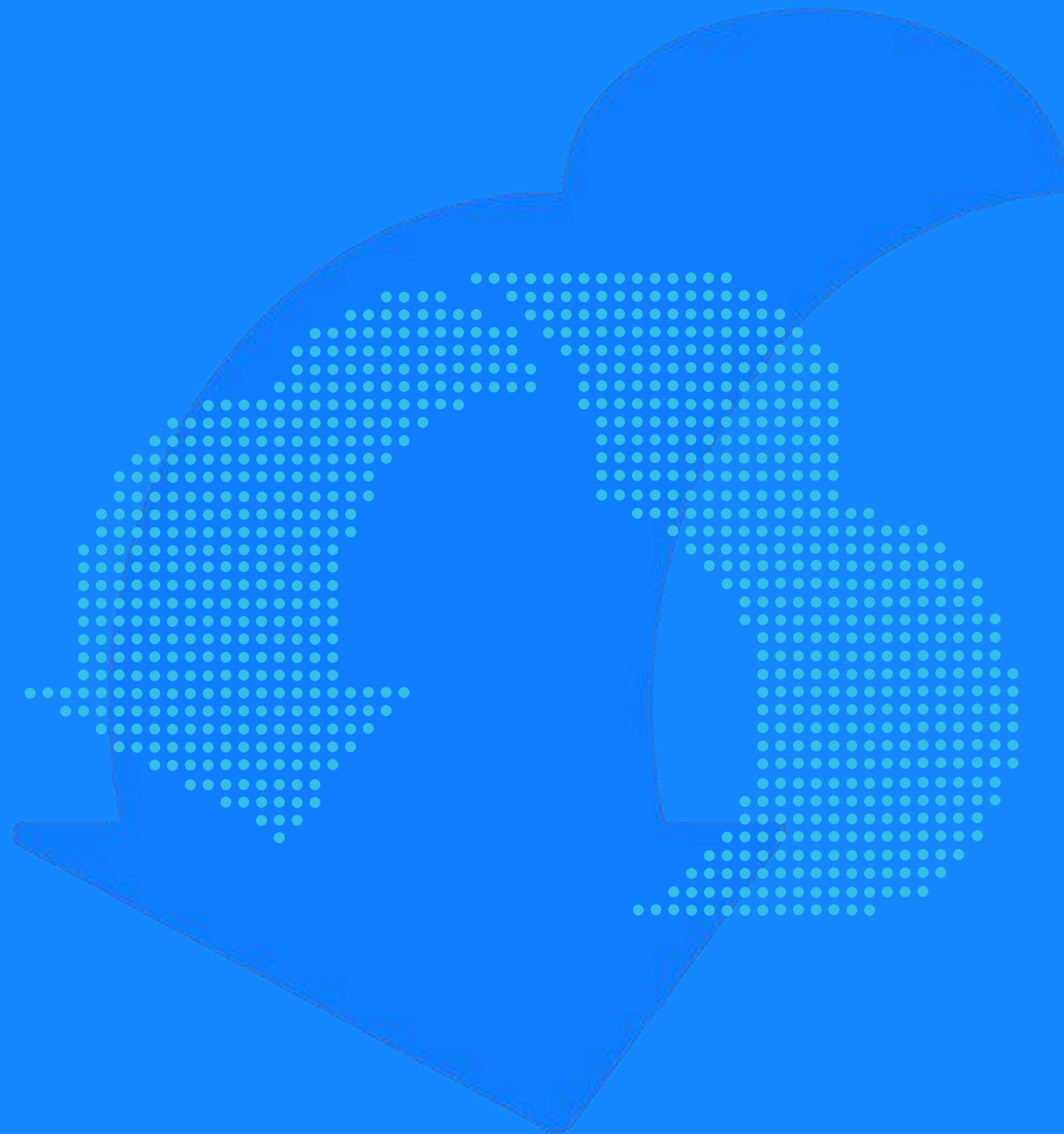


Hertz-Session

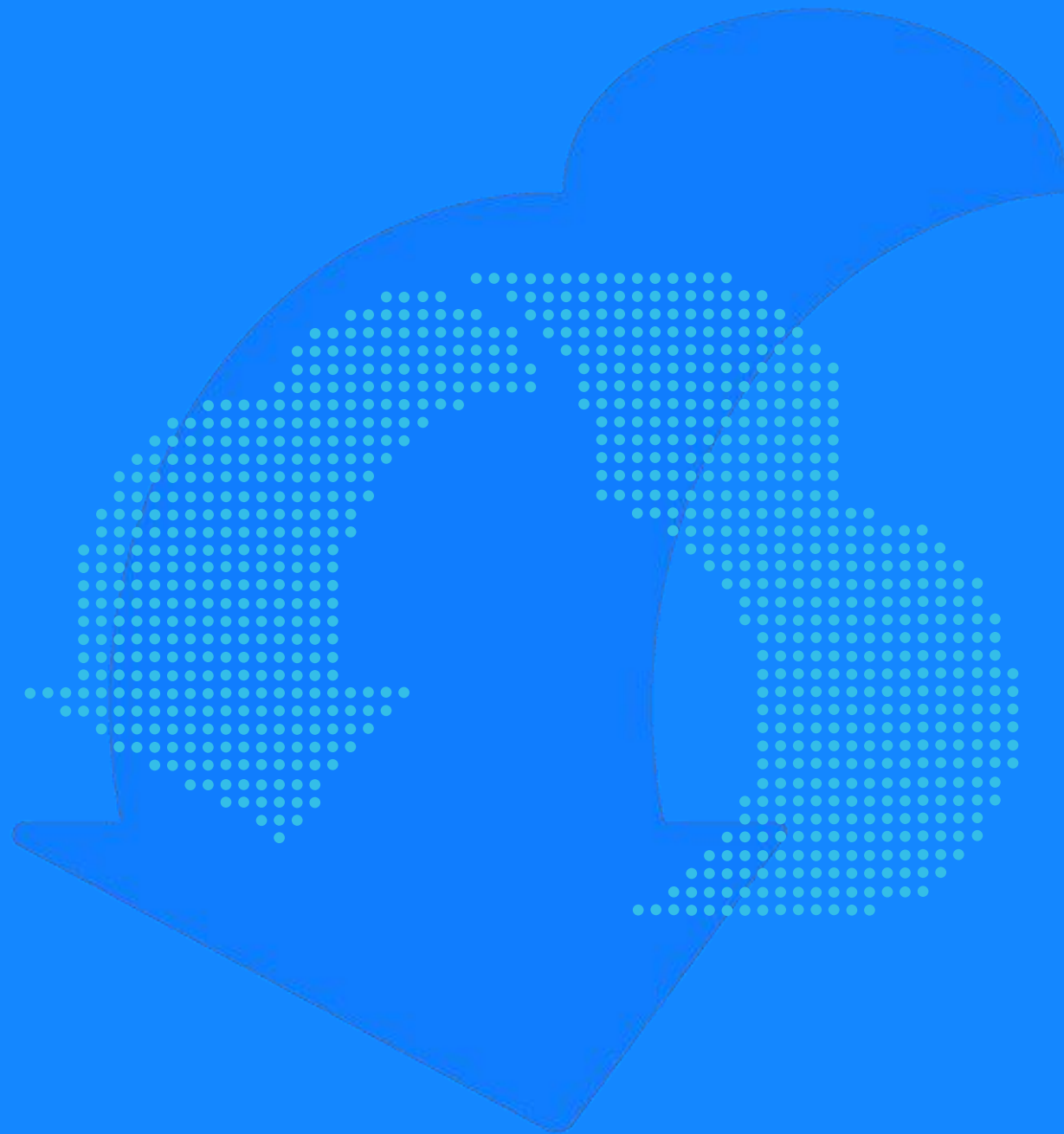
Hertz Middleware

李宜茗 - Hertz Reviewer

2023/06/05



- **李宜茗**
(<https://github.com/justlorain>)
- **在校大学生**
- **CloudWeGo - Hertz Reviewer**
- **喜欢看动漫和睡觉**



CONTENT

目录

01.

什么是 Hertz Middleware

Hello, middleware!

02.

如何编写 Hertz Middleware

How?

03.

Hertz-Session 解读

biz-demo



01

什么是 Hertz Middleware

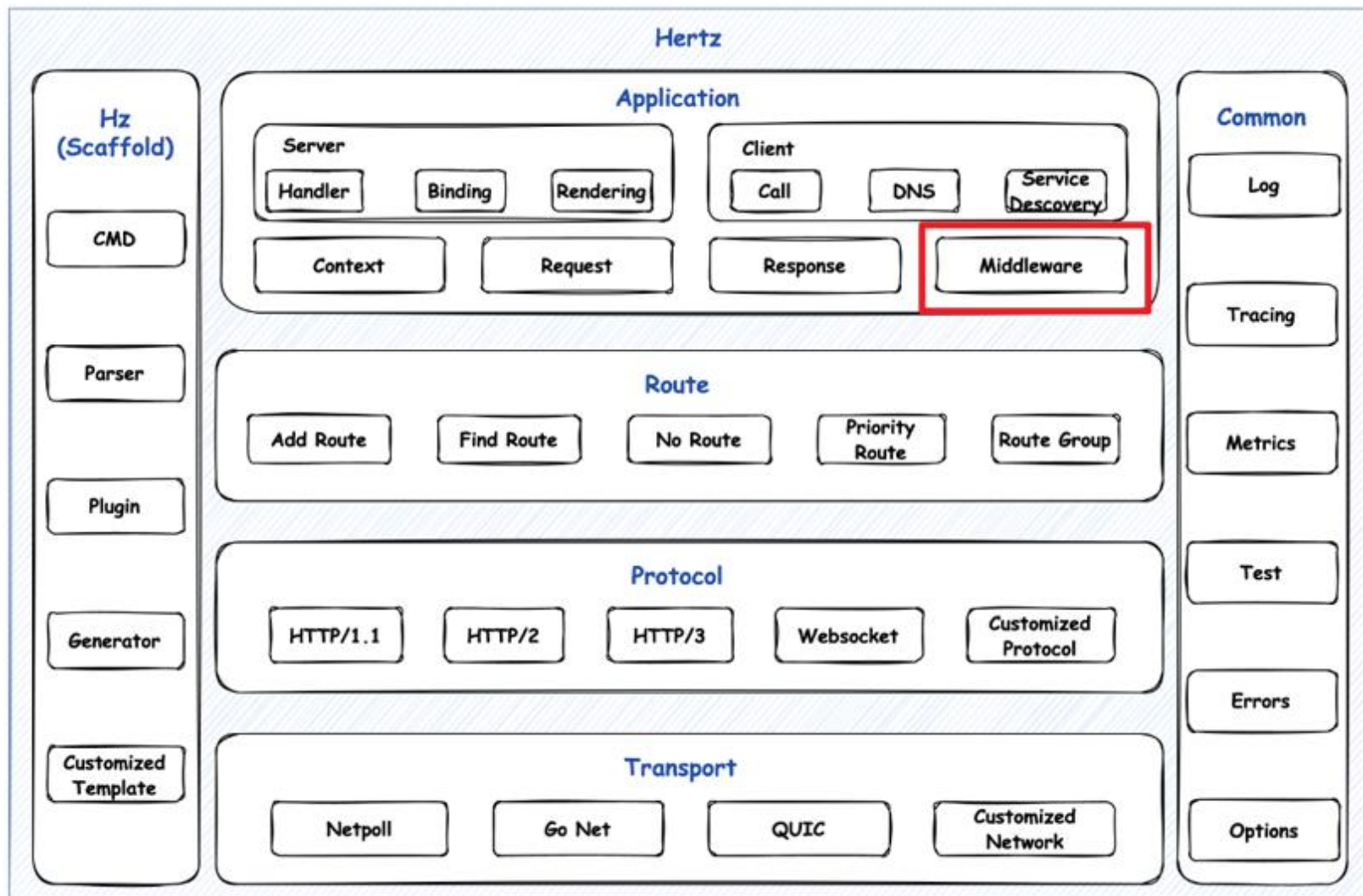
Hello, middleware!

中间件概览

<https://www.cloudwego.io/zh/docs/hertz/tutorials/basic-feature/middleware/>

Hertz中间件的种类是多种多样的，简单分为两大类：

- 服务端中间件
- 客户端中间件



Hertz Server 中间件的拓展

目前，在社区同学的贡献下，Hertz 已经拥有了丰富的中间件生态，其实现放到了 [hertz-contrib](#) 下

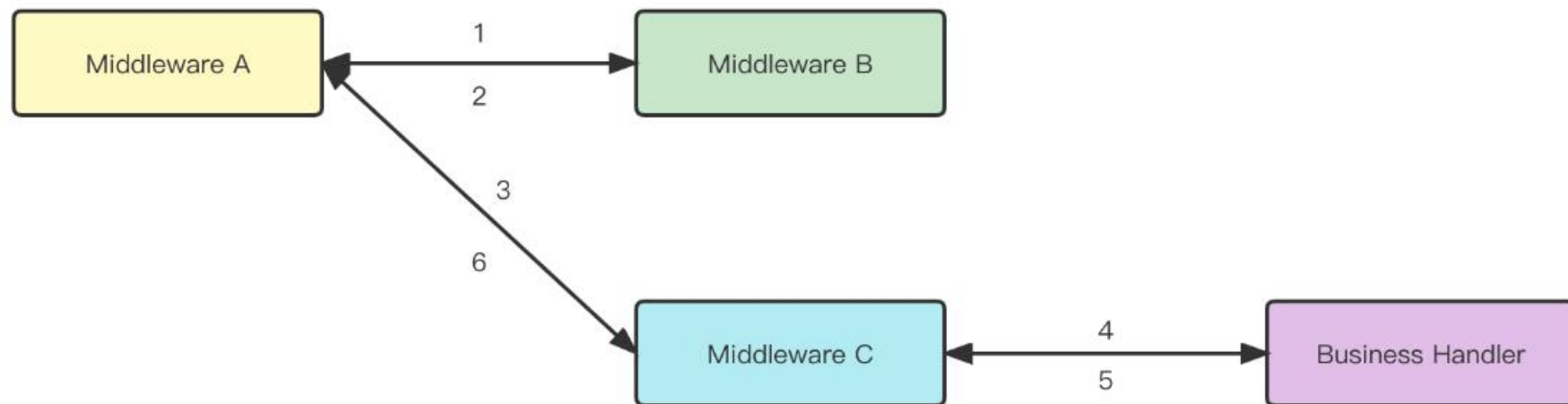
类型	中间件	简述
认证/授权	Casbin 中间件	基于Casbin的授权中间件，用于实现RBAC、ABAC、ACL等多种访问控制模型，可以通过配置文件进行动态的权限管理
	JWT 中间件	用于验证JWT令牌，并从中提取用户信息，以便进行授权和身份验证
	KeyAuth 中间件	基于密钥的认证，用于验证请求中的密钥，并根据密钥来授权访问
	Paseto 中间件	用于处理 Paseto 令牌，可以验证Paseto令牌，并从中提取用户信息，以便进行授权和身份验证
	BasicAuth 中间件	用于HTTP基本身份验证，可以用于保护应用程序的敏感资源
网络安全	CSRF 中间件	用于防止跨站请求伪造攻击（CSRF），可以通过生成和验证token来保护请求的安全性。
	CORS 中间件	用于处理跨域资源共享，防止跨域资源共享（CORS）等安全问题
性能	Etag 中间件	用于生成和验证ETag，可以实现缓存验证和减少网络传输的数据量。
	HttpCache 中间件	基于Souin HTTP缓存的 Hertz 的分布式 HTTP 缓存中间件
	Cache 中间件	用于缓存请求数据，用于提高某些热点内容的查询速度等

Hertz Server 中间件的拓展

类型	中间件	简述
HTTP 通用能力	RequestID 中间件	用于生成和管理请求ID，可以方便地跟踪请求的流程。
	SSE 中间件	用于实现服务器推送事件（Server-Sent Events），可以实现实时通信。
	Gzip 中间件	用于压缩/解压缩HTTP Body，减少网络传输的数据量。
	Sessions 中间件	用于管理用户会话，可以实现用户登录状态的维护。
	Secure 中间件	用于设置HTTP响应头，提高应用的安全性，包括X-XSS-Protection、X-Frame-Options、X-Content-Type-Options等
可观测性	OpenTelemetry 中间件	用于分布式追踪和指标收集，可以跟踪请求的链路和性能，并收集应用的指标
	OpenTracing 中间件	用于分布式追踪，可以跟踪请求的链路和性能。
服务治理	OpenSergo 中间件	用于接入OpenSergo项目，提供服务治理在内的多种能力。
	Sentry 中间件	用于集成Sentry错误监控平台，可以实现应用的错误监控和告警。
	Limiter 中间件	用于限制请求的频率，可以防止恶意攻击和DDoS攻击等
其他	Recovery 中间件	用于在发生panic时恢复应用程序，避免应用程序崩溃。
	I18n 中间件	用于国际化支持，可以根据用户的语言设置返回不同的响应内容。

服务端中间件

Hertz 服务端中间件是 HTTP 请求 - 响应周期中的一个函数，提供了一种方便的机制来检查和过滤进入应用程序的 HTTP 请求，例如记录每个请求或者启用 CORS。



中间件可以在请求更深入地传递到业务逻辑之前或之后执行：

- 中间件可以在请求到达业务逻辑之前执行，比如执行身份认证和权限认证
- 中间件也可以在执行过业务逻辑之后执行，比如记录响应时间和从异常中恢复。

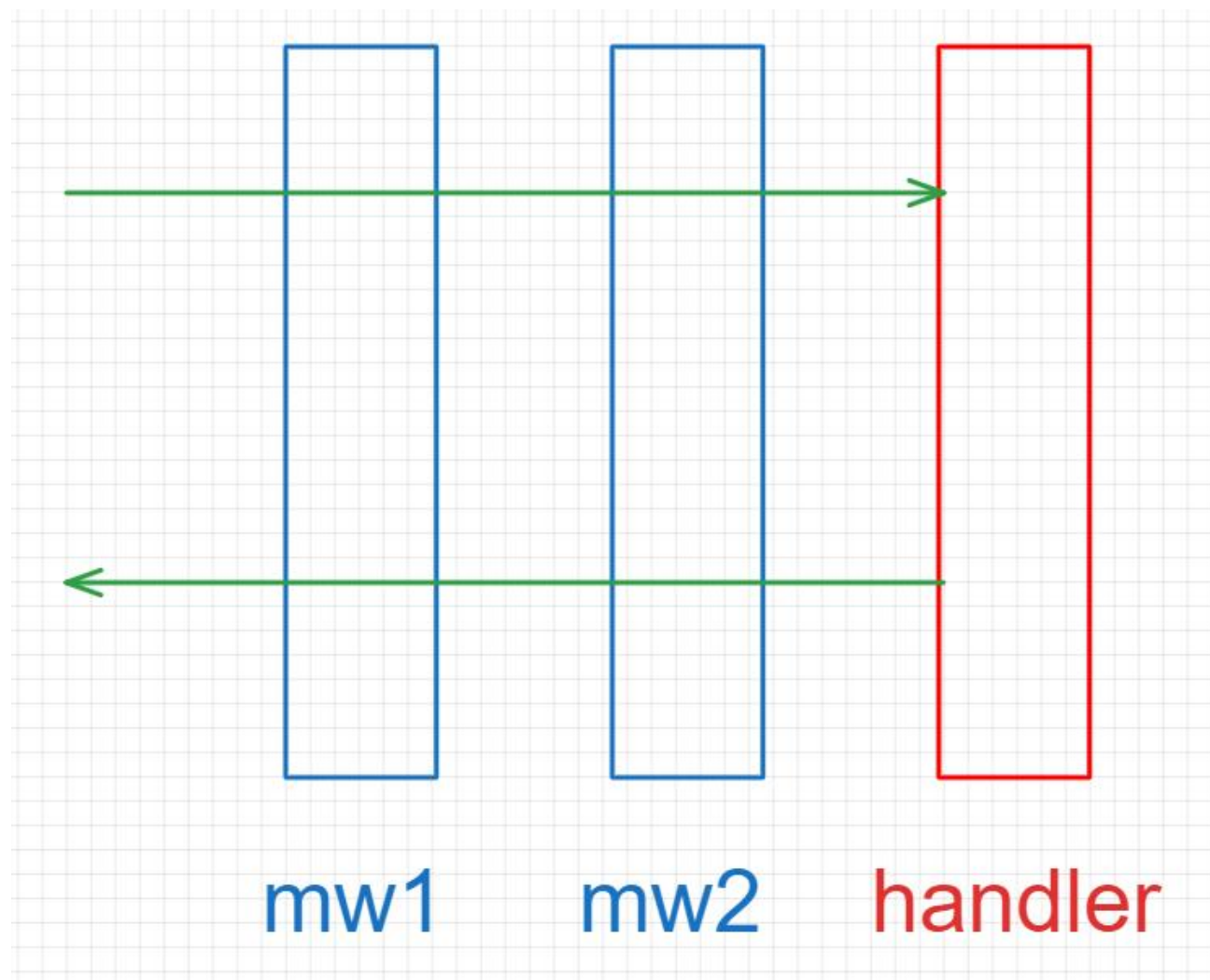
Example

```
func main() { new *
    h := server.Default()
    h.Use(func(ctx context.Context, c *app.RequestContext) {
        fmt.Println(a...: "pre-handle-1")
        c.Next(ctx)
        fmt.Println(a...: "post-handle-1")
    }, func(ctx context.Context, c *app.RequestContext) {
        fmt.Println(a...: "pre-handle-2")
        c.Next(ctx)
        fmt.Println(a...: "post-handle-2")
    })
    h.GET(relativePath: "/mw", func(ctx context.Context, c *app.RequestContext) {
        fmt.Println(a...: "in-handle")
    })
    h.Spin()
}
```

output:

pre-handle-1
pre-handle-2
in-handle
post-handle-2
post-handle-1

Example



output:

pre-handle-1
pre-handle-2
in-handle
post-handle-2
post-handle-1



02

如何编写 Hertz Middleware

How?

前置准备

开发目的，可以解决的问题

- 项目业务功能需求
- Hertz Issue

中间件的基本架构

- 服务端中间件，客户端中间件
- 中间件执行的时机

实现方式

- 是否有其他框架的实现参考（gin, fiber...）
- 是否可以依赖其他开源库（gorilla...）

中间件实现示例-RequestID


Hertz 通过使用 RequestID 中间件，可以在响应头中添加一个键为 X-Request-ID 的标识符，如果在请求头中设置了 X-Request-ID 字段，则会在响应头中将 X-Request-ID 原样返回。

Request ID 中间件提供了默认配置，用户也可以依据业务场景使用 WithGenerator，WithCustomHeaderStrKey，WithHandler 函数对以下配置项进行定制。

中间件实现示例-RequestID

RequestID 是一个服务端中间件，在业务 Handler 之前执行。

```
// New initializes the RequestID middleware.  
func New(opts ...Option) app.HandlerFunc {
```

```
    return func(ctx context.Context, c *app.RequestContext) {  
        // Get id from request  
        rid := c.Request.Header.Get(string(cfg.headerKey))  
        if rid == "" {  
            rid = cfg.generator(ctx, c)  
        }  
        headerXRequestID = string(cfg.headerKey)  
        if cfg.handler != nil {  
            cfg.handler(ctx, c, rid)  
        }  
        // Set the id to ensure that the request id is in the response  
        c.Header(headerXRequestID, rid)  
        ctx = context.WithValue(ctx, headerXRequestID, rid)  
        c.Next(ctx)   
    }  
}
```

```
h.Use(  
    requestid.New(),  
)
```

中间件实现示例-RequestID

Functional Options 编程模式提供多样化选项

```
// WithGenerator set generator function
func WithGenerator(g Generator) Option { 4 usages  👤 Shuyang Fan
    return func(cfg *config) {
        |   cfg.generator = g
    }
}

// WithCustomHeaderStrKey set custom header key for request id
func WithCustomHeaderStrKey(s HeaderStrKey) Option { 3 usages  👤 S
    return func(cfg *config) {
        |   cfg.headerKey = s
    }
}

// WithHandler set handler function for request id with context
func WithHandler(handler Handler) Option { 2 usages  👤 Shuyang Fan
    return func(cfg *config) {
        |   cfg.handler = handler
    }
}
```

- 直觉式的编程;
- 高度的可配置化;
- 很容易维护和扩展;
- 自文档;
- 容易上手

中间件实现示例-RequestID

- 单测
- 使用文档
- Examples

RequestID (This is a community driven project)

Request ID middleware for Hertz framework, inspired by [requestid](#). This project would not have been possible without the support from the CloudWeGo community and previous work done by the gin community.

- Adds an identifier to the response using the `X-Request-ID` header.
- Passes the `X-Request-ID` value back to the caller if it's sent in the request headers.

Install

```
go get github.com/hertz-contrib/requestid
```

Usage

usage	description
default	This is using requestid by default
custom key	How to use requestid for custom key
custom generator	How to use requestid for custom generator
custom handler	How to use requestid for custom handler
get requestid	How to get requestid
log with hertzlogrus	How to log requestid with hertzlogrus



03

Hertz-Session 解读

biz-demo

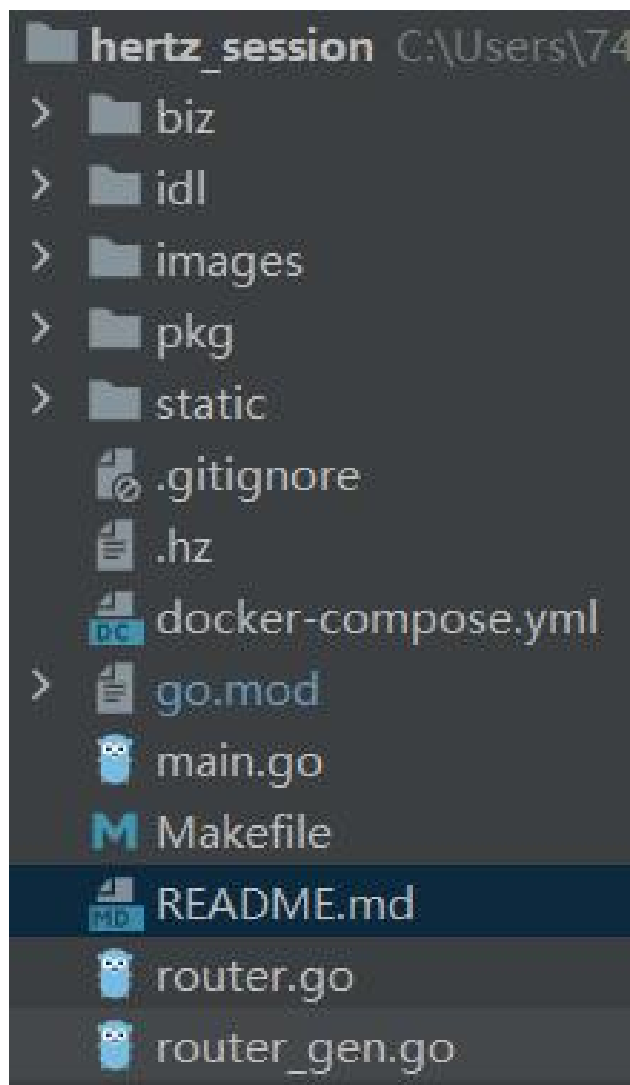
简介

Hertz-Session 旨在帮助用户快速上手 Hertz 的 **Session** 中间件和 **CSRF** 中间件，并展示基于 Redis 的分布式 Session 解决方案。

主要特点如下：

- 使用 thrift IDL 定义 HTTP 接口
- 使用 hz 工具生成脚手架代码
- 使用 hertz-contrib/sessions 存储 Session
- 使用 hertz-contrib/csrf 防御跨站请求伪造攻击
- 使用 GORM 和 MySQL
- 使用 AdminLTE 的前端页面

目录结构



- biz 存放主要业务逻辑代码，包括处理 HTTP 请求的 handler，进行数据库操作的 dal，中间件 mw
- idl 存放 thrift IDL
- pkg 为一些工具方法，业务常量，errmsg，模板渲染等
- static 存放前端的静态文件，均截取自 AdminLTE
- 其他主目录下的文件包括主启动 main.go，docker 配置文件等

中间件流程



中间件的使用 - Session

基于 Redis 的分布式 Session 解决方案是指将不同服务器的 Session 统一存储在 Redis 或 Redis 集群中，旨在解决分布式系统下多个服务器的 Session 不同步的问题。

1. 所以这里我们通过使用 Hertz 的 Session 中间件将 Session 存储在 Redis 中，初始化代码如下：

```
func InitSession(h *server.Hertz) { 1 usage  👤 Lorain
    store, err := redis.NewStore(consts.MaxIdleNum, consts.TCP, consts.RedisAddr, consts.RedisPasswd, []byte(consts.SessionSecretKey))
    if err != nil : err *
    h.Use(sessions.New(consts.HertzSession, store))
}
```

- 首先通过传入地址密码等参数使用 redis.NewStore 与 Redis 建立连接
- 通过 h.Use 方法使用 Session 中间件，并刚刚返回的存储连接对象传入，其中第一个形参为 Cookie 的名字，此处通过定义常量传入

中间件的使用 - Session

2. 在初始化完毕后，我们就可以在用户通过验证并登录后将用户的信息（此处为用户名）存储在 Session 中，以下为 Login Handler Session 部分的核心代码：

```
session := sessions.Default(c)
session.Set(consts.Username, req.Username)
_ = session.Save()
```

- 首先通过 `sessions.Default` 获取一个 Session 对象
- 通过 `Set` 方法将用户名存储到 Session 中
- 最后调用 `Save` 方法进行保存 Session 对象

这样在用户登录后我们就会在 Redis 存储的 Session 对象中保存一份用户的信息，用户下一次就无需进行登录即可访问对应的页面了。

中间件的使用 - Session

3. 在本例中，只设置了 index.html 这一个主页，用户对主页进行访问时会进行判断，如果用户已经登录则可以访问，如果未登录则会重定向到登录页面 login.html，判断是否登录的核心代码如下：

其实就是跟第二步的 Login 是差不多的流程，只不过把 Set 换成了 Get，并且不用 Save

```
// index.html
h.GET(relativePath: "/index.html", func(ctx context.Context, c *app.RequestContext) {
    if !utils.IsLogout(ctx, c) {
        token = csrf.GetToken(c)
    }
    session := sessions.Default(c)
    username := session.Get(consts.Username)
    if username == nil {
        c.HTML(http.StatusOK, name: "index.html", hutils.H{
            "message": utils.BuildMsg(consts.PageErr),
            "token":    utils.BuildMsg(token),
        })
        c.Redirect(http.StatusMovedPermanently, []byte("/login.html"))
        return
    }
    c.HTML(http.StatusOK, name: "index.html", hutils.H{
        "message": utils.BuildMsg(username.(string)),
        "token":    utils.BuildMsg(token),
    })
})
})
```

中间件的使用 - Session

4. 最后在用户登出时，我们需要对用户的 **Session** 进行清理，核心代码如下：

```
session := sessions.Default(c)
session.Delete(consts.Username)
_ = session.Save()
```

跟上面同理，注意需要 **Save** 否则这次删除操作就是无效的。

以上就是 **Session** 中间件的使用，是不是非常简单，**Session** 中间件将大多数需要考虑的复杂逻辑都进行了封装，例如不同用户 **Session** 在 **Redis** 中的存储，而我们只需要调用简单的接口即可完成对应的业务流程。

中间件的使用 - CSRF

CSRF (Cross-Site Request Forgery) 攻击是一种常见的网络安全威胁，它利用了Web应用程序对用户另一个不相关的网站上已经进行过认证的会话。在 CSRF 攻击中，攻击者通过欺骗用户使其执行恶意操作，而用户自己并不知情。

为了防止 CSRF 攻击，开发人员可以采取以下措施：

- 实施CSRF令牌：网站可以生成并在每个页面请求中包含一个随机的CSRF令牌。这个令牌与用户的会话绑定，并且在提交请求时需要验证令牌的有效性。
- 同源策略：浏览器的同源策略限制了跨域请求的执行。网站A可以设置适当的同源策略，确保仅接受来自同一域的请求。
- 双重验证：网站A可以要求用户在关键操作（如修改个人信息或进行金钱交易）之前进行额外的身份验证，例如输入密码或提供二次认证代码。

中间件的使用 - CSRF

我们这里利用 CSRF 中间件去保护注册，登录的 POST 表单提交以及 POST 登出。

1. 我们同样来看一下 CSRF 的初始化和使用，具体代码如下：

```
func InitCSRF(h *server.Hertz) { 1 usage  👤 Lorain
    h.Use(csrf.New(
        csrf.WithSecret(consts.CSRFSecretKey),
        csrf.WithKeyLookup(consts.CSRFKeyLookup),
        csrf.WithNext(utils.IsLogout),
        csrf.WithErrorFunc(func(ctx context.Context, c *app.RequestContext) {
            c.String(http.StatusBadRequest, errors.New(consts.CSRFErr).Error())
            c.Abort()
        }),
    ))
}
```

- 调用 `h.Use` 使用 CSRF 中间件
- 利用 `csrf.WithSecret` 定义令牌
- 利用 `csrf.WithKeyLookup` 定义从表单中获取 CSRF Token，默认为请求头中获取
- 利用 `csrf.WithNext` 跳过没有登录的情况，即不存在 Cookie
- 利用 `csrf.WithErrorFunc` 自定义异常处理

中间件的使用 - CSRF

2. 初始化完毕后我们只需在登录后的表单提交时通过 hidden 域提交生成的 CSRF Token，中间件会自动帮助我们验证是否有效，如果出现错误则会走刚刚定义的异常处理函数，由于本 Demo 使用的是模板渲染的模式，核心代码如下（以注册为例，登录，登出同理）：

判断用户已登录后获取对应的 Token 然后放到表单中提交即可，剩下的交给中间件自行处理。

```
h.GET( relativePath: "/register.html", func(ctx context.Context, c *app.RequestContext) {  
    if !utils.IsLogout(ctx, c) {  
        token = csrf.GetToken(c)  
    }  
    c.HTML(http.StatusOK, name: "register.html", hutils.H{  
        "message": utils.BuildMsg( msg: "Register a new membership"),  
        "token":    utils.BuildMsg(token),  
    })  
})
```

```
<div>  
    <input type="hidden" name="csrf" value="{[{ .token | BuildMsg }]}">  
</div>
```

运行效果演示

总结回顾

- 什么是 Hertz Middleware
- 如何编写 Hertz Middleware
- Hertz-Session 解读

欢迎大家为 Hertz 贡献更多优秀的中间件:)



THANKS