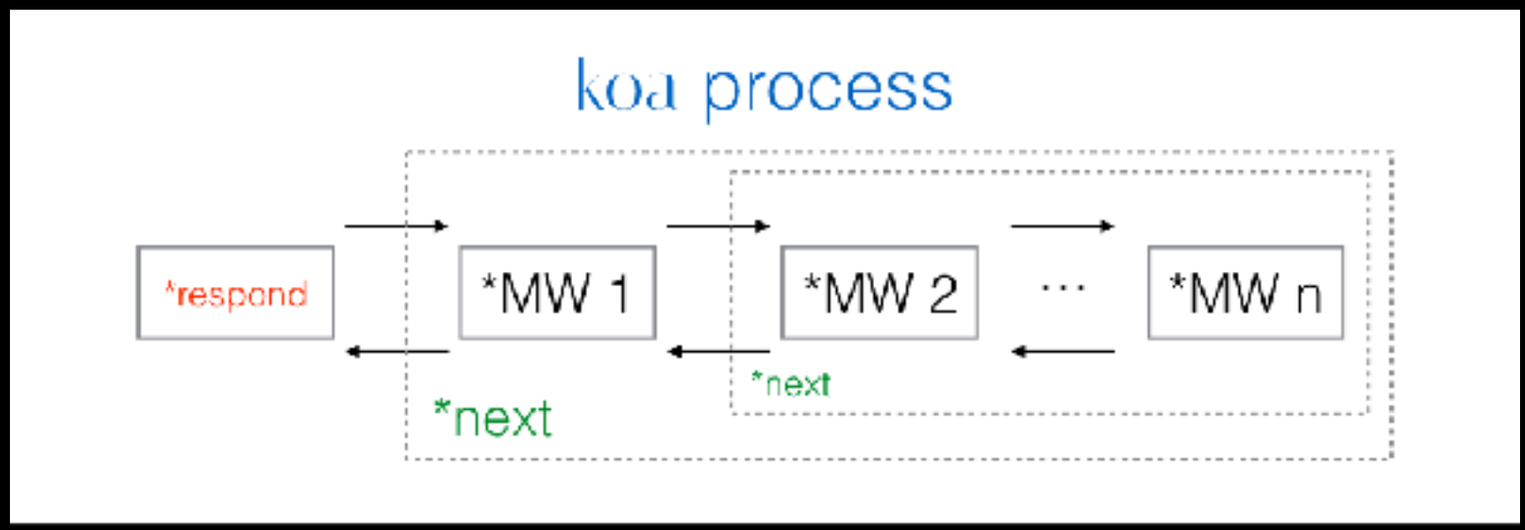
# Middlewares 的组合：流程之顺序控制

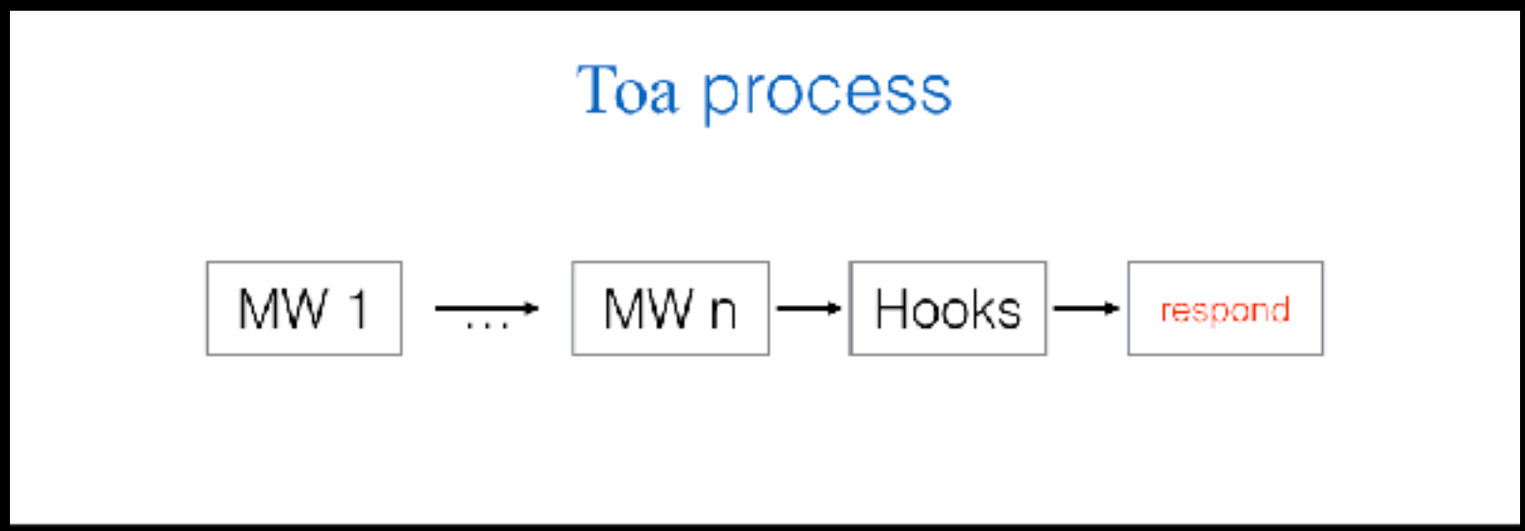
```go
app.Use(func(ctx *gear.Context) error {
  fmt.Println("A")
  fmt.Println("B")
  return nil
})

app.Use(func(ctx *gear.Context) error {
  fmt.Println("C")
  return ctx.JSON(200, someBody)
})
// ABC
```

**Gear 的中间件流程控制，是符合直觉的顺序流程**
**错误或者写入数据会自动（原子锁）终止后续中间件运行**

Middleware 模式及其控制

# 级联流程控制 VS 顺序流程控制




级联模式下的思考：

1. next 函数有没有写入数据？会不会出错？出错怎么处理？再次写入？
2. 通过 context 设置请求超时控制，超时或 cancel 后在运行的中间件处理流会终止吗？

集中、智能、灵活的异常处理

**func(ctx *gear.Context) *error***

**框架捕获了某个中间件的 *error* 怎么处理?**

**根据 HTTP 定义，只要有错误码和错误消息就能处理了**

**只需要在 error interface 之上增加一个 status interface**

```go
type HTTPError interface {
  // Error returns error's message.
  Error() string
  // Status returns error's http status code.
  Status() int
}
```

# 不过，*Gear* 的 *Error* 还可以更强大

```go
type Error struct {
  Code  int         `json:"-"`
  Err   string      `json:"error"`
  Msg   string      `json:"message"`
  Data  interface{} `json:"data,omitempty"`
  Stack string      `json:"-"`
}
```

```go
func (err *Error) Status() int
func (err *Error) Error() string
func (err Error) String() string
func (err Error) GoString() string
func (err Error) WithMsg(msgs ...string) *Error
func (err Error) WithMsgf(format string, args ...interface{}) *Error
func (err Error) WithCode(code int) *Error
func (err Error) WithStack(skip ...int) *Error
func (err Error) From(e error) *Error
```

**充分利用非引用定义方法的复制特性，实现了 errors 模板机制**

集中、智能、灵活的异常处理

# *Gear* 预定义的 *errors* 模板常量及使用观感

```go
// Predefined errors
var (
  Err = &Error{Code: http.StatusInternalServerError, Err: "Error"}

  ErrBadRequest          = Err.WithCode(http.StatusBadRequest)
  ErrUnauthorized        = Err.WithCode(http.StatusUnauthorized)
  ErrPaymentRequired     = Err.WithCode(http.StatusPaymentRequired)
  ErrForbidden           = Err.WithCode(http.StatusForbidden)
  ErrNotFound            = Err.WithCode(http.StatusNotFound)
  ErrMethodNotAllowed    = Err.WithCode(http.StatusMethodNotAllowed)
  // and more errors...
)

// 使用效果
err := gear.ErrBadRequest.WithMsg("invalid email”)
err := gear.ErrBadRequest.WithMsgf(`invalid email: "%s"`, email)
err := gear.Err.WithMsg("some error").WithStack()
```

**还要支持 i18n 的 errors 机制？没问题，参考 gear.Error 实现一个，你行的！**

集中、智能、灵活的异常处理

# *Gear* 框架自定义 *error* 的响应

```
// 默认响应为 “application/json”，支持自定义响应类型
app.Set(gear.SetOnError, func(ctx *gear.Context, err gear.HTTPError) {
  // 例如响应为纯文本
  // 还可以基于 err 类型做不同响应
  ctx.Type(MIMETextPlainCharsetUTF8)
  ctx.End(err.Status(), []byte(err.Error()))
})
```

# Web 服务必备的操作 HTTP 的方法及扩展能力

```
  // implement context.Context interface
func (ctx *Context) Deadline() (time.Time, bool)
func (ctx *Context) Done() <-chan struct{}
  // and more…

func (ctx *Context) IP() net.IP
func (ctx *Context) AcceptType(preferred ...string) string
func (ctx *Context) Param(key string) (val string)
func (ctx *Context) Query(name string) string
  // and more…
```

```
func (ctx *Context) HTML(code int, str string) error
func (ctx *Context) JSON(code int, val interface{}) error
func (ctx *Context) XML(code int, val interface{}) error
  // and more…

func (ctx *Context) Render(code int, name string, data interface{}) (err error)
func (ctx *Context) Stream(code int, contentType string, r io.Reader) (err error)
func (ctx *Context) Attachment(string, time.Time, io.ReadSeeker, ...bool) (err error)
  // and more…

func (ctx *Context) Redirect(url string) (err error)
func (ctx *Context) Error(e error) error
  // and more…
```

# 不一样的 ctx.ParseBody

```go
// 定义请求数据模板和验证逻辑:
type loginTemplate struct {
  Name string `json:"name" form:"name"`
  Pass string `json:"pass" form:"pass"`
}
func (t *loginTemplate) Validate() error {
  if len(t.Name) < 3 || len(t.Pass) < 6 {
    return gear.ErrBadRequest.WithMsg("invalid name or pass")
  }
  return nil
}

// 在 API 中间件中使用它:
func (api *User) Login(ctx *gear.Context) error {
  body := loginTemplate{}
  if err := ctx.ParseBody(&body) {
    return err
  }
  // more…
}
```

# 和不一样的 ctx.ParseURL

```go
type taskTemplate struct {
  ID       bson.ObjectId `json:"_taskID" param:"_taskID"`
  StartAt time.Time     `json:"startAt" query:"startAt"`
}

func (b *taskTemplate) Validate() error {
  if !b.ID.Valid() {
    return gear.ErrBadRequest.WithMsg("invalid task id")
  }
  if b.StartAt.IsZero() {
    return gear.ErrBadRequest.WithMsg("invalid task start time")
  }
  return nil
}
```

**ctx.ParseURL 与 ctx.ParseBody 一致的使用方式，强大和灵活，可自定义**

强类型的疼，再也不能像 JS 一样任性的扩展 context 了

类似解析请求数据这种需求，可做成类似 util 工具包

但像 session 这种需要在中间件间传递状态的怎么解?

```go
type Any interface {
  New(ctx *Context) (interface{}, error)
}
func (ctx *Context) Any(any interface{}) (val interface{}, err error)
func (ctx *Context) SetAny(key, val interface{})
```

超约原生 context 的 *gear.Context

强大、实用的 HTTP 操作语法糖

# 基于 gear.Any interface 扩展能力实现的 Session

```go
// 定义符合自己业务需求的 Session 结构
type Session struct {
  *sessions.Meta `json:"-"`
  UserID         string `json:"_userId"`
  Name           string `json:"name"`
  Avatar         string `json:"avatar"`
}
func FromCtx(ctx *gear.Context) (*Session, error) {
  val, err := ctx.Any(gearSession)
  return val.(*Session), err
}
// 在中间件中使用自定义的 Session
app.Use(func(ctx *gear.Context) error {
  sess, err := FromCtx(ctx)
  sess.Avatar = "avatar.png"
  sess.Save() // update session
  return ctx.JSON(200, sess)
})
// https://github.com/teambition/gear-session
```

还有很多中间件利用了 any interface，如 logging，auth，router 等

https://github.com/teambition/gear

积攒 Star…

谢谢！