

一、简介

- 1.1 项目描述
- 1.2 开发环境与工具

二、需求分析

- 2.1 数据库的搭建

三、技术架构

- 3.1 项目环境搭建
 - 3.1.1 pom.xml配置文件
 - 3.1.2 配置类
 - 3.1.3 过滤器

四、实现设计

- 4.1 前端页面的搭建
 - 4.1.1 管理端
 - 1) 登陆页面
 - 2) 目录页面
 - 3) 员工管理页面
 - 4) 分类管理页面
 - 5) 菜品/套餐管理页面
 - 6) 订单明细页面
 - 4.1.2 客户端
 - 1) 登陆页面
 - 2) 功能主页面
 - 3) 个人中心页面
 - 4) 地址管理页面
 - 5) 下单页面
 - 6) 支付成功页面
- 4.2 功能模块实现
 - 4.2.1 管理端
 - 1) 管理员登录
 - 2) 管理员退出
 - 3) 员工管理
 - a.添加管理员（员工）
 - b.员工信息分页查询
 - c.修改员工帐号状态及信息功能
 - d.公共字段填充
 - 4) 分类管理
 - a.删除菜品分类
 - b.自定义业务异常类
 - 5) 菜品管理
 - a.新增菜品
 - b.菜品信息的分页查询
 - c.修改菜品信息
 - d.批量管理菜品操作
 - 6) 套餐管理
 - 7) 订单明细
 - a.订单明细的分页查询
 - b.查看订单明细和修改订单状态
 - 4.2.2 移动端
 - 1) 用户登录
 - 2) 导入用户地址簿
 - 3) 菜品展示
 - 5) 下单

五、总结与思考

一、简介

1.1 项目描述

某企业为了方便员工用餐，为餐厅开发了一个订餐外卖系统，员工作为顾客可通过企业内联网使用该系统。系统用户为顾客、餐厅员工、餐厅经理等。

- 顾客：可以查看菜单、订餐（如果未登录，需先登录）、支付、投诉等。
- 餐厅员工：可以进行接受订单、生成付费请求、备餐和请求送餐。
- 餐厅经理：餐厅管理层，可以管理菜单、查看每日销售情况及简单数据统计、投诉。

顾客订餐过程如下：

1. 顾客请求查看系统显示的菜单；
2. 顾客选菜；
3. 系统显示订单和价格及可送餐时间；
4. 顾客确认订单，指定送餐时间、地点并支付；
5. 系统确认接受订单，给顾客的界面可显示订单状态（确认中、备餐中、送餐中及预计到达时间），同时发送相关订餐信息通知给餐厅员工。

1.2 开发环境与工具

OS: Windows 11

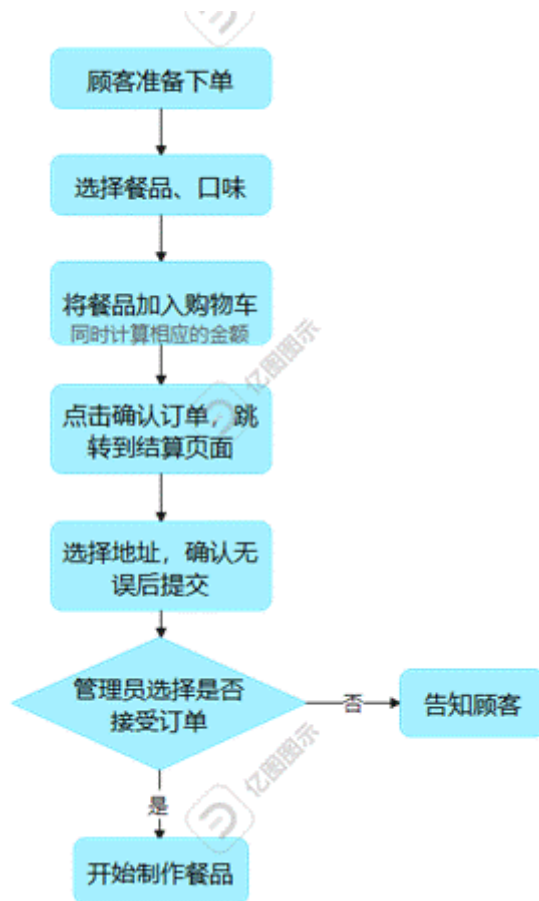
后端: Java 1.8, apache-maven-3.8.4, mysql-connector-java-8.0.31

前端: H5、Vue.js、ElementUI

二、需求分析

对于题目进行分析得，具体要求实现不同人员的功能。而最主要的人员是顾客以及餐厅管理员。在进行分析后，我们认为：由于该订餐外卖系统是用于企业内网以及员工餐厅的，其商家实际上只有承包公司一家，而且其窗口数量也是有限的。因此考虑对上述题目中的功能与人员进行简化：将其认为是一家商家，但是有很多的餐品种类。对于每一个窗口，应该都有餐厅员工，但是有些员工主要负责的是餐品的制作，而有些主要是管理订单。但是员工的权

限显然不可能很高。因此对其进行简化：将顾客生成订单，并将该订单的请求传递给餐厅经理。(如果该餐厅的规格比较大，那将考虑将权利下放给餐厅员工)



由于不确定时间是否足够，因此我们确定下最主要的消费的员工以及管理员的数据库的构建以及功能的确定。对于数据库的表结构的创建，需要与实际相结合：

- 在实际的下单过程中，我们会选择自己心仪的商家与餐品，并对份数以及口味进行选择；
- 同时在下栏的购物车中，可以显示已选择的餐品的详细信息（指份数、口味之类）以及总计金额等。还能够进行加减先前勾选的餐品的功能以及一键清空购物车；
- 在点击下单后，将会跳转到相应的结算页面。将会对消费者显示详细的点餐信息。同时，消费者将选择送餐的地址；最终将提交订单。
- 当订单选择好之后将会将该信息发送到管理端：管理员会选择是否接受这份订单。接收后应将餐品信息发送到相应的窗口的员工那进行管理。

而对于管理员，其主要的功能是：

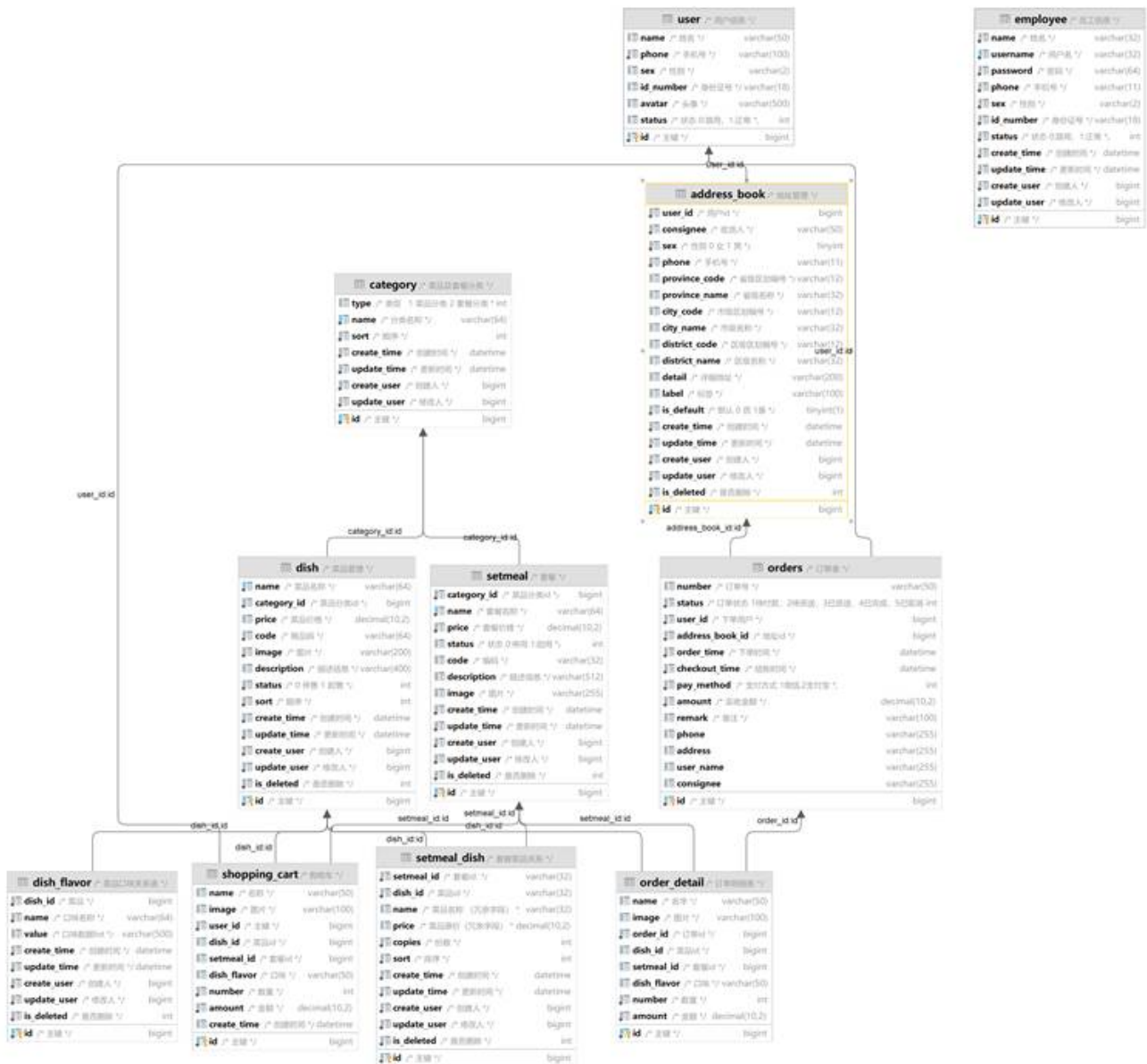
- 登入登出
- 对员工的信息进行管理(增删改查)
- 对餐品、套餐、口味、分类等进行管理
- 对订单进行管理

对于顾客的主要功能如下：

- 登入登出
- 点菜下单
- 地址管理
- 获知订单状态
- 历史订单显示

2.1 数据库的搭建

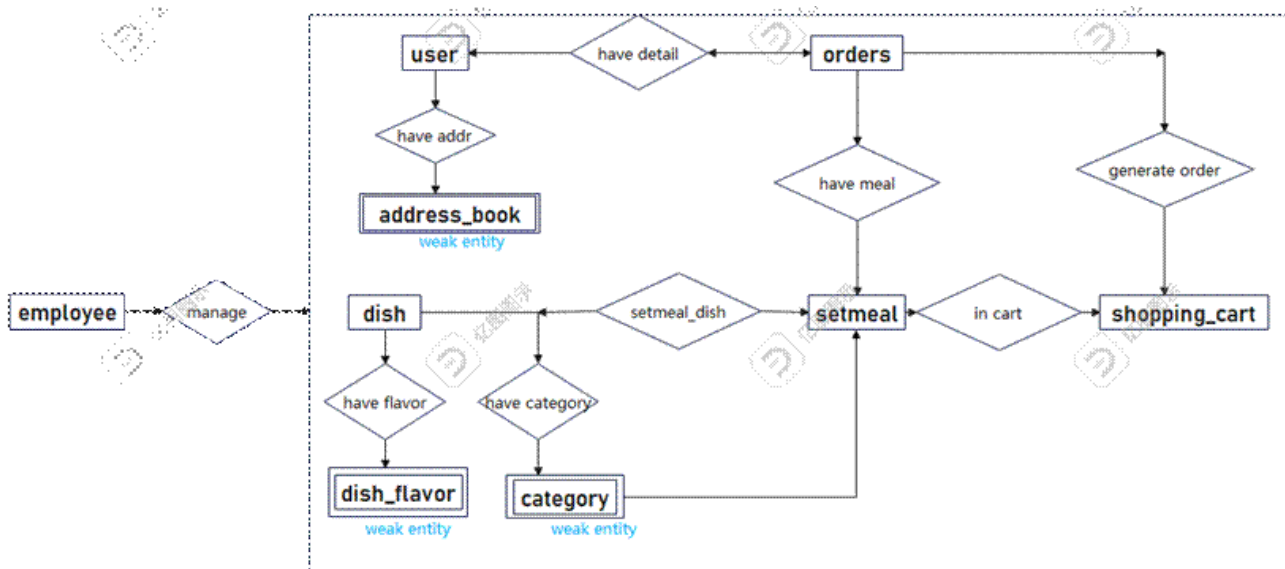
首先确定表结构，然后准备好相应的样例数据先输入。



在此我们确定了一共11个表结构，分别是：

- employee：管理员与员工信息表。
- user：顾客信息表。
- address_book：地址簿，每位顾客可以设置多个地址。
- dish：菜品管理表。
- dish_flavor：菜品关口味关系表。
- setmeal：套餐表。
- setmeal_dish：套餐菜品关系表。
- category：菜品以及套餐分类表。
- order：订单表，存放订单的状态以及顾客的地址。
- order_detail：订单中餐品的详细信息表。
- shopping_cart：购物车表，显示选中的菜品以及详细信息。

其相应的E-R图如下：



三、技术架构

3.1 项目环境搭建

后端：Java 1.8, apache-maven-3.8.4, mysql-connector-java-8.0.31

前端：H5、Vue.js、ElementUI

3.1.1 pom.xml配置文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.4.5</version>
11        <relativePath/>
12        <!-- lookup parent from repository -->
13    </parent>
14
15    <groupId>org.tx</groupId>
16    <artifactId>TakeawaySystem</artifactId>
17    <version>1.0-SNAPSHOT</version>
18
19    <properties>
```

```
20     <java.version>1.8</java.version>
21     <maven.compiler.source>8</maven.compiler.source>
22     <maven.compiler.target>8</maven.compiler.target>
23     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
24 </properties>
25
26 <dependencies>
27     <dependency>
28         <groupId>org.springframework.boot</groupId>
29         <artifactId>spring-boot-starter</artifactId>
30     </dependency>
31     <dependency>
32         <groupId>org.springframework.boot</groupId>
33         <artifactId>spring-boot-starter-test</artifactId>
34         <scope>test</scope>
35     </dependency>
36     <dependency>
37         <groupId>org.springframework.boot</groupId>
38         <artifactId>spring-boot-starter-web</artifactId>
39         <scope>compile</scope>
40     </dependency>
41     <dependency>
42         <groupId>com.baomidou</groupId>
43         <artifactId>mybatis-plus-boot-starter</artifactId>
44         <version>3.4.2</version>
45     </dependency>
46     <dependency>
47         <groupId>org.projectlombok</groupId>
48         <artifactId>lombok</artifactId>
49         <version>1.18.20</version>
50     </dependency>
51     <dependency>
52         <groupId>com.alibaba</groupId>
53         <artifactId>fastjson</artifactId>
54         <version>1.2.76</version>
55     </dependency>
56     <dependency>
57         <groupId>commons-lang</groupId>
58         <artifactId>commons-lang</artifactId>
59         <version>2.6</version>
60     </dependency>
61     <dependency>
62         <groupId>mysql</groupId>
63         <artifactId>mysql-connector-java</artifactId>
64         <scope>runtime</scope>
65     </dependency>
66     <dependency>
67         <groupId>com.alibaba</groupId>
68         <artifactId>druid-spring-boot-starter</artifactId>
69         <version>1.1.23</version>
70     </dependency>
71 </dependencies>
72 <build>
73     <plugins>
74         <plugin>
75             <groupId>org.springframework.boot</groupId>
76             <artifactId>spring-boot-maven-plugin</artifactId>
77             <version>2.4.5</version>
```

```

78         </plugin>
79     </plugins>
80 </build>
81
82 </project>

```

3.1.2 配置类

1. springboot中的目录结构中的static和templates:

static: 默认存放静态资源文件，可以直接通过访问地址进行访问，或者通过控制器方法

templates: 默认存放动态资源文件，需要引入thymeleaf引擎，通过控制器方法进行访问

静态页面的return默认是跳转到/static/目录下，当在pom.xml中引入了thymeleaf组件，动态跳转会覆盖默认的静态跳转，默认就会跳转到/templates/下，注意看两者return代码也有区别，动态没有html后缀。

2. 将backend(网页端)和front(移动端)以包的方式将页面放在resource下，然后通过配置信息类去进行资源绑定的映射。

```

1  @Slf4j
2  @Configuration
3  public class WebMvcConfig extends WebMvcConfigurationSupport {
4      /*设置静态资源映射*/
5      @Override
6      protected void addResourceHandlers(ResourceHandlerRegistry registry) {
7          log.info("start mapping static resources");
8
9          registry.addResourceHandler("/backend/**").addResourceLocations("classpath:/backend/");
10         registry.addResourceHandler("/front/**").addResourceLocations("classpath:/front/");
11     }
12 }

```

3. 配置R类（前后端交互协议的书写）

```

1  //通用返回结果，服务器响应的数据最终都会封装成此对象
2  @Data
3  public class R<T> {
4
5      private Integer code; //编码：1成功，0和其它数字为失败
6
7      private String msg; //错误信息
8
9      private T data; //数据
10
11     private Map map = new HashMap(); //动态数据
12
13     public static <T> R<T> success(T object) {
14         R<T> r = new R<T>();
15         r.data = object;
16         r.code = 1;
17         return r;
18     }
19 }

```

```

18     }
19
20     public static <T> R<T> error(String msg) {
21         R r = new R();
22         r.msg = msg;
23         r.code = 0;
24         return r;
25     }
26
27     public R<T> add(String key, Object value) {
28         this.map.put(key, value);
29         return this;
30     }
31 }

```

3.1.3 过滤器

主要作用：Filter对用户请求进行预处理，接着将请求交给Servlet进行处理并生成响应，最后Filter再对服务器响应进行后处理。这里通过设置过滤器来进行登录的校验和页面数据的保护。

```

1  //检查用户是否已经完成登录（过滤器）
2  @WebFilter(filterName = "LoginCheckFilter", urlPatterns = "/*")
3  @Slf4j
4  public class LoginCheckFilter implements Filter {
5      //路径匹配器
6      public static final AntPathMatcher PATH_MATCHER = new AntPathMatcher();
7      @Override
8      public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
9                          FilterChain filterChain) throws IOException, ServletException {
10         HttpServletRequest request = (HttpServletRequest) servletRequest;           //HTTP请求
11         HttpServletResponse response = (HttpServletResponse) servletResponse;       //HTTP响应
12
13         //1. 获取本次请求的URI
14         String requestURI = request.getRequestURI();
15         log.info("拦截到请求: {}", requestURI);
16         //不需要处理的请求路径
17         String[] urls = new String[]{
18             "/employee/login",
19             "/employee/logout",
20             "/backend/**",
21             "/front/**",
22             "/common/**"
23         };
24
25         //2. 判断本次请求是否需要处理
26         boolean check = check(urls, requestURI);
27
28         //3. 如果不需要处理，则直接放行
29         if (check){
30             log.info("本次请求不需要处理: {}", requestURI);
31             filterChain.doFilter(request, response);
32             return;
33         }
34     }
35 }

```



```

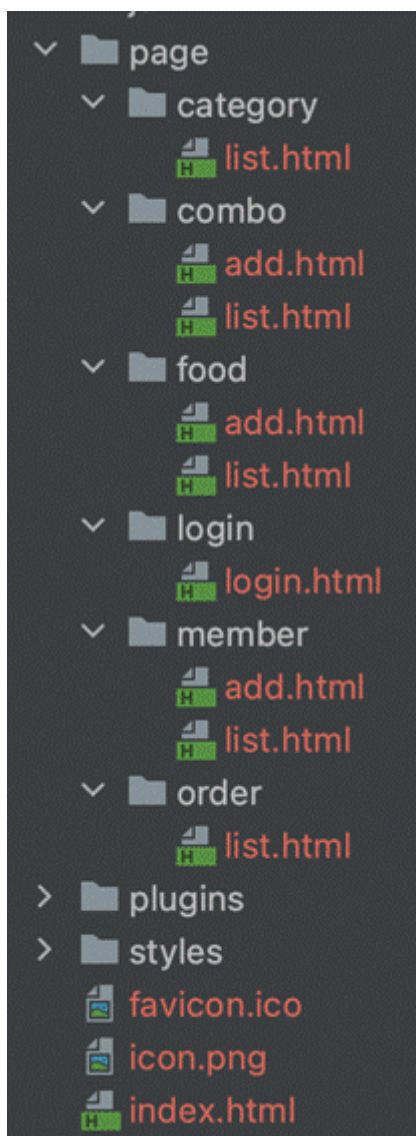
32     }
33
34     //4.判断登陆状态, 如果已经登录, 则直接放行
35     if(request.getSession().getAttribute("employee") != null){
36         log.info("用户已登录, 用户ID为: {}",
request.getSession().getAttribute("employee"));
37
38         Long empId = (Long)request.getSession().getAttribute("employee");
39         BaseContext.setCurrentId(empId);
40
41         filterChain.doFilter(request, response);
42         return;
43     }
44
45     if(request.getSession().getAttribute("user") != null){
46         log.info("用户已登录, 用户ID为: {}", request.getSession().getAttribute("user"));
47
48         Long userId = (Long)request.getSession().getAttribute("user");
49         BaseContext.setCurrentId(userId);
50
51         filterChain.doFilter(request, response);
52         return;
53     }
54
55     //5.如果未登录则返回为登陆结果,通过输出流方式向客户端页面响应数据
56     log.info("用户未登录");
57     response.getWriter().write(JSON.toJSONString(R.error("NOTLOGIN")));
58     return;
59 }
60
61 //路径匹配, 检查本次请求是否需要放行
62 public boolean check(String[] urls, String requestURI){
63     for (String url : urls) {
64         boolean match = PATH_MATCHER.match(url, requestURI);
65         if(match){
66             return true;
67         }
68     }
69     return false;
70 }
71 }

```

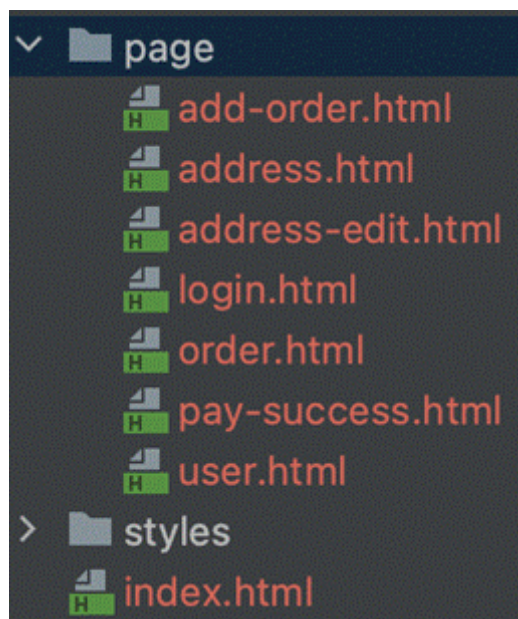
四、实现设计

4.1 前端页面的搭建

前端页面可以根据用户类型分为管理端和客户端。



管理端



客户端

4.1.1 管理端

1) 登陆页面



对登陆界面的密码输入框做了遮掩效果（`type="password"`），账号和密码输入框做了空缺提示，登陆的按钮实现了非加载和加载中两种状态，分别显示为“登陆”和“登陆中...”。

右边输入框和登陆按钮具体代码如下：

```

1 <el-form-item prop="username">
2   <el-input v-model="loginForm.username" type="text" auto-complete="off" placeholder="账号" max-length="20" prefix-icon="iconfont icon-user" />
3 </el-form-item>
4 <el-form-item prop="password">
5   <el-input v-model="loginForm.password" type="password" placeholder="密码" prefix-icon="iconfont icon-lock" max-length="20" @keyup.enter.native="handleLogin" />
6 </el-form-item>
7 <el-form-item style="width:100%;">
8   <el-button :loading="loading" class="login-btn" size="medium" type="primary" style="width:100%;" @click.native.prevent="handleLogin">
9     <span v-if="!loading">登录</span>
10    <span v-else>登录中...</span>
11  </el-button>
12 </el-form-item>

```

2) 目录页面



目录页面集成了“员工管理”、“分类管理”、“菜品管理”、“套餐管理”、“订单明细”这五大页面，对这五大页面进行了调用再增加了左侧的一个索引列表。

左侧页面索引列表的实现代码如下：

```

1 <div v-for="item in menuList" :key="item.id">
2   <el-submenu :index="item.id" v-if="item.children && item.children.length>0">
3     <template slot="title">
4       <i class="iconfont" :class="item.icon"></i>
5       <span>{{item.name}}</span>
6     </template>
7     <el-menu-item
8       v-for="sub in item.children"
9       :index="sub.id"
10      :key="sub.id"
11      @click="menuHandle(sub,false)"
12    >
13      <i :class="iconfont" :class="sub.icon"></i>
14      <span slot="title">{{sub.name}}</span>
15    </el-menu-item>
16  </el-submenu>
17  <el-menu-item v-else :index="item.id" @click="menuHandle(item,false)">
18    <i class="iconfont" :class="item.icon"></i>
19    <span slot="title">{{item.name}}</span>
20  </el-menu-item>
21 </div>

```

menuList如下

```
1  menuList: [  
2    {  
3      id: '1',  
4      name: '员工管理',  
5      url: 'page/member/list.html',  
6      icon: 'icon-member'  
7    },  
8    {  
9      id: '2',  
10     name: '分类管理',  
11     url: 'page/category/list.html',  
12     icon: 'icon-category'  
13   },  
14   {  
15     id: '3',  
16     name: '菜品管理',  
17     url: 'page/food/list.html',  
18     icon: 'icon-food'  
19   },  
20   {  
21     id: '4',  
22     name: '套餐管理',  
23     url: 'page/combo/list.html',  
24     icon: 'icon-combo'  
25   },  
26   {  
27     id: '5',  
28     name: '订单明细',  
29     url: 'page/order/list.html',  
30     icon: 'icon-order'  
31   }  
32 ]
```

3) 员工管理页面



页表有“员工姓名”、“账号”、“手机号”、“状态”、“操作”五个字段，能自动分隔，占满页面，不需要手动调整距离。左上还有查找功能，便于管理员快速找到对应的员工信息。右上角有“添加员工”按钮，便于管理员对员工进行增加操作。

员工信息页表实现代码如下：

```

1 <el-table :data="tableData" stripe class="tableBox">
2   <el-table-column prop="name" label="员工姓名"></el-table-column>
3   <el-table-column prop="username" label="账号"></el-table-column>
4   <el-table-column prop="phone" label="手机号"></el-table-column>
5   <el-table-column label="账号状态">
6     <template slot-scope="scope">
7       {{ String(scope.row.status) === '0' ? '已禁用' : '正常' }}
8     </template>
9   </el-table-column>
10  <el-table-column label="操作" width="160" align="center">

```

员工搜索框与添加按钮实现代码如下：

```

1 <div class="tableBar">
2   <el-input v-model="input" placeholder="请输入员工姓名" style="width: 250px" clearable
  @keyup.enter.native="handleQuery">
3     <i slot="prefix" class="el-input__icon el-icon-search" style="cursor: pointer"
  @click="handleQuery"></i>
4   </el-input>
5   <el-button type="primary" @click="addMemberHandle('add')">
6     + 添加员工
7   </el-button>
8 </div>

```

* 账号:

* 员工姓名:

* 手机号:

性别: ☒ 男 ☐ 女

* 身份证号:

取消 保存 保存并继续添加

点击右上的“添加员工”按钮能响应click事件，调用方法add，跳转到add.html页面，进行添加操作

添加员工具体信息实现代码如下：

```

1 <el-form-item label="账号:" prop="username">
2   <el-input v-model="ruleForm.username" placeholder="请输入账号" maxlength="20"/>
3 </el-form-item>
4 <el-form-item label="员工姓名:" prop="name">
5   <el-input v-model="ruleForm.name" placeholder="请输入员工姓名" maxlength="20"/>
6 </el-form-item>
7 <el-form-item label="手机号:" prop="phone">
8   <el-input v-model="ruleForm.phone" placeholder="请输入手机号" maxlength="20"/>
9 </el-form-item>
10 <el-form-item label="性别:" prop="sex">
11   <el-radio-group v-model="ruleForm.sex">
12     <el-radio label="男"></el-radio>
13     <el-radio label="女"></el-radio>
14   </el-radio-group>

```

```

15 </el-form-item>
16 <el-form-item label="身份证号:" prop="idNumber">
17   <el-input v-model="ruleForm.idNumber" placeholder="请输入身份证号" maxlength="20"/>
18 </el-form-item>

```

4) 分类管理页面



页表有“分类名单”、“分类类型”、“操作时间”、“排序”、“操作”五个字段，左上“新增菜品分类”和“新增套餐分类”两个按钮可以对数据表进行扩充

新增菜品分类和套餐分类实现代码如下：

```

1 <el-button type="primary" class="continue" @click="addClass('class')">
2   + 新增菜品分类
3 </el-button>
4 <el-button type="primary" @click="addClass('meal')">
5   + 新增套餐分类
6 </el-button>

```

```

1 // 弹窗添加
2 addClass(st) {
3   if (st == 'class') {
4     this.classData.title = '新增菜品分类'
5     this.type = '1'
6   } else {
7     this.classData.title = '新增套餐分类'
8     this.type = '2'
9   }
10  this.action = 'add'
11  this.classData.name = ''
12  this.classData.sort = ''
13  this.classData.dialogVisible = true
14  },
15  editHandle(dat) {
16    this.classData.title = '修改分类'
17    this.action = 'edit'
18    this.classData.name = dat.name
19    this.classData.sort = dat.sort
20    this.classData.id = dat.id
21    this.classData.dialogVisible = true
22  },
23 // 关闭弹窗
24 handleClose(st) {
25   this.classData.dialogVisible = false

```

5) 菜品/套餐管理页面



页表有“菜品名称”、“图片”、“菜品分类”、“售价”、“售卖状态”、“最后操作时间”、“操作”七个字段。左上有搜索框可以直接对菜品信息进行搜索，右上有“批量处理”和“新增菜品”四个按钮可以对菜品进行批量管理和增添，在有导入菜品数据之后还可以对菜品做删减处理

菜品搜索框和批量处理、新增菜品按钮实现代码如下：

```

1 <template slot-scope="{ row }">
2   <el-button type="text" @click="goDetail(row)" class="blueBug">
3     查看
4   </el-button>
5   <el-divider v-if="row.status === 2" direction="vertical"></el-divider>
6   <el-button v-if="row.status === 2" type="text" @click="cancelOrDeliveryOrComplete(3,
7     row.id)" class="blueBug">
8     派送
9   </el-button>
10  <el-divider v-if="row.status === 3" direction="vertical"></el-divider>
11  <el-button v-if="row.status === 3" type="text" @click="cancelOrDeliveryOrComplete(4,
12    row.id)" class="blueBug">
13    完成
14  </el-button>
15 </template>

```

6) 订单明细页面



对每个订单有五种状态“待付款”、“正在配送”、“已派送”、“已完成”、“已取消”。

订单状态代码设计如下：

```

1  switch(row.status){
2      case 1:
3          str = '待付款'
4          break;
5      case 2:
6          str = '正在派送'
7          break;
8      case 3:
9          str = '已派送'
10         break;
11     case 4:
12         str = '已完成'
13         break;
14     case 5:
15         str = '已取消'
16         break;
17 }
18 return str

```

管理员可对每个订单进行“查看”、“派送”、“完成”三种操作。订单状态为“正在配送”时，管理员能对订单进行“配送”操作；同理，当订单状态为“已配送”时，管理员能对订单进行“完成”操作

订操作代码实现如下：

```

1  <template slot-scope="{ row }">
2      <el-button type="text" @click="goDetail(row)" class="blueBug">
3          查看
4      </el-button>
5      <el-divider v-if="row.status === 2" direction="vertical"></el-divider>
6      <el-button v-if="row.status === 2" type="text" @click="cancelOrDeliveryOrComplete(3,
7          row.id)" class="blueBug">
8          派送
9      </el-button>
10     <el-divider v-if="row.status === 3" direction="vertical"></el-divider>
11     <el-button v-if="row.status === 3" type="text" @click="cancelOrDeliveryOrComplete(4,
12         row.id)" class="blueBug">
13         完成
14     </el-button>
15 </template>

```

4.1.2 客户端

1) 登陆页面



与管理端登陆页面类似，对密码做了遮掩处理，登陆的按钮实现了非加载和加载中两种状态，分别显示为“登陆”和“登陆中...”。

登陆框设计代码如下：

```
1 <div class="divHead">登录</div>
2 <div class="divContainer">
3   <el-input placeholder=" 请输入账号" v-model="form.username" maxlength='20'/></el-input>
4   <el-input type="password" placeholder=" 请输入密码" v-model="form.password"
   maxlength='20'/></el-input>
5   <el-button :loading="loading" class="login-btn" size="medium" style="width:100%;"
   type="primary" @click="btnLogin">
6     <span v-if="!loading">登录</span>
7     <span v-else>登录中...</span>
8   </el-button>
9 </div>
```

2) 功能主页面



页面顶部是店铺的简单介绍，包含“配送距离”、“配送费”、“配送时长”以及店铺的logo和口号标语。

顶部店铺介绍设计代码如下

```

1 <div class="divTitle">
2   <div class="divStatic">
3     
4     <div class="divDesc">
5       <div class="divName">外卖订餐</div>
6       <div class="divSend">
7         <span> 距离1.5km</span>
8         <span> 配送费6元</span>
9         <span> 预计时长12min</span>
10      </div>
11    </div>
12  </div>
13  <div class="divDesc">
14    不闻人间烟火，但食人间美味！
15  </div>
16 </div>

```

中间主体部分，左侧是“菜品分类”的滑动列表，右侧是“菜品细则”的滑动列表，皆能根据index自动索取相应数据。

滑动列表设计代码如下：

```

1 <div class="divBody">
2   //左侧分类列表
3   <div class="divType">
4     <ul>
5       <li v-for="(item,index) in categoryList" :key="index"
6         @click="categoryClick(index,item.id,item.type)" :class="{active:activeType === index}">
7         {{item.name}}</li>
8     </ul>
9   </div>
10  //右侧菜单列表
11  <div class="divMenu">
12    <div class="divItem" v-for="(item,index) in dishList" :key="index"
13    @click="dishDetails(item)">
14      <el-image :src="imgPathConvert(item.image)" >
15        <div slot="error" class="image-slot">
16          
17        </div>
18      </el-image>
19      <div class="divName">{{item.name}}</div>
20      <div class="divDesc">{{item.description}}</div>
21      <div class="divDesc">{{'月销' + (item.saleNum ? item.saleNum : 0) }}</div>
22      <div class="divBottom"><span>¥</span><span>{{item.price/100}}</span></div>
23      <div class="divNum">
24        <div class="divSubtract" v-if="item.number > 0">
25          
26        </div>
27        <div class="divDishNum">{{item.number}}</div>
28        <div class="divTypes" v-if="item.flavors && item.flavors.length > 0 &&
29          !item.number " @click.prevent.stop="chooseFlavorClick(item)">选择规格</div>
30        <div class="divAdd" v-else>
31          
32        </div>
33      </div>
34    </div>
35  </div>

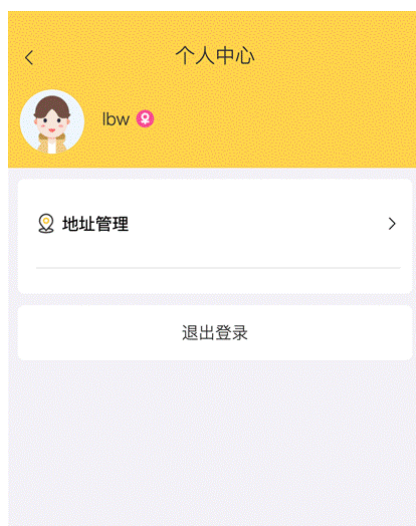
```

点击左上角的“头像图标”可以进入“个人中心”页面。

跳转个人中心页面实现代码如下：

```
1 <div class="divHead">
2   
3 </div>
```

3) 个人中心页面



个人中心页面包含用户的“账号名”、“电话”和“地址”，其数据会与登陆者的数据相关联并自动初始化。点击“地址管理”条框，便能响应click事件调用toAddressPage方法跳转至address.html进行地址编辑操作。点击“退出登陆”按钮便能响应click事件调用toPageLogin方法跳转至login.html重新登录。

个人中心页面设计代码如下：

```
1 <div id="user" class="app">
2   <div class="divHead">
3     <div class="divTitle">
4       <i class="el-icon-arrow-left" @click="goBack"></i>个人中心
5     </div>
6     <div class="divUser">
7       
8       <div class="desc">
9         <div class="divName">
10          {{userName}}
11          
12        </div>
13        <div class="divPhone">{{userPhone}}</div>
14      </div>
15    </div>
16  </div>
17  <div class="divContent">
18    <div class="divLinks">
19      <div class="item" @click="toAddressPage">
20        
21        <span>地址管理</span>
22        <i class="el-icon-arrow-right"></i>
23      </div>
24      <div class="divSplit"></div>
```

```

25         </div>
26         <div class="quitLogin" @click="toPageLogin">
27             退出登录
28         </div>
29     </div>
30 </div>

```

个人信息获取实现代码如下：

```

1 created(){
2     this.userName =sessionStorage.getItem("userName")
3     this.userPhone =sessionStorage.getItem("userPhone")
4     this.initData()
5 },

```

4) 地址管理页面



每个用户可以添加若干个地址，每个地址有三种类别“家”、“学校”、“公司”可以设定，收货人的性别是“男”则会自动加上称呼“先生”，收货人的性别是“女”则会自动加上称呼“女士”，点击“圆圈”可以响应click事件调用setDefaultAddress(item)方法设定默认地址，点击右侧的“铅笔”图标可以响应click事件调用toAddressEditPage(item)方法对地址进行修改编辑，点击下方的“+添加收货地址”按钮可以跳转到add-address.html页面进行地址添加。

地址页面设计代码如下：

```

1 <div class="divContent">
2     <div class="divItem" v-for="(item,index) in addressList" :key="index"
3     @click.capture="itemClick(item)">
4         <div class="divAddress">
5             <span :class="{spanCompany:item.label === '公司',spanHome:item.label ===
6             '家',spanSchool:item.label === '学校'}">{{item.label}}</span>
7             {{item.detail}}

```

```

8         <span>{{item.consignee}}</span>
9         <span>{{item.sex === '0' ? '女士' : '先生'}}</span>
10        <span>{{item.phone}}</span>
11    </div>
12    
13    <div class="divSplit"></div>
14    <div class="divDefault" >
15        
16        设为默认地址
18    </div>
19 </div>

```

5) 下单页面



页面上方显示的是“收货地址信息”，页面中间显示的是“订单明细”，包括“菜品名称”、“数量”、“价格”，页面下方是总金额和支付按钮。

下单页面设计代码如下：

```

1  <div class="divAddress">
2      <div @click="toAddressPage">
3          <div class="address">{{address.detail}}</div>
4          <div class="name">
5              <span>{{address.consignee}}{{address.sex === '1' ? '先生': '女士'}}</span>
6              <span>{{address.phone}}</span>
7          </div>
8          <i class="el-icon-arrow-right"></i>
9      </div>
10     <div class="divSplit"></div>
11     <div class="divFinishTime">预计{{finishTime}}送达</div>
12 </div>
13 <div class="order">
14     <div class="title">订单明细</div>
15     <div class="divSplit"></div>

```

```

16     <div class="itemList">
17         <div class="item" v-for="(item,index) in cartData" :key="index">
18             <el-image :src="imgPathConvert(item.image)">
19                 <div slot="error" class="image-slot">
20                     
21                 </div>
22             </el-image>
23             <div class="desc">
24                 <div class="name">{{item.name}}</div>
25                 <div class="numPrice">
26                     <span class="num">x{{item.number}}</span>
27                     <div class="price">
28                         <span class="spanMoney">¥</span>{{item.amount}}
29                     </div>
30                 </div>
31             </div>
32         </div>
33     </div>
34 </div>

```

底部总金额与支付按钮设计代码如下：

```

1  <div class="divCart">
2      <div :class="{imgCartActive: cartData && cartData.length > 0, imgCart:!cartData ||
3          cartData.length<1}"></div>
4      <div :class="{divGoodsNum:1===1, moreGoods:cartData && cartData.length > 99}" v-
5          if="cartData && cartData.length > 0">{{ goodsNum }}</div>
6      <div class="divNum">
7          <span>¥</span>
8          <span>{{goodsPrice}}</span>
9      </div>
10     <div class="divPrice"></div>
11     <div :class="{btnSubmitActive: cartData && cartData.length > 0, btnSubmit:!cartData ||
12         cartData.length<1}" @click="goToPaySuccess">
13         去支付
14     </div>
15 </div>

```

6) 支付成功页面



支付成功之后，可以点击左上的“回退”按钮或者右上的“home”按钮，跳转到主页面，也可以点击下方的“查看订单”按钮跳转到订单页面。

跳转页面方法设计如下：

```
1  methods:{
2      goBack(){
3          window.requestAnimationFrame(()=>{
4              window.location.replace('/front/index.html')
5          })
6      },
7      toOrderPage(){
8          window.requestAnimationFrame(()=>{
9              window.location.replace('/front/page/order.html')
10         })
11     },
12     toMainPage(){
13         window.requestAnimationFrame(()=>{
14             window.location.replace('/front/index.html')
15         })
16     },
17 }
```

4.2 功能模块实现

创建entity, controller, service, mapper, 分别对应：订餐系统中实现的类、类的方法、对应类提供的服务、对应类的映射数据库表。

4.2.1 管理端

1) 管理员登录

给EmployeeController类添加一个login方法

@RequestBody主要用于接收前端传递给后端的json字符串（请求体中的数据）

HttpServletRequest request作用：如果登录成功，将员工对应的id存到session一份，这样想获取一份登录用户的信息就可以随时获取出来。

```
1 public R<Employee> login(HttpServletRequest request, @RequestBody Employee employee){
2     //1.将页面提交的pw进行md5加密
3     String password = employee.getPassword();
4     password = DigestUtils.md5DigestAsHex(password.getBytes());
5
6     //2.根据页面提交的用户名查询数据库
7     LambdaQueryWrapper<Employee> queryWrapper = new LambdaQueryWrapper<>();
8     queryWrapper.eq(Employee::getUsername, employee.getUsername());
9     Employee emp = employeeService.getOne(queryWrapper);
10
11     //3.没有查询到结果返回登陆失败
12     if(emp == null){
13         return R.error("此用户不存在");
14     }
15
16     //4.密码比对，不一致返回登陆失败
17     if(!emp.getPassword().equals(password)){
18         return R.error("密码错误，登陆失败");
19     }
20
21     //5.查看管理员状态，如果为已禁用，则返回员工已禁用结果
22     if(emp.getStatus() == 0){
23         return R.error("账号已禁用");
24     }
25
26     //6.登陆成功，将管理员id存入Session并返回登陆结果
27     request.getSession().setAttribute("employee", emp.getId());
28     return R.success(emp);
29 }
```

2) 管理员退出

员工登录成功后，页面跳转到后台系统首页面(backend/index.html),此时会显示当前登录用户的姓名，如果员工需要退出系统，直接点击右侧的退出按钮即可退出系统，退出系统后页面应跳转回登录页面。

在EmployeeController类中添加logout方法


```

1 //7.退出登录
2 @PostMapping("/logout")
3 public R<String> logout(HttpServletRequest request){
4     //清理Session上保存的当前管理员的ID
5     request.getSession().removeAttribute("employee");
6     return R.success("退出成功");
7 }

```

3) 员工管理

a. 添加管理员 (员工)

前端通过表单提交的方式向服务器发送请求携带参数(json格式), 请求的url为:

http://localhost:8014/employee

然后后端请求的url在controller层编写代码, 提交的参数中没有的属性则需要在控制器方法中进行手动的设置, 然后调用service层, service层调用dao层来保存员工数据, 最后以json的格式相应给客户端。

```

1 //8.添加管理员
2 @PostMapping
3 public R<String> save(HttpServletRequest request, @RequestBody Employee employee){
4     log.info("新增管理员, 管理员信息: {}", employee.toString());
5
6     //设置初始密码123456并进行MD5加密
7     employee.setPassword(DigestUtils.md5DigestAsHex("123456".getBytes()));
8
9     //employee.setCreateTime(LocalDateTime.now());
10    //employee.setUpdateTime(LocalDateTime.now());
11    //获得当前登陆用户的ID
12    //Long empId = (Long)request.getSession().getAttribute("employee");
13    //employee.setCreateUser(empId);
14    //employee.setUpdateUser(empId);
15
16    employeeService.save(employee);
17    return R.success("新增管理员成功! ");
18 }

```

由于在数据库中的username字段设置了unique, 所以当前端的表单中出现了同一个username时, 后端由于dao层出现sql异常无法添加, 所以我们应该设置一个全局异常处理器来处理出现的异常情况响应给浏览器。

```

1 //进行异常处理
2 @ExceptionHandler(SQLIntegrityConstraintViolationException.class)
3 public R<String> exceptionHandler(SQLIntegrityConstraintViolationException ex){
4     log.error(ex.getMessage());
5
6     if(ex.getMessage().contains("Duplicate entry")){
7         String[] split = ex.getMessage().split(" ");
8         String msg = split[2] + "已存在";
9         return R.error(msg);
10    }
11
12    return R.error("未知错误");
13 }

```

目前存在的管理员为admin



点击添加员工并填入信息

* 账号:

* 员工姓名:

* 手机号:

性别: ☒ 男 ☐ 女

* 身份证号:

添加成功



b. 员工信息分页查询

登录到员工管理页面后，我们需要将数据库中的员工信息进行分页查询到前端进行展示，这里我们使用了mybatisplus提供的分页插件：

```
1 //配置MybatisPlus分页插件
2 @Configuration
3 public class MybatisPlusConfig {
4     @Bean
5     public MybatisPlusInterceptor mybatisPlusInterceptor(){
6         MybatisPlusInterceptor mybatisPlusInterceptor = new MybatisPlusInterceptor();
7         mybatisPlusInterceptor.addInnerInterceptor(new PaginationInnerInterceptor());
8         return mybatisPlusInterceptor;
9     }
10 }
```

前端请求路径为：

```
http: //localhost : 8080/ employee/ page ?page=1&pageSize=10
```

然后根据前端需要的请求路径和参数来编写我们的controller方法来处理请求：

```
1 //9.管理员信息分页查询
2 @GetMapping("/page")
3 public R<Page> page(int page, int pageSize, String name){
4     log.info("page = {}, pageSize = {}, name = {}", page, pageSize, name);
5
6     //构造分页构造器
7     Page pageInfo = new Page(page, pageSize);
8
9     //构造条件构造器
10    LambdaQueryWrapper<Employee> queryWrapper = new LambdaQueryWrapper();
11
12    //添加过滤条件
13    queryWrapper.like(StringUtils.isEmpty(name), Employee::getName, name);
14
15    //添加排序条件
16    queryWrapper.orderByDesc(Employee::getUpdateTime);
17
18    //执行查询
19    employeeService.page(pageInfo, queryWrapper);
20
21    return R.success(pageInfo);
22 }
```

c.修改员工帐号状态及信息功能

编辑功能就是修改员工的基本信息，当我们点击时会出现员工的原来的基本信息，将员工的基本信息回显到表单中，也就是前端通过一个get请求去通过请求参数id来调用service层进而调用dao层获取员工的基本信息，最后再进行修改。

```
1 //10.根据ID修改管理员信息
2 @PutMapping
3 public R<String> update(HttpServletRequest request, @RequestBody Employee employee){
4     log.info(employee());
5
6     long id = Thread.currentThread().getId();
7     log.info("线程ID为: {}", id);
8
9     //Long empId = (Long)request.getSession().getAttribute("employee");
10    //employee.setUpdateTime(LocalDateTime.now());
11    //employee.setUpdateUser(empId);
12    employeeService.updateById(employee);
13
14    return R.success("管理员信息修改成功!");
15 }
```

后端@RequestBody注解对应的类在将HTTP的输入流(含请求体)装配到目标类(即:@RequestBody后面的类)时，会根据json字符串中的key来匹配对应实体类的属性，如果匹配一致且json中的该key对应的值符合(或可转换为)实体类的对应属性的类型要求时，会调用实体类的setter方法将值赋给该属性。

当我们修改完后点击'保存'按钮时，会出现更新失败的情况。

原因：通过id修改员工的信息，而前端请求接收到的id是js处理过后的，与后端java中的在数据库保存的id是有精度损失的情况，所以我们无法通过比较前后端的Long型id来进行修改员工信息，即无法处理此次请求。(由上图可知响应给浏览器的id非String类型)

```
1 //11.根据id查询管理员信息
2 @GetMapping("/{id}")
3 public R<Employee> getById(@PathVariable Long id){
4     Employee employee = employeeService.getById(id);
5     if(employee != null){
6         return R.success(employee);
7     }
8     return R.error("没有查询到对应管理员信息");
9 }
```

所以解决的办法就是在服务端给页面响应json数据时进行处理，将long型数据统一转换为String字符串。

解决方法：

1. 提供对象转换器JacksonObjectMapper,基于jackson进行java对象到json的转换
2. 在webMvcConfig 配置类中扩展 Spring MVC 的消息转换器，在此消息转换器中使用对象转换器 JacksonObjectMapper进行java对象到json数据的转换

```
1 /**
2  * 对象映射器:基于jackson将Java对象转为json，或者将json转为Java对象
3  * 将JSON解析为Java对象的过程称为 [从JSON反序列化Java对象]
4  * 从Java对象生成JSON的过程称为 [序列化Java对象到JSON]
5  */
6 public class JacksonObjectMapper extends ObjectMapper {
7
8     public static final String DEFAULT_DATE_FORMAT = "yyyy-MM-dd";
9     public static final String DEFAULT_DATE_TIME_FORMAT = "yyyy-MM-dd HH:mm:ss";
10    public static final String DEFAULT_TIME_FORMAT = "HH:mm:ss";
11
12    public JacksonObjectMapper() {
13        super();
14        //收到未知属性时不报异常
15        this.configure(FAIL_ON_UNKNOWN_PROPERTIES, false);
16
17        //反序列化时，属性不存在的兼容处理
18
19        this.getDeserializationConfig().withoutFeatures(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
20
21        SimpleModule simpleModule = new SimpleModule()
22            .addDeserializer(LocalDateTime.class, new
23                LocalDateTimeDeserializer(DateTimeFormatter.ofPattern(DEFAULT_DATE_TIME_FORMAT)))
24            .addDeserializer(LocalDate.class, new
25                LocalDateDeserializer(DateTimeFormatter.ofPattern(DEFAULT_DATE_FORMAT)))
26            .addDeserializer(LocalTime.class, new
27                LocalTimeDeserializer(DateTimeFormatter.ofPattern(DEFAULT_TIME_FORMAT)))
28
29            .addSerializer(BigInteger.class, ToStringSerializer.instance)
30            .addSerializer(Long.class, ToStringSerializer.instance)
```

```

28         .addSerializer(LocalDate.class, new
LocalDateTimeSerializer(DateTimeFormatter.ofPattern(DEFAULT_DATE_TIME_FORMAT)))
29         .addSerializer(LocalDate.class, new
LocalDateSerializer(DateTimeFormatter.ofPattern(DEFAULT_DATE_FORMAT)))
30         .addSerializer(LocalTime.class, new
LocalTimeSerializer(DateTimeFormatter.ofPattern(DEFAULT_TIME_FORMAT)));
31
32         //注册功能模块 例如，可以添加自定义序列化器和反序列化器
33         this.registerModule(simpleModule);
34     }
35 }
36 //扩展mvc框架的消息转换器
37 @Override
38 protected void extendMessageConverters(List<HttpMessageConverter<?>> converters){
39     //创建消息转换器对象
40     MappingJackson2HttpMessageConverter messageConverter = new
MappingJackson2HttpMessageConverter();
41     //设置对象转换器，底层使用jackson将java对象转成json
42     messageConverter.setObjectMapper(new JacksonObjectMapper());
43     //将上面的消息转换器对象追加到mvc框架的转换器容器
44     converters.add(0, messageConverter);
45 }

```

+ 添加员工

员工姓名	账号	手机号	账号状态	操作
booooooox	lbw	18655598678	正常	编辑 禁用
管理员	admin	13812312312	正常	编辑 禁用

共 2 条
 10条/页
 <
 >
 前往
 1 页

d.公共字段填充

由于在每个类中都存在createTime, updateTime等每个表(类)都有的字段，但这些字段都是需要在插入表中为非null，这就需要在每个保存数据的方法中都需要手动set值进行插入，mybatisplus就为我们提供了一种为所有表中存在的公共字段进行填充。

步骤：

在需要填充的属性中加上@TableField，其中里面的属性fill对应的值对应三个值，FieldFill.INSERT(插入时填充该字段)，FieldFill.UPDATE(更新时填充该字段)，FieldFill.INSERT_UPDATE(插入或者更新时填充该字段)。创建一个类去实现MetaObjectHandler接口，并且重写insertFill和updateFill两个方法

```

1 public class MyMetaObjectHandler implements MetaObjectHandler {
2     //插入自动填充
3     @Override
4     public void insertFill(MetaObject metaObject) {
5         metaObject.setValue("createTime", LocalDateTime.now());
6         metaObject.setValue("updateTime", LocalDateTime.now());
7         metaObject.setValue("createUser", BaseContext.getCurrentId());
8         metaObject.setValue("updateUser", BaseContext.getCurrentId());
9     }
10    //更新自动填充
11    @Override
12    public void updateFill(MetaObject metaObject) {

```

```

13     long id = Thread.currentThread().getId();
14     log.info("线程ID为: {}", id);
15
16     metaObject.setValue("updateTime", LocalDateTime.now());
17     metaObject.setValue("updateUser", BaseContext.getCurrentId());
18 }
19 }

```

4) 分类管理

a. 删除菜品分类

删除菜品分类，由于dish表和setmeal表中有菜品分类的字段(category-id)，所以我们在删除该菜品分类信息时需要去判断该菜品分类是否关联了某个菜品或者某个套餐，如果关联了则抛出异常，并向前端展示；如果没有关联则直接删除即可。

在CategoryService业务层进行创建一个remove方法(实现删除菜品分类信息功能)，然后让CategoryServiceImpl去实现该业务逻辑。

```

1  public class CategoryServiceImpl extends ServiceImpl<CategoryMapper, Category> implements
    CategoryService {
2
3      @Autowired
4      private DishService dishService;
5      @Autowired
6      private SetMealService setMealService;
7
8      //根据id删除分类，删除之前进行判断
9      @Override
10     public void remove(Long id) {
11         LambdaQueryWrapper<Dish> dishLambdaQueryWrapper = new LambdaQueryWrapper<>();
12         //添加查询条件，根据分类id进行查询
13         dishLambdaQueryWrapper.eq(Dish::getCategoryId, id);
14         int count1 = dishService.count(dishLambdaQueryWrapper);
15
16         //查询当前分类是否关联菜品，如果关联，抛出异常
17         if(count1 > 0){
18             throw new CustomException("当前分类有菜品，不能删除!");
19         }
20         LambdaQueryWrapper<Setmeal> setmealLambdaQueryWrapper = new LambdaQueryWrapper<>();
21         //添加查询条件，根据分类id进行查询
22         setmealLambdaQueryWrapper.eq(Setmeal::getCategoryId, id);
23         int count2 = setMealService.count(setmealLambdaQueryWrapper);
24
25         //查询当前分类是否关联套餐，如果关联，抛出异常
26         if(count2 > 0){
27             throw new CustomException("当前分类有套餐，不能删除!");
28         }
29         //正常删除
30         super.removeById(id);
31     }
32 }

```

b.自定义业务异常类

```
1 //自定义业务异常
2 public class CustomException extends RuntimeException{
3     public CustomException(String message){
4         super(message);
5     }
6 }
7 @ExceptionHandler(CustomException.class)
8 public R<String> exceptionHandler(CustomException ex){
9     return R.error(ex.getMessage());
10 }
```

5) 菜品管理

a.新增菜品

1. 菜品分类下拉框功能

* 菜品分类: 请选择菜品分类 ^

* 菜品价格:

口味做法配置:

* 菜品图片:

- 家常菜
- 主食
- 徽菜
- 粤菜
- 川菜
- 湘菜
- 饮品

在CategoryController层去实现该请求:

```
1 //根据条件查询分类数据
2 @GetMapping("/list")
3 public R<List<Category>> list(Category category){
4     LambdaQueryWrapper<Category> queryWrapper = new LambdaQueryWrapper<>();
5     queryWrapper.eq(category.getType() != null, Category::getType, category.getType());
6     queryWrapper.orderByAsc(Category::getSort).orderByDesc(Category::getUpdateTime);
7     List<Category> list = categoryService.list(queryWrapper);
8     return R.success(list);
9 }
```

2. 文件上传与下载

页面发送请求进行图片的上传，请求服务端将图片保存到服务器。专门创建一个控制层来处理文件的上传和下载的请求处理。

```
1 //文件上传下载
2 @RestController
3 @RequestMapping("/common")
4 public class CommonController {
5
6     @Value("${takeaway.path}")
7     private String basePath;
```

```

8
9 //文件上传
10 @PostMapping("/upload")
11 public R<String> upload(MultipartFile file){
12     //file是一个临时文件，需要转存到指定位置，否则本次请求完成之后临时文件会删除
13
14     //原始文件名
15     String originalFilename = file.getOriginalFilename();
16     String suffix = originalFilename.substring(originalFilename.lastIndexOf("."));
17
18     //使用UUID重新生成文件名
19     String fileName = UUID.randomUUID().toString() + suffix;
20
21     //创建目录对象
22     File dir = new File(basePath);
23     if(!dir.exists()){
24         dir.mkdirs();
25     }
26
27     try {
28         file.transferTo(new File(basePath + fileName));
29     } catch (IOException e) {
30         e.printStackTrace();
31     }
32     return R.success(fileName);
33 }
34
35 //文件下载
36 @GetMapping("/download")
37 public void download(String name, HttpServletResponse response){
38     //输入流，读取文件内容
39     try {
40         FileInputStream fileInputStream = new FileInputStream(new File(basePath +
name));
41
42         //输出流，通过输出流将文件写回浏览器
43         ServletOutputStream outputStream = response.getOutputStream();
44
45         response.setContentType("image/jpeg");
46
47         int len = 0;
48         byte[] bytes = new byte[1024];
49         while((len = fileInputStream.read(bytes)) != -1){
50             outputStream.write(bytes, 0, len);
51             outputStream.flush();
52         }
53         outputStream.close();
54         fileInputStream.close();
55
56     } catch (Exception e) {
57         throw new RuntimeException(e);
58     }
59 }
60 }

```

3. 将新增菜品保存到数据库

将数据填入表单后进行提交到后端请求保存到数据库中，但是表单提交的信息不仅只有category一张表的信息，还有dish_flavor表的信息，所以我们无法通过简单的一个实体类去保存数据。所以需要导入一个DishDto，用于封装页面提交的数据。

DTO,全称为Data Transfer Object,即数据传输对象，一般用于展示层与服务层之间的数据传输。

然后在dishService业务层创建一个方法处理保存菜品同时保存对应的口味数据到数据库，然后dishServiceImpl去实现该业务逻辑。

```
1 @Transactional
2 public void saveWithFlavor(DishDto dishDto) {
3     //保存菜品基本信息到dish
4     this.save(dishDto);
5     Long dishId = dishDto.getId(); //菜品id
6
7     List<DishFlavor> flavors = dishDto.getFlavors(); //菜品口味
8     flavors = flavors.stream().map((item)->{
9         item.setDishId(dishId);
10        return item;
11    }).collect(Collectors.toList());
12
13    //保存菜品口味到dish_flavor
14    dishFlavorService.saveBatch(flavors);
15 }
```

在此方法上我们开始事务@Transactional并且在主程序中开启事务管理@EnableTransactionManagement：

[批量删除](#) [批量启售](#) [批量停售](#) [+ 新建菜品](#)

<input type="checkbox"/> 菜品名称	图片	菜品分类	售价	售卖状态	最后操作时间	操作
<input type="checkbox"/> 王老吉		饮品	¥ 5	启售	2022-11-15 22:26:15	修改 停售 删除
<input type="checkbox"/> 辣子鸡		川菜	¥ 30	启售	2022-11-15 22:25:29	修改 停售 删除
<input type="checkbox"/> 红烧肉		家常菜	¥ 20	启售	2022-11-15 22:24:03	修改 停售 删除

共 13 条 [<](#) [1](#) [2](#) [>](#) 前往 页

* 菜品名称:

澳洲龙虾

* 菜品分类:

家常菜

* 菜品价格:

588

口味做法配置:

口味名 (3个字内) 口味标签 (输入标签回车添加)

忌口

不要葱 X

不要蒜 X


不要香菜 X











不要辣 X

删除

添加口味

* 菜品图片:



菜品名称	图片	菜品分类	售价	售卖状态	最后操作时间	操作
<input type="checkbox"/> 澳洲龙虾		家常菜	¥ 588	售罄	2022-11-16 22:58:22	修改 编辑 删除
<input type="checkbox"/> 邵阳灌血丸子		湘菜	¥ 30	售罄	2022-11-15 22:39:35	修改 编辑 删除
<input type="checkbox"/> 泡椒鱼唇		湘菜	¥ 120	售罄	2022-11-15 22:39:11	修改 编辑 删除
<input type="checkbox"/> 白斩鸡		粤菜	¥ 40	售罄	2022-11-15 22:37:57	修改 编辑 删除
<input type="checkbox"/> 徽州臭鳃鱼		徽菜	¥ 40	售罄	2022-11-15 22:35:32	修改 编辑 删除
<input type="checkbox"/> 李鸿章杂碎		徽菜	¥ 100	售罄	2022-11-15 22:35:02	修改 编辑 删除
<input type="checkbox"/> 鱼香肉丝		家常菜	¥ 20	售罄	2022-11-15 22:31:58	修改 编辑 删除
<input type="checkbox"/> 麻辣兔头		川菜	¥ 40	售罄	2022-11-15 22:30:30	修改 编辑 删除
<input type="checkbox"/> 蛋卷		主食	¥ 5	售罄	2022-11-15 22:29:47	修改 编辑 删除
<input type="checkbox"/> 米饭		主食	¥ 2	售罄	2022-11-15 22:27:22	修改 编辑 删除

共 14 条

10条/页

1 2

前往 1 页

b.菜品信息的分页查询

分页查询将菜品信息展示:难点在于分页查询不仅需要展示属于菜品的字段进行展示,并且还需要展示该菜品的菜品分类的名(属于另一张表的字段),这时候dish就无法满足,就需要一个dto来满足响应的内容。

```

1  @GetMapping("/page")
2  public R<Page> page(int page, int pageSize, String name){
3      //构造分页构造器
4      Page<Dish> pageInfo = new Page<>(page, pageSize);
5      Page<DishDto> dishDtoPage = new Page<>();
6
7      //构造条件构造器
8      LambdaQueryWrapper<Dish> queryWrapper = new LambdaQueryWrapper<>();
9
10     //添加过滤条件

```

```

11     queryWrapper.like(name != null, Dish::getName, name);
12
13     //添加排序条件
14     queryWrapper.orderByDesc(Dish::getUpdateTime);
15
16     //执行查询
17     dishService.page(pageInfo, queryWrapper);
18
19     //对象拷贝
20     BeanUtils.copyProperties(pageInfo, dishDtoPage, "records");
21
22     List<Dish> records = pageInfo.getRecords();
23     List<DishDto> list = records.stream().map((item)->{
24         DishDto dishDto = new DishDto();
25
26         BeanUtils.copyProperties(item, dishDto);
27
28         Long categoryId = item.getCategoryId();
29         Category category = categoryService.getById(categoryId);
30         if(category != null){
31             String categoryName = category.getName();
32             dishDto.setCategoryName(categoryName);
33         }
34         return dishDto;
35     }).collect(Collectors.toList());
36
37     dishDtoPage.setRecords(list);
38
39     return R.success(dishDtoPage);
40 }

```

c.修改菜品信息

1. 菜品分类下拉框中数据展示并回显原菜品分类

将复用新增菜品时同一个请求路径`@GetMapping("/list")`所映射的控制器方法`R<List<Category>> list(Category category)`来达到菜品分类下拉框中数据展示，并且回显原菜品分类，要结合第2点一起才能实现。

2. 将dish基本信息和dishFlavor回显到修改页面

口味做法配置:

口味名 (3个字内) 口味标签 (输入标签回车添加)

忌口

不要葱 X

不要蒜 X

不要香菜 X

不要辣 X

删除

添加口味

这里的逻辑其实和前面的分页查询菜品差不多，都是基本的实体类dish和dish_flavor都无法满足响应给浏览器页面进行展示，所以才需要一个类既有dish的基本信息，又有每个dish对应的dish_flavor，所以会创建一个DishDto类封装并且响应给页面，也是因为后端数据库设计时一(dish)对多(dish_flavor)实体表的对应关系，基本业务逻辑如下：

```
1 //根据id查询菜品信息和对应口味信息
2 public DishDto getByIdWithFlavor(Long id) {
3     //查询菜品基本信息
4     Dish dish = this.getById(id);
5
6
7     DishDto dishDto = new DishDto();
8     BeanUtils.copyProperties(dish, dishDto);
9
10    //查询当前菜品对应口味
11    LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>();
12    queryWrapper.eq(DishFlavor::getDishId, dish.getId());
13    List<DishFlavor> flavors = dishFlavorService.list(queryWrapper);
14    dishDto.setFlavors(flavors);
15
16    return dishDto;
17 }
```

3. 图片的回显到修改页面

复用已经写好的commonController类中的download控制器方法即可。

4. 将修改的信息更新到数据库

更新操作与新增菜品操作的业务逻辑有点不同，更新数据的请求是put请求，而插入数据的请求是post请求，并且如果把新增菜品的功能的业务逻辑复用到更新菜品的功能会出现异常，由于主键唯一无法进行插入数据，并且更新操作是根据唯一id进行更新，所以更新数据操作的业务逻辑的代码如下：（写在dish的业务层）

```
1 @Override
2 public void updateWithFlavor(DishDto dishDto) {
3     //更新dish表
4     this.updateById(dishDto);
5     //更新dish_flavor
6     LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>();
7     queryWrapper.eq(DishFlavor::getDishId, dishDto.getId());
8
9     dishFlavorService.remove(queryWrapper);
10
11    List<DishFlavor> flavors = dishDto.getFlavors();
12
13    flavors = flavors.stream().map((item)->{
14        item.setDishId(dishDto.getId());
15        return item;
16    }).collect(Collectors.toList());
```

```

17
18     dishFlavorService.saveBatch(flavors);
19 }

```

d.批量管理菜品操作

单独停售和删除的操作：

菜品名称	图片	菜品分类	售价	售卖状态	最后操作时间	操作
<input type="checkbox"/> 澳洲龙虾		家常菜	¥ 588	启售	2022-11-16 22:58:22	修改 停售 删除

批量停售和删除的操作：

Q 请输入菜品名称		批量删除 批量启售 批量停售			+ 新建菜品	
<input checked="" type="checkbox"/> 菜品名称	图片	菜品分类	售价	售卖状态	最后操作时间	操作
<input checked="" type="checkbox"/> 澳洲龙虾		家常菜	¥ 588	启售	2022-11-16 22:58:22	修改 停售 删除
<input checked="" type="checkbox"/> 邵阳猪血丸子		湘菜	¥ 30	启售	2022-11-15 22:39:35	修改 停售 删除

并且对正在用户端售卖的菜品无法进行批量删除操作。

将批量删除和修改售卖状态的业务逻辑代码写到DishController层，将菜品与正在售卖的套餐进行关联而无法删除的逻辑写在DishServer层。

```

1  //对菜品批量或者是单个进行停售或者是起售
2  @PostMapping("/status/{status}")
3  public R<String> status(@PathVariable("status") Integer status,@RequestParam List<Long>
4  ids){
5      LambdaQueryWrapper<Dish> queryWrapper = new LambdaQueryWrapper();
6      queryWrapper.in(ids !=null,Dish::getId,ids);
7      //根据数据进行批量查询
8      List<Dish> list = dishService.list(queryWrapper);
9      for (Dish dish : list) {
10         if (dish != null){
11             dish.setStatus(status);
12             dishService.updateById(dish);
13         }
14     }
15     return R.success("售卖状态修改成功");
16 }
17 //
18 @DeleteMapping
19 public R<String> delete(@RequestParam("ids") List<Long> ids){
20     //根据菜品id在stemeal_dish表中查出哪些套餐包含该菜品
21     LambdaQueryWrapper<SetmealDish> setmealDishLambdaQueryWrapper = new
22 LambdaQueryWrapper<>();
23     setmealDishLambdaQueryWrapper.in(SetmealDish::getDishId,ids);
24     List<SetmealDish> SetmealDishList =
25 setMealDishService.list(setmealDishLambdaQueryWrapper);
26     //如果菜品没有关联套餐，直接删除就行 其实下面这个逻辑可以抽离出来，这里我就不抽离了
27     if (SetmealDishList.size() == 0){
28         //这个deleteByIds中已经做了菜品起售不能删除的判断力

```

```

27         dishService.deleteByIds(ids);
28         LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>();
29         queryWrapper.in(DishFlavor::getDishId,ids);
30         dishFlavorService.remove(queryWrapper);
31         return R.success("菜品删除成功");
32     }
33
34     //如果菜品有关联套餐，并且该套餐正在售卖，那么不能删除
35     //得到与删除菜品关联的套餐id
36     ArrayList<Long> Setmeal_idList = new ArrayList<>();
37     for (SetmealDish setmealDish : SetmealDishList) {
38         Long setmealId = setmealDish.getSetmealId();
39         Setmeal_idList.add(setmealId);
40     }
41     //查询出与删除菜品相关联的套餐
42     LambdaQueryWrapper<Setmeal> setmealLambdaQueryWrapper = new LambdaQueryWrapper<>();
43     setmealLambdaQueryWrapper.in(Setmeal::getId,Setmeal_idList);
44     List<Setmeal> setmealList = setMealService.list(setmealLambdaQueryWrapper);
45     //对拿到的所有套餐进行遍历，然后拿到套餐的售卖状态，如果有套餐正在售卖那么删除失败
46     for (Setmeal setmeal : setmealList) {
47         Integer status = setmeal.getStatus();
48         if (status == 1){
49             return R.error("删除的菜品中有关联在售套餐,删除失败!");
50         }
51     }
52
53     //要删除的菜品关联的套餐没有在售，可以删除
54     //这下面的代码并不一定会执行,因为如果前面的for循环中出现status == 1,那么下面的代码就不会再
    执行
55     dishService.deleteByIds(ids);
56     LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>();
57     queryWrapper.in(DishFlavor::getDishId,ids);
58     dishFlavorService.remove(queryWrapper);
59     return R.success("菜品删除成功");
60 }
61
62 @Override
63 @Transactional
64 public void deleteByIds(List<Long> ids) {
65     //构造条件查询器
66     LambdaQueryWrapper<Dish> queryWrapper = new LambdaQueryWrapper<>();
67     //先查询该菜品是否在售卖，如果是则抛出业务异常
68     queryWrapper.in(ids!=null,Dish::getId,ids);
69     List<Dish> list = this.list(queryWrapper);
70     for (Dish dish : list) {
71         Integer status = dish.getStatus();
72         //如果不是在售卖,则可以删除
73         if (status == 0){
74             this.removeById(dish.getId());
75         }else {
76             //此时应该回滚,因为可能前面的删除了，但是后面的是正在售卖
77             throw new CustomException("删除菜品中有正在售卖菜品,无法全部删除");
78         }
79     }
80 }
81

```

6) 套餐管理

新增套餐，分页查询，修改套餐和批量管理套餐的实现套路和逻辑基本和菜品管理模块相同，不再赘述。

7) 订单明细

a. 订单明细的分页查询

用户在用户端进行外卖下单后，可以从订单明细模块中分页实时显示所有用户们的订单信息，并且可以通过订单号，订单下单的开始时间和结束时间两种查询条件进行分页查询。

```
1 @GetMapping("/page")
2 public R<Page> page(int page, int pageSize, String number, String beginTime, String
  endTime){
3     //分页构造器对象
4     Page<Orders> pageInfo = new Page<>(page, pageSize);
5     //构造条件查询对象
6     LambdaQueryWrapper<Orders> queryWrapper = new LambdaQueryWrapper<>();
7
8     //添加查询条件 动态sql 字符串使用StringUtils.isEmpty这个方法来判断
9     //这里使用了范围查询的动态SQL，这里是重点!!!
10    queryWrapper.like(number!=null, Orders::getNumber, number)
11        .gt(StringUtils.isEmpty(beginTime), Orders::getOrderTime, beginTime)
12        .lt(StringUtils.isEmpty(endTime), Orders::getOrderTime, endTime);
13
14    orderService.page(pageInfo, queryWrapper);
15    return R.success(pageInfo);
16 }
```

请输入订单号

开始日期

至

结束日期

查询

订单号	订单状态	用户	手机号	地址	下单时间	实收金额	操作
188686868757575	已派送	lbw	15222954679	兴隆街道	2022-11-17 14:15:00	¥ 1598	查看 完成

共 1 条

10条/页

< 1 >

前往 1 页

b. 查看订单明细和修改订单状态

订单信息

订单号:

188686868757575

订单状态:

已派送

收货人:

lbw

联系电话:

15222954679

地址:

兴隆街道

支付金额:

1598

下单时间:

2022-11-17 14:15:00

将已经完成派送的订单修改状态为订单已完成，代码实现在DishController层：

```
1 @PutMapping
2 public R<String> orderStatusChange(@RequestBody Map<String,String> map){
```

```

3
4     String id = map.get("id");
5     Long orderId = Long.parseLong(id);
6     Integer status = Integer.parseInt(map.get("status"));
7
8     if(orderId == null || status==null){
9         return R.error("传入信息不合法");
10    }
11    Orders orders = orderService.getById(orderId);
12    orders.setStatus(status);
13    orderService.updateById(orders);
14
15    return R.success("订单状态修改成功");
16 }

```

可见订单状态变为已完成。

<div> <input type="text" value="请输入订单号"/> <div> <div>开始日期</div> <div>至</div> <div>结束日期</div> </div> <div>查询</div> </div> <div> <div>订单已完成</div> </div>							
订单号	订单状态	用户	手机号	地址	下单时间	实收金额	操作
188686868757575	已完成	lbw	15222954679	兴隆街道	2022-11-17 14:15:00	¥ 1598	查看
<div> <div>共1条</div> <div>10条/页</div> <div>< ></div> <div>前往</div> <div>1</div> <div>页</div> </div>							

4.2.2 移动端

1) 用户登录

实现方法和Employee一样，给UserController类添加一个login方法。

```

1  /**
2   * 移动端用户登录
3   */
4   @PostMapping("/login")
5   public R<User> login(HttpServletRequest request, @RequestBody User user){
6       //1.将页面提交的pw进行md5加密
7       String password = user.getPassword();
8       password = DigestUtils.md5DigestAsHex(password.getBytes());
9
10      //2.根据页面提交的用户名查询数据库
11      LambdaQueryWrapper<User> queryWrapper = new LambdaQueryWrapper<>();
12      queryWrapper.eq(User::getUsername, user.getUsername());
13      User use = userService.getOne(queryWrapper);
14
15      //3.没有查询到结果返回登陆失败
16      if(use == null){
17          return R.error("此用户不存在");
18      }
19
20      //4.密码比对，不一致返回登陆失败
21      if(!use.getPassword().equals(password)){

```



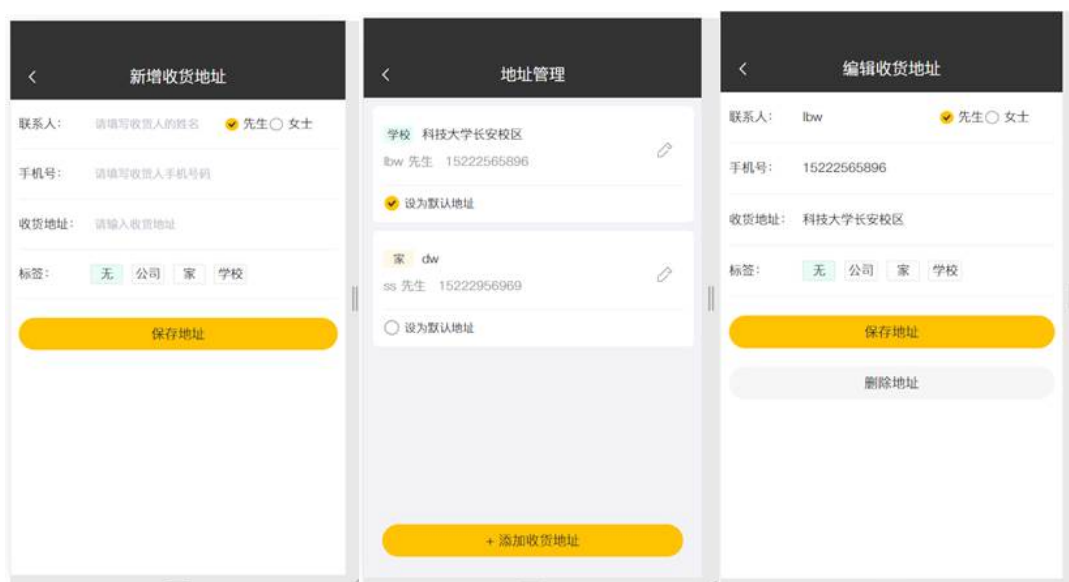
```

22         return R.error("密码错误，登陆失败");
23     }
24
25     //5.查看管理员状态，如果为已禁用，则返回员工已禁用结果
26     if(use.getStatus() == 0){
27         return R.error("账号已禁用");
28     }
29
30     //6.登陆成功，将管理员id存入Session并返回登陆结果
31     request.getSession().setAttribute("user", use.getId());
32     return R.success(use);
33 }

```

2) 导入用户地址簿

地址簿指的是移动端消费者用户的地址信息，用户登录成功后可以维护自己的地址信息。同一个用户可以有多个地址信息，但是只能有一个默认地址。



地址簿表：

```

mysql> desc address_book;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id         | bigint        | NO   | PRI | NULL    |       |
| user_id    | bigint        | NO   |     | NULL    |       |
| consignee  | varchar(50)   | NO   |     | NULL    |       |
| sex        | tinyint       | NO   |     | NULL    |       |
| phone      | varchar(11)   | NO   |     | NULL    |       |
| province_code | varchar(12)   | YES  |     | NULL    |       |
| province_name | varchar(32)   | YES  |     | NULL    |       |
| city_code  | varchar(12)   | YES  |     | NULL    |       |
| city_name   | varchar(32)   | YES  |     | NULL    |       |
| district_code | varchar(12)   | YES  |     | NULL    |       |
| district_name | varchar(32)   | YES  |     | NULL    |       |
| detail     | varchar(200)  | YES  |     | NULL    |       |
| label      | varchar(100)  | YES  |     | NULL    |       |
| is_default | tinyint(1)    | NO   |     | 0       |       |
| create_time | datetime      | NO   |     | NULL    |       |
| update_time | datetime      | NO   |     | NULL    |       |
| create_user | bigint        | NO   |     | NULL    |       |
| update_user | bigint        | NO   |     | NULL    |       |
| is_deleted | int           | NO   |     | 0       |       |
+-----+-----+-----+-----+-----+-----+
19 rows in set (0.01 sec)

```

对应Controller层实现的逻辑:

```
1 //查询指定用户的全部地址
2 @GetMapping("/list")
3 public R<List<AddressBook>> list(AddressBook addressBook) {
4     addressBook.setUserId(BaseContext.getCurrentId());
5     log.info("addressBook:{}", addressBook);
6
7     //条件构造器
8     LambdaQueryWrapper<AddressBook> queryWrapper = new LambdaQueryWrapper<>();
9     queryWrapper.eq(null != addressBook.getUserId(), AddressBook::getUserId,
addressBook.getUserId());
10    queryWrapper.orderByDesc(AddressBook::getUpdateTime);
11
12    //SQL:select * from address_book where user_id = ? order by update_time desc
13    return R.success(addressBookService.list(queryWrapper));
14 }
```

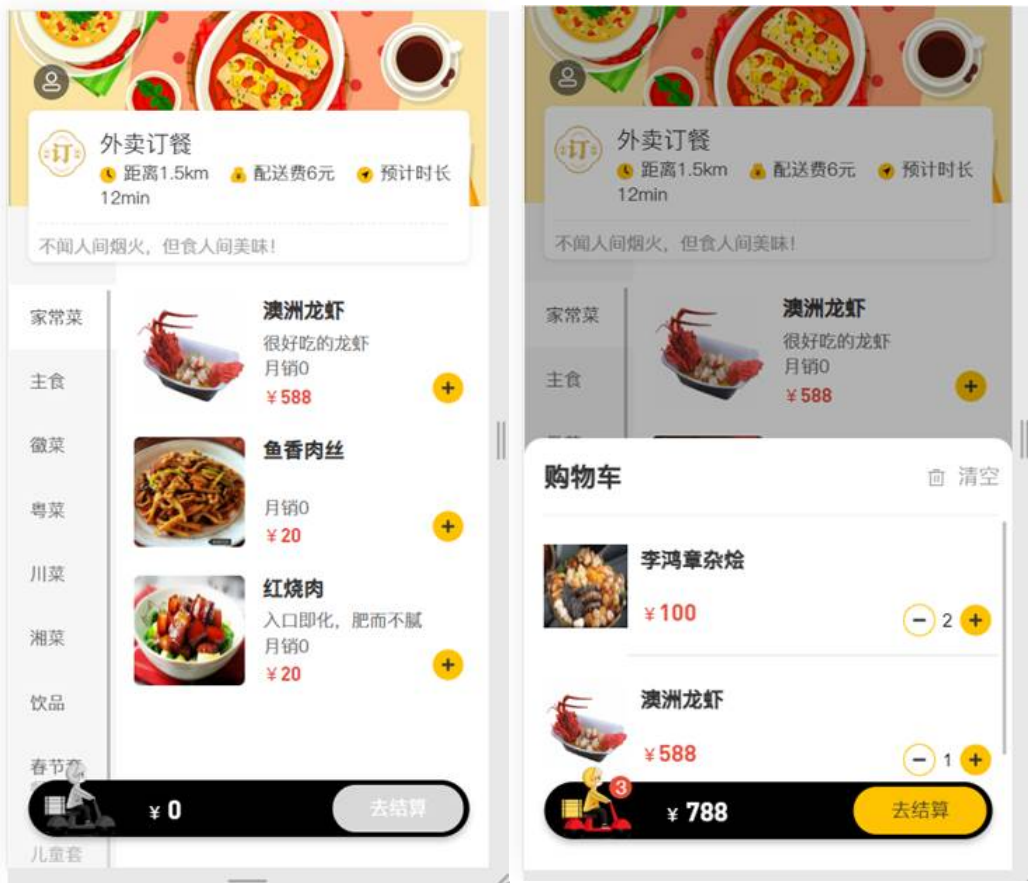
3) 菜品展示

- 页面(front/index.html)发送ajax请求, 获取分类数据(菜品分类和套餐分类)
- 页面发送ajax请求, 获取第一个分类下的菜品或者套餐

开发菜品展示功能, 其实就是在服务端编写代码去处理前端页面发送的这2次请求即可。首页加载完成后, 还发送了一次ajax请求用于加载购物车数据, 从静态json文件获取数据。

```
1 //根据条件查询对应菜品数据
2 @GetMapping("/list")
3 public R<List<Dish>> list(Dish dish){
4     LambdaQueryWrapper<Dish> queryWrapper = new LambdaQueryWrapper<>();
5     queryWrapper.eq(dish.getCategoryId() != null, Dish::getCategoryId,
dish.getCategoryId());
6     queryWrapper.eq(Dish::getStatus, 1);
7     queryWrapper.orderByAsc(Dish::getSort).orderByDesc(Dish::getUpdateTime);
8     List<Dish> list = dishService.list(queryWrapper);
9     return R.success(list);
10 }
```

移动端用户可以将菜品或者套餐添加到购物车。对于菜品来说, 如果设置了口味信息, 则需要选择规格后才能加入购物车; 对于套餐来说, 可以直接点击加号将当前套餐加入购物车。在购物车中可以修改菜品和套餐的数量, 也可以清空购物车。



对应逻辑实现在ShoppingCartController层:

```

1  /**
2   * 添加购物车
3   * @param shoppingCart
4   * @return
5   */
6  @PostMapping("/add")
7  public R<ShoppingCart> add(@RequestBody ShoppingCart shoppingCart){
8      log.info("购物车数据:{}",shoppingCart);
9
10     //设置用户id, 指定当前是哪个用户的购物车数据
11     Long currentId = BaseContext.getCurrentId();
12     shoppingCart.setUserId(currentId);
13
14     Long dishId = shoppingCart.getDishId();
15
16     LambdaQueryWrapper<ShoppingCart> queryWrapper = new LambdaQueryWrapper<>();
17     queryWrapper.eq(ShoppingCart::getUserId,currentId);
18
19     if(dishId != null){
20         //添加到购物车的是菜品
21         queryWrapper.eq(ShoppingCart::getDishId,dishId);
22
23     }else{
24         //添加到购物车的是套餐
25         queryWrapper.eq(ShoppingCart::getSetmealId,shoppingCart.getSetmealId());
26     }
27
28     //查询当前菜品或者套餐是否在购物车中
29     //SQL:select * from shopping_cart where user_id = ? and dish_id/setmeal_id = ?
30     ShoppingCart cartServiceOne = shoppingCartService.getOne(queryWrapper);

```

```
31
32     if(cartServiceOne != null){
33         //如果已经存在,就在原来数量基础上加一
34         Integer number = cartServiceOne.getNumber();
35         cartServiceOne.setNumber(number + 1);
36         shoppingCartService.updateById(cartServiceOne);
37     }else{
38         //如果不存在,则添加到购物车,数量默认就是一
39         shoppingCart.setNumber(1);
40         shoppingCart.setCreateTime(LocalDateTime.now());
41         shoppingCartService.save(shoppingCart);
42         cartServiceOne = shoppingCart;
43     }
44
45     return R.success(cartServiceOne);
46 }
47
48 /**
49  * 查看购物车
50  * @return
51  */
52 @GetMapping("/list")
53 public R<List<ShoppingCart>> list(){
54     log.info("查看购物车...");
55
56     LambdaQueryWrapper<ShoppingCart> queryWrapper = new LambdaQueryWrapper<>();
57     queryWrapper.eq(ShoppingCart::getUserId,BaseContext.getCurrentId());
58     queryWrapper.orderByAsc(ShoppingCart::getCreateTime);
59
60     List<ShoppingCart> list = shoppingCartService.list(queryWrapper);
61
62     return R.success(list);
63 }
64
65 /**
66  * 清空购物车
67  * @return
68  */
69 @DeleteMapping("/clean")
70 public R<String> clean(){
71     //SQL:delete from shopping_cart where user_id = ?
72
73     LambdaQueryWrapper<ShoppingCart> queryWrapper = new LambdaQueryWrapper<>();
74     queryWrapper.eq(ShoppingCart::getUserId,BaseContext.getCurrentId());
75
76     shoppingCartService.remove(queryWrapper);
77
78     return R.success("清空购物车成功");
79 }
```

5) 下单

- 在购物车中点击去结算按钮, 页面跳转到订单确认页面
- 在订单确认页面, 发送ajax请求, 请求服务端获取当前登录用户的默认地址
- 在订单确认页面, 发送ajax请求, 请求服务端获取当前登录用户的购物车数据
- 在订单确认页面点击去支付按钮, 发送ajax请求, 请求服务端完成下单操作

开发用户下单功能, 其实就是在服务端编写代码去处理前端页面发送的请求即可。

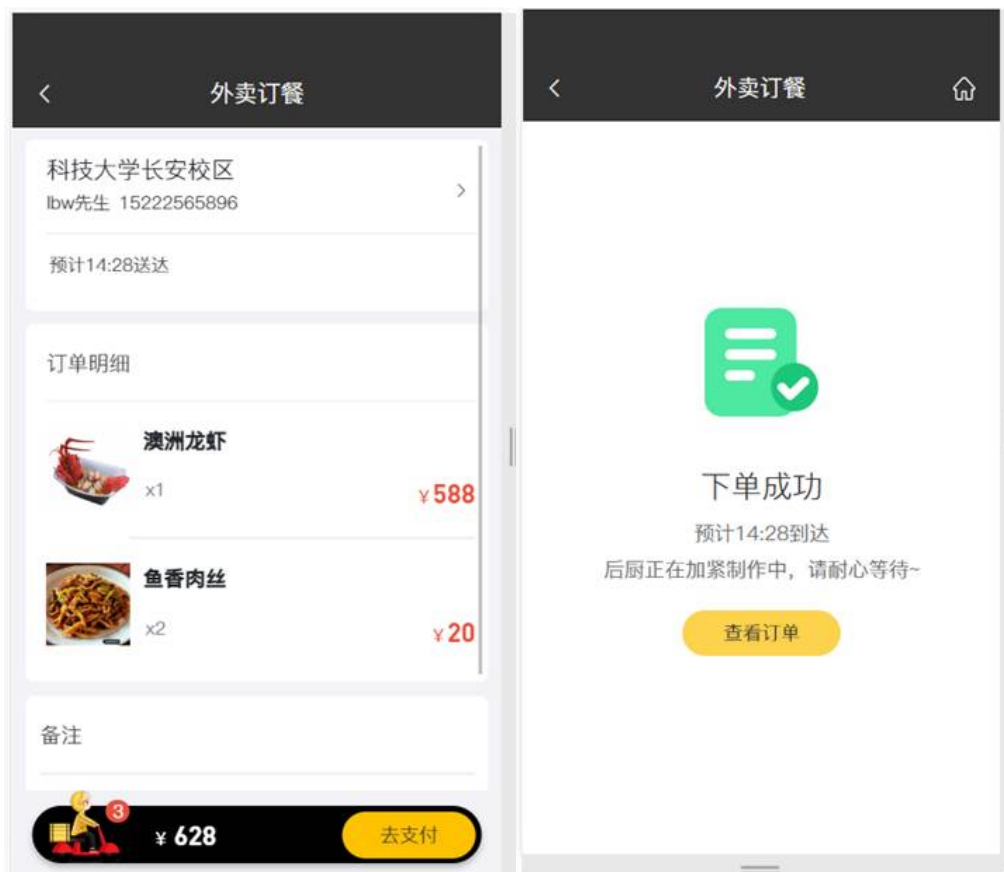
业务实现复杂, 逻辑在OrderServiceImpl, 在Controller层调用:

```
1  @Transactional
2      public void submit(Orders orders) {
3          //获得当前用户id
4          Long userId = BaseContext.getCurrentId();
5
6          //查询当前用户的购物车数据
7          LambdaQueryWrapper<ShoppingCart> wrapper = new LambdaQueryWrapper<>();
8          wrapper.eq(ShoppingCart::getUserId,userId);
9          List<ShoppingCart> shoppingCarts = shoppingCartService.list(wrapper);
10
11         if(shoppingCarts == null || shoppingCarts.size() == 0){
12             throw new CustomException("购物车为空, 不能下单");
13         }
14
15         //查询用户数据
16         User user = userService.getById(userId);
17
18         //查询地址数据
19         Long addressBookId = orders.getAddressBookId();
20         AddressBook addressBook = addressBookService.getById(addressBookId);
21         if(addressBook == null){
22             throw new CustomException("用户地址信息有误, 不能下单");
23         }
24
25         long orderId = IdWorker.getId();//订单号
26
27         AtomicInteger amount = new AtomicInteger(0);
28
29         List<OrderDetail> orderDetails = shoppingCarts.stream().map((item) -> {
30             OrderDetail orderDetail = new OrderDetail();
31             orderDetail.setOrderId(orderId);
32             orderDetail.setNumber(item.getNumber());
33             orderDetail.setDishFlavor(item.getDishFlavor());
34             orderDetail.setDishId(item.getDishId());
35             orderDetail.setSetmealId(item.getSetmealId());
36             orderDetail.setName(item.getName());
37             orderDetail.setImage(item.getImage());
38             orderDetail.setAmount(item.getAmount());
39             amount.addAndGet(item.getAmount().multiply(new
BigDecimal(item.getNumber()).intValue()));
40             return orderDetail;
41         }).collect(Collectors.toList());
42
43
44         orders.setId(orderId);
45         orders.setOrderTime(LocalDateTime.now());
```

```

46     orders.setCheckoutTime(LocalDateTime.now());
47     orders.setStatus(2);
48     orders.setAmount(new BigDecimal(amount.get())); //总金额
49     orders.setUserId(userId);
50     orders.setNumber(String.valueOf(orderId));
51     orders.setUserName(user.getName());
52     orders.setConsignee(addressBook.getConsignee());
53     orders.setPhone(addressBook.getPhone());
54     orders.setAddress((addressBook.getProvinceName() == null ? "" :
addressBook.getProvinceName())
55         + (addressBook.getCityName() == null ? "" : addressBook.getCityName())
56         + (addressBook.getDistrictName() == null ? "" :
addressBook.getDistrictName())
57         + (addressBook.getDetail() == null ? "" : addressBook.getDetail()));
58     //向订单表插入数据，一条数据
59     this.save(orders);
60
61     //向订单明细表插入数据，多条数据
62     orderDetailService.saveBatch(orderDetails);
63
64     //清空购物车数据
65     shoppingCartService.remove(wrapper);
66 }

```



五、总结与思考

由于实训的时间有限，因此我们只能在较短的时间内完成最主要的功能。关于系统测试的部分主要见讲解视频，同时在完成每一项功能之后，都会对其进行测试。但是我们一致认为我们还有许多值得改进的地方，列出如下：

1. 此次实现的是web页面的形式来展现相应的功能，但是实际上大家点外卖大多是在微信小程序或是相应的APP上进行实现。在较短的时间内掌握微信小程序的开发、Android开发或是IOS开发，对于我们目前的水平与精力来说着实有限，因此我们可以在课余时间内学习这方面的有关知识对齐进行完善。
2. 同时，老师利用两节课的时间为我们介绍了低代码技术。利用低代码平台，我们可以很快的对页面以及相应的功能进行开发；而不是利用java对前后端的一个按钮的点击以及数据的传送都要调试半天。在低代码平台下，利用鼠标的点击以及设置相应的命令，可以屏蔽掉相应的前后端交互的大量代码，从而可以快速构建出一个系统，并将其部署到服务器上进行产品的多次迭代。
3. 对于送餐员的数据表schema的构建以及相应功能的实现：送餐员可以查看没有配送的订单，并选择合适的订单进行接单。对于已经有配送员接单的订单，将不会在接单页面进行显示。并且把订单的状态实时更新到顾客端。