

# 瑞吉外卖项目

## 一、软件开发整体介绍

### 1.1 软件开发流程

#### 1. 需求分析

确定产品原型，需求规格说明书

#### 2. 设计

设计产品文档，UI界面设计，概要设计，详细设计，数据库设计

#### 3. 编码

项目代码、单元测试，一般由开发人员完成

#### 4. 测试

设计测试用例，编写测试报告

#### 5. 上线运维

软件环境安装，配置

### 1.2 角色分工

一般规范的角色分工有：

- 项目经理：对整个项目负责，任务分配，把控进度
- 产品经理：进行需求调研，输出需求调研文档，产品原型等
- UI设计师：根据产品原型输出界面效果图
- 架构师：项目整体架构设计，技术选型等
- 开发工程师：代码实现
- 测试工程师：编写测试用例，输出测试报告
- 运维工程师：软件环境搭建、项目上线

### 1.3 软件环境

一般分为如下几种：

- 开发环境（development）：开发人员在开发阶段使用的环境，一般外部用户无法访问
- 测试环境（testing）：专门给测试人员使用的环境，用于测试项目，一般外部用户无法访问
- 生产环境（production）：即线上环境，正式提供对外服务的环境

## 二、项目整体需求

### 2.1 项目介绍

专门为餐饮定制的一款软件产品，包括系统管理后台和移动端应用两部分

- 系统管理后台主要提供给餐饮企业内部员工使用，可以对餐厅的菜品、套餐、订单等进行管理维护
- 移动端主要提供给消费者使用，可以在线浏览菜品，添加购物车，下单等

基本开发步骤：

- 第一期主要实现基本需求，其中移动端通过H5实现，用户可以通过手机浏览器访问
- 第二期主要针对移动端应用进行改进，使用微信小程序实现
- 第三期主要针对系统进行优化升级

### 2.2 具体功能

#### 管理后台

- 管理员登录
- 订单明细
- 分类管理
- 菜品管理
- 套餐管理
- 员工管理

#### 用户移动端

- 地址管理
- 点餐系统
- 购物车
- 个人信息
- 提交订单

### 2.3 技术选型

#### 用户层

- H5
- vue.js
- ElementUI
- 微信小程序

## 网关层

- nginx

## 应用层

- springboot
- spring session
- swagger
- lombok

## 数据层

- MySQL
- Mybatis
- Mybatis-plus
- Redis

## 工具

- git
- maven
- junit

# 2.4 功能架构

## 移动端前台

- 手机号登录 / 微信登录
- 地址管理
- 历史订单
- 菜品规格
- 购物车
- 下单
- 菜品浏览

## 系统管理后台

- 分类管理 / 菜品管理 / 套餐管理
- 菜品口味管理
- 员工登录
- 员工退出
- 员工管理
- 订单管理

## 2.5 面向用户

**后台系统管理员**: 登录后台管理系统，拥有后台系统中的所有操作权限

**后台系统普通员工**: 登录后台管理系统，对菜品、套餐、订单等进行管理

**客户端用户**: 登录客户端应用，浏览菜品，添加购物车，设置地址，在线下单等

## 三、开发环境搭建

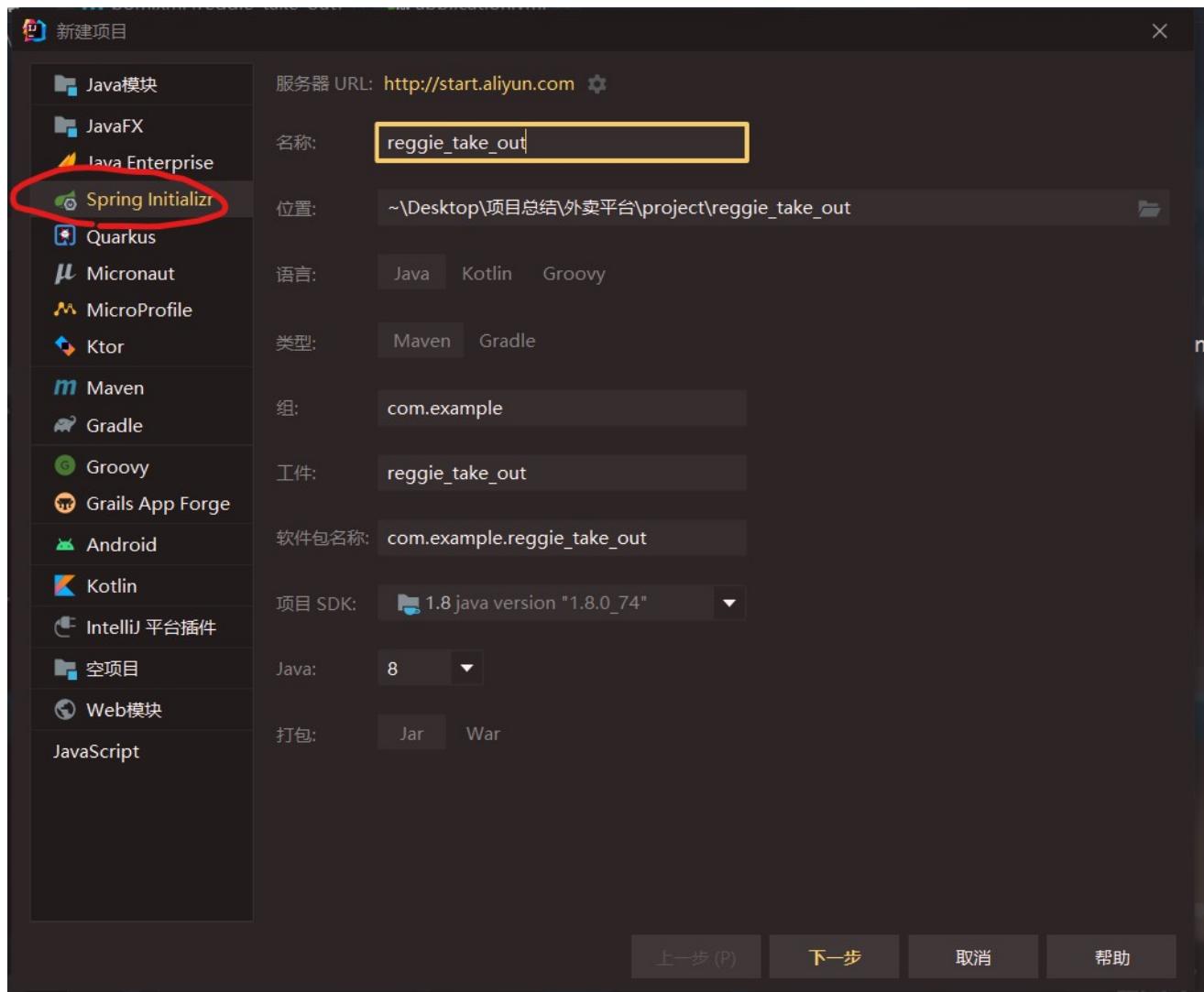
### 3.1 数据环境

#### 整体表结构

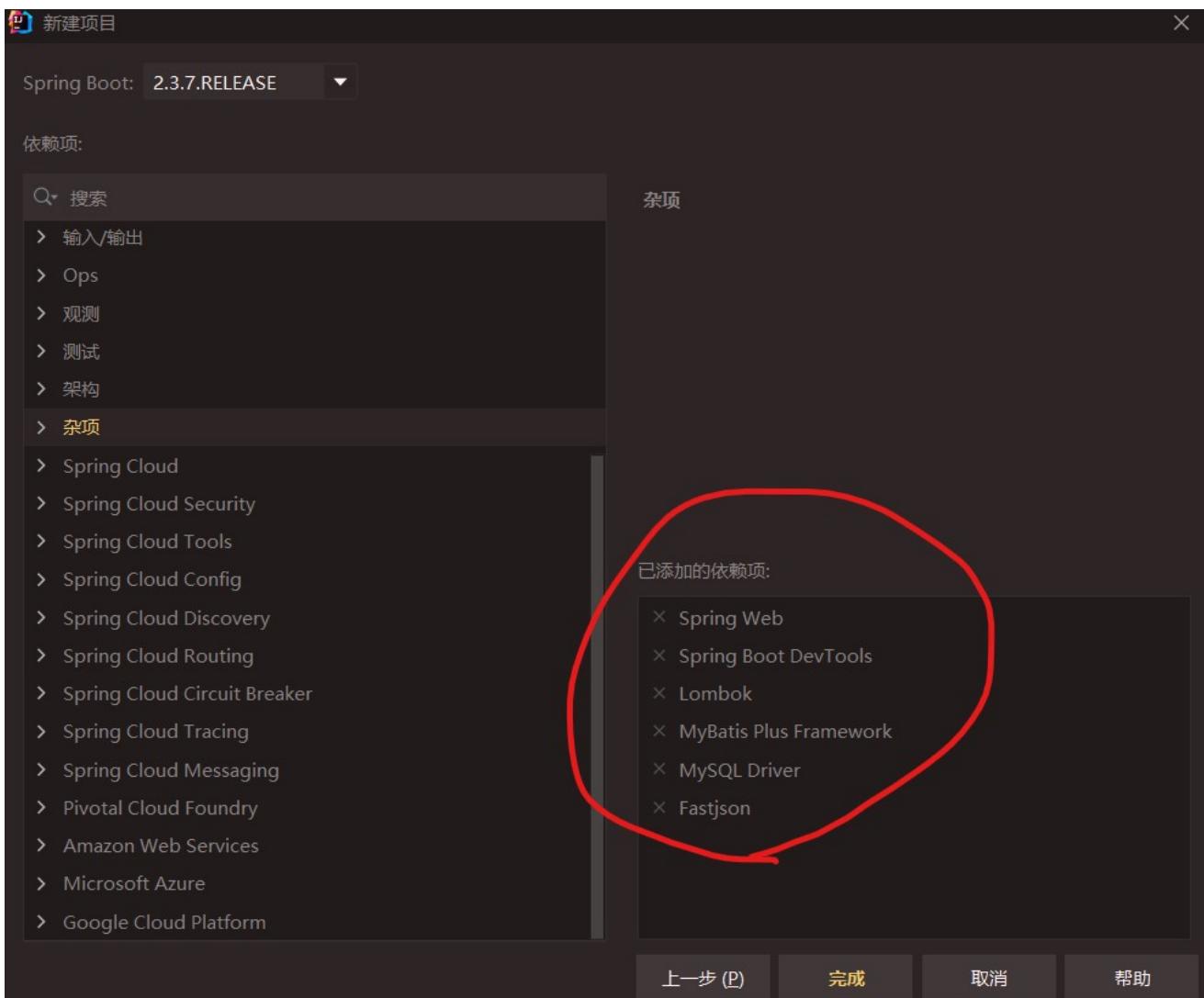
- `employee` : 员工表，存放员工的登录信息
- `category` : 菜品和套餐的分类表
- `dish` : 菜品表
- `setmeal` : 套餐表
- `setmeal_dish` : 套餐菜品关系表
- `dish_flavor` : 菜品口味关系表
- `user` : 用户表，用户登录信息
- `address_book` : 用户送餐地址
- `shopping_cart` : 购物车表
- `orders` : 订单表
- `order_detail` : 订单明细表

### 3.2 Maven搭建

IDEA新建springboot项目



选择需要导入的包



修改pom文件，导入一些新的包

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.23</version>
</dependency>

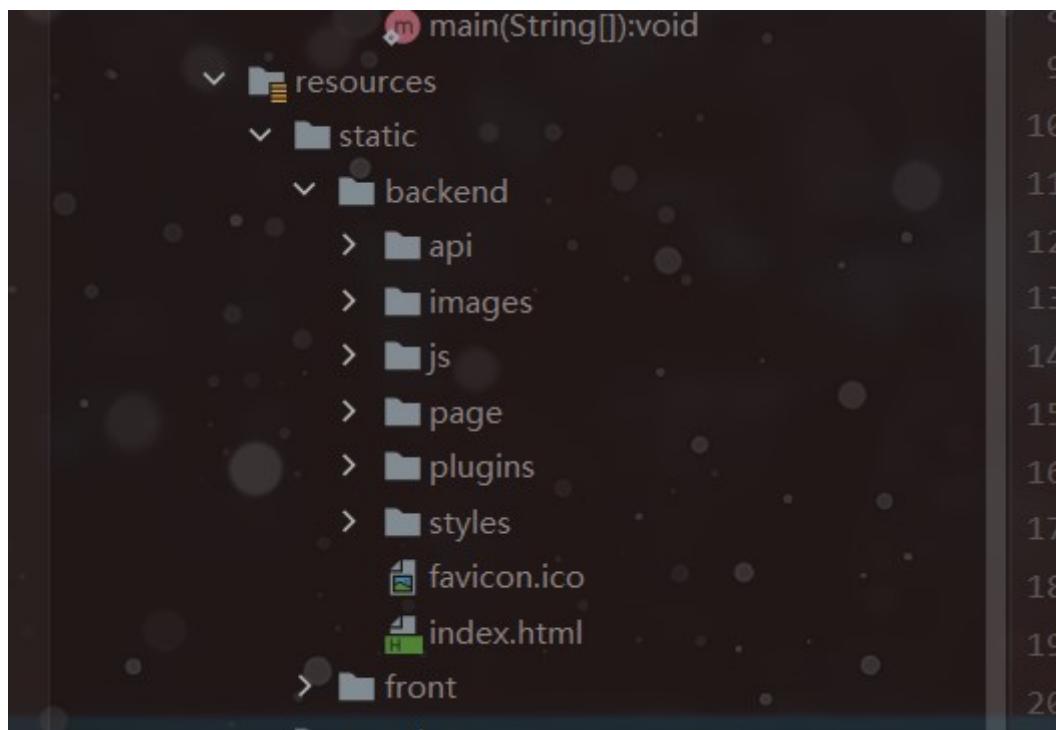
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
```

修改application.yml文件，进行基础配置

```
server:
  port: 8080
spring:
  application:
    name: reggie_take_out
  datasource:
    druid:
```

```
driver-class-name: com.mysql.cj.jdbc.Driver
url: jdbc:mysql://localhost:3306/reggie?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-
8&zeroDateTimeBehavior=convertToNull&useSSL=false&allowPublicKeyRetrieval=true
username: root
password: 123456
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
    map-underscore-to-camel-case: true
  global-config:
    db-config:
      #主键生成策略
      id-type: assign_id
```

## 导入前端资源



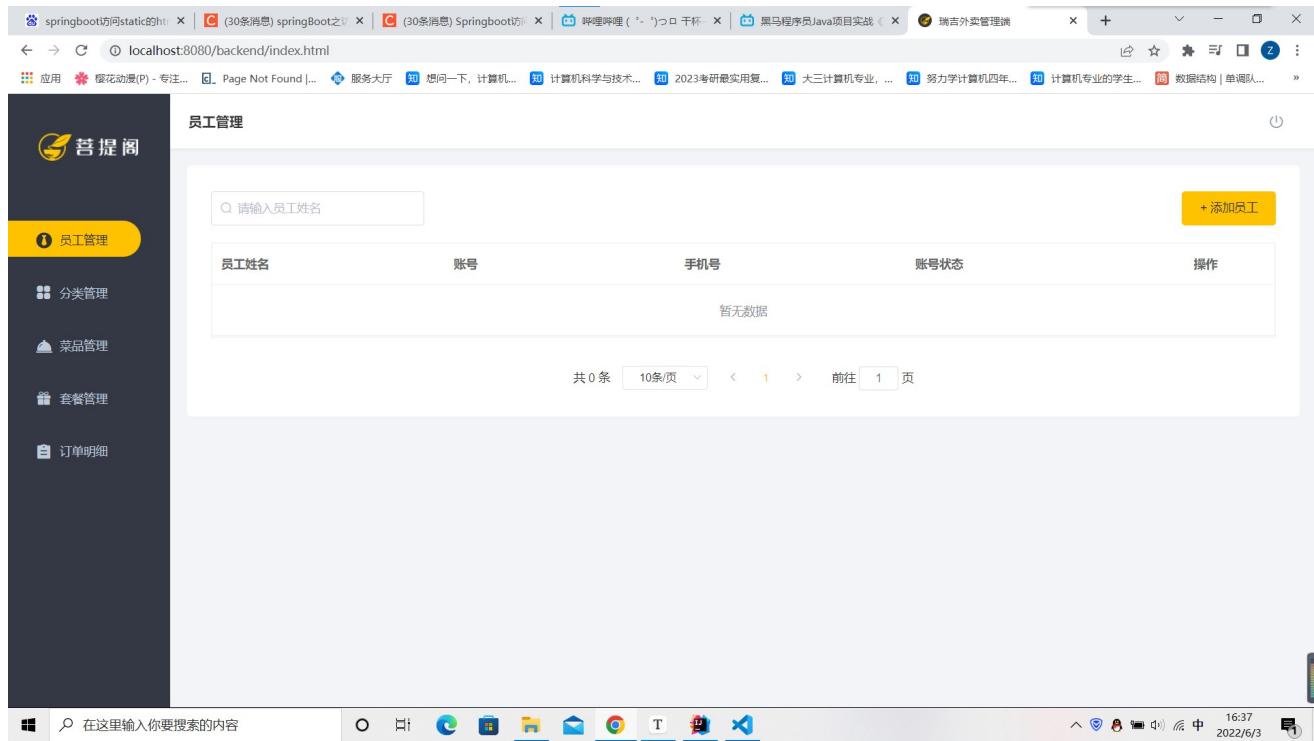
前端资源也不一定要放在static目录下，如果没有放在static或者templates目录下，则需要编写一个配置类来进行资源映射

```
@Configuration
public class WebMvcConfig extends WebMvcConfigurationSupport {

    /*
     * 配置静态资源映射
     * 如果静态资源没有放在springboot默认的目录static或者templates下，则可以通过
     * 这个配置类进行资源映射
     */
    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry)
    {
```

```
    log.info("静态资源映射");
    //addResourceHandler()表示资源访问路径/**表示某目录下的所有文件
    //addResourceLocation()表示资源在服务器中的位置
    registry.addResourceHandler("/backend/**")
        .addResourceLocations("classpath:/backend/");
}
}
```

测试：启动springboot项目，看到页面展示表示启动成功

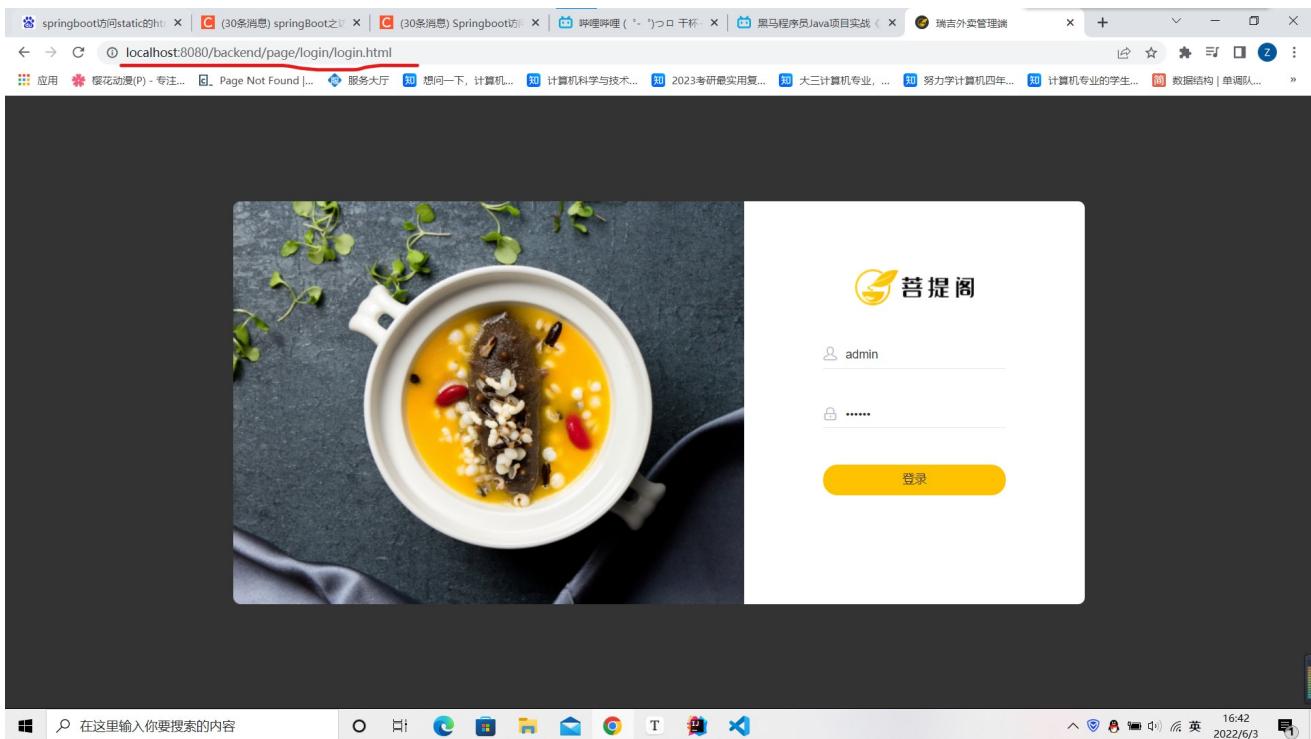


## 四、后台登录功能开发

### 4.1 需求分析

#### 登录页面展示

由于前端资源已经引入项目文件中，访问一下前端资源，查看登录页面



## 注意登录页面的路径

### 请求

查看页面之后，分析登录页面的功能：

- 点击登录按钮之后，前端会发送请求
- 后端的controller层接收请求，向下将数据传给service层，由service层对数据进行一些业务操作，操控mapper直接获取数据库中的数据
- 需要查询数据库中的employee表

### 总体流程

- 首先接收前端的请求数据，并用md5的方式对密码进行加密处理
- 根据接收到的用户名去数据库中查询用户信息
  - 如果没有查询到用户则返回登录失败信息
- 如果查询到用户，进行密码对比
  - 如果密码不一致，则返回登录失败信息
- 如果密码一致，则查看员工状态
  - 如果账号状态已经被禁用，则返回“已被禁用”的结果
- 如果账号状态为正常使用，则登录成功，返回登录成功信息，并且跳转到index.html页面（这个由前端完成）

## 4.2 代码开发

### 创建实体类

创建Employee实体类，对应数据库中的employee表，所有实体类都放在entity包下

@Data

```
@NoArgsConstructor
@AllArgsConstructor
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    /*
     * 用户id
     */
    private Long id;

    /*
     * 用户真实姓名
     */
    private String name;

    /*
     * 用户登录用户名
     */
    private String username;

    /*
     * 用户登录密码
     */
    private String password;

    /*
     * 用户手机号
     */
    private String phone;

    /*
     * 用户性别
     */
    private String sex;

    /*
     * 用户身份证号
     */
    private String idNumber;

    /*
     * 用户状态，1表示正常；0表示禁用
     */
    private Integer status;

    /*
     * 用户创建时间
     */
    private LocalDateTime createTime;

    /*
     * 用户信息更新时间
     */
    private LocalDateTime updateTime;
```

```
* */
private LocalDateTime updateTime;

@TableField(fill = FieldFill.INSERT)
private Long createUser;

@TableField(fill = FieldFill.INSERT_UPDATE)
private Long updateUser;

}
```

## 创建操作数据库的mapper接口

使用mybatis-plus创建直接操作数据库的mapper接口，并加上@Mapper注解

```
/** EmployeeMapper操作employee表
 * @author ProgZhou
 * @createTime 2022/06/03
 */
@Mapper
public interface EmployeeMapper extends BaseMapper<Employee> { }
```

## 创建业务层接口和实现类

```
/** EmployeeService接口
 * @author ProgZhou
 * @createTime 2022/06/03
 */
public interface EmployeeService extends IService<Employee> { }

/** EmployeeService的实现类
 * @author ProgZhou
 * @createTime 2022/06/03
 */
@Service
public class EmployeeServiceImpl extends ServiceImpl<EmployeeMapper, Employee>
implements EmployeeService { }
```

## 编写controller层的代码

controller层的代码不宜写的过于复杂，之后会将判断逻辑优化到service层，由service层去处理主要的业务逻辑

```
@Slf4j
@RestController
```

```
@RequestMapping("/employee")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping("/login")
    public CommonResult<Employee> login(@RequestBody Employee employee,
HttpServletRequest request) {
        String username = employee.getUsername();
        //进行md5加密
        String password =
DigestUtils.md5DigestAsHex(employee.getPassword().getBytes(StandardCharsets.UTF_8));

        log.info("get data: {}, {}", username, password);

        LambdaQueryWrapper<Employee> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(Employee::getUsername, username);

        Employee user = employeeService.getOne(wrapper);

        if(null == user) {
            return CommonResult.error("登陆失败，用户不存在");
        }

        if (!user.getPassword().equals(password)) {
            return CommonResult.error("登录失败，用户名或密码不正确");
        }
        if (user.getStatus() == 0) {
            return CommonResult.error("登录失败，账号已禁用");
        }

        request.getSession().setAttribute("employee", user.getId());
        return CommonResult.success(user);
    }
}
```

## 4.3 功能测试

## 4.4 退出登录功能

### 需求

当用户点击页面中的退出按钮之后，会发出一个请求

后端需要接收这个请求，做用户登出处理

- 清理Session中的用户id
- 返回结果

## 代码编写

```
@PostMapping("/logout")
public CommonResult<String> logout(HttpServletRequest request) {

    //清理Session中保存的id
    log.info("session: {}", request.getSession());
    request.getSession().removeAttribute("employee");

    return CommonResult.success("成功退出");
}
```

# 五、管理业务开发

## 5.1 登录功能完善

### 问题分析

在之前的代码中，如果用户没有登录，也能够通过url直接输入index.html地址来访问首页。这就存在安全问题，最终的效果应该是，只有登录成功之后才可以访问系统中的其他页面，如果系统察觉到用户没有登录，则需要跳转到登录页面。

具体做法：可以配置拦截器或者过滤器

### 代码实现

流程：

- 获取本次请求的URI
- 判断本次请求是否需要处理（登录请求不需要处理）
- 如果不需要处理，则直接放行
- 判断登录状态，如果已登录，则直接放行
- 如果未登录则返回未登录状态

这个代码实现可以使用原生Servlet的Filter，也可以使用SpringMVC提供的Interceptor实现。

Filter实现：如果使用Filter实现，需要在启动类上加上@ServletComponentScan注解。

```
@WebFilter(filterName = "loginFilter", urlPatterns = "/*")
@Slf4j
public class LoginFilter implements Filter {

    //字符串匹配，支持正则表达式
    public static AntPathMatcher pathMatcher = new AntPathMatcher();

    @Override
```

```
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException
{
    HttpServletRequest request = (HttpServletRequest) servletRequest;
    HttpServletResponse response = (HttpServletResponse) servletResponse;
    log.info("拦截到请求: {}", request.getRequestURL());

    String uri = request.getRequestURI();

    /*
     * 登录登出方法不需要拦截
     * 静态资源不需要拦截
     */
    String[] urls = {
        "/employee/login",
        "/employee/logout",
        "/backend/**",
        "/front/**"
    };

    //判断请求是否需要处理
    if (check(urls, uri)) {
        filterChain.doFilter(request, response);
        return;
    }

    //判断登录状态
    if (request.getSession().getAttribute("employee") != null) {
        filterChain.doFilter(request, response);
        return;
    }

    //返回未登录信息

    response.getWriter().write(JSON.toJSONString(CommonResult.error("NOTLOGIN")));
;

}

//检查本次请求是否能够放行
public boolean check(String[] urls, String uri) {
    for (String url : urls) {
        boolean match = pathMatcher.match(url, uri);
        if(match) {
            return true;
        }
    }
    return false;
}
}
```

Interceptor实现：Interceptor实现需要增加一个配置类

```
@Slf4j
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        //执行登录检查
        HttpSession session = request.getSession();

        log.info("拦截到请求: {}", request.getRequestURL());

        Object employee = session.getAttribute("employee");

        //如果已登录，则放行
        if(employee != null) {
            log.info("用户已登陆");
            return true;
        }

        //如果未登录则拦截
        log.info("用户未登录" );
        response.getWriter().write(JSON.toJSONString(CommonResult.error("NOTLOGIN")));
    }

    return false;
}
}

@Configuration
public class LoginConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())
            .addPathPatterns("/**")
            //放行请求
            .excludePathPatterns("/employee/login", "/employee/logout",
                "/backend/**", "/front/**");
    }
}
```

## 5.2 新增员工功能

### 需求分析

用户登录后跳转到index页面的员工管理部分，在右上角有一个添加员工按钮，点击添加员工按钮，跳转到：

The screenshot shows a 'Add Employee' form. At the top left is a 'Return' button and a 'Add Employee' title. The form contains four input fields with red asterisks: 'Account' (请输入账号), 'Employee Name' (请输入员工姓名), 'Mobile Number' (请输入手机号), and 'ID Card Number' (请输入身份证号). Below these is a gender selection section with 'Male' (男) and 'Female' (女) radio buttons, where 'Male' is selected. At the bottom are three buttons: 'Cancel' (取消), 'Save' (保存), and 'Save and Continue Add' (保存并继续添加).

点击保存按钮之后的执行流程：

- 页面发送ajax请求，将新增员工的信息以json的形式提交到服务器
- 服务端Controller接收页面提交的数据并调用service接口将数据进行保存
- service调用mapper操作数据库，将员工信息保存

### 数据模型

在数据库中的employee表中，对用户名有唯一性限制，所以，不能添加重复的用户名，出现重复的用户名会报错

### 代码实现

当前暂时将这些简单的业务逻辑写在controller层，到最后优化时，优化到service层

```
/*
 * 新增员工
 */
@PostMapping
public CommonResult<String> addEmp(@RequestBody Employee employee,
HttpServletRequest request) {
    log.info("新增员工: {}", employee);

    // 设置初始密码，统一为123456

    employee.setPassword(DigestUtils.md5DigestAsHex("123456".getBytes(StandardCharsets.UTF_8)));
}
```

```

    //设置员工账号的创建时间和更新时间
    employee.setCreateTime(LocalDateTime.now());
    employee.setUpdateTime(LocalDateTime.now());

    Long userId = (Long) request.getSession().getAttribute("employee");

    //设置创建员工账号的userId, 一般为管理员id
    employee.setCreateUser(userId);
    employee.setUpdateUser(userId);

    employeeService.save(employee);
    return CommonResult.success("添加成功");
}

```

单写上面的代码会出现异常，如果添加了重复的员工，那么代码就会报错，对于报错的处理有两种方式：

- 在代码块中直接使用try - catch捕获异常并处理
- 添加一个全局的异常处理器

添加全局处理器的做法：

```

/** 全局异常处理
 * @author ProgZhou
 * @createTime 2022/06/04
 */
@ControllerAdvice(annotations = {RestController.class, Controller.class}) //标识需要捕获哪些类的异常信息
@ResponseBody //上面两个注解也可以合并为@RestControllerAdvice注解
@Slf4j
public class GlobalExceptionHandler {

    /**
     * 进行异常处理方法
     * @return 返回异常信息
     */
    @ExceptionHandler(SQLIntegrityConstraintViolationException.class)
    public CommonResult<String>
    handler(SQLIntegrityConstraintViolationException ex) {
        log.error(ex.getMessage());
        return CommonResult.error("请求失败");
    }
}

```

## 5.3 员工信息分页查询

### 需求分析

查看员工管理页面：

分页显示员工的信息，需求有：

- 每页显示的员工信息可以根据下拉框中的选择动态调整
- 可以通过"<" 或者">"进行前后页面切换，也可以直接输入"前往x页"直接跳转到x页
- 在左上方出现的输入框中，输入员工姓名，可以直接显示员工的信息

具体流程：

- 当登录页面之后，前端会自动使用ajax发送请求，查询分页数据，将参数（page, pageSize, username）提交到服务器
- 服务端Controller接收页面提交的数据并调用Service查询数据
- Service调用Mapper操作数据库，查询分页数据
- Controller将查询到的分页数据响应给页面

### 代码开发

借助mybatis plus中的分页插件实现分页查询，首先需要在配置类中配置mybatis plus的分页插件

```
/** mybatis plus配置类
 * @author ProgZhou
 * @createTime 2022/06/04
 */
@Configuration
public class MybatisPlusConfig {
    //配置mybatis plus分页插件
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor mybatisPlusInterceptor = new
MybatisPlusInterceptor();
        mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
        return mybatisPlusInterceptor;
    }
}
```

## 在controller层中添加分页查询的方法

```
@GetMapping("/page")
public CommonResult<Page<Employee>> getPage(int page, int pageSize, String
name) {
    log.info("page: {}, pageSize: {}, name: {}", page, pageSize, name);

    //Mybatis plus中的分页插件
    Page<Employee> pageInfo = new Page<>(page, pageSize);

    LambdaQueryWrapper<Employee> wrapper = new LambdaQueryWrapper<>();
    /*
     * 如果name不为空，才会将like name条件添加到sql语句中
     */
    wrapper.like(!StringUtils.isEmpty(name), Employee::getName, "%" + name +
    "%");
    wrapper.orderByDesc(Employee::getUpdateTime);

    //执行查询，使用page方法进行查询时，会自动分页，统计总数据
    employeeService.page(pageInfo, wrapper);

    return CommonResult.success(pageInfo);
}
```

## 5.4 启用 / 禁用员工账号

### 需求分析

管理员登录之后，可以对某个员工的账号进行禁用或者启用操作，对于账号被禁用的员工，则不能登录系统

如果员工的账号状态本身就是禁用，则操作中会显示“启用”

The screenshot shows a web-based employee management system. At the top, there is a search bar labeled "请输入员工姓名" and a yellow button labeled "+添加员工". Below this is a table with the following columns: 员工姓名 (Employee Name), 账号 (Account), 手机号 (Mobile Number), 账号状态 (Account Status), and 操作 (Operations). The table contains the following data:

员工姓名	账号	手机号	账号状态	操作
小凡凡	BBQ	17426035112	正常	<a href="#">编辑</a> <span style="color: red;">禁用</span>
张三	zzzq	13124578911	正常	<a href="#">编辑</a> <span style="color: red;">禁用</span>
隔壁老王	13857	18856316787	正常	<a href="#">编辑</a> <span style="color: red;">禁用</span>
管理员	admin	13812312312	正常	<a href="#">编辑</a> <span style="color: green;">启用</span>

At the bottom of the table, there is a pagination control with the text "共 4 条" and "10条/页" followed by a dropdown menu, and navigation buttons < >.

具体流程：

- 当点击"禁用"或者"启用"按钮后，前端会发送ajax请求，并将携带的参数(userId, status)提交到服务器端
- 服务端接收到页面发送的数据之后，调用service更新数据

## 代码开发

启用 / 禁用员工账号本质上是一个更新操作，也就是对status字段进行修改

```
@PutMapping
public CommonResult<String> forbiddenEmp(@RequestBody Employee employee,
HttpServletResponse request) {
    log.info("empId: {}, statue: {}", employee.getId(), employee.getStatus());

    Long id = (Long) request.getSession().getAttribute("employee");
    employee.setUpdateTime(LocalDateTime.now());
    employee.setUpdateUser(id);

    employeeService.updateById(employee);

    return CommonResult.success("信息修改成功");
}
```

这里调试的时候会出现一定的问题，查看控制台窗口，会发现发送过来的empId并不是数据库中的id，因为前端js在处理数据的时候，会丢失长整型数据的精度，所以，需要对长整型的数据进行数据类型转换，转换为字符串类型，具体做法就是在配置类中添加类型转换器

```
@Configuration
public class ObjectMapperConfig implements WebMvcConfigurer {

    /*
     * 配置消息转换器，主要是将Long类型的数据转换为字符串类型
     */
    @Override
    public void extendMessageConverters(List<HttpMessageConverter<?>>
converters) {
        MappingJackson2HttpMessageConverter converter = new
        MappingJackson2HttpMessageConverter();
        //设置对象转换器
        converter.setObjectMapper(new JacksonObjectMapper());
        //将新建的转换器追加到mvc框架的转换器中
        converters.add(0, converter);
    }
}
```

## 5.5 编辑员工信息

### 需求分析

点击每个员工信息之后的编辑按钮，进入修改员工的界面

The screenshot shows a web-based 'Edit Employee' form. At the top left is a '返回' (Back) button and a '修改员工' (Edit Employee) title. At the top right is a '管理员' (Administrator) user icon. The form contains the following fields:

- \* 账号:
- \* 员工姓名:
- \* 手机号:
- 性别:  男  女
- \* 身份证号:

At the bottom are two buttons: '取消' (Cancel) and '保存' (Save), with '保存' being highlighted by a yellow background.

在基本信息栏需要回显员工的基本信息，点击保存按钮修改信息

基本流程：

- 在修改员工的界面发送ajax请求，同时传送员工id给服务端
- 服务端接收到用户请求之后，根据用户id查询员工信息，将员工信息以json的形式发送给客户端
- 前端接收到数据，并显示基本数据
- 当点击保存按钮，再次发送ajax请求，将页面中的新数据发送给后端
- 后端接收到数据之后，将数据更新至数据库，并给前端响应

### 代码开发

关于更新的方法可以复用之前启用/禁用员工账号的方法，因为都是更新操作，并且传输的数据也都是Employee对象

当跳转到add.html页面时，会发送一个get请求，查询待编辑员工的信息，并显示到页面上，所以，只需要添加这个方法即可

```

/*
 * 根据员工的id查询员工信息，并返回
 */
@GetMapping("/{id}")
public CommonResult<Employee> getEmpById(@PathVariable Long id) {
    log.info("id: {}", id);
    Employee employee = employeeService.getById(id);
    if (employee != null) {
        return CommonResult.success(employee);
    } else {
        return CommonResult.error("没有查询到员工信息");
    }
}

```

## 六、分类管理功能

### 6.1 公共字段填充

#### 问题分析

在前面添加、修改员工信息时，会发现每一次操作都需要对员工表的update\_time，update\_user字段进行修改

而这些字段，包括create\_time，create\_user在大多表中都存在，也就是所谓的公共字段

那么就可以对这些公共字段进行统一处理，简化开发

#### 代码实现

mybatis plus公共字段自动填充，也就是在插入或者更新的时候为指定字段赋予指定的值，使用它的好处就是可以统一对这些字段进行处理，避免了重复代码

实现步骤：

- 在实体类的属性上加入@TableField注解，指定自动填充的策略
- 按照框架要求编写元数据对象处理器，在此类中统一公共字段赋值，类需要实现MetaObjectHandler接口

```

/** 自定义元数据对象处理器，配置mybatis plus字段的自动填充策略
 * @author ProgZhou
 * @createTime 2022/06/05
 */
@Component
@Slf4j
public class MyMetaObjectHandler implements MetaObjectHandler {

    //在数据插入的时候自动填充字段的值
    @Override

```

```

public void insertFill(MetaObject metaObject) {
    log.info("insert 插入自动填充...");
    metaObject.setValue("createTime", LocalDateTime.now());
    metaObject.setValue("updateTime", LocalDateTime.now());
    metaObject.setValue("createUser", 1L);
    metaObject.setValue("updateUser", 1L);

}

//在数据更新的时候自动填充字段的值
@Override
public void updateFill(MetaObject metaObject) {
    log.info("update 更新自动填充...");
    metaObject.setValue("updateTime", LocalDateTime.now());
    metaObject.setValue("updateUser", 1L);
}
}

```

改进：

- 动态获取当前用户id，由于在MetaObjectHandler中无法获取request实例，也就无法获得session，所以需要通过其他方式来获取当前登录用户的id
- 可以使用ThreadLocal，jdk中提供的一个类

客户端发送的每次http请求，对应的在服务端都会分配一个新的线程来处理，在处理过程中涉及到：

- Filter的doFilter方法
- Controller中的方法
- MetaObjectHandler中的updateFill方法

都属于同一个线程去执行的，所以可以使用ThreadLocal来保存线程中的局部变量

### ThreadLocal简介：

ThreadLocal是Thread的局部变量，当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以，每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本；ThreadLocal为每个线程提供一份单独存储空间，具有线程隔离效果，只有在线程内才能获取对应的值，线程外不能访问

具体实现步骤：

- 编写BaseContext工具类，基于ThreadLocal封装的工具类
- 在LoginCheckFilter的doFilter方法中调用BaseContext来设置当前登录用户的id
- 在MyMetaObjectHandler的方法中调用BaseContext获取登录用户的id

```

/** 基于ThreadLocal的封装工具类，用于保存和获取当前登录用户id
 * @author ProgZhou
 * @createTime 2022/06/05

```

```

/*
public abstract class BaseContext {
    private static ThreadLocal<Long> threadLocal = new ThreadLocal<>();

    public static void setCurrentUserId(Long id) {
        threadLocal.set(id);
    }

    public static Long getCurrentUserId() {
        return threadLocal.get();
    }
}

//在LoginFilter中添加代码:
//判断登录状态
if (request.getSession().getAttribute("employee") != null) {
    Long id = (Long) request.getSession().getAttribute("employee");
    BaseContext.setCurrentUserId(id);
    log.info("当前登录用户id: {}", id);
    filterChain.doFilter(request, response);
    return;
}

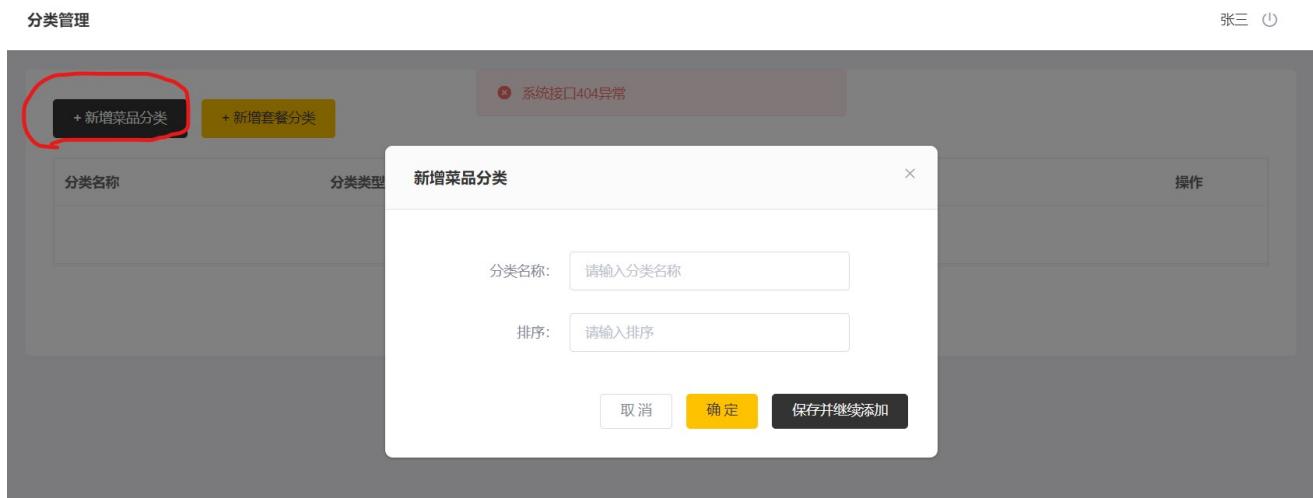
```

## 6.2 新增分类

### 需求分析

后台系统中可以管理分类信息，分类包括两种类型，分别是菜品分类和套餐分类；当我们在后台系统中添加菜品时需要选择一个菜品分类；当在后台系统中添加一个套餐时需要选择一个套餐分类，在移动端也会按照菜品分类和套餐分类来展示对应的菜品和套餐

新增页面：



菜品分类的信息都保存在数据库中的category表中，其中分类名称需要唯一

### 代码开发

- 编写实体类Category对应category表

- CategoryMapper接口
- CategoryService接口 / CategoryServiceImpl实现类
- CategoryController控制层

整体流程：

- 当信息输入完毕，点击确定按钮，分类管理页面发送ajax请求，将填入的信息以json的格式传送给后端
- controller接收信息并处理，调用service层再由service调用mapper操作数据库将信息持久化至数据库中

Category实体类

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;
    /*
     * 菜品分类id
     */
    private Long id;

    /*
     * 类型 1表示菜品分类 2表示套餐分类
     */
    private Integer type;

    /*
     * 分类名称，唯一
     */
    private String name;

    /*
     * 分类显示顺序
     */
    private Integer sort;

    /*
     * 分类创建时间
     */
    @TableField(fill = FieldFill.INSERT)
    private LocalDateTime createTime;

    /*
     * 分类更新时间
     */
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private LocalDateTime updateTime;
}

```

```
/*
 * 创建分类的用户
 */
@TableField(fill = FieldFill.INSERT)
private Long createUser;

/*
 * 更新分类的用户
 */
@TableField(fill = FieldFill.INSERT_UPDATE)
private Long updateUser;
}
```

## CategoryController控制层

```
@RestController
@Slf4j
@RequestMapping("/category")
public class CategoryController {

    @Autowired
    private CategoryService categoryService;
    /*
     * 新增分类
     */
    @PostMapping
    public CommonResult<String> addCategory(@RequestBody Category category) {
        log.info("接收到分类信息: {}", category);
        categoryService.save(category);
        return CommonResult.success("添加成功");
    }

}
```

## 6.3 分类信息分页查询

与之前的员工信息分页查询类似，直接上代码

```
@GetMapping("/page")
public CommonResult<Page<Category>> getPage(int page, int pageSize) {
    Page<Category> pageInfo = new Page<>(page, pageSize);

    LambdaQueryWrapper<Category> wrapper = new LambdaQueryWrapper<>();
    wrapper.orderByAsc(Category::getSort);

    categoryService.page(pageInfo, wrapper);
    return CommonResult.success(pageInfo);
}
```

## 6.4 删除分类

### 需求分析

再分类管理页面可以对某个分类进行删除操作

当分类关联了菜品或者套餐的时候，分类不允许删除

### 代码开发

```
@DeleteMapping
public CommonResult<String> delete(@RequestParam("ids") Long id) {
    log.info("待分类的id为: {}", id);
    //如果没有特定的业务需求，直接删除即可
    categoryService.removeById(id);
    return CommonResult.success("分类删除成功");
}
```

### 功能完善

由于需求中有 **当分类关联了菜品或者套餐的时候，分类不允许删除** 这条规定，那么就不能直接删除分类，需要进行一定的判断

首先需要菜品和套餐的实体类、Mapper接口、Service接口，用于查询当前的分类是否关联了菜品或者套餐

紧接着，在CategoryService中增加一个方法，代替原始的直接删除的方法

```
@Autowired
private DishMapper dishMapper;

@Autowired
private SetmealMapper setmealMapper;

@Autowired
private CategoryMapper categoryMapper;
/*
 * 根据id删除分类
*/
```

```

* */
@Override
public void remove(Long id) {
    //查询当前分类是否关联了菜品
    LambdaQueryWrapper<Dish> dishWrapper = new LambdaQueryWrapper<>();
    dishWrapper.eq(Dish::getCategoryId, id);
    Integer dishCount = dishMapper.selectCount(dishWrapper);
    if(dishCount > 0) {
        //说明已经当前分类已经关联了菜品，抛出异常
        log.info("有{}个菜品关联了此分类", dishCount);
        throw new ServiceException("当前分类向关联了菜品，不能删除");
    }

    //查询当前分类是否关联了套餐
    LambdaQueryWrapper<Setmeal> setMealWrapper = new LambdaQueryWrapper<>();
    setMealWrapper.eq(Setmeal::getCategoryId, id);
    Integer setMealCount = setmealMapper.selectCount(setMealWrapper);
    if(setMealCount > 0) {
        //说明已经关联了套餐，抛出异常
        log.info("有{}个套餐关联了此分类", setMealCount);
        throw new ServiceException("当前分类向关联了套餐，不能删除");
    }

    //如果都没有，则正常删除
    categoryMapper.deleteById(id);
}

```

还需要一个自定义的异常类

```

/** 自定义业务异常类
 * @author ProgZhou
 * @createTime 2022/06/05
 */
public class ServiceException extends RuntimeException {

    public ServiceException(String message) {
        super(message);
    }
}

```

并且在之前处理全局异常的handler中增加捕获自定义异常类的方法

```

@ExceptionHandler(ServiceException.class)
public CommonResult<String> serviceHandler(ServiceException sex) {
    log.error(sex.getMessage());
    return CommonResult.error(sex.getMessage());
}

```

## 6.5 修改分类

### 需求分析

当点击每行信息末尾的"修改"按钮之后，弹出编辑框



信息的回显由前端完成，只能修改分类的名称和排序信息，注意，分类的名称不能重复

### 代码开发

代码比较简单：

```
/*
 * 修改分类
 */
@PutMapping
public CommonResult<String> updateCategory(@RequestBody Category category) {
    log.info("修改分类信息: {}", category);
    categoryService.updateById(category);
    return CommonResult.success("修改成功");
}
```

## 七、菜品管理业务

### 7.1 文件的上传和下载

#### 文件上传

文件上传对前端页面的要求：

```
<form method="post" action="address" enctype="multipart/form-data">
    <input name="xxx" type="file"/>
    <input type="submit" value="提交"/>
</form>
```

服务端要接收前端上传的文件，通常都使用Apache提供的两个组件：

- commons - fileupload
- commons - io

spring框架在spring-web包中对文件的上传和下载做了封装，操作更加简便，只要在controller的方法中声明一个MultipartFile类型的参数即可接收到上传的文件

## 上传代码实现

前端选择一个待上传的文件，后端通过MultipartFile类型的参数接收，并调用transferTo方法存储到指定位置

```
/*
 * 文件上传处理
 */
@PostMapping("/upload")
public CommonResult<String> upload(MultipartFile file) {
    log.info("文件上传处理: {}", file.toString());

    // 使用UUID生成在服务器中的文件名
    String filename = UUID.randomUUID().toString();
    // 原始文件名
    String originalFilename = file.getOriginalFilename();
    // 截取文件后缀
    String suffix =
        originalFilename.substring(originalFilename.lastIndexOf("."));
    // 组合成新的存储在服务器中的文件名
    String newFilename = filename + suffix;
    // 指定存储路径，basePath可以在application.yml中配置
    File dir = new File(basePath);
    if(!dir.exists()) {
        boolean mkdirs = dir.mkdirs();
    }
    log.info("新文件名: {}", newFilename);
    try {
        file.transferTo(new File(basePath + File.separator + newFilename));
    } catch (IOException e) {
        e.printStackTrace();
    }
    return CommonResult.success(newFilename);
}
```

## 文件下载

通常浏览器进行文件下载，通常有两种表现显示：

- 以附件的形式下载，弹出对话框，将文件保存到指定磁盘目录
- 直接在浏览器中打开

通过浏览器进行文件下载，本质上就是服务端将文件以流的形式写回浏览器的过程

前端界面可使用标签展示下载的图片

```

```

## 下载代码实现

当上传的文件存储到服务器之后，前端会自动发送一个请求，请求获取文件的内容展示到标签内，在后端需要编写一个方法去处理这个请求

```
/*
 * 文件下载处理
 */
@GetMapping("/download")
public void download(@RequestParam("name") String name, HttpServletResponse
response) {
    FileInputStream inputStream = null;
    ServletOutputStream outputStream = null;
    try {
        //通过输入流读取文件内容
        inputStream = new FileInputStream(basePath + File.separator + name);
        //通过输出流将文件内容写回浏览器
        outputStream = response.getOutputStream();
        response.setContentType("image/jpeg");
        //通过byte数组存储
        byte[] bytes = new byte[1024];
        int len = 0;
        while ((len = inputStream.read(bytes)) != -1) {
            outputStream.write(bytes, 0, len);
            outputStream.flush();
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if(inputStream != null) {
            try {
                inputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(outputStream != null) {
            try {
                outputStream.close();
            } catch (IOException e) {

```

```
        e.printStackTrace();
    }
}
}
}
```

## 7.2 新增菜品

### 需求分析

后台系统中可以管理菜品的相关信息，通过新增功能来添加一个新的菜品

The screenshot shows a user interface for adding a new dish. At the top, there are three required fields: 'Dish Name' (请输入菜品名称), 'Dish Category' (请选择菜品分类, highlighted with a red border), and 'Dish Price' (请输入菜品价格). Below these is a section for 'Flavor Configuration' (口味做法配置) with a '+ Add Flavor' button. There is also a dashed box for 'Dish Image' (菜品图片) with a 'Upload Image' (上传图片) button. A text area for 'Dish Description' (菜品描述, up to 200 characters) is shown below. At the bottom right are three buttons: 'Cancel' (取消), 'Save' (保存), and a dark button labeled 'Save and Continue Adding Dishes' (保存并继续添加菜品).

将输入的基本信息插入到数据表中，分别涉及到：

- dish 菜品表
- dish\_flavor 菜品口味表

### 代码开发

准备工作：

- 创建DishFlavor实体类
- DishFlavorMapper接口操作数据库
- DishFlavorService接口处理业务逻辑
- DishFlavorServiceImpl业务层实现类
- DishController控制层

其他都差不多，贴一下实体的代码：

```
/** 菜品口味实体类
```

```
* @author ProgZhou
* @createTime 2022/06/06
*/
@Data
@NoArgsConstructor
@AllArgsConstructor
public class DishFlavor implements Serializable {
    private static final long serialVersionUID = 1L;

    /*
     * 菜品口味id
     */
    private Long id;

    /*
     * 对应菜品id
     */
    private Long dishId;

    /*
     * 口味名称
     */
    private String name;

    /*
     * 口味对应的值
     */
    private String value;

    /*
     * 口味创建时间
     */
    @TableField(fill = FieldFill.INSERT)
    private LocalDateTime createTime;

    /*
     * 菜品口味更新时间
     */
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private LocalDateTime updateTime;

    /*
     * 创建菜品口味的用户
     */
    @TableField(fill = FieldFill.INSERT)
    private Long createUser;

    /*
     * 更新菜品口味的用户
     */
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Long updateUser;
```

```
/*
 * 菜品口味是否删除 0表示未删除 1表示已删除
 */
@TableField("is_deleted")
private Integer deleted;

}
```

具体流程：

- 在点击"新增菜品"按钮时，页面会自动发送一个ajax请求，请求获取菜品分类的数据，显示在"分类"这个下拉框中  
在CategoryController中添加 `getList()` 方法，处理请求

```
/*
 * 根据查询分类数据
 */
@GetMapping("/list")
public CommonResult<List<Category>> getList(Category category) {
    LambdaQueryWrapper<Category> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(category.getType() != null, Category::getType,
    category.getType());

    wrapper.orderByAsc(Category::getSort).orderByDesc(Category::getUpdateTime)
    ;
    List<Category> categoryList = categoryService.list(wrapper);
    return CommonResult.success(categoryList);
}
```

- 页面发送请求进行图片的上传和下载并且回显至浏览器
- 当数据输入完毕，点击"保存"按钮，发送ajax请求，将菜品数据持久化至数据库中

不能仅仅调用DishService的save方法，需要修改两张表的数据，在DishService中添加一个方法，并且从前端接收到的数据也不能单单封装成一个Dish对象，因为：

## ▼ Request Payload [view source](#)

```
▼ {name: "hjhh", price: 5500, code: "", image: "90c708b5-804e-4661-a5c1-718d4ebf7f5a.  
    categoryId: "1397844263642378242"  
    code: ""  
    description: "rrr"  
    ▶ flavors: [{name: "甜味", value: "[\"无糖\", \"少糖\", \"多糖\"]", showOption: false}]  
        ▶ 0: {name: "甜味", value: "[\"无糖\", \"少糖\", \"多糖\"]", showOption: false}  
        image: "90c708b5-804e-4661-a5c1-718d4ebf7f5a.jpeg"  
        name: "hjhh"  
        price: 5500  
        status: 1
```

从前端传送过来的数据包含有菜品的口味信息，由于在实体类Dish中并没有这个字段，所以，需要新建一个DTO（数据传输对象）来接受这个参数

```
/** 菜品类的数据传输对象  
 * @author ProgZhou  
 * @createTime 2022/06/06  
 */  
@EqualsAndHashCode(callSuper = true)  
@Data  
public class DishDto extends Dish {  
  
    /*  
     * 口味列表  
     * */  
    private List<DishFlavor> flavors = new ArrayList<>();  
  
    /*  
     * 分类名称  
     * */  
    private String categoryName;  
  
    /*  
     *  
     * */  
    private Integer copies;  
}
```

然后再在DishService中新增一个方法，同时操作dish表和dish\_flavor表

```
/*  
 * 新增菜品，同时插入菜品对应的口味数据  
 * */  
void addWithFlavor(DishDto dishDto);  
  
//并在DishServiceImpl中实现这个方法  
/*
```

```

    * 新增菜品，同时保存菜品口味数据
    * */
    @Override
    @Transactional //由于操作了多张表，所以需要加入事务管理，要想事务管理生效，需要
在启动类上加上@EnableTransactionManagement注解
    public void addWithFlavor(DishDto dishDto) {
        //将菜品数据保存到菜品表
        dishMapper.insert(dishDto);

        //为封装在dishDto中的DishFlavor对象添加dishId
        Long dishId = dishDto.getId();

        List<DishFlavor> flavors = dishDto.getFlavors();

        flavors = flavors.stream().peek((item) ->
item.setDishId(dishId)).collect(Collectors.toList());

        //批量添加数据
        dishFlavorService.saveBatch(flavors);

    }
}

```

## 7.3 菜品信息分页查询

菜品信息的分页查询与之前的分页查询稍有不同

- 菜品分页查询需要同时显示菜品的图片
- 需要显示菜品的信息

菜品名称	图片	菜品分类	售价	售卖状态	最后操作时间	操作
暂无数据						

所以需要使用之前的DishDto来返回查询结果

```

/*
* 菜品信息分页查询
*/
@GetMapping("/page")
public CommonResult<Page<DishDto>> getPages(int page, int pageSize, String
name) {

    Page<Dish> dishPage = new Page<>(page, pageSize);
    Page<DishDto> pageInfo = new Page<>();

    LambdaQueryWrapper<Dish> wrapper = new LambdaQueryWrapper<>();

    wrapper.like(name != null, Dish::getName, name);
    wrapper.orderByDesc(Dish::getUpdateTime);
}

```

```
dishService.page(dishPage, wrapper);

//对对象拷贝，将Dish的查询结果dishPage中的属性拷贝给pageInfo，除了records属性
BeanUtils.copyProperties(dishPage, pageInfo, "records");

List<Dish> dishes = dishPage.getRecords();

//对dishPage的records属性进行处理，封装为DishDto的list
List<DishDto> list = dishes.stream().map((item) -> {
    DishDto dishDto = new DishDto();

    //将item中的属性拷贝到dishDto中
    BeanUtils.copyProperties(item, dishDto);
    //根据id查询分类对象
    Long categoryId = item.getCategoryId();
    Category category = categoryService.getById(categoryId);
    //设置分类名称
    dishDto.setCategoryName(category.getName());
    return dishDto;
}).collect(Collectors.toList());

pageInfo.setRecords(list);
return CommonResult.success(pageInfo);
}
```

## 7.4 修改菜品信息

### 需求分析

当点击每行末尾的修改按钮，跳转到修改菜品的页面，在菜品页面回显相关的菜品信息，并进行修改

[← 返回](#) | [修改菜品](#)

\* 菜品名称: 宫保鸡丁

\* 菜品价格: 25

\* 菜品分类: 川菜

**回显菜品信息**

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	
温度	热饮 X	常温 X	去冰 X	少冰 X	多冰 X	删除

**添加口味**

\* 菜品图片:



菜品描述: 无

取消
保存

具体流程：

- 前端页面发送ajax请求，根据菜品的id查询菜品的信息，用于菜品信息的回显
- 修改页面与添加页面复用同一个add.html，所以仍然会发送查询分类信息的请求，这个请求已经在之前处理过了
- 还会发送请求用于图片的回显
- 点击保存按钮，将修改后的数据持久化至数据库

## 代码实现

更新菜品信息需要同时更新菜品信息表和口味表，口味表的更新策略可以稍微粗暴一点，将原有的口味信息全部删除，再将新的口味信息添加进去

在DishService中新增方法（一般同时操作两张表时，都需要新增方法）

```

/*
 * 更新菜品信息，同时更新相应的口味信息
 */
void updateDishWithFlavor(DishDto dishDto);

//在DishServiceImp1中实现此方法
@Override
    public void updateDishWithFlavor(DishDto dishDto) {
        //更新dish表
    }

```

```

dishMapper.updateById(dishDto);
//更新口味表
//1. 清理当前菜品对应的口味数据
LambdaQueryWrapper<DishFlavor> wrapper = new LambdaQueryWrapper<>();
//将口味表中dishId所对应的口味全部删除
wrapper.eq(DishFlavor::getDishId, dishDto.getId());
dishFlavorMapper.delete(wrapper);
//2. 添加当前提交过来的口味数据, flavors中的对象只封装口味名和口味值的信息, 并没有封装dishId的信息, 所以需要进行一定的处理
List<DishFlavor> flavors = dishDto.getFlavors();
flavors = flavors.stream().peek((item) ->
item.setDishId(dishDto.getId())).collect(Collectors.toList());
dishFlavorService.saveBatch(flavors);
}

```

## 八、套餐管理业务开发

### 8.1 新增套餐

#### 需求分析

套餐就是菜品的集合，后台系统可以管理套餐的信息，通过新增套餐功能来添加一个新的套餐，在添加套餐时需要选择当前套餐所属的套餐分类和包含的菜品，并且需要上传套餐的图片

The screenshot shows a user interface for adding a new setmeal. It includes fields for setmeal name, price, category selection, ingredient addition, image upload, description, and a footer with cancel, save, and continue buttons.

- 套餐名称:** Input field with placeholder "请填写套餐名称".
- 套餐分类:** Input field with placeholder "请选择套餐分类".
- 套餐价格:** Input field with placeholder "请设置套餐价格".
- 套餐菜品:** A button labeled "+ 添加菜品" and a note "添加相应的菜品, 需要查询菜品表". Below this is a dashed box for image upload with the placeholder "上传图片".
- 套餐图片:** A note "套餐图片" next to the image upload area.
- 套餐描述:** Input field with placeholder "套餐描述, 最长200字".
- Category Selection:** A dropdown menu with three options: 商务套餐 (Business Meal), 儿童套餐 (Children's Meal), and 超值套餐A (Super Value Meal A). The first option is highlighted in red, indicating it has been selected or is the default.
- Footer:** Buttons for "取消" (Cancel), "保存" (Save), and "保存并继续添加套餐" (Save and Continue Adding Setmeal).

新增套餐就是将套餐的信息录入到数据库中的setmeal表和setmeal\_dish表（菜品关系表）

## 代码开发

首先处理查询菜品表的请求：

在DishController中增加方法

```
/*
 * 根据分类id查询菜品
 */
@GetMapping("/list")
public CommonResult<List<Dish>> getDishByCategoryId(Dish dish) {
    log.info("得到categoryId: {}", dish.getCategoryId());

    LambdaQueryWrapper<Dish> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(dish.getCategoryId() != null, Dish::getCategoryId,
    dish.getCategoryId());
    //只查询正在售卖的菜品
    wrapper.eq(Dish::getStatus, 1);

    wrapper.orderByAsc(Dish::getSort).orderByDesc(Dish::getUpdateTime);
    List<Dish> dishList = dishService.list(wrapper);

    return CommonResult.success(dishList);
}
```

然后处理新增菜品的请求

- 从前端传来的json数据中，同时包含了套餐中含有的菜品，这在Setmeal实体类中没有这个属性，所以需要新建一个数据传输对象SetmealDto
- 保存时，需要同时将套餐的信息保存到setmeal表已经将套餐与菜品的对应关系保存到setmeal\_dish表中，需要在service接口中新增一个方法

```
/** 套餐值传输对象
 * @author ProgZhou
 * @createTime 2022/06/07
 */
@EqualsAndHashCode(callSuper = true)
@Data
@NoArgsConstructor
public class SetmealDto extends Setmeal {
    private List<SetmealDish> setmealDishes;

    /*
     * 套餐分类名称
     */
    private String categoryName;
}
```

```
@Override
```

```

@Transactional
public void addSetmealWithDish(SetmealDto setmealDto) {
    //保存套餐的基本信息
    setmealMapper.insert(setmealDto);

    List<SetmealDish> setmealDishList = setmealDto.getSetmealDishes();

    //添加每个套餐的分类id
    setmealDishList = setmealDishList.stream().peek((item) ->
        item.setSetmealId(setmealDto.getId())).collect(Collectors.toList());

    //保存套餐和菜品的关联信息
    setmealDishService.saveBatch(setmealDishList);
}

//controller层
/*
 * 新增套餐
 */
@PostMapping
public CommonResult<String> addSetMeal(@RequestBody SetmealDto setmealDto)
{
    log.info("接收到数据: {}", setmealDto);

    setmealService.addSetmealWithDish(setmealDto);
    return CommonResult.success("新增套餐成功");
}

```

## 8.2 套餐信息分页查询

与之前的菜品信息分页查询相似，需要使用SetmealDto作为数据传输对象接收

```

/*
 * 套餐分页查询
 */
@GetMapping("/page")
public CommonResult<Page<SetmealDto>> getPages(int page, int pageSize, String
name) {
    Page<SetmealDto> pageInfo = new Page<>();
    Page<Setmeal> setmealPage = new Page<>(page, pageSize);

    //首先分页查询套餐信息
    LambdaQueryWrapper<Setmeal> wrapper = new LambdaQueryWrapper<>();
    wrapper.like(name != null, Setmeal::getName, name);
    wrapper.orderByDesc(Setmeal::getUpdateTime);
    setmealService.page(setmealPage, wrapper);
    //将除了记录的属性复制到SetmealDto的page封装对象中
    BeanUtils.copyProperties(setmealPage, pageInfo, "records");

    List<Setmeal> setmeals = setmealPage.getRecords();
}

```

```

//将分类的名称添加到套餐数据传输对象实体中
List<SetmealDto> list = setmeals.stream().map((item) -> {
    SetmealDto setmealDto = new SetmealDto();

    Category category = categoryService.getById(item.getCategoryId());

    if(category != null) {
        BeanUtils.copyProperties(item, setmealDto);
        setmealDto.setCategoryName(category.getName());
    }
    return setmealDto;
}).collect(Collectors.toList());

//将SetmealDto中的records设置为处理之后的集合
pageInfo.setRecords(list);

return CommonResult.success(pageInfo);
}

```

## 8.3 删除套餐

### 需求分析

在套餐管理列表页面点击删除按钮，可以删除对应的套餐信息，也可以通过复选框选择多个套餐，点击批量删除一次删除多个套餐

对于状态为起售的套餐不能直接删除，需要先停售再删除

在删除套餐信息的同时，需要删除setmeal\_dish表中套餐与菜品的对应信息

### 代码开发

由于需要操作两张表，在SetmealService中添加一个方法：

```

/*
 * 按照id删除套餐，同时删除套餐和菜品的关联数据
 */
void deleteWithDish(List<Long> ids);

@Override
@Transactional
public void deleteWithDish(List<Long> ids) {
    //查询套餐状态，确定是否能够删除套餐
    LambdaQueryWrapper<Setmeal> wrapper = new LambdaQueryWrapper<>();
    wrapper.in(Setmeal::getId, ids);
    wrapper.eq(Setmeal::getStatus, 1);
}

```

```

Integer count = setmealMapper.selectCount(wrapper);
if(count > 0) {
    //如果count > 0说明待删除的套餐中有正在售卖的，则需要抛出异常
    throw new ServiceException("套餐正在售卖中，不能删除");
}

//删除套餐表中的数据
setmealMapper.deleteBatchIds(ids);

//删除菜品的关联数据
LambdaQueryWrapper<SetmealDish> queryWrapper = new LambdaQueryWrapper<>();
queryWrapper.in(SetmealDish::getSetmealId, ids);

setmealDishMapper.delete(queryWrapper);
}

```

controller层：

```

//对传过来的ids进行一定处理
@DeleteMapping
public CommonResult<String> deleteSetMeals(@RequestParam("ids") String ids) {
    log.info("待删除的套餐id: {}", ids);

    List<Long> setmealIds = new ArrayList<>();
    if(ids.contains(",")) {
        //批量删除
        String[] temp = ids.split(",");
        for (String id : temp) {
            setmealIds.add(Long.parseLong(id));
        }
        setmealService.deleteWithDish(setmealIds);
    } else {
        //单个删除
        setmealIds.add(Long.parseLong(ids));
        setmealService.deleteWithDish(setmealIds);
    }

    return null;
}

```

## 九、手机验证码登录

# 9.1 手机验证码登录功能

## 需求分析

为了方便用户登录，移动端通常会提供通过手机验证码登录的功能

手机验证码登录的优点：

- 方便快捷，无需注册，直接登录
- 使用短信验证码作为登录凭证
- 安全

登录流程：

输入手机号 -> 获得验证码 -> 输入验证码 -> 点击登录

## 交互过程

- 在登录页面输入手机号，点击【获取验证码】按钮，页面发送ajax请求，在服务端调用短信服务API给指定手机发送验证码短信
- 在登录界面输入验证码，点击登录，前端页面再发送一个ajax请求，并在服务端处理这个请求

## 代码开发

当前端页面点击【获取验证码】时，会发送ajax请求，后端需要接收并处理这个请求

```
/*
 * 发送验证码
 * 由于没有阿里云短信服务，这个就做个样子看看
 */
public CommonResult<String> sendMessage(@RequestBody User user, HttpSession session) {

    String phone = user.getPhone();

    if(!StringUtils.isEmpty(phone)) {
        //使用工具类生成随机的4位验证码
        String code = ValidateCodeUtils.generateValidateCode(4).toString();
        log.info("code = {}", code);
        //使用阿里云短信服务发送服务，第一个参数为注册的签名，第二个为注册审核通过之后
        的代码
        //SMSUtils.sendMessage("", "", phone, code);

        //将生成的验证码保存起来，暂时保存在session中
        session.setAttribute(phone, code);
        return CommonResult.success("短信发送成功");
    }

    return CommonResult.error("短信发送失败");
}
```

点击登录按钮之后会发送登录请求

```
/*
 * 登录方法
 */
@PostMapping("/login")
public CommonResult<User> login(@RequestBody Map<String, Object> map,
 HttpSession session) {
    log.info("接收到的数据: {}", map);

    //获取手机号
    String phone = map.get("phone").toString();
    //获取验证码
    //String code = map.get("code");
    //对比接收到的验证码和session中保存的code
    //String codeInSession = session.getAttribute(phone);
    //if(codeInSession != null && code.equals(codeInSession)) {

        //判断当前手机号是否是新用户
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getPhone, phone);
        User user = userService.getOne(wrapper);
        if(null == user) {
            //如果是新用户，则自动注册
            user = new User();
            user.setPhone(phone);
            userService.save(user);
        }
        session.setAttribute("user", user.getId());
        return CommonResult.success(user);
    //}
}
```

## 十、菜品展示、购物车、下单

### 10.1 导入用户地址簿

#### 需求分析

用户登录之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

可以设置一个默认地址，也可以修改已经存在的地址

#### 代码开发

```
/**  
 * @author ProgZhou  
 * @createTime 2022/06/08  
 */  
  
@RestController  
@RequestMapping("/addressBook")  
@Slf4j  
public class AddressBookController {  
  
    @Autowired  
    private AddressBookService addressBookService;  
  
    /*  
     * 新增用户收货地址  
     * */  
    @PostMapping  
    public CommonResult<AddressBook> addAddress(@RequestBody AddressBook addressBook) {  
        log.info("接收到的地址信息: {}", addressBook);  
        //设置用户id  
        addressBook.setUserId(BaseContext.getCurrentUserId());  
        addressBookService.save(addressBook);  
  
        return CommonResult.success(addressBook);  
    }  
  
    /*  
     * 设置默认收获地址  
     * */  
    @PutMapping("/default")  
    public CommonResult<AddressBook> setDefault(@RequestBody AddressBook addressBook) {  
        log.info("address: {}", addressBook);  
        LambdaUpdateWrapper<AddressBook> wrapper = new LambdaUpdateWrapper<>()  
            .eq(AddressBook::getUserId, BaseContext.getCurrentUserId())  
            .set(AddressBook::getIsDefault, 0)  
            //先将所有地址都不设置为默认地址  
            .update(wrapper);  
  
        addressBookService.update(wrapper);  
  
        addressBook.setIsDefault(1);  
        addressBookService.updateById(addressBook);  
        return CommonResult.success(addressBook);  
    }  
  
    /*  
     * 根据id查询地址信息  
     * */  
    @GetMapping("/{id}")  
    public CommonResult<AddressBook> getAddress(@PathVariable("id") Long id) {
```

```
    log.info("待查询的id: {}", id);
    AddressBook addressBook = addressBookService.getById(id);
    if(null == addressBook) {
        return CommonResult.error("没有查询到地址");
    }
    return CommonResult.success(addressBook);
}

/*
 * 查询默认地址信息
 */
@GetMapping("/default")
public CommonResult<AddressBook> getDefault() {
    LambdaQueryWrapper<AddressBook> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(AddressBook::getUserId, BaseContext.getCurrentUserId());
    wrapper.eq(AddressBook::getIsDefault, 1);

    AddressBook addressBook = addressBookService.getOne(wrapper);
    if(null == addressBook) {
        return CommonResult.error("还没有设置默认地址, 请设置");
    }

    return CommonResult.success(addressBook);
}

/*
 * 查询某个用户的所有地址信息
 */
@GetMapping("/list")
public CommonResult<List<AddressBook>> getList(AddressBook addressBook) {
    addressBook.setUserId(BaseContext.getCurrentUserId());
    LambdaQueryWrapper<AddressBook> wrapper = new LambdaQueryWrapper<>();

    wrapper.eq(addressBook.getUserId() != null, AddressBook::getUserId,
addressBook.getUserId());
    wrapper.orderByDesc(AddressBook::getUpdateTime);
    List<AddressBook> addressBookList = addressBookService.list(wrapper);

    return CommonResult.success(addressBookList);
}

/*
 * 根据id删除某个用户的地址信息
 */
@DeleteMapping
public CommonResult<String> deleteAddress(@RequestParam("ids") Long ids) {
    log.info("待删除的地址: {}", ids);
    addressBookService.removeById(ids);
    return CommonResult.success("删除成功");
}
```

```
}
```

## 10.2 菜品展示

### 需求分析

用户登录成功之后跳转到系统首页，在首页需要根据分类来展示菜品和套餐，如果菜品设置了口味信息，如果需要显示【选择规格】按钮

### 代码开发

之前写过按照分类查询菜品的代码，但需要修改一下，由于在客户端展示的信息需要根据菜品是否有口味来显示相应的按钮，但Dish实体类中并没有相关的口味信息，所以，需要将返回值的类型改成DishDto

```
/*
 * 根据分类id查询菜品信息，并且返回DishDto对象
 */
@GetMapping("/list")
public CommonResult<List<DishDto>> getDishByCategoryId(Dish dish) {
    log.info("得到categoryId: {}", dish.getCategoryId());

    LambdaQueryWrapper<Dish> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(dish.getCategoryId() != null, Dish::getCategoryId,
    dish.getCategoryId());
    //只查询正在售卖的菜品
    wrapper.eq(Dish::getStatus, 1);

    wrapper.orderByAsc(Dish::getSort).orderByDesc(Dish::getUpdateTime);
    List<Dish> dishList = dishService.list(wrapper);

    List<DishDto> dishDtoList = dishList.stream().map((item) -> {
        DishDto dto = new DishDto();
        BeanUtils.copyProperties(item, dto);
        //根据id查询分类对象
        Long categoryId = item.getCategoryId();
        if(categoryId != null) {
            Category category = categoryService.getById(categoryId);
            //设置分类名称
            dto.setCategoryName(category.getName());
        }
        LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>
    ());
        queryWrapper.eq(DishFlavor::getDishId, item.getId());
        List<DishFlavor> dishFlavors = dishFlavorService.list(queryWrapper);
        dto.setFlavors(dishFlavors);
        return dto;
    }).collect(Collectors.toList());

    return CommonResult.success(dishDtoList);
}
```

# 10.3 购物车

## 需求分析

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，则需要选择规格之后才能加入购物车；对于套餐来说，可以直接加入购物车

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

## 代码开发

购物车可以添加和减少，所以需要两个controller方法

```
/*
 * 将菜品或者套餐添加到购物车
 */
@PostMapping("/add")
public CommonResult<ShoppingCart> addShoppingCart(@RequestBody ShoppingCart
shoppingCart) {
    log.info("待添加的商品: {}", shoppingCart);
    //设置用户id
    Long userId = BaseContext.getCurrentUserId();
    shoppingCart.setUserId(userId);
    //查询当前表中是否有此商品
    Long dishId = shoppingCart.getDishId();
    LambdaQueryWrapper<ShoppingCart> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(ShoppingCart::getUserId, userId);
    if(dishId != null) {
        //添加的是菜品
        wrapper.eq(ShoppingCart::getDishId, shoppingCart.getDishId());
        //由于前端在添加第二份时不能选择口味，这个条件就不成立，如果有这个需求可以添加
        // wrapper.eq(ShoppingCart::getDishFlavor,
        shoppingCart.getDishFlavor());
    } else {
        //添加的是套餐
        wrapper.eq(ShoppingCart::getSetmealId, shoppingCart.getSetmealId());
    }
    ShoppingCart cart = shoppingCartService.getOne(wrapper);
    if(cart != null) {
        //如果已经存在，就在原来数量的基础上 + 1
        cart.setNumber(cart.getNumber() + 1);
        shoppingCartService.updateById(cart);
    } else {
        //如果不存在，就直接添加到数据表中
        shoppingCart.setNumber(1);
        shoppingCart.setCreateTime(LocalDateTime.now());
        shoppingCartService.save(shoppingCart);
    }
    return CommonResult.success(shoppingCart);
}
```

```

@PostMapping("/sub")
public CommonResult<ShoppingCart> subShoppingCart(@RequestBody ShoppingCart
shoppingCart) {
    Long userId = BaseContext.getCurrentUserId();
    shoppingCart.setUserId(userId);
    LambdaQueryWrapper<ShoppingCart> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(ShoppingCart::getUserId, userId);
    if(shoppingCart.getDishId() != null) {
        wrapper.eq(ShoppingCart::getDishId, shoppingCart.getDishId());
    } else {
        wrapper.eq(ShoppingCart::getSetmealId, shoppingCart.getSetmealId());
    }
    ShoppingCart cart = shoppingCartService.getOne(wrapper);
    if(cart.getNumber() > 1) {
        cart.setNumber(cart.getNumber() - 1);
        shoppingCartService.updateById(cart);
    } else {
        shoppingCartService.remove(wrapper);
    }

    return CommonResult.success(cart);
}

```

并且需要查询当前用户所有添加到购物车中的商品：

```

@GetMapping("/list")
public CommonResult<List<ShoppingCart>> getList() {
    LambdaQueryWrapper<ShoppingCart> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(ShoppingCart::getUserId, BaseContext.getCurrentUserId());
    wrapper.orderByDesc(ShoppingCart::getCreateTime);

    List<ShoppingCart> shoppingCarts = shoppingCartService.list(wrapper);

    return CommonResult.success(shoppingCarts);
}

```

## 10.4 下单

### 需求分析

移动端的用户将菜品或者套餐加入购物车之后，可以点击【去结算】按钮，页面跳转到订单确认页面，点击【去支付】完成下单操作

由于支付业务需要申请，所以，这里的支付业务仅仅将订单保存至数据库中  
交互流程：

- 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，首先前端会发送一次ajax请求，获取当前用户的默认地址

- 前端还会发送一次ajax请求来获取当前用户的购物车信息
- 点击【去支付】，再次发送ajax请求，完成下单功能

# 十一、缓存优化

## 11.1 项目存在问题

由于每一次的请求都需要去访问数据库，如果用户数量多，系统访问量大，导致请求频繁地访问数据库，致使性能下降，用户体验差

## 11.2 缓存短信验证码

### 实现思路

之前的版本中，生成的验证码是保存在session中，session的过期时间默认是30min，而一般情况下，收集验证码的过期时间是2min或者5min，所以可以存储在redis中并设置过期时间

- 在服务端的UserController中注入redisTemplate，用于操作redis
- 在服务端的sendMessage中，将随机生成的验证码缓存到redis中，并设置有效期，这里设置为5min
- 修改服务端的login方法，从redis中获取缓存的验证码，如果登录成功，则删除redis中的验证码信息

### 代码开发

```
/*
 * 发送验证码
 * 由于没有阿里云短信服务，这个就做个样子看看
 */
public CommonResult<String> sendMessage(@RequestBody User user,
HttpSession session) {

    String phone = user.getPhone();

    if(!StringUtils.isEmpty(phone)) {
        //使用工具类生成随机的4位验证码
        String code =
ValidateCodeUtils.generateValidateCode(4).toString();
        log.info("code = {}", code);
        //使用阿里云短信服务发送服务
        //SMSUtils.sendMessage("", "", phone, code);
        //将生成的验证码缓存到redis中，并设置过期时间为5min
        redisTemplate.opsForValue().set(phone, code, 5, TimeUnit.MINUTES);
        return CommonResult.success("短信发送成功");
    }
}
```

```
}

    /*
 * 登录方法
 */
@PostMapping("/login")
public CommonResult<User> login(@RequestBody Map<String, Object> map,
 HttpSession session) {
    log.info("接收到的数据: {}", map);

    //获取手机号
    String phone = map.get("phone").toString();
    //获取验证码
    //String code = map.get("code");

    //方案二: 从redis中获取缓存的校验码
    String codeInRedis =
redisTemplate.opsForValue().get(phone).toString();

    //if(codeInSession != null && code.equals(codeInSession)) {}

    //判断当前手机号是否是新用户
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(User::getPhone, phone);
    User user = userService.getOne(wrapper);
    if(null == user) {
        //如果是新用户, 则自动注册
        user = new User();
        user.setPhone(phone);
        userService.save(user);
    }
    session.setAttribute("user", user.getId());

    //用户登录成功, 删除缓存中的验证码
    redisTemplate.delete(phone);

    return CommonResult.success(user);
}
```

## 11.3 缓存菜品数据

### 实现思路

之前写的方法, 当用户登录之后, 点击菜品分类之后, 都会去查询数据库展示分类下的菜品, 在高并发的情况下, 数据库的压力很大, 所以需要对数据进行缓存

- 改造DishController的方法, 先从redis中获取菜品的数据, 如果没有, 再去数据库查询

- 并且需要改造更新的方法，数据库中数据更新的时候，需要清楚缓存中的旧数据，一般有两种清理方式：
  - 清除redis中所有的数据
  - 精确清理修改过分类下的菜品数据
- 由于查询是按照菜品分类查询的，所以在redis中缓存数据的时候，需要按照分类的id来缓存数据

```

/*
 * 根据分类id查询菜品信息，并且返回DishDto对象
 */
@GetMapping("/list")
public CommonResult<List<DishDto>> getDishByCategoryId(Dish dish) {
    log.info("得到categoryId: {}", dish.getCategoryId());

    List<DishDto> dishDtoList = null;

    //1. 尝试从redis中获取数据
    //拼接key
    String dishKey = "Dish_" + dish.getCategoryId() + "_status_" +
dish.getStatus();
    dishDtoList = (List<DishDto>) redisTemplate.opsForValue().get(dishKey);
    if(dishDtoList != null) {
        //2. 如果存在，则直接返回
        return CommonResult.success(dishDtoList);
    }

    //3. 如果不存在，则需要查询数据库
    LambdaQueryWrapper<Dish> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(dish.getCategoryId() != null, Dish::getCategoryId,
dish.getCategoryId());
    //只查询正在售卖的菜品
    wrapper.eq(Dish::getStatus, 1);

    wrapper.orderByAsc(Dish::getSort).orderByDesc(Dish::getUpdateTime);
    List<Dish> dishList = dishService.list(wrapper);

    dishDtoList = dishList.stream().map((item) -> {
        DishDto dto = new DishDto();
        BeanUtils.copyProperties(item, dto);
        //根据id查询分类对象
        Long categoryId = item.getCategoryId();
        if(categoryId != null) {
            Category category = categoryService.getById(categoryId);
            //设置分类名称
            dto.setCategoryName(category.getName());
        }
        LambdaQueryWrapper<DishFlavor> queryWrapper = new LambdaQueryWrapper<>
();
        queryWrapper.eq(DishFlavor::getDishId, item.getId());
        List<DishFlavor> dishFlavors = dishFlavorService.list(queryWrapper);
        dto.setFlavors(dishFlavors);
    });
}

```

```
        return dto;
    }).collect(Collectors.toList()));

    //将查询出来的数据缓存到redis中
    redisTemplate.opsForValue().set(dishKey, dishDtoList, 30,
TimeUnit.MINUTES);

    return CommonResult.success(dishDtoList);
}
```

## 11.4 缓存套餐数据

### 实现思路

在移动端查询套餐功能的方法中SetmealController的list方法，此方法会根据前端提交的查询条件进行数据库查询操作，在高并发的情况下，频繁查询数据库会导致系统性能下降，服务端响应时间增长，现在需要对此方法进行缓存优化，提高系统性能

- 导入Spring Cache和Redis的maven坐标
- 在application.yml中配置缓存数据的过期时间
- 在启动类上加入@EnableCaching注解，开启缓存注解功能
- 在SetmealController的list方法上加入@Cacheable注解
- 在SetmealController的save和delete方法上加入CacheEvict注解

### 代码实现

```
<!--导入spring cache的maven坐标-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

修改application.yml配置文件

```
spring:
  cache:
    redis:
      time-to-live: 180000 #设置缓存数据的过期时间
      type: redis
```

### 添加缓存

```
/*
 * 根据分类查询套餐信息
 */
```

```

@GetMapping("/list")
//将方法的返回结果存入redis缓存，返回结果如果是引用数据类型需要实现Serializable接口实现序列化
@Cacheable(value = "setmealCache", key = "'setMeal' + #setmeal.categoryId +
'_' + #setmeal.status")
public CommonResult<List<Setmeal>> getList(Setmeal setmeal) {
    LambdaQueryWrapper<Setmeal> wrapper = new LambdaQueryWrapper<>();
    wrapper.eq(Setmeal::getCategoryId, setmeal.getCategoryId());
    wrapper.eq(Setmeal::getStatus, 1);
    wrapper.orderByDesc(Setmeal::getUpdateTime);

    List<Setmeal> setmealList = setmealService.list(wrapper);

    return CommonResult.success(setmealList);
}

//在更新和删除套餐时，也需要将缓存中的数据删除
@DeleteMapping
@CacheEvict(value = "setmealCache", allEntries = true)      //当删除某个套餐时，需要删除缓存中的套餐数据
public CommonResult<String> deleteSetMeals(@RequestParam("ids") String ids)

@PostMapping
@CacheEvict(value = "setmealCache", allEntries = true)
public CommonResult<String> addSetMeal(@RequestBody SetmealDto setmealDto)

```

测试效果：控制台的日志中再次访问时不会输出sql语句，并且redis缓存中有相应的数据

```

2022-06-17 16:44:08.141 INFO 10104 --- [nio-8080-exec-1] com.project.reggie.filter.LoginFilter : 拦截到请求: http://localhost:8080/setmeal/list
2022-06-17 16:44:08.142 INFO 10104 --- [nio-8080-exec-1] com.project.reggie.filter.LoginFilter : 当前登录用户id: 1534440010319663106
2022-06-17 16:44:08.153 INFO 10104 --- [nio-8080-exec-7] com.project.reggie.filter.LoginFilter : 拦截到请求: http://localhost:8080/common/download
2022-06-17 16:44:08.153 INFO 10104 --- [nio-8080-exec-7] com.project.reggie.filter.LoginFilter : 当前登录用户id: 1534440010319663106
2022-06-17 16:44:08.161 INFO 10104 --- [nio-8080-exec-3] com.project.reggie.filter.LoginFilter : 拦截到请求: http://loc: Key Promoter X
2022-06-17 16:44:08.161 INFO 10104 --- [nio-8080-exec-3] com.project.reggie.filter.LoginFilter : 当前登录用户id: 1534440010319663106
                                         Command 失败程度 missed 235 time(s)
                                         'eg... (Disable alert for this shortcut'

```

```

127.0.0.1:6379> keys *
1) "setmealCache::setMeall1413386191767674881_1"
2) "setmealCache::setMeall1413342269393674242_1"
3) "Dish_1397844263642378242_status_1"

```

## 十二、读写分离

### 12.1 MySQL主从复制

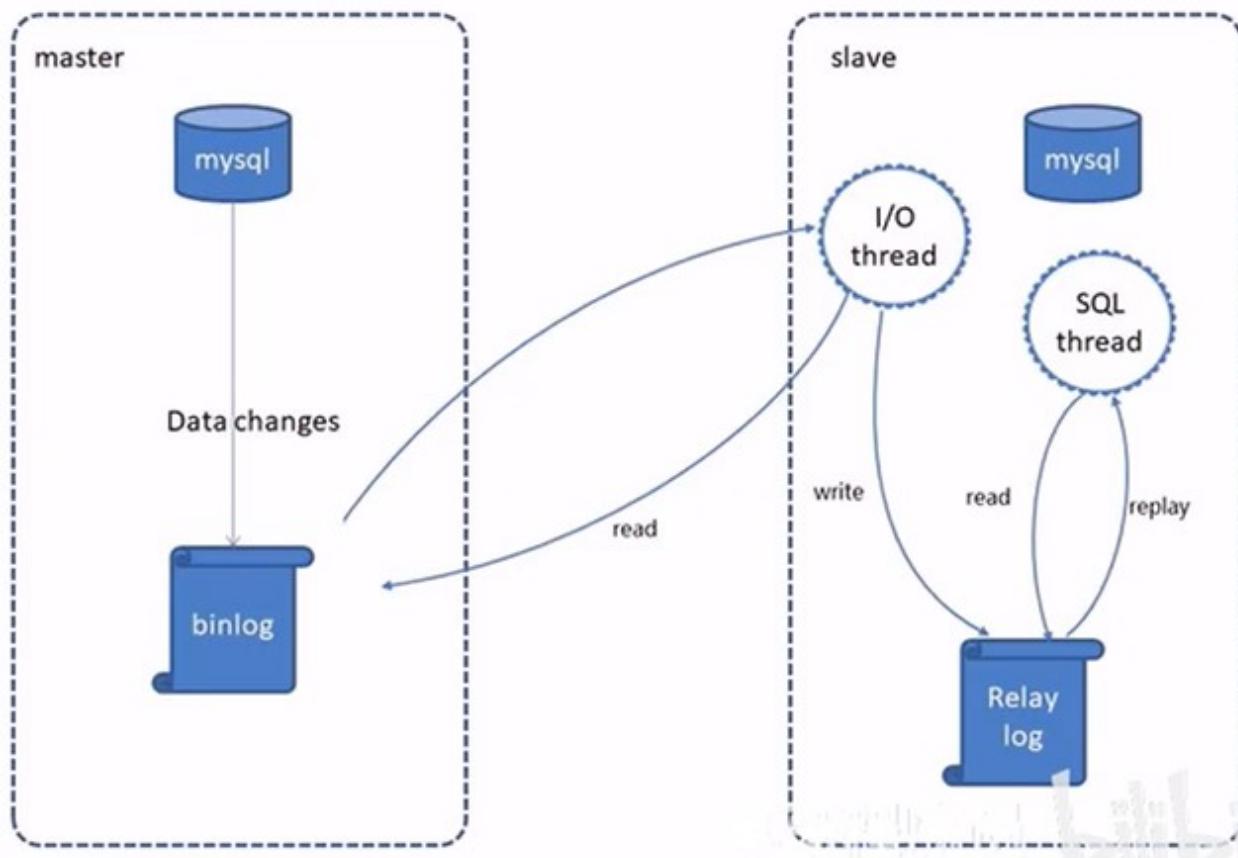
#### 基本介绍

MySQL主从复制是一个异步的复制过程，底层基于MySQL数据库自带的二进制日志功能，就是一台或多台MySQL数据库（slave，即从库）从另一台MySQL数据库（master，即主库）进行日志的复制然后再解析日志并应用到自身，最终实现从库的数据和主库的数据保持一致，MySQL主从复制是MySQL数据库底层支持的功能，无需借助第三方工具

主库进行增删改操作，从库进行查询操作

MySQL主从复制过程分为三步：

- master将写数据的操作记录到二进制日志中（binlog）
- slave将master的binlog拷贝到它的中继日志（relay log）
- slave重做中继日志中的事件，将改变应用到自己的数据库中



## 基本主从搭建

使用一台centos8的虚拟机作为主机（阿里云ECS云服务器），另一台centos8的虚拟机作为从机（本地centos8虚拟机）

第一步：搭建主机

- 修改主机的mysql配置文件

```
> vim /etc/my.cnf #Linux中mysql默认的配置文件路径

# For advice on how to change settings please see
# http://dev.mysql.com/doc/refman/8.0/en/server-configuration-defaults.html

[mysqld]
#添加以下配置内容
```

```

#[必须]主服务器唯一ID
server-id=100
#[必须]启用二进制日志,指名路径。比如:自己本地的路径/log/mysqlbin
log-bin=mysql-bin

#配置文件内容
...
> systemctl restart mysqld    #修改完配置文件之后,重启mysql服务

```

- 登录主机的mysql服务器,建立一个新账户并授权,注意在mysql8.0版本之后一定要这么干

```

mysql> CREATE USER 'slave1'@'%' IDENTIFIED BY 'Root@123456';
mysql> GRANT REPLICATION SLAVE ON *.* TO 'slave1'@'%';  #授予slave用户主从复制的权限
mysql> ALTER USER 'slave1'@'%' IDENTIFIED WITH mysql_native_password BY
'Root@123456';  #在mysql8.0版本下必须执行这个语句
mysql> flush privileges;

#如果不执行第三行中的指令,在之后的配置中会报错Last_IO_Error: error connecting
to master 'slave1@192.168.1.150:3306' - retry-time: 60 retries: 1
#message: Authentication plugin 'caching_sha2_password' reported error:
Authentication requires
#secure connection.

```

- 使用show master status;命令查看主机的binlog状态,为之后的从机配置做准备

```

mysql> show master status;
+-----+-----+-----+-----+
| File      | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_
+-----+-----+-----+-----+
| mysql-bin.000001 |     1145 |           |           |           |
+-----+-----+-----+-----+

```

- 到此为止,主机配置完毕

## 第二步: 从机配置

- 在本机的虚拟机中,修改从机的配置文件

```

> vim /etc/my.cnf    #Linux中mysql默认的配置文件路径

# For advice on how to change settings please see
# http://dev.mysql.com/doc/refman/8.0/en/server-configuration-defaults.html

[mysqld]
#添加以下配置内容
#[必须]主服务器唯一ID
server-id=101
#[可选]启用中继日志

```

```
relay-log=mysql-relay  
#配置文件内容  
...  
> systemctl restart mysql      #修改完成之后仍然需要重启mysql服务
```

- 登录mysql服务器，输入以下命令

```
mysql> change master to master_host='8.130.8.14', master_user='slave1',  
master_password='Root@123456', master_log_file='mysql-bin.000001',  
master_log_pos=1145;
```

#其中master\_user是刚刚在主机中授权的用户名  
#master\_password是授权用户名的密码  
#master\_log\_file是之前使用show master status查询出来的主机的binlog名  
#master\_log\_pos是配置文件的位置，也是由master中查出来的数据

- 使用start slave开启主从同步

```
mysql> start slave;
```

- 接着，查看从机状态，看看主从是否搭建成功

```
mysql> show slave status\G;  
*****  
          1. row *****  
Slave_IO_State: Waiting for source to send event  
Master_Host: 8.130.8.14  
Master_User: slave1  
Master_Port: 3306  
Connect_Retry: 60  
Master_Log_File: mysql-bin.000001  
Read_Master_Log_Pos: 1145  
Relay_Log_File: mysql-relay.000002  
Relay_Log_Pos: 325  
Relay_Master_Log_File: mysql-bin.000001  
Slave_IO_Running: Yes  
Slave_SQL_Running: Yes      #只要IO_Running和SQL_Running两个项中是  
yes状态，表明主从搭建成功  
#其余信息  
....  
1 row in set, 1 warning (0.00 sec)
```

- 可以使用stop slave停止主从，停止之后仍然可以使用start slave再次开启主从同步

## 12.2 读写分离demo

### 为什么要读写分离

面对日益增加的系统访问量，数据库的吞吐量面临巨大瓶颈，对于同一时刻有 **大量并发操作** 和 **较少写操作** 类型的应用系统来说，将数据库拆分为主库和从库，主库负责处理事务的增删改操作，从库负责处理查询操作，能够有效的避免由数据更新导致的行锁，使得整个系统的查询性能得到极大的改善

### Java层面如何实现读写分离

Sharding-JDBC框架：定位为轻量级Java框架，在Java的JDBC层提供的额外服务，他使用客户端直连数据库，以jar包形式提供服务，无需额外部署和依赖，可理解为增强版JDBC驱动，完全兼容JDBC和各种ORM框架

使用Sharding JDBC可以在程序中轻松的实现数据库读写分离

- 适用于任何基于JDBC的ORM框架，比如：JPA，Hibernate，Mybatis，Spring JDBC Template或直接使用JDBC
- 支持任何第三方数据库连接池，比如：DBCP，C3P0，Druid，HikariCP（Springboot默认使用的连接池）等
- 支持任意实现JDBC规范的数据库，目前支持MySQL，Oracle，SQLServer，PostgreSQL以及任何遵循SQL92标准的数据库

### 简单demo

首先需要搭建好MySQL主从复制的整体架构

- 导入maven坐标
- 在配置文件中配置读写分离规则
- 在配置文件中配置允许bean定义覆盖配置项

新建一个工程，作为Sharding JDBC的demo工程，在主库中创建一张数据表，并在工程中创建好表对应的实体类，mapper和service

```
mysql> create database master_slave;
mysql> use master_slave;
mysql> create table `user`(
>   id int(4) primary key,
>   name varchar(10) not null,
>   age int(4) not null
> );
```

在工程的pom文件中导入Sharding JDBC的maven坐标

```
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
  <version>4.0.0-RC1</version>
</dependency>
```

## 配置读写分离的规则

```
spring:
  shardingsphere:
    datasource:
      names: #可能有多个数据源，如果有多个就定义多个，但是需要与下面配置数据源的配置项相同
        master,slave
      #主数据源
      master: #对应names中的master
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: 主库的url
        username: xxx
        password: xxx
      slave: #对用names中的slave
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: 从库的url
        username: xxx
        password: xxx
    masterslave:
      load-balance-algorithm-type: round_robin #负载均衡算法，如果有多个从库，对于查询操作的从库选择
        name: dataSource #最终数据源的名称，即DataSource对象的名称
        master-data-source-name: master #主库的名称，即前面配置的name属性中主库的名称
        slave-data-source-names: slave #从库的名称，如果有多个，可以用","分隔
    pros:
      sql:
        show: true #开启sql显示，默认为false
```

## 在配置文件中配置允许bean定义覆盖配置项

```
#在配置文件中增加一项，在spring级下
main:
  #这个配置默认为false
  allow-bean-definition-overriding: true #由于引入了Sharding JDBC和Druid两个jar包，而里面都含有一个dataSource，在springboot启动时就会起冲突
```

配置完成以后，不用修改其他的Java代码，即可实现主从复制的功能

## 12.3 优化外卖项目 -- 实现读写分离

# 十三、前后端分离

## 13.1 当前项目存在的问题

前后端耦合，在整个项目中，既有前端代码，也有后端代码，开发人员需要同时维护前端代码和后端代码，分工不明确，并且开发效率比较低

前后端代码合成在一起，不利于管理

解决方法：**前后端分离开发**

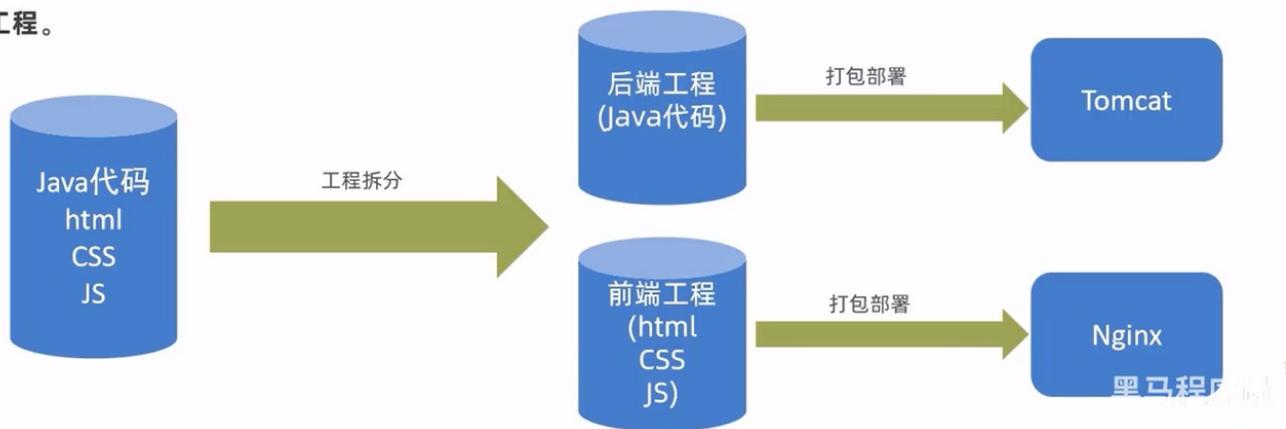
## 13.2 前后端分离开发

### 前后端分离开发介绍

在项目开发过程中，对于前端代码的开发由专门的前端开发人员负责，后端代码则由后端开发人员负责，这样就可以做到分工明确，各司其职，提高开发效率，前后端代码并行开发，可以加快项目开发进度

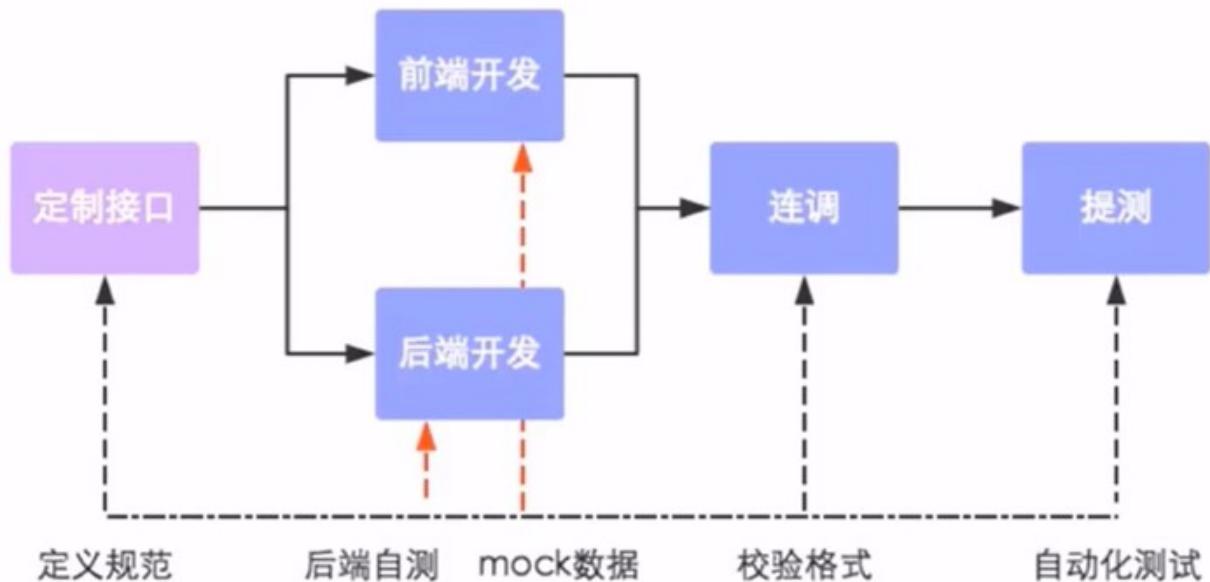
前后端分离开发后，从工程结构上也会发生变化，即前后端代码不再混合在同一个maven工程中，而是分为前端工程和后端工程

工程。



### 开发流程

总体的开发流程：



- 定制接口：这个接口指的是一个http请求，主要定义请求路径、请求方式、请求参数和响应数据等内容

**这个步骤非常关键**

- 前后端并行开发，后端使用postman或者Swagger进行测试
- 连调：前后端结合测试

## 前端常用技术栈

开发工具：

- Visual Studio Code
- Hbuilder

技术架构：

- node.js
- vue
- ElementUI
- mock (测试工具)
- webpack (打包工具)

## 13.3 YApi

### 介绍

YApi是高效、易用。功能强大的api管理平台，旨在为开发、产品、测试人员提供更优雅的接口管理服务，可以帮助开发者轻松创建、发布、维护API，YApi还为用户提供了优秀的交互体验，开发人员只需要利用平台提供的接口数据写入工具以及简单的点击操作就可以实现接口管理。

## 13.4 Swagger

### 介绍

使用Swagger只需要按照它定义的规范去定义接口以及接口相关的信息，再通过Swagger衍生出来的一系列项目和工具，就可以做到生成各种格式的接口文档，以及在线接口调试页面等等

官网：<https://swagger.io/>

knife4j是为Java MVC框架集成Swagger生成API文档的增强解决方案

```
<!--maven坐标-->
<dependency>
    <groupId>com.github.xiaoymin</groupId>
    <artifactId>knife4j-spring-boot-starter</artifactId>
    <version>3.0.2</version>
</dependency>
```

### 使用方式

- 导入knife4j的maven坐标
- 导入相关配置类
- 设置静态资源，否则接口文档页面无法访问
- 在Filter或者Interceptor中设置不需要拦截的请求路径

### Swagger的常用注解

注解	说明
@Api	用在请求的类上，例如Controller，表示对类的说明
@ApiModelProperty	用在类上，通常是实体类，表示一个返回响应数据的信息
@ApiModelPropertyProperty	用在属性上，描述响应类的属性
@ApiOperation	用在请求的方法上，说明方法的用途、作用
@ApiImplicitParams	用在请求的方法上，表示一组参数说明
@ApiImplicitParam	用在@ApiImplicitParams注解中，指定一个请求参数的各个方面

## 13.5 项目部署

# 附录1 Git的使用

## Git的作用

- 代码回溯，如果代码出错，可以回溯到历史版本
- 版本切换
- 多人协作
- 远程备份

## Git代码托管服务

### 常用的git代码托管服务

- github
- gitee
- GitLab
- BitBucket

## Git常用命令

### 设置用户信息

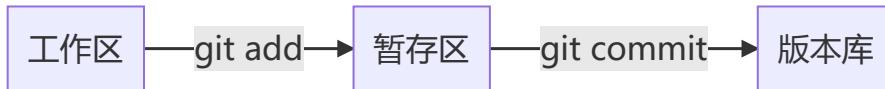
```
git config --global user.name = "xxx"  
git config --global user.email = "xxx"  
  
#查看配置信息  
git config --list
```

### 获取git仓库

```
#获取本地初始化仓库  
git init  
  
#获取远程仓库  
git clone https://xxx #仓库地址
```

### Git的相关概念

- **版本库**：在git仓库中会有一个隐藏文件.git，这个就是仓库的版本库，版本库中存储了很多配置信息、日志信息和文件版本等
- **工作区**：含.git文件夹的目录就是工作区，也称为工作目录，主要用于存放开发的代码
- **暂存区**：.git文件夹中有很多文件，其中就有一个index文件，这个就是暂存区，临时保存修改文件的地方



## Git工作区中文件的状态

- **unchecked** 未跟踪（未被纳入版本控制）
- **tracked** 已跟踪（被纳入版本控制）
  - **unmodified** 未修改状态
  - **modified** 已修改状态
  - **staged** 暂存状态

## 本地仓库操作

```

#查看文件的状态
git status

#将文件的修改加入暂存区
git add

#将暂存区的文件取消暂存或者是切换到指定版本
git reset

#将暂存区的文件修改提交到本地库
git commit

#查看git日志
git log
  
```

## 远程仓库操作

```

#查看远程仓库
git remote
(git remote -v 查看详细信息)

#添加远程仓库
git remote add

#从远程仓库克隆
git clone

#从远程仓库拉取
git pull [remote name][branch name]
  
```

```
#推动到远程仓库  
git push [remote name][branch name]
```

## 分支操作

使用分支意味着开发时可以从主线开发中分离开来，以免影响主线开发

同一个仓库可以有多个分支，各个分支相互独立，互补干扰

通过git init命令会默认创建一个master分支

```
#查看分支  
git branch  
git branch -r #列出所有远程远程仓库的分支  
git branch -a #列出所有远程、本地仓库的分支  
  
#创建分支  
git branch [branch name]  
  
#切换分支  
git checkout [branch name]  
  
#推送本地分支至远程仓库分支  
git push [short name][branch name]  
  
#合并分支  
git merge [branch name]
```

# 附录2 Linux系统的使用

## Linux常用命令

### Linux最常使用基本的命令

序号	命令	全名	作用
1	ls	list	查看当前目录下的内容
2	pwd	print work directory	查看当前所在目录的路径
3	cd [目录名]	change directory	切换目录
4	touch [文件名]	touch	如果文件不存在，则新建文件

序号	命令	全名	作用
5	mkdir [目录名]	make directory	创建目录
6	rm [文件名]	remove	删除指定文件

```
echo 'LANG = "en_US.UTF-8" ' >> /etc/profile
source /etc/profile
```

解决乱码问题

## 拷贝移动命令

```
#用于复制文件或目录
cp [-r] source dest
# 参数说明
#-r 如果复制的是目录，需要使用当前选项，将复制该目录下的所有子目录和文件
#命令作用：可以复制文件，也可以给文件改名
```

```
#为文件或目录改名、或将文件或目录移动到其它位置
mv source dest
```

## 打包压缩命令

```
#对文件进行打包，解包，压缩，解压
tar [-zcvf] fileName [files]
#Linux常见的有两种后缀：
#.tar 表示只是完成了打包，并没有进行压缩
#.tar.gz 表示打包的同时进行了压缩

#选项说明：
#-z 代表gzip，通过gzip命令处理文件，可以对文件进行压缩或者解压
#-c 代表create，即创建新的包文件
#-x 代表extract，实现从包文件中还原文件
#-v 代表的是verbose，显示命令的执行过程
#-f 代表file，用于指定包文件的名称
```

## 查找命令

```
#在指定目录下查找文件  
find dirName -option filename  
  
find . -name "*.java" #在当前目录下找以java为后缀的文件  
  
#从指定的文件中查找指定的文本内容  
grep word filename  
  
grep hello HelloWorld.java #在HelloWorld.java文件中查找"hello"字符串
```

## 防火墙相关命令

```
#查看防火墙状态  
systemctl status firewalld  
  
#暂时关闭防火墙  
systemctl stop firewalld  
  
#永久关闭防火墙  
systemctl disable firewalld  
  
#开启防火墙  
systemctl start firewalld  
  
#开放指定端口  
firewall-cmd --zone=public --add-port=8080/tcp --permanent  
  
#关闭指定端口  
firewall-cmd --zone=public --remove-port=8080/tcp --permanent  
  
#查看开放的端口  
firewall-cmd --zone=public --list-port
```

## Redis设置密码

使用后台启动的方式启动redis

```
redis-server redis.conf
#连接redis
redis-cli

#查看redis原始密码， 默认是没有密码的
config get requirepass

#设置redis连接密码
config set requirepass "password"

redis-cli
127.0.0.1:6379> auth password #密码可以不加引号
```

## 附录3 项目部署

### 手工部署

第一步：在IDEA中开发springboot项目并打成jar包，并将jar包上传到Linux服务器

第二步：在linux上使用java -jar命令运行jar包（Linux必须先安装jdk环境）

按照上述的部署过程其实已经可以部署一个项目，但会有一定的问题，就是springboot会独占一个linux连接

第三步：改为后台运行springboot程序，并将日志输出到日志文件

- 线上程序不会采用控制台霸屏的形式运行程序，而是将程序放入后台运行
- 线上程序不会讲日志输出到控制台，而是输出到指定的日志文件

可以使用nohup命令，全称：no hang up，用于不挂断地运行指定命令，退出终端不会影响指定命令的运行

```
nohup command [args...][&]
#command: 待执行的命令
#args: 一些命令参数，比如说日志输出的目标文件
#&: 让命令在后台运行

#示例：后台运行java -jar命令，并将日志文件输出到hello.log文件中
nohup java -jar xxx.jar >hello.log &
```

用这种方法如果想要停止springboot程序，可以使用kill命令

# shell自动部署部署

第一步：在Linux安装git

```
yum install git
```

第二步：在Linux安装maven

```
#将主机中的maven压缩包上传到linux中并解压  
tar -zxvf apache-maven-3.6.3-bin.tar.gz  
#添加maven的环境变量（在原有java的环境变量上添加即可）  
MAVEN_HOME=/usr/local/maven/apache-maven-3.6.3  
PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin:$MAVEN_HOME/bin  
export JAVA_HOME JRE_HOME CLASS_PATH MAVEN_HOME PATH  
#在/usr/local下创建repo目录作为maven的本地仓库，并配置  
vim /usr/local/maven/apache-maven-3.6.3/conf/settings.xml  
#添加  
<localRepository>/usr/local/repo</localRepository>
```

第三步：编写shell脚本（拉取代码、编译、打包、启动）

第四步：为用户授予执行shell脚本的权限

第五步：执行shell脚本

## 附录4 Spring Cache

### Spring Cache简介

Spring Cache是一个框架，实现了基于注解的缓存功能，只需要简单地加上一个注解就可以实现缓存的功能

Spring Cache提供了一层抽象，底层可以切换不同的Cache实现，具体就是通过CacheManager接口来统一不同的缓存技术

CacheManager	描述
EhCacheCacheManager	使用EhCache作为缓存技术
GuavaCacheManager	使用Google的GuavaCache作为缓存技术
RedisCacheManager	使用redis作为缓存技术

# Spring Cache常用注解

注解	说明
@EnableCaching	开启缓存注解功能
@Cacheable	在方法执行前spring先查看缓存中是否有数据，如果有数据，则直接返回缓存数据；若没有数据，调用方法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

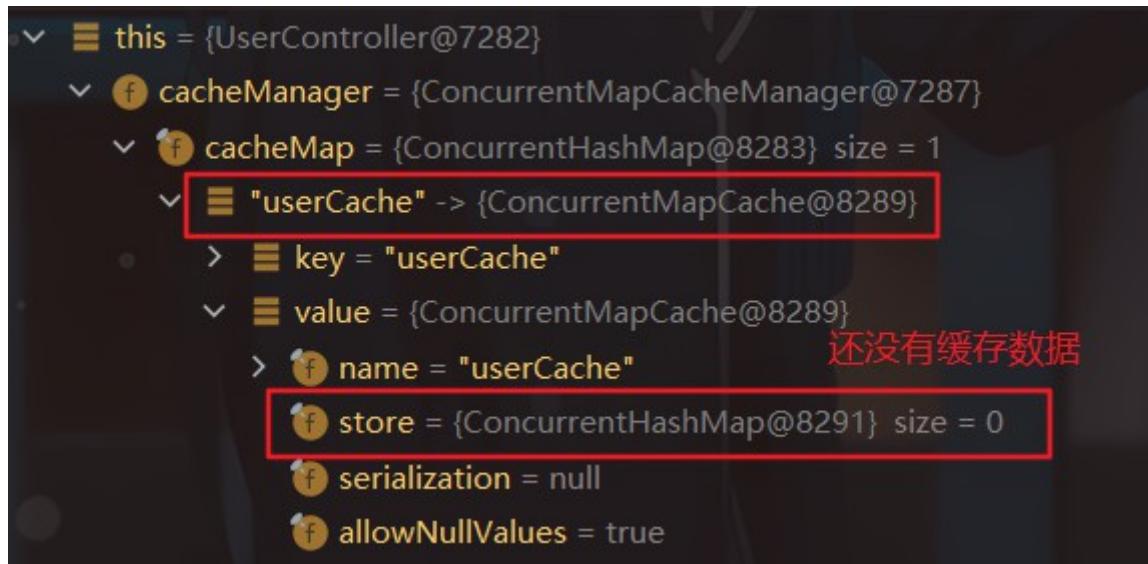
在springboot项目中，使用缓存技术只需要在项目中导入相关缓存技术的依赖包，并在启动类上使用@EnableCaching开启缓存支持即可

## Demo演示

### @CachePut注解

```
/*
 * @CachePut注解:
 *   value: 缓存的名称, 每个缓存名称下面可以有多个key
 *   key: 缓存的key
 */
@CachePut(value = "userCache", key = "#user.id")
@PostMapping
public User save(User user){
    userService.save(user);
    return user;
}
```

使用postman第一次发送请求，查看缓存数据



等再次发送请求的时候



注意：这个SpringCache默认的缓存是内存级缓存，当服务停掉以后就会被自动消除

## @CacheEvict

```
/*
 * @CacheEvict: 清理指定缓存
 *   value: 缓存的名称，每个缓存名称下面可以有多个key
 *   key: 缓存的key
 */
@CacheEvict(value = "userCache", key = "#id")
@DeleteMapping("/{id}")
public void delete(@PathVariable Long id){
    userService.removeById(id);
}
```

当方法执行完毕的时候，会自动清理缓存中的数据，一般情况下，当对数据库进行增删改操作时，都需要对缓存数据进行清理

## @Cacheable

```

/*
 * @Cacheable: 在方法执行之前, spring会先去查询缓存, 如果缓存中有数据则直接返回, 如果缓存中没有数据, 则执行方法
 *   value: 缓存的名称, 每个缓存名称下面可以有多个key
 *   key: 缓存的key
 *   condition: 当满足某个条件的时候, 缓存数据
 *   unless: 当不满足某个条件的时候, 去缓存数据
 */
@Cacheable(value = "userCache", key = "#id", condition = "#result != null")
@GetMapping("/{id}")
public User getById(@PathVariable Long id){
    User user = userService.getById(id);
    return user;
}

```

如果从数据库中没有查询到数据, 也会进行缓存, 可以根据需要设置缓存的条件

## 附录5 Nginx的使用

### Nginx概述

Nginx是一款轻量级的Web服务器 / 反向代理服务器及电子邮件 (IMAP/POP3) 代理服务器, 其特点是占有内存少, 并发能力强

官网: <https://nginx.org/>

#### nginx的下载

在官网上下载安装包, 传到linux服务器上

在linux的/usr/local目录下创建nginx目录, 按照以下步骤操作:

```

> mkdir nginx
> cd nginx
nginx> yum install pcre-devel zlib-devel openssl-devel
nginx> cd nginx-1.22.0
nginx-1.22.0> ./configure --prefix=/usr/local/nginx#指定安装目录, 执行此命令之后,
系统会检查是否支持安装, 如果报错就把系统没有的东西下载过去即可
nginx-1.22.0> make && make install #编译并安装
#安装完成之后跳转到/usr/local/nginx目录下
#conf: 存放配置文件
#html: 存放静态文件
#logs: 存放日志文件
#sbin: 存放nginx的脚本文件

```

一些重要的文件:

- conf/nginx.conf nginx的配置文件

- html 存放静态文件 (html, css, js等)
- logs 日志目录, 存放日志文件
- sbin/nginx nginx二进制文件, 用于启动、停止nginx服务

## Nginx常用命令

```
#在nginx/sbin目录下  
.nginx -v #查看nginx的版本信息  
  
.nginx -t #检查配置文件是否有误  
  
.nginx #启动nginx  
  
.nginx -s stop #停止nginx服务  
  
.nginx -s reload #重新加载配置文件
```

## Nginx配置文件结构

Nginx的配置文件整体分为三部分：

- 全局块：和nginx运行相关的全局配置
- events块：和网络连接相关的配置
- http块：代理，缓存，日志记录，虚拟主机配置 是nginx配置文件中最重要的一块
  - http全局块
  - server块

一个http块中可以配置多个server块，每个server块中可以配置多个location块

```
#=====全局块=====  
worker_processes 1;  
  
#=====event块=====  
events {  
    worker_connections 1024;  
}  
#=====http块=====  
  
http {  
    include mime.types;
```

```
default_type application/octet-stream;
sendfile on;
keepalive_timeout 65;

server {
    listen 80;      #监听80端口
    server_name localhost;      #指定服务器的名称，一般指服务器的域名或者ip

    location / {    #匹配客户端请求url
        root html;      #指定静态资源根目录
        index index.html index.htm;  #指定默认首页
    }

    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root html;
    }
}

}

#=====
```

# Nginx具体应用

## 部署静态资源

nginx可以作为静态web服务器来部署静态资源，静态资源指在服务端真实存在并且能够直接展示的一些文件，比如常见的html页面，css文件，js文件，图片等资源，相对于Tomcat，Nginx处理静态资源的能力更加高效，所以在生产环境下，一般都会将资源部署到nginx中  
部署的过程非常简单，只需要将静态资源文件复制到nginx安装目录下的html目录中即可  
然后修改配置文件中http块内server块的内容，匹配相应的路径

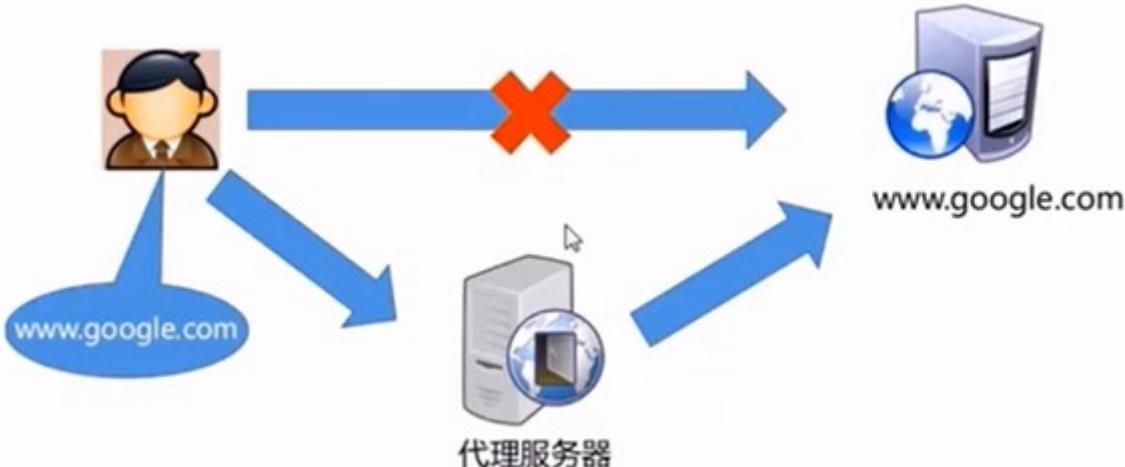
## 反向代理

正向代理：

是一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标（原始服务器），然后代理向原始服务器转交请求并将获得的内容返回给客户端

正向代理的典型用途是为在防火墙内的局域网客户端提供访问Internet的途径（VPN）

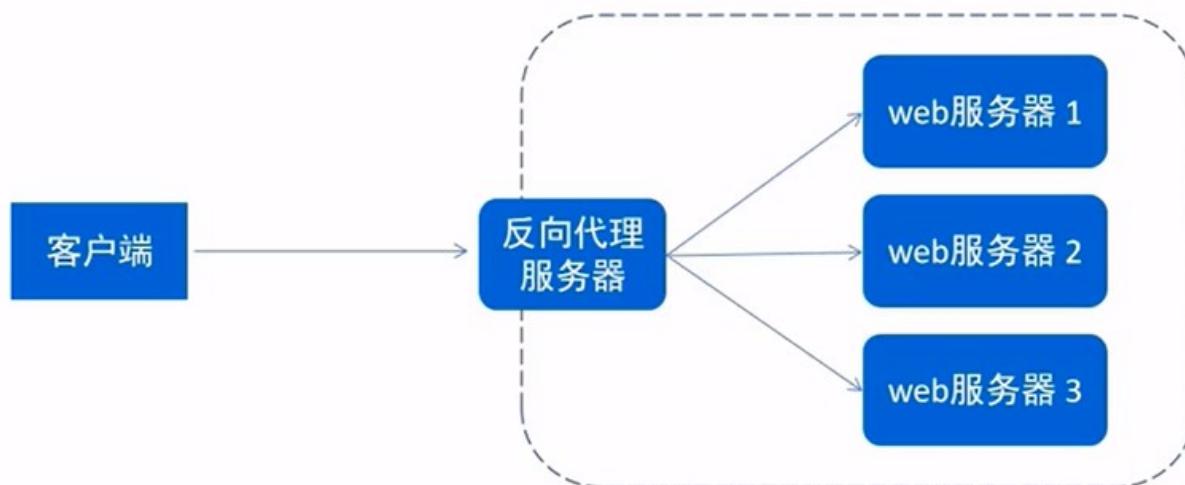
正向代理一般在客户端设置代理服务器，通过代理服务器转发请求，最终访问到目标服务器



## 反向代理

反向代理服务器位于用户与目标服务器之间，但是对于用户而言，反向代理服务器就相当于目标服务器，即用户直接访问反向代理服务器就可以获得目标服务器资源，反向代理服务器负责将请求转发给目标服务器

用户不需要知道目标服务器的确切地址，也无须在用户端做任何设定



## 反向代理的配置

## 负载均衡

早期的网站流量和业务功能都比较简单，单台服务器就可以满足基本需求，但是随着互联网的发展，业务流量越来越大并且业务逻辑也越来越复杂，单台服务器的性能及单点故障的问题就凸显出来了，因此需要多台服务器组成应用集群，进行性能的水平扩展以及避免单点故障出现

- 应用集群：将同一应用部署到多台服务器上，组成应用集群，接收负载均衡器分发的请求，进行业务处理并返回响应数据
- 负载均衡器：将用户请求根据对应负载均衡算法分发到应用集群中的一台服务器进行处理



## 负载均衡的配置

```
#在http块中添加配置
http {
    include      mime.types;
    default_type application/octet-stream;
    sendfile     on;
    keepalive_timeout  65;

    upstream target{
        server 192.168.138.101:8080
        server 192.168.138.101:8081
        ...   #可以有多台服务器
    }

    server{
        listen  8080;
        server_name localhost;
        location / {
            proxy_pass http://target;    #名字与upstream定义的名字对应
        }
    }

    server {
        listen      80;    #监听80端口
        server_name localhost;      #指定服务器的名称，一般指服务器的域名或者ip

        location / {    #匹配客户端请求url
            root  html;    #指定静态资源根目录
            index index.html index.htm;  #指定默认首页
        }

        error_page  500 502 503 504  /50x.html;
        location = /50x.html {
            root  html;
        }
    }
}
```

}

## nginx的负载均衡策略

名称	说明
round-robin 轮询	默认方式
weight 权重	权重方式
ip_hash	依据ip分配方式
least_conn	依据最少连接方式
url_hash	依据url分配方式
fair	依据响应时间方式