# 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 = productFeignClient.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 = productFeignClient.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;  
}

ProductFeignClient（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 = userFeignClient.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  
// 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 > name  
 StringBuilder name = **new** StringBuilder();  
 **if** (!StringUtils.*isEmpty*(userInfo.getNickName())) { // 昵称  
 name.append(userInfo.getNickName());  
 } **else if**(!StringUtils.*isEmpty*(userInfo.getName())) { // 用户名  
 name.append(userInfo.getName());  
 }  
 // 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**;  
 }  
 }  
}

模态框用户登录

**直接调用下面做登录页写的接口**

加入购物车

**前端传入参数：**token，orderItem

**请求方式：**post

**请求url：**/auth/join/orderItem

**后端方法：**public Result joinOrderItem(@RequestBody OrderItem orderItem, HttpServletRequest request)

**后端返回数据参数：** Result.ok(bool)

OrderItemApi

// 加入到购物车  
@ApiOperation("加入到购物车")  
@PostMapping("/auth/join/orderItem")  
**public** Result joinOrderItem(  
 @ApiParam(name = "orderItem", value = "订单项", required = **true**)  
 @RequestBody OrderItem orderItem,  
  
 @ApiParam(name = "request", value = "request", required = **true**)  
 HttpServletRequest request) {  
  
 String token = request.getHeader("token");  
 Boolean flag = orderItemService.joinOrderItem(token, orderItem);  
 **return** Result.*ok*(flag);  
}

OrderItemService

// 加入到购物车  
Boolean joinOrderItem(String token, OrderItem orderItem);

OrderItemServiceImpl

*/\*\*  
 \* 加入到购物车  
 \** ***@param*** *token  
 \** ***@param*** *orderItem  
 \*/*@Override  
**public** Boolean joinOrderItem(String token, OrderItem orderItem) {  
 String userId = JwtHelper.*getUserId*(token);  
  
 // 根据userId查询是否有未下单的购物车，且商品id相同，如果相同更新购物车中商品数量即可，不用新增加数据  
 QueryWrapper<OrderItem> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("user\_id", userId);  
 wrapper.eq("product\_id", orderItem.getProductId());  
 wrapper.isNull("order\_id");  
 OrderItem item = baseMapper.selectOne(wrapper);  
  
 // 远程调用service-product模块更新商品数量  
 Boolean flag = productFeignClient.updateProductNumber(orderItem);  
 **if** (flag) {  
 **int** insert;  
 // 如果购物车中没有该商品则添加数据  
 **if** (item == **null**) {  
 // 添加到数据库  
 orderItem.setUserId(userId);  
 insert = baseMapper.insert(orderItem);  
 } **else** { // 否则只更新商品数量  
 item.setNumber(item.getNumber() + orderItem.getNumber());  
 insert = baseMapper.updateById(item);  
 }  
  
 **return** insert == 1 ? **true** : **false**;  
 }  
  
 **return false**;  
}

ProductFeignClient（service\_product\_client模块内）

// 更新商品库存  
@PostMapping("/api/product/productInfo/inner/updateProductNumber")  
Boolean updateProductNumber(@RequestBody OrderItem orderItem);

ProductInfoApi （service-product模块内）

// 更新商品数量  
@ApiOperation("更新商品数量")  
@PostMapping("/inner/updateProductNumber")  
**public** Boolean updateProductNumber(  
 @ApiParam(name = "orderItem", value = "订单项", required = **true**)  
 @RequestBody OrderItem orderItem) {  
  
 **return** productInfoService.updateProductNumber(orderItem);  
}

ProductInfoService

// 更新商品库存  
Boolean updateProductNumber(OrderItem orderItem);

ProductInfoServiceImpl

*/\*\*  
 \* 更新商品库存  
 \** ***@param*** *orderItem  
 \** ***@return*** *\*/*@Override  
**public** Boolean updateProductNumber(OrderItem orderItem) {  
 // 取出商品信息  
 String productId = orderItem.getProductId();  
 Integer number = orderItem.getNumber();  
 **if** (StringUtils.*isEmpty*(productId) || number == **null** || number <= 0) {  
 **throw new** TmallException(ResultCodeEnum.*PARAM\_ERROR*);  
 }  
 // 根据id查询商品  
 ProductInfo productInfo = baseMapper.selectById(productId);  
 **if** (productInfo == **null**) { // 如果查询结果为null，则抛出数据异常  
 **throw new** TmallException(ResultCodeEnum.*DATA\_ERROR*);  
 }  
 productInfo.setStock(productInfo.getStock() - number);  
 // 更新数据库  
 **int** i = baseMapper.updateById(productInfo);  
  
 **return** i == 1 ? **true** : **false**;  
}

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 = "keyGeneratorKeywordPage") // redis缓存  
 // 这里暂时不用redis缓存，因为不同的关键字会都会添加进redis中，这样增加了redis的负担  
 **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

**账号登录（可以通过会员名，手机号和邮箱号登录）**

**给邮箱发送验证码**

**微信扫码登录（每1分钟获取微信登录二维码，扫码登录）**

**验证码（发邮箱里）**

控制层

UserInfoApi

url: /api/user/userInfo

WeChatApi

url: /api/user/weChat

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）

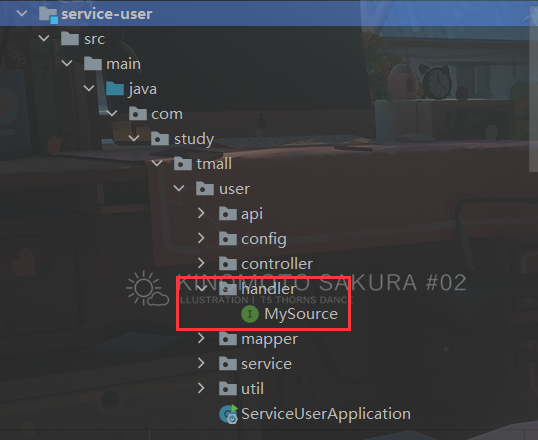
**请求方式：**post

**请求url：**/login/send/emailCode

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

**后端返回数据参数：**成功或失败信息

在service-user中的handler创建MySource



MySource

**public interface** MySource {  
 String *OUTPUT* = "output"; // 标准消息发送通道  
 String *CODE\_SEND* = "codeSend"; // 验证码发送消息发送  
 String *CODE\_DEL\_SEND* = "codeDelSend"; // 验证码删除消息发送  
  
 @Output("output")  
 MessageChannel output();  
  
 @Output("codeSend")  
 MessageChannel codeSend();  
  
 @Output("codeDelSend")  
 MessageChannel codeDelSend();  
}

UserInfoApi

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

UserInfoService

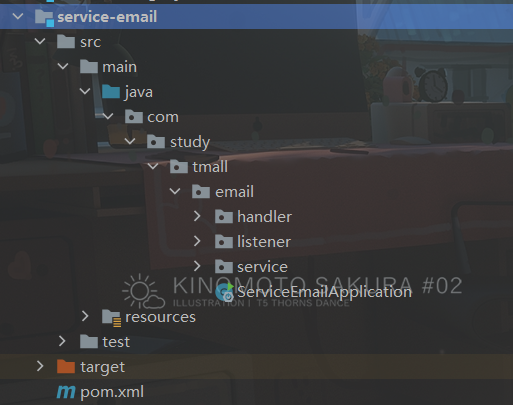
// 发送邮箱验证码  
**void** sendEmailCode(UserLoginVo userLoginVo);

UserInfoServiceImpl （类上增加注解： @EnableBinding(MySource.class) ）

@Resource**private**MessageChannelcodeSend*; // 验证码发送  
/\*\*  
 \* 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  
 codeSend.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



MySink

**public interface** MySink {  
 String *INPUT* = "input"; // 标准消息接收通道  
 String *CODE\_RECEIVE* = "codeReceive"; // 验证码发送消息接收  
 String *CODE\_DEL\_RECEIVE* = "codeDelReceive"; // 验证码删除消息接收  
  
 @Input("input")  
 SubscribableChannel input();  
  
 @Input("codeReceive")  
 SubscribableChannel codeReceive();  
  
 @Input("codeDelReceive")  
 SubscribableChannel codeDelReceive();  
}

EmailMqListener

@Component  
@EnableBinding(MySink.**class**)  
**public class** EmailMqListener {  
 @Resource  
 **private** EmailService emailService;  
  
 */\*\*  
 \* 发送邮件  
 \** ***@param*** *message  
 \*/* @StreamListener(MySink.*CODE\_RECEIVE*)  
 **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  
 }  
}

账号登录

**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：**public 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(name = "userLoginVo", value = "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;  
}

获取微信登录二维码

**前端传入参数：**type（二维码类型，映射回调方法）

**请求方式：**post

**请求url：**/get/QRCode/{type}

**后端方法：**public Result weChatQRCode(@PathVariable Integer type)

**后端返回数据参数：**二维码链接

微信登录相关工具类

ConstantWxPropertiesUtil

@Component  
**public class** ConstantWxPropertiesUtil **implements** InitializingBean {  
 @Value("${wx.open.app\_id}")  
 **private** String appId;  
  
 @Value("${wx.open.app\_secret}")  
 **private** String appSecret;  
  
 @Value("${wx.open.redirect\_url}")  
 **private** String redirectUrl;  
  
 @Value("${wx.open.binding\_url}")  
 **private** String bindingUrl;  
  
 **public static** String *WX\_OPEN\_APP\_ID*;  
 **public static** String *WX\_OPEN\_APP\_SECRET*;  
 **public static** String *WX\_OPEN\_REDIRECT\_URL*;  
 **public static** String *WX\_OPEN\_BINDING\_URL*;  
  
 @Override  
 **public void** afterPropertiesSet() **throws** Exception {  
 *WX\_OPEN\_APP\_ID* = appId;  
 *WX\_OPEN\_APP\_SECRET* = appSecret;  
 *WX\_OPEN\_REDIRECT\_URL* = redirectUrl;  
 *WX\_OPEN\_BINDING\_URL* = bindingUrl;  
 }  
}

HttpClientUtils

**public class** HttpClientUtils {  
  
 **public static final int** *connTimeout*=10000;  
 **public static final int** *readTimeout*=10000;  
 **public static final** String *charset*="UTF-8";  
 **private static** HttpClient *client* = **null**;  
  
 **static** {  
 PoolingHttpClientConnectionManager cm = **new** PoolingHttpClientConnectionManager();  
 cm.setMaxTotal(128);  
 cm.setDefaultMaxPerRoute(128);  
 *client* = HttpClients.*custom*().setConnectionManager(cm).build();  
 }  
  
 **public static** String postParameters(String url, String parameterStr) **throws** ConnectTimeoutException, SocketTimeoutException, Exception{  
 **return** *post*(url,parameterStr,"application/x-www-form-urlencoded",*charset*,*connTimeout*,*readTimeout*);  
 }  
  
 **public static** String postParameters(String url, String parameterStr,String charset, Integer connTimeout, Integer readTimeout) **throws** ConnectTimeoutException, SocketTimeoutException, Exception{  
 **return** *post*(url,parameterStr,"application/x-www-form-urlencoded",charset,connTimeout,readTimeout);  
 }  
  
 **public static** String postParameters(String url, Map<String, String> params) **throws** ConnectTimeoutException,  
 SocketTimeoutException, Exception {  
 **return** *postForm*(url, params, **null**, *connTimeout*, *readTimeout*);  
 }  
  
 **public static** String postParameters(String url, Map<String, String> params, Integer connTimeout,Integer readTimeout) **throws** ConnectTimeoutException,  
 SocketTimeoutException, Exception {  
 **return** *postForm*(url, params, **null**, connTimeout, readTimeout);  
 }  
  
 **public static** String get(String url) **throws** Exception {  
 **return** *get*(url, *charset*, **null**, **null**);  
 }  
  
 **public static** String get(String url, String charset) **throws** Exception {  
 **return** *get*(url, charset, *connTimeout*, *readTimeout*);  
 }  
  
 */\*\*  
 \* 发送一个 Post 请求, 使用指定的字符集编码.  
 \*  
 \** ***@param*** *url  
 \** ***@param*** *body RequestBody  
 \** ***@param*** *mimeType 例如 application/xml "application/x-www-form-urlencoded" a=1&b=2&c=3  
 \** ***@param*** *charset 编码  
 \** ***@param*** *connTimeout 建立链接超时时间,毫秒.  
 \** ***@param*** *readTimeout 响应超时时间,毫秒.  
 \** ***@return*** *ResponseBody, 使用指定的字符集编码.  
 \** ***@throws*** *ConnectTimeoutException 建立链接超时异常  
 \** ***@throws*** *SocketTimeoutException 响应超时  
 \** ***@throws*** *Exception  
 \*/* **public static** String post(String url, String body, String mimeType,String charset, Integer connTimeout, Integer readTimeout)  
 **throws** ConnectTimeoutException, SocketTimeoutException, Exception {  
 HttpClient client = **null**;  
 HttpPost post = **new** HttpPost(url);  
 String result = "";  
 **try** {  
 **if** (StringUtils.*isNotBlank*(body)) {  
 HttpEntity entity = **new** StringEntity(body, ContentType.*create*(mimeType, charset));  
 post.setEntity(entity);  
 }  
 // 设置参数  
 Builder customReqConf = RequestConfig.*custom*();  
 **if** (connTimeout != **null**) {  
 customReqConf.setConnectTimeout(connTimeout);  
 }  
 **if** (readTimeout != **null**) {  
 customReqConf.setSocketTimeout(readTimeout);  
 }  
 post.setConfig(customReqConf.build());  
  
 HttpResponse res;  
 **if** (url.startsWith("https")) {  
 // 执行 Https 请求.  
 client = *createSSLInsecureClient*();  
 res = client.execute(post);  
 } **else** {  
 // 执行 Http 请求.  
 client = HttpClientUtils.*client*;  
 res = client.execute(post);  
 }  
 result = IOUtils.*toString*(res.getEntity().getContent(), charset);  
 } **finally** {  
 post.releaseConnection();  
 **if** (url.startsWith("https") && client != **null**&& client **instanceof** CloseableHttpClient) {  
 ((CloseableHttpClient) client).close();  
 }  
 }  
 **return** result;  
 }  
  
  
 */\*\*  
 \* 提交form表单  
 \*  
 \** ***@param*** *url  
 \** ***@param*** *params  
 \** ***@param*** *connTimeout  
 \** ***@param*** *readTimeout  
 \** ***@return*** *\** ***@throws*** *ConnectTimeoutException  
 \** ***@throws*** *SocketTimeoutException  
 \** ***@throws*** *Exception  
 \*/* **public static** String postForm(String url, Map<String, String> params, Map<String, String> headers, Integer connTimeout,Integer readTimeout) **throws** ConnectTimeoutException,  
 SocketTimeoutException, Exception {  
  
 HttpClient client = **null**;  
 HttpPost post = **new** HttpPost(url);  
 **try** {  
 **if** (params != **null** && !params.isEmpty()) {  
 List<NameValuePair> formParams = **new** ArrayList<NameValuePair>();  
 Set<Entry<String, String>> entrySet = params.entrySet();  
 **for** (Entry<String, String> entry : entrySet) {  
 formParams.add(**new** BasicNameValuePair(entry.getKey(), entry.getValue()));  
 }  
 UrlEncodedFormEntity entity = **new** UrlEncodedFormEntity(formParams, Consts.*UTF\_8*);  
 post.setEntity(entity);  
 }  
  
 **if** (headers != **null** && !headers.isEmpty()) {  
 **for** (Entry<String, String> entry : headers.entrySet()) {  
 post.addHeader(entry.getKey(), entry.getValue());  
 }  
 }  
 // 设置参数  
 Builder customReqConf = RequestConfig.*custom*();  
 **if** (connTimeout != **null**) {  
 customReqConf.setConnectTimeout(connTimeout);  
 }  
 **if** (readTimeout != **null**) {  
 customReqConf.setSocketTimeout(readTimeout);  
 }  
 post.setConfig(customReqConf.build());  
 HttpResponse res = **null**;  
 **if** (url.startsWith("https")) {  
 // 执行 Https 请求.  
 client = *createSSLInsecureClient*();  
 res = client.execute(post);  
 } **else** {  
 // 执行 Http 请求.  
 client = HttpClientUtils.*client*;  
 res = client.execute(post);  
 }  
 **return** IOUtils.*toString*(res.getEntity().getContent(), "UTF-8");  
 } **finally** {  
 post.releaseConnection();  
 **if** (url.startsWith("https") && client != **null** && client **instanceof** CloseableHttpClient) {  
 ((CloseableHttpClient) client).close();  
 }  
 }  
 }  
  
 */\*\*  
 \* 发送一个 GET 请求  
 \*/* **public static** String get(String url, String charset, Integer connTimeout,Integer readTimeout)  
 **throws** ConnectTimeoutException,SocketTimeoutException, Exception {  
  
 HttpClient client = **null**;  
 HttpGet get = **new** HttpGet(url);  
 String result = "";  
 **try** {  
 // 设置参数  
 Builder customReqConf = RequestConfig.*custom*();  
 **if** (connTimeout != **null**) {  
 customReqConf.setConnectTimeout(connTimeout);  
 }  
 **if** (readTimeout != **null**) {  
 customReqConf.setSocketTimeout(readTimeout);  
 }  
 get.setConfig(customReqConf.build());  
  
 HttpResponse res = **null**;  
  
 **if** (url.startsWith("https")) {  
 // 执行 Https 请求.  
 client = *createSSLInsecureClient*();  
 res = client.execute(get);  
 } **else** {  
 // 执行 Http 请求.  
 client = HttpClientUtils.*client*;  
 res = client.execute(get);  
 }  
  
 result = IOUtils.*toString*(res.getEntity().getContent(), charset);  
 } **finally** {  
 get.releaseConnection();  
 **if** (url.startsWith("https") && client != **null** && client **instanceof** CloseableHttpClient) {  
 ((CloseableHttpClient) client).close();  
 }  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* 从 response 里获取 charset  
 \*/* @SuppressWarnings("unused")  
 **private static** String getCharsetFromResponse(HttpResponse ressponse) {  
 // Content-Type:text/html; charset=GBK  
 **if** (ressponse.getEntity() != **null** && ressponse.getEntity().getContentType() != **null** && ressponse.getEntity().getContentType().getValue() != **null**) {  
 String contentType = ressponse.getEntity().getContentType().getValue();  
 **if** (contentType.contains("charset=")) {  
 **return** contentType.substring(contentType.indexOf("charset=") + 8);  
 }  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* 创建 SSL连接  
 \** ***@return*** *\** ***@throws*** *GeneralSecurityException  
 \*/* **private static** CloseableHttpClient createSSLInsecureClient() **throws** GeneralSecurityException {  
 **try** {  
 SSLContext sslContext = **new** SSLContextBuilder().loadTrustMaterial(**null**, **new** TrustStrategy() {  
 **public boolean** isTrusted(X509Certificate[] chain,String authType) **throws** CertificateException {  
 **return true**;  
 }  
 }).build();  
  
 SSLConnectionSocketFactory sslsf = **new** SSLConnectionSocketFactory(sslContext, **new** X509HostnameVerifier() {  
  
 @Override  
 **public boolean** verify(String arg0, SSLSession arg1) {  
 **return true**;  
 }  
  
 @Override  
 **public void** verify(String host, SSLSocket ssl)  
 **throws** IOException {  
 }  
  
 @Override  
 **public void** verify(String host, X509Certificate cert)  
 **throws** SSLException {  
 }  
  
 @Override  
 **public void** verify(String host, String[] cns,  
 String[] subjectAlts) **throws** SSLException {  
 }  
 });  
 **return** HttpClients.*custom*().setSSLSocketFactory(sslsf).build();  
  
 } **catch** (GeneralSecurityException e) {  
 **throw** e;  
 }  
 }  
}

WeChatApi

// 获取登录二维码  
@ApiOperation("获取登录二维码")  
@GetMapping("/get/QRCode/{type}")  
**public** Result weChatQRCode(  
 @ApiParam(name = "type", value = "回调接口", required = **true**)  
 @PathVariable Integer type) {  
 Map<String, String> map = weChatService.weChatQRCode(type);  
 **return** map == **null** ? **null** : Result.*ok*(map);  
}

WeChatService

// 获取微信登录二维码  
Map<String, String> weChatQRCode(Integer type);

WeChatServiceImpl

*/\*\*  
 \* 获取微信登录二维码  
 \** ***@param*** *type 回调类型，映射回调地址  
 \** ***@return*** *\*/*@Override  
**public** Map<String, String> weChatQRCode(Integer type) {  
 **try** {  
 Map<String, String> map = **new** HashMap<>();  
 String encode = "";  
 // 把回调地址进行utf-8编码  
 **if** (type != **null**) {  
 **if** (type == WeChatRedirectTypeEnum.*LOGIN*.getType()) { // 微信登录回调地址  
 encode = URLEncoder.*encode*(ConstantWxPropertiesUtil.*WX\_OPEN\_REDIRECT\_URL*, "utf-8");  
 } **else if** (type == WeChatRedirectTypeEnum.*BINDING*.getType()) { // 微信绑定回调地址  
 encode = URLEncoder.*encode*(ConstantWxPropertiesUtil.*WX\_OPEN\_BINDING\_URL*, "utf-8");  
 }  
 }  
 // 生成一个uuid，便于向redis进行轮询，判别用户是否扫码  
 String uuid = IdUtil.*randomUUID*().replaceAll("-", "");  
 // 对uuid进行base64编码并进行加密  
 uuid = Base64.*encode*(uuid);  
  
 // 拼接二维码url地址  
 */\*\*  
 \* appid: 测试号有提供  
 \* redirect\_uri: 扫码成功后的回调地址，就自己些的那个api接口的地址  
 \* response\_type: 响应类型，直接就二维码链接，所以填code  
 \* scope: 如果是静默登录（用户无感知）就填snsapi\_userinfo  
 \* state: 搞个uuid，以便于在redis中做轮询查询  
 \* #wechat\_redirect：无参数，就写在url最末尾  
 \*  
 \* 微信官方例子：  
 \* https://open.weixin.qq.com/connect/oauth2/authorize  
 \* ?appid=wxf0e81c3bee622d60  
 \* &redirect\_uri=http%3A%2F%2Fnba.bluewebgame.com%2Foauth\_response.php  
 \* &response\_type=code  
 \* &scope=snsapi\_userinfo  
 \* &state=STATE  
 \* #wechat\_redirect  
 \*/* String url = "https://open.weixin.qq.com/connect/oauth2/authorize"  
 + "?appid=" + ConstantWxPropertiesUtil.*WX\_OPEN\_APP\_ID* + "&redirect\_uri=" + encode  
 + "&response\_type=code"  
 + "&scope=snsapi\_userinfo" // snsapi\_userinfo 是公众号的scope  
 + "&state=" + uuid // 官方说搞一个state要安全些，值可以是随机一个字符串，这里用uuid，为了方便查redis  
 + "#wechat\_redirect";  
  
 map.put("QRCodeUrl", url);  
 map.put("uuid", uuid);  
 **return** map;  
 } **catch** (UnsupportedEncodingException e) {  
 e.printStackTrace();  
 **return null**;  
 }  
}

微信扫码回调

**前端传入参数：**code（获取微信token的票据），state（从redis中拿取tmall 用户token的票据）

**请求方式：**all

**请求url：**/callback

**后端方法：**public Result callback(String code, String state)

**后端返回数据参数：**true 或 false

WeChatApi

// 用户扫码成功后的回调接口  
@ApiOperation("用户扫码成功后的回调接口")  
@RequestMapping("/callback")  
**public** Result callback(  
 @ApiParam(name = "code", value = "从微信获取accessToken的票据", required = **true**)  
 String code,  
  
 @ApiParam(name = "state", value = "把获取到的accessToken存入redis中的key", required = **true**)  
 String state) {  
  
 Boolean flag = weChatService.callback(code, state);  
 **return** flag ? Result.*ok*("登录成功！") : Result.*fail*("登陆失败！");  
}

WeChatService

// 用户扫码成功后的回调接口  
Boolean callback(String code, String state);

WeChatServiceImpl

*/\*\*  
 \* 用户扫码成功后的回调接口  
 \** ***@param*** *code  
 \** ***@param*** *state  
 \** ***@return*** *\*/*@Override  
**public** Boolean callback(String code, String state) {  
 **try** {  
 Map<String, Object> map = **new** HashMap<>();  
 */\*\*  
 \* 通过code获取accessToken的官方案例：  
 \* https://api.weixin.qq.com/sns/oauth2/access\_token  
 \* ?appid=APPID  
 \* &secret=SECRET  
 \* &code=CODE  
 \* &grant\_type=authorization\_code  
 \*/* // 因为每次的code不一样，所以拼接通用请求地址接收内存空间  
 StringBuffer baseAccessTokenUrl = **new** StringBuffer()  
 .append("https://api.weixin.qq.com/sns/oauth2/access\_token")  
 .append("?appid=%s")  
 .append("&secret=%s")  
 .append("&code=%s")  
 .append("&grant\_type=authorization\_code"); // 固定参数  
 // 格式化请求地址  
 String accessTokenUrl = String.*format*(baseAccessTokenUrl.toString(),  
 ConstantWxPropertiesUtil.*WX\_OPEN\_APP\_ID*,  
 ConstantWxPropertiesUtil.*WX\_OPEN\_APP\_SECRET*,  
 code);  
 // 向获取accessToken的url地址发送请求  
 String jsonString = HttpClientUtils.*get*(accessTokenUrl);  
 JSONObject jsonObject = JSONObject.*parseObject*(jsonString);  
 String access\_token = jsonObject.getString("access\_token");  
 String openid = jsonObject.getString("openid");  
  
 // 根据获取到的access\_token获取用户详情信息  
 // 先根据openId查询数据库，如果数据库为空才去微信请求获取  
 UserInfo userInfo = userInfoService.getByOpenid(openid);  
 **if** (userInfo == **null**) {  
 // 请求微信获取用户信息  
 StringBuffer baseUserInfoUrl = **new** StringBuffer()  
 .append("https://api.weixin.qq.com/sns/userinfo")  
 .append("?access\_token=%s")  
 .append("&openid=%s")  
 .append("&lang=zh\_CN");  
 String getUserInfoUrl = String.*format*(baseUserInfoUrl.toString(), access\_token, openid);  
 String resultInfo = HttpClientUtils.*get*(getUserInfoUrl);  
 JSONObject resultUserInfoJson = JSONObject.*parseObject*(resultInfo);  
 // 解析用户信息  
 String nickname = resultUserInfoJson.getString("nickname"); // 用户昵称  
 //String headimgurl = resultUserInfoJson.getString("headimgurl"); //用户头像  
 // 写入到数据库中  
 userInfo = **new** UserInfo();  
 userInfo.setOpenid(openid);  
 userInfo.setNickName(nickname);  
 userInfoService.save(userInfo);  
 // 邮箱未绑定，传入前端，前端跳转到邮箱绑定页面  
 map.put("state", ResultCodeEnum.*EMAIL\_UNBIND*.getCode());  
 } **else** {  
 **if** (StringUtils.*isEmpty*(userInfo.getEmail())) {  
 // 邮箱未绑定，传入前端，前端跳转到邮箱绑定页面  
 map.put("state", ResultCodeEnum.*EMAIL\_UNBIND*.getCode());  
 } **else** {  
 // 邮箱已绑定，传入前端，前端跳转到首页  
 map.put("state", ResultCodeEnum.*EMAIL\_BIND*.getCode());  
 }  
 }  
 // 做成token存入redis，有效时间为60s  
 String token = JwtHelper.*createToken*(userInfo.getId(), userInfo.getPassword());  
 map.put("token", token);  
 redisTemplate.opsForValue().set(state, map, 60, TimeUnit.*MINUTES*);  
 **return true**;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 **return false**;  
 }  
}

用户登录扫码状态轮询

**前端传入参数：** state（从redis中拿取tmall 用户token的票据）

**请求方式：**post

**请求url：**/polling

**后端方法：**public Result polling(String state)

**后端返回数据参数：**map （包含用户token和邮箱绑定状态）

WeChatApi

// 用户登录扫码状态轮询  
@ApiOperation("用户登录扫码状态轮询")  
@PostMapping("/polling/{state}")  
**public** Result polling(  
 @ApiParam(name = "state", value = "登录信息查询凭证", required = **true**)  
 @PathVariable String state) {  
  
 Map<String, Object> map = weChatService.polling(state);  
 **return** Result.*ok*(map);  
}

WeChatService

// 用户登录扫码状态轮询  
Map<String, Object> polling(String state);

WeChatServiceImpl

*/\*\*  
 \* 用户登录扫码状态轮询  
 \** ***@param*** *state  
 \** ***@return*** *\*/*@Override  
**public** Map<String, Object> polling(String state) {  
 Map<String, Object> map = (Map<String, Object>) redisTemplate.opsForValue().get(state);  
 **return** map;  
}

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(name = "name", value = "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(name = "email", value = "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(name = "email", value = "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  
 codeSend.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(name = "userRegisterVo", value = "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 key = EmailCodeTypeEnum.*REGISTER\_CODE*.getType() + email;  
 String code = stringRedisTemplate.opsForValue().get(key);  
 **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);  
  
 // 注册成功后删除验证码  
 codeDelSend.send(MessageBuilder.*withPayload*(key).build());  
}

用户邮箱绑定

**前端传入参数：**token, userRegisterVo（用户注册vo）

**请求方式：**post

**请求url：**/register/emailBinding/{token}

**后端方法：**public Result emailBinding(@PathVariable String token, @RequestBody UserRegisterVo userRegisterVo)

**后端返回数据参数：**绑定成功后返回token

**备注：**此接口和用户注册类似

UserInfoApi

// 用户邮箱绑定  
@ApiOperation("用户邮箱绑定")  
@PostMapping("/register/emailBinding/{token}")  
**public** Result emailBinding(  
 @ApiParam(name = "token", value = "token", required = **true**)  
 @PathVariable String token,  
  
 @ApiParam(name = "userRegisterVo", value = "用户注册对象", required = **true**)  
 @RequestBody UserRegisterVo userRegisterVo) {  
  
 token = userInfoService.emailBinding(token, userRegisterVo);  
 **return** Result.*ok*(token);  
}

UserInfoService

// 用户邮箱绑定  
String emailBinding(String token, UserRegisterVo userRegisterVo);

UserInfoServiceImpl

*/\*\*  
 \* 用户邮箱绑定  
 \** ***@param*** *token  
 \** ***@param*** *userRegisterVo  
 \*/*@Override  
**public** String emailBinding(String token, UserRegisterVo userRegisterVo) {  
 // 解析token  
 String userId = JwtHelper.*getUserId*(token);  
  
 // 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.setId(userId);  
 userInfo.setName(name);  
 userInfo.setEmail(email);  
 userInfo.setPassword(password);  
 // 写入到数据库中去  
 baseMapper.updateById(userInfo);  
  
 // 返回一个全新的token  
 String newToken = JwtHelper.*createToken*(userId, password);  
 **return** newToken;  
}

header：页面头

主要业务

**根据token获取用户信息**

控制层

UserInfoApi

url: /api/user/userInfo

根据token获取用户信息

**前端传入参数：**token

**请求方式：**post

**请求url：**/auth/getUserInfo

**后端方法：**Result getUserInfoByToken(HttpServletRequest request)

**后端返回数据参数：**绑定成功后返回token

UserInfoApi

// 根据token获取用户信息  
@ApiOperation("获取用户信息")  
@PostMapping("/auth/getUserInfo")  
**public** Result getUserInfoByToken(  
 @ApiParam(name = "request", value = "HttpServletRequest", required = **true**)  
 HttpServletRequest request) {  
 String token = request.getHeader("token");  
 UserInfoVo userInfoVo = userInfoService.getUserInfoByToken(token);  
 **return** Result.*ok*(userInfoVo);  
}

UserInfoService

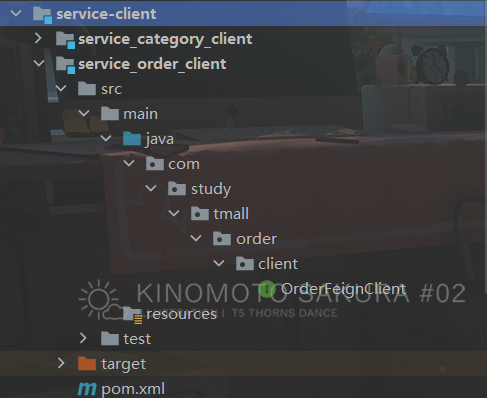
// 根据token获取用户信息  
UserInfoVo getUserInfoByToken(String token);

UserInfoServiceImpl

*/\*\*  
 \* 根据token获取用户信息  
 \** ***@param*** *token  
 \** ***@return*** *\*/*@Override  
**public** UserInfoVo getUserInfoByToken(String token) {  
 // 获取用户id和用户密码  
 String userId = JwtHelper.*getUserId*(token);  
 String password = JwtHelper.*getPassword*(token);  
  
 // 根据id和密码进行查询  
 QueryWrapper<UserInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("id", userId);  
 wrapper.eq("password", password);  
 UserInfo userInfo = baseMapper.selectOne(wrapper);  
 **if** (userInfo == **null**) {  
 **throw new** TmallException(ResultCodeEnum.*FETCH\_USERINFO\_ERROR*);  
 }  
  
 // 对用户数据进行包装  
 UserInfoVo userInfoVo = **new** UserInfoVo();  
 userInfoVo.setId(userId);  
 // 拿取用户名称  
 String name = **this**.getName(userInfo);  
 userInfoVo.setName(name);  
 // 远程调用获得用户购物车商品数量  
 Integer number = orderFeignClient.getProductNumberByUserId(userId);  
 userInfoVo.setProductNumber(number);  
  
 **return** userInfoVo;  
}

// 拿取用户名称  
**private** String getName(UserInfo userInfo) {  
 **if** (!StringUtils.*isEmpty*(userInfo.getNickName())) {  
 // 如果昵称不为空则返回昵称  
 **return** userInfo.getNickName();  
 } **else** {  
 // 否则返回会员名  
 **return** userInfo.getName();  
 }  
}

创建service\_order\_client（service-client模块下）



OrderFeignClient（service\_order\_client模块）

@Component  
@FeignClient("service-order")  
**public interface** OrderFeignClient {  
 // 根据用户id获取购物车商品数量（内部调用）  
 @GetMapping("/api/order/orderItem/inner/getProductNumber/{userId}")  
 Integer getProductNumberByUserId(@PathVariable("userId") String userId);  
  
}

OrderItemApi（service-order模块）

// 根据用户id获取购物车商品数量（内部调用）  
@ApiOperation("根据用户id获取购物车商品数量（内部调用）")  
@GetMapping("/inner/getProductNumber/{userId}")  
**public** Integer getProductNumberByUserId(  
 @ApiParam(name = "userId", value = "用户id", required = **true**)  
 @PathVariable String userId) {  
  
 **return** orderItemService.getProductNumberByUserId(userId);  
}

OrderItemService

// 根据用户id获取购物车商品数量（内部调用）  
Integer getProductNumberByUserId(String userId);

OrderItemServiceImpl

*/\*\*  
 \* 根据用户id获取购物车商品数量（内部调用）  
 \** ***@param*** *userId  
 \** ***@return*** *\*/*@Override  
**public** Integer getProductNumberByUserId(String userId) {  
  
 **return** baseMapper.getProductNumberByUserId(userId);  
}

OrderItemMapper

// 根据用户id获取购物车商品数量（内部调用）  
Integer getProductNumberByUserId(String userId);

OrderItemMapper.xml

<**select** id="getProductNumberByUserId" resultType="java.lang.Integer">  
 SELECT  
 SUM(number)  
 FROM  
 order\_item  
 WHERE  
 user\_id=#{userId}  
 AND (order\_id IS NULL)  
</**select**>

[account](javascript:;)\_set ：账户设置页

主要业务

* + - **基本信息**
      * **获取用户详情信息**
      * **用户头像更改**
      * **用户昵称更改**
    - **安全设置**
      * **密码修改**
      * **邮箱改绑**
      * **手机号绑定**
      * **手机号解绑**
      * **微信号绑定**
      * **微信号解绑**
    - **实名认证**
    - **其它设置**

控制层

UserInfoApi

url: /api/user/userInfo

WeChatApi

url: /api/user/weChat

根据token获取用户详情信息

**前端传入参数：**token

**请求方式：**post

**请求url：**/auth/getUserDetailsInfo

**后端方法：**Result getUserDetailsInfoByToken(HttpServletRequest request)

**后端返回数据参数：**返回用户详情信息

UserInfoApi

// 根据token获取用户详情信息  
@ApiOperation("根据token获取用户详情信息")  
@PostMapping("/auth/getUserDetailsInfo")  
**public** Result getUserDetailsInfoByToken(  
 @ApiParam(name = "request", value = "HttpServletRequest", required = **true**)  
 HttpServletRequest request) {  
 String token = request.getHeader("token");  
 UserInfo userInfo = userInfoService.getUserDetailsInfoByToken(token);  
 **return** Result.*ok*(userInfo);  
}

UserInfoService

// 根据token获取用户详情信息  
UserInfo getUserDetailsInfoByToken(String token);

UserInfoServiceImpl

*/\*\*  
 \* 根据token获取用户详情信息  
 \** ***@param*** *token  
 \** ***@return*** *\*/*@Override  
**public** UserInfo getUserDetailsInfoByToken(String token) {  
 // 获取用户id和用户密码  
 String userId = JwtHelper.*getUserId*(token);  
 String password = JwtHelper.*getPassword*(token);  
  
 // 根据id和密码进行查询  
 QueryWrapper<UserInfo> wrapper = **new** QueryWrapper<>();  
 wrapper.eq("id", userId);  
 wrapper.eq("password", password);  
 UserInfo userInfo = baseMapper.selectOne(wrapper);  
 **if** (userInfo == **null**) {  
 **throw new** TmallException(ResultCodeEnum.*FETCH\_USERINFO\_ERROR*);  
 }  
 // 密码 id 置空  
 userInfo.setPassword(**null**);  
 userInfo.setId(**null**);  
  
 **return** userInfo;  
}

修改用户昵称

**前端传入参数：**token，nickName（用户昵称）

**请求方式：**post

**请求url：**/auth/update/nickName/{nickName }

**后端方法：**Result updateUserNickName(HttpServletRequest request, @PathVariable String nickName)

**后端返回数据参数：**Result.ok()

UserInfoApi

// 修改用户昵称  
@ApiOperation("修改用户昵称")  
@PostMapping("/auth/update/nickName/{nickName}")  
**public** Result updateNickName(  
 @ApiParam(name = "nickName", value = "用户昵称", required = **true**)  
 @PathVariable String nickName,  
  
 @ApiParam(name = "request", value = "request", required = **true**)  
 HttpServletRequest request) {  
 String token = request.getHeader("token");  
 userInfoService.updateNickName(token, nickName);  
 **return** Result.*ok*();  
}

UserInfoService

// 修改用户昵称  
**void** updateNickName(String token, String nickName);

UserInfoServiceImpl

*/\*\*  
 \* 修改用户昵称  
 \** ***@param*** *token  
 \** ***@param*** *nickName  
 \*/*@Override  
**public void** updateNickName(String token, String nickName) {  
 String userId = JwtHelper.*getUserId*(token);  
 UserInfo userInfo = baseMapper.selectById(userId);  
 userInfo.setNickName(nickName);  
 baseMapper.updateById(userInfo);  
}

修改用户头像

**前端传入参数：**token，nickName（用户昵称）

**请求方式：**post

**请求url：**/auth/update/nickName/{nickName }

**后端方法：**Result updateUserNickName(HttpServletRequest request, @PathVariable String nickName)

**后端返回数据参数：**Result.ok()

UserInfoApi

// 用户头像上传  
@ApiOperation("用户头像上传")  
@PostMapping("/auth/upload/avatar")  
**public** Result uploadAvatar(  
 @ApiParam(name = "file", value = "头像文件", required = **true**)  
 MultipartFile file,  
  
 @ApiParam(name = "request", value = "request", required = **true**)  
 HttpServletRequest request) {  
  
 String token = request.getHeader("token");  
 userInfoService.uploadAvatar(token, file);  
 **return** Result.*ok*();  
}

UserInfoService

// 用户头像上传  
**void** uploadAvatar(String token, MultipartFile file);

UserInfoServiceImpl

*/\*\*  
 \* 用户头像上传  
 \** ***@param*** *token  
 \** ***@param*** *file  
 \*/*@Override  
**public void** uploadAvatar(String token, MultipartFile file) {  
 **try** {  
 String userId = JwtHelper.*getUserId*(token);  
 // 查询数据库  
 UserInfo userInfo = baseMapper.selectById(userId);  
  
 // 先拿到原来的头像，等新头像写入到数据库后再删除旧的头像  
 String oldAvatar = userInfo.getAvatar();  
  
 // 写入到fastDFS中  
 String filename = file.getOriginalFilename();  
 String[] upload = FastDFSUtil.*upload*(file.getBytes(), filename);  
 String avatarUrl = ImageUtil.*compoundUrl*(upload); // 头像图片访问地址  
  
 // 写入数据库中  
 userInfo.setAvatar(avatarUrl);  
 baseMapper.updateById(userInfo);  
  
 // 如果头像不等于空那么还要删除原本的头像  
 **if** (!StringUtils.*isEmpty*(oldAvatar)) {  
 // 取得图片 服务器协议+ip group fastDFS中的路径  
 String[] strings = ImageUtil.*splitUrl*(oldAvatar);  
 // 传入group和fastDFS中的路径 删除fastDFS中的图片  
 FastDFSUtil.*delete*(strings[1], strings[2]);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
}

密码修改

**前端传入参数：**token

**请求方式：**post

**请求url：**

**后端方法：**

**后端返回数据参数：**Result.ok()

UserInfoApi

UserInfoService

UserInfoServiceImpl

邮箱改绑

**前端传入参数：**token

**请求方式：**post

**请求url：**

**后端方法：**

**后端返回数据参数：**Result.ok()

UserInfoApi

UserInfoService

UserInfoServiceImpl

手机号绑定

**前端传入参数：**token

**请求方式：**post

**请求url：**

**后端方法：**

**后端返回数据参数：**Result.ok()

UserInfoApi

UserInfoService

UserInfoServiceImpl

手机号解绑

**前端传入参数：**token

**请求方式：**post

**请求url：**

**后端方法：**

**后端返回数据参数：**Result.ok()

UserInfoApi

UserInfoService

UserInfoServiceImpl

微信绑定

微信绑定二维码

获取微信绑定二维码，接口已经实现，直接调用，传入type为1

微信扫码回调接口

**前端传入参数：**code, state

**请求方式：**RequestMapper

**请求url：**/binding/callback

**后端方法：**public Result bindingCallback(String code, String state)

**后端返回数据参数：**Result.ok()

WeChatApi

// 微信绑定回调函数  
@ApiOperation("微信绑定回调函数")  
@RequestMapping("/binding/callback")  
**public** Result bindingCallback(  
 @ApiParam(name = "code", value = "从微信获取accessToken的票据", required = **true**)  
 String code,  
  
 @ApiParam(name = "state", value = "把获取到的accessToken存入redis中的key", required = **true**)  
 String state  
) {  
  
 Boolean flag = weChatService.bindingCallback(code, state);  
 **return** flag ? Result.*ok*("扫码成功，请在网页上确认绑定！") : Result.*ok*("扫码失败！");  
}

WeChatService

// 微信绑定回调函数  
Boolean bindingCallback(String code, String state);

WeChatServiceImpl

*/\*\*  
 \* 微信绑定回调函数  
 \** ***@param*** *code  
 \** ***@param*** *state  
 \** ***@return*** *\*/*@Override  
**public** Boolean bindingCallback(String code, String state) {  
 **try** {  
 Map<String, String> map = **new** HashMap<>();  
 // 对state（base64编码并加密后的uuid）进行解密解码  
 state = Base64.*decode*(state);  
  
 // 因为每次的code不一样，所以拼接通用请求地址接收内存空间  
 StringBuffer baseAccessTokenUrl = **new** StringBuffer()  
 .append("https://api.weixin.qq.com/sns/oauth2/access\_token")  
 .append("?appid=%s")  
 .append("&secret=%s")  
 .append("&code=%s")  
 .append("&grant\_type=authorization\_code"); // 固定参数  
 // 格式化请求地址  
 String accessTokenUrl = String.*format*(baseAccessTokenUrl.toString(),  
 ConstantWxPropertiesUtil.*WX\_OPEN\_APP\_ID*,  
 ConstantWxPropertiesUtil.*WX\_OPEN\_APP\_SECRET*,  
 code);  
 // 向获取accessToken的url地址发送请求  
 String jsonString = HttpClientUtils.*get*(accessTokenUrl);  
 JSONObject jsonObject = JSONObject.*parseObject*(jsonString);  
 String access\_token = jsonObject.getString("access\_token");  
 String openid = jsonObject.getString("openid");  
  
 // 装入map中，然后再写入redis中  
 map.put("access\_token", access\_token);  
 map.put("openid", openid);  
 redisTemplate.opsForValue().set(state, map, 120, TimeUnit.*MINUTES*);  
 **return true**;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 **return false**;  
 }  
}

绑定扫码轮询

**前端传入参数：**state（扫码查询凭证）

**请求方式：**post

**请求url：**/auth/polling/binding/{state}

**后端方法：**public Result pollingBinding(@PathVariable String state)

**后端返回数据参数：**Result.ok(bool)

WeChatApi

// 微信绑定轮询  
@ApiOperation("微信绑定轮询")  
@PostMapping("/auth/polling/binding/{state}")  
**public** Result pollingBinding(  
 @ApiParam(name = "state", value = "扫码查询凭证", required = **true**)  
 @PathVariable String state) {  
  
 Boolean flag = weChatService.pollingBinding(state);  
 **return** Result.*ok*(flag);  
}

WeChatService

// 微信绑定轮询  
Boolean pollingBinding(String state);

WeChatServiceImpl

*/\*\*  
 \* 微信绑定轮询  
 \** ***@param*** *state  
 \** ***@return*** *\*/*@Override  
**public** Boolean pollingBinding(String state) {  
 // 对state（base64编码并加密后的uuid）进行解密解码  
 state = Base64.*decode*(state);  
 Object o = redisTemplate.opsForValue().get(state);  
 **return** o == **null** ? **false** : **true**;  
}

确认绑定

**前端传入参数：**token， state

**请求方式：**post

**请求url：**/auth/confirm/weChat/binding/{state}

**后端方法：**public Result confirmWeChatBinding(@PathVariable String state, HttpServletRequest request)

**后端返回数据参数：**Result.ok(bool)

WeChatApi

// 确认进行微信绑定  
@ApiOperation("确认进行微信绑定")  
@PostMapping("/auth/confirm/weChat/binding/{state}")  
**public** Result confirmWeChatBinding(  
 @ApiParam(name = "state", value = "微信绑定凭证", required = **true**)  
 @PathVariable String state,  
  
 @ApiParam(name = "request", value = "request", required = **true**)  
 HttpServletRequest request) {  
  
 // 获得用户id  
 String token = request.getHeader("token");  
 Boolean flag = weChatService.confirmWeChatBinding(token, state);  
 **return** Result.*ok*(flag);  
}

WeChatService

// 确认进行微信绑定  
Boolean confirmWeChatBinding(String token, String state);

WeChatServiceImpl

*/\*\*  
 \* 确认进行微信绑定  
 \** ***@param*** *token  
 \** ***@param*** *state  
 \** ***@return*** *\*/*@Override  
**public** Boolean confirmWeChatBinding(String token, String state) {  
 **try** {  
 String userId = JwtHelper.*getUserId*(token);  
 // 对state（base64编码并加密后的uuid）进行解密解码  
 state = Base64.*decode*(state);  
 // 根据state查询出openid和account\_token  
 Map<String, String> map = (Map<String, String>) redisTemplate.opsForValue().get(state);  
 String access\_token = map.get("access\_token");  
 String openid = map.get("openid");  
  
 // 绑定到数据库  
 UserInfo userInfo = userInfoService.getById(userId);  
 userInfo.setOpenid(openid); // 设置openid  
 // 如果用户没有设置昵称，那么就把微信昵称当作用户昵称  
 **if** (StringUtils.*isEmpty*(userInfo.getNickName())) {  
 // 请求微信获取用户信息  
 StringBuffer baseUserInfoUrl = **new** StringBuffer()  
 .append("https://api.weixin.qq.com/sns/userinfo")  
 .append("?access\_token=%s")  
 .append("&openid=%s")  
 .append("&lang=zh\_CN");  
 String getUserInfoUrl = String.*format*(baseUserInfoUrl.toString(), access\_token, openid);  
  
 String resultInfo = HttpClientUtils.*get*(getUserInfoUrl);  
 JSONObject resultUserInfoJson = JSONObject.*parseObject*(resultInfo);  
 // 解析用户信息  
 String nickname = resultUserInfoJson.getString("nickname"); // 用户昵称  
 //String headimgurl = resultUserInfoJson.getString("headimgurl"); //用户头像  
  
 userInfo.setNickName(nickname);  
 }  
 // 更新数据库  
 userInfoService.updateById(userInfo);  
  
 **return true**;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 **return false**;  
 }  
}

微信解绑

**前端传入参数：**token

**请求方式：**post

**请求url：**/auth/unWechatBinding

**后端方法：**public Result unWechatBinding(HttpServletRequest request)

**后端返回数据参数：**Result.ok()

UserInfoApi

// 解除微信绑定  
@ApiOperation("解除微信绑定")  
@PostMapping("/auth/unWechatBinding")  
**public** Result unWechatBinding(  
 @ApiParam(name = "request", value = "HttpServletRequest", required = **true**)  
 HttpServletRequest request) {  
 String token = request.getHeader("token");  
 userInfoService.unWechatBinding(token);  
 **return** Result.*ok*();  
}

UserInfoService

// 解除微信绑定  
**void** unWechatBinding(String token);

UserInfoServiceImpl

*/\*\*  
 \* 解除微信绑定  
 \** ***@param*** *token  
 \*/*@Override  
**public void** unWechatBinding(String token) {  
 String userId = JwtHelper.*getUserId*(token);  
 // 根据id设置openid  
 baseMapper.setOpenidById(userId, **null**);  
}

UserInfoMapper

// 根据id设置openid  
**void** setOpenidById(@Param("id") String id, @Param("openid") String openid);

UserInfoMapper.xml

<**update** id="setOpenidById">  
 UPDATE  
 user\_info  
 SET  
 openid=#{openid}  
 WHERE  
 id=#{id}  
</**update**>

order\_item：购物车

主要业务

获取订单项（订单项对应的商品信息）

删除订单项

控制层

OrderItemApi

url: /api/order/orderItem

获取订单项（包括订单项对应的商品信息）

**前端传入参数：**token

**请求方式：**post

**请求url：**/auth/getOrderItem

**后端方法：**Result getOrderItmeByToken(HttpServletRequest request)

**后端返回数据参数：**订单项和订单项下的商品信息

删除订单项

**前端传入参数：**token, id（订单项id）

**请求方式：**post

**请求url：**/auth/remove/orderItem/{id}

**后端方法：**Result removeOrdeItemById(HttpServletRequest request, @PathVariable String id)

**后端返回数据参数：**Result.ok()

forebudy：下单页

pay：支付页

pay\_succeed：支付成功页

order\_info：订单页

confirm\_receipt：确认收货页

[trade\_successfully](javascript:;)：交易成功页

review：产品评价页