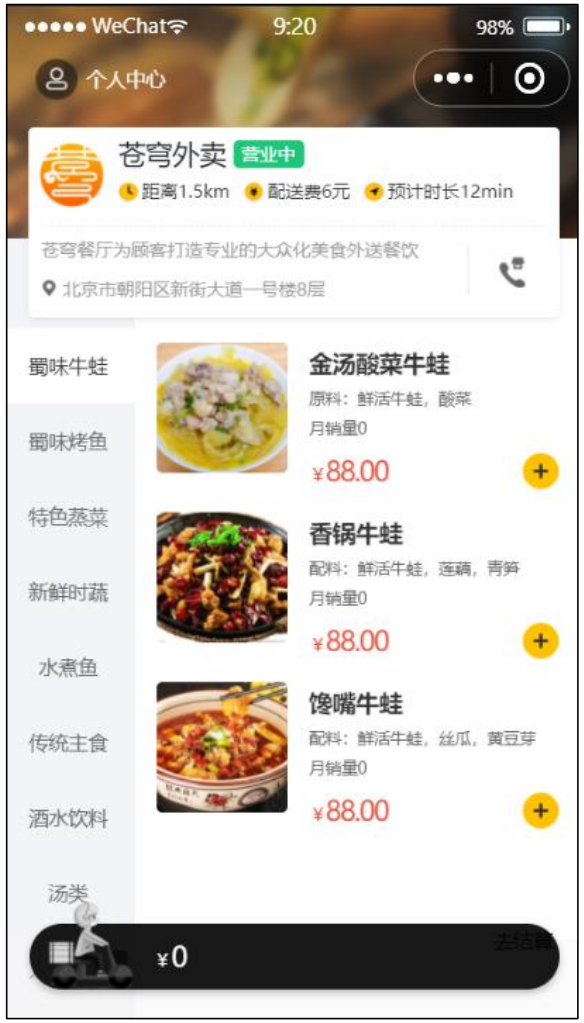
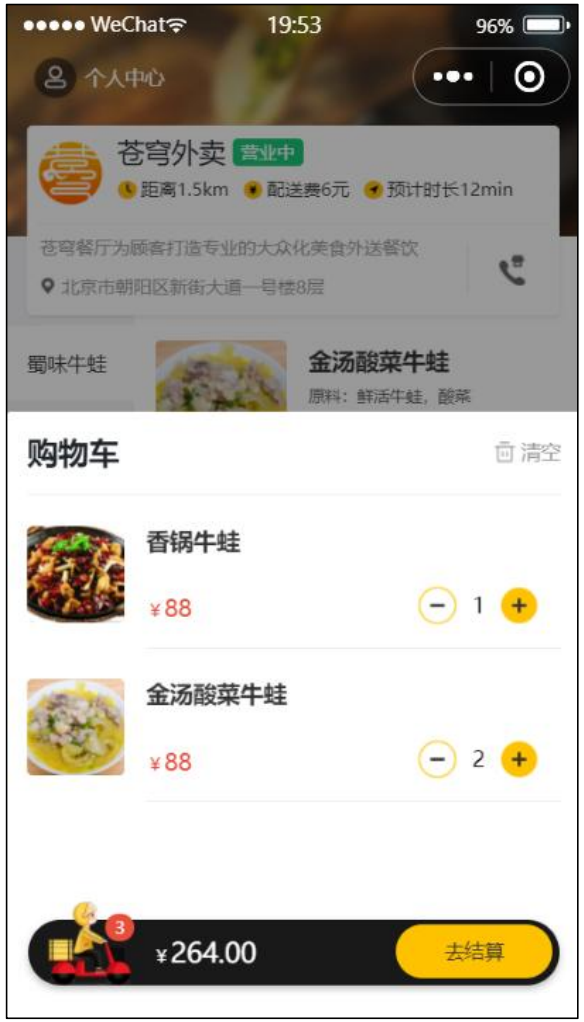
缓存商品、购物车

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌


WeChat 9:20 98%
个人中心
苍穹外卖 营业中
距离1.5km 配送费6元 预计时长12min
苍穹餐厅为顾客打造专业的大众化美食外送餐饮
北京市朝阳区新街大道一号楼8层
蜀味牛蛙
蜀味烤鱼
特色蒸菜
新鲜时蔬
水煮鱼
传统主食
酒水饮料
汤类
金汤酸菜牛蛙
原料：鲜活牛蛙，酸菜
月销量0
¥88.00
香锅牛蛙
配料：鲜活牛蛙，莲藕，青笋
月销量0
¥88.00
馋嘴牛蛙
配料：鲜活牛蛙，丝瓜，黄豆芽
月销量0
¥88.00
¥0
去结算


WeChat 9:20 98%
个人中心
苍穹外卖 营业中
距离1.5km 配送费6元 预计时长12min
苍穹餐厅为顾客打造专业的大众化美食外送餐饮
北京市朝阳区新街大道一号楼8层
新鲜时蔬
水煮鱼
传统主食
酒水饮料
汤类
人气套餐
商务套餐
人气套餐A计划
人气套餐A计划
月销量0
¥45.00
¥0
去结算


WeChat 19:53 96%
个人中心
苍穹外卖 营业中
距离1.5km 配送费6元 预计时长12min
苍穹餐厅为顾客打造专业的大众化美食外送餐饮
北京市朝阳区新街大道一号楼8层
蜀味牛蛙
金汤酸菜牛蛙
原料：鲜活牛蛙，酸菜
购物车
清空
香锅牛蛙
¥88
1
金汤酸菜牛蛙
¥88
2
3
¥264.00
去结算

目录

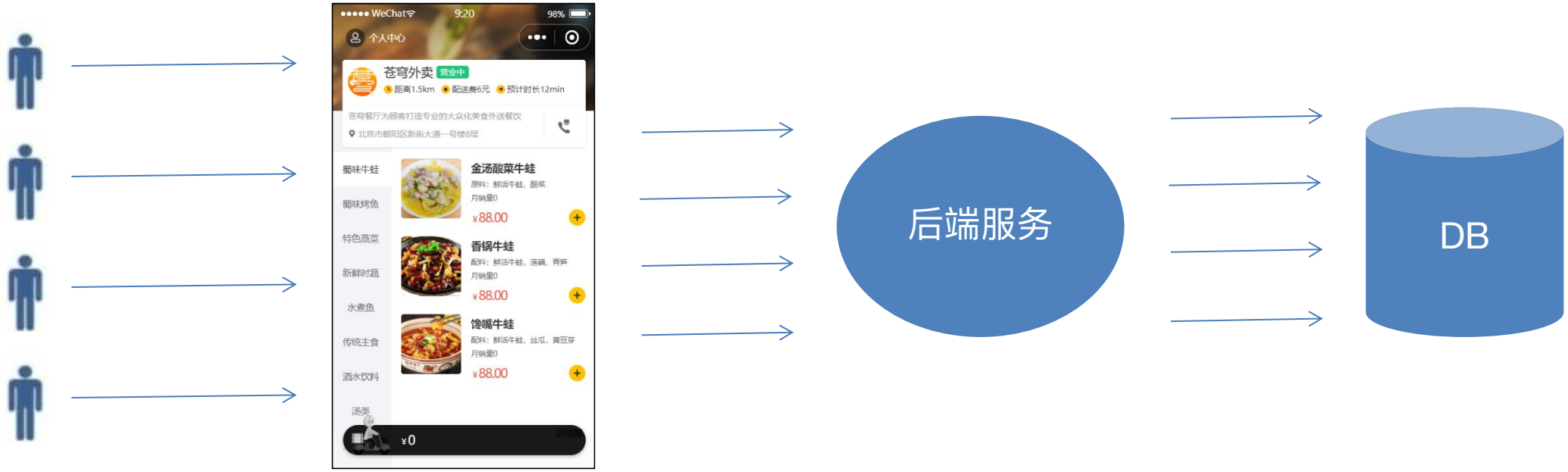
Contents

缓存菜品

- 问题说明
- 实现思路
- 代码开发
- 功能测试

问题说明

用户端小程序展示的菜品数据都是通过查询数据库获得，如果用户端访问量比较大，数据库访问压力随之增大。



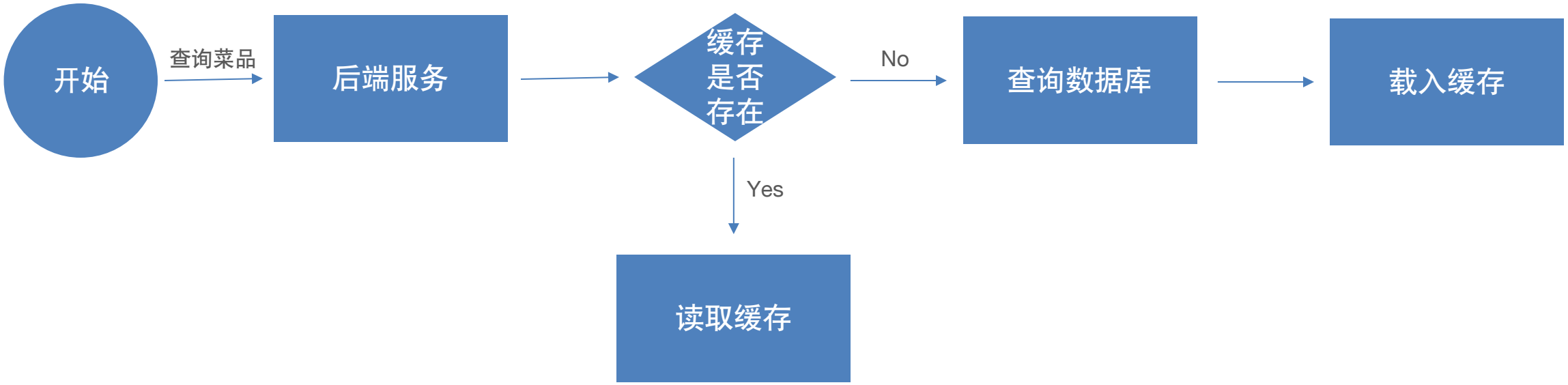
结果：系统响应慢、用户体验差

缓存菜品

- 问题说明
- 实现思路
- 代码开发
- 功能测试

实现思路

通过Redis来缓存菜品数据，减少数据库查询操作。

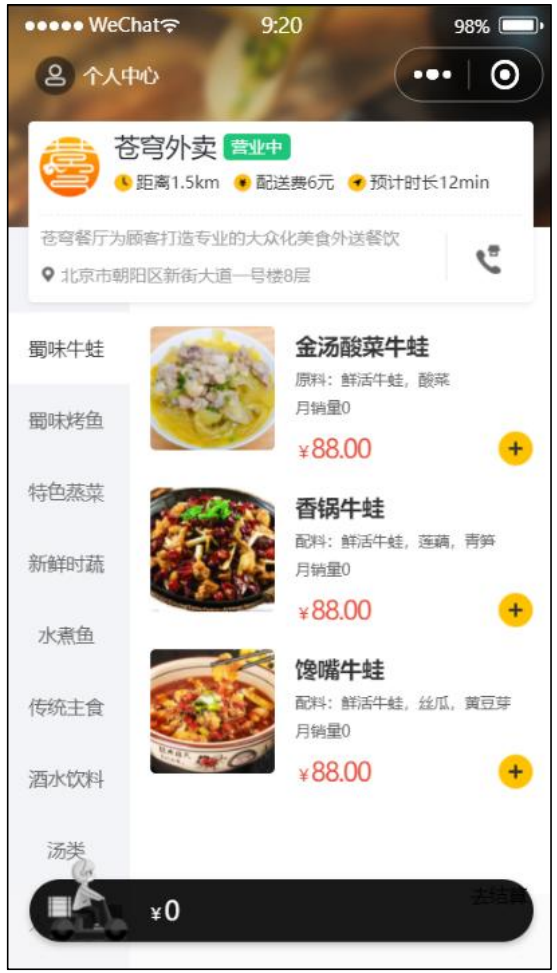


实现思路

缓存逻辑分析：

- 每个分类下的菜品保存一份缓存数据
- 数据库中菜品数据有变更时清理缓存数据

key	value
dish_1	string(…)
dish_2	string(…)
dish_3	string(…)



缓存菜品

- 问题说明
- 实现思路
- 代码开发
- 功能测试

代码开发

修改用户端接口 DishController 的 list 方法，加入缓存处理逻辑：

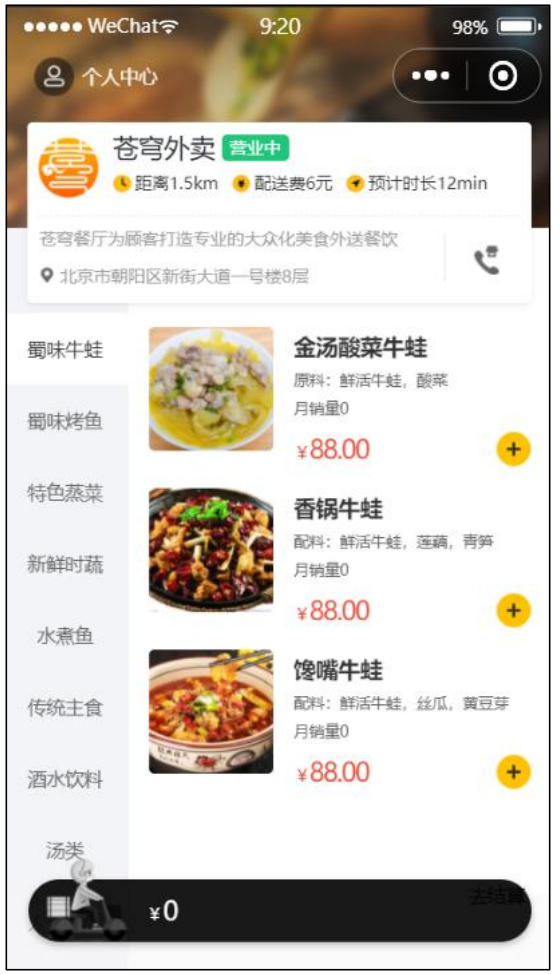
```
//构造redis缓存key，规则为：dish_分类id
String key = "dish_" + categoryId;

//查询redis中是否有缓存数据
List<DishVO> list = (List<DishVO>) redisTemplate.opsForValue().get(key);

//存在缓存数据，直接返回给前端
if(list != null && list.size() > 0){
    return Result.success(list);
}

Dish dish = new Dish();
dish.setCategoryId(categoryId);
dish.setStatus(StatusConstant.ENABLE);//查询起售中的菜品

list = dishService.listWithFlavor(dish);

//将查询到的数据载入缓存
redisTemplate.opsForValue().set(key, list);

return Result.success(list);
```

DishController

代码开发

修改管理端接口 DishController 的相关方法，加入清理缓存的逻辑，需要改造的方法：

- 新增菜品
- 修改菜品
- 批量删除菜品
- 起售、停售菜品



代码开发

抽取清理缓存的方法：

```
/**
 * 清理缓存数据
 */
private void cleanCache(String pattern){
    Set keys = redisTemplate.keys(pattern);
    redisTemplate.delete(keys);
}
```

DishController

代码开发

调用清理缓存的方法，保证数据一致性：

```
@Autowired
private RedisTemplate redisTemplate;

/**
 * 新增菜品
 * @param dishDTO
 * @return
 */
@PostMapping
@ApiOperation("新增菜品")
public Result save(@RequestBody DishDTO dishDTO){
    log.info("新增菜品：{}", dishDTO);
    dishService.saveWithFlavor(dishDTO);

    Long categoryId = dishDTO.getCategoryId();
    String key = "dish_" + categoryId;
    cleanCache(key);

    return Result.success();
}
```

DishController

代码开发

调用清理缓存的方法，保证数据一致性：

```
/**
 * 菜品批量删除
 * @param ids
 * @return
 */
@DeleteMapping
@ApiOperation("菜品批量删除")
public Result delete(@RequestParam List<Long> ids){
    log.info("菜品批量删除：{}", ids);
    dishService.deleteBatch(ids);

    //删除所有菜品的缓存数据
    cleanCache("dish_*");

    return Result.success();
}
```

DishController

代码开发

调用清理缓存的方法，保证数据一致性：

```
/**
 * 修改菜品
 * @param dishDTO
 * @return
 */
@PutMapping
@ApiOperation("修改菜品")
public Result<String> update(@RequestBody DishDTO dishDTO){
    log.info("修改菜品：{}", dishDTO);
    dishService.updateWithFlavor(dishDTO);

    //删除所有菜品的缓存数据
    cleanCache("dish_*");

    return Result.success();
}
```

DishController

代码开发

调用清理缓存的方法，保证数据一致性：

```
/**
 * 菜品起售停售
 * @param status
 * @param id
 * @return
 */
@PostMapping("/status/{status}")
@ApiOperation("菜品起售停售")
public Result<String> startOrStop(@PathVariable Integer status, Long id){
    dishService.startOrStop(status,id);

    //删除所有菜品的缓存数据
    cleanCache("dish_*");

    return Result.success();
}
```

DishController

缓存菜品

- 问题说明
- 实现思路
- 代码开发
- 功能测试

功能测试

可以通过如下方式进行测试：

- 查看控制台sql
- 前后端联调
- 查看Redis中的缓存数据

目录

Contents

缓存套餐

- Spring Cache
- 实现思路
- 代码开发
- 功能测试

Spring Cache

Spring Cache 是一个框架，实现了基于注解的缓存功能，只需要简单地加一个注解，就能实现缓存功能。

Spring Cache 提供了一层抽象，底层可以切换不同的缓存实现，例如：

- EHCache
- Caffeine
- Redis

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
    <version>2.7.3</version>
</dependency>
```

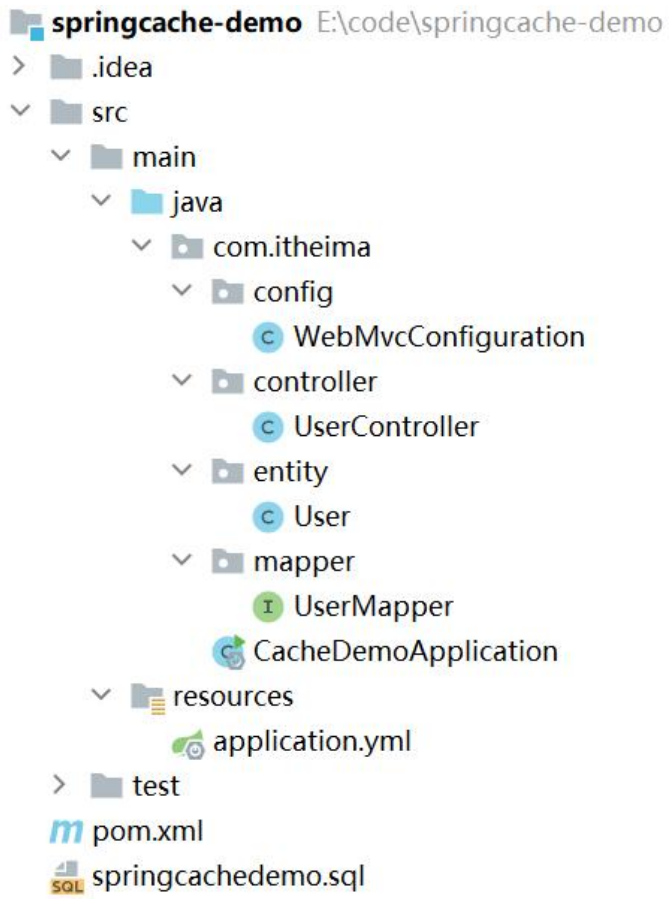
Spring Cache

常用注解：

注解	说明
@EnableCaching	开启缓存注解功能，通常加在启动类上
@Cacheable	在方法执行前先查询缓存中是否有数据，如果有数据，则直接返回缓存数据；如果没有缓存数据，调用方法并将方法返回值放到缓存中
@CachePut	将方法的返回值放到缓存中
@CacheEvict	将一条或多条数据从缓存中删除

Spring Cache

入门案例：导入资料中的初始工程，在此基础上加入Spring Cache注解即可



缓存套餐

- Spring Cache
- 实现思路
- 代码开发
- 功能测试

实现思路

具体的实现思路如下：

- 导入Spring Cache和Redis相关maven坐标
- 在启动类上加入@EnableCaching注解，开启缓存注解功能
- 在用户端接口SetmealController的 list 方法上加入@Cacheable注解
- 在管理端接口SetmealController的 save、delete、update、startOrStop等方法上加入CacheEvict注解

缓存套餐

- Spring Cache
- 实现思路
- 代码开发
- 功能测试

代码开发

在用户端接口SetmealController的 list 方法上加入@Cacheable注解：

```
/**
 * 条件查询
 *
 * @param categoryId
 * @return
 */
@GetMapping("/list")
@ApiOperation("根据分类id查询套餐")
@Cacheable(cacheNames = "setmealCache",key = "#categoryId")
public Result<List<Setmeal>> list(Long categoryId) {
    Setmeal setmeal = new Setmeal();
    setmeal.setCategoryId(categoryId);
    setmeal.setStatus(StatusConstant.ENABLE);

    List<Setmeal> list = setmealService.list(setmeal);
    return Result.success(list);
}
```

代码开发

在管理端接口SetmealController的 save、delete、update、startOrStop等方法上加入CacheEvict注解：

```
@PostMapping
@ApiOperation("新增套餐")
@CacheEvict(cacheNames = "setmealCache",key = "#setmealDTO.categoryId")
public Result save(@RequestBody SetmealDTO setmealDTO) {
    setmealService.saveWithDish(setmealDTO);
    return Result.success();
}

@DeleteMapping
@ApiOperation("批量删除套餐")
@CacheEvict(cacheNames = "setmealCache",allEntries = true)
public Result delete(@RequestParam List<Long> ids){
    setmealService.deleteBatch(ids);
    return Result.success();
}
```

缓存套餐

- Spring Cache
- 实现思路
- 代码开发
- 功能测试

功能测试

通过前后端联调方式来进行测试，同时观察redis中缓存的套餐数据。

目录

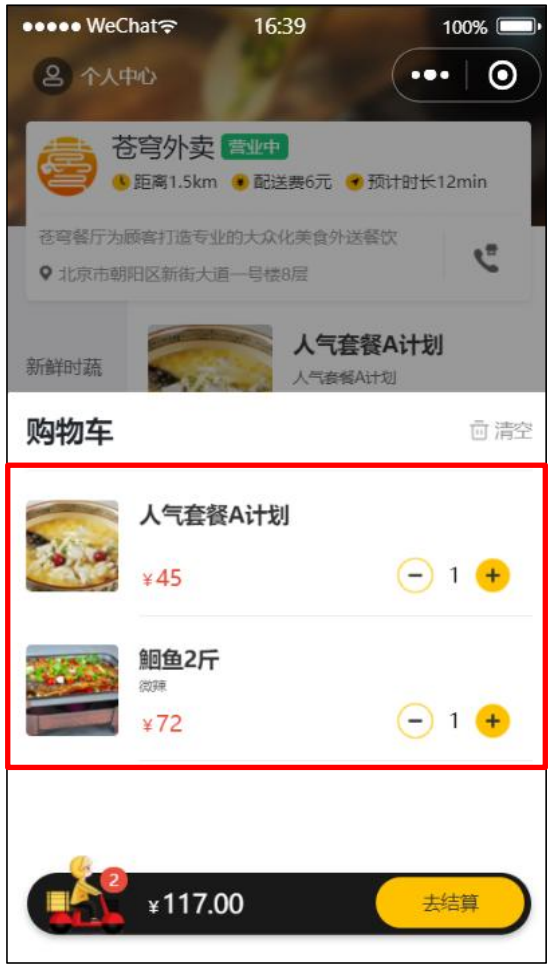
Contents

添加购物车

- 需求分析和设计
- 代码开发
- 功能测试

需求分析和设计

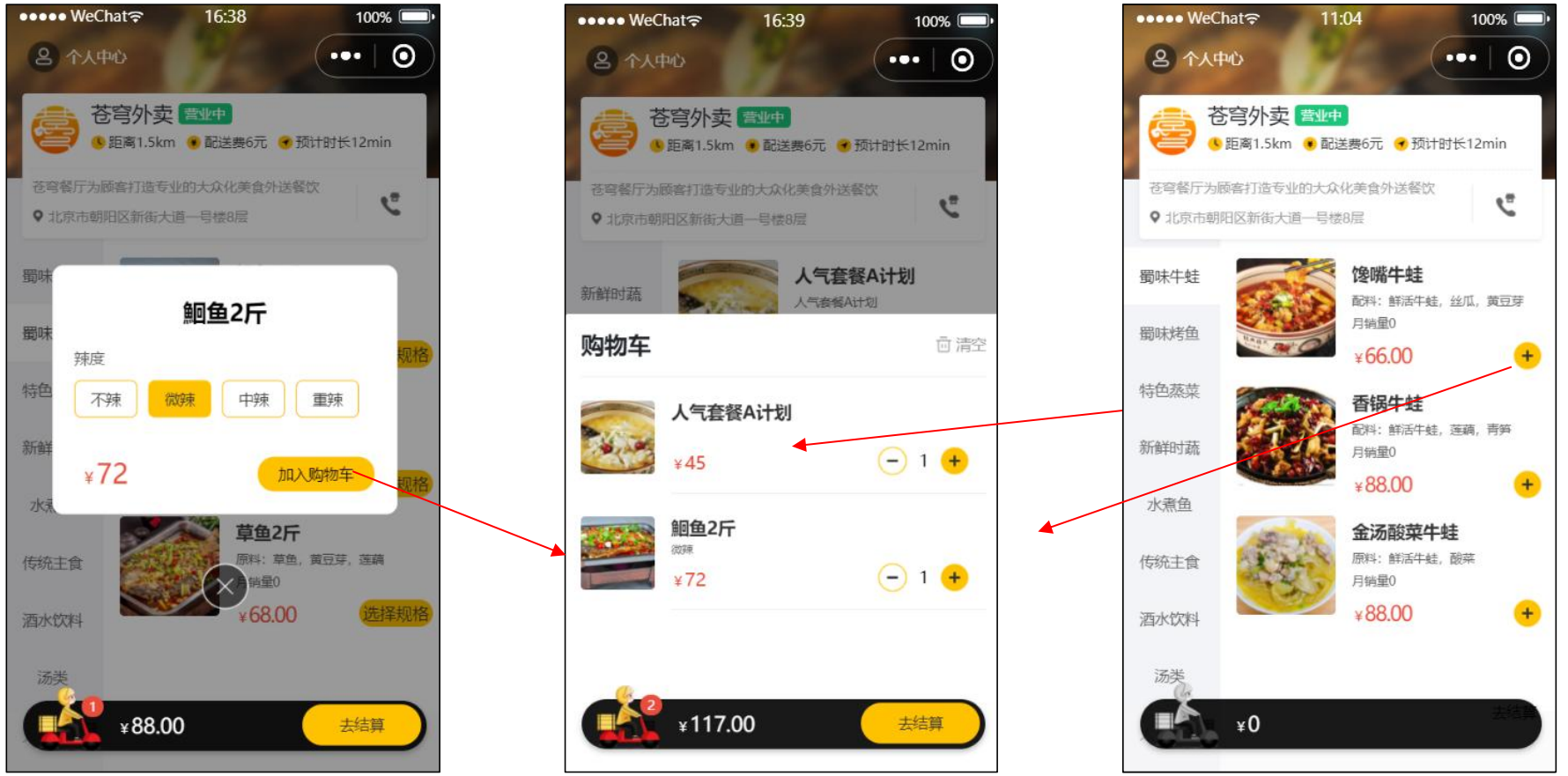
产品原型：



生活中的购物车：用于暂时存放所选商品的一种手推车

需求分析和设计

产品原型：



需求分析和设计

接口设计：

- 请求方式：POST
- 请求路径：/user/shoppingCart/add
- 请求参数：套餐id、菜品id、口味
- 返回结果：code、data、msg

需求分析和设计

接口设计：

基本信息

Path： /user/shoppingCart/add

Method： POST

接口描述：

请求参数

Headers

参数名称	参数值	是否必须	示例	备注
Content-Type	application/json	是		

Body

名称	类型	是否必须	默认值	备注	其他信息
dishFlavor	string	非必须		口味	
dishId	integer	非必须		菜品id	format: int64
setmealId	integer	非必须		套餐id	format: int64

返回数据

名称	类型	是否必须	默认值	备注	其他信息
code	integer	必须			format: int32
data	string	非必须			
msg	string	非必须			

需求分析和设计

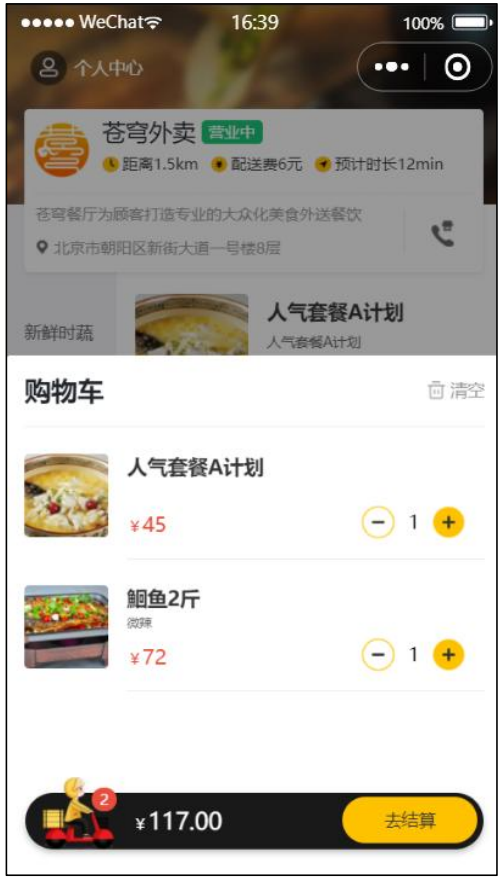
数据库设计：

- 作用：暂时存放所选商品的地方
- 选的什么商品
- 每个商品都买了几个
- 不同用户的购物车需要区分开

需求分析和设计

数据库设计（shopping_cart表）：

字段名	数据类型	说明	备注
id	bigint	主键	自增
name	varchar(32)	商品名称	冗余字段
image	varchar(255)	商品图片路径	冗余字段
user_id	bigint	用户id	逻辑外键
dish_id	bigint	菜品id	逻辑外键
setmeal_id	bigint	套餐id	逻辑外键
dish_flavor	varchar(50)	菜品口味	
number	int	商品数量	
amount	decimal(10,2)	商品单价	冗余字段
create_time	datetime	创建时间	



添加购物车

- 需求分析和设计
- 代码开发
- 功能测试

代码开发

根据添加购物车接口的参数设计DTO：

Body

名称	类型	是否必须	默认值	备注	其他信息
dishFlavor	string	非必须		口味	
dishId	integer	非必须		菜品id	format: int64
setmealId	integer	非必须		套餐id	format: int64

```
public class ShoppingCartDTO implements Serializable {

    private Long dishId;
    private Long setmealId;
    private String dishFlavor;

}
```

代码开发

根据添加购物车接口创建ShoppingCartController：

```
@RestController
@RequestMapping("/user/shoppingCart")
@Slf4j
@Api(tags = "C端-购物车接口")
public class ShoppingCartController {

    @Autowired
    private ShoppingCartService shoppingCartService;

    /**
     * 添加购物车
    * @param shoppingCartDTO
     * @return
     */
    @PostMapping("/add")
    @ApiOperation("添加购物车")
    public Result add(@RequestBody ShoppingCartDTO shoppingCartDTO){
        log.info("添加购物车：{}", shoppingCartDTO);
        shoppingCartService.addShoppingCart(shoppingCartDTO);
        return Result.success();
    }
}
```

代码开发

创建ShoppingCartService接口：

```
public interface ShoppingCartService {

    /**
     * 添加购物车
    * @param shoppingCartDTO
     */
    void addShoppingCart(ShoppingCartDTO shoppingCartDTO);
}
```

代码开发

创建ShoppingCartServiceImpl实现类，并实现add方法：

```
@Service
public class ShoppingCartServiceImpl implements ShoppingCartService {
    @Autowired
    private ShoppingCartMapper shoppingCartMapper;
    @Autowired
    private DishMapper dishMapper;
    @Autowired
    private SetmealMapper setmealMapper;

    /**
     * 添加购物车
     * @param shoppingCartDTO
     */
    public void addShoppingCart(ShoppingCartDTO shoppingCartDTO) {
        ShoppingCart shoppingCart = new ShoppingCart();
        BeanUtils.copyProperties(shoppingCartDTO, shoppingCart);
        //只能查询自己的购物车数据
      shoppingCart.setUserId(BaseContext.getCurrentId());

        //判断当前商品是否在购物车中
      List<ShoppingCart> shoppingCartList = shoppingCartMapper.list(shoppingCart);
```

代码开发

创建ShoppingCartServiceImpl实现类，并实现add方法：

```
        if (shoppingCartList != null && shoppingCartList.size() == 1) {
            //如果已经存在，就更新数量，数量加1
            shoppingCart = shoppingCartList.get(0);
            shoppingCart.setNumber(shoppingCart.getNumber() + 1);
            shoppingCartMapper.updateNumberById(shoppingCart);
        } else {
            //如果不存在，插入数据，数量就是1
          Long dishId = shoppingCartDTO.getDishId();
            if (dishId != null) {
                //添加到购物车的是菜品
             Dish dish = dishMapper.getById(dishId);
                shoppingCart.setName(dish.getName());
                shoppingCart.setImage(dish.getImage());
                shoppingCart.setAmount(dish.getPrice());
            } else {
                //添加到购物车的是套餐
             Setmeal setmeal = setmealMapper.getById(shoppingCartDTO.getSetmealId());
                shoppingCart.setName(setmeal.getName());
                shoppingCart.setImage(setmeal.getImage());
                shoppingCart.setAmount(setmeal.getPrice());
            }
            shoppingCart.setNumber(1);
            shoppingCart.setCreateTime(LocalDateTime.now());
            shoppingCartMapper.insert(shoppingCart);
        }
    }
}
```

代码开发

创建ShoppingCartMapper接口：

```
@Mapper
public interface ShoppingCartMapper {
    /**
     * 条件查询
     * @param shoppingCart
     * @return
     */
    List<ShoppingCart> list(ShoppingCart shoppingCart);

    /**
     * 更新商品数量
     * @param shoppingCart
     */
    @Update("update shopping_cart set number = #{number} where id = #{id}")
    void updateNumberById(ShoppingCart shoppingCart);

    /**
     * 插入购物车数据
     * @param shoppingCart
     */
    @Insert("insert into shopping_cart (name, user_id, dish_id, setmeal_id, dish_flavor, number, amount, image, create_time) " +
            " values (#{name},#{userId},#{dishId},#{setmealId},#{dishFlavor},#{number},#{amount},#{image},#{createTime})")
    void insert(ShoppingCart shoppingCart);
}
```

代码开发

创建ShoppingCartMapper.xml：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.sky.mapper.ShoppingCartMapper">

    <select id="list" parameterType="ShoppingCart" resultType="ShoppingCart">
        select * from shopping_cart
        <where>
            <if test="userId != null">
                and user_id = #{userId}
            </if>
            <if test="dishId != null">
                and dish_id = #{dishId}
            </if>
            <if test="setmealId != null">
                and setmeal_id = #{setmealId}
            </if>
            <if test="dishFlavor != null">
                and dish_flavor = #{dishFlavor}
            </if>
        </where>
        order by create_time desc
    </select>

</mapper>
```

添加购物车

- 需求分析和设计
- 代码开发
- 功能测试

功能测试

可以通过如下方式进行测试：

- 查看控制台sql
- Swagger接口文档测试
- 前后端联调

目录

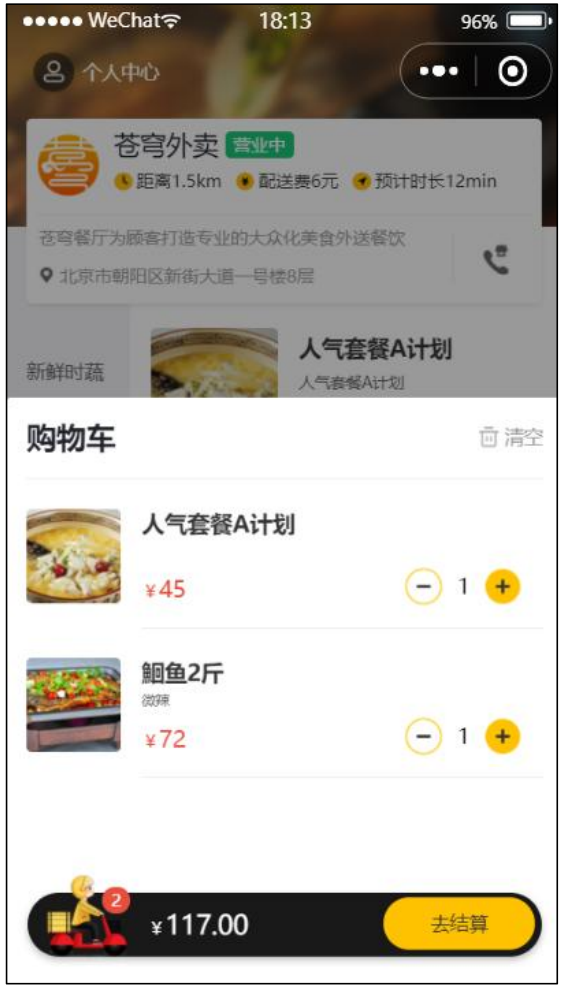
Contents

查看购物车

- 需求分析和设计
- 代码开发
- 功能测试

需求分析和设计

产品原型：



需求分析和设计

接口设计：

基本信息

Path： /user/shoppingCart/list

Method： GET

接口描述：

请求参数

返回数据

名称	类型	是否必须	默认值	备注	其他信息
code	number	必须			
msg	null	非必须			
data	object []	必须			item 类型: object
├─ id	number	必须			
├─ name	string	必须			
├─ userId	number	必须			
├─ dishId	null,number	必须			
├─ setmealId	number,null	必须			
├─ dishFlavor	string	必须			
├─ number	number	必须			
├─ amount	number	必须			
├─ image	string	必须			
├─ createTime	string	必须			

查看购物车

- 需求分析和设计
- 代码开发
- 功能测试

代码开发

在ShoppingCartController中创建查看购物车的方法：

```
/**
 * 查看购物车
 * @return
 */
@GetMapping("/list")
@ApiOperation("查看购物车")
public Result<List<ShoppingCart>> list(){
    return Result.success(shoppingCartService.showShoppingCart());
}
```

代码开发

在ShoppingCartService接口中声明查看购物车的方法：

```
/**
 * 查看购物车
 * @return
 */
List<ShoppingCart> showShoppingCart();
```

代码开发

在ShoppingCartServiceImpl中实现查看购物车的方法：

```
/**
 * 查看购物车
 * @return
 */
public List<ShoppingCart> showShoppingCart() {
    return shoppingCartMapper.list(ShoppingCart.builder().userId(BaseContext.getCurrentId()).build());
}
```

查看购物车

- 需求分析和设计
- 代码开发
- 功能测试

功能测试

可以通过接口文档进行测试，最后完成前后端联调测试即可

目录

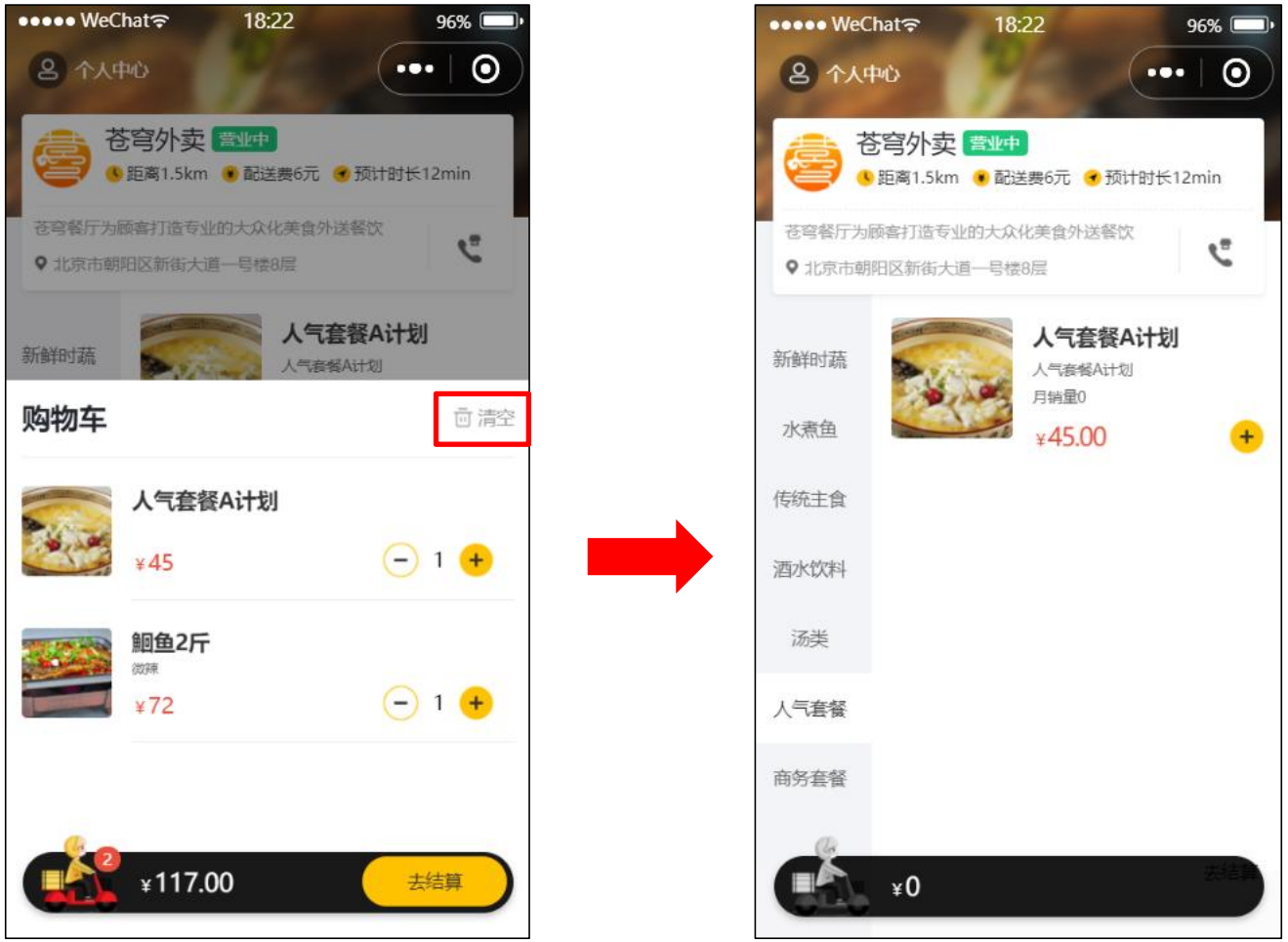
Contents

清空购物车

- 需求分析和设计
- 代码开发
- 功能测试

需求分析和设计

产品原型：



需求分析和设计

接口设计：

基本信息

Path： /user/shoppingCart/clean

Method： DELETE

接口描述：

请求参数

返回数据

名称	类型	是否必须	默认值	备注	其他信息
code	integer	必须			format: int32
data	string	非必须			
msg	string	非必须			

清空购物车

- 需求分析和设计
- 代码开发
- 功能测试

代码开发

在ShoppingCartController中创建清空购物车的方法：

```
/**
 * 清空购物车
 * @return
 */
@DeleteMapping("/clean")
@ApiOperation("清空购物车")
public Result<String> clean(){
    shoppingCartService.cleanShoppingCart();
    return Result.success();
}
```

代码开发

在ShoppingCartService接口中声明清空购物车的方法：

```
/**
 * 清空购物车
 */
void cleanShoppingCart();
```

代码开发

在ShoppingCartServiceImpl中实现清空购物车的方法：

```
/**
 * 清空购物车
 */
public void cleanShoppingCart() {
    shoppingCartMapper.deleteByUserId(BaseContext.getCurrentId());
}
```

代码开发

在ShoppingCartMapper接口中创建删除购物车数据的方法：

```
/**
 * 根据用户id删除购物车数据
 * @param userId
 */
@Delete("delete from shopping_cart where user_id = #{userId}")
void deleteByUserId(Long userId);
```

清空购物车

- 需求分析和设计
- 代码开发
- 功能测试

功能测试

通过Swagger接口文档进行测试，通过后再前后端联调测试即可

传智教育旗下高端IT教育品牌