## 1.需求概述

### 1.1.商户注册

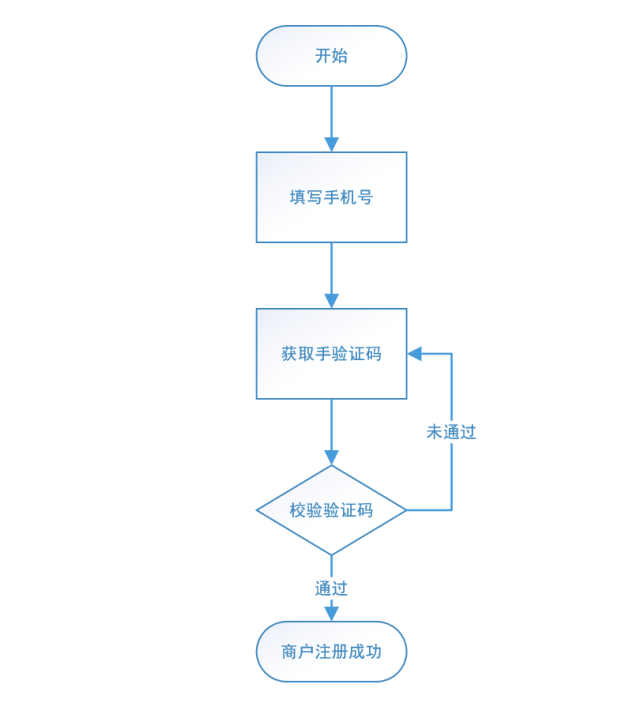
宜亩支付为商户提供聚合支付业务，线下商户和线上商户都可以使用宜亩支付平台。

什么是线下和线上商户？

1）**线下场所支付商户** 使用线下场所支付的商户是指有实体经营场所的商家，也称为地面商户，一般包含酒店、餐厅、酒吧、美容、美发、媒体、影楼、家政、艺廊、KTV、会所等。

2）线上支付商户 使用线上支付的商户是指通过互联网进行经营服务的商家，常见的有：电商网站、团购网站、旅游网站等。 商户使用宜亩支付平台第一步要在平台进行注册。 商户填写手机号、账号、密码、获取验证码申请注册，注册成功后商户成为宜亩平台的用户，即可使用宜亩支付平 台提供的服务。

商户注册的业务流程如下：



1. 用户填写手机号、账号、密码等信息

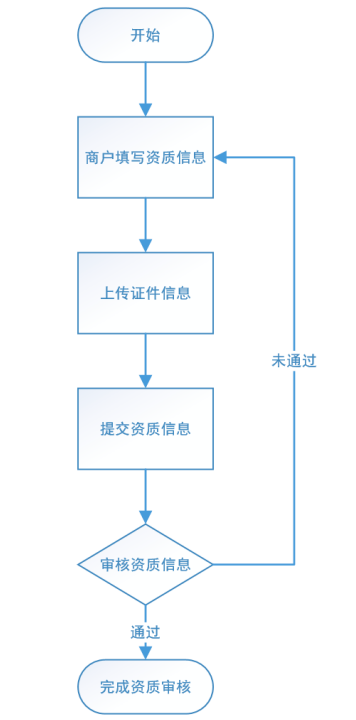


1. 点击获取手机验证码
2. 输入验证码，点击注册
3. 商户注册成功

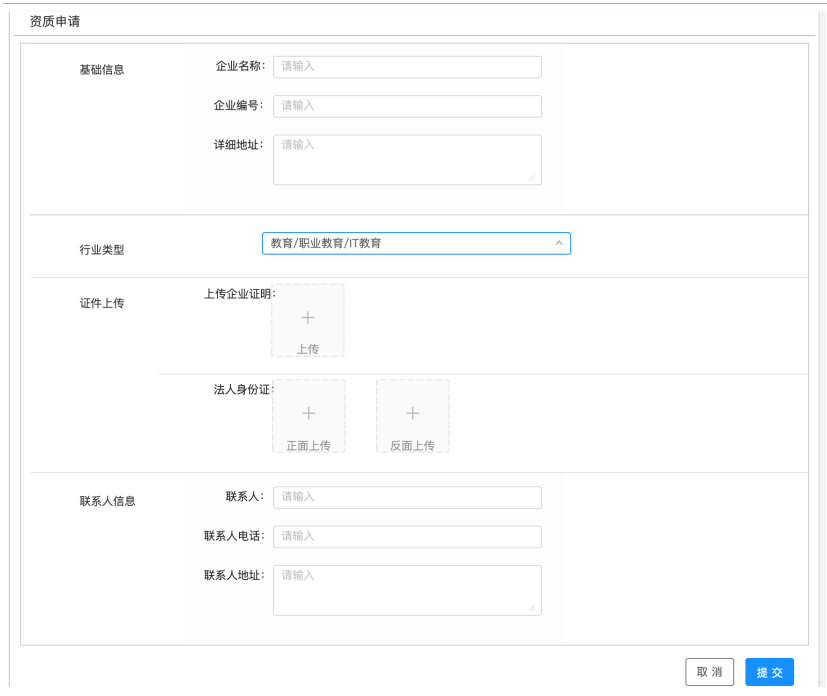
### 资质申请

商户在平台注册成功后，需要完善商户信息，将营业执照、申请人身份证以扫描件的形式上传到平台，由平台运营 人员对商户资质进行审核，审核通过方可使用平台提供的服务。

资质申请业务流程如下：



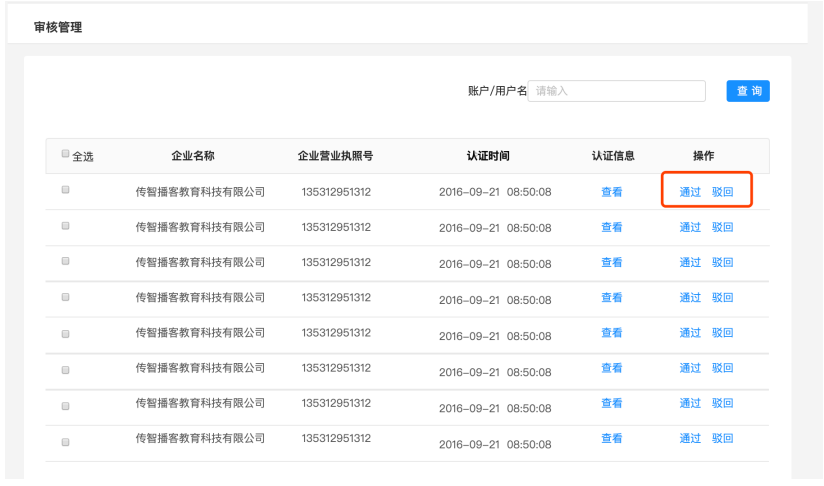
1. 商户填写资质信息



1. 上传营业执照和法人身份证图片



1. 提交资质信息
2. 平台运营人员对商户资质信息进行审核



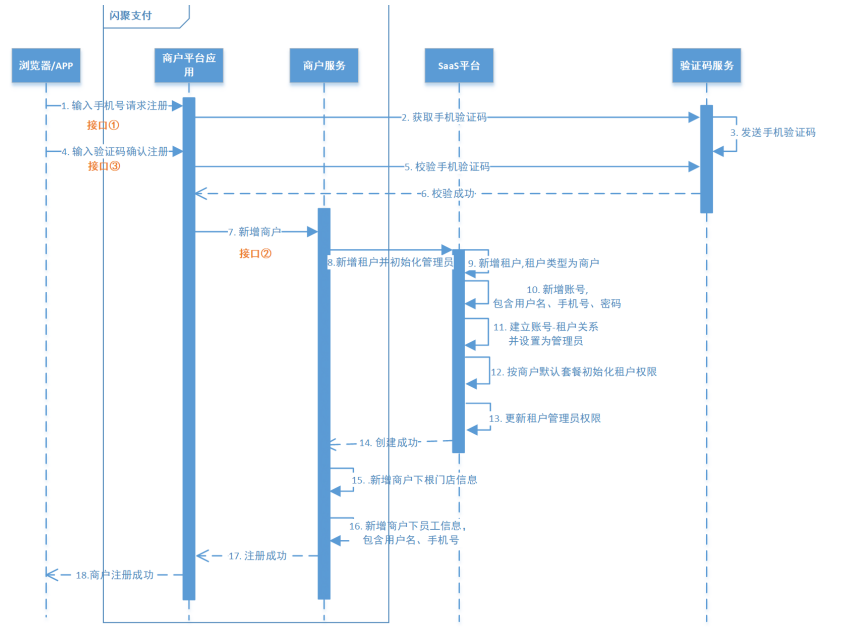
1. 审核通过后，完成资质申请

## 2.商户注册

### 2.1.需求分析

#### 2.1.1.系统交互流程

商户注册交互流程如下：



商户注册的流程由商户平台应用、商户服务、[**SaaS平台**](https://zhidao.baidu.com/question/560334435782643812.html)**（平台即服务）**、验证码服务四个微服务之间进行交互完成，各微服务的职责介绍如下：

1）商户平台应用：此应用主要为商户提供业务功能，包括：商户资质申请、员工管理、门店管理等功能。

2）商户服务： 提供商户管理的相关服务接口，供其它微服务调用，主要为商户平台应用提供接口服务，功能包括：商户基本信息管理、资质申请、商户应用管理、渠道参数配置、商户员工信息管理、商户门店管理等。

3）SaaS平台：宜亩支付项目是一个SaaS平台 ，**所谓SaaS平台即多个用户租用平台的业务功能，这样用户即可省去软件系统开发的成本，每个商户就是一个租户，所以又称为多租户系统。** SaaS平台提供租户管理、账号管理、权限管理、资源管理、套餐管理、系统认证授权等功业务功能。在上图商户注册的流程中，商户注册的账号等信息需要写SaaS平台，由SaaS平台统一管理账号，分配权限，商户统一通过 SaaS平台登录宜亩支付。

4）验证码服务：提供获取短信验证码、校验验证码的接口。 商户使用手机号进行注册，平台通过校验手机验证码来确认是否本人在注册。

交互流程如下：

1.前端请求商户平台应用进行注册

2.商户平台应用获取短信验证码

3.前端携带手机验证码、账号、密码等信息请求商户平台应用确认注册

4.验证码校验通过后请求商户服务新增商户

5.商户服务请求SaaS平台新增租户并初始化管理员

6.SaaS平台返回创建成功给商户服务商户服务新增商户下根门店信息

7.商户服务新增商户下员工信息

8.注册成功

#### 2.1.2.开发步骤

整个商户注册流程比较复杂，本模块采用迭代开发方式，具体开发步骤如下：

1.首先实现商户信息在商户服务注册成功（暂时不与SaaS平台交互） 商户信息只写入商户数据库，暂时不与SaaS平台交互。

2.待商户信息注册成功，资质申请通过、支付参数配置完成再与SaaS平台进行对接。 与SaaS平台交互前需要部署SaaS平台，学习SaaS暴露的接口及认证接口，接通SaaS方可实现用户登录，此部分放在本章节最后实现。

### 2.2.部署验证码服务

系统中所有验证码相关的功能由验证码服务提供，验证码服务是一个开源的项目，接入了腾讯、阿里等短信接口， 本系统只需要和验证码服务接入即可使用腾讯、阿里等短信接口。

参照：[1.验证码服务使用指南.docx](../00234002%20宜亩网（支付项目）2-各种手册/1.验证码服务使用指南.docx)

### 2.3.获取短信验证码

根据系统整体交互流程，需要首先获取短信验证码。

#### 2.3.1.RestTemplate技术预研

##### 2.3.1.1.认识RestTemplate

验证码服务对外提供http接口，我们使用的postman和swagger-ui都属于http客户端的一种，使用它们可以调用验证码服务的接口获取验证码。现在我们需要使用Java程序模拟http客户端调用验证码服务的接口获取验证码。

RestTemplate是Spring提供的用于访问RESTful服务的客户端，RestTemplate提供了多种便捷访问远程Http服务的方法,能够大大提高客户端的编写效率。RestTemplate默认依赖JDK提供http连接的能力 （HttpURLConnection），也可以通过替换为例如Apache HttpComponents、Netty或OkHttp等其它HTTP 客户端，OkHttp的性能优越，本项目使用OkHttp，

官网：<https://square.github.io/okhttp/>，

github： <https://github.com/square/okhttp>。

在yimupay-merchant-application工程中引入依赖：

|  |
| --- |
| <!-- okhttp3依赖 -->  <dependency>  <groupId>com.squareup.okhttp3</groupId>  <artifactId>okhttp</artifactId>  </dependency> |

在父工程已规范了okhttp的版本:

|  |
| --- |
| <dependency>  <groupId>com.squareup.okhttp3</groupId>  <artifactId>okhttp</artifactId>  <version>3.9.1</version>  </dependency> |

在MerchantApplicationBootstrap类中添加RestTemplate初始化：

|  |
| --- |
| /\*\*  \* RestTemplate初始化  \* **@return**  \*/  @Bean  **public** RestTemplate restTemplate() {  **return** **new** RestTemplate(**new** OkHttp3ClientHttpRequestFactory());  } |

在test下创建测试程序如下：

使用RestTemplate获取百度的网页内容。

|  |
| --- |
| **package** com.me.yimu.pay.merchant;  **import** com.alibaba.fastjson.JSON;  **import** lombok.extern.slf4j.Slf4j;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.http.\*;  **import** org.springframework.test.context.junit4.SpringRunner;  **import** org.springframework.util.LinkedMultiValueMap;  **import** org.springframework.util.MultiValueMap;  **import** org.springframework.web.client.RestTemplate;  @SpringBootTest  @RunWith(SpringRunner.**class**)  @Slf4j  **public** **class** RestTemplateTest {  @Autowired  RestTemplate restTemplate;    /\*\*  \* 使用RestTemplate获取百度的网页内容  \*/  @Test  **public** **void** getHtml() {  String url = "http://www.baidu.com/";  **ResponseEntity<String> forEntity = restTemplate.getForEntity(url, String.class);**  String body = forEntity.getBody();  System.***out***.println(body);  }    } |

通过测试发现可以成功获取百度的网页内容。

网页内容中中文乱码解决方案：

原因： 当RestTemplate默认使用String存储body内容时默认使用ISO\_8859\_1字符集。

解决：配置StringHttpMessageConverter消息转换器，使用utf-8字符集。修改RestTemplate的定义方法。

|  |
| --- |
| /\*\*  \* RestTemplate初始化  \* **@return**  \*/  @Bean  **public** RestTemplate restTemplate() {  RestTemplate restTemplate = **new** RestTemplate(**new** OkHttp3ClientHttpRequestFactory());  //消息转换器列表  List<HttpMessageConverter<?>> messageConverters = restTemplate.getMessageConverters();  //配置消息转换器StringHttpMessageConverter，并设置utf‐8  messageConverters.set(1,  **new** StringHttpMessageConverter(StandardCharsets.***UTF\_8***));//支持中文字符集，默认ISO‐8859‐1，支持utf‐8  **return** restTemplate;  } |

##### 2.3.1.2.使用RestTemplate获取验证码

下边通过RestTemplate请求验证码服务获取验证码。

|  |
| --- |
| /\*\*  \* 获取验证码测试方法  \*/  @Test  **public** **void** testGetSmsCode(){  String url = "http://localhost:56085/sailing/generate?name=sms&effectiveTime=600";//验证码过期时间为600秒  String phone = "13434343434";  ***log***.info("调用短信微服务发送验证码：url:{}", url);  //请求体  Map<String, Object> body = **new** HashMap();  body.put("mobile", phone);  //请求头  HttpHeaders httpHeaders = **new** HttpHeaders();  //设置数据格式为json  httpHeaders.setContentType(MediaType.***APPLICATION\_JSON***);  //封装请求参数  HttpEntity entity = **new** HttpEntity(body,httpHeaders);  Map responseMap = **null**;  **try** {  //post请求  ResponseEntity<Map> exchange = restTemplate.exchange(url, HttpMethod.***POST***, entity, Map.**class**);  ***log***.info("调用短信微服务发送验证码: 返回值:{}", JSON.*toJSONString*(exchange));  //获取响应  responseMap = exchange.getBody();  } **catch** (Exception e) {  ***log***.info(e.getMessage(), e);  }    //取出body中的result数据  **if** (responseMap != **null** || responseMap.get("result") != **null**) {  Map resultMap = (Map) responseMap.get("result");  String value = resultMap.get("key").toString();  System.***out***.println(value);  }  } |

#### 2.3.2.商户平台应用获取验证码(接口①)

##### 2.3.2.1.定义接口

在商户平台应用工程定义获取验证码接口：

1. 接口交互流程如下：

1）获取手机号

2）向验证码服务请求发送验证码并得到响应

3）响应前端验证码发送结果

1. 在MerchantController类中添加getSMSCode接口方法，此接口供前端调：

|  |
| --- |
| @ApiOperation("获取手机验证码")  @ApiImplicitParam(name = "phone", value = "手机号", required = **true**, dataType = "String",  paramType = "query")  @GetMapping("/sms")  **public** String getSMSCode(@RequestParam String phone) {  **return** phone;  } |

3）Service接口定义如下：

为了方便程序复用在 SmsService 接口中添加sendMsg方法用来获取验证码：

|  |
| --- |
| /\*\*  \* 获取短信验证码  \* **@param** phone  \* **@return**  \*/  String sendMsg(String phone); |

##### 2.3.2.2.接口实现

1）配置验证码服务的接口地址

在nacos上配置验证码的接口地址：

1. 登录nacos，编辑merchant-application.yaml配置

|  |
| --- |
|  |

1. 配置如下参数

|  |
| --- |
| sms:  url: "http://localhost:56085/sailing"  effectiveTime: 600 |

url：验证码服务地址

effectiveTime：验证码过期时间

2）编写接口实现方法

编写SmsServiceImpl实现sendMsg方法。

|  |
| --- |
| /\*\*  \* **@author** Administrator  \* **@version** 1.0  \*\*/  @org.springframework.stereotype.Service //实例为一个bean  @Slf4j  **public** **class** SmsServiceImpl **implements** SmsService{  @Value("${sms.url}")  String url;  @Value("${sms.effectiveTime}")  String effectiveTime;  @Autowired  RestTemplate restTemplate;  /\*\*  \* 发送手机验证码  \*  \* **@param** phone 手机号  \* **@return** 验证码对应的key  \*/  @Override  **public** String sendMsg(String phone) {  //向验证码服务发送请求的地址  String sms\_url = url+"/generate?name=sms&effectiveTime="+effectiveTime;  //请求体  Map<String,Object> body = **new** HashMap<>();  body.put("mobile",phone);  //请求头  HttpHeaders httpHeaders =**new** HttpHeaders();  //指定Content-Type: application/json  httpHeaders.setContentType(MediaType.***APPLICATION\_JSON***);  //请求信息,传入body，header  HttpEntity httpEntity = **new** HttpEntity(body,httpHeaders);  //向url请求  ResponseEntity<Map> exchange = **null**;  Map bodyMap = **null**;  **try** {  exchange = restTemplate.exchange(sms\_url, HttpMethod.***POST***, httpEntity, Map.**class**);  ***log***.info("请求验证码服务，得到响应:{}", JSON.*toJSONString*(exchange));  bodyMap = exchange.getBody();  } **catch** (RestClientException e) {  e.printStackTrace();  **throw** **new** RuntimeException("发送验证码失败");  }  **if**(bodyMap == **null** || bodyMap.get("result") == **null**){  **throw** **new** RuntimeException("发送验证码失败");  }  Map result = (Map) bodyMap.get("result");  String key = (String) result.get("key");  ***log***.info("得到发送验证码对应的key:{}",key);  **return** key;  } |

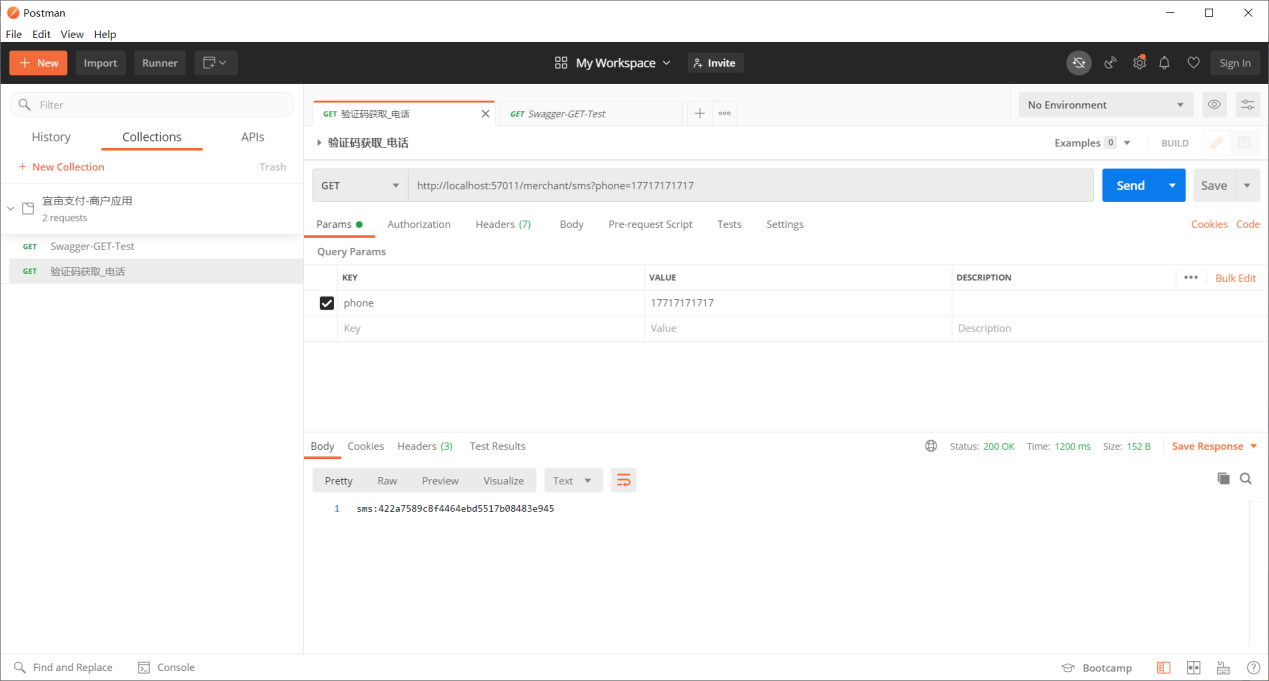
3)Controller层实现

|  |
| --- |
| @ApiOperation("获取手机验证码")  @ApiImplicitParam(name = "phone", value = "手机号", required = **true**, dataType = "String",  paramType = "query")  @GetMapping("/sms")  **public** String getSMSCode(@RequestParam String phone) {  ***log***.info("向手机号:{}发送验证码", phone);  **return** smsService.sendMsg(phone);  } |

3）接口测试

使用PostMan获取验证码接口：http://localhost:57011/merchant/sms?phone=17717171717

##### 2.3.2.3.接口测试



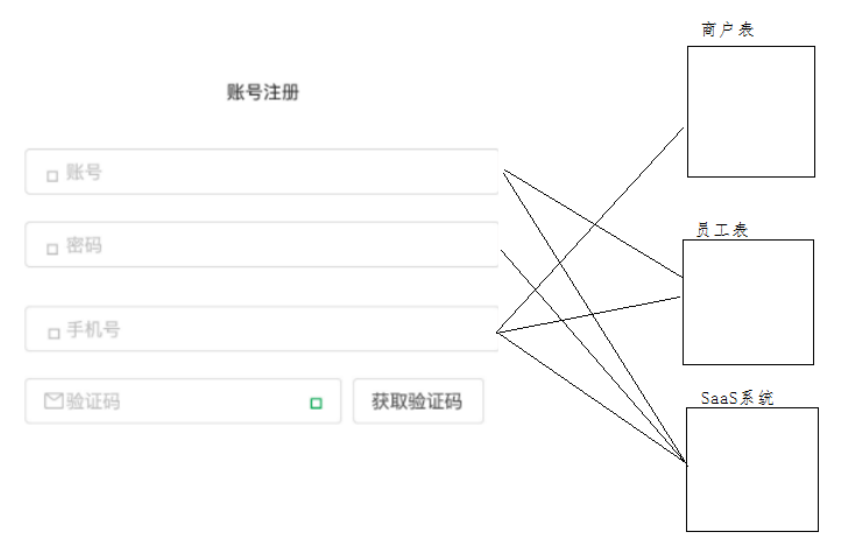
### 2.4.商户注册

#### 2.4.1.系统设计

商户拿到获取的验证码即可提交注册信息，完成商户注册，本节完成商户注册功能。

商户注册信息向三个地方写入数据 ，如下图：

当前阶段暂时只向商户表写入数据 ，待接入SaaS系统时向员工表、SaaS系统写入。



商户注册表的数据模型如下：



#### 2.4.2.商户服务注册商户(接口②)

商户服务提供商户注册的服务接口供商户平台应用使用，本节实现商户服务注册商户接口。

##### 2.4.2.1.接口定义

在商户服务定义商户注册接口

1、接口描述如下：

1）校验商户注册的基础数据：账号、密码、手机号 。

2）添加商户。

注意：本节暂不实现对接SaaS系统，仅实现新增一个商户，并将手机号填写到商户表。

3）返回注册结果给商户平台应用 。

2、定义商户注册[DTO](https://baike.baidu.com/item/DTO/16016821?fr=aladdin) 此DTO作为商户注册、资质申请的公共类。

在yimupay-merchant-api工程的com.me.yimu.pay.merchant.api.dto包下添加：

MerchantDTO：

|  |
| --- |
| package com.me.yimu.pay.merchant.api.dto;  import java.io.Serializable;  import io.swagger.annotations.ApiModel;  import io.swagger.annotations.ApiModelProperty;  import lombok.Data;  /\*\*  \* @author Administrator  \* @version 1.0  \*\*/  @ApiModel(value = "MerchantDTO", description = "商户信息")  @Data  public class MerchantDTO implements Serializable {  @ApiModelProperty("商户id")  private Long id;  @ApiModelProperty("企业名称")  private String merchantName;  @ApiModelProperty("企业编号")  private String merchantNo;  @ApiModelProperty("企业地址")  private String merchantAddress;  @ApiModelProperty("行业类型")  private String merchantType;  @ApiModelProperty("营业执照")  private String businessLicensesImg;  @ApiModelProperty("法人身份证正面")  private String idCardFrontImg;  @ApiModelProperty("法人身份证反面")  private String idCardAfterImg;  @ApiModelProperty("联系人")  private String username;  @ApiModelProperty("密码")  private String password;  @ApiModelProperty("手机号,关联统一账号")  private String mobile;  @ApiModelProperty("联系人地址")  private String contactsAddress;  @ApiModelProperty("审核状态,0-未申请,1-已申请待审核,2-审核通过,3-审核拒绝")  private String auditStatus;  @ApiModelProperty("租户ID")  private Long tenantId;  } |

DTO可以使用“代码生成工具”自动生成：

修改代码生成工具中的MyBatisPlusGenerator类：

|  |
| --- |
| **private** **static** **final** Boolean ***IS\_DTO*** = **true**;  // 生成DTO文件配置  **if** (***IS\_DTO***) {  globalConfig.setSwagger2(**true**);  globalConfig.setEntityName("%sDTO");  packageConfig.setEntity("dto");  } |

当生成dto时将IS\_DTO设置为true，运行生成工具即可。

3、接口定义如下： 在MerchantService接口类中定义如下接口：

|  |
| --- |
| /\*\*  \* 商户注册  \* **@return**  \*/  MerchantDTO createMerchant(); |

##### 2.4.2.2.接口实现

实现MerchantServiceImpl中的createMerchant方法：

|  |
| --- |
| /\*\*  \* 商户注册  \* **@param** merchantDTO  \* **@return**  \*/  @Override  @Transactional  **public** MerchantDTO createMerchant(MerchantDTO merchantDTO) {  Merchant merchant = **new** Merchant();    // 设置审核状态0‐未申请,1‐已申请待审核,2‐审核通过,3‐审核拒绝  merchant.setAuditStatus("0");  // 设置手机号  merchant.setMobile(merchantDTO.getMobile());    // 保存商户  merchantMapper.insert(merchant);    // 返回新增商户id  merchantDTO.setId(merchant.getId());    **return** merchantDTO;  } |

#### 2.4.3.商户平台应用注册用户(接口③)

商户平台应用调用商户服务提供的商户注册接口即可完成商户注册业务，本节实现商户平台应用注册商户接口的开发。

##### 2.4.3.1.接口定义

1、接口描述如下：

1）接收商户填写的注册数据

2）商户平台应用校验手机验证码

3）请求商户服务进行商户注册

4）返回注册结果给前端

2、定义商户注册VO：

在com.me.yimu.pay.merchant.vo包下添加VO类型，VO类型负责接收前端请求的数据。

下图是商户注册的用户界面，根据用户界面中的元素定义VO类的属性：



|  |
| --- |
| package com.me.yimu.pay.merchant.vo;  import io.swagger.annotations.ApiModel;  import io.swagger.annotations.ApiModelProperty;  import lombok.Data;  import java.io.Serializable;  /\*\*  \* 用于接收前端提交的数据  \* @author Administrator  \* @version 1.0  \*\*/  @ApiModel(value = "MerchantRegisterVO", description = "商户注册信息")  @Data  public class MerchantRegisterVO implements Serializable {  @ApiModelProperty("商户手机号")  private String mobile;  @ApiModelProperty("商户用户名")  private String username;  @ApiModelProperty("商户密码")  private String password;  @ApiModelProperty("验证码的key")  private String verifiykey;  @ApiModelProperty("验证码")  private String verifiyCode;  } |

3、接口定义如下：

在MerchantController类中定义如下接口：

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  // 校验验证码    // 注册商户    **return** merchantRegister;  } |

##### 2.4.3.2.校验验证码实现

1. 在SmsService中定义校验验证码的service接口

|  |
| --- |
| /\*\*  \* 校验手机验证码  \* **@param** verifiyKey 验证码的key  \* **@param** verifiyCode 验证码  \*/  **void** checkVerifiyCode(String verifiyKey,String verifiyCode) **throws** BusinessException; |

2. 在SmsServiceImpl中定义校验验证码的service 实现方法

|  |
| --- |
| /\*\*  \* 校验手机验证码  \*  \* **@param** verifiyKey 验证码的key  \* **@param** verifiyCode 验证码  \*/  @Override  **public** **void** checkVerifiyCode(String verifiyKey, String verifiyCode) **throws** BusinessException {  //String url = "http://localhost:56085/sailing/verify?name=sms&verificationCode="+verifiyCode+"&verificationKey="+verifiyKey;  //校验验证码的url  String smsUrl = url + "/verify?name=sms&verificationCode="+verifiyCode+"&verificationKey="+verifiyKey;    Map bodyMap = **null**;  **try** {  //使用restTemplate请求验证码服务  ResponseEntity<Map> exchange = restTemplate.exchange(smsUrl, HttpMethod.***POST***, HttpEntity.***EMPTY***, Map.**class**);  ***log***.info("请求验证码服务，得到响应:{}", JSON.*toJSONString*(exchange));  bodyMap = exchange.getBody();  }**catch** (Exception e){  e.printStackTrace();  ***log***.info(e.getMessage(), e);  **throw** **new** BusinessException(CommonErrorCode.***E\_100102***);  }  **if**(bodyMap == **null** || bodyMap.get("result") == **null** || !(Boolean) bodyMap.get("result")){  **throw** **new** BusinessException(CommonErrorCode.***E\_100102***);  }  }  } |

3. 在controller中调用校验验证码

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  **// 校验验证码**  **smsService.checkVerifiyCode(merchantRegister.getVerifiykey(),**  **merchantRegister.getVerifiyCode());**    // 注册商户    **return** merchantRegister;  } |

##### 2.4.3.3.注册商户实现

在Controller中调用Dubbo接口实现注册商户:

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  // 校验验证码  smsService.checkVerifiyCode(merchantRegister.getVerifiykey(),  merchantRegister.getVerifiyCode());    **// 注册商户**  **MerchantDTO merchantDTO = new MerchantDTO();**  **merchantDTO.setUsername(merchantRegister.getUsername());**  **merchantDTO.setMobile(merchantRegister.getMobile());**  **merchantDTO.setPassword(merchantRegister.getPassword());**  **merchantService.createMerchant(merchantDTO);**  **return** merchantRegister;  } |

##### 2.4.3.4.接口测试

下面使用Postman对注册商户整体流程进行测试：

1.获取短信验证码: http://localhost:57011/merchant/sms?phone=17717171717 过程略。

2.商户注册：http://localhost:57011/merchant/merchants/register，填写注册信息：

|  |
| --- |
| {  "mobile": "17717171715",  "username": "test14",  "password": "989504",  "verifiykey":"sms:614b9bcf39ce420fb03e59ebeb9cc8f6",  "verifiyCode":"130919"  } |

|  |
| --- |
|  |

3.注册成功

#### 2.4.4.MapStruct对象转换

##### 2.4.4.1.数据传输对象

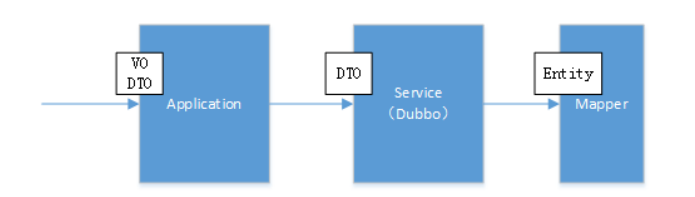
在商户注册开发过程中用于数据传输的对象有MerchantRegisterVO、MerchantDTO、entity(实体类)，它们的用 途如下：

1、MerchantRegisterVO用于**应用层接收前端请求及响应前端数据**。

2、MerchantDTO 用于服务层传入及响应数据。

3、entity(实体类) 用于持久层传入及响应数据。

如下图：



数据传输对象（Data Transfer Object）是系统在交互过程中根据需要及规范将数据封装到数据对象中进行传输。

本项目数据传输对象的规范：

1、应用层

如没有接口参数的特殊要求，应用层使用DTO结尾的对象传输，否则单独定义VO结尾的对象传输。

2、服务层

统一使用DTO结尾的对象传输。

3、持久层

统一使用Entity对象传输。

##### 2.4.4.2.MapStruct

1、数据传输对象转换的繁琐

每层有自己的数据传输对象，当数据流程到该层由需要将数据转成符合要求的格式，比如：当数据由应用层流转到服务层则需要将数据转成DTO格式，当数据由服务层流向持久层则需要将数据转成Entity格式数据

下边的代码数据由服务层流向持久层：

|  |
| --- |
| /\*\*  \* 商户注册  \* **@param** merchantDTO  \* **@return**  \*/  @Override  @Transactional  **public** MerchantDTO createMerchant(MerchantDTO merchantDTO) {  Merchant merchant = **new** Merchant();    // 设置审核状态0‐未申请,1‐已申请待审核,2‐审核通过,3‐审核拒绝  merchant.setAuditStatus("0");  // 设置手机号  merchant.setMobile(merchantDTO.getMobile());    // 保存商户  merchantMapper.insert(merchant);    // 返回新增商户id  merchantDTO.setId(merchant.getId());    **return** merchantDTO;  } |

上边代码的问题是：由merchantDTO转成entity实现过程繁琐。

2、MapStruct解决数据传输对象转换的繁琐 MapStruct是一个代码生成器，它基于约定优于配置的方法大大简化了Java Bean对象之间的映射转换的实现。 MapStruct 使用简单的方法即可完成对象之间的转换，它速度快、类型安全且易于理解。

官方地址：<https://mapstruct.org/>

1）添加依赖

在使用MapStruct的工程添加MapStruct依赖：

|  |
| --- |
| <!-- 数据转换利用 -->  <dependency>  <groupId>org.mapstruct</groupId>  <artifactId>mapstruct-jdk8</artifactId>  </dependency>  <dependency>  <groupId>org.mapstruct</groupId>  <artifactId>mapstruct-processor</artifactId>  <version>${org.mapstruct.version}</version>  </dependency> |

2）服务层对象转换

在商户服务工程定义商户对象转换类。

定义MerchantConvert转换类，使用**@Mapper注解**快速实现对象转换。

|  |
| --- |
| @Mapper //对象属性的映射  **public** **interface** MerchantConvert {  //转换类实例  MerchantConvert ***INSTANCE*** = Mappers.*getMapper*(MerchantConvert.**class**);  //把dto转换成entity  Merchant dto2entity(MerchantDTO merchantDTO);  //把entity转换成dto  MerchantDTO entity2dto(Merchant merchant); |

在MerchantCovert中定义测试方法：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **//将dto转成entity**  **Merchant merchant =new Merchant();**  **merchant.setUsername("测试");**  **merchant.setMobile("123456");**  **MerchantDTO merchantDTO = MerchantConvert.*INSTANCE*.entity2dto(merchant);**  **System.*out*.println(merchantDTO);**  **//将entity转成dto**  **merchantDTO.setMerchantName("商户名称");**  **Merchant merchant1 = MerchantConvert.*INSTANCE*.dto2entity(merchantDTO);**  **System.*out*.println(merchant1);**  //定义的list  List entityList = **new** ArrayList();  entityList.add(merchant);  //将lIST转成包含dto的list  List list = MerchantConvert.***INSTANCE***.entityList2dtoList(entityList);  System.***out***.println(list);  } |

List数据也可以转换：

在MerchantCovert中定义list的方法，如下：

|  |
| --- |
| //list之间也可以转换，很entity的List转成MerchantDTO list  List<MerchantDTO> entityList2dtoList(List<Merchant> merchants); |

测试：

在main方法编写list之间的转换测试：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //将dto转成entity  Merchant merchant =**new** Merchant();  merchant.setUsername("测试");  merchant.setMobile("123456");  MerchantDTO merchantDTO = MerchantConvert.***INSTANCE***.entity2dto(merchant);  System.***out***.println(merchantDTO);  //将entity转成dto  merchantDTO.setMerchantName("商户名称");  Merchant merchant1 = MerchantConvert.***INSTANCE***.dto2entity(merchantDTO);  System.***out***.println(merchant1);  **//定义的list**  **List entityList = new ArrayList();**  **entityList.add(merchant);**  **//将lIST转成包含dto的list**  **List list = MerchantConvert.*INSTANCE*.entityList2dtoList(entityList);**  **System.*out*.println(list);**  } |

3）应用层对象转换

在商户平台应用工程定义商户对象转换类。

|  |
| --- |
| package com.me.yimu.pay.merchant.convert;  import org.mapstruct.Mapper;  import org.mapstruct.factory.Mappers;  import com.me.yimu.pay.merchant.api.dto.MerchantDTO;  import com.me.yimu.pay.merchant.vo.MerchantRegisterVO;  /\*\*  \* 将商户注册vo和dto进行转换  \* Created by Administrator.  \*/  @Mapper  public interface MerchantRegisterConvert {  MerchantRegisterConvert INSTANCE = Mappers.getMapper(MerchantRegisterConvert.class);  //将dto转成vo  MerchantRegisterVO dto2vo(MerchantDTO merchantDTO);  //将vo转成dto  MerchantDTO vo2dto(MerchantRegisterVO merchantRegisterVO);  } |

##### 2.4.4.3.代码优化

1、优化服务层代码

修改商户服务工程MerchantServiceImpl中的createMerchant方法：

|  |
| --- |
| /\*\*  \* 商户注册  \* **@param** merchantDTO  \* **@return**  \*/  @Override  @Transactional  **public** MerchantDTO createMerchant(MerchantDTO merchantDTO) {  // 将DTO转换为Entity  Merchant merchantEntity = MerchantConvert.***INSTANCE***.dto2entity(merchantDTO);    // 设置审核状态0‐未申请,1‐已申请待审核,2‐审核通过,3‐审核拒绝  merchantEntity.setAuditStatus("0");    // 保存商户  merchantMapper.insert(merchantEntity);    //将entity转成 dto  MerchantDTO merchantDTONew = MerchantConvert.***INSTANCE***.entity2dto(merchantEntity);    **return** merchantDTONew;  } |

2、代码应用层代码

修改商户平台应用工程MerchantController中的registerMerchant方法：

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  // 校验验证码  smsService.checkVerifiyCode(merchantRegister.getVerifiykey(),  merchantRegister.getVerifiyCode());    // 注册商户  MerchantDTO merchantDTO =  MerchantRegisterConvert.***INSTANCE***.vo2dto(merchantRegister);  merchantService.createMerchant(merchantDTO);  **return** merchantRegister;  } |

#### 2.4.5.异常处理

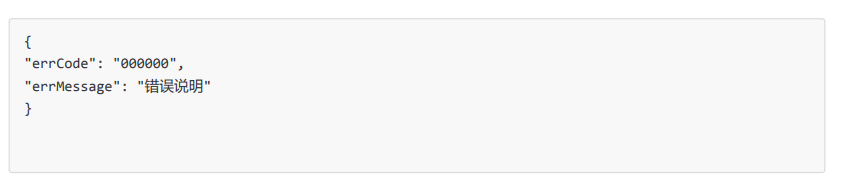
##### 2.4.5.1.异常信息格式

系统在交互中难免会有异常发生，前端为了解析异常信息向用户提示特定义了异常信息的返回格式，如下：

1. 返回response状态说明



1. 自定义错误体



##### 2.4.5.2.异常处理流程

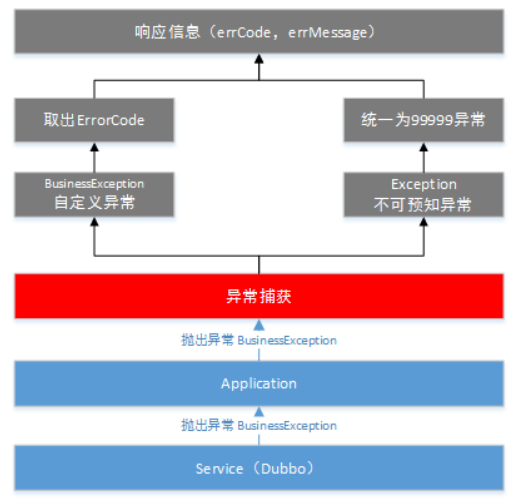
截至目前系统并没有按照前端要求返回异常信息，测试如下：

注册商户时输入一个错误的验证码，返回信息如下：



从上边的返回信息得知，状态码为500符合要求，按前端的规范定义的错误信息要写在“errMessage” 中，显然不符合要求。

系统规范了异常处理流程，如下：



1.在服务层抛出自定义异常类型及不可预知异常类型。

上图中BusinessException为系统的自定义异常类型，程序中在代码显示抛出该异常，此类异常是程序员可预知的。

另一部分是系统无法预知的异常，如：数据库无法连接，服务器宕机等场景下所抛出的异常，此类异常是程序员无法预知的异常。

2.应用层接收到服务层抛出异常继续向上抛出，应用层自己也可以抛出自定义异常类型及不可预知异常类型。

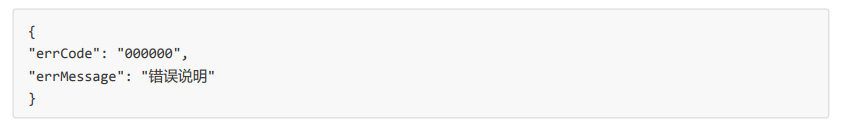
3.统一异常处理器捕获到异常进行解析。

判断如果为自定义异常则直接取出错误代码及错误信息，因为程序员在抛出自定义异常时已将错误代码和异常信息指定。

如果为不可预知的异常则统一定义为99999异常代码。

4.统一异常处理器将异常信息格式为前端要求的格式响应给前端。

服务端统一将异常信息封装在下边的Json格式中返回：



##### 2.4.5.3.自定义业务异常类

1.在yimupay-common工程的com.me.yimu.pay.domain包下添加业务异常类BusinessException：

|  |
| --- |
| **package** com.me.yimu.pay.domain;  /\*\*  \* 自定义的异常类型  \* **@author** Administrator  \* **@version** 1.0  \*\*/  **public** **class** BusinessException **extends** RuntimeException {  **private** ErrorCode errorCode;  **public** BusinessException(ErrorCode errorCode) {  **super**();  **this**.errorCode = errorCode;  }  **public** BusinessException() {  **super**();  }  **public** **void** setErrorCode(ErrorCode errorCode) {  **this**.errorCode = errorCode;  }  **public** ErrorCode getErrorCode() {  **return** errorCode;  }  } |

2. 定义错误代码

在common工程专门定义了ErrorCode接口及CommonErrorCode通用代码。

|  |
| --- |
| **package** com.me.yimu.pay.domain;  /\*\*  \* 异常编码  \*/  **public** **enum** CommonErrorCode **implements** ErrorCode {  ////////////////////////////////////公用异常编码 //////////////////////////  ***E\_100101***(100101,"传入参数与接口不匹配"),  ***E\_100102***(100102,"验证码错误"),  ***E\_100103***(100103,"验证码为空"),  ***E\_100104***(100104,"查询结果为空"),  ***E\_100105***(100105,"ID格式不正确或超出Long存储范围"),  ***E\_100106***(100106,"上传错误"),  ***E\_100107***(100107,"发送验证码错误"),  ***E\_100108***(100108,"传入对象为空"),  ***E\_100109***(100109,"手机号格式不正确"),  ***E\_100110***(100110,"用户名为空"),  ***E\_100111***(100111,"密码为空"),  ***E\_100112***(100112,"手机号为空"),  ***E\_100113***(100113,"手机号已存在"),  ***E\_100114***(100114,"用户名已存在"),  ***E\_100115***(100115,"密码不正确"),  ////////////////////////////////////SAAS服务异常编码110 //////////////////////////  ***E\_110001***(110001,"账号不存在"),  ***E\_110002***(110002,"角色编码在同一租户中已存在，不可重复"),  ***E\_110003***(110003,"角色为空"),  ***E\_110004***(110004,"角色已绑定账号，被使用中不可删除"),  ***E\_110005***(110005,"权限集合为空"),  ***E\_110006***(110006,"参数为空"),  ***E\_110007***(110007,"未查询到租户关联的角色"),  ***E\_110008***(110008,"账号被其他租户使用，不可删除"),  ////////////////////////////////////商户服务异常编码200//////////////////////////  ***E\_200001***(200001,"企业名称不能为空"),  ***E\_200002***(200002,"商户不存在"),  ***E\_200003***(200003,"商户还未通过认证审核，不能创建应用"),  ***E\_200004***(200004,"应用名称已经存在，请使用其他名称"),  ***E\_200005***(200005,"应用不属于当前商户"),  ***E\_200006***(200006,"门店不属于当前商户"),  ***E\_200007***(200007,"二维码生成失败"),  ***E\_200008***(200008,"授权码为空"),  ***E\_200009***(200009,"订单标题为空"),  ***E\_200010***(200010,"订单金额为空"),  ***E\_200011***(200011,"授权码格式有误"),  ***E\_200012***(200012,"租户不存在"),  ***E\_200013***(200013,"员工不存在"),  ***E\_200014***(200014,"商户下未设置根门店"),  ***E\_200015***(200015,"未查询到该门店"),  ***E\_200016***(200016,"资质申请已通过，无需重复申请"),  ***E\_200017***(200017,"商户在当前租户下已经注册，不可重复注册"),  ////////////////////////////////////交易服务异常编码300//////////////////////////  ***E\_300001***(300001,"支付金额为空"),  ***E\_300002***(300002,"openId为空"),  ***E\_300003***(300003,"appId为空"),  ***E\_300004***(300004,"商户id为空"),  ***E\_300005***(300005,"服务类型编码为空"),  ***E\_300006***(300006,"订单金额转换异常"),  ***E\_300007***(300007,"原始支付渠道为空"),  ***E\_300008***(300008,"已存在相同的支付参数，不可重复配置"),  ***E\_300009***(300009,"传入对象为空或者缺少必要的参数"),  ***E\_300110***(300110,"交易单号不能为空"),  ////////////////////////////////////支付渠道代理服务异常编码400//////////////////  ***E\_400001***(400001,"微信确认支付失败"),  ////////////////////////////////////运营服务异常编码500//////////////////  ////////////////////////////////////特殊异常编码/////////////////////////////////////  ***E\_999991***(999991,"调用微服务-授权服务 被熔断"),  ***E\_999992***(999992,"调用微服务-用户服务 被熔断"),  ***E\_999993***(999993,"调用微服务-资源服务 被熔断"),  ***E\_999994***(999994,"调用微服务-同步服务 被熔断"),  ***E\_999910***(999910,"调用微服务-没有传tenantId租户Id"),  ***E\_999911***(999911,"调用微服务-没有json-token令牌"),  ***E\_999912***(999912,"调用微服务-json-token令牌解析有误"),  ***E\_999913***(999913,"调用微服务-json-token令牌有误-没有当前租户信息"),  ***E\_999914***(999914,"调用微服务-json-token令牌有误-该租户下没有权限信息"),  ***E\_NO\_AUTHORITY***(999997,"没有访问权限"),  ***CUSTOM***(999998,"自定义异常"),  /\*\*  \* 未知错误  \*/  ***UNKNOWN***(999999,"未知错误");  **private** **int** code;  **private** String desc;  **public** **int** getCode() {  **return** code;  }  **public** String getDesc() {  **return** desc;  }  **private** CommonErrorCode(**int** code, String desc) {  **this**.code = code;  **this**.desc = desc;  }  **public** **static** CommonErrorCode setErrorCode(**int** code) {  **for** (CommonErrorCode errorCode : CommonErrorCode.*values*()) {  **if** (errorCode.getCode()==code) {  **return** errorCode;  }  }  **return** **null**;  }  } |

##### 2.4.5.4.自定义业务异常处理器

1.在yimupay‐common工程的com.me.yimu.pay.domain包下添加错误响应包装类RestErrorResponse：

|  |
| --- |
| package com.me.yimu.pay.domain;  import io.swagger.annotations.ApiModel;  import lombok.Data;  /\*\*  \* @author Administrator  \* @version 1.0  \*\*/  @ApiModel(value = "RestErrorResponse", description = "错误响应参数包装")  @Data  public class RestErrorResponse {  private String errCode;  private String errMessage;  public RestErrorResponse(String errCode,String errMessage){  this.errCode = errCode;  this.errMessage= errMessage;  }  } |

2. 定义全局异常处理器

全局异常处理器使用ControllerAdvice注解实现，ControllerAdvice是SpringMVC3.2提供的注解，用 ControllerAdvice可以方便实现对Controller面向切面编程，具体用法如下：

1、ControllerAdvice和ExceptionHandler注解实现全局异常处理

2、ControllerAdvice和ModelAttribute注解实现全局数据绑定

3、ControllerAdvice生InitBinder注解实现全局数据预处理

ControllerAdvice和ExceptionHandler结合可以捕获Controller抛出的异常，根据异常处理流程，Service和持久层 最终都会抛给Controller，所以此方案可以实现全局异常捕获，异常被捕获到即可格式为前端要的信息格式响应给前端。

在yimupay‐merchant‐application工程的com.shanjupay.merchant.common.intercept添加 GlobalExceptionHandler：

|  |
| --- |
| **package** com.me.yimu.pay.merchant.intercept;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **import** org.springframework.http.HttpStatus;  **import** org.springframework.web.bind.annotation.ControllerAdvice;  **import** org.springframework.web.bind.annotation.ExceptionHandler;  **import** org.springframework.web.bind.annotation.ResponseBody;  **import** org.springframework.web.bind.annotation.ResponseStatus;  **import** com.me.yimu.pay.domain.BusinessException;  **import** com.me.yimu.pay.domain.CommonErrorCode;  **import** com.me.yimu.pay.domain.ErrorCode;  **import** com.me.yimu.pay.domain.RestErrorResponse;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  /\*\*  \* 全局异常处理器  \* **@author** Administrator  \* **@version** 1.0  \*\*/  @ControllerAdvice//与@Exceptionhandler配合使用实现全局异常处理  **public** **class** GlobalExceptionHandler {  **private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(GlobalExceptionHandler.**class**);  //捕获Exception异常  @ExceptionHandler(value = Exception.**class**)  @ResponseBody  @ResponseStatus(HttpStatus.***INTERNAL\_SERVER\_ERROR***)  **public** RestErrorResponse processExcetion( HttpServletRequest request,  HttpServletResponse response,  Exception e){  //解析异常信息  //如果是系统自定义异常，直接取出errCode和errMessage  **if**(e **instanceof** BusinessException){  ***LOGGER***.info(e.getMessage(),e);  //解析系统自定义异常信息  BusinessException businessException= (BusinessException) e;  ErrorCode errorCode = businessException.getErrorCode();  //错误代码  **int** code = errorCode.getCode();  //错误信息  String desc = errorCode.getDesc();  **return** **new** RestErrorResponse(String.*valueOf*(code),desc);  }  ***LOGGER***.error("系统异常：",e);  //统一定义为99999系统未知错误  **return** **new** RestErrorResponse(String.*valueOf*(CommonErrorCode.***UNKNOWN***.getCode()),CommonErrorCode.***UNKNOWN***.getDesc());  }  } |

##### 2.4.5.5.抛出自定义异常

按照异常处理流程，应用层抛出自定义异常由异常处理器进行解析。

1、校验验证码接口抛出BusinessException 修改商户平台应用工程中SmsService的verificationMessageCode接口

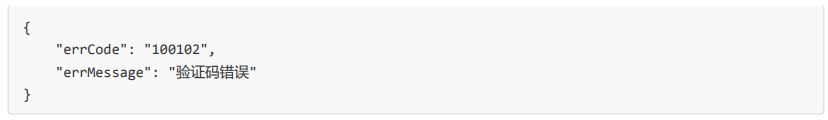
|  |
| --- |
| /\*\*  \* 校验手机验证码  \* **@param** verifiyKey 验证码的key  \* **@param** verifiyCode 验证码  \*/  **void** checkVerifiyCode(String verifiyKey,String verifiyCode) throws BusinessException; |

接口实现中抛出异常自定义异常类型:

|  |
| --- |
| /\*\*  \* 校验手机验证码  \*  \* **@param** verifiyKey 验证码的key  \* **@param** verifiyCode 验证码  \*/  @Override  **public** **void** checkVerifiyCode(String verifiyKey, String verifiyCode) **throws** **BusinessException** {  //String url = "http://localhost:56085/sailing/verify?name=sms&verificationCode="+verifiyCode+"&verificationKey="+verifiyKey;  //校验验证码的url  String smsUrl = url + "/verify?name=sms&verificationCode="+verifiyCode+"&verificationKey="+verifiyKey;    Map bodyMap = **null**;  **try** {  //使用restTemplate请求验证码服务  ResponseEntity<Map> exchange = restTemplate.exchange(smsUrl, HttpMethod.***POST***, HttpEntity.***EMPTY***, Map.**class**);  ***log***.info("请求验证码服务，得到响应:{}", JSON.*toJSONString*(exchange));  bodyMap = exchange.getBody();  }**catch** (Exception e){  e.printStackTrace();  ***log***.info(e.getMessage(), e);  **throw** **new** BusinessException(CommonErrorCode.***E\_100102***);  }  **if**(bodyMap == **null** || bodyMap.get("result") == **null** || !(Boolean) bodyMap.get("result")){  **throw** **new** BusinessException(CommonErrorCode.***E\_100102***);  }  } |

2.测试

请求商户注册，输出一个错误的验证码，返回信息如下：

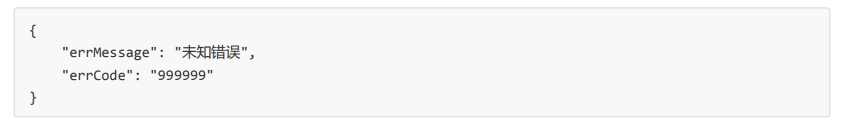


3.测试不可预知异常

故意在Controller中制造异常，测试是否抛出未知错误异常。

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  **int i=1/0;**  // 校验验证码  smsService.checkVerifiyCode(merchantRegister.getVerifiykey(),  merchantRegister.getVerifiyCode());    // 注册商户  MerchantDTO merchantDTO =  MerchantRegisterConvert.***INSTANCE***.vo2dto(merchantRegister);  merchantService.createMerchant(merchantDTO);  **return** merchantRegister;  } |

请商户注册，返回信息如下：



#### 2.4.6.校验商户手机号

##### 2.4.6.1.实现思路

校验商户手机号的唯一性，根据商户的手机号查询商户表，如果存在记录则说明已有相同的手机号重复，手机号不唯一则抛出异常自定义异常。

##### 2.4.6.2.完善代码

1）修改商户服务注册商户接口，添加抛出异常声明

|  |
| --- |
| /\*\*  \* 商户注册  \* **@param** merchantDTO  \* **@return**  \*/  MerchantDTO createMerchant(MerchantDTO merchantDTO) **throws** BusinessException; |

2）修改商户服务注册商户接口实现方法

|  |
| --- |
| /\*\*  \* 商户注册  \* **@param** merchantDTO  \* **@return**  \*/  @Override  @Transactional  **public** MerchantDTO createMerchant(MerchantDTO merchantDTO) **throws** BusinessException {  // DTO为空的校验  **if**(merchantDTO == **null**) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100108***);  }    // 手机号非空校验  **if**(StringUtils.*isBlank*(merchantDTO.getMobile())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100112***);  }    //校验手机号的合法性  **if** (!PhoneUtil.*isMatches*(merchantDTO.getMobile())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100109***);  }    //联系人非空校验  **if** (StringUtils.*isBlank*(merchantDTO.getUsername())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100110***);  }    //密码非空校验  **if** (StringUtils.*isBlank*(merchantDTO.getPassword())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100111***);  }    //校验商户手机号的唯一性,根据商户的手机号查询商户表，如果存在记录则说明已有相同的手机号重复  LambdaQueryWrapper<Merchant> lambdaQueryWrapper =  **new** LambdaQueryWrapper<Merchant>().eq(  Merchant::getMobile, merchantDTO.getMobile());  Integer count = merchantMapper.selectCount(lambdaQueryWrapper);  **if**(count > 0) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100113***);  }    // 将DTO转换为Entity  Merchant merchantEntity = MerchantConvert.***INSTANCE***.dto2entity(merchantDTO);    // 设置审核状态0‐未申请,1‐已申请待审核,2‐审核通过,3‐审核拒绝  merchantEntity.setAuditStatus("0");    // 保存商户信息  merchantMapper.insert(merchantEntity);    //将entity转成 dto  MerchantDTO merchantDTONew = MerchantConvert.***INSTANCE***.entity2dto(merchantEntity);    **return** merchantDTONew;  } |

3）修改商户应用平台注册商户接口

添加对注册信息的非空校验。

|  |
| --- |
| @ApiOperation("注册商户")  @ApiImplicitParam(name = "merchantRegister", value = "注册信息",  required = **true**, dataType = "MerchantRegisterVO",  paramType = "body")  @PostMapping("/merchants/register")  **public** MerchantRegisterVO registerMerchant(  @RequestBody MerchantRegisterVO merchantRegister) {  // 1.校验  **if** (merchantRegister == **null**) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100108***);  }  //手机号非空校验  **if** (StringUtils.*isBlank*(merchantRegister.getMobile())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100112***);  }  //校验手机号的合法性  **if** (!PhoneUtil.*isMatches*(merchantRegister.getMobile())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100109***);  }    //联系人非空校验  **if** (StringUtils.*isBlank*(merchantRegister.getUsername())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100110***);  }  //密码非空校验  **if** (StringUtils.*isBlank*(merchantRegister.getPassword())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100111***);  }  //验证码非空校验  **if** (StringUtils.*isBlank*(merchantRegister.getVerifiyCode()) ||  StringUtils.*isBlank*(merchantRegister.getVerifiykey())) {  **throw** **new** BusinessException(CommonErrorCode.***E\_100103***);  }    // 校验验证码  smsService.checkVerifiyCode(merchantRegister.getVerifiykey(),  merchantRegister.getVerifiyCode());    // 注册商户  MerchantDTO merchantDTO =  MerchantRegisterConvert.***INSTANCE***.vo2dto(merchantRegister);  merchantService.createMerchant(merchantDTO);  **return** merchantRegister;  } |

## 3.参照

<https://blog.csdn.net/vvoennvv/article/details/128450103>

https://zhidao.baidu.com/question/560334435782643812.html