数据库系统实验课程设计项目报告

引言

我们小组实现的是一个餐厅点餐系统，该应用系统基于vue + springboot实现，在该数据库的基础上搭建了Web界面，实现了一个点餐系统，包括菜品信息管理、订单管理、餐桌管理等功能。

以下是小组成员及分工内容

姓名	学号	分工
熊明	20305055	主要负责前后端开发、数据库设计，参与实验报告的编写
洪俊东	21307333	主要负责产品设计、数据库实现、编写实验报告

设计目的

该餐厅点餐系统的设计旨在提供一个方便、高效的点餐解决方案，以改善顾客点餐体验，并提升餐厅内部运营效率。此系统旨在实现菜单浏览、下订单、支付和订单管理等功能。

设计要求

1. 针对给定的系统进行需求分析，设计系统结构图和系统功能模块图；
2. 针对需求分析，画出E-R图表示的概念模型，并将其转换为至少满足3NF的关系模式，设计较为合理的数据库模式；
3. 系统中应能体现对数据库的保护（安全性、完整性等）；
4. 系统应该有较为友好的用户界面；

项目环境

- 编程语言：Java
- 数据库：MySQL
- 开发环境（相关依赖见源码readme）
    - jdk17
    - mysql8
    - vue2

- springboot3

概要设计

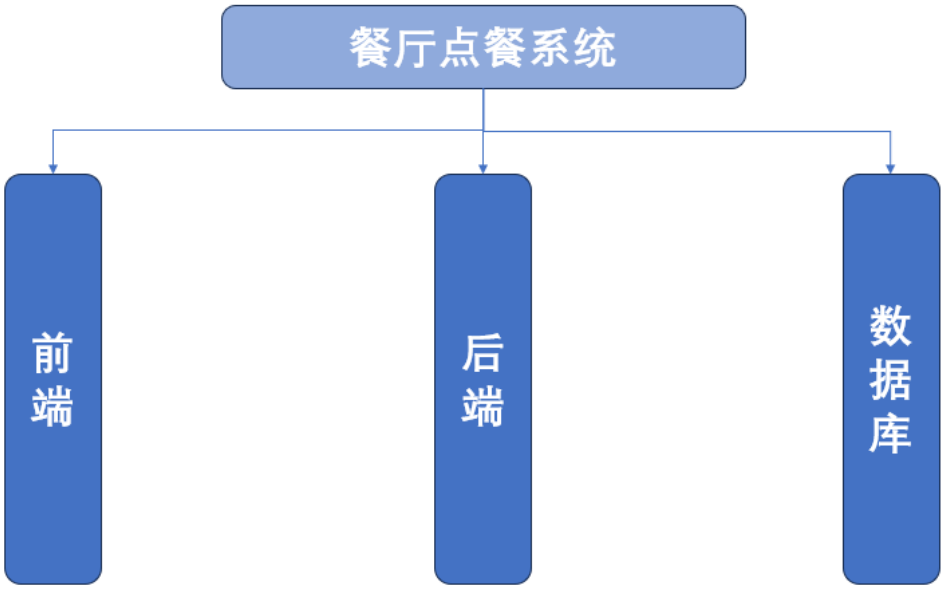
需求分析

- **菜单模块**：展示餐厅菜单，包括菜品详情、价格等信息。
- **订单模块**：允许用户浏览菜单并下订单，管理购物车，进行支付等。
- **管理模块**：提供管理员权限，包括对菜单的管理、订单的监控等功能。

结构设计

系统将采用**前后端分离**的架构，前端通过Web页面与后端API进行交互。系统结构将包括：

- **前端**：负责用户界面展示和用户交互。
- **后端**：负责处理前端请求、逻辑处理和数据交互。
- **数据库**：存储用户信息、菜单数据、订单信息等。

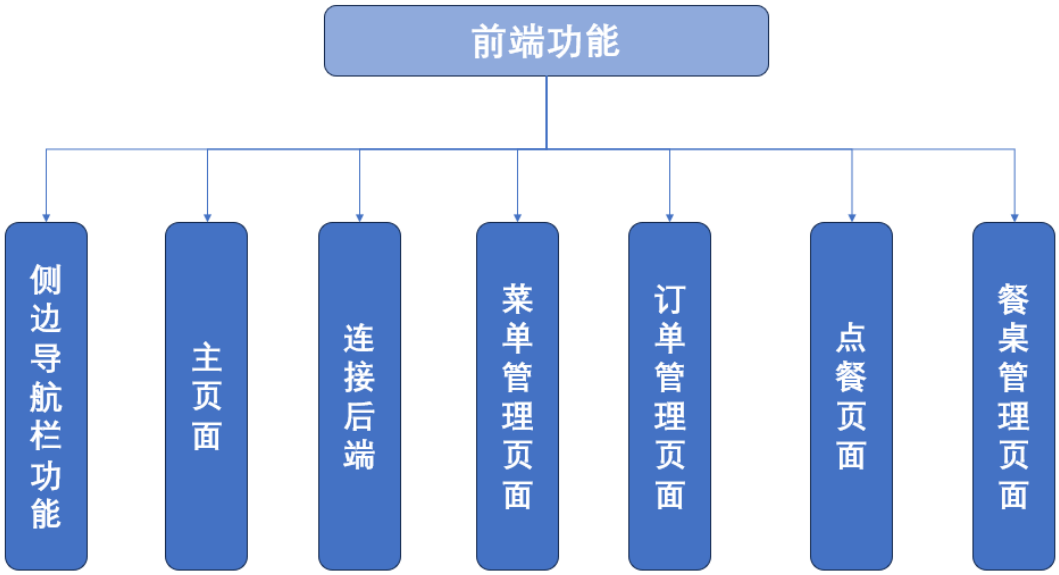


模块设计

前端

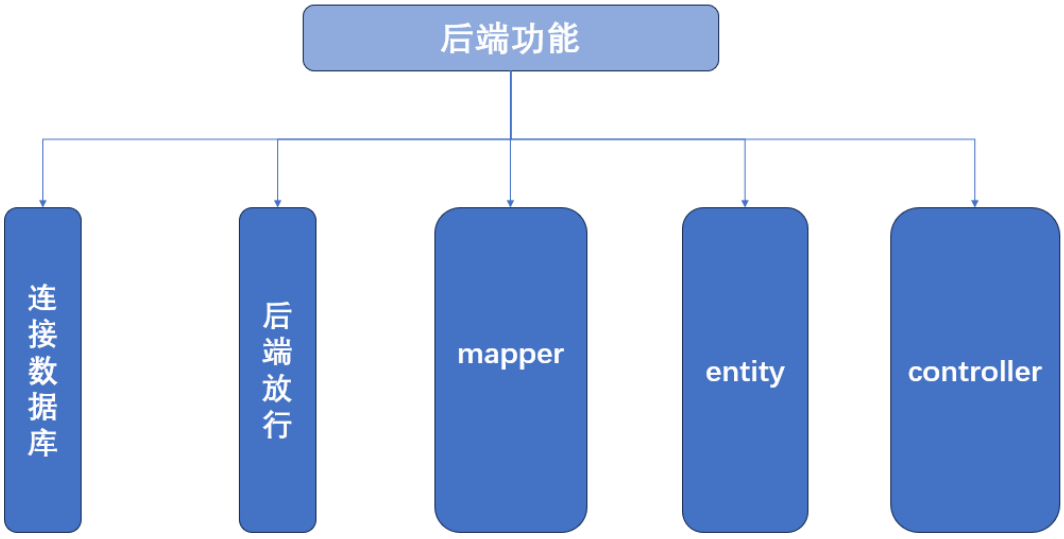
- 侧边导航栏功能：在侧边栏实现导航。
- home页面：实现了一个跑马灯展示菜品图片。
- 连接后端：利用`axios`与后端连接。
- menu菜单管理页面：实现对`menu`的管理。
- order订单管理页面：分左右页分别展示订单大致信息和详细信息。
- choices点餐页面：分左右页分别展示菜单和已选菜品。

- tablelist餐桌管理页面：实现对`table`的管理。



后端

- 连接数据库：与数据库连接。
- 后端放行：基于`WebMvcConfigurer`类实现跨域。
- mapper：利用Mapper注解，并将该接口类继承于BaseMapper即可调用已有的sql语句接口。
- entity：建立实体类，该类代表一张表或者表的集合。
- controller：实现前后端的交互。



详细设计

数据库设计

根据系统需求分析的结果，我们设计了restaurant数据库，其包含以下表：

餐桌(table)

为实现餐桌管理，餐桌除了作为主键的餐桌号`tid`，还有占有信息`used`和预定信息`reserved`，分别表示桌子是否被用和桌子是否被预定。

```
CREATE TABLE `table`  (
  `tid` int NOT NULL COMMENT '餐桌号',
  `used` tinyint NULL COMMENT '桌子是否被用',
  `reserved` tinyint NULL COMMENT '桌子是否被预定',
  PRIMARY KEY (`tid`)
);
```

菜单(menu)

菜单表需要菜品号、菜名、单价、库存来实现菜品信息管理。

```
CREATE TABLE `menu`  (
  `mid` int AUTO_INCREMENT NOT NULL COMMENT '菜的序号',
  `fname` varchar(255) NULL COMMENT '菜名',
  `price` FLOAT NOT NULL COMMENT '菜价',
  `rest` int NULL COMMENT '库存',
  PRIMARY KEY (`mid`)
);
```

订单(ordering)

订单表包含主键**订单号no**、外键**餐桌号tid**，以及总金额price。可以实现订单管理、结账等功能。

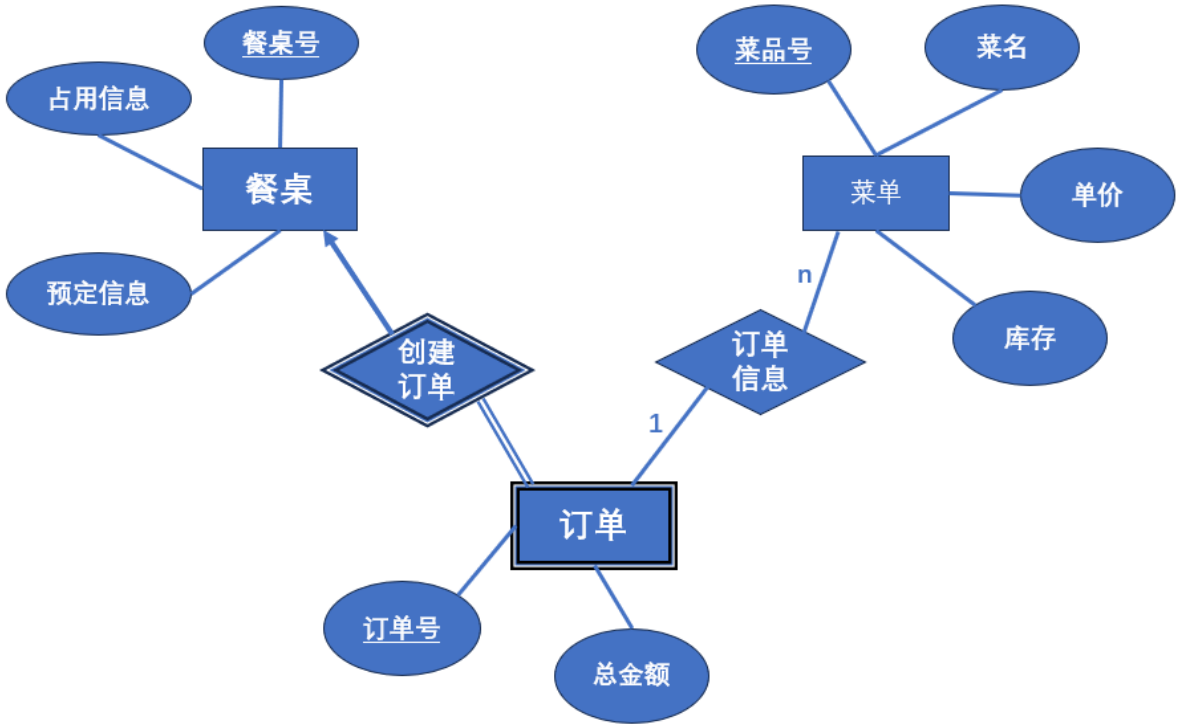
```
CREATE TABLE `ordering`  (
  `no` int AUTO_INCREMENT NOT NULL COMMENT '订单序号',
  `tid` int NOT NULL,
  `price` FLOAT,
  PRIMARY KEY (`no`)
);
ALTER TABLE `ordering` ADD CONSTRAINT `fk_ordering_table_1`
FOREIGN KEY (`tid`) REFERENCES `table` (`tid`);
```

订单信息(orderinfo)

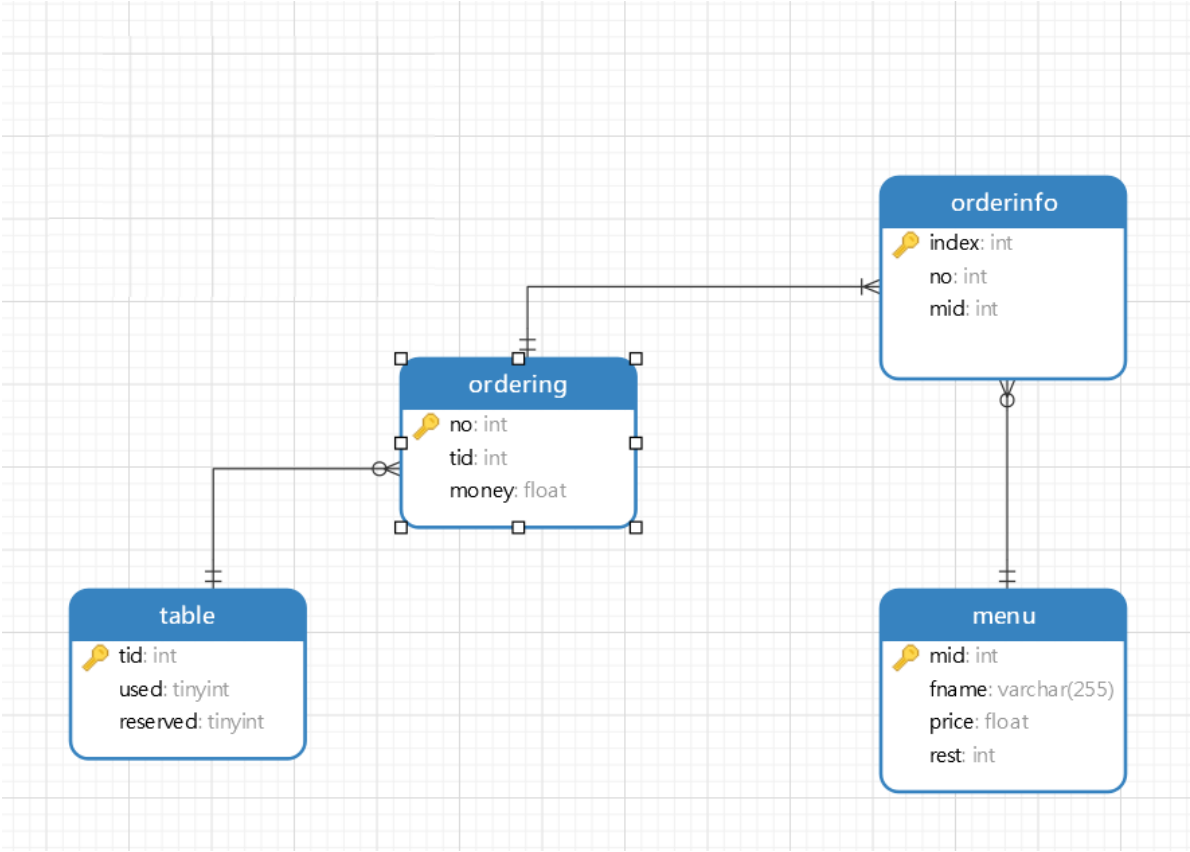
订单信息表示某一订单包含的菜品，一个订单可以包含多个菜品。

```
CREATE TABLE `orderinfo` (
    `index` INT AUTO_INCREMENT NOT NULL,
    `no` INT NOT NULL,
    `mid` int NOT NULL,
     PRIMARY KEY (`index`)
);
ALTER TABLE `orderinfo` ADD CONSTRAINT `fk_orderinfo_menu_1`
FOREIGN KEY (`mid`) REFERENCES `menu` (`mid`);
ALTER TABLE `orderinfo` ADD CONSTRAINT `fk_orderinfo_ordering_1`
FOREIGN KEY (`no`) REFERENCES `ordering` (`no`);
```

该数据库的ER图如下所示



对应的关系模型如下



功能模块实现

前端

修改前端端口为8081，否则会与后端冲突。修改配置文件`vue.config.js`

```
const { defineConfig } = require('@vue/cli-service')
// vue.config.js 配置说明
//官方vue.config.js 参考文档 https://cli.vuejs.org/zh/config/#css-loaderoptions
// 这里只列一部分，具体配置参考文档
module.exports = {
  // 部署生产环境和开发环境下的URL。
  // 默认情况下，Vue CLI 会假设你的应用是被部署在一个域名的根路径上
  //例如 https://www.my-app.com/。如果应用被部署在一个子路径上，你就需要用这个选项指定这个子路径。例如，如果你的应用被部署在 https://www.my-app.com/my-app/，则设置 baseUrl 为 /my-app/。
      //baseUrl 从 Vue CLI 3.3 起已弃用，请使用publicPath
    //baseUrl: process.env.NODE_ENV === "production" ? "./" : "/",
    publicPath: process.env.NODE_ENV === "production" ? "./" : "/",

  // outputDir: 在npm run build 或 yarn build 时 ，生成文件的目录名称（要和baseUrl的生产环境路径一致）
  outputDir: "mycli3",
```

```
  //用于放置生成的静态资源 (js、css、img、fonts) 的；（项目打包之后，静态资源会放在这个文件夹下）
  assetsDir: "assets",
  //指定生成的 index.html 的输出路径  (打包之后，改变系统默认的index.html的文件名)
  // indexPath: "myIndex.html",
  //默认情况下，生成的静态资源在它们的文件名中包含了 hash 以便更好的控制缓存。你可以通过将这个选项设为 false 来关闭文件名哈希。(false的时候就是让原来的文件名不改变)
  filenameHashing: false,

  //   lintOnSave：{ type:Boolean default:true } 问你是否使用eslint
  lintOnSave: true,
  //如果你想要在生产构建时禁用 eslint-loader，你可以用如下配置
  // lintOnSave: process.env.NODE_ENV !== 'production',

  //是否使用包含运行时编译器的 Vue 构建版本。设置为 true 后你就可以在 Vue 组件中使用 template 选项了，但是这会让你的应用额外增加 10kb 左右。(默认false)
  // runtimeCompiler: false,

  /**
   * 如果你不需要生产环境的 source map，可以将其设置为 false 以加速生产环境构建。
   *  打包之后发现map文件过大，项目文件体积很大，设置为false就可以不输出map文件
   *  map文件的作用在于：项目打包后，代码都是经过压缩加密的，如果运行时报错，输出的错误信息无法准确得知是哪里的代码报错。
   *  有了map就可以像未加密的代码一样，准确的输出是哪一行哪一列有错。
   * */
  productionSourceMap: false,

  // 它支持webPack-dev-server的所有选项
  devServer: {
    host: "localhost",
    port: 8081, // 端口号
    https: false, // https:{type:Boolean}
    open: true, //配置自动启动浏览器
    // proxy: 'http://localhost:4000' // 配置跨域处理,只有一个代理

    // 配置多个代理
    proxy: {
      "/api": {
        target:
"https://www.joinpay.com/trade/uniPayApi.action",// 要访问的接口域名
        ws: true,// 是否启用websockets
```

```
        changeOrigin: true, //开启代理：在本地会创建一个虚拟服务端，然后发送请求的数据，并同时接收请求的数据，这样服务端和服务端进行数据的交互就不会有跨域问题
        pathRewrite: {
            '^/api': '' //这里理解成用'/api'代替target里面的地址,比如我要调用'http://40.00.100.100:3002/user/add'，直接写'/api/user/add'即可
          }
      }
    }
  }
};
```

侧边栏导航

利用`VueRouter`将侧边栏的对应路径填写到对应的出口

```
const routes = [
    {
        // 主路由
        path: '/',
        component: Main,
        redirect:'/home', //重定向
        children: [
            // 子路由
            { path: 'home', component: Home }, // 首页
            { path: 'menu', component: Menu }, // 菜品管理
            { path: 'order', component: Order }, // 订单管理
            { path: 'choice', component: Choices},
            { path: 'tablelist', component: TableList},

        ]
    }
  ]
```

将其路由出口放置在main.vue中，并且在CommonAside中定义方法

```
clickMenu(item) {
            console.log(item) // 打印item
            // 当当前页面的路由与跳转路由不一致才允许跳转
            if (this.$route.path !== item.path && !(this.$route.path === '/home' && (item.path === '/'))) {
                this.$router.push(item.path)
            }
        }
```

以上代码实现了对于`/home`和已经在该页面重新点击该页面跳转的两种情况进行了特判

home页面

首页窗口是打开程序最先跳转的地方，所以实现了一个跑马灯展示菜品图片

```
<template>
    <div class="index">
        <!-- 跑马灯   未实现自适应-->
        <div>
            <el-carousel :interval="4000" type="card"
height="500px">
                <el-carousel-item v-for="item in imgList"
:key="item.id">
                    <img :src="item.idView" class="image">
                </el-carousel-item>
            </el-carousel>
            <h1>欢迎来到餐厅!!!</h1>
        </div>
    </div>
</template>
<script>
export default {
    data() {
        return {
            imgList: [
                { id: 0, idView: require('../assets/food1.jpg')
},
                { id: 1, idView: require('../assets/food2.jpg')
},
                { id: 2, idView: require('../assets/food3.jpg')
},
                { id: 3, idView: require('../assets/food4.jpg')
},
                { id: 4, idView: require('../assets/food5.jpg') }
            ]
        };
        methods: {
        }
    }
}
</script>
```

正确填写图片路径即可得到结果

连接后端

利用axios连接后端。在utils文件夹下新建request.js文件，配置后端接口并且添加响应拦截器，这里配置可以参照axios官方文档

```
import axios from "axios";
// 创建axios实例
const http = axios.create({
    // 通用请求地址前缀
    baseURL:'http://localhost:8080',
    timeout:10000, // 超时时间
})

// 添加请求拦截器
http.interceptors.request.use(function (config) {
    // 在发送请求之前做些什么
    return config;
  }, function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  });

// 添加响应拦截器
http.interceptors.response.use(function (response) {
    // 2xx 范围内的状态码都会触发该函数。
    // 对响应数据做点什么
    return response;
  }, function (error) {
    // 超出 2xx 范围的状态码都会触发该函数。
    // 对响应错误做点什么
    return Promise.reject(error);
  });

export default http
```

在api文件夹下创建index.js文件，专门存放前端访问后端的api，这里展示一部分

```
import http from "@/utils/request";
// table list
// 请求数据
export const getTableData = () =>{
    // 返回一个promise对象
    return http.get('/tablelist')
}
```

```
// 添加数据
export const addTable =(data) =>{
    return http.post('/tablelist/addtable',data)
}
// 修改数据
export const editTable=(data) =>{
    return http.post('/tablelist/edittable',data)
}
// 删除数据
export const delectTable=(data) =>{
    return http.post('/tablelist/delecttable',data)
}
...
```

menu菜单管理页面

1. 获取后端数据库的信息

   利用前面提到的api，获取后端的文件，并且放在menudata中以便展示

   ```
   getList() {
               // 获取列表数据
               getMenuData().then(data => data.data).then((data)
   => {
                   console.log(data)
                   this.menuData = data
               })
           },
   ```

   将getlist放置在mounted中，在页面被打开之后自动调用该函数，实现获取数据

2. 将后端传过来的数据展示在el-table中

   ```
   <el-table :data="menuData" border size="medium" height="520"
   style="width: 100%">
                   <el-table-column prop="mid" label="菜序号"
   width="100">
                   </el-table-column>
                   <el-table-column prop="fname" label="菜名"
   width="150">
                   </el-table-column>
                   <el-table-column prop="price" label="价格"
   width="100">
                   </el-table-column>
                   <el-table-column prop="rest" label="库存"
   width="100">
   ```

```
                </el-table-column>
                <!-- fixed="right" -->
                <el-table-column label="操作">
                    <template slot-scope="scope">
                        <el-button
@click="handleEdit(scope.row)" type="text" size="medium">修改
</el-button>
                        <el-button
@click="handleDelete(scope.row)" type="text" size="medium">删除
</el-button>
                    </template>
                </el-table-column>
            </el-table>
```

3. 收集填写信息

   利用el-dialog组件，收集填写的信息，dialogVisible控制打开和关闭

```
<el-dialog title="提示" :visible.sync="dialogVisible"
width="30%" :befor-close="handleClose">
                <!-- 表单信息 -->
                <el-form ref="form" :rules="rules"
:model="form" label-width="80px">
                    <el-form-item label="菜品序号" prop="mid">
                        <el-input placeholder="输入菜品序号" v-
model="form.mid"></el-input>
                    </el-form-item>
                    <el-form-item label="菜名" prop="fname">
                        <el-input placeholder="输入菜品名称" v-
model="form.fname"></el-input>
                    </el-form-item>
                    <el-form-item label="价格" prop="price">
                        <el-input placeholder="输入价格" v-
model="form.price"></el-input>
                    </el-form-item>
                    <el-form-item label="库存" prop="rest">
                        <!-- <el-input placeholder="输入库存数
目" v-model="form.rest"></el-input> -->
                        <el-input-number v-model="form.rest"
@change="handleChange" :min="0" :max="100" label="描述文字">
</el-input-number>
                    </el-form-item>
                </el-form>
                <span slot="footer" class="dialog-footer">
```

```
                    <el-button @click="cancel">取 消</el-
button>
                    <el-button type="primary" @click="submit">
确 定</el-button>
                </span>
            </el-dialog>
```

将写好的信息放在form中，并利用post请求传递给后端以添加数据。

4. 添加操作

点击添加按钮，打开dialog填写信息，因为填写和修改共用一个表单，所以将表单逻辑复用如下：

```
// 点击提交时调用
submit(){
            this.$refs.form.validate((valid)=>{
                if(valid){
                    this.form.mid = parseInt(this.form.mid,10)
                    if(this.modalType==0){
                        addMenu(this.form).then(()=>{
                            // console.log(val,'val')
                            this.getList()
                        })
                    }else{
                        editMenu(this.form).then(()=>{
                            this.getList()
                        })
                    }
                    this.handleClose()
                }
            })
        },
```

modalType初始为0，表示为添加操作，为1则是修改操作

5. 修改操作

利用scope.row获取该行的信息，然后利用深复制将form修改为该行的信息以便修改方便，最后修改成功以后再将数据推向后端

```
// 点击按钮时调用
handleEdit(row) {
            this.modalType = 1
            this.dialogVisible = true
            // 对当前行数据进行深拷贝
            this.form = JSON.parse(JSON.stringify(row))
        },
```

6. 删除操作

```
handleDelete(row) {
            this.$confirm('此操作将永久删除该菜品, 是否继续?', '提
示', {
                confirmButtonText: '确定',
                cancelButtonText: '取消',
                type: 'warning'
            }).then(() => {
                row.mid = parseInt(row.mid, 10)
                delectMenu(row).then(() => {
                    this.$message({
                        type: 'success',
                        message: '删除成功!'
                    });
                    // 重新获取列表接口
                    this.getList()
                })
            }).catch(() => {
                this.$message({
                    type: 'info',
                    message: '已取消删除'
                });
            });
        },
```

删除时会跳出提示框提示是否确认删除

choices点餐页面

分为两张表，左表表示菜单供人选择，右表表示已选菜品

1. 展示菜单数据逻辑与menu类似
2. 选择餐桌逻辑，如果该餐桌被预定或者被使用则不允许选择，然后在点击提交之后将该餐桌状态改为被占用。将其数据传给value.tid

```
<el-select v-model="value.tid" placeholder="请选择餐桌号">
                <el-option v-for="item in tableData"
:key="item.tid" :label="item.tid" :value="item.tid"
                    :disabled="item.used === 1 ||
item.reserved === 1">
                </el-option>
            </el-select>
```

3. 利用变量tempData来实现暂存功能

   跟上面menu修改的逻辑一样，点击添加按钮后，将该行元素添加到tempData中，并将其展示在右表中

   1. 右表清空

      直接将tempData赋值为空

   2. 右表删除

      遍历tempData元素，将符合scope.row.mid的第一个元素删除

4. 右表提交

```
submit() {
            this.value.tid = parseInt(this.value.tid, 10)
            this.nums = this.tempData.length
            //
            this.getmoney()
            this.value.money = parseFloat(this.value.money)
            this.value.food = this.tempData
            console.log(this.value.tid,this.value.money)
            // 传给后端
            addorder(this.value).then(() => {
                // 重新获取列表接口
                this.getTList()
                this.getTList()
            })
            this.tempData = []
            // console.log(this.value.tid)
            this.reset()
        },
```

这里需要同时向ordering（订单表）和orderinfo（订单详情）表提交数据，所以新建了一个变量格式，将其传给后端，剩下的处理交由后端操作。提交过后，将对应菜品库存相应减少，此逻辑在后端实现，所以提交过后需手动刷新。

```
value: {
                tid: '',
                nums: 0,
                money: 0,
                food: [],
            },
```

ordering订单页面

跟choices页面一样，分为左右两表，左表用来展示订单的大致信息（订单号，餐桌号，总价），展示逻辑和menu类似。

1. 详细按钮的逻辑

```
moreorder(row){
            getorderinfo(row.no).then(data
=>data.data).then((data)=>{
                this.orderinfo = data
            })
        },
```

点击详细按钮后，将搜索对应行的no，将其返回的信息传给右表所展示的orderinfo之中。

这里注意

```
export const getorderinfo =(no)=>{
    return http.get(`/order/orderinfo?no=${no}`)
}
```

这里使用了字符串拼接将row.no传递给了后端

2. 删除按钮的逻辑与上面删除逻辑类似

tablelist餐桌管理

1. 展示tablelist与展示menu逻辑一致
2. 删除餐桌逻辑也与menu逻辑一致
3. 修改逻辑也一致

后端

连接数据库

在application.properties中填写配置

```
#spring.devtools.restart.enabled=true
#spring.devtools.restart.additional-paths=src/main/java
#spring.devtools.restart.exclude=static/**

spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/restaurant?
useSSl=false
spring.datasource.username=root
spring.datasource.password=x
mybatis-plus.configuration.log-
impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

用户名和密码基于本机密码修改

后端放行

基于WebMvcConfigurer类实现跨域

```
package com.example.mpdemo.common;

import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.CorsRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigure
r;

// 跨域
@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry){
        registry.addMapping("/**")
                // 是否发送Cookie
                .allowCredentials(true)
                // 放行哪些原始域
                .allowedOriginPatterns("*")
                .allowedMethods(new String[]
{"GET","POST","PUT","DELETE"})
                .allowedHeaders("*")
                .exposedHeaders("*");
```

```
    }
}
```

mapper建立

利用Mapper注解，并将该接口类继承于BaseMapper即可调用已有的sql语句接口，如果需要自己写sql语句的话，需要利用注解，在注解里写sql

```
@Mapper
public interface OrderInfoMapper extends BaseMapper<OrderInfo> {
    @Select("select max(`index`) from orderinfo")
    int selectindex();
    @Delete("delete from orderinfo where `no`=#{no}")
    int delectorderinfo(int no);
}
```

entity实体类建立

该类代表一张表或者表的集合，以table表举例，一般来说该类的属性名与表名一直，如果不一致需要加注解`@TableField("columnname")`

```
@TableName("`table`")
public class Table {
    private int tid;
    private int used;
    private int reserved;

    public int getUsed() {
        return used;
    }

    @Override
    public String toString() {
        return "Table{" +
                "tid=" + tid +
                ", reserved=" + reserved +
                ", used=" + used +
                '}';
    }

    public void setUsed(int used) {
        this.used = used;
    }

    public int getTid() {
```

```
        return tid;
    }

    public void setTid(int tid) {
        this.tid = tid;
    }


    public int getReserved() {
        return reserved;
    }

    public void setReserved(int reserved) {
        this.reserved = reserved;
    }


}
```

controller建立

这里是前后端交互的地方，需要将之前的mapper注入在此处，然后需要写对应接口的url，以tablecontroller为例

```
@RestController
public class TableController {
    @Autowired
    private TableMapper tableMapper;//注解为自动注入的意思
    @GetMapping("/tablelist")
    List<Table> query(){
        return tableMapper.selecttablelist();
    }
    @PostMapping("/tablelist/addtable")
    int addtable(@RequestBody Table t){
//        System.out.println(t.toString());
        return tableMapper.insert(t);
    }
    @PostMapping("/tablelist/edittable")
    int edittable(@RequestBody Table t){
        return tableMapper.updateBytid(t);
    }
    @PostMapping("/tablelist/delecttable")
    int delecttable(@RequestBody Table t) {
        return tableMapper.delectBytid(t);
    }
```

```
    @PostMapping("/tablelist/clear")
    int cleartableused(@RequestBody Table t){
        return tableMapper.changeTableStatusTo0(t.getTid());
    }
}
```

get请求和post请求分开，并且有对应的url。一般前端传回来的格式为json，索引需要加@RequestBody来标明其为json格式，springboot会自动将其放在实体类对应的位置中，前提是传回来的数据名与类的属性名要一致。

特殊的values实现

```
@PostMapping("/choice/orderingsubmit")
    void tranform(@RequestBody Values v){
        int tid = v.getTid();
        System.out.println(tid);
        tableMapper.changeTableStatus(tid);
        float price = v.getMoney();
        int no = orderingMapper.getmaxno()+1;
        int index = orderInfoMapper.selectindex()+1;
        Ordering o = new Ordering(no,tid,price);
        orderingMapper.insert(o);
        int nums = v.getNums();
//        System.out.println(nums);
        List<Menu> me = v.getFood();
//        System.out.println(me.get(1).toString());
        for(int i=0;i<me.size();i++){
            Menu temp = me.get(i);
            System.out.println(temp.toString());
            OrderInfo oi = new OrderInfo(index,no,temp.getMid(),
temp.getFname());
            index++;
//            System.out.println(oi);
            menuMapper.countRestBymid(temp.getMid());
            orderInfoMapper.insert(oi);
        }
    }
```

该接口实现了将前端传回来的values传递给values类，然后再将其拆分给ordering和orderinfo两个表中存储。

调试与运行结果

这次实验主要利用的技术栈有springboot和vue。在编写后端程序时，出现了jdbc正常但是连接不上数据库的情况。后来经过翻阅资料，springboot现在基本都是第三版本，而最新版的mybatis-plus中内嵌的mybatis版本过低，导致连接不上。最后手动导入mybatis 3.0.3版本解决了该问题。

前端vue框架主要使用了element-ui组件来构建前端页面。过程中因为不熟悉前端html和js的逻辑，踩了很多坑。比如循环js数组元素放进dialog里面等。最后查找资料解决了该问题。

首页图片跑马灯还未做自适应，照片是固定大小，没有窗口变化而变化。

最终运行结果如下（主页面）

总结

在本次数据库课程设计项目中，我们选择了实现餐厅点餐系统。根据设计目的和设计要求，对餐厅点餐系统进行需求分析，采用前后端分离的架构，运用模块化设计的技巧，将一个大的目标分解成多个小问题去解决。根据这学期所学的数据库相关知识，我们设计了一个restaurant数据库，并根据我们的设计绘制ER图和关系模型图。在每个模块实现后，将它们汇总起来，就得到了我们最终的餐厅点餐系统。

然而，我们的工作也存在不足之处，比如没有设置顾客和管理员的区分等，为此，我们总结了以下几点未来工作的方向：

- 实现成员与管理员的区分，即实现user表以做用户管理。
- 实现搜索订单，订单分页功能，方便查看订单。
- 实现对已有订单进行加菜的操作，即加点。

  ……

附录

因为时间有限没有部署到云端，通过内网穿透可以访问，因为域名是动态的，并且该程序一直运行在本机也不合适，所以选择跟老师约时间开内网穿透供老师检查。

我们还以demo的形式展示最终实验结果，见本目录下的`demo.mp4`

源代码已上传Github https://github.com/MightyXM/DataBase_Final-_Project.git