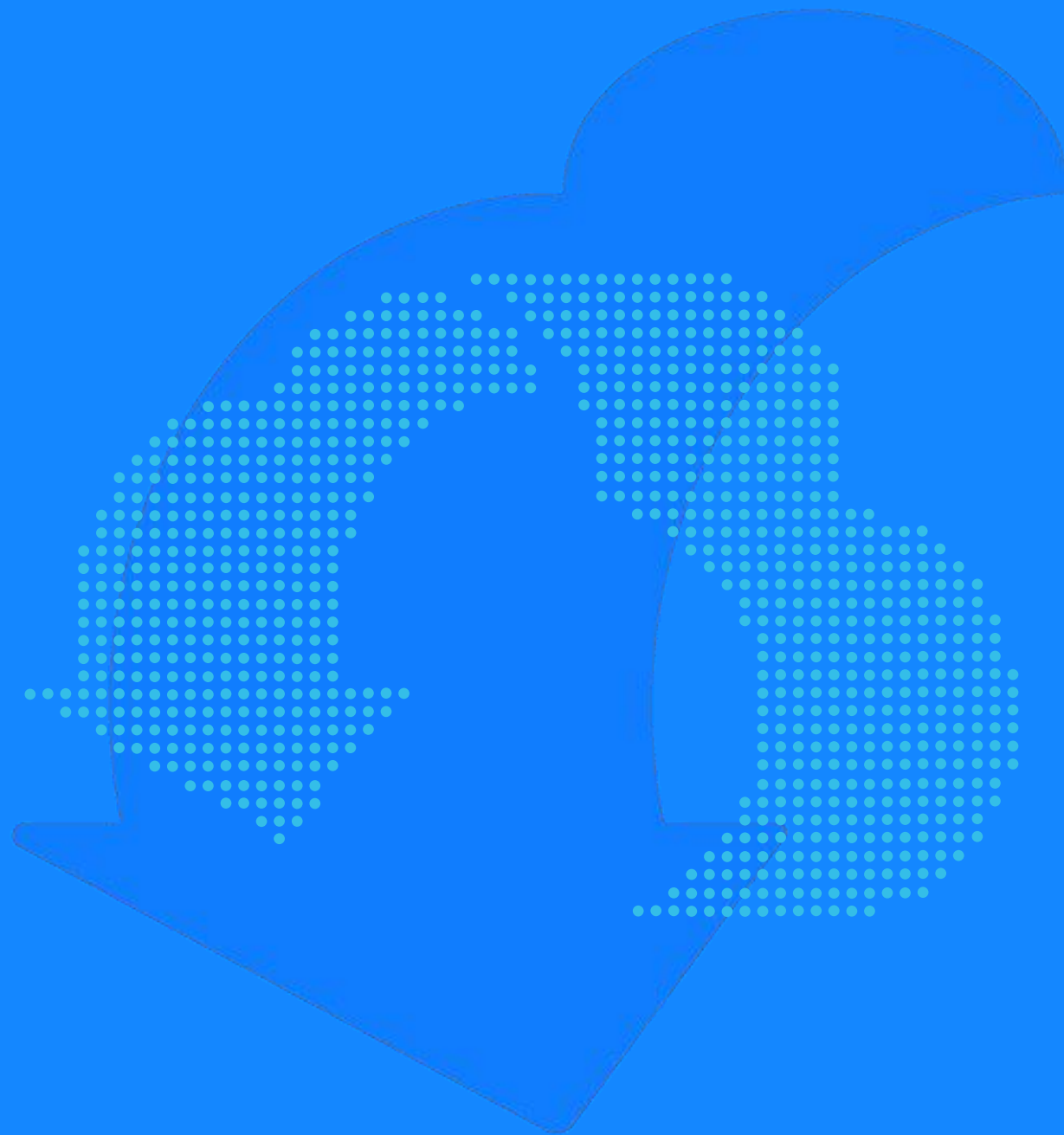


快速掌握适合自己业务的 Hertz 中间件

袁鑫浩


2023/6/17



自我介绍



- 重庆邮电大学计科专业大二在读
- 太极图形 DevOps 团队实习生
- CloudWeGo Member & Hertz Committer
- Hertz Casbin / CSRF / Loadbalance 中间件作者



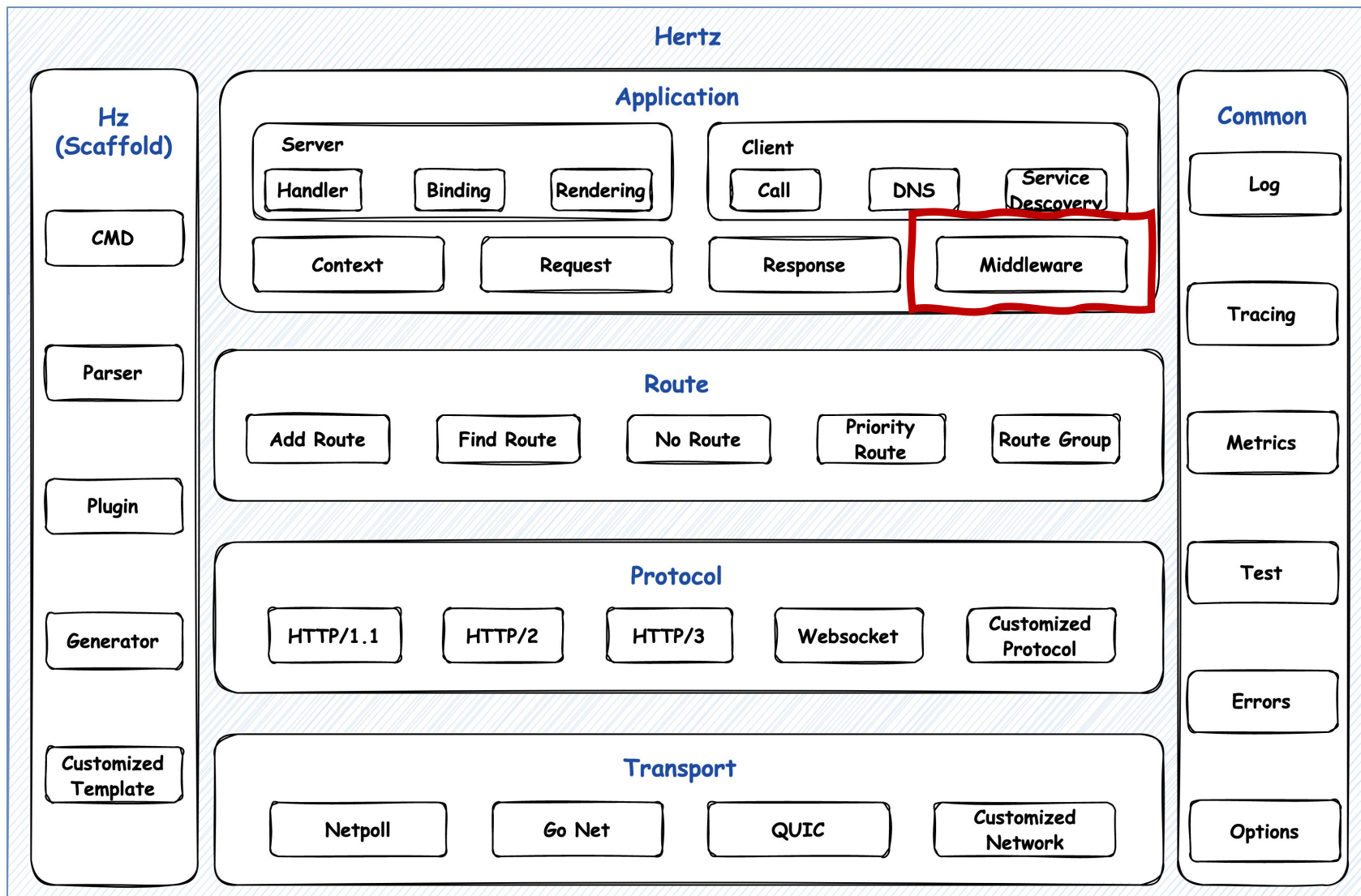
00

Hertz 和 Hertz 中间件

什么是 Hertz 和 Hertz 中间件

Hertz 介绍

Hertz 是一个 Golang 微服务 HTTP 框架，具有高易用性、高性能、高扩展性等特点。



Hertz 中间件的作用

1. **预处理请求**：中间件可以在请求到达业务逻辑之前对请求进行预处理，例如解析请求参数、验证请求头、权限等。
2. **后处理响应**：中间件可以在请求处理完成后对响应进行后处理，例如添加响应头、修改响应数据、记录日志等。
3. **实现逻辑复用**：中间件可以将某些通用逻辑封装起来，以达到复用的目的。例如，认证中间件可以用于多个业务逻辑，避免在每个业务逻辑中都写认证逻辑。
4. **实现功能扩展**：中间件可以用于实现框架本身不具备的功能，例如跨域资源共享、请求解压缩、性能分析等。

目录

01.

学习中间件第一步 —— 使用中间件

进行 CSRF 中间件的简单实用

02.

学习中间件第二步 —— 窥探底层实现

了解 CSRF 中间件的底层原理

03.

学习中间件第三步 —— 学习优秀中间件设计

了解服务注册发现中间件的设计

01

学习中间件第一步 —— 使用中间件

进行 CSRF 中间件的简单实用

什么是 CSRF 中间件

项目地址

<https://github.com/hertz-contrib/csrf>

CSRF 攻击

跨站点请求伪造（Cross Site Request Forgery）又被称作 CSRF，是恶意站点或程序通过已认证用户的浏览器在受信任站点上执行非正常操作。可进行的恶意操作局限于已在网站通过身份验证的用户的功能。

如何通过中间件进行预防

为了防止 CSRF 攻击，可以通过在请求中添加 Token 来进行防御。服务器通过校验请求是否携带正确的 Token，来把正常的请求和攻击的请求区分开，从而防范 CSRF 的攻击。

如何使用 CSRF 中间件

CSRF 实战完整地址

<https://juejin.cn/post/7173187571088900104>

1. 首先我们调用 sessions 拓展自定义一个测试用的 session，因为后续的 token 即是通过 session 进行的生成。接下来直接使用 CSRF 中间件即可。
2. 我们注册两个路由进行测试，先使用 GET 方法调用 *GetToken()* 函数获得通过 CSRF 中间件产生的 token。由于我们没有自定义 *KeyLookup* 选项，所以默认的值 *header:X-CSRF-TOKEN*，我们将获得的 token 放入到 Key 为 *X-CSRF-TOKEN* 头部中即可，若 token 无效或 Key 值设置不正确都会调用 *ErrorFunc* 返回错误。

```
1 package main
2
3 import (
4     "context"
5
6     "github.com/cloudwego/hertz/pkg/app"
7     "github.com/cloudwego/hertz/pkg/app/server"
8     "github.com/hertz-contrib/csrf"
9     "github.com/hertz-contrib/sessions"
10    "github.com/hertz-contrib/sessions/cookie"
11 )
12
13 func main() {
14     h := server.Default()
15
16     store := cookie.NewStore([]byte("secret"))
17     h.Use(sessions.New("session", store))
18     h.Use(csrf.New())
19
20     h.GET("/protected", func(c context.Context, ctx *app.RequestContext) {
21         ctx.String(200, csrf.GetToken(ctx))
22     })
23
24     h.POST("/protected", func(c context.Context, ctx *app.RequestContext) {
25         ctx.String(200, "CSRF token is valid")
26     })
27
28     h.Spinner()
29 }
```

02

学习中间件第二步 —— 窥探底层实现

了解 CSRF 中间件的底层原理

窥探 CSRF 中间件底层设计

这里碍于篇幅，仅展示出了核心的实现代码

1. 从传入的选项中获取配置。
2. 解析 KeyLookup 选项并根据其值设置 Extractors 选项。
3. 从会话中获取 salt。
4. 调用 Extractor 函数提取 token。
5. 将 salt 和 secret 组合并哈希，然后与提取的 token 进行比较。
6. 如果 token 有效，则继续处理请求，否则调用错误处理函数。

其中，salt 和 secret 都是用于哈希生成 token 的参数。Extractors 选项用于从请求中提取 token，支持从 header、form、query 和 param 中提取。如果没有设置，则默认从 header 中提取。

```
// New validates CSRF token.
func New(opts ...Option) app.HandlerFunc {
    cfg := NewOptions(opts...)
    // ...
    if cfg.Extractor == nil {
        // ...
    }
    return func(ctx context.Context, c *app.RequestContext) {
        // ...
        session := sessions.Default(c)
        c.Set(csrfSecret, cfg.Secret)

        if ignored(cfg.IgnoreMethods, string(c.Request.Method())) {
            // ...
        }

        salt, ok := session.Get(csrfSalt).(string)
        if !ok || len(salt) == 0 {
            // ...
        }

        token, err := cfg.Extractor(ctx, c)
        if err != nil {
            // ...
        }

        if tokenize(cfg.Secret, salt) != token {
            // ...
        }

        c.Next(ctx)
    }
}
```

窥探 CSRF 中间件底层设计

这里碍于篇幅，仅展示出了核心的实现代码

GetToken 函数用于生成 CSRF token，其主要流程如下：

1. 从会话中获取 salt。
2. 如果不存在，则生成一个随机的 salt，并保存到会话中。
3. 通过调用 tokenize 函数生成 token。
4. 将生成的 token 保存到请求上下文中，并返回。

```
// GetToken returns a CSRF token.
func GetToken(c *app.RequestContext) string {
    session := sessions.Default(c)
    secret := c.MustGet(csrfSecret).(string)

    if t, ok := c.Get(csrfToken); ok {
        return t.(string)
    }

    salt, ok := session.Get(csrfSalt).(string)
    if !ok {
        salt = randStr(16)
        session.Set(csrfSalt, salt)
        session.Save()
    }
    token := tokenize(secret, salt)
    c.Set(csrfToken, token)

    return token
}

// tokenize generates token through secret and salt.
func tokenize(secret, salt string) string {
    h := sha256.New()
    io.WriteString(h, salt+"-"+secret)
    hash := base64.URLEncoding.EncodeToString(h.Sum(nil))

    return hash
}
```



03

学习中间件第三步 —— 学习优秀中间件设计

了解服务注册发现中间件的设计

服务发现

服务注册中心（Registry）

用于保存 RPC Server 的注册信息，当 RPC Server 节点发生变更时，Registry 会同步变更，RPC Client 感知后会刷新本地 内存中缓存的服务节点列表。

服务提供者（RPC Server）

在启动时，向 Registry 注册自身服务，并向 Registry 定期发送心跳汇报存活状态。

服务消费者（RPC Client）

在启动时，向 Registry 订阅服务，把 Registry 返回的服务节点列表缓存在本地内存中，并与 RPC Sever 建立连接。

在项目中使⤢用服务注册中间件

函数的输入参数是一个整数类型的 Port，用于指定服务端口。函数内部首先创建一个 consul 注册中心对象。在创建注册中心对象时，函数还调用了 **consul.WithCheck** 函数来设置健康检查参数，用于检测服务是否可用。

接着，函数使用 snowflake 库来生成一个基于 Twitter 的分布式唯一 ID，用于作为服务的唯一标识。这个 ID 会被转换为 36 进制字符串，并存储在注册信息对象的标签中。

最后，函数会返回一个包含服务名、地址和标签信息的注册信息对象。其中服务名来自于全局配置中的服务名，地址由主机名和端口号组成，标签包含了上面生成的唯一 ID。

```
// InitRegistry to init consul
2 个用法 L2ncE
func InitRegistry(Port int) (registry.Registry, *registry.Info) {
    r, err := consul.NewConsulRegister(net.JoinHostPort(
        config.GlobalConsulConfig.Host,
        strconv.Itoa(config.GlobalConsulConfig.Port)),
        consul.WithCheck(&api.AgentServiceCheck{
            Interval:           consts.ConsulCheckInterval,
            Timeout:             consts.ConsulCheckTimeout,
            DeregisterCriticalServiceAfter: consts.ConsulCheckDeregisterCriticalServiceAfter,
        }))
    if err != nil {
        klog.Fatalf("new consul register failed: %s", err.Error())
    }

    // Using snowflake to generate service name.
    sf, err := snowflake.NewNode(node: 2)
    if err != nil {
        klog.Fatalf("generate service name failed: %s", err.Error())
    }
    info := &registry.Info{
        ServiceName: config.GlobalServerConfig.Name,
        Addr:        utils.NewNetAddr(consts.TCP, net.JoinHostPort(config.GlobalServerConfig.Host, strconv.Itoa(Port))),
        Tags: map[string]string{
            "ID": sf.Generate().Base36(),
        },
    }
    return r, info
}
```

```
server.WithRegistry(r),
```

在项目中使用时服务发现中间件

服务发现部分类似。函数内部创建一个 consul 服务发现中心对象。

最后，在创建新的客户端时会将服务发现中心的值传进去

```
// init resolver
r, err := consul.NewConsulResolver(fmt.Sprintf(format: "%s:%d",
    config.GlobalConsulConfig.Host,
    config.GlobalConsulConfig.Port))
if err != nil {
    klog.Fatalf(format: "new consul client failed: %s", err.Error())
}
```

```
// create a new client
c, err := userservice.NewClient(
    config.GlobalServerConfig.UserSrvInfo.Name,
    client.WithResolver(r), // service discovery
    client.WithLoadBalancer(loadbalance.NewWeightedBalancer()), // load balance
    client.WithMuxConnection(connNum: 1), // multiplexing
    client.WithSuite(tracing.NewClientSuite()),
    client.WithClientBasicInfo(&rpcinfo.EndpointBasicInfo{ServiceName: config.GlobalServerConfig.UserSrvInfo.Name}),
)
```


详解服务注册发现中间件

项目地址

<https://github.com/hertz-contrib/registry>

我们以 Consul 为例进行讲解，不同的组件实现基本差不多，代码行数也都在 200 ~ 300 行的样子，适合初学者进行学习。

在学习完之后可以去尝试参与一些社区的 *good first issue* 的贡献。

<https://github.com/cloudwego/hertz/issues/485>

85

Add a more detailed example for service discovery. #485

New issue

Open

2 of 7 tasks

li-jin-gou opened this issue on Dec 23, 2022 · 12 comments



li-jin-gou commented on Dec 23, 2022 · edited

Member

We hope more developers join CloudWeGo. This is a task issue that needs your help.

Is your feature request related to a problem? Please describe.

The current examples of service discovery are too simple and we need some complex examples to help users and we need to improve detailed code examples to the [registry](#).

picked

- ☐ zookeeper
- ☒ consul
- ☐ nacos
- ☐ etcd
- ☐ polaris
- ☐ eureka
- ☒ servicecomb

Additional context

- 能不能出一个post请求的方式哇 请求参数是json类型，孩子弄了半天也传不过去参数 [hertz-contrib/registry#30](#)
- <https://github.com/hertz-contrib/registry/>



li-jin-gou added the **good first issue** label on Dec 23, 2022

Assignees

- qiuyuyin
- dengWuuu
- chaoranz758

Labels

good first issue

Milestone

No milestone

Development

No branches or pull requests

Notifications

Customize

Unsubscribe

You're receiving notifications because you commented.

6 participants



Convert to discussion

相关地址

CloudWeGo 官网

<https://cloudwego.io>

Hertz 仓库地址

<https://github.com/cloudwego/hertz>

Hertz 中间件集合

<https://github.com/hertz-contrib>

CloudWeGo 微信公众号



THANKS