

# Open Payment Platform

基于CloudWeGo 实现 API Gateway

---

baiyutang

2023/3/8



## 关于我



王伟超

ID: baiyutang

CloudWeGo Reviewer

InfoQ 签约作者

# CONTENT

## 目录

01.

背景介绍

02.

架构设计

03.

工程实践

04.


总结





01


背景介绍

# 背景介绍



[Pull requests](#) [Issues](#) [Codespaces](#) [Marketplace](#) [Explore](#)

 + 


 [cloudwego / kitex](#) Public

[Edit Pins](#) [Unwatch](#) 111 [Fork](#) 625 [Star](#) 5.7k

[Code](#) [Issues](#) 44 [Pull requests](#) 7 [Discussions](#) [Actions](#) [Wiki](#) [Security](#) 3 [Insights](#)

## 关于泛化调用及如何从集群外访问具体的某服务 #419

[Edit](#)


 **baiyutang** on Apr 21, 2022 Collaborator

背景：  
服务间的调用，服务发现可以通过在客户端sdk做，选择一个服务节点，这个可以理解。但是集群外，如何从 C 端或者普通用户端走到具体某一个服务还没看明白，找到的资料例如可以通过[HTTP 映射泛化调用](#)，但是这里只有示例代码，看的不是很明白。

问题：  
按照 kitex 在内部的实践来看，怎么把服务暴露出去的。比如：

- 域名？
- nginx / apisix / kong 等网关二开？
- k8s ingress service 等云原生方案？

这里推荐或通常的做法是怎么样的？


[↑ 1](#) 

✓ Answered by **jayantxie** on May 6, 2022

我理解得看具体的业务需求。如果c端是走http接口来访问，可以通过暴露http网关，再由泛化映射到rpc服务。实际上还可以由业务来控制。例如业务对外暴露http接口，而只对内部微服务间的调用走rpc。

[View full answer ↓](#)


Category

 Q&A

Labels

None yet

2 participants



Notifications

[Unsubscribe](#)

You're receiving notifications because you're watching this repository.

[Lock conversation](#)

[Transfer this discussion](#)

[Pin discussion](#)

[Pin discussion to Q&A](#)

[Create issue from discussion](#)

[Delete discussion](#)

1 suggested answer · 1 reply

[Oldest](#) [Newest](#) [Top](#)

## 背景介绍



罗广明

谁有兴趣来给 Kitex 写一个使用泛化调用在 API 网关场景的完整 biz demo，可以贡献到这个仓库：[GitHub - cloudwego/biz-demo](#)

cloudwego/**biz-**

 github.com

...

**GitHub - cloudwego/biz-demo**

Contribute to cloudwego/biz-demo development by creating an account on GitHub.

0 Issues   3 Stars   3 Forks

## 支付开放平台

开放给服务商或商户提供收款记账等能力的服务入口

常见于支付宝、微信、银联等第三方或第三方支付渠道商

特别是前几年发展起来的聚合支付方向

## 支付开放平台

- 对外暴露的是 HTTP 接口，可以用 Hertz
- 需要加签、验签，可以演示 Hertz 自定义 middleware
- 业务模块清晰：商户、支付、对账、安全
- 关注工程化，如 ORM、分包、代码分层、错误统一定义及优雅处理

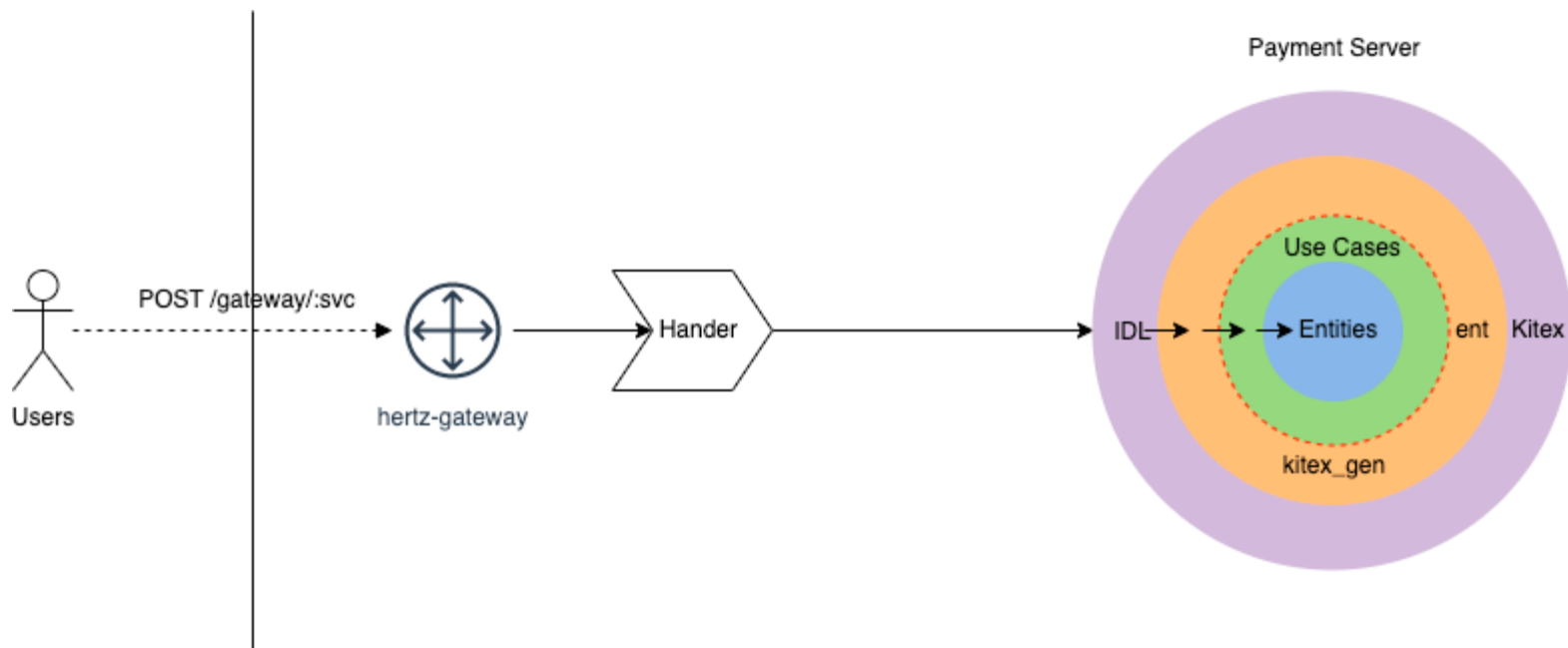




02

架构设计

# 整体架构



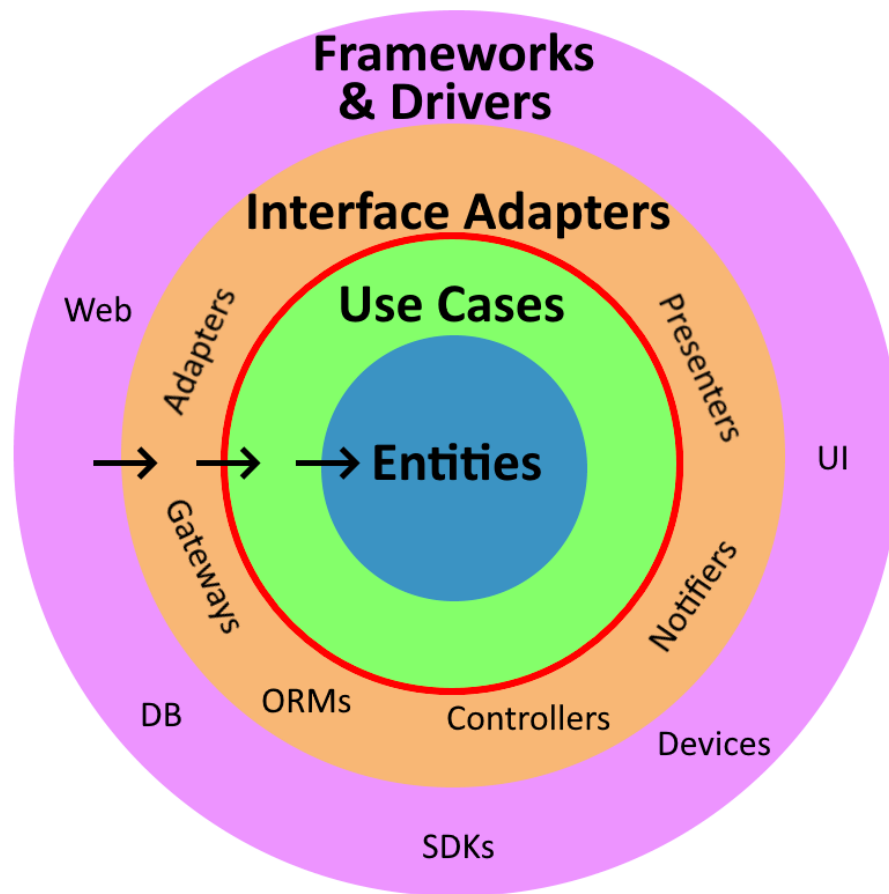
# 整体架构

```
1  .
2  ├── Makefile
3  ├── README.md
4  ├── cmd                                # RPC 服务入口
5  │   ├── payment
6  │   │   ├── main.go
7  │   │   ├── wire.go
8  │   │   └── wire_gen.go
9  ├── configs                            # demo 启动的初始化配置
10 │   ├── sql
11 │   └── payment.sql
12 ├── docker-compose.yml                 # demo 依赖的中间件环境
13 ├── go.mod
14 ├── go.sum
15 ├── hertz-gateway                      # 网关
16 │   ├── README.md
17 │   ├── biz                            # 网关主要业务逻辑
18 │   ├── main.go
19 │   ├── router.go
20 │   └── router_gen.go
21 ├── idl                                # IDL 文件目录, 泛化调用时会遍历一遍这里
22 │   ├── common.thrift
23 │   └── payment.thrift
24 ├── internal                           # RPC 业务逻辑
25 │   ├── README.md
26 │   └── payment                        # RPC 服务之一
27 ├── kitex_gen                          # IDL 协议生成的 Kitex 客户端代码
28 ├── pkg                                # 公共库
29 └── auth
```

# Hertz-gateway

```
1 .
2 |— README.md
3 |— biz          # 业务逻辑
4 |   |— errors   # 规范错误处理
5 |   |   |— errors.go
6 |   |— handler   # HTTP 请求处理逻辑
7 |   |   |— gateway.go
8 |   |— middleware # 请求处理链的 middleware
9 |   |   |— gateway_auth.go
10 |  |— router     # Hertz 生成目录
11 |  |   |— register.go
12 |  |— types      # 自定义数据, 如规范化响应体
13 |  |   |— response.go
14 |— main.go       # 程序入口文件
15 |— router.go     # 自定义路由规则, 我们在这里解析 IDL, 生成泛化调用的客户端
16 |— router_gen.go
```

# 整洁架构



## 整洁架构

- 独立于框架：系统的架构不应该与某个框架，某个库强绑定
- 可被测试：单测不能依赖外部存储或者环境，单测必须能在本地，只依赖自己的代码来快速运行
- 独立于数据库：不要写出强依赖你的存储组件的系统，底层的存储可能随着业务增长而不再使用

# Payment Server

```
1 .
2 |— Makefile
3 |— entity          # 业务实体
4 |   |— order.go
5 |— infrastructure
6 |   |— ent
7 |       |— client.go
8 |       |— ...
9 |       |— schema  # 表结构定义
10 |          |— order.go
11 |          |— tx.go
12 |   |— repository # 仓储层的具体实现
13 |       |— order_sql.go
14 |— usecase        # 业务逻辑
15 |   |— interface.go # 依赖的仓储层的抽象接口
16 |   |— service.go
17
```



03

工程实践



## 工程实践

1. 抽象业务，定义 IDL
2. 创建一个微服务模块（Payment Server）
3. 微服务工程目录
4. 网关，开放能力
5. 规范和细节调整

# 抽象业务

```
1 namespace go payment
2
3 struct UnifyPayReq {
4     1: string out_order_no,
5     2: i64 total_amount,
6     3: string subject,
7     4: string merchant_id,
8     5: string pay_way,
9     6: string app_id,
10    7: string sub_open_id,
11    8: string notify_url,
12    9: string client_ip,
13   10: i32 order_expiration
14 }
15
16 struct UnifyPayResp {
17     1: string merchant_id,
18     2: string sub_merchant_id,
19     3: string out_order_no,
20     4: string jspay_info,
21     5: string pay_way,
22 }
23
24 service PaymentSvc {
25     UnifyPayResp UnifyPay(1: UnifyPayReq req) (
26         api.post = '/payment/unifypay',
27         api.param = 'true'
28     )
29 }
```

# 实现微服务

```
1 var _ payment.PaymentSvc = (*Service)(nil)
2
3 type Service struct {
4     repo Repository
5 }
6
7 // UnifyPay implements payment.PaymentSvc.UnifyPay
8 func (s *Service) UnifyPay(ctx context.Context, req *payment.UnifyPayReq) (r *payment.UnifyPayResp, err error) {
9     // TODO
10    return &payment.UnifyPayResp{}, nil
11 }
12
13 // NewService create new service
14 func NewService(r Repository) payment.PaymentSvc {
15     return &Service{
16         repo: r,
17     }
18 }
```

# 工程目录

```
1 var _ payment.PaymentSvc = (*Service)(nil)
2
3 type Service struct {
4     repo Repository
5 }
6
7 // UnifyPay implements payment.PaymentSvc.UnifyPay
8 func (s *Service) UnifyPay(ctx context.Context, req *payment.UnifyPayReq) (r *payment.Un
9     o := entity.NewOrder()
10    o.PayWay = req.PayWay
11    o.SubOpenid = req.SubOpenid
12    o.MerchantID = req.MerchantId
13    o.OutOrderNo = req.OutOrderNo
14    o.Channel = "1"
15
16    err = s.repo.Create(ctx, o)
17    if err != nil {
18        return nil, err
19    }
20    return &payment.UnifyPayResp{
21        MerchantId:    o.MerchantID,
22        SubMerchantId: o.SubOpenid,
23        OutOrderNo:    o.OutOrderNo,
24        JspayInfo:     "xxxxxx",
25        PayWay:        o.PayWay,
26    }, nil
27 }
28
29 // NewService create new service
30 func NewService(r Repository) payment.PaymentSvc {
31     return &Service{
32         repo: r,
33     }
34 }
```

# 工程目录

/internal/payment/usecase/service.go

```
1 var _ payment.PaymentSvc = (*Service)(nil)
2
3 type Service struct {
4     repo Repository
5 }
6
7 // UnifyPay implements payment.PaymentSvc.UnifyPay
8 func (s *Service) UnifyPay(ctx context.Context, req *payment.UnifyPayReq) (r *payment.UnifyPayResp, err error) {
9     o := entity.NewOrder()
10    o.PayWay = req.PayWay
11    o.SubOpenid = req.SubOpenid
12    o.MerchantID = req.MerchantID
13    o.OutOrderNo = req.OutOrderNo
14    o.Channel = "1"
15
16    err = s.repo.Create(ctx, o)
17    if err != nil {
18        return nil, err
19    }
20    return &payment.UnifyPayResp{
21        MerchantId:    o.MerchantID,
22        SubMerchantId: o.SubOpenid,
23        OutOrderNo:    o.OutOrderNo,
24        JspayInfo:     "xxxxxx",
25        PayWay:        o.PayWay,
26    }, nil
27 }
28
29 // NewService create new service
30 func NewService(r Repository) payment.PaymentSvc {
31     return &Service{
32         repo: r,
33     }
34 }
```

## 工程目录

/internal/payment/usecase/interface.go

```
1 package usecase
2
3 import (
4     "context"
5
6     "github.com/cloudwego/biz-demo/open-payment-platform/internal/payment/entity"
7 )
8
9 // Repository is the interface of usecase dependent on.
10 // the interface is the part of usecase logic,so we put it here.
11 type Repository interface {
12     Create(ctx context.Context, order *entity.Order) error
13 }
```

# 工程目录

/internal/payment/entity/order.go

```
1 package entity
2
3 type Order struct {
4     ID          int
5     MerchantID  string
6     Channel     string
7     PayWay      string
8     OrderStatus int8
9     OutOrderNo  string
10    TotalAmount  uint64
11    Body         string
12    AuthCode    string
13    WxAppid     string
14    SubOpenid   string
15    JumpURL     string
16    NotifyURL   string
17    ClientIP    string
18    Attach      string
19    OrderExpiration string
20    ExtendParams string
21 }
22
23 func NewOrder() *Order {
24     return &Order{}
25 }
```

# 工程目录

/internal/payment/infrastructure/repository/order\_sql.go

```
1 package repository
2
3 // Create implements usecase.Repository.Create
4 func (o *OrderRepository) Create(ctx context.Context, order *entity.Order) error {
5     ret, err := o.db.Order.Create().
6         SetMerchantID(order.MerchantID).
7         SetPayWay(order.PayWay).
8         SetTotalAmount(order.TotalAmount).
9         SetBody(order.Body).
10        SetAttach(order.Attach).
11        SetChannel(order.Channel).
12        SetClientIP(order.ClientIP).
13        SetAuthCode(order.AuthCode).
14        SetJumpURL(order.JumpURL).
15        SetNotifyURL(order.NotifyURL).
16        SetOrderExpiration(order.OrderExpiration).
17        SetSubOpenid(order.SubOpenid).
18        SetOutOrderNo(order.OutOrderNo).
19        SetWxAppid(order.WxAppid).
20        SetOrderStatus(order.OrderStatus).
21        SetExtendParams(order.ExtendParams).
22        Save(ctx)
23     if err != nil {
24         return err
25     }
26     order.ID = ret.ID
27     return nil
28 }
29
30 // NewOrderSQL creates a new OrderRepository.
31 // This is the concrete implementation of Repository with SQL.
32 func NewOrderSQL(dbClient *ent.Client) usecase.Repository {
33     return &OrderRepository{
34         db: dbClient,
35     }
36 }
```



# 工程目录

/internal/payment/infrastructure/ent/schema/order.go

```
1 package schema
2
3 import (
4     "entgo.io/ent"
5     "entgo.io/ent/schema/field"
6 )
7
8 // Order holds the schema definition for the Order entity.
9 type Order struct {
10     ent.Schema
11 }
12
13 // Fields of the Order.
14 func (Order) Fields() []ent.Field {
15     return []ent.Field{
16         field.String("merchant_id"),
17         field.String("channel"),
18         field.String("pay_way"),
19         field.String("out_order_no"),
20         field.Uint64("total_amount"),
21         field.String("body"),
22         field.Int8("order_status"),
23         field.String("auth_code"),
24         field.String("wx_appid"),
25         field.String("sub_openid"),
26         field.String("jump_url"),
27         field.String("notify_url"),
28         field.String("client_ip"),
29         field.String("attach"),
30         field.String("order_expiration"),
31         field.String("extend_params"),
32     }
33 }
```

# 网关

/hertz-gateway/router.go

```
1 // registerGateway registers the router of gateway
2 func registerGateway(r *server.Hertz) {
3     group := r.Group("/gateway").Use(middleware.GatewayAuth()...)
4     idlPath := "./idl/"
5     c, err := os.ReadDir(idlPath)
6     if err != nil {
7         hlog.Fatalf("new thrift file provider failed: %v", err)
8     }
9     nacosResolver, err := resolver.NewDefaultNacosResolver()
10    if err != nil {
11        hlog.Fatalf("err:%v", err)
12    }
13
14    for _, entry := range c {
15        svcName := strings.ReplaceAll(entry.Name(), ".thrift", "")
16
17        provider, err := generic.NewThriftFileProvider(entry.Name(), idlPath)
18        if err != nil {
19            hlog.Fatalf("new thrift file provider failed: %v", err)
20            break
21        }
22        g, err := generic.HTTPThriftGeneric(provider)
23        if err != nil {
24            hlog.Fatal(err)
25        }
26        cli, err := genericclient.NewClient(
27            svcName,
28            g,
29            // ...
30        )
31        // ...
32        group.POST("/:svc", handler.Gateway)
33    }
34 }
```

# 网关

/hertz-gateway/biz/handler/gateway.go

```
1 package handler
2
3 type requiredParams struct {
4     Method      string `form:"method,required" json:"method"`
5     MerchantId   string `form:"merchant_id,required" json:"merchant_id"`
6     BizParams    string `form:"biz_params,required" json:"biz_params"`
7 }
8
9 var SvcMap sync.Map // key->value: svcName->genericclient.Client
10
11 // Gateway handle the request with the query path of prefix `/gateway`.
12 func Gateway(ctx context.Context, c *app.RequestContext) {
13     svcName := c.Param("svc")
14     cli, ok := SvcMap.load(svcName)
15     // ...
16     req, err := http.NewRequest(http.MethodPost, "", bytes.NewBuffer([]byte(params.B
17     // ...
18     req.URL.Path = fmt.Sprintf("/%s/%s", svcName, params.Method)
19     customReq, err := generic.FromHTTPRequest(req)
20     // ...
21     resp, err := cli.GenericCall(ctx, "", customReq)
22     respMap := make(map[string]interface{})
23     if err != nil {
24         // ...
25     }
26     realResp, ok := resp.(*generic.HTTPResponse)
27     // ...
28     realResp.Body[types.ResponseErrCode] = 0
29     realResp.Body[types.ResponseErrMsg] = "ok"
30     c.JSON(http.StatusOK, realResp.Body)
31 }
```



## 效果演示



04

总结

## 关于项目

两大项目：Kitex、Hertz

两个特性：泛化调用、整洁架构

演示理念：简洁、清晰

# 关于工程化

## cwgo

### 概览

cwgo 是 CloudWeGo All in one 代码生成工具，整合了各个组件的优势，提高开发者提体验。

cwgo 工具可以方便生成工程化模版，其主要功能特点如下：

### 工具特点

- 用户友好生成方式

cwgo 工具同时提供了交互式命令行和静态命令行两种方式。交互式命令行可以低成本生成代码，不用再去关心传递什么参数，也不用执行多次命令，适合大部分用户；而对高级功能有需求的用户，仍可使用常规的静态命令行构造生成命令。

- 支持生成工程化模板

cwgo 工具支持生成 MVC 项目 Layout，用户只需要根据不同目录的功能，在相应的位置完成自己的业务代码即可，聚焦业务逻辑。

- 支持生成 Server、Client 代码

cwgo 工具支持生成 Kitex、Hertz 的 Server 和 Client 代码，提供了对 Client 的封装。用户可以开箱即用的调用下游，免去封装 Client 的繁琐步骤

- 支持生成数据库代码

cwgo 工具支持生成数据库 CURD 代码。用户无需再自行封装繁琐的 CURD 代码，提高用户的工作效率。

- 支持回退为 Kitex、Hz 工具

如果之前是 Kitex、Hz 的用户，仍然可以使用 cwgo 工具。cwgo 工具支持回退功能，可以当作 Kitex、Hz 使用，真正实现一个工具生成所有。

## 关于社区

- 自由
- 开放
- 彼此成就



## 参考

- 泛化调用 <https://www.cloudwego.io/zh/docs/kitex/tutorials/advanced-feature/generic-call/>
- Proposal: 工程化模板或标准化的讨论 <https://github.com/cloudwego/kitex/issues/500>
- Clean Architecture – The Dependency Rule <https://codecoach.co.nz/clean-architecture-the-dependency-rule/>

THANKS