# 天猫前台系统后端开发手册

V 1.0

2Executioner

home：首页

主要业务

**展示所有分类及其分类产品8\*8的小标题首段（需要64个产品）**

**每个分类展示五个热销产品**

**把查询结果存入redis便于数据二次使用**

控制层

CategoryInfoApi

url: /api/category/categoryInfo

ProductInfoApi

url: /api/product/productInfo

显示分类列表和热销商品

**前端传入参数：**无

**请求方式：**get

**请求url：**/list

**后端方法：**public Result listCategoryInfo()

**后端返回数据参数：**List<CategoryInfo> （首页分类菜单，包括分类名，分类id，8\*8小标题首段，和小标题首段对应的商品id）和 热销商品

CategoryInfoApi

// 首页显示分类列表和热销商品  
@ApiOperation(value = "首页显示分类列表和热销商品")  
@GetMapping("/list")  
**public** Result listCategoryInfo() {  
 List<CategoryInfo> list = categoryInfoService.homeListCategoryInfo();  
 **return** Result.*ok*(list);  
}

CategoryInfoService

// 首页显示分类列表和热销商品  
List<CategoryInfo> homeListCategoryInfo();

CategoryInfoServiceImpl

*/\*\*  
 \* 首页显示分类列表和热销商品  
 \*/*@Override  
**public** List<CategoryInfo> homeListCategoryInfo() {  
 // 查询出所有分类  
 List<CategoryInfo> categoryInfos = baseMapper.selectList(**null**);  
  
 **if** (categoryInfos == **null**) {  
 **return null**;  
 }  
  
 // 为了减少后面的远程调用次数，把所有分类的id封装到一个list集合中  
 List<String> idList = **new** ArrayList<>();  
 categoryInfos.stream().forEach(item -> {  
 idList.add(item.getId());  
 });  
  
 // 根据分类分别按销量查询出热销的前5个商品信息，包含第一张缩略图url（远程调用）  
 Map<String, List<ProductInfo>> hotMap = productInfoFeignClient.listProductInfoHot(idList);  
  
 // 根据分类分别按销量排序查询出前64个商品的小标题和id（远程调用）  
 Map<String, List<ProductInfo>> productInfoSubTitleList = productInfoFeignClient.listProductInfoSubTitle(idList);  
  
 // 封装数据  
 **for** (**int** x = 0; x < idList.size(); x++) {  
 String id = idList.get(x);  
  
 // 热销产品  
 List<ProductInfo> hotList = hotMap.get(id);  
 categoryInfos.get(x).getParams().put("hotList", hotList);  
  
 // 把这64个商品的小标题做包装，以 8\*8 的方式存储  
 List<ProductInfo> productInfoList = productInfoSubTitleList.get(id);  
 // 如果查询出来 productInfoList 为空则说明该分类没有商品，那么就跳过则此循环  
 **if** (productInfoList == **null**) {  
 **continue**;  
 }  
 // 浮动菜单  
 List<List<Map<String, String>>> floatMenu = **new** ArrayList<>();  
 List<Map<String, String>> rows = **null**; // 行  
 **for** (**int** i = 0; i < productInfoList.size(); i++) {  
 ProductInfo productInfo = productInfoList.get(i);  
 **if** (i % 8 == 0) { // 一行8个，循环到8个就new一个新的行  
 rows = **new** ArrayList<>(); // 新的行  
 floatMenu.add(rows);  
 }  
 String pid = productInfo.getId(); // productId  
 String subTitle = productInfo.getSubTitle(); // 小标题  
 // 把小标题进行拆分，取首段  
 **if** (StringUtils.*isEmpty*(subTitle)) { // 如果为空则直接置为空字符串  
 subTitle = "";  
 } **else** {  
 subTitle = subTitle.split(" ")[0]; // 取第一段  
 }  
 // 存入rows的map中  
 Map<String, String> item = **new** HashMap<>();  
 item.put("id", pid);  
 item.put("subTitle", subTitle);  
 rows.add(item);  
 }  
 // 封装到 categoryInfos  
 categoryInfos.get(x).getParams().put("floatMenu", floatMenu);  
 }  
  
 **return** categoryInfos;  
}

ProductInfoFeignClient（service\_product\_client模块）

// 返回商品小标题集合（内部调用）  
@PostMapping("/api/product/productInfo/inner/list/simple")  
Map<String, List<ProductInfo>> listProductInfoSubTitle(@RequestBody List<String> idList);  
  
// 显示每个分类前五个热销商品（内部调用）  
@PostMapping("/api/product/productInfo/inner/list/hot")  
Map<String, List<ProductInfo>> listProductInfoHot(@RequestBody List<String> idList);

ProductInfoApi（service-product模块）

// 返回商品小标题集合（内部调用）  
@ApiOperation(value = "内部调用，返回商品小标题集合")  
@PostMapping("/inner/list/simple")  
**public** Map<String, List<ProductInfo>> listProductInfoSubTitle(  
 @ApiParam(name = "idList", value = "分类id集合", required = **true**)  
 @RequestBody List<String> idList) {  
  
 **return** productInfoService.listProductInfoSubTitle(idList);  
}  
  
// 显示每个分类前五个热销商品（内部调用）  
@ApiOperation(value = "显示每个分类前五个热销商品（内部调用）")  
@PostMapping("/inner/list/hot")  
**public** Map<String, List<ProductInfo>> listProductInfoHot(  
 @ApiParam(name = "idList", value = "分类id集合", required = **true**)  
 @RequestBody List<String> idList) {  
  
 **return** productInfoService.listProductInfoHot(idList);  
}

ProductInfoService

// 返回商品小标题集合 内部调用  
Map<String, List<ProductInfo>> listProductInfoSubTitle(List<String> idList);  
  
// 显示每个分类前五个热销商品（内部调用）  
Map<String, List<ProductInfo>> listProductInfoHot(List<String> idList);

ProductInfoServiceImpl

*/\*\*  
 \* 返回商品小标题集合 内部调用  
 \** ***@param*** *idList  
 \** ***@return*** *\*/*@Override  
**public** Map<String, List<ProductInfo>> listProductInfoSubTitle(List<String> idList) {  
 List<ProductInfo> productInfoList = baseMapper.listProductInfoSubTitle(idList);  
 **if** (productInfoList == **null**) {  
 **return null**;  
 }  
 // 对查询出来的结果按categoryId进行分组  
 Map<String, List<ProductInfo>> collect = productInfoList.stream().collect(Collectors.*groupingBy*(ProductInfo::getCategoryId));  
 **return** collect;  
}  
  
*/\*\*  
 \* 显示每个分类前五个热销商品（内部调用）  
 \** ***@param*** *idList  
 \** ***@return*** *\*/*@Override  
**public** Map<String, List<ProductInfo>> listProductInfoHot(List<String> idList) {  
 // 查询五条记录  
 Page<ProductInfo> page = **new** Page<>(1, 5);  
 QueryWrapper<ProductInfo> wrapper = **new** QueryWrapper<>();  
  
 // 放入到map集合中，分类id作为key，查询结果的Records作为value  
 Map<String, List<ProductInfo>> map = **new** HashMap<>();  
 **for** (String str : idList) {  
 // 按月销量进行降序排序  
 wrapper.eq("category\_id", str);  
 wrapper.orderByDesc("monthly\_sales");  
 IPage<ProductInfo> pageModule = baseMapper.selectPage(page, wrapper);  
 wrapper.clear(); // 清空条件  
 // 对商品数据再进行处理  
 pageModule.getRecords().stream().forEach(item -> {  
 // 把第一张缩略图封装进去  
 **this**.packImage(item);  
 // 把每个商品的名称缩短，方便前端显示（显示商品名称的前25个字符）  
 item.setName(item.getName().substring(0, 25));  
 });  
 map.put(str, pageModule.getRecords());  
 }  
 **return** map;  
}

category：分类页

主要业务

**展示分类图片**

**分页展示分类下所有产品**

**可按价格进行查询**

**可按条件进行排序**

控制层

CategoryInfoApi

url: /api/category/categoryInfo

ProductInfoApi

url: /api/product/productInfo

显示分类图片

**前端传入参数：**categoryId（分类id）

**请求方式：**get

**请求url：**/{id}

**后端方法：**public Result getCategoryInfo()

**后端返回数据参数：** CategoryInfo (分类信息)

CategoryInfoApi

// 根据id查询出分类  
@ApiOperation(value = "根据id查询出分类")  
@GetMapping("/{id}")  
**public** Result getCategoryInfo(  
 @ApiParam(value = "id", name = "分类id", required = **true**)  
 @PathVariable String id) {  
 CategoryInfo categoryInfo = categoryInfoService.getById(id);  
 **return** Result.*ok*(categoryInfo);  
}

条件查询商品

**前端传入参数：**current（起始页），limit（每页大小），searchObj（查询对象，包含分类id）

**请求方式：**get

**请求url：**/list/{current}/{limit}

**后端方法：**public Result listProductInfo(@PathVariable Long current, @PathVariable Long limit, @RequestBody ProductInfoFrontQueryVo productInfoFrontQueryVo)

**后端返回数据参数：**IPage<ProductInfo> （商品信息List集合）

ProductInfoApi

// 分类页，根据分类id和查询条件显示商品  
@ApiOperation(value = "分类页，根据分类id和查询条件显示商品")  
@GetMapping("/list/{current}/{limit}")  
**public** Result listProductInfo(  
 @ApiParam(name = "current", value = "起始页", required = **true**)  
 @PathVariable Long current,  
  
 @ApiParam(name = "limit", value = "每页大小", required = **true**)  
 @PathVariable Long limit,  
  
 @ApiParam(name = "productInfoFrontQueryVo", value = "前台分类页商品查询对象", required = **true**)  
 ProductInfoFrontQueryVo productInfoFrontQueryVo) {  
  
 Page<ProductInfo> page = **new** Page<>(current, limit);  
 IPage<ProductInfo> pageModule = productInfoService.listProductInfo(page, productInfoFrontQueryVo);  
 **return** Result.*ok*(pageModule);  
}

ProductInfoService

// 分类页，根据分类id和查询条件显示商品  
IPage<ProductInfo> listProductInfo(Page<ProductInfo> page, ProductInfoFrontQueryVo productInfoFrontQueryVo);

ProductInfoServiceImpl

*/\*\*  
 \* 分类页，根据分类id和查询条件显示商品  
 \** ***@param*** *page  
 \** ***@param*** *productInfoFrontQueryVo  
 \** ***@return*** *\*/*@Override  
**public** IPage<ProductInfo> listProductInfo(Page<ProductInfo> page, ProductInfoFrontQueryVo productInfoFrontQueryVo) {  
 String categoryId = productInfoFrontQueryVo.getCategoryId(); // 分类id  
 BigDecimal lowPrice = productInfoFrontQueryVo.getLowPrice(); // 最低价格  
 BigDecimal highPrice = productInfoFrontQueryVo.getHighPrice(); // 最高价格  
 Integer sortField = productInfoFrontQueryVo.getSortField(); // 排序字段  
 Integer lostSortField = productInfoFrontQueryVo.getLostSortField(); // 上一次排序字段  
 Integer sortType = productInfoFrontQueryVo.getSortType(); // 排序方式  
  
 String typeBySort = SortTypeEnum.*getTypeBySort*(sortType); // 取得排序方式的字符  
 String sortFieldStr = ProductSortStatusEnum.*getFieldByNumber*(sortField); // 取得本次排序字段  
 String lostSortFieldStr = ProductSortStatusEnum.*getFieldByNumber*(lostSortField); // 取得上次排序字段  
  
 QueryWrapper<ProductInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("category\_id", categoryId);  
 // 设置价格区间  
 **if** (lowPrice != **null**) {  
 wrapper.ge("promote\_price", lowPrice);  
 }  
 **if** (highPrice != **null**) {  
 wrapper.le("promote\_price", highPrice);  
 }  
 // 定义排序条件  
 **if** (!StringUtils.*isEmpty*(sortFieldStr) && // 首要条件是本次排序字段不为空 并且 不是默认排序  
 sortField != ProductSortStatusEnum.*DEFAULT*.getNumber()) {  
  
 **if** (SortTypeEnum.*ASC*.getType().equals(typeBySort)) { // 是升序排序  
 wrapper.orderByAsc(sortFieldStr);  
 } **else** { // 否则就是降序排序  
 wrapper.orderByDesc(sortFieldStr);  
 }  
 } **else if**(!StringUtils.*isEmpty*(lostSortFieldStr) && // 如果本次排序条件为空，上次排序条件不为空也不是默认排序，则按上次排序条件排序  
 lostSortField != ProductSortStatusEnum.*DEFAULT*.getNumber()) {  
  
 **if** (SortTypeEnum.*ASC*.getType().equals(typeBySort)) { // 是升序排序  
 wrapper.orderByAsc(lostSortFieldStr);  
 } **else** { // 否则就是降序排序  
 wrapper.orderByDesc(lostSortFieldStr);  
 }  
 }  
 IPage<ProductInfo> productInfoIPage = baseMapper.selectProductInfoPageOrderBy(page, wrapper);  
  
 // 对商品数据再进行处理  
 productInfoIPage.getRecords().stream().forEach(item -> {  
 // 封装商品参数  
 **this**.packParams(item);  
 });  
 **return** productInfoIPage;  
}  
  
// 封装商品参数  
**private void** packParams(ProductInfo item) {  
 // 把第一张缩略图封装进去  
 **this**.packImage(item);  
 // 把每个商品的名称缩短，方便前端显示（显示商品名称的前25个字符）  
 item.setName(item.getName().substring(0, 25));  
 // 评价数量  
 QueryWrapper<Review> queryWrapper = **new** QueryWrapper<>();  
 queryWrapper.eq("product\_id", item.getId());  
 **int** count = reviewService.count(queryWrapper);  
 item.getParams().put("reviewNumber", count);  
}

ProductInfoMapper

// 查询出所有商品以及按指定方式排序，因为要查询出评价数所以自己写多表查询语句  
IPage<ProductInfo> selectProductInfoPageOrderBy(IPage<ProductInfo> page, @Param(Constants.*WRAPPER*) Wrapper<ProductInfo> queryWrapper);

ProductInfoMapper.xml

<**select** id="selectProductInfoPageOrderBy" resultMap="productInfoBean">  
 SELECT  
 id,  
 category\_id,  
 name,  
 sub\_title,  
 orignal\_price,  
 promote\_price,  
 stock,  
 sales\_volume,  
 monthly\_sales,  
 create\_time,  
 update\_time,  
 is\_deleted  
 FROM  
 product\_info p  
 LEFT JOIN (  
 SELECT  
 product\_id,  
 count(\*) AS reviews  
 FROM  
 review  
 GROUP BY  
 product\_id  
 ) r ON r.product\_id = p.id  
 ${ew.customSqlSegment}  
</**select**>

productInfo：产品页

主要业务

**展示分类图片**

**展示产品图片**

**展示分类属性**

**展示商品属性值**

**展示评论以及评论的用户名（匿名，做分页，后续升级为懒加载）**

**加入购物车**

**立即购买**

**模态框用户登录**

控制层

CategoryInfoApi

url: /api/category/categoryInfo

ProductInfoApi

url: /api/product/productInfo

ReviewApi

url:/api/product/review

显示分类图片

**前端传入参数：**categoryId（分类id）

**请求方式：**get

**请求url：**/{id}

**后端方法：**public Result getCategoryInfo()

**后端返回数据参数：** CategoryInfo (分类信息)

参考分类页中的方法，此处直接调用的分类页写了的api

查询商品信息

**前端传入参数：**productId（商品id）

**请求方式：**get

**请求url：**/{id}

**后端方法：**public Result getProductInfoById(@PathVariable String id)

**后端返回数据参数：** ProductInfo (商品信息)

ProductInfoApi

// 根据商品id查询出商品信息  
@ApiOperation(value = "根据商品id查询出商品信息")  
@GetMapping("/{id}")  
**public** Result getProductInfoById(  
 @ApiParam(name = "id", value = "商品id", required = **true**)  
 @PathVariable String id) {  
 ProductInfo productInfo = productInfoService.getProductInfoById(id);  
 **return** Result.*ok*(productInfo);  
}

ProductInfoService

// 根据商品id查询出商品信息  
ProductInfo getProductInfoById(String id);

ProductInfoServiceImpl

*/\*\*  
 \* 根据商品id查询出商品信息  
 \** ***@param*** *id  
 \** ***@return*** *\*/*@Override  
**public** ProductInfo getProductInfoById(String id) {  
 ProductInfo productInfo = baseMapper.selectById(id);  
 **if** (productInfo == **null**) { // 如果查询出来为空，则说明传入的商品id不正确  
 **throw new** TmallException(ResultCodeEnum.*PARAM\_ERROR*);  
 }  
 String pid = productInfo.getId();  
  
 // 查询评价数  
 Integer countReview = reviewService.count(**new** QueryWrapper<Review>().eq("product\_id", pid));  
  
 // 根据商品id查询商品属性值和分类属性（在后台管理系统中已经实现，这里直接用）  
 List<PropertyAndValueVo> propertyAndValueVoList = propertyValueService.show(pid);  
  
 // 根据商品id查询出商品图片  
 Map<String, List<ProductImageReturnVo>> productImage = productImageService.showByProductId(pid);  
  
 productInfo.getParams().put("countReview", countReview);  
 productInfo.getParams().put("propertyAndValueVoList", propertyAndValueVoList);  
 productInfo.getParams().put("productImage", productImage);  
  
 **return** productInfo;  
}

分页查询用户对商品的评价

**前端传入参数：**current，limit，productId（商品id）

**请求方式：**get

**请求url：**/{id}

**后端方法：**public Result listReviewByProductId(@PathVariable Long current, @PathVariable Long limit, @PathVariable String productId)

**后端返回数据参数：** IPage<ProductReviewReturnVo> (商品评价返回vo集合)

ReviewApi

// 根据商品id分页查询商品评价  
@ApiOperation(value = "根据商品id分页查询商品评价")  
@GetMapping("/list/{current}/{limit}/{productId}")  
**public** Result listReviewByProductId(  
 @ApiParam(name = "current", value = "起始页", required = **true**)  
 @PathVariable Long current,  
  
 @ApiParam(name = "limit", value = "每页大小", required = **true**)  
 @PathVariable Long limit,  
  
 @ApiParam(name = "productId", value = "商品id", required = **true**)  
 @PathVariable String productId) {  
  
 Page<ProductReviewReturnVo> page = **new** Page<>(current, limit);  
 IPage pageModule = reviewService.listReviewByProductId(page, productId);  
 **return** Result.*ok*(pageModule);  
}

ReviewService

// 根据商品id分页查询商品评价  
IPage listReviewByProductId(Page<ProductReviewReturnVo> page, String productId);

ReviewServiceImpl

*/\*\*  
 \* 根据商品id分页查询商品评价  
 \** ***@param*** *page  
 \** ***@param*** *productId  
 \** ***@return*** *\*/*@Override  
@Cacheable(value = "product", keyGenerator = "keyGeneratorPage") // 缓存到redis  
**public** IPage listReviewByProductId(Page<ProductReviewReturnVo> page, String productId) {  
 // 分页查询  
 IPage<ProductReviewReturnVo> reviewPage = baseMapper.selectProductReviewByProductId(page, productId);  
 // idList，用户id集合，降低远程调用次数  
 List<String> idList = **new** ArrayList<>();  
 reviewPage.getRecords().forEach(item -> {  
 ProductReviewReturnVo vo = **new** ProductReviewReturnVo();  
 BeanUtils.*copyProperties*(item, vo);  
 idList.add(item.getUserId());  
 });  
  
 // 远程调用查询用户信息  
 List<UserInfo> userInfoList = userInfoFeignClient.listUserInfoOfInner(idList);  
 reviewPage.getRecords().forEach(item -> {  
 // *TODO 后续给用户提供匿名可选，不匿名则显示全名称* // 封装匿名用户名称  
 **this**.packUserName(item, userInfoList);  
 });  
  
 **return** reviewPage;  
}  
  
// 封装匿名用户名称  
**private void** packUserName(ProductReviewReturnVo item, List<UserInfo> userInfoList) {  
 **for** (**int** i = 0; i < userInfoList.size(); i++) {  
 UserInfo userInfo = userInfoList.get(i);  
 **if** (item.getUserId().equals(userInfo.getId())) {  
 // 取得用户名，nick\_name > openid > email  
 StringBuilder name = **new** StringBuilder();  
 **if** (!StringUtils.*isEmpty*(userInfo.getNickName())) { // 昵称  
 name.append(userInfo.getNickName());  
 } **else if**(!StringUtils.*isEmpty*(userInfo.getOpenid())) { // 微信名  
 name.append(userInfo.getOpenid());  
 } **else if**(!StringUtils.*isEmpty*(userInfo.getEmail())) { // 邮箱号  
 name.append(userInfo.getEmail());  
 }  
 // 把用户名处理为星星  
 // 例如：张（\*） 张三（张\*） 张小三（张\*三） 张小小三（张\*\*三）  
 **if** (name.length() <= 1) {  
 name.delete(0, 1);  
 name.append("\*");  
 } **else if**(name.length() <= 2) {  
 name.delete(1, 2);  
 name.append("\*");  
 } **else** {  
 Integer length = name.length() - 2;  
 **for** (**int** j = 1; j <= length; j++) {  
 name.replace(j, j + 1, "\*");  
 }  
 }  
 item.setAnonymity(name.toString());  
 **break**;  
 }  
 }  
}

search：搜索页

主要业务

**在搜索框搜索商品关键字返回符合条件的商品（后续用elastic search实现）**

控制层

ProductInfoApi

url: /api/product/productInfo

搜索商品

**前端传入参数：**current，limit，keyword（商品搜索关键字）

**请求方式：**get

**请求url：**/search/productInfo/{current}/{limit}/{keyword}

**后端方法：**public Result searchProductInfo(@PathVariable Long current, @PathVariable Long limit, @PathVariable String keyword)

**后端返回数据参数：** ProductInfo (商品信息)

ProductInfoApi

// 商品搜索，分页显示  
@ApiOperation(value = "商品搜索，分页显示")  
@GetMapping("/search/productInfo/{current}/{limit}/{keyword}")  
**public** Result searchProductInfo(  
 @ApiParam(name = "current", value = "起始页", required = **true**)  
 @PathVariable Long current,  
  
 @ApiParam(name = "limit", value = "每页大小", required = **true**)  
 @PathVariable Long limit,  
  
 @ApiParam(name = "keyword", value = "商品搜索关键字", required = **true**)  
 @PathVariable String keyword) {  
  
 Page<ProductInfo> page = **new** Page<>(current, limit);  
 IPage<ProductInfo> pageModule= productInfoService.searchProductInfo(page, keyword);  
 **return** Result.*ok*(pageModule);  
}

ProductInfoService

// 商品搜索，分页显示  
IPage<ProductInfo> searchProductInfo(Page<ProductInfo> page, String keyword);

ProductInfoServiceImpl

*/\*\*  
 \* 商品搜索，分页显示  
 \* TODO 后续用搜索引擎 elastic search 实现  
 \** ***@param*** *page  
 \** ***@param*** *keyword  
 \** ***@return*** *\*/*@Override  
@Cacheable(value = "product", keyGenerator = "keyGeneratorPage") // redis缓存  
**public** IPage<ProductInfo> searchProductInfo(Page<ProductInfo> page, String keyword) {  
 // 如果查询的关键字为空则直接返回空  
 **if** (StringUtils.*isEmpty*(keyword)) {  
 **return null**;  
 }  
 // 查询条件  
 QueryWrapper<ProductInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.like("name", keyword).or().like("sub\_title", keyword);  
 IPage<ProductInfo> productInfoPage = baseMapper.selectPage(page, wrapper);  
  
 // 封装商品首张缩略图 评价数  
 productInfoPage.getRecords().stream().forEach(item -> {  
 // 封装商品参数  
 **this**.packParams(item);  
 });  
  
 **return** productInfoPage;  
}

login：登录页

主要业务

**实现用户登录功能**

**给邮箱发送验证码**

**微信扫码登录**

**邮箱登录**

**验证码（发邮箱里）**

控制层

UserInfoApi

url: /api/user/userInfo

发送验证码

**前端传入参数：**account（base64转码又加密了的用户名/手机号/邮箱），password（md5加密后的密码）

**请求方式：**post

**请求url：**/login/send/emailCode

**后端方法：**private Result sendEmailCode(@RequestBody UserLoginVo userLoginVo)

**后端返回数据参数：**成功或失败信息

UserInfoApi

// 发送邮箱验证码  
@ApiOperation("发送邮箱验证码")  
@PostMapping("/login/send/emailCode")  
**public** Result sendEmailCode(  
 @ApiParam(value = "userLoginVo", name = "用户登录信息", required = **true**)  
 @RequestBody UserLoginVo userLoginVo) {  
  
 userInfoService.sendEmailCode(userLoginVo);  
 **return** Result.*ok*();  
}

UserInfoService

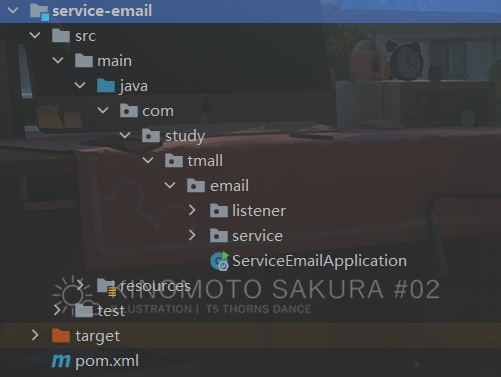
// 发送邮箱验证码  
**void** sendEmailCode(UserLoginVo userLoginVo);

UserInfoServiceImpl

*/\*\*  
 \* 发送邮箱验证码  
 \** ***@param*** *userLoginVo  
 \*/*@Override  
**public void** sendEmailCode(UserLoginVo userLoginVo) {  
 // 对account进行base64解码，这个是前端的base64编码后的，需要知道前端的编码后加密的方式  
 String account = Base64.*decode*(userLoginVo.getAccount());  
 // 对密码再进行MD5加密（前端进行了一次加密，后端还要进行加密，数据库中存入的密码是两次加密后的）  
 String password = MD5.*encrypt*(userLoginVo.getPassword());  
  
 // 通过account格式识别出用户以什么方式登录的，已进行相应的数据库查询  
 UserInfo userInfo = **this**.getUserInfoOfLogin(account, password);   
  
 // 查看用户是否被锁定  
 **if** (userInfo.getStatus() == UserLockStatusEnum.*LOCK*.getStatus()) {  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_DISABLED\_ERROR*);  
 }  
  
 // 获得用户绑定的邮箱，用于发登录验证码  
 String email = userInfo.getEmail();  
 EmailCodeVo emailCodeVo = **new** EmailCodeVo();  
 emailCodeVo.setEmail(email);  
 emailCodeVo.setType(EmailCodeTypeEnum.*LOGIN\_CODE*.getNumber());  
 // 发送消息到rabbitMQ队列，内容为邮箱地址  
 output.send(MessageBuilder.*withPayload*(emailCodeVo).build());  
}

*/\*\*  
 \* 通过account格式识别出用户以什么方式登录的，已进行相应的数据库查询  
 \** ***@param*** *account  
 \** ***@param*** *password  
 \** ***@return*** *\*/***private** UserInfo getUserInfoOfLogin(String account, String password) {  
 QueryWrapper<UserInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("password", password);  
 UserInfo userInfo;  
 **if** (Pattern.*matches*(RegularEnum.*USER\_NAME\_LOGIN*.getRegex(), account)) {  
 // 是用户名登录，就 用户名+密码 查询数据库  
 wrapper.eq("name", account);  
 userInfo = baseMapper.selectOne(wrapper);  
 **if** (userInfo == **null**) {  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_NAME\_FAIL*);  
 }  
 } **else if**(Pattern.*matches*(RegularEnum.*USER\_PHONE\_LOGIN*.getRegex(), account)) {  
 // 是手机号登录，就 手机号+密码 查询数据库  
 wrapper.eq("phone", account);  
 userInfo = baseMapper.selectOne(wrapper);  
 **if** (userInfo == **null**) {  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_PHONE\_FAIL*);  
 }  
 } **else if**(Pattern.*matches*(RegularEnum.*USER\_EMAIL\_LOGIN*.getRegex(), account)) {  
 // 是邮箱登录，就 邮箱+密码 查询数据库  
 wrapper.eq("email", account);  
 userInfo = baseMapper.selectOne(wrapper);  
 **if** (userInfo == **null**) {  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_EMAIL\_FAIL*);  
 }  
 } **else** {  
 // 如果都不满足，就用户名和密码错误  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_NAME\_FAIL*);  
 }  
 **return** userInfo;  
}

在service下创建service-email 模块



EmailMqListener

@Component  
@EnableBinding(Sink.**class**)  
**public class** EmailMqListener {  
 @Resource  
 **private** EmailService emailService;  
  
 */\*\*  
 \* 发送邮件  
 \*/* @StreamListener(Sink.*INPUT*)  
 **public void** sendEmail(Message<EmailCodeVo> message) {  
 EmailCodeVo emailCodeVo = message.getPayload();  
 emailService.sendCode(emailCodeVo);  
 }  
}

EmailService

// 发送验证码到目标邮箱  
**void** sendCode(EmailCodeVo emailCodeVo);

EmailServiceImpl

@Service  
**public class** EmailServiceImpl **implements** EmailService {  
 @Resource  
 **private** JavaMailSender mailSender;  
 @Resource  
 **private** StringRedisTemplate stringRedisTemplate;  
  
 */\*\*  
 \* 用来发送模版邮件  
 \*/* @Resource  
 **private** TemplateEngine templateEngine;  
  
 @Value("${spring.mail.username}")  
 **private** String from;  
  
 */\*\*  
 \* 发送验证码到目标邮箱  
 \** ***@param*** *emailCodeVo  
 \*/* @Override  
 **public void** sendCode(EmailCodeVo emailCodeVo) {  
 // 获取邮箱  
 String to = emailCodeVo.getEmail();  
 // 获取验证码类型  
 String codeType = EmailCodeTypeEnum.*getTypeByNumber*(emailCodeVo.getType());  
 // 先查询redis中是否已经有验证码了，有就返回  
 Object o = stringRedisTemplate.opsForValue().get(codeType + to);  
 **if** (o != **null**) {  
 **return**;  
 }  
 // 生成6位随机验证码  
 String code = String.*valueOf*(RandomUtil.*randomInt*(100000, 999999));  
  
 // 设置template中的参数  
 Context context = **new** Context();  
 context.setVariable("project", "tmall-v1.0");  
 context.setVariable("author", "2Executioner");  
 context.setVariable("code", code);  
 // 把template当作内容发送  
 String emailContent = templateEngine.process("mail", context);  
  
 MimeMessage message = mailSender.createMimeMessage();  
 MimeMessageHelper helper = **null**;  
 **try** {  
 helper = **new** MimeMessageHelper(message, **true**);  
 helper.setFrom(from); // 发送方  
 helper.setTo(to); // 接收方  
 helper.setSubject("tmall login code"); // 发送主题  
 helper.setText(emailContent, **true**); // 发送的内容（一个网页）  
 } **catch** (MessagingException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("开始发送");  
 mailSender.send(message);  
 System.*out*.println("已经发送");  
  
 // 写入到redis中去  
 stringRedisTemplate.opsForValue().set(codeType + to, code, 10, TimeUnit.*MINUTES*); // 验证码有效时间 十分钟  
 }  
}

用户登录

**前端传入参数：**account（base64转码又加密了的用户名/手机号/邮箱），password（md5加密后的密码），emailCode（邮箱验证码）

**请求方式：**post

**请求url：**/login

**后端方法：**private Result sendEmailCode(@RequestBody UserLoginVo userLoginVo)

**后端返回数据参数：**token（存了用户id和用户密码的token）

UserInfoApi

// 用户登录  
@ApiOperation("用户登录")  
@PostMapping("/login")  
**public** Result userLogin(  
 @ApiParam(value = "userLoginVo", name = "用户登录信息", required = **true**)  
 @RequestBody UserLoginVo userLoginVo) {  
  
 String token = userInfoService.userLogin(userLoginVo);  
 **return** Result.*ok*(token);  
}

UserInfoService

// 用户登录  
String userLogin(UserLoginVo userLoginVo);

UserInfoServiceImpl

*/\*\*  
/\*\*  
 \* 用户登录  
 \** ***@param*** *userLoginVo  
 \*/*@Override  
**public** String userLogin(UserLoginVo userLoginVo) {  
 // 对account进行base64解码，这个是前端的base64编码后的，需要知道前端的编码后加密的方式  
 String account = Base64.*decode*(userLoginVo.getAccount());  
 // 对验证码进行base64解码  
 String emailCode = Base64.*decode*(userLoginVo.getEmailCode());  
 // 对密码再进行MD5加密（前端进行了一次加密，后端还要进行加密，数据库中存入的密码是两次加密后的）  
 String password = MD5.*encrypt*(userLoginVo.getPassword());  
  
 // 通过account格式识别出用户以什么方式登录的，已进行相应的数据库查询  
 UserInfo userInfo = **this**.getUserInfoOfLogin(account, password);  
  
 // 从redis中查询出验证码  
 String code = stringRedisTemplate.opsForValue()  
 .get(EmailCodeTypeEnum.*LOGIN\_CODE*.getType() + userInfo.getEmail());  
 **if** (StringUtils.*isEmpty*(code) || !code.equals(emailCode)) {  
 **throw new** TmallException(ResultCodeEnum.*CODE\_ERROR*);  
 }  
  
 // 查看用户是否被锁定  
 **if** (userInfo.getStatus() == UserLockStatusEnum.*LOCK*.getStatus()) {  
 **throw new** TmallException(ResultCodeEnum.*LOGIN\_DISABLED\_ERROR*);  
 }  
  
 // 创建token  
 String token = JwtHelper.*createToken*(userInfo.getId(), userInfo.getPassword());  
 **return** token;  
}

regist：注册页

主要业务

**用户名重复性检测**

**检测邮箱是否被使用**

**给邮箱发送验证码**

**实现用户注册功能**

控制层

UserInfoApi

url: /api/user/userInfo

检测用户名是否被使用

**前端传入参数：**name（base64转码并加密了的用户名）

**请求方式：**post

**请求url：**/register/ nameRepeatCheck/{name}

**后端方法：**private Result userNameRepeatCheck (@PathVariable String name)

**后端返回数据参数：**重复或不重复（bool表示 True重复，False不重复）

UserInfoApi

// 检测用户名是否被使用  
@ApiOperation("检测用户名是否被使用")  
@PostMapping("/register/nameRepeatCheck/{name}")  
**public** Result userNameRepeatCheck (  
 @ApiParam(value = "name", name = "用户名", required = **true**)  
 @PathVariable String name) {  
  
 Boolean flag = userInfoService.userNameRepeatCheck(name);  
 **return** Result.*ok*(flag);  
}

UserInfoService

// 用户名重复检测  
Boolean userNameRepeatCheck(String name);

UserInfoServiceImpl

*/\*\*  
 \* 用户名重复检测  
 \** ***@param*** *name  
 \** ***@return*** *\*/*@Override  
**public** Boolean userNameRepeatCheck(String name) {  
 // 前端传来的是经过了base64编码并加密了的，先对用户名进行解密解码  
 String account = Base64.*decode*(name);  
 // 查询数据库  
 QueryWrapper<UserInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("name", account);  
 Integer integer = baseMapper.selectCount(wrapper);  
 // 如果查询结果不为0，则用户名已经重复，返回true  
 **if** (integer != 0) {  
 **return true**;  
 }  
 **return false**;  
}

检测邮箱是否被使用

**前端传入参数：**email（base64转码并加密了的邮箱）

**请求方式：**post

**请求url：**/register/emailRepeatCheck/{email}

**后端方法：**private Result userEmailRepeatCheck (@PathVariable String email)

**后端返回数据参数：**重复或不重复（bool表示 True重复，False不重复）

UserInfoApi

// 检测邮箱是否被使用  
@ApiOperation("检测邮箱是否被使用")  
@PostMapping("/register/emailRepeatCheck/{email}")  
**public** Result userEmailRepeatCheck (  
 @ApiParam(value = "email", name = "邮箱地址", required = **true**)  
 @PathVariable String email) {  
  
 Boolean flag = userInfoService.userEmailRepeatCheck(email);  
 **return** Result.*ok*(flag);  
}

UserInfoService

// 检测邮箱是否被使用  
Boolean userEmailRepeatCheck(String email);

UserInfoServiceImpl

*/\*\*  
 \* 检测邮箱是否被使用  
 \** ***@param*** *email  
 \** ***@return*** *\*/*@Override  
**public** Boolean userEmailRepeatCheck(String email) {  
 // 前端传来的是经过了base64编码并加密了的，先对邮箱地址进行解密解码  
 String decodeEmail = Base64.*decode*(email);  
 // 查询数据库  
 QueryWrapper<UserInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("email", decodeEmail);  
 Integer integer = baseMapper.selectCount(wrapper);  
 // 如果查询结果不为0，则邮箱地址已经重复，返回true  
 **if** (integer != 0) {  
 **return true**;  
 }  
 **return false**;  
}

发送验证码

**前端传入参数：**email（base64转码并加密了的邮箱地址）

**请求方式：**post

**请求url：**/register/send/{email}

**后端方法：**private Result sendEmailCode(@PathVariable String email)

**后端返回数据参数：**成功或失败信息

UserInfoApi

// 注册发送验证码到邮箱  
@ApiOperation("注册发送验证码到邮箱")  
@PostMapping("/register/send/{email}")  
**public** Result sendEmailCode (  
 @ApiParam(value = "email", name = "邮箱地址", required = **true**)  
 @PathVariable String email) {  
  
 userInfoService.sendEmailCode(email);  
 **return** Result.*ok*();  
}

UserInfoService

// 发送邮箱验证码  
**void** sendEmailCode(String email);

UserInfoServiceImpl

*/\*\*  
 \* 注册发送验证码到邮箱  
 \** ***@param*** *email  
 \*/*@Override  
**public void** sendEmailCode(String email) {  
 // 对邮箱地址进行解码  
 String decode = Base64.*decode*(email);  
 EmailCodeVo emailCodeVo = **new** EmailCodeVo();  
 emailCodeVo.setEmail(decode);  
 emailCodeVo.setType(EmailCodeTypeEnum.*REGISTER\_CODE*.getNumber());  
 // 发送到rabbitMQ中  
 output.send(MessageBuilder.*withPayload*(emailCodeVo).build());  
}

用户注册

**前端传入参数：**name，password，email，emailCode

**请求方式：**post

**请求url：**/register/user

**后端方法：**private Result userRegister(@RequestBody UserRegisterVo userRegisterVo)

**后端返回数据参数：**成功或失败信息

UserInfoApi

// 用户注册  
@ApiOperation("用户注册")  
@PostMapping("/register/user")  
**public** Result userRegister(  
 @ApiParam(value = "userRegisterVo", name = "用户注册对象", required = **true**)  
 @RequestBody UserRegisterVo userRegisterVo) {  
  
 userInfoService.userRegister(userRegisterVo);  
 **return** Result.*ok*();  
}

UserInfoService

// 用户注册  
**void** userRegister(UserRegisterVo userRegisterVo);

UserInfoServiceImpl

*/\*\*  
 \* 用户注册  
 \** ***@param*** *userRegisterVo  
 \*/*@Override  
**public void** userRegister(UserRegisterVo userRegisterVo) {  
 // 对用户名，邮箱，验证码 进行解码  
 String name = Base64.*decode*(userRegisterVo.getName());  
 String email = Base64.*decode*(userRegisterVo.getEmail());  
 String emailCode = Base64.*decode*(userRegisterVo.getEmailCode());  
  
 // 从redis中查询验证码是否正确  
 String code = stringRedisTemplate.opsForValue()  
 .get(EmailCodeTypeEnum.*REGISTER\_CODE*.getType() + email);  
 **if** (StringUtils.*isEmpty*(emailCode) || !emailCode.equals(code)) {  
 **throw new** TmallException(ResultCodeEnum.*CODE\_ERROR*);  
 }  
  
 // 对密码再进行一次加密  
 String password = MD5.*encrypt*(userRegisterVo.getPassword());  
  
 // 设置实体类  
 UserInfo userInfo = **new** UserInfo();  
 userInfo.setName(name);  
 userInfo.setEmail(email);  
 userInfo.setPassword(password);  
 // 写入到数据库中去  
 baseMapper.insert(userInfo);  
}

[account](javascript:;)Set ：账户设置页

orderItem：购物车

forebudy：下单页

pay：支付页

paySucceed：支付成功页

orderInfo：订单页

confirmReceipt：确认收货页

[tradeSuccessfully](javascript:;)：交易成功页

review：产品评价页