万信金融 第2章 讲义-开户

1 用户登录

1.1 传统实现方式

注册功能搞定后，紧接着就该实现登录功能了。这个功能对于大家来说应该是很熟悉的，业务熟悉，代码也熟悉，传统实现方式的思路：查询数据库，确定账号和密码是否存在(正确)。

1. 接口定义

   在AccountAPI接口中定义登录方法login：

```
/**
     * 用户登录
     * @param accountLoginDTO  封装登录请求数据
     * @return
*/
RestResponse<AccountDTO> login(AccountLoginDTO accountLoginDTO);
```

   在AccountController中实现login方法：

```
@ApiOperation("用户登录")
@ApiImplicitParam(name = "accountLoginDTO",  value = "登录信息",  required =
true,
                                    dataType = "AccountLoginDTO",  paramType
= "body")
@PostMapping(value = "/1/accounts/session")
@Override
public RestResponse<AccountDTO> login(@RequestBody AccountLoginDTO
accountLoginDTO) {
   return null;
}
```

2. 功能实现

   在AccountService中新增登录接口login：

```
/**
 登录功能
 @param accountLoginDTO 封装登录请求数据
 @return 用户及权限信息
*/
AccountDTO login(AccountLoginDTO accountLoginDTO);
```

   在AccountServiceImpl类中实现login方法：

```
@Override
public AccountDTO login(AccountLoginDTO accountLoginDTO) {
   Account account = null;
   if (accountLoginDTO.getDomain().equalsIgnoreCase("c")) {
```

```
        account = getAccountByMobile(accountLoginDTO.getMobile());//获取c端用户
    } else {
        account = getAccountByUsername(accountLoginDTO.getUsername());//获取b端
用户
    }
    if (account == null) {
        throw new BusinessException(AccountErrorCode.E_130104); // 用户不存在
    }
    AccountDTO accountDTO = convertAccountEntityToDTO(account);
    if (smsEnable) {// 如果smsEnable=true，说明是短信验证码登录，不做密码校验
        return accountDTO;
    }  //验证密码
    if (PasswordUtil.verify(accountLoginDTO.getPassword(),
account.getPassword())) {
        return accountDTO;
    }
    throw new BusinessException(AccountErrorCode.E_130105);
}

/**
     根据手机获取账户信息
     @param mobile 手机号
     @return 账户实体
*/
private Account getAccountByMobile(String mobile) {
     return getOne(new QueryWrapper<Account>().lambda()
                                        .eq(Account::getMobile,  mobile));
}

/**
     根据用户名获取账户信息
     @param username 用户名
     @return 账户实体
*/
private Account getAccountByUsername(String username) {
     return getOne(new QueryWrapper<Account>().lambda()
                                       .eq(Account::getUsername,  username));
}
/**
     entity转为dto
     @param entity对象
     @return dto对象
*/
private AccountDTO convertAccountEntityToDTO(Account entity) {
     if (entity == null) {
          return null;
     }
     AccountDTO dto = new AccountDTO();
     BeanUtils.copyProperties(entity, dto);
     return dto;
}
```

3. 完善AccountController代码，调用AccountService完成登录功能：

```
@ApiOperation("用户登录")
@ApiImplicitParam(name = "accountLoginDTO", value = "登录信息", required =
true,
                                    dataType = "AccountLoginDTO", paramType
= "body")
@PostMapping(value = "/1/accounts/session")
@Override
public RestResponse<AccountDTO> login(@RequestBody AccountLoginDTO
accountLoginDTO) {
   return RestResponse.success(accountService.login(accountLoginDTO));
}
```

1.2 问题思考

传统登录实现方式在应付分布式、微服务场景时存在的问题：

1. 每个微服务都要进行登录校验，十分麻烦，我们需要的是单点登录
2. 会话保持问题
3. 认证方式单一，无法适应各种认证场景(扫码，指纹...)，毫无扩展性
4. ... ...

P2P平台作为网络贷款平台，采用了前后端分离、分布式、微服务等架构，这就决定了传统的登录实现方式在这里无法胜任。为了解决这个问题，我们要在P2P平台引入独立的UAA服务。 UAA全称是User Account and Authentication，简称为认证服务，UAA服务使用Spring Security+Oauth2+JWT技术栈实现，结合前面的网关服务(gateway)即可搞定P2P平台的认证和授权业务功能。

1.3 部署UAA认证服务

我们需要能够将UAA服务集成到P2P项目中并实现用户登录和登录拦截，请参考“UAA服务集成指南.pdf”。

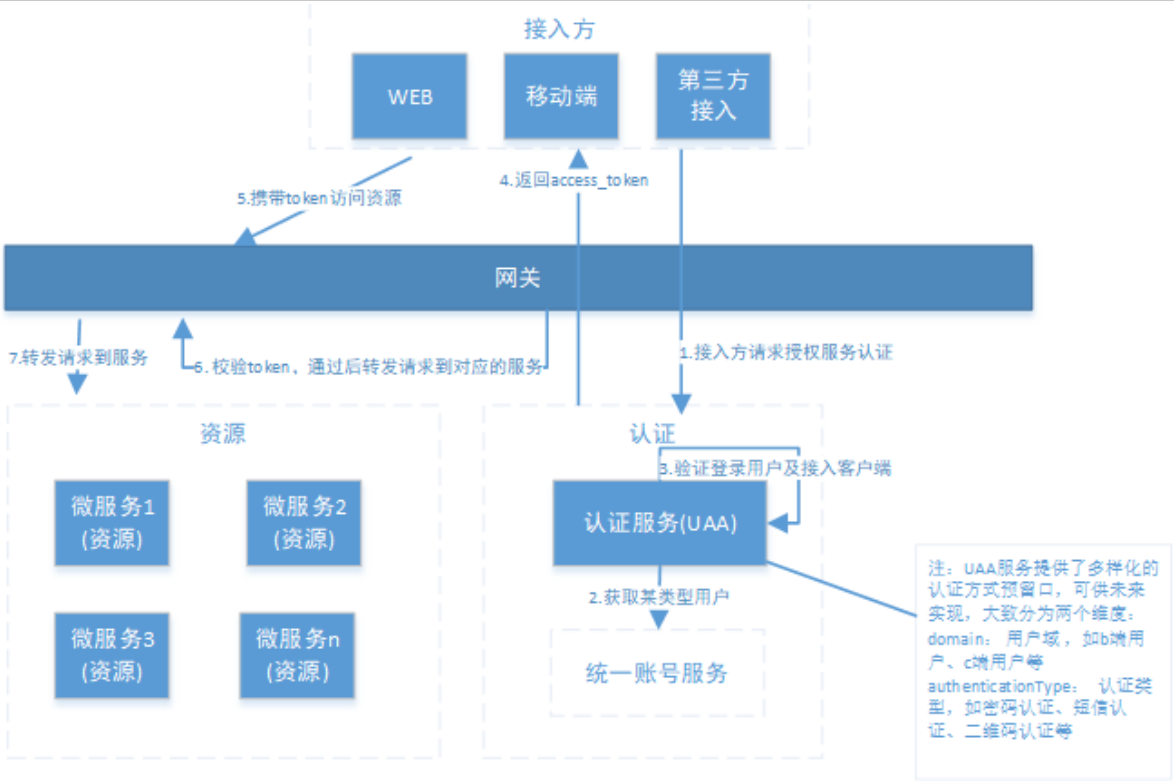
1.4 认证拦截功能实现

1.4.1 需求分析

前面我们已经接触过、使用过网关服务(gateway-server)，该服务目前只是做了路由转发，并没有起到真正网关的作用，现在终于有了用武之地，接下来网关服务要在P2P认证授权体系里负责两件事：

（1）令牌解析并转发当前登录用户信息给微服务

（2）作为OAuth2.0的资源服务器角色，实现接入方权限拦截



1.4.2 环境准备

此功能同样涉及到Spring Security OAuth2，不做过多介绍，作为基础功能直接提供给大家使用。

1. 给gateway-server工程增加maven依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

2. 从资料文件夹中拷贝以下文件到gateway-server工程的config包中：

cn.itcast.wanxinp2p.gateway.config.ClientDefaultAccessTokenConverter

描述：明文令牌与spring OAuth2Authentication的相互转换

cn.itcast.wanxinp2p.gateway.config.JWTConfig

描述：配置Spring Security OAuth2采用jwt令牌方式

cn.itcast.wanxinp2p.gateway.config.ResouceServerConfig

描述：Spring Security OAuth2 资源服务实现，用于网关对接入客户端的权限拦截，可以指定某个接入客户端只允许访问部分微服务

cn.itcast.wanxinp2p.gateway.config.RestAccessDeniedHandler

描述：修改网关错误响应与平台整体风格一致，无需关注。

cn.itcast.wanxinp2p.gateway.config.RestOAuth2AuthExceptionEntryPoint

描述：修改网关错误响应与平台整体风格一致，无需关注。

3. 相关配置：

（1）每增加一个微服务，为了使前端能够通过网关访问到该微服务，都需要在Apollo的gateway-server上新增路由配置：

（2）在UAA服务数据库(p2p_uaa)中的oauth_client_details表配置了接入客户端的信息，其中authorities字段决定了该客户端的接入权限，如下图所示，wanxin-p2p-web-h5这个接入客户端，有ROLE_CONSUMER，ROLE_API两个权限。目前，在P2P项目中只开发了wanxin-p2p-web-h5这一个客户端。

client_id	client_secret	authorized_grant_types	authorities
wanxin-p2p-web-admin	itcastb	client_credentials,password,authorization_code,implicit	ROLE_ADMIN,ROLE_CONSUMER,ROLE_A
wanxin-p2p-web-app	123456	client_credentials,password,authorization_code,implicit	ROLE_CONSUMER,ROLE_API
wanxin-p2p-web-h5	itcasth5	client_credentials,password,authorization_code,implicit	ROLE_CONSUMER,ROLE_API

（3）在gateway-server工程中的ResouceServerConfig类里定义资源服务配置，主要配置的内容就是定义一些匹配规则，描述某个接入客户端需要什么样的权限才能访问某个微服务。

```
/**
 * 用户中心 资源服务器定义
 */
@Configuration
@EnableResourceServer
public class ConsumerServerConfig extends
        ResourceServerConfigurerAdapter {
    @Autowired
    private TokenStore tokenStore;
    @Override
    public void configure(ResourceServerSecurityConfigurer resources)
            throws Exception {
        resources.tokenStore(tokenStore).resourceId(RESOURCE_ID)
                                                .stateless(true);
```

```
            resources.authenticationEntryPoint(point).accessDeniedHandler(handler);
    }
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                .and()
                .authorizeRequests()
                .antMatchers("/consumer/l/**").denyAll()
                .antMatchers("/consumer/my/**").access("#oauth2.hasScope('read')
and
#oauth2.clientHasRole('ROLE_CONSUMER')")
                .antMatchers("/consumer/m/**").access("#oauth2.hasScope('read')
and
#oauth2.clientHasRole('ROLE_ADMIN')")
                .antMatchers("/consumer/**").permitAll();
    }
}
```

如果网关接收到的请求url符合/consumer/**表达式，该请求将被转发至consumer-service(用户中心)。为了便于权限拦截，我们将微服务内部的接口进行了细分，大体分为三类：

1. 受保护的c端用户接口(C端用户登录后可访问)

   Url格式：/服务名称/my/资源名称/*

   访问方式：需要携带C端用户认证所获取的Access Token才可访问。

2. 受保护的b端用户接口(B端管理员用户登录后可访问)

   Url格式：/服务名称/m/资源名称/*

   访问方式：需要携带B端用户认证所获取的Access Token才可访问。

3. 公开资源

   Url格式：/服务名称/资源名称/*

   访问方式：无限制

```
.antMatchers("/consumer/my/**").access("#oauth2.hasScope('read') and

#oauth2.clientHasRole('ROLE_CONSUMER')")
```

以上匹配规则描述了：某个发往用户中心的请求，要想访问url匹配/my/**规则的资源，接入客户端需要有scope中包含read，并且authorities(权限)中需要包含ROLE_CONSUMER，这跟oauth_client_details表中的数据是相对应的。

1.4.3 功能实现

认证拦截功能主要由框架(Spring Security OAuth2)实现，我们无需处理。我们需要做的就是在过滤器中把令牌中的用户信息取出来并转发给微服务使用。

```
@Component
public class AuthFilter extends ZuulFilter {
    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
```

```
    public String filterType() {
        return "pre"; //前置过滤器，可以在请求被路由之前调用
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    @Override
    public Object run() {
        //1.获取Spring Security OAuth2的认证信息对象
        Authentication authentication = SecurityContextHolder.getContext()
                .getAuthentication();
        if(authentication==null || !(authentication instanceof
OAuth2Authentication)){
            return null;// 无token访问网关内资源，直接返回null
        }
        //2.将当前登录的用户以及接入客户端的信息放入Map中
        OAuth2Authentication oauth2Authentication=
(OAuth2Authentication)authentication;

        Map<String,String> jsonToken = new HashMap<>

(oauth2Authentication.getOAuth2Request().getRequestParameters());
        /*3.将jsonToken写入转发微服务的request中*/
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        request.getParameterMap();// 关键步骤，一定要get一下，下面这行代码才能取到值
        Map<String,List<String>> requestQueryParams =
ctx.getRequestQueryParams();
        if (requestQueryParams == null) {
            requestQueryParams = new HashMap<>();
        }
        List<String> arrayList = new ArrayList<>();

 arrayList.add(EncryptUtil.encodeUTF8StringBase64(JSON.toJSONString(jsonToken)))
;
        requestQueryParams.put("jsonToken", arrayList);
        ctx.setRequestQueryParams(requestQueryParams);
        return null;
    }
}
```

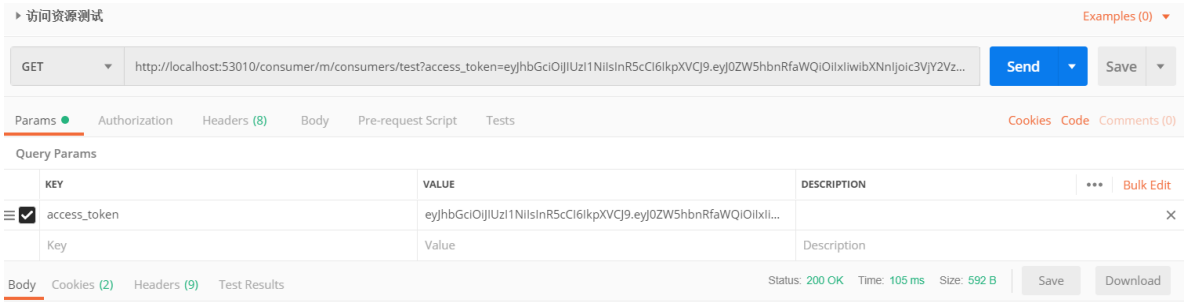
1.4.4 功能测试

1. 在wanxinp2p-consumer-service工程的ConsumerController类中新定义一个受保护的资源方法用来测试：

```
@ApiOperation("过网关受保护资源，进行认证拦截测试")
@ApiImplicitParam(name = "jsonToken",  value = "访问令牌",  required = true,
                                                              dataType =
"String")
@GetMapping(value = "/m/consumers/test")
public RestResponse<String> testResources(String jsonToken) {
    return RestResponse.success(EncryptUtil.decodeUTF8StringBase64(jsonToken));
}
```

2. 启动相关服务后，在Postman中发起请求：

```
GET http://localhost:53010/consumer/m/consumers/test?
access_token=eyJhbGciOiJIUzI1Ni...
```



如果令牌正确，则返回如下内容：

```
{
  "code": 0,
  "msg": "",
  "result": "{\"tenant_id\":\"1\", \"department_id\":\"1\", \"payload\":\"
{\\\"res\\\":\\\"res1111111\\\"}\", \"user_name\":\"admin\",
\"mobile\":\"18611106983\", \"user_authorities\":\"{\\\"ROLE1\\\":[\\\"p1\\\",
\\\"p2\\\"]}\", \"client_id\":\"wanxin-p2p-web-admin\"}"
}
```

如果令牌错误，则返回如下内容：

```
{
  "code": 401,
  "msg": "Cannot convert access token to JSON"
}
```

1.4.5 微服务获取请求中的数据

在认证拦截功能中，如果网关校验令牌成功，就会把当前登录用户的一些信息存放到请求中，然后转发给各个微服务，那微服务如何从请求中取出这些数据呢？怎么取会更方便呢？

考虑到 "从请求中取数据" 的功能可能会被多次使用，所以这里采用SpringMVC的拦截器实现该功能。网关转发到微服务的请求会被该拦截器接收，然后在该拦截器中取出数据并做相应处理。

```
/**
  Token拦截处理
*/
public class TokenInterceptor implements HandlerInterceptor {

    @Override
```

```
    public boolean preHandle(HttpServletRequest httpServletRequest,
                             HttpServletResponse httpServletResponse, Object o)
{
        String jsonToken = httpServletRequest.getParameter("jsonToken");
        if (StringUtils.isNotBlank(jsonToken)) {
            LoginUser loginUser = JSON
                    .parseObject(EncryptUtil.decodeUTF8StringBase64(jsonToken),
                                 new TypeReference<LoginUser>() {
                    });
            httpServletRequest.setAttribute("jsonToken", loginUser);
        }
        return true;
    }
}
```

我们自定义的拦截器要想起到作用，必须向SpringMVC进行注册，因此这里需要增加配置。

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new TokenInterceptor()).addPathPatterns("/**");
    }
}
```

2 注册功能中的分布式事务

在用户注册流程中一次注册请求需要由用户中心服务和统一账号服务协调共同完成，由于种种原因，当其中一个服务操作失败时会导致数据不一致，这样就产生了事务问题，本章节就要解决注册功能中的事务问题。

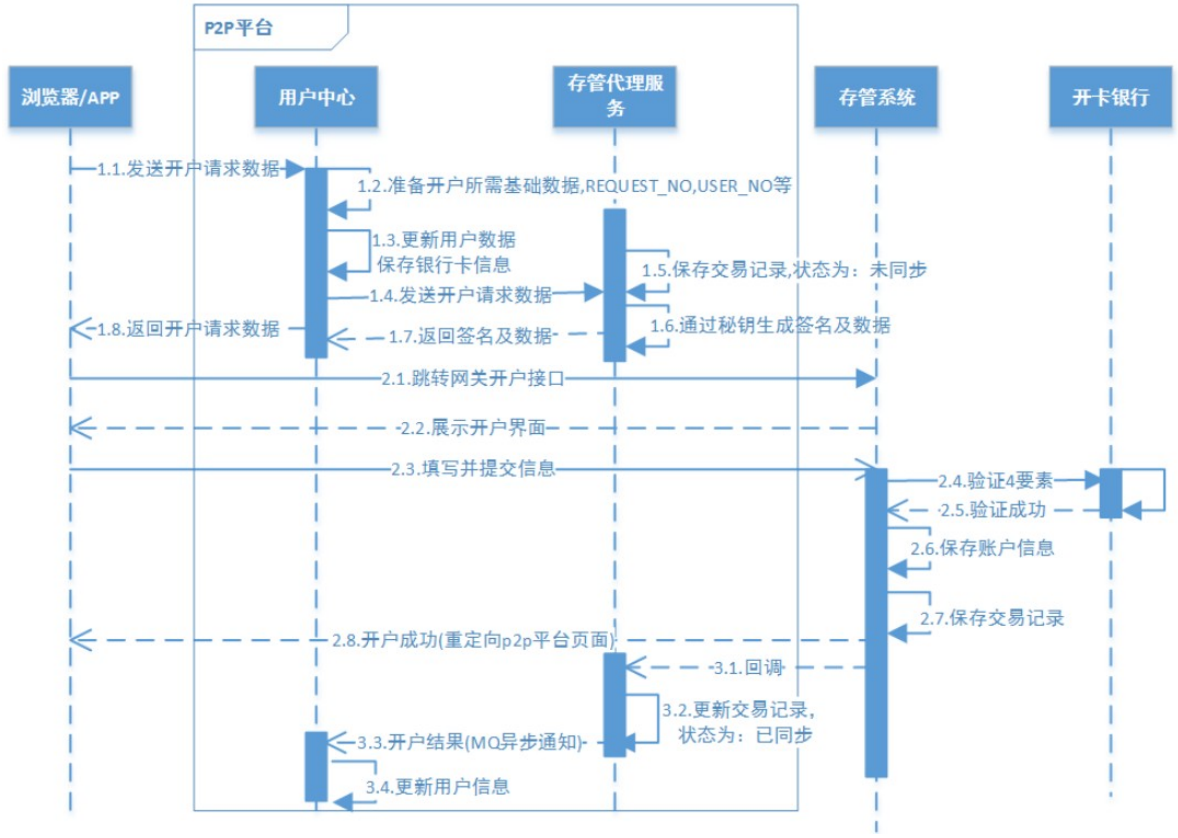
请参考资料文件夹中的“分布式事务”，用户注册功能使用Hmily框架解决分布式事务问题。

3 用户开户

3.1 需求分析

开户是指借款用户和投资用户在注册后、交易前都需要在银行存管系统开通个人存管账户，在开户流程中银行存管系统是一个很重要的系统，它是当前P2P平台最常见的一种模式，为了保证资金不流向P2P平台，由银行存管系统去管理借款用户和投资用户的资金，P2P平台与银行存管系统进行接口交互为借款用户和投资用户搭建交易的桥梁，它们之间的关系如下：

借款人和投资人需要开通两个账户，一个是P2P平台的，一个是银行存管系统的。两者的账户数据需要进行同步。具体业务流程如下图所示：



系统介绍：

- 存管系统：不属于P2P平台，属于银行系统，专门负责对接P2P账户及交易，此系统在银行部署，P2P平台的交易主要与存管系统交互。
- 存管代理服务：属于P2P平台，为了使P2P平台与银行存管系统松耦合，专门设立存管代理服务与存管系统对接，P2P平台的各个服务都通过存管代理服务与银行存管系统交互。
- 开卡银行：不属于P2P平台，属于银行系统，是银行用于管理储蓄卡信息的系统。为了使用方便，把该系统合并到了存管系统中。

  由于存管系统属于银行，我们无需开发，直接提供。

业务流程：

第一阶段：生成开户数据(图中1.1-1.8)

1、前端填写开户信息

前端会先查询开户信息，如果曾经填写了开户信息则在界面直接显示，用户可以修改; 如果曾经没有填写开户信息则用户在界面填写开户信息

2、前端请求用户中心服务开户

3、用户中心服务准备开户数据，并把开户信息保存到用户中心

4、用户中心服务请求存管代理服务生成交易记录（未同步），并对开户数据进行签名

5、存管代理服务将签名后的开户数据返回给用户中心

6、用户中心将开户数据返回给前端

第二阶段：请求开户(图中2.1-2.8)

7、前端携带开户信息请求银行存管系统

8、银行存管系统向前端返回开户信息确定页面

9、前端确认完成提交开户请求到银行存管系统

10、银行存管系统接收开户数据并进行校验，校验银行卡信息，信息无误则将开户信息保存至存管系统（校验过程中存管系统会请求开户银行校验银行卡信息）

第三阶段：开户结果通知(图中3.1-3.4)

11、开户成功后，银行存管系统异步通知存管代理服务

12、存管代理服务接收到开户成功通知后更新交易状态为同步

13、存管代理服务通知用户中心服务

14、用户中心服务接收到开户成功的消息保存开户信息

3.2 部署银行存管系统

由于银行存管系统不属于P2P平台，我们无需开发，所以直接提供了一个系统供大家使用，我们需要在本地部署该系统，并熟悉它的接口信息。

1. 参考：资料 \ 银行存管系统文件夹中的“银行存管系统部署指南.pdf”
2. 参考：资料 \ 银行存管系统文件夹中的“银行存管系统接口说明.pdf”

3.3 第一阶段: 生成开户数据

3.3.1 需求分析

1. 参考前面的“开户业务流程图”，熟悉该阶段的具体需求和业务流程
2. 查阅银行存管系统接口说明.pdf中的“个人绑卡开户”接口说明，熟悉接口接收参数和返回值

3.3.2 接口定义

3.3.2.1 用户中心生成开户数据接口

1. 在wanxinp2p-api工程中的ConsumerAPI接口中新增createConsumer方法：

```
/**
 * 生成开户请求数据
 * @param consumerRequest 开户信息
 * @return
 */
RestResponse<GatewayRequest> createConsumer(ConsumerRequest consumerRequest);
```

2. 在wanxinp2p-consumer-service工程的ConsumerController类中实现该方法：

```
@Override
@ApiOperation("生成开户请求数据")
@ApiImplicitParam(name = "consumerRequest", value = "开户信息", required = true,
dataType = "ConsumerRequest", paramType = "body")
@PostMapping("/my/consumers")
public RestResponse<GatewayRequest> createConsumer(@RequestBody ConsumerRequest
consumerRequest) {
    return null;
}
```

3.3.2.2 存管代理生成开户数据接口

1. 从资料文件夹中导入P2P存管代理微服务工程(wanxinp2p-depository-agent-service)，并搞定Apollo上的配置，在application名称空间中增加如下配置，其他名称空间请自行检查并修改。

```
depository.publicKey =
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJKcP4TjCb9+OKfOuvHkDO6njI8b9KKlu3ZdCkom4SONf8Kk
Z1jVl6A7XWnJ33gBLnbTGVUm5I+XvFEG5bSWVbkCAwEAAQ==
p2p.privateKey =
MIIBVAIBADANBgkqhkiG9w0BAQEFAASCAT4wggE6AgEAAkEApkqNoES+5O8OiULK5UIEuZ9WxIUG7fB9
2vOvEi1FyNJgzMc2gi5hy8eGcyYyLWJdEt5h1vC8jclCgEcMY3lp3QIDAQABAkAUhQia6UDBXEEH8QUG
azIYEbBsSZoETHPLGbOQQ6Pj1tb6CVC57kioBjwtNBnY2jBDWi5K815LnOBcJSSjJPwhAiEA2eO6VZMT
kdjQAkpB5dhy/OC3i8zsOcOM1rPoTA/RpkUCIQDDYHJPqHLkQyd//7sEeYcm8cMBTvDKBXyiuGk8eLRa
uQIgQo6IlalGmg+Dgp+SP5Z9kjD/oCmpOXBOUoVEGS/f14OCIQCsG9YXHgi31ACD3T9eHcBVKjvidyve
ix7UKSdrQdl+4QIgNCtRVLV+783e7PX5hRXD+knsWTQxDEMEsHi1KsAWtPk=
p2p.publicKey =
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAKZKjaBEvudPDolCyuVCBLmfVsSFBu3wfdldLxItRcjSYMzH
NoIuYcvHhnMmMi1iXRLeYdbwvI3JQoBHDGN5ad0CAwEAAQ==
p2p.code = wanxinp2p
depository.url = http://localhost:55010/wanxindepository
```

启动该工程进行环境测试（注意设置启动参数：-Denv=dev -Dapollo.cluster=DEFAULT -Dserver.port=53070）

2. 在wanxinp2p-api工程中新建depository包，在该包中新建一个DepositoryAgentApi的接口，然后定义createConsumer方法：

```
/**
 * 银行存管系统代理服务API
 */
public interface DepositoryAgentApi {
    /**
     * 开通存管账户
     * @param consumerRequest 开户信息
     * @return
     */
    RestResponse<GatewayRequest> createConsumer(ConsumerRequest
consumerRequest);
}
```

3. 在wanxinp2p-depository-agent-service工程中定义DepositoryAgentController类，并实现createConsumer方法：

```
/**
 * 存管代理服务
 */
@Api(value = "存管代理服务", tags = "depository-agent")
@RestController
public class DepositoryAgentController implements DepositoryAgentApi {
    @Override
    @ApiOperation("生成开户请求数据")
    @ApiImplicitParam(name = "consumerRequest", value = "开户信息", required =
true,
                      dataType = "ConsumerRequest", paramType = "body")
    @PostMapping("/l/consumers")
    public RestResponse<GatewayRequest> createConsumer(@RequestBody
ConsumerRequest

consumerRequest) {
        return null;
    }
}
```

3.3.3 功能实现

3.3.3.1 用户中心生成开户数据

1. 数据访问层

   单独抽取出来一个处理银行卡业务的接口，在mapper包中新建BankCardMapper接口：

```
/**
用户绑定银行卡信息 Mapper 接口
*/
public interface BankCardMapper extends BaseMapper<BankCard> {
}
```

在mapper包中新建BankCardMapper.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.itcast.wanxinp2p.consumer.mapper.BankCardMapper">
</mapper>
```

ConsumerMapper及映射配置文件之前已经存在，这里不用再管。

2. 在agent包中新建一个远程调用存管代理服务的Feign代理：

```
@FeignClient(value = "depository-agent-service")
public interface DepositoryAgentApiAgent {
   @PostMapping("/depository-agent/l/consumers")
   RestResponse<GatewayRequest> createConsumer(@RequestBody ConsumerRequest
                                                consumerRequest);
}
```

3. 新建一个BankCardService接口，定义如下两个方法：

```
/**
 * 用户绑定银行卡信息 服务类
 */
public interface BankCardService extends IService<BankCard> {

   /**
    * 获取银行卡信息
    * @param consumerId 用户id
    * @return
    */
   BankCardDTO getByConsumerId(Long consumerId);

   /**
    * 获取银行卡信息
    * @param cardNumber 卡号
    * @return
    */
   BankCardDTO getByCardNumber(String cardNumber);

}
```

4. 新建BankCardServiceImpl类实现上述两个方法：

```
@Service
public class BankCardServiceImpl extends ServiceImpl<BankCardMapper, BankCard>
implements BankCardService {

   @Override
   public BankCardDTO getByConsumerId(Long consumerId) {
      BankCard bankCard = getOne(new QueryWrapper<BankCard>().lambda()
                                 .eq(BankCard::getConsumerId, consumerId);
      return convertBankCardEntityToDTO(bankCard);
   }

   @Override
   public BankCardDTO getByCardNumber(String cardNumber) {
```

```
        BankCard bankCard = getOne(new QueryWrapper<BankCard>().lambda()
                                    .eq(BankCard::getCardNumber, cardNumber);
        return convertBankCardEntityToDTO(bankCard);
    }

    /**
     * entity转为dto
     * @param entity
     * @return
     */
    private BankCardDTO convertBankCardEntityToDTO(BankCard entity) {
        if (entity == null) {
            return null;
        }
        BankCardDTO dto = new BankCardDTO();
        BeanUtils.copyProperties(entity, dto);
        return dto;
    }
}
```

5. 在ConsumerService接口中新增createConsumer方法：

```
/**
生成开户数据
@param consumerRequest
@return
*/
RestResponse<GatewayRequest> createConsumer(ConsumerRequest consumerRequest);
```

6. 在ConsumerServiceImpl类中实现该方法：

```
@Override
@Transactional
public RestResponse<GatewayRequest> createConsumer(ConsumerRequest
consumerRequest) {
    ConsumerDTO consumerDTO=getByMobile(consumerRequest.getMobile());

    //判断用户是否已开户
    if (consumerDTO.getIsBindCard() == 1) {
        throw new BusinessException(ConsumerErrorCode.E_140105);
    }

    //判断银行卡是否已被绑定
    BankCardDTO bankCardDTO = bankCardService

.getByCardNumber(consumerRequest.getCardNumber());
    if (bankCardDTO != null &&
bankCardDTO.getStatus()==StatusCode.STATUS_IN.getCode()) {
        throw new BusinessException(ConsumerErrorCode.E_140151);
    }

    //更新用户开户信息
    consumerRequest.setId(consumerDTO.getId());
    //产生请求流水号和用户编号
```

```
 consumerRequest.setUserNo(CodeNoUtil.getNo(CodePrefixCode.CODE_CONSUMER_PREFIX)
);

 consumerRequest.setRequestNo(CodeNoUtil.getNo(CodePrefixCode.CODE_REQUEST_PREFI
X));

    //设置查询条件和需要更新的数据
    UpdateWrapper<Consumer> updateWrapper = new UpdateWrapper<>();
    updateWrapper.lambda().eq(Consumer::getMobile, consumerDTO.getMobile());
    updateWrapper.lambda().set(Consumer::getUserNo,
consumerRequest.getUserNo());
    updateWrapper.lambda().set(Consumer::getRequestNo,
consumerRequest.getRequestNo());
    updateWrapper.lambda().set(Consumer::getFullname,
consumerRequest.getFullname());
    updateWrapper.lambda().set(Consumer::getIdNumber,
consumerRequest.getIdNumber());
    updateWrapper.lambda().set(Consumer::getAuthList, "ALL");
    update(updateWrapper);

    //保存用户绑卡信息
    BankCard bankCard = new BankCard();
    bankCard.setConsumerId(consumerDTO.getId());
    bankCard.setBankCode(consumerRequest.getBankCode());
    bankCard.setCardNumber(consumerRequest.getCardNumber());
    bankCard.setMobile(consumerRequest.getMobile());
    bankCard.setStatus(StatusCode.STATUS_OUT.getCode());

    BankCardDTO existBankCard = bankCardService
                                    .getByConsumerId(bankCard.getConsumerId());
    if (existBankCard != null) {
        bankCard.setId(existBankCard.getId());
    }
    bankCardService.saveOrUpdate(bankCard);
    return depositoryAgentApiAgent.createConsumer(consumerRequest);
}
```

7. 完善ConsumerController的代码

```
@PostMapping("/my/consumers")
public RestResponse<GatewayRequest> createConsumer(@RequestBody ConsumerRequest
consumerRequest) {
   consumerRequest.setMobile(SecurityUtil.getUser().getMobile());
   return consumerService.createConsumer(consumerRequest);
}
```

3.3.3.2 存管代理生成开户数据

1. 数据访问层

在mapper包中新建DepositoryRecordMapper接口：

```
/**
 * 存管交易记录表 Mapper 接口
 */
public interface DepositoryRecordMapper extends BaseMapper<DepositoryRecord> {

}
```

在mapper包中新建DepositoryRecordMapper.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper
namespace="cn.itcast.wanxinp2p.depository.mapper.DepositoryRecordMapper">

</mapper>
```

2. 新建DepositoryRecordService接口, 并定义createConsumer方法：

```
public interface DepositoryRecordService extends IService<DepositoryRecord> {
    /**
     * 开通存管账户
     * @param consumerRequest 开户信息
     * @return
     */
    GatewayRequest createConsumer(ConsumerRequest consumerRequest);
}
```

3. 新建DepositoryRecordServiceImpl类并实现createConsumer方法：

```
@Service
public class DepositoryRecordServiceImpl extends
ServiceImpl<DepositoryRecordMapper, DepositoryRecord> implements
DepositoryRecordService{

    @Autowired
    private ConfigService configService;

    @Override
    public GatewayRequest createConsumer(ConsumerRequest consumerRequest) {

        //1.保存交易记录
        saveDepositoryRecord(consumerRequest);

        //2.签名数据并返回
        String reqData=JSON.toJSONString(consumerRequest);
        String sign=RSAUtil.sign(reqData,configService.getP2pPrivateKey(),"utf-
8");
        GatewayRequest gatewayRequest=new GatewayRequest();
        gatewayRequest.setServiceName("PERSONAL_REGISTER");
        gatewayRequest.setPlatformNo(configService.getP2pCode());
        gatewayRequest.setReqData(EncryptUtil.encodeURL(EncryptUtil
                .encodeUTF8StringBase64(reqData)));
        gatewayRequest.setSignature(EncryptUtil.encodeURL(sign));
```

```
        gatewayRequest.setDepositoryUrl(configService.getDepositoryUrl() +
"/gateway");
        return gatewayRequest;

    }

    private void saveDepositoryRecord(ConsumerRequest consumerRequest){
        DepositoryRecord depositoryRecord=new DepositoryRecord();
        depositoryRecord.setRequestNo(consumerRequest.getRequestNo());
        depositoryRecord.setRequestType(DepositoryRequestTypeCode
                                        .CONSUMER_CREATE.getCode());
        depositoryRecord.setObjectType("Consumer");
        depositoryRecord.setObjectId(consumerRequest.getId());
        depositoryRecord.setCreateDate(LocalDateTime.now());
        depositoryRecord.setRequestStatus(StatusCode.STATUS_OUT.getCode());
        save(depositoryRecord);
    }
}
```

4. 完善DepositoryAgentController代码：

```
@Override
@ApiOperation("生成开户请求数据")
@ApiImplicitParam(name = "consumerRequest", value = "开户信息", required = true,
dataType = "ConsumerRequest", paramType = "body")
@PostMapping("/l/consumers")
public RestResponse<GatewayRequest> createConsumer(@RequestBody ConsumerRequest
consumerRequest) {
    return
RestResponse.success(depositoryRecordService.createConsumer(consumerRequest));
}
```

5. 功能测试

3.4 第二阶段：请求开户

参考前面的“开户业务流程图”，该阶段涉及到前端和银行存管服务，这些都不需要我们开发，直接进行测试即可

注意：

1. 为了测试方便，可以考虑关闭短信验证码功能
2. 一定要从数据库中挑选C端用户进行登录
3. 身份证、银行卡等信息要从数据库中挑选

3.5 第三阶段：开户结果通知

3.5.1 问题分析

参考前面的“开户业务流程图”以及第二阶段的演示效果，我们发现其实在第二阶段的时候开户业务就完成了。但是我们还要考虑到P2P平台与银行存管系统之间属于跨系统交互，银行存管系统的业务处理时间和结果，用户中心无法干预，只能被动等待银行存管系统处理完毕，那如果银行存管系统忙，不能很快响应，用户将不能很快看到开户成功的页面，或者响应时网络出现问题，用户就无法看到开户成功的页面，又该怎么办呢?

3.5.2 解决方案

银行存管系统处理完该开户请求后，除了会重定向到P2P中开户成功的页面之外，还将回调P2P存管代理服务通知处理结果，若通知失败，则按一定策略进行重试。同时，银行存管系统会提供**开户结果查询**的接口，供P2P用户中心主动查询开户结果。

P2P存管代理服务接收到开户通知后，还要告知给P2P用户中心服务，但是由于P2P用户中心服务是被动等待方，所以这里采用RocketMQ进行通知。也就是说，P2P存管代理服务接收到开户通知后会发消息给RocketMQ，而P2P用户中心服务会从RocketMQ中获取消息。

3.5.3 功能实现

3.5.3.1 RocketMQ环境搭建

1. 自行安装和配置RocketMQ，然后去安装目录的bin文件中执行下面命令启动RocketMQ

```
start mqnamesrv.cmd

start mqbroker.cmd -n 127.0.0.1:9876 autoCreateTopicEnable=true

java -jar rocketmq-console-ng-1.0.1.jar --server.port=60000 --
rocketmq.config.namesrvAddr=127.0.0.1:9876
```

2. 检查用户中心和存管代理服务的pom依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
</dependency>
```

3. 检查用户中心和存管代理服务在Apollo上的mq配置，只需要为consumer-service增加如下配置即可

```
rocketmq.consumer.group = CID_P2P_GATEWAY_NOTIFY
```

4. 检查用户中心和存管代理服务application.yml中是否引入mq的名称空间

3.5.3.2 存管代理服务

1. 新建一个DepositoryNotifyController，用来接收银行存管系统的开户结果回调通知

```
@Api(value = "银行存管系统通知服务", tags = "depository-agent")
@RestController
public class DepositoryNotifyController {

   @ApiOperation("接受银行存管系统开户回调结果")
   @ApiImplicitParams({
   @ApiImplicitParam(name = "serviceName", value = "请求的存管接口名",
```

```
                            required = true, dataType = "String", paramType =
"query"),
   @ApiImplicitParam(name = "platformNo", value = "平台编号，平台与存管系统签约时获
取",
                            required = true, dataType = "String", paramType =
"query"),
   @ApiImplicitParam(name = "signature", value = "对reqData参数的签名",
                            required = true, dataType = "String", paramType =
"query"),
   @ApiImplicitParam(name = "reqData", value = "业务数据报文，json格式",
                            required = true, dataType = "String", paramType =
"query"),})
   @RequestMapping(value = "/gateway", method = RequestMethod.GET,
                            params = "serviceName=PERSONAL_REGISTER")
   public String receiveDepositoryCreateConsumerResult(
         @RequestParam("serviceName") String serviceName,
         @RequestParam("platformNo") String platformNo,
         @RequestParam("signature") String signature,
         @RequestParam("reqData") String reqData) {
      //1.更新数据

      //2.给用户中心发送消息

      //3.给银行存管系统返回结果
      return "OK";
   }
}
```

2. 在DepositoryRecordService接口中新增modifyRequestStatus方法：

```
/**
 * 根据请求流水号更新请求状态
 * @param requestNo
 * @param requestsStatus
 * @return
*/
Boolean modifyRequestStatus(String requestNo, Integer requestsStatus);
```

在DepositoryRecordServiceImpl类中实现modifyRequestStatus方法：

```
@Override
public Boolean modifyRequestStatus(String requestNo, Integer requestsStatus) {
   return update(Wrappers.<DepositoryRecord>lambdaUpdate()
         .eq(DepositoryRecord::getRequestNo, requestNo)
         .set(DepositoryRecord::getRequestStatus, requestsStatus)
         .set(DepositoryRecord::getConfirmDate, LocalDateTime.now()));
}
```

3. 新建message包，在该包中定义消息生产者：

```
/**
 * 存管代理服务异步通知消息生产者
 */
@Component
public class GatewayMessageProducer {
```

```
    @Resource
    private RocketMQTemplate rocketMQTemplate;

    public void personalRegister(DepositoryConsumerResponse response) {

 rocketMQTemplate.convertAndSend("TP_GATEWAY_NOTIFY_AGENT:PERSONAL_REGISTER",
                                         response);
    }
}
```

4. 完善DepositoryNotifyController代码

```
public String receiveDepositoryCreateConsumerResult(
          @RequestParam("serviceName") String serviceName,
          @RequestParam("platformNo") String platformNo,
          @RequestParam("signature") String signature,
          @RequestParam("reqData") String reqData) {
       //1.更新数据
       DepositoryConsumerResponse response = JSON
             .parseObject(EncryptUtil.decodeUTF8StringBase64(reqData),
                          DepositoryConsumerResponse.class);
       depositoryRecordService.modifyRequestStatus(response.getRequestNo(),
                                                   response.getStatus());
       //2.给用户中心发送消息
       gatewayMessageProducer.personalRegister(response);
       //3.给银行存管系统返回结果
       return "OK";
}
```

3.5.3.3 用户中心服务

1. 在ConsumerService接口中新增modifyResult方法：

```
/**
 * 更新开户结果
 * @param response
 * @return
 */
Boolean modifyResult(DepositoryConsumerResponse response);
```

在ConsumerServiceImpl类中实现modifyResult方法：

```
@Override
@Transactional
public Boolean modifyResult(DepositoryConsumerResponse response) {
    //1.获取状态
    int status = DepositoryReturnCode.RETURN_CODE_00000.getCode()
            .equals(response.getRespCode()) ? StatusCode.STATUS_IN.getCode()
            :StatusCode.STATUS_FAIL.getCode();
    //2.更新开户结果
    Consumer consumer = getByRequestNo(response.getRequestNo());
    update(Wrappers.<Consumer>lambdaUpdate().eq(Consumer::getId,
consumer.getId())
```

```
            .set(Consumer::getIsBindCard, status).set(Consumer::getStatus,
status));
    //3.更新银行卡信息
    return bankCardService.update(Wrappers.<BankCard>lambdaUpdate()
            .eq(BankCard::getConsumerId, consumer.getId())
            .set(BankCard::getStatus, status).set(BankCard::getBankCode,
                                                  response.getBankCode())
            .set(BankCard::getBankName, response.getBankName()));
}

private Consumer getByRequestNo(String requestNo){
    return getOne(Wrappers.
<Consumer>lambdaQuery().eq(Consumer::getRequestNo,requestNo));
}
```

2. 在message包中新建GatewayNotifyConsumer类，用来接收存管代理服务的开户结果通知

```
@Component
public class GatewayNotifyConsumer {

   @Value("${rocketmq.consumer.group}")
   private String consumerGroup;

   @Value("${rocketmq.name-server}")
   private String mqNameServer;

   @Autowired
   private ConsumerService consumerService;

   public GatewayNotifyConsumer() throws MQClientException {
      DefaultMQPushConsumer defaultMQPushConsumer=new DefaultMQPushConsumer

(consumerGroup);
      defaultMQPushConsumer.setNamesrvAddr(nameServer);
      defaultMQPushConsumer.setConsumeFromWhere(ConsumeFromWhere
                                                .CONSUME_FROM_LAST_OFFSET);
      defaultMQPushConsumer.subscribe("TP_GATEWAY_NOTIFY_AGENT","*");

      //注册监听器
      defaultMQPushConsumer.registerMessageListener(new
MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs,
                                                    ConsumeConcurrentlyContext
context) {
                try {
                    Message message = msgs.get(0);
                    String topic = message.getTopic();
                    String tag = message.getTags();
                    String body = new String(message.getBody(),
StandardCharsets.UTF_8);
                    if(tag.equals("PERSONAL_REGISTER")){
                        DepositoryConsumerResponse response =
JSON.parseObject(body,

 DepositoryConsumerResponse.class);
```

```
                        consumerService.modifyResult(response);
                    }
                    //if...

                }catch (Exception e){
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });

        defaultMQPushConsumer.start();
    }
}
```

3.5.4 功能测试

1. 在测试前，请先去银行存管系统中，找到UserServiceImpl类，把图中红框里的代码取消注释。这两处代码用来在银行存管系统处理完业务后发送消息。之前为了方便测试出开户第二阶段的效果，把这两处代码注释掉了。

2. 启动如下服务：

- 需要启动RocketMQ和rocketmq-console-ng-1.0.1.jar
- 需要启动Apollo
- 需要启动银行存管系统
- 需要启动P2P所有微服务工程
- 需要启动P2P前端工程

3. 为了方便测试，这里提供sql脚本可以新添加几个新用户进行登录和开户测试(密码即手机号)：

```
USE `p2p_consumer`;
```

```
INSERT  INTO `consumer`(`ID`,`USERNAME`,`MOBILE`,`IS_BIND_CARD`) VALUES
(1001,'15378791981','15378791981',0),
(1002,'15378791982','15378791982',0),
(1003,'15378791983','15378791983',0),
(1004,'15378791984','15378791984',0),
(1005,'15378791985','15378791985',0);

USE `p2p_account`;

INSERT  INTO `account`(`ID`,`USERNAME`,`MOBILE`,`PASSWORD`,`DOMAIN`) VALUES
(2001,'15378791981','15378791981','84ca7ae9e232f8ad4de6604c61668b465b15e8b77e25414e','c'),
(2002,'15378791982','15378791982','a7926810be7659b025958b43469126895c6299757f39cd32','c'),
(2003,'15378791983','15378791983','349966a4092c00000f57a483267280d3c228b5eb8a90b80b','c'),
(2004,'15378791984','15378791984','e9552ff2707ad9204121408ae84b8617f62948752403af22','c'),
(2005,'15378791985','15378791985','e9c390d3ac0116940034b97974b08bd35051b03f3c50887f','c');
```

4. 由于我们要在GatewayNotifyConsumer类的构造方法中使用@Value注入值，所以代码需要做如下修改：

```
@Component
public class GatewayNotifyConsumer {

    @Autowired
    private ConsumerService consumerService;

    public GatewayNotifyConsumer( @Value("${rocketmq.consumer.group}")
                                        String consumerGroup,
                                  @Value("${rocketmq.name-server}")
                                        String nameServer) throws
MQClientException {
        ... ...
    }
}
```