

2022 年 8 月 5 日 星期五

飞收项目遇到的问题

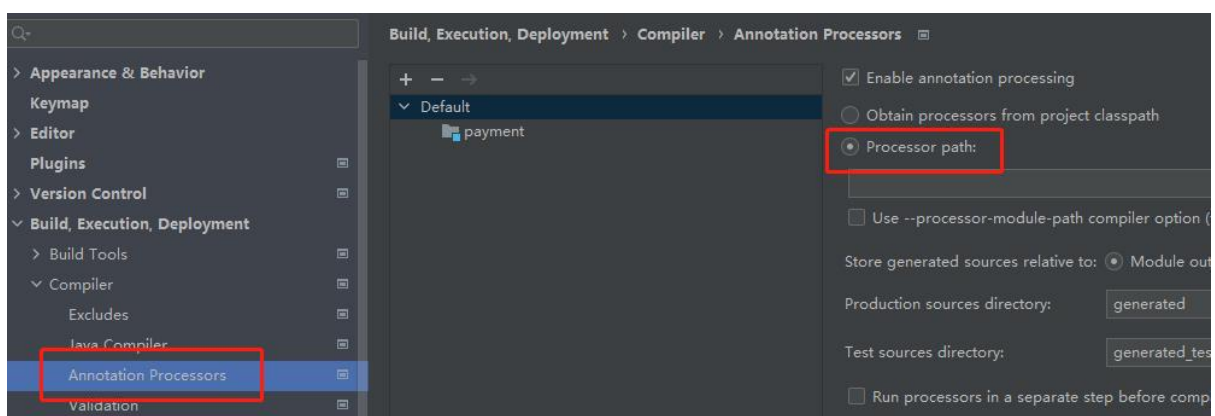
1. jar 包依赖导入问题

- IDEA 中 使用 project structure 中 Modeles 或者 libraries 中导入相关 jar 依赖, 且需要导入 tomcat 中 jsp-api.jar 和 servlet-api.jar, 可能还需要导入 websocket-api.jar

2. applicationContext.xml 报未找到

- 右键 mark as 资源文件

```
C:\Users\allinpayqd\TLWorkplace\development\un
java: 找不到符号
  符号:   类 BaseData_
  位置: 程序包 com.csy.libraries.common.model
```



3. 出现找不到类名后接下划线

在设置(不是项目设置)里面搜索 annotation 把 processpath 那玩意勾上

3. 前端后端跑不通

- 可能是后端配置的服务器地址有错, (在 xxx.properties 中修改), 需要将服务器地址改成公网 ip(局域网 ip), 否者前端找不到

4. 某项服务(文件上传)报 404, 路径找不到

- 原因为 IDEA 中发布路径为 xxx_exploaded, 和前端写死的项目名对不上, 将发布路径改为项目名则解决

5. 测试代理配置

前端项目中配置 config.js

```
// devUri = "https://fstest.allinpaygd.com/proxy/xjq"
```

后端项目中配置前端的回调地址(init.properties)

```
#server.currentViewRoot=https://fstest.allinpaygd.com/proxy/xjq/views/  
#server.currentSerUrl=https://fstest.allinpaygd.com/proxy/xjq/payment/
```

访问的主页:

<https://fstest.allinpaygd.com/proxy/xjq/views/#/home/index>

具体原理暂未理解怎么才能访问远端但是本地应用能够 debug, 有点神奇

6. 数据库连接地址

#【测试环境】本地

```
jdbc.driverClassName=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://10.47.0.117:3306/ursa_fp?useUnicode=true&characterEncoding=UTF-8&useSSL=false&serverTimezone=GMT%2B8  
jdbc.username=ursa_fp  
jdbc.password=ursa_fp
```

7. 云服务器连接信息

地址 10.47.0.102

账号 root

密码 Allinpaygd@102

项目发布路径 /app/tomcat8/webapps/payment/

飞收正式环境管理后台地址

<https://fp.aipgd.com/payment/system/index.html#/>

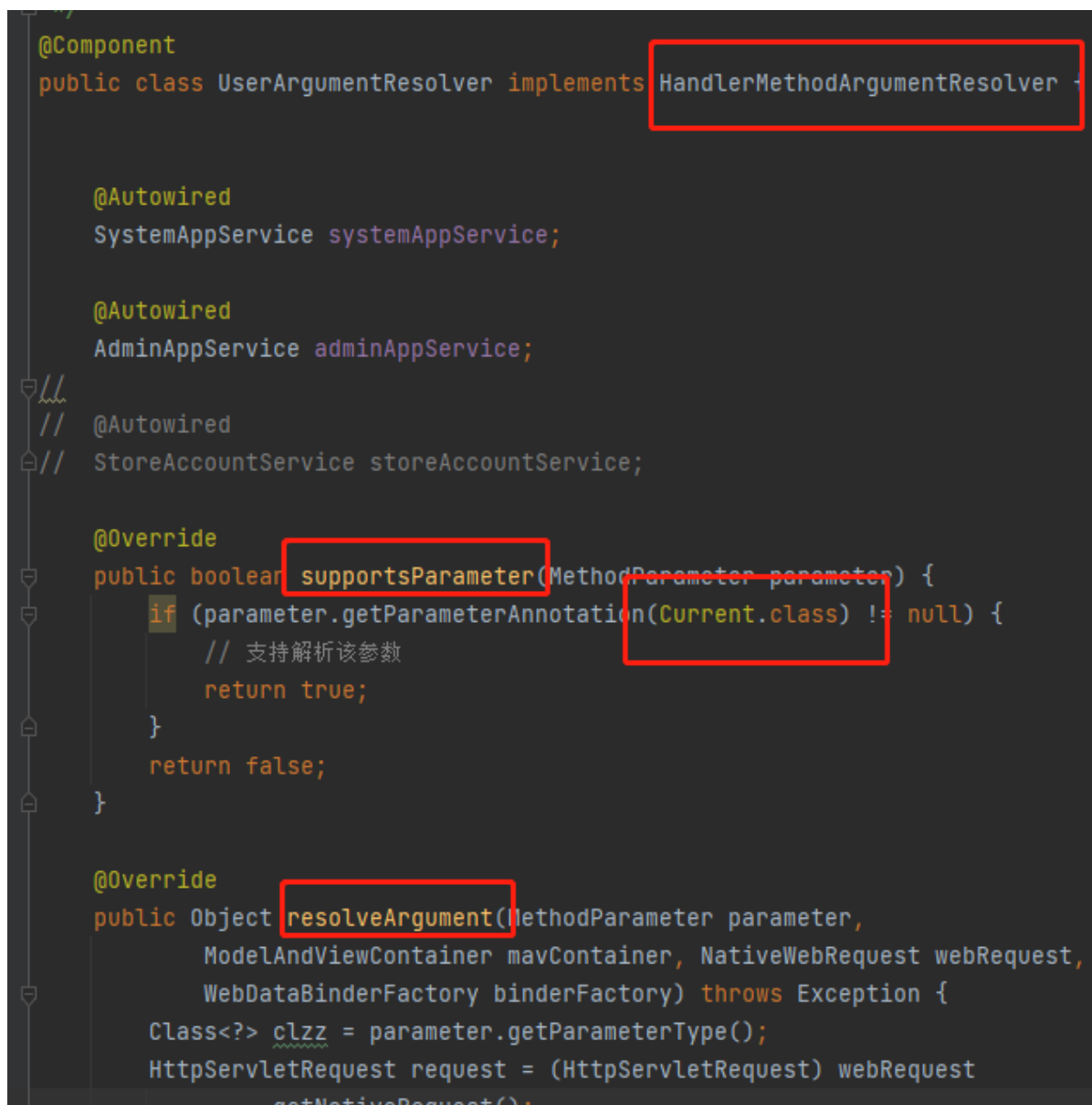
账号及密码: 15622285262

事务失效问题

由于历史遗留问题, Action 控制层代码中大量调用了不同 service 提供的 add/save/update 方法, 导致事务失效,

解决: 历史遗留问题, 暂不打算解决

@current 自定义注解是如何工作的



```
@Component
public class UserArgumentResolver implements HandlerMethodArgumentResolver {

    @Autowired
    SystemAppService systemAppService;

    @Autowired
    AdminAppService adminAppService;

    // @Autowired
    // StoreAccountService storeAccountService;

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        if (parameter.getParameterAnnotation(Current.class) != null) {
            // 支持解析该参数
            return true;
        }
        return false;
    }

    @Override
    public Object resolveArgument(MethodParameter parameter,
        ModelAndViewContainer mavContainer, NativeWebRequest webRequest,
        WebDataBinderFactory binderFactory) throws Exception {
        Class<?> clzz = parameter.getParameterType();
        HttpServletRequest request = (HttpServletRequest) webRequest
            getNativeRequest();
```

- 通过搜索 current.class 可看到

可以看到, 解析该注解的地方是一个实现了 HandlerMethodArgumentResolver 的一个自定义类, 这是一个 SpringMVC 提供的拓展解析器之一, 主要用于对 Controller 中方法参数进行处理

supportParameter() 方法用于判断是否需要处理该参数分解, 为 true 则需要, 并会调用下面的方法 resolveArgument()

```
/**
 * 获取当前会话的商家用户
 * @param request
 * @return
 * @throws Exception
 */
public StoreUserInfo getCurrentStoreUser(HttpServletRequest request) throws Exception {
    String token = request.getHeader(BaseAction.TOKEN_KEY_WEB);
    Map<String, Object> userData = ServerToken.parserToken(token);
    if (userData == null) {
        SourceLibrary.reLogin(DataConstant.USER_STORE);
    }
    String uid = (String) userData.get("uid");
    String sid = (String) userData.get("sid");
    String key = (String) userData.get("key");
    String eid = (String) userData.get("eid");
    String cid = (String) userData.get("cid"); // 预留
    if (!JUtil.isEmpty(uid) && JUtil.isEmpty(key) && DataConstant.USER_STORE) {
        SourceLibrary.reLogin(DataConstant.USER_STORE);
    }
    StoreUserInfo u = new StoreUserInfo();
    u.setId(uid);
    u.setStoreId(sid);
    u.setSysId(eid);
    return u;
}
```

resolveArgument() 为处理参数分解的方法, 返回的 Object 对象为 controller 方法上的形参对象

可以看出: 在 resolveArgument() 方法中调用了以下方法

其做了从 token 中提取数据构造 storeUserInfo 对象进行返回, 就起到了注入 Controller 请求参数的作用

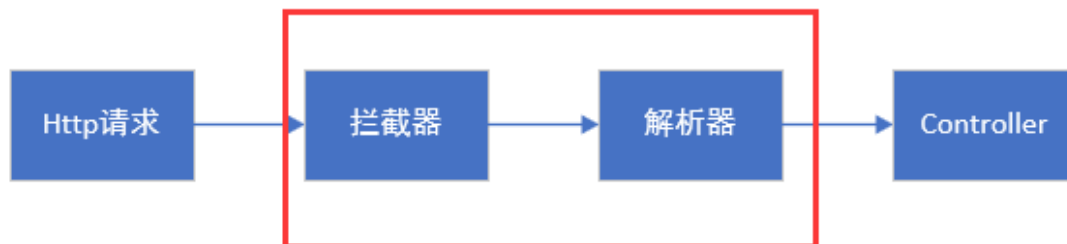
值得注意的是, 自定义 HandlerArgumentResolver 解析器需要在 xml 文件中进行注册

```

<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <bean class="com.csy.core.basis.advice.UserArgumentResolver"></bean>
  </mvc:argument-resolvers>
</mvc:annotation-driven>

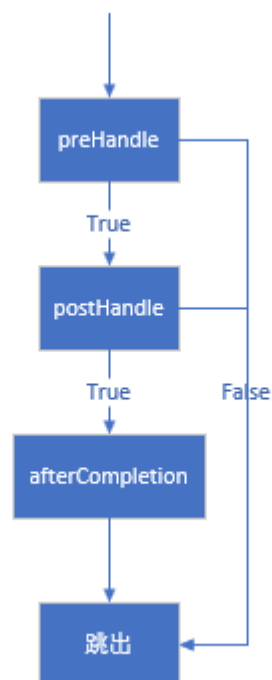
```

@current 拓展: SpringMVC 拦截器和解析器



- 拦截器: HandlerInterceptor 接口
- 解析器: HandlerMethodArgumentResolver 接口

基本流程图:



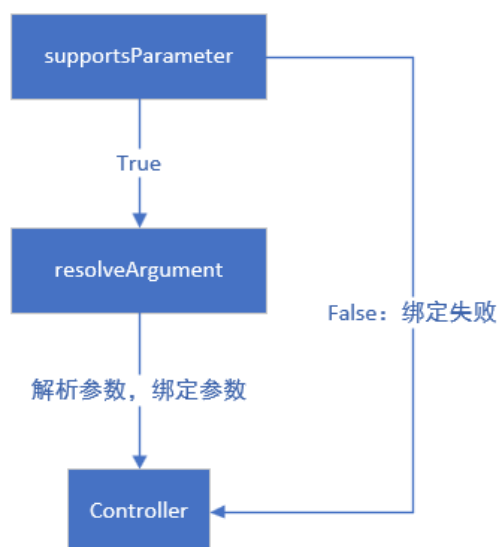
HandlerInterceptor

为方法拦截器, 对 http 请求进行拦截, 做权限校验处理

主要有三种方法:

- preHandle: 在业务处理器处理请求之前调用, 预处理, 可编码/安全控制/权限校验
- postHandle: 在业务处理执行完, 生成视图之前执行
- afterCompletion: 在 DispatcherServlet 完全处理完成后被调用, 可用于清理资源, 返回处理

用法:(略) 详看: https://blog.csdn.net/qq_32703777/article/details/103310146



HandlerMethodArgumentResolver 介绍

- SpringMVC 解析器用于解析 request 请求参数并绑定到 Controller 的入参上

```
1 | @Target(value = ElementType.PARAMETER)
2 | @Retention(value = RetentionPolicy.RUNTIME)
3 | @Documented
4 | public @interface EditRequired {
5 |     String editFlag() default "";
6 | }
```



```

1 public class EditRequiredMethodArgumentResolver implements HandlerMethodArgumentResolver {
2     @Autowired
3     private HelpSystemService helpSystemService;
4     public EditRequiredMethodArgumentResolver(){
5     }
6     @Override
7     public boolean supportsParameter(MethodParameter parameter){
8         return parameter.hasParameterAnnotation(EditRequired.class);
9     }
10    @Override
11    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest w
12        if (parameter.getParameterType().equals(String.class)) {
13            HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);
14            String operator = request.getHeader("operator");
15            if (helpSystemService.hasAuthority(operator)){
16                return "1";
17            }else {
18                return "0";
19            }
20        }
21        return "0";
22    }
23 }

```

- 自定义一个参数解析器需要实现 HandlerMethodArgumentResolver 接口, 重写

```

<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <bean class="com.resolver.EditRequiredMethodArgumentResolver"/>
    </mvc:argument-resolvers>
</mvc:annotation-driven>

```

```

1 @RequestMapping("/g")
2 @ResponseBody
3 public ResultModel getExactMsg(@EditRequired String editFlag) throws Exception{
4     System.out.println(editFlag);
5     return ResultModel.success("获取成功");
6 }

```

supportParameter 和 resolveArgument 方法, 配置加入 resolver 配置

- 如果需要配置多个解析器同时生效需要在一个解析器中对其他解析器兼容

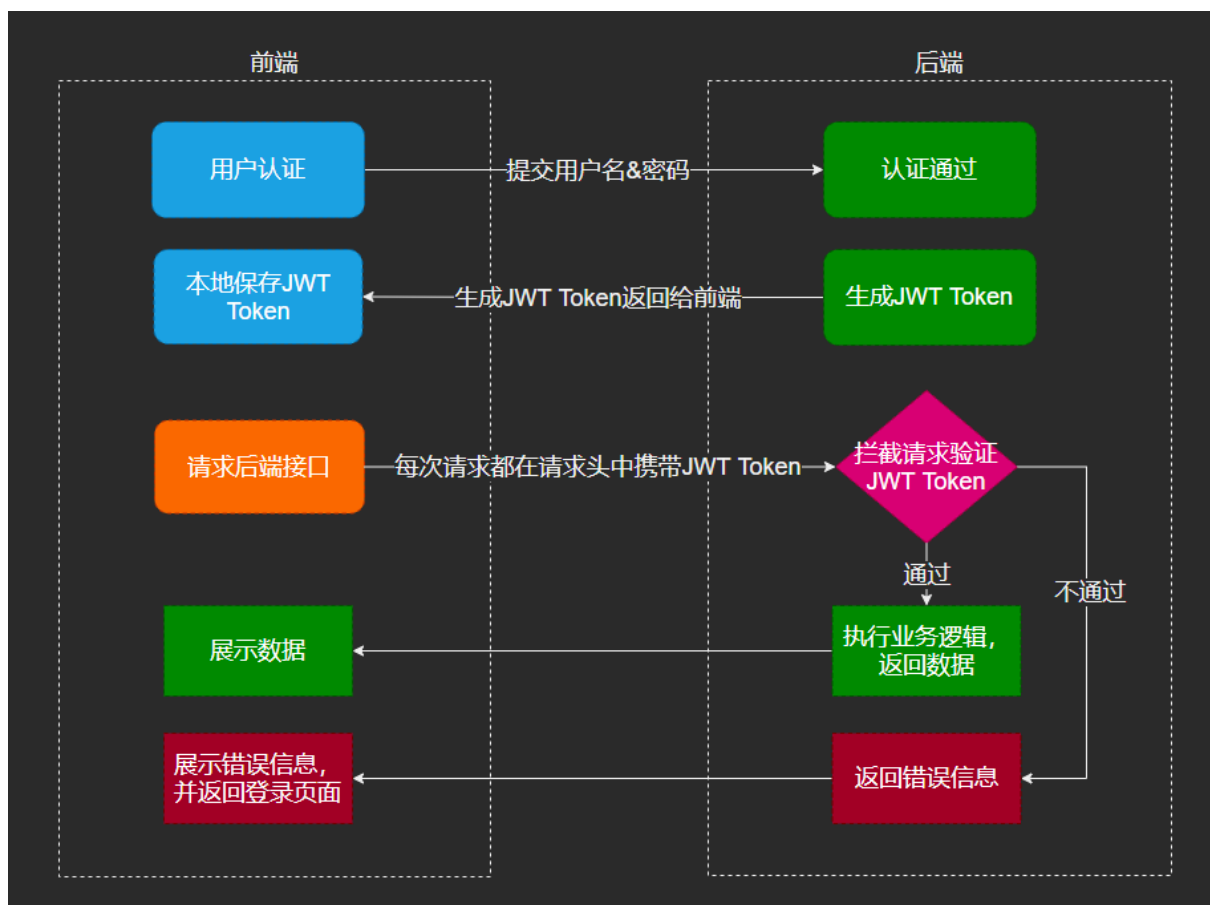
实现:

- 自定义注解
- 自定义解析器实现
- springMVC 配置文件

- Controller 中使用

拦截器和解析器对比

HandlerInterceptor 拦截器虽然可以对 request 进行一些处理, 但是它不具备往 controller 中绑定数据的功能, 原因是它只有 setAttribute 等, 而 HandlerMethodArgumentResolver 不仅仅可以解析 request 中的数据, 还可以绑定数据到 controller 中去



Token 理解

- Token 是无状态的: 无状态是指服务器无须记住用户请求的状态,服务器不保存 Token 信息, 这样就方便了分布式架构添加服务
- Token 是有时效的, Token 过期后用户需要重新认证, Token 的时效写在 Token 信息的内部
- Token 可以解决跨资源共享问题(CROS), 用户只要有一个通过验证的 token 数据和资源就能在任何域被请求到

员工收款统计

1	庄育龙	¥0.2	>
2	测试停用	¥0.03	>
3	庄玉龙	¥0.02	>
4	庄玉龙kkk	¥0.01	>

- Token 主要包括三部分,token head ,token payload, signature, 分别存储签名算法, 负载信

```
@click="to('..order/list?serUserId='+item.type+'&beginDate='+beginDate+'&endDate='+endDate+'&tradeType=1&statusValue=2' + '&employeeName=' + item.name)">
```

息, 签名

token 验证的流程

1. 前端传输用户名和密码到后端
2. 核对成功后, 将包含用户信息的数据作为 JWT token 的 payload 保存起来,返回给前端

3. 前端保存起来 token, 退出登录则可以删除 token, 每次请求将 JWT token 放入请求 head 里面
4. 后端验证 token, 验证有效性(签名正确/时效过期等)
5. 验证通过后, 后端解析 token 中的用户信息等进行逻辑操作(一般是权限操作)

9. 前端没发出请求 url

全部	进行中	交易完成	交易失败
统计区间	2022-08-01 至 2022-08-31		
交易		交易完成	
2997520716036837376		¥0.01	
扫码付款			
2022-08-24 13:57		详情	
交易		交易完成	
2997206174136274944		¥0.01	
扫码付款			
2022-08-23 17:07		详情	

检查下是不是 api.js 中的路由配错了文件, vue 中是一个项目对应配置 api.js 的, 别配到其他项

```

getListData(data, page) {
  data.serUserId = this.param("serUserId");
  data.beginCreatedDate = this.beginDate;
  data.endCreatedDate = this.endDate;
  if (this.param("employeeName")){
    data.serUserName = this.param("employeeName");
  }
  return this.request(this.api.FindTransactionOrderInfos4Store, data, page).then((data) => {
    return data;
  });
},

```

```

/**
 * 交易查询
 * @param
 * @param
 * @param
 * @return
 * @throws Exception
 */
@RequestMapping(value = "/findTransactionOrderInfos4Store")
public List<TransactionOrderInfo> findTransactionOrderInfos(ParamData d, Pageable p, @Current StoreUserInfo u) throws Exception {
  Map<String, Object> m = d.getDataObj();
  return transactionOrderInfoService.findTransactionOrderInfos(m, p, u);
}

```

目上去了

10. 限制项传入筛选出新名字

因为做了管理员名字修改, 但是订单表之前用的是旧的名(id 不变的), 因为现在做分类筛选的时

```
@Override
public List<TransactionOrderInfo> findTransactionOrderInfos(Map<String, Object> map, Pageable p) throws Exception {
    StringBuffer hql = new StringBuffer();
    Map<String, Object> param = new HashMap<String, Object>();
    hql.append(" from TransactionOrderInfo as e where 1=1 ");
    genTransactionOrderInfoQuery(map, hql, param);
    hql.append(" order by e.createdDate desc");//e.sort desc,
    return super.find(hql.toString(), param, p);
}
```

候会查出旧的订单, 故应调整查出新的订单排除旧的订单.

```
/**
 * 条件查询参数
 *
 * @param map
 * @param hql
 * @param param
 * @throws Exception
 */
private void genTransactionOrderInfoQuery(Map<String, Object> map, StringBuffer hql, Map<String, Object> param) throws Exception {
    checkData(map, hql, param);
    for (Entry<String, Object> m : map.entrySet()) {
        String key = m.getKey();
        Object val = m.getValue();
        if (JUtil.isEmpty(val)) {
            continue;
        }
    }
}
```

```
2922
2923     } else {
2924         hql.append(" and e." + key + " =:" + key + " ");
2925         param.put(key, val);
2926     }
2927 }
2928
2929 hql.append(" and e.status =:status ");
2930 param.put("status", BaseInfo.STATUS_VALID);
2931 }
2932
```

Output Tomcat Localhost Log Tomcat Catalina Log

Variables

```
> hql.toString() = " from TransactionOrderInfo as e where 1=1 and e.serUserName =:serUserName and e.serUserName =:se
> this = {TransactionOrderInfoServiceImpl@9895}
> map = {JSONObject@10768} size = 7
> "serUserName" -> "庄玉龙"
> "dataStatus" -> "4,6,13,14"
> "beginCreatedDate" -> "2022-08-01"
> "endCreatedDate" -> "2022-08-31"
> "storeId" -> "8c4ce91e4e334d22aadaeccf3ce268c4"
> "serUserId" -> "36f3cf7b111a46c49dc921a31d809dde"
> "tradeType" -> {Integer@9877} 1
> p = {Pageable@10917}
> hql = {StringBuffer@10769} " from TransactionOrderInfo as e where 1=1 and e.serUserName =:serUserName and e.serUs
> param = {HashMap@10770} size = 11
```


点击触发的链接如下:

可见传入了 employeeName, 即筛选条件

× 标头 载荷 预览 响应 启动器 时间

HTTP Status 500 - Servlet execution threw an exception

type Exception report

message [Servlet execution threw an exception](#)

description [The server encountered an internal error that prevented it from fulfilling this request.](#)

exception

```
javax.servlet.ServletException: Servlet execution threw an exception
    org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
    org.springframework.web.servlet.view.InternalResourceView.renderMergedOutputModel(InternalResourceView.java:145)
    org.springframework.web.servlet.view.AbstractView.render(AbstractView.java:303)
    org.springframework.web.servlet.DispatcherServlet.render(DispatcherServlet.java:1257)
    org.springframework.web.servlet.DispatcherServlet.processDispatchResult(DispatcherServlet.java:1037)
    org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:980)
    org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:897)
    org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:970)
```

```
Server x Tomcat Localhost Log x Tomcat Catalina Log x
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
at javax.servlet.http.HttpServletRequestWrapper.getUserPrincipal(HttpServletRequestWrapper.java:184)
```

```
onSubmit(){
    this.request(this.api.SaveStoreInfo4Sys,this.dataInfo).then((res)=>{
        let obj={};
        obj.id=res.id;
        obj.liaisons=res.liaisons;
        obj.telephone=res.telephone;
        this.request(this.api.ChangeNameAndPhone,obj).then((res)=>{
            this.message("保存成功")
            return this.redirectTo("info?id="+res.id);
        });
    });
},
```

点进去页面展示内容由该方法实现

后端(传入参数, 分页参数, 和注解获取的当前用户)

```
/**
 * @author xiaojq
 * @date 2022/8/23 10:32
 * @description 改用户表手机号和名字
 */
@RestController
@RequestMapping( value=BaseAction.API_SRC+"store")
public class StoreBusinessInfoAction extends BaseAction {
```

```
//一个商户可能有多个员工(一个管理员多个员工)
StoreUserInfo newStoreUserInfo = storeUserInfoService.getStoreUserInfoList(info.getId()).stream()
    .filter(v -> StoreUserInfo.AUTH_TYPE_SUPER == v.getAuthType())
    .collect(Collectors.toList())
    .get(0);
```



如需使用产品完整功能，请点击此处关注
「通联飞收」公众号使用

去关注



修改商户名称测试



已认证 庄玉龙007 超级管理员



```
<span class="sui-text sui-inline sui-shm">{{app.currentUser.name}}</span>
```

```
refreshCurrentUser(){//刷新缓存的变量
  let data = this.getUserData();
  let storeInfo = this.getStoreDataByLocalStorage();
  this.app.cacheData = {};//TODO 清除全局缓存
  this.app.currentUser = data||{};//TODO 缓存到全局变量
  this.app.currentUser.storeInfo = storeInfo||{};//TODO 缓存到全局变量
  this.app.currentAuthValue = data.authValue || 0;//TODO 缓存用户的权限
  this.initPayMode()
},
```

```
//获取缓存数据
getUserData() {
  let user = this.util.getStorageData( key: this.env.userInfoKey || "UserInfo") || {};//获取用户缓存
  return user;
},
```

那么是如何实现的仅仅多传入一个参数不需要修改后端代码就可以实现筛选功能的呢？一步步跟进代码....

跟进代码，可以看到进行了动态获取前端传入的参数进行 hql 语句拼接
中间省略了代码

其中 param 这个 hashMap 中保存了真正的 key 与 value, 到时候拼接完的 hql 语句和 param

```
created() {
    console.info( data: "进入主窗口");
    //TODO 全局缓存数据, 重新登录会清空, 如果为空需要重新缓存, 不要太过依赖这里的数据
    this.app.cacheData={};

    this.app.globalData={//全局变量数据
        envType:this.getEnvType(),//当前环境
    }

    this.refreshUser();

    this.initListener();//初始化监听
    this.initAudioServer();
    this.initSocketServer();//websocket 监听消息
    this.refreshCurrentUser();//刷新缓存的变量

    this.app.getUserData=this.getUserData;//全局方法
    this.app.loginUserInfo=this.loginUserInfo;
    this.app.checkAuth=this.checkAuth;//全局判断权限
    this.app.createBackListener=this.createBackListener;//全局监听返回方法

    //获取全局的信息
    this.app.getStoreData=this.getStoreData;
    this.app.getStoreModule=this.getStoreModule;
    this.app.chooseImageUpload=this.chooseImageUpload;//选择图片上传
    this.app.chooseResourceUpload=this.chooseResourceUpload;
```

```
loginUserInfo(token,expiresIn){
    this.util.setStorageData(this.env.userTokenKey||'token',token);//保存token
    this.util.setStorageData(this.env.expiresTimeKey || 'expires', (Date.now() + (expiresIn||(2*60*60)) * 1000));
    return this.refreshUserInfo();//刷新用户信息
},
```

一起传入底层 hiberante 进行调用 find()即可完成查询操作

操作成功了但是写的异步请求却一直出现 500 错误

debug 断点进去方法可发现, 一直在循环调用该方法, 说明请求一直在循环发, 查看前端代码发现, 并没有写循环操作.

```

refreshUserInfo(){
    return this.request(this.api.GetCurrentUser, {}).then((data)=>{//data
        data = data || {};
        this.util.setStorageData(this.env.userInfoKey || 'UserInfo', data || {});
        this.refreshCurrentUser();
        return data;
    });
},

```

最终排查是后端使用了@Controller注解,进入了视图解析器,不知道底层进行了什么操作,最终换成@RestController解决(跳过视图解析器)

注意: @Controller 默认返回的是页面,经过视图解析器的解析,而@RestController 是 Controller + @ResponseBody 默认返回的是 json 数据,不经过视图解析器,

如果是要进行页面的返回,则使用@Controller注解

如果只需要返回 RestFul 数据,则大多使用@RestController注解(或者@Controller+@ResponseBody组合),

可能是由于用了 Controller 返回了页面然后又一直请求导致爆栈了,具体底层暂不了解,

注意: 记得继承 BaseAction 使用 BaseAction.API_SRC 进行拼接(规范点)

集合的流操作

这种写法点赞

更新了用户名数据首页展示未刷新

根据页面定位

点进去看到, 将获取到的 data 信息放进了全局变量 app.currentUser 中

跟进去得知是从缓存获取的

那要解决该问题, 得知从那里获取数据并不能解决问题, 我们还要知道从那里设置缓存的, 因为我们要做到在改变数据的时候顺便把缓存更新一下, 这样在加载这个页面的同时, 获取的缓存数据才不会是旧数据

根据业务可知, 登录时应该保存缓存, 则从该页面找用户登录做的操作

```
<!-- 协商缓存 -->
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache, no-store, must-revalidate">
<meta http-equiv="expires" content="0">
```

vue.runtime.min.js	304	script	(index)	179 B
vue-router.min.js	304	script	(index)	179 B
axios.min.js	304	script	(index)	179 B
jweixin-1.4.0.js	304	script	(index)	179 B
yMengconfig.js	304	script	(index)	178 B
chunk-vendors.js	304	script	(index)	181 B
chunk-common.js	304	script	(index)	181 B

```

server {

    listen      80;

    server_name 域名;

    root    文件目录;

    index    index.html;


    location / { // 不加这一句，会出现nginx欢迎页面，无法正确加载资源

        try_files $uri /index.html;

    }


    location ~ .*\. (html)$ { // 对html文件限制缓存

        add_header Cache-Control no-store; // 不缓存

        // 或者用add_header Cache-Control no-cache;替代上面那句，协商缓存

        add_header Pragma no-cache;

    }

}

```

可以找到登录做了刷新用户信息

跟进到这个方法里面, 查看到在这里将请求数据库查出信息放进了浏览器 localStorage 存储中 (这里点进去看到的), 故到了这里我们旧可以解决问题了, 在更新用户名字的时候调用这个方法进行更新到本地浏览器的缓存即可.

每次更新服务器后用户浏览器缓存不及时刷新问题

更新了正式环境后, 用户使用的 css 和 js 文件是缓存在本地浏览器中的, 不能及时刷新, 原因是使用了本地缓存导致我们更新了服务器用户不能及时使用到最新的内容。

浏览器请求有四种状态码

1. 200 from memory cache : 内存缓存, js、字体、图片
2. 200 from disk cache: 硬盘缓存, css 等
3. 200 数值大小 : 从浏览器中下载最新资源, 数值是浏览器获取的全部资源大小
4. 304 数值大小 : 访问浏览器协商, 没变则用本地, 数值是报文的大小不是资源的大小

浏览器三级缓存

1. 内存缓存 : 先从内存查
2. 硬盘缓存 : 内存不在则从硬盘查
3. 网络请求 : 硬盘不在则从网络查

浏览器的缓存分类

1. 强制缓存 :
 1. 用户第一次访问后, 缓存网页数据, 过期时间内都不再请求服务器。超出过期时间则协商缓存。



486	894.7	0
54	1274.2	0
9	584.2	0
0	8.05	0
0	0	0
0	1035	0
49.5	3144.1	0
13.5	510.6000000000	0

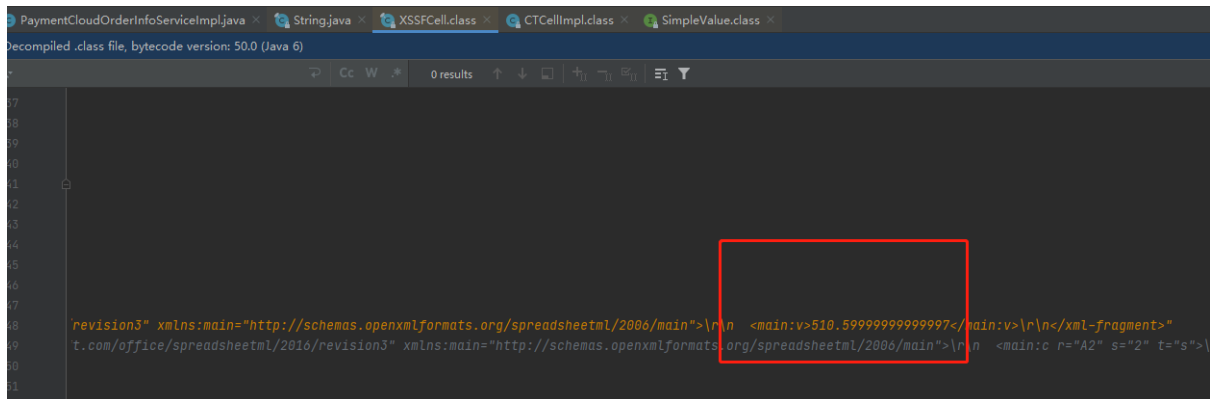
2. 强制缓存控制字段是 Expire 和 Cache-Control，其中 Cache-Control 比 Expire 优先级高
 3. 200 from memory 和 200 from disk 都属于强制缓存
2. 协商缓存
1. 浏览器和服务端协商，每次都和服务端通信。第一次请求时，返回资源和标识缓存到浏览器。后面的通信浏览器先发送标识给服务器询问是否改变，如果标识匹配则没变，返回 304，浏览器知道没变则读取缓存即可。
 2. 缓存协商字段是 Last-Modified/IF-Modified-Since、Etag/IF-None-Match
 3. Last-Modified 和 Etag 可以一起使用，服务器优先验证 Etag，一致的情况下才会继续对比 Last-Modified，最后才决定是否返回 304

缓存头 Cache-Control 的含义

1. public: 路径中所有节点都可缓存
2. private: 只有发起的浏览器中可缓存
3. no-cache: 向服务器发起协商决定是否使用缓存
4. no-store: 不使用缓存

分析更新时始终使用缓存的原因：

- vue 打包默认将静态文件使用了文件名加 hash 码的情况，所以每次更新后文件都发生变化
- 那么再 vue 程序入口 index.html 中可以将请求头的缓存情况设为协商缓存
- 更新本地环境可以看到，全部都走了 304 协商缓存



- 但是把本地环境更新到测试环境，发现还是全是 200 from disk 或者 from memory，走了强制缓存，分析原因，测试环境配置了 nginx，缓存了入口 index.html 页面导致
- 可以配置 nginx 解决（未测试，环境不敢乱动，怕被屌）

云订单模板

点击下载

上传

订单内容有误: 错误详情: undefined. 错误发生在第4行



点击上传模板文件

Excel 导入精度丢失问题

```
onChange(event){
  let that=this;
  let files = event.target.files;

  if(that.change) {
    let fn=that.change.call(that, files);
    fn&&fn.then((res) => {
      that.value = res;
      //回调input 出入事件
      return res
    }).then(res => that.$emit('change', res));
    // 2022年9月13, 解决云订单导入后输入框导入错误文件后再次导入正确文件仍提示错误文件
    event.target.value=''
  }
}
```

总所周知, 计算机存储浮点型数据会存在精度丢失问题. 原理如下:

1. 整数转换成二进制不存在精度丢失问题(除 2 取余总能除完)
 2. 小数会出现问题(乘 2 取整)
- 例如:小数 0.9
 - $0.9 \times 2 \rightarrow 1$
 - $0.8 \times 2 \rightarrow 1$
 - $0.6 \times 2 \rightarrow 1$
 - $0.2 \times 2 \rightarrow 0$
 - $0.4 \times 2 \rightarrow 0$
 - $0.8 \times 2 \rightarrow 1$
 - $0.6 \times 2 \rightarrow 1$
 -
 - 以上计算过程循环了,也就是说,算法将移植没法结束. 故小数的二进制表示有时候是不可能精确的.

float内存存储结构				
4bytes	31	30	29-----23	22-----0
表示	实数符号位	指数符号位	指数位	有效数位
其中符号位1表示正，0表示负。有效位24位，其中一位是实数符号位。				
将一个float型转化为内存存储格式的步骤为：				
(1) 先将这个实数的绝对值化为二进制格式，注意实数的整数部分和小数部分的二进制方法在上面已经探讨过了。 (2) 将这个二进制格式实数的小数点左移或右移n位，直到小数点移动到第一个有效数字的右边。 (3) 从小数点右边第一位开始数出二十三位数字放入第22到第0位。 (4) 如果实数是正的，则在第31位放入“0”，否则放入“1”。 (5) 如果n是左移得到的，说明指数是正的，第30位放入“1”。如果n是右移得到的或n=0，则第30位放入“0”。 (6) 如果n是左移得到的，则将n减去1后化为二进制，并在左边加“0”补足七位，放入第29到第23位。如果n是右移得到的或n=0，则将n化为二进制后在左边加“0”补足七位，再各位求反，再放入第29到第23位。				
举例说明：11.9的内存存储格式				
(1) 将11.9化为二进制后大约是“1011.1110011001100110011001100...”。				
(2) 将小数点左移三位到第一个有效位右侧：“1.01111100110011001100110 ”。保证有效位数24位，右侧多余的截取（误差在这里产生了）。				
(3) 这已经有了二十四位有效数字，将最左边一位“1”去掉，得到“01111100110011001100110 ”共23bit。将它放入float存储结构的第22到第0位。				
(4) 因为11.9是正数，因此在第31位实数符号位放入“0”。				
(5) 由于我们把小数点左移，因此在第30位指数符号位放入“1”。				
(6) 因为我们是把小数点左移3位，因此将3减去1得2，化为二进制，并补足7位得到0000010，放入第29到第23位。				
最后表示11.9为：0 1 0000010 011 1100110011001100110				

- 其中一个浮点数的二进制是整数部分和小数部分分别进行转换的,如下

有了以上理论基础, 那么可以看下业务中存在的问题:

可以看出，excel 导入到系统中的时候出现了精度解析错误的问题，定位 bug 发现，开始怀疑是代码出现问题，结果处理之前就已经发生了精度丢失，故翻阅 apache POI 的 excel 解析源码，debug 进去，发现是 excel 保存就已经发生了精度丢失，猜测应该是 excel 保存小数发生的浮点型精度丢失

- 解决办法：让客户输入数据为 text，不输入数字类型 numeric，保证 excel 的数据准确
-
- 备注：excel 中的坑：数据框中有数字的情况下，哪怕选中一列，改为文本，其实数据仍然是安装 numeric 的形式来存储的，必须手动点数据框回车一下才变为文本，导致改了文本数据格式仍然有问题。

BigDecimal.longValue()转后后为 0

实际上 BigDecimal 中的 longvalue()会自动对小数进行四舍五入，当输入 0.1 的时候，自然就变为 0 了

如果不想四舍五入，可以使用 doubleValue()

文件上传二次上传失败

在业务中，出现上传第一份错误文件再上传一份正确文件，正确文件仍然提示为错误

- 原因:

input 是通过 onchange 事件来触发 js 代码的，由于两次文件是重复的，所以这个时候 onchange 事件是没有触发到的。

- 解决:

读取文件后，记得把 input 的 value 重新设置为空即 e.target.value=""

Java 枚举的使用

- java 中，使用 enum 关键字定义一个枚举
- 枚举的 set 方法不能设置值进去，因为属性是用 static final 修饰的
- 枚举给我们提供了一个 values 方法来获取所有常量数组
-

```

* @description 传入的表单导出模式，用于业务判断匹配导出的sheet数据
*/
public enum ExportModeEnum {

    SUMMARY_SHEET(1),
    PAYMENT_DETAIL_SHEET(2),
    PAID_SHEET(3),
    UNPAID_SHEET(4);

    private int exportMode;

    ExportModeEnum(int mode) {
        this.exportMode = mode;
    }

    public int getExportMode() {
        return exportMode;
    }

    /**
     * 静态方法获取枚举
     *
     * @param exportMode
     * @return
     */
    public static ExportModeEnum getExportModeByInt(int exportMode) {
        for (ExportModeEnum exportModeEnum : ExportModeEnum.values()) {
            if (exportModeEnum.getExportMode() == exportMode) {
                return exportModeEnum;
            }
        }
        return null;
    }

    /**
     * 检查导出模式
     *
     * @param exportMode
     * @throws Exception
     */
    public static void check(int exportMode) throws Exception {
        for (ExportModeEnum e : ExportModeEnum.values()) {
            if (e.getExportMode() == exportMode) {
                return;
            }
        }
        throw new RuntimeException ("不支持的导出模式 " + exportMode);
    }
}

```

Vue2 中使用 Vant2 并使用 less 定制主题

- 如果需要自定义 vant2 组件的主题，需要使用 less 进行覆盖
- 需要使用 npm 安装 less 和 less-loader，有可能 less-loader 版本太高，需要降低

- 手动引入单个组件 vant 默认的 less 样式

```
// 引入单个组件样式
import 'vant/lib/button/style/less';
```

- 在 vue.config.js 中引入全局 less 文件

```
// vue.config.js
module.exports = {
  css: {
    loaderOptions: {
      less: {
        // 若 less-loader 版本小于 6.0, 请移除 lessOptions 这一级, 直接配置选项。
        lessOptions: {
          modifyVars: {
            // 直接覆盖变量
            'text-color': '#111',
            'border-color': '#eee',
            // 或者可以通过 less 文件覆盖 (文件路径为绝对路径)
            hack: `true; @import "your-less-file-path.less";`,
          },
        },
      },
    },
  },
};
```

引入的全局 less 文件可以为空文件, 空则代表不覆盖 vant2 的默认主题

- 如果需要在单个组件文件中使用 less 进行定制 vant 的组件样式, 需要在 style 中将 lang="less" 并使用 scoped 设置为单组件享有, 还必须使用 ::v-deep 进行组件样式的穿透, (因为 vant 组件属于的是全局的概念, scoped 限制了我们的组件, 写的 less 就对 vant 的组件无效了, 故需要使用 ::v-deep 将 less 的作用域穿透到 vant 的组件上去) 如下图:

```

<style lang="less" scoped>
  // 提示字体颜色
  @field-placeholder-text-color: #AC1B1B;
  // 输入框高度
  @field-text-area-min-height: 200px;
  .van-field{
    background-color: #f0f0f0;
    border-radius: 10px;
  }
  ::v-deep .van-field--min-height .van-field__control {
    min-height: @field-text-area-min-height;
  }
  ::v-deep textarea.van-field__control {
    font: message-box;
  }
</style>

```

解决前端 debugger 断点定位不准

- 在 vue.config.js 加上 source-map

```

configureWebpack: {
  // 解决断点定位不准
  devtool: 'source-map',
}

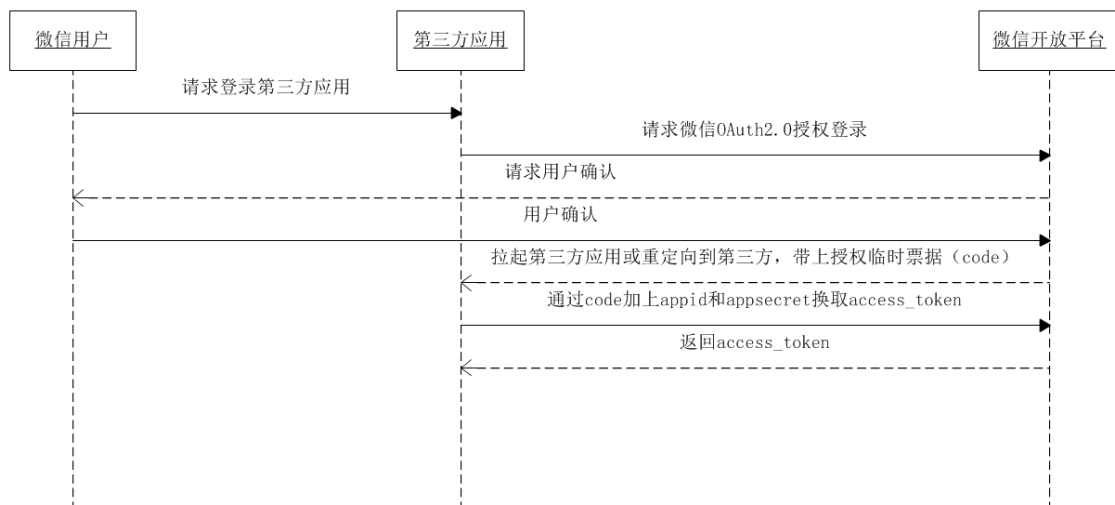
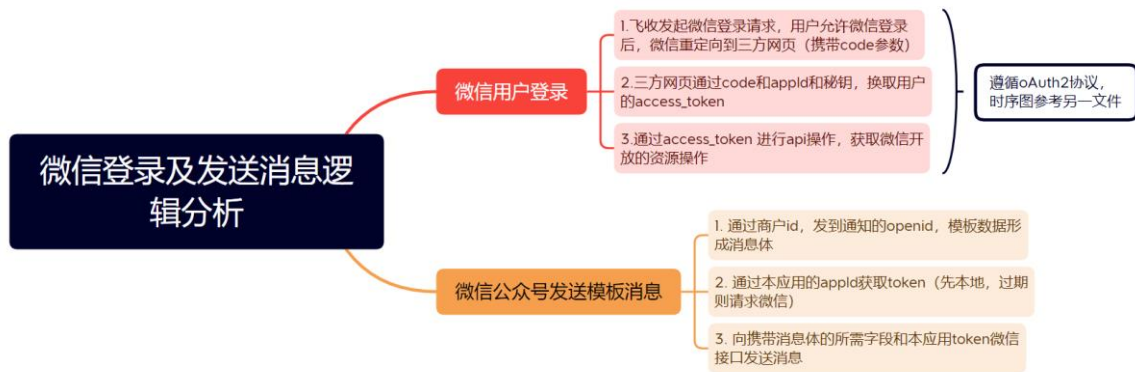
```

解决导出展示退款金额为 0 的 bug

原因：payment_order_info 表中的 refundAmount 字段只在全部退款的时候才更新，部分退款时 refundAmount 字段需要在 paymentOrderPays 中取

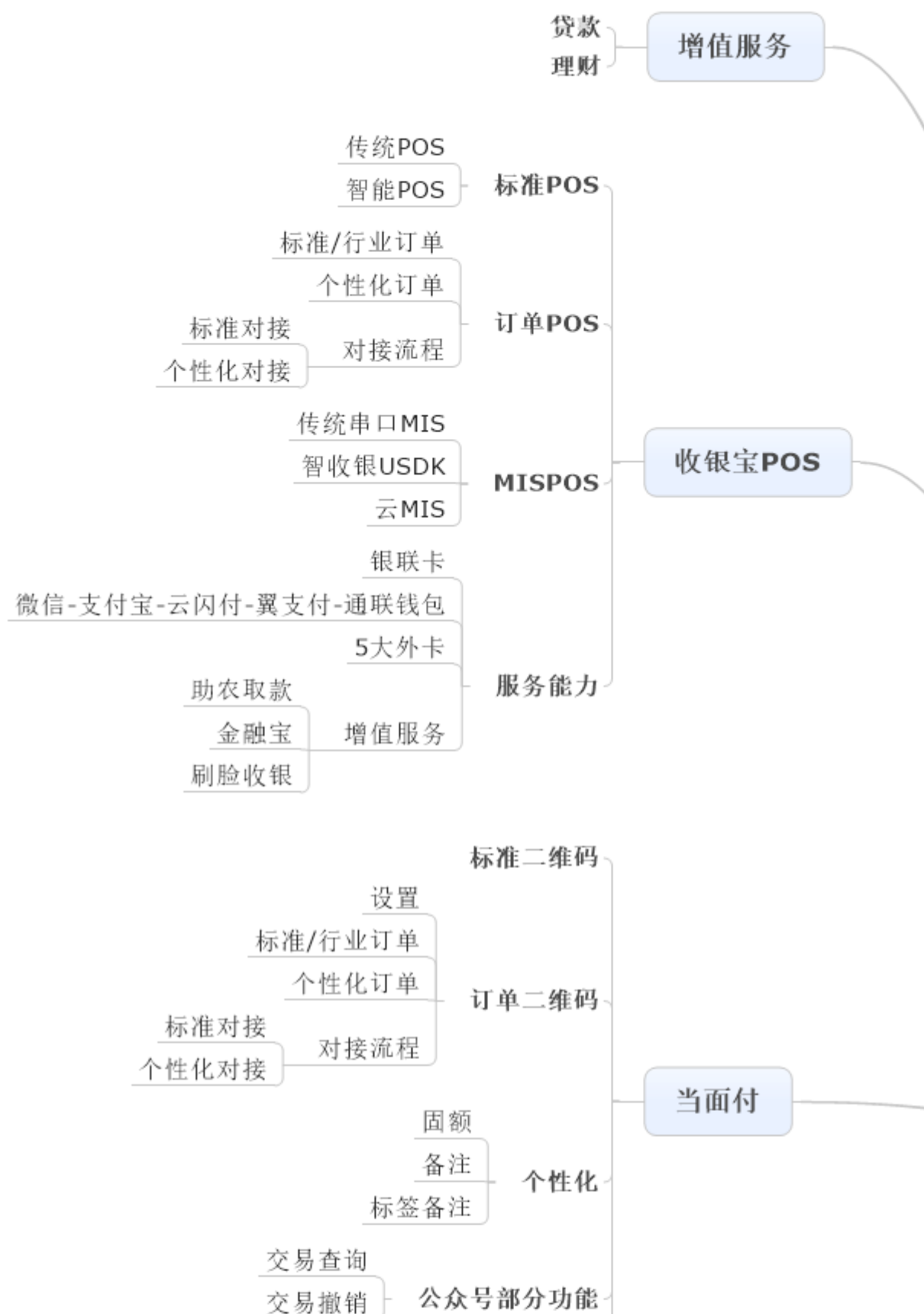
重构 open 微信交互项目的总结

微信交互所需要用到的接口逻辑总结：



收银宝业务培训

导图



自定义注解与切面编程

- 先定义一个注解

```
/**
 * Controller 层api环绕通知日志注解
 * 实现记录访问接口、入参、出参、接口性能
 */
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
xiaojq, Today · fix(wechatopen):添加环境切换，加入切面日志
public @interface ApiLog {
    String module() default "";

    String operation() default "";
}
```

- 定义切面

```

/**
 * @author xiaojq
 * @date 2023/5/19 10:47
 * @description 处理@ApiLog注解，给方法加入日志
 */
@Component
@Aspect // 切面，定义了通知和切点的关系
@Slf4j
public class LogAspect {
    // 切点
    @Pointcut("@annotation(com.future.modules.open.basic.annotation.ApiLog)")
    public void pointCut() {
    }

    // 环绕通知
    @Around("pointCut()")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        Stopwatch stopWatch = new Stopwatch(id: "api执行时间");
        stopWatch.start();
        // 执行方法
        Object result = joinPoint.proceed();
        // 执行时间
        stopWatch.stop();

        // 保存日志
        recordLog(joinPoint, stopWatch);
        return result;
    }
}

```

```

private void recordLog(ProceedingJoinPoint joinPoint, Stopwatch stopWatch) {
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Method method = signature.getMethod();
    ApiLog apiLog = method.getAnnotation(ApiLog.class);
    log.info("=====api log start=====");
    log.info("module:{", apiLog.module());
    log.info("operation:{", apiLog.operation());

    // 请求的方法名
    String className = joinPoint.getTarget().getClass().getName();
    String methodName = signature.getName();
    log.info("request method:{", className + "." + methodName + "()");

    // 请求的参数
    Object[] args = joinPoint.getArgs();
    String params = JSON.toJSONString(args[0]);
    log.info("params:{", params);
    log.info("execute time(second):{", stopWatch.getTotalTimeSeconds());
    log.info("=====api log end=====");
}
}

```