**外卖系统软件架构文档**

目录

**[外卖系统软件架构文档 1](#_Toc28971)**

**[1. 引言 （李雨桐） 2](#_Toc19996)**

[1.1 标识 2](#_Toc25629)

[标识号：A0001 3](#_Toc13517)

[标题：外卖系统（Online Food Delivery System） 3](#_Toc32535)

[缩略词语：OFDS（Online Food Delivery System） 3](#_Toc19933)

[版本号： 2024.1.0.0 3](#_Toc28811)

[发行号： 2024 3](#_Toc21425)

[1.2 系统概述 3](#_Toc8223)

[1.3文档概述 3](#_Toc4829)

[1.4 适用范围 4](#_Toc17788)

**[2. 总体架构 （李雨桐） 4](#_Toc5192)**

[2.1 概述 4](#_Toc7771)

[2.2 架构图 4](#_Toc23913)

[2.3 主要特性 6](#_Toc1626)

**[3. 前端架构 （李雨桐） 8](#_Toc8037)**

[3.1 技术栈 8](#_Toc1567)

[3.2 架构模式 8](#_Toc13329)

[3.3 页面结构 8](#_Toc23753)

**[4. 后端架构 （张勇） 11](#_Toc46)**

[4.1 技术栈 11](#_Toc24005)

[4.2 架构模式 12](#_Toc19544)

[4.3 主要模块 12](#_Toc3719)

**[5. 数据库设计 （张勇） 13](#_Toc29301)**

[5.1 数据库类型 13](#_Toc2733)

[5.2 数据库结构 13](#_Toc27568)

**[• username: String 13](#_Toc10902)**

**[6. 部署架构 （丁弘扬） 15](#_Toc27461)**

[6.1 环境 15](#_Toc1921)

[6.2 部署方式 19](#_Toc21669)

**[1.准备服务器 21](#_Toc4510)**

**[2. 配置服务器安全 21](#_Toc4026)**

[防火墙：设置防火墙规则以限制对服务器的访问。 21](#_Toc20414)

**[3. 部署Node.js应用 21](#_Toc5208)**

[代码版本控制：使用Git来管理代码。 21](#_Toc23949)

**[4. 部署MongoDB 22](#_Toc3606)**

**[5. 配置网络连接 22](#_Toc9328)**

**[6. 监控和日志记录 22](#_Toc17455)**

[系统监控：使用工具来监控服务器的CPU、内存和磁盘使用情况。 22](#_Toc27776)

**[7.备份和恢复策略 23](#_Toc3346)**

**[7. 安全性考虑 （丁弘扬） 23](#_Toc3645)**

[7.1 身份验证与授权 23](#_Toc5067)

**[1.设计用户权限系统 26](#_Toc4838)**

[用户角色：定义不同的用户角色，如管理员、普通用户、访客等。 26](#_Toc9586)

**[2.认证机制 26](#_Toc16548)**

**[7. 权限验证 26](#_Toc12090)**

[请求拦截：在服务器端，对所有敏感接口请求进行拦截 26](#_Toc14562)

**[8. 权限控制策略 26](#_Toc6620)**

[白名单策略：只允许具有特定权限的用户访问敏感接口。 26](#_Toc17031)

**[9. 日志和监控 27](#_Toc21870)**

[记录请求日志：记录所有敏感接口请求的详细信息，包括请求者、请求时间、请求内容等。 27](#_Toc26354)

**[6. 安全性考虑 27](#_Toc16618)**

[最小权限原则：为用户分配最小的必要权限，以减少潜在的安全风险。 27](#_Toc32743)

**[7. 客户端和API文档 27](#_Toc13038)**

[明确接口权限：在API文档中明确说明每个接口所需的权限级别。 27](#_Toc1487)

[7.2 数据安全 27](#_Toc21886)

**[8. 性能优化 （丁弘扬） 28](#_Toc10480)**

[8.1 前端性能优化 28](#_Toc21708)

[8.2 后端性能优化 30](#_Toc21908)

**[9. 扩展性考虑 （王攀） 33](#_Toc8534)**

[9.1 前端扩展性 33](#_Toc565)

[9.2 后端扩展性 35](#_Toc18391)

**[10. 运维策略 （王攀） 38](#_Toc24609)**

[10.1 监控与日志 38](#_Toc13792)

[10.2 自动化部署 41](#_Toc2620)

**[11. 维护策略 （刘易） 43](#_Toc14353)**

[11.1 定期更新 43](#_Toc27330)

[11.2 Bug修复 44](#_Toc10063)

**[12. 风险管理 （刘易） 45](#_Toc6205)**

[12.1 高并发 45](#_Toc9316)

[12.2 安全漏洞 46](#_Toc11020)

1. **引言 （李雨桐）**

1.1 标识

## 标识号：A0001

## 标题：外卖系统（Online Food Delivery System）

## 缩略词语：OFDS（Online Food Delivery System）

## 版本号： 2024.1.0.0

## 发行号： 2024

1.2 系统概述

本项目（外卖系统）是专门为餐饮企业（餐厅、饭店）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。其中系统管理后台主要提供给餐饮企业内部员工使用，可以对餐厅的分类、菜品、套餐、订单、员工等进行管理维护。移动端应用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。

与外卖系统相关的其他文档包括：

可行性分析报告(FAR): 评估项目可行性。

需求规格说明书（SRS）：描述系统的功能需求和性能需求。

软件设计文档（SDD）：描述系统的软件设计和架构。

软件测试计划（STP）：描述系统的测试策略和测试计划。

## 1.3文档概述

软件结构设计说明（SDD）是一份用于描述软件系统结构和组成部分的文档，其主要用途是为软件开发人员和其他相关人员提供一个清晰的、可理解的软件结构设计方案，以便于实现和维护软件系统。

该SDD的内容包括以下几个方面：

软件系统结构：包括软件系统的组成部分、模块之间的关系和接口等。

数据结构设计：包括软件系统中使用的数据结构和数据类型。

CSCI设计等。

与SDD使用有关的保密性或私密性要求可能包括以下几个方面：

保护知识产权：SDD中包含了软件系统的设计方案和实现细节，很可能涉及到知识产权的保护。因此，在使用SDD时需要注意保护相关的知识产权，避免泄露。

保护商业机密：SDD中可能包含了商业机密信息，如软件系统的商业模式、市场定位等。在使用SDD时需要注意保护这些商业机密信息，避免被竞争对手获取。

1.4 适用范围

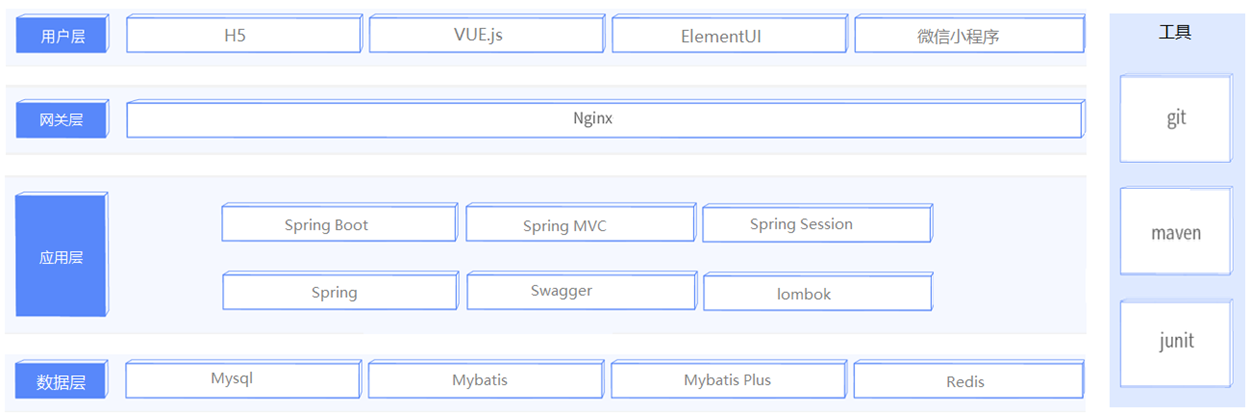
本文档适用于外卖系统的软件架构，覆盖了系统的核心功能和组件。

**2. 总体架构 （李雨桐）**

2.1 概述

关于本项目的技术选型, 我们将会从 用户层、网关层、应用层、数据层 这几个方面进行介绍，而对于我们服务端开发工程师来说，在项目开发过程中，我们主要关注应用层及数据层技术的应用。

2.2 架构图



**1）用户层** 本项目中在构建系统管理后台的前端页面，我们会用到H5、Vue.js、ElementUI等技术。而在构建移动端应用时，我们会使用到微信小程序。

**2）网关层** Nginx是一个服务器，主要用来作为Http服务器，部署静态资源，访问性能高。在Nginx中还有两个比较重要的作用： 反向代理和负载均衡， 在进行项目部署时，要实现Tomcat的负载均衡，就可以通过Nginx来实现。

**3）应用层** SpringBoot： 快速构建Spring项目, 采用 "约定优于配置" 的思想, 简化Spring项目的配置开发。 Spring: 统一管理项目中的各种资源(bean), 在web开发的各层中都会用到。 SpringMVC：SpringMVC是spring框架的一个模块，springmvc和spring无需通过中间整合层进行整合，可以无缝集成。 SpringSession: 主要解决在集群环境下的Session共享问题。 lombok：能以简单的注解形式来简化java代码，提高开发人员的开发效率。例如开发中经常需要写的javabean，都需要花时间去添加相应的getter/setter，也许还要去写构造器、equals等方法。 Swagger： 可以自动的帮助开发人员生成接口文档，并对接口进行测试。

**4）. 数据层** MySQL： 关系型数据库, 本项目的核心业务数据都会采用MySQL进行存储。 MybatisPlus： 本项目持久层将会使用MybatisPlus来简化开发, 基本的单表增删改查直接调用框架提供的方法即可。 Redis： 基于key-value格式存储的内存数据库, 访问速度快, 经常使用它做缓存(降低数据库访问压力, 提供访问效率), 在后面的性能优化中会使用。

**5). 工具** git: 版本控制工具, 在团队协作中, 使用该工具对项目中的代码进行管理。 maven: 项目构建工具。 junit：单元测试工具，开发人员功能实现完毕后，需要通过junit对功能进行单元测试。

### **功能架构图**

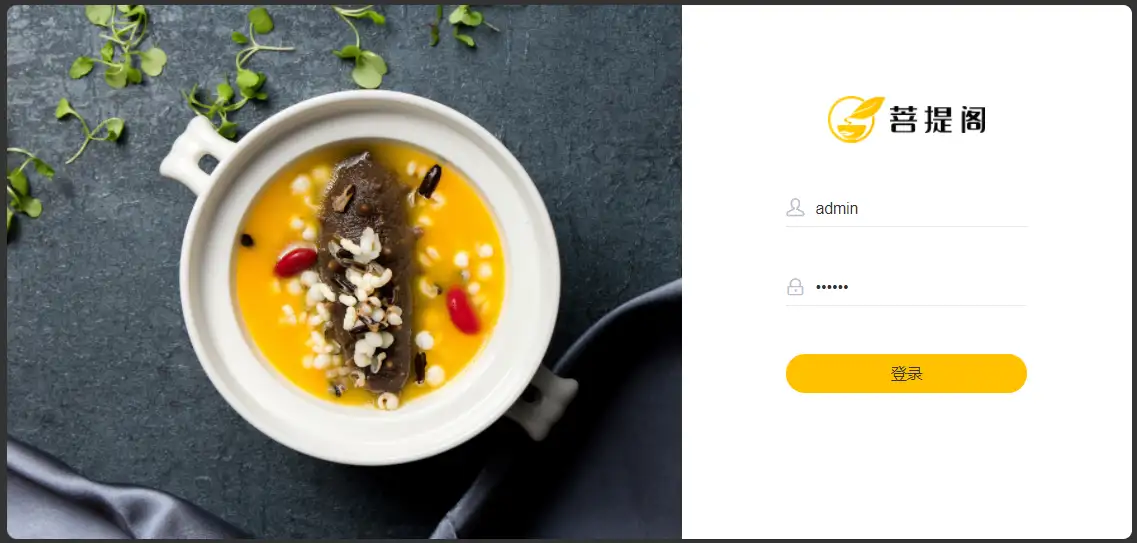


1. **. 移动端前台功能** 手机号登录 , 微信登录 , 收件人地址管理 , 用户历史订单查询 , 菜品规格查询 , 购物车功能 , 下单 , 分类及菜品浏览。

**2). 系统管理后台功能** 员工登录/退出 , 员工信息管理 , 分类管理 , 菜品管理 , 套餐管理 , 菜品口味管理 , 订单管理 。

2.3 主要特性

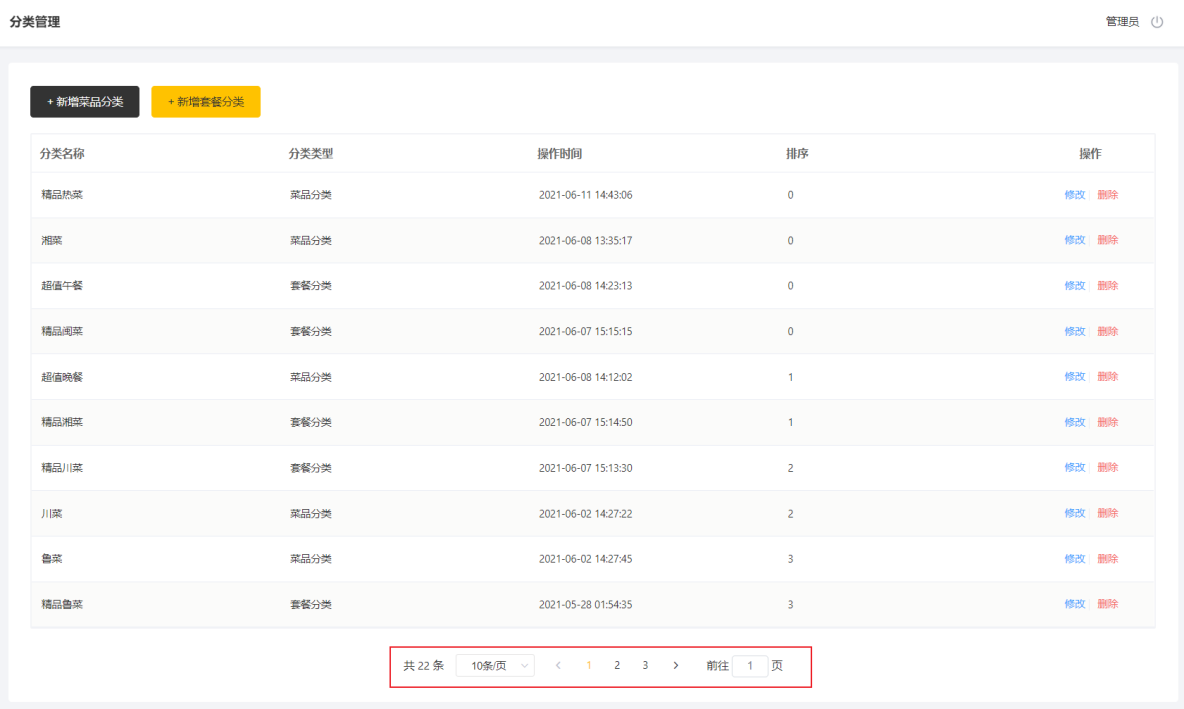
（1）用户注册与登录：用户可以注册新账号并登录系统，也可以使用社交账号进行快速登录。



（2）商家管理：商家可以注册入驻系统并管理自己的店铺信息、菜单和订单。



（3）菜单浏览与下单：用户可以浏览商家提供的菜单，查看菜品详情，并将菜品添加到购物车进行下单。



（4）订单处理与配送跟踪：系统支持用户下单后的订单处理流程，包括商家接单、配送员接单、订单配送和配送跟踪。



（5）评价与反馈：用户可以对完成的订单进行评价和反馈，提供意见和建议，帮助商家改进服务质量。

**3. 前端架构 （李雨桐）**

3.1 技术栈

（1）前端框架：Vue.js

（2）状态管理：Vuex

（3）UI 组件库：Element UI

（4）HTTP 请求库：axios

3.2 架构模式

前端采用了单页面应用（SPA）架构，通过组件化开发实现页面的模块化和复用。

3.3 页面结构

（1）**登录页：用户和商家登录入口**

登录页是外卖系统的起点，为用户和商家提供了安全、快捷的登录入口。在登录页上，用户可以通过输入手机号、邮箱或用户名以及对应的密码来登录自己的账号。对于商家而言，登录页同样提供了专门的商家登录入口，确保商家能够方便地管理自己的店铺和订单。此外，为了提升用户体验和安全性，登录页还采用了验证码机制，防止恶意登录和机器人攻击。

（2）**注册页：新用户和商家注册页面**

注册页是外卖系统吸引新用户和商家加入的重要页面。新用户可以通过填写手机号、验证码、密码等信息来注册账号，并设置自己的昵称、头像等个性化信息。商家在注册时，除了填写基本的账号信息外，还需要提供店铺名称、地址、联系方式等详细信息，以便后续进行店铺管理和订单处理。注册页还提供了友好的提示和错误反馈机制，帮助用户顺利完成注册流程。

（3）**商家列表页：展示所有可供选择的商家列表**

商家列表页是用户浏览和选择商家的主要页面。在商家列表页上，系统会根据用户的地理位置和搜索条件，展示附近或符合要求的商家列表。每个商家都会以卡片或列表的形式展示，包括商家名称、评分、距离、优惠活动等信息。用户可以通过点击商家卡片或列表项，进入商家的详情页面，进一步了解商家的菜品、评价等信息。商家列表页还提供了筛选和排序功能，帮助用户快速找到符合自己需求的商家。

（4）**菜单页：展示商家的菜单和菜品详情**

菜单页是用户了解商家菜品和下单的关键页面。在菜单页上，系统会展示商家的所有菜品，并按照分类或排序方式呈现给用户。每个菜品都会以图片和文字的形式展示，包括菜品名称、价格、口味、配料等信息。用户可以通过点击菜品图片或名称，查看菜品的详情和介绍。同时，菜单页还提供了搜索和筛选功能，方便用户快速找到自己想要的菜品。用户可以在菜单页上直接下单购买菜品，享受美食的乐趣。

（5）**购物车页：用户可以查看购物车中的商品，并进行编辑**

购物车页是用户管理购买商品的重要页面。在购物车页上，系统会展示用户已经加入购物车的所有商品，包括商品名称、价格、数量等信息。用户可以随时查看购物车中的商品，并进行编辑操作，如修改商品数量、删除商品等。购物车页还提供了结算功能，用户可以选择支付方式并完成订单支付。同时，购物车页还会根据用户的购买行为和偏好，推荐相关的商品给用户，提升用户的购物体验和满意度。

（6）**订单页：展示用户的订单列表和订单详情**

订单页是用户查看和管理自己订单的主要页面。在订单页上，系统会展示用户的所有订单列表，包括待支付、待发货、待收货、已完成等状态的订单。用户可以随时查看订单的详情和状态，了解订单的处理进度和物流信息。同时，订单页还提供了订单管理功能，如取消订单、申请退款等，方便用户处理订单问题。通过订单页，用户可以清晰地掌握自己的订单情况，确保订单能够按时、准确地送达。

（7）**评价页：用户可以对订单进行评价和反馈**

评价页是用户对外卖服务进行评价和反馈的重要渠道。在评价页上，用户可以对已完成的订单进行评价，包括对菜品口味、配送速度、服务态度等方面的评价。用户可以选择给商家打分、留下文字评价或上传图片等方式来表达自己的意见和感受。评价页还提供了筛选和排序功能，帮助用户快速找到需要评价的订单。通过用户的评价和反馈，商家可以及时了解自己的服务质量和不足之处，并采取相应的措施进行改进和提升。同时，用户的评价也可以为其他用户提供参考和借鉴，帮助他们做出更好的选择。

**4. 后端架构 （张勇）**

4.1 技术栈

（1）后端框架：Node.js + Express

• Node.js：基于Chrome V8引擎的JavaScript运行环境，适合构建高性能、可扩展的网络应用。

• Express：基于Node.js平台的轻量级web开发框架，提供了丰富的HTTP工具。

（2）数据库：MongoDB

• MongoDB：一个基于分布式文件存储的数据库，为web应用提供可扩展的高性能数据存储解决方案，其灵活性高、易于扩展的特性非常适合处理复杂的、半结构化的数据。

（3）身份验证：JWT（JSON Web Tokens）

• JWT：一种开放标准（RFC 7519）的方法，用于在双方之间安全地传输信息。这些信息可以被验证和信任，因为它是数字签名的。

4.2 架构模式

后端采用了RESTful API架构，通过HTTP协议提供服务。RESTful API使用HTTP方法（如GET、POST、PUT、DELETE等）来操作资源，并且具有明确的URL路径来标识这些资源。

4.3 主要模块

（1）用户管理模块：处理用户的注册、登录和个人信息管理。

• 功能包括：用户注册时的数据验证和存储。包括用户登录时的身份验证和会话管理，以及用户个人信息的更新和查询。还应考虑安全性措施，如密码强度校验、验证码防机器人注册等。

（2）商家管理模块：处理商家的注册、登录和店铺信息管理。

• 功能包括：商家注册时的资质验证和店铺信息存储，商家登录时的身份验证和店铺信息管理，如添加/修改菜品、营业时间等。

（3）订单管理模块：处理订单的创建、处理和配送流程。

• 功能包括：订单生成、订单状态更新（如待支付、已支付、待配送、已配送等）、订单查询和订单统计等。

（4）菜单管理模块：管理商家的菜单信息和菜品详情。

• 功能包括：菜单的添加、修改、删除和查询，菜品信息的更新和查询，以及菜品图片的上传和下载。

（5）支付管理模块：集成支付接口，处理订单的支付流程。

• 功能包括：支付接口的集成和配置，支付请求的生成和发送，支付结果的接收和处理，以及支付状态的更新和查询。

**5. 数据库设计 （张勇）**

5.1 数据库类型

外卖系统使用了非关系型数据库MongoDB，适用于大量数据的存储和高性能的读写操作。

5.2 数据库结构

（1）用户信息表（User）：存储用户的账号信息和个人资料，包括用户名、密码、邮箱、联系方式等字段。

• \_id: ObjectId

• username: String

• password: String (hashed)

• email: String

• phone: String

• address: String

• created\_at: Date

• updated\_at: Date

（2）商家信息表（Merchant）：存储商家的账号信息和店铺资料，包括商家名称、店铺地址、联系方式等字段。

• \_id: ObjectId

• username: String

• password: String (hashed)

• email: String

• phone: String

• address: String

• store\_name: String

• created\_at: Date

• updated\_at: Date

（3）菜单信息表（Menu）：存储商家的菜单信息和菜品详情，包括菜品名称、价格、描述、所属商家等字段。

• \_id: ObjectId

• merchant\_id: ObjectId (关联商家信息表)

• category: String

• name: String

• description: String

• price: Number

• image\_url: String

• created\_at: Date

• updated\_at: Date

（4）订单信息表（Order）：存储订单的相关信息，包括用户、商家、商品和配送信息，包括订单号、用户ID、商家ID、商品详情、配送地址、订单状态等字段。

• \_id: ObjectId

• user\_id: ObjectId (关联用户信息表)

• merchant\_id: ObjectId (关联商家信息表)

• items: Array of Objects

• menu\_id: ObjectId (关联菜单信息表)

• quantity: Number

• total\_price: Number

• status: String (订单状态，如"待处理"、"配送中"、"已完成")

• delivery\_address: String

• created\_at: Date

• updated\_at: Date

**6. 部署架构 （丁弘扬）**

6.1 环境

（1）开发环境：本地开发机

**选择操作系统**：根据项目需求，选择适合的操作系统。

**安装必要的开发工具**：**代码编辑器/IDE**：根据编程语言和项目类型选择。

**版本控制系统**：Git，用于代码管理和协作。

**数据库管理系统**：MySQL、MongoDB等，根据项目需求选择。

**终端/命令行工具**：Windows的PowerShell或CMD。

**配置开发环境**：**安装编程语言**：Java。

**安装依赖管理工具**：npm（Node.js），Maven（Java）等，用于管理项目依赖。

**配置环境变量**：设置PATH、JAVA\_HOME等环境变量，以便系统可以找到安装的工具和库。

**安装项目依赖**：进入项目目录，使用项目的依赖管理工具安装项目所需的依赖。

**配置本地服务器**：对于Web开发，需要设置一个本地服务器来预览你的网页或API。使用Nginx、Node.js等工具来设置。

**测试和调试**：使用测试框架编写和运行测试。使用调试工具来调试代码。

**备份和版本控制**：定期备份你的开发环境和项目代码，以防止数据丢失。用版本控制系统（如Git）来跟踪和管理你的代码更改。

**文档和注释**：编写和维护清晰的文档和代码注释，以便其他开发者可以理解代码和工作。

**保持更新**：定期检查并更新你的操作系统、开发工具、编程语言和库，以确保你的开发环境保持最新和安全。

（2）测试环境：单独测试服务器

**硬件和云资源选择**：根据测试需求选择合适的硬件或云服务器资源。测试环境可能不需要与生产环境相同的硬件配置，但应确保资源足够支持测试活动。如果使用云服务，选择适当的实例类型和配置，以平衡性能和成本。

**操作系统和软件安装**：在测试服务器上安装与生产环境相同的操作系统版本和补丁。安装所有必要的软件，包括数据库、中间件、开发工具等，确保版本与生产环境一致。

**网络配置**：配置测试服务器的网络设置，确保它能够访问必要的外部资源（如代码仓库、测试数据等）。如果可能，将测试服务器放在与生产环境相似的网络环境中，以模拟实际的网络延迟和带宽。

**数据和环境配置**：准备测试数据，确保数据的完整性和准确性。可以使用生产数据的子集或模拟数据。配置测试环境以模拟生产环境的各种场景，包括用户数量、负载模式、并发请求等。

**备份和恢复策略**：虽然测试环境不像生产环境那样关键，但定期备份测试数据仍有必要。这有助于在测试过程中出现问题时快速恢复。

制定测试环境的恢复策略，确保在需要时可以快速重建测试环境。

**安全性和隔离性**：确保测试服务器具有适当的安全措施，如防火墙、入侵检测系统等。将测试服务器与生产环境隔离，以防止未经授权的访问和潜在的安全风险。

**版本控制和持续集成**：设置版本控制系统，用于管理测试环境的配置和代码更改。配置持续集成/持续部署流程，以便在代码更改时自动构建、测试和部署应用程序。

**监控和日志记录**：在测试服务器上安装监控工具，以跟踪服务器性能、网络流量、应用程序响应时间等指标。配置日志记录，以便在测试过程中捕获和诊断问题。

**访问权限管理**：限制对测试服务器的访问权限，仅允许必要的团队成员访问。使用强密码和定期更换密码的策略，确保账户安全。

**环境管理和文档**：建立测试环境管理文档，记录环境配置、软件版本、测试数据等信息。定期更新文档，以反映测试环境的任何更改。

**测试策略和方法**：制定明确的测试策略和方法，包括功能测试、性能测试、安全测试等。使用自动化测试工具来提高测试效率和准确性。

**测试执行和报告**：定期执行测试，并跟踪测试进度和结果。生成详细的测试报告，记录测试结果、问题和建议。

**问题跟踪和修复**：在测试过程中发现的问题应及时跟踪和修复。

使用问题跟踪系统来管理问题和修复进度。

（3）生产环境：云服务器

**选择云服务提供商**：选择一个可靠且符合你需求的云服务提供商。

**选择服务器类型和配置**：根据应用需求选择合适的云服务器实例类型。确定服务器的规格，包括CPU核心数、内存大小、存储类型和容量等。

**设置网络和安全性**：配置虚拟私有云（VPC）和子网，确保网络安全和隔离。设置安全组（Security Groups）或网络访问控制列表（Network ACLs），限制对服务器的访问。使用VPN或Direct Connect等私有网络连接方式，提高数据传输的安全性。配置防火墙和入侵检测系统（IDS/IPS），防止恶意攻击。

**操作系统和配置**：选择合适的操作系统。安装和配置必要的系统更新、补丁和安全设置。配置服务器时区、语言和其他基础设置。

**应用部署**：准备好应用代码和依赖项。使用自动化部署工具或容器化技术进行应用部署。监控应用性能和健康状况，确保应用的稳定运行。

**数据库和存储**：选择合适的数据库系统。配置数据库的备份、恢复和容灾策略。使用云存储服务存储静态文件和备份数据。

**负载均衡和伸缩性**：使用负载均衡器分发应用流量，提高应用的可用性和性能。配置自动伸缩组，根据应用负载自动调整服务器实例数量。

**日志和监控**：配置日志收集和分析系统，收集和分析应用和系统日志。使用云监控服务监控服务器和应用性能。设置告警和通知机制，及时发现和解决问题。

**备份和恢复**：定期对服务器和应用进行备份，确保数据的完整性和可恢复性。制定灾难恢复计划，测试备份和恢复流程的有效性。

**合规性和审计**：确保你的生产环境符合相关的法规和行业标准。定期进行安全审计和渗透测试，评估系统的安全性。

6.2 部署方式

（1）前端部署：静态资源部署到CDN，动态请求部署到Web服务器 **选择CDN服务提供商**：首先，选择一个适合需求的CDN服务提供商。在选择时，可以考虑服务的稳定性、可扩展性、成本以及是否支持特定需求。

**注册并配置CDN服务**：注册并登录选择的CDN服务提供商的管理控制台。添加你的域名，并进行相关的配置，如设置源服务器地址、缓存策略等。

**上传静态资源**：在CDN服务提供商的管理控制台中，找到对应的存储空间或资源管理界面。上传静态资源文件，如图片、CSS、JavaScript等。

**配置缓存策略**：根据你的需要配置CDN的缓存策略，包括缓存时间、缓存规则、缓存位置等。静态资源通常可以设置较长的缓存时间，以提高访问速度。

**配置加速域名**：在CDN服务提供商的管理控制台中，配置加速域名，将域名与CDN服务绑定。这样，当用户访问你的网站时，CDN会根据用户的地理位置和网络状况，将静态资源从最近的CDN节点提供给用户，从而加快访问速度。

**部署Web服务器**：对于动态请求，部署一个Web服务器来处理这些请求。在Web服务器上配置应用程序和数据库，确保它们能够处理动态请求并返回正确的响应。

**配置路由和负载均衡**：根据需要配置路由和负载均衡策略，确保静态请求被路由到CDN，而动态请求被路由到Web服务器。

（2）后端部署：Node.js应用部署到应用服务器，MongoDB部署到数据库服务器

### **1.准备服务器**

### **应用服务器**：选择一个适合运行Node.js的服务器，确保它已安装Node.js和npm（Node包管理器）。

### **数据库服务器**：选择一个性能强大的服务器来运行MongoDB。确保已安装MongoDB。

### **配置服务器安全**

### **防火墙**：设置防火墙规则以限制对服务器的访问。

### **SSH**：如果可能，仅允许通过SSH进行远程访问，并使用强密码或密钥认证。

### **更新和补丁**：定期更新服务器和所有软件到最新版本，以利用最新的安全补丁。

### **部署Node.js应用**

### **代码版本控制**：使用Git来管理代码。

### **传输代码**：通过SSH或SFTP将代码从本地计算机传输到应用服务器。

**依赖项管理**：在应用服务器上，使用npm安装所有必需的依赖项。

**环境变量**：配置任何必要的环境变量，如数据库连接字符串、密钥等。

**启动应用**：使用node命令来启动你的Node.js应用。

**进程管理**：考虑使用像PM2这样的进程管理器来确保应用始终在运行，并在崩溃时自动重启。

### **部署MongoDB**

### **配置MongoDB**：编辑MongoDB的配置文件，根据需要设置参数，如存储路径、端口、日志级别等。

**创建数据目录**：确保MongoDB的数据目录存在并具有适当的权限。

**启动MongoDB**：使用mongod命令启动MongoDB服务。

**创建数据库和用户**：根据应用需求，在MongoDB中创建必要的数据库和用户，并设置适当的权限。

**连接测试**：使用MongoDB shell或Node.js应用来测试与数据库的连接。

### **配置网络连接**

### **应用服务器到数据库服务器**：确保应用服务器可以访问数据库服务器上的MongoDB端口。

**外部访问**：确保相关的端口在应用服务器上已打开并正确配置。

### **监控和日志记录**

### **系统监控**：使用工具来监控服务器的CPU、内存和磁盘使用情况。

**应用监控**：使用如New Relic、Datadog或Prometheus等工具来监控Node.js应用的性能。

**日志记录**：确保Node.js应用和MongoDB都配置了适当的日志记录，以便在出现问题时可以快速诊断。

### **7.备份和恢复策略**

### **MongoDB备份**：定期备份MongoDB数据库，并确保备份文件存储在安全的位置。

**Node.js应用备份**：定期备份Node.js应用代码和配置文件。

**恢复测试**：定期测试备份恢复流程，以确保在需要时可以快速恢复数据或服务。

**7. 安全性考虑 （丁弘扬）**

7.1 身份验证与授权

（1）用户登录使用JWT生成的Token进行身份验证

当用户登录时，服务器可以生成一个 JWT，并将其发送回客户端。客户端可以在后续的请求中带上这个 JWT，以证明其身份或其他声明。

1. **用户登录**：用户向服务器发送登录请求，包含用户名和密码。服务器验证用户名和密码是否有效。
2. **生成 JWT**：如果用户名和密码验证成功，服务器会生成一个 JWT。JWT 通常由三部分组成：头部（Header）、载荷（Payload）和签名（Signature）。
   1. 头部通常包含令牌的类型（JWT）和所使用的签名算法（如 HMAC SHA256 或 RSA）。
   2. 载荷包含有关用户的信息（如用户ID、角色等）和声明。声明是有关实体（通常是用户）和其他数据的声明。
   3. 签名是通过对头部和载荷进行编码后的字符串进行签名得到的，用于验证 JWT 的完整性和真实性。
3. **发送 JWT**：服务器将生成的 JWT 发送回客户端，通常作为 HTTP 响应的一部分。
4. **客户端存储 JWT**：客户端在接收到 JWT 后，通常会将其存储在本地。
5. **客户端发送 JWT**：在后续的请求中，客户端会在请求头中带上 JWT（通常在 Authorization 头部中，格式为 Bearer <token>）。
6. **服务器验证 JWT**：当服务器收到带有 JWT 的请求时，它会首先验证 JWT 的签名是否有效，以确保 JWT 没有被篡改。
   1. 然后，服务器会解码 JWT 的载荷，并从中提取有关用户的信息和声明。
   2. 最后，服务器会基于这些信息和声明来授权或拒绝请求。
7. **JWT 过期**：JWT 可以设置过期时间（Expiration time），一旦 JWT 过期，客户端需要再次进行登录以获取新的 JWT。
8. **刷新令牌（Refresh Tokens）**：为了安全起见，会使用两种类型的令牌：访问令牌（Access Tokens，即 JWT）和刷新令牌（Refresh Tokens）。
   1. 访问令牌用于访问资源，但它们的过期时间较短（例如，几分钟到几小时）。
   2. 当访问令牌过期时，客户端可以使用刷新令牌向服务器请求一个新的访问令牌。刷新令牌的过期时间通常较长（例如，几天到几个月），并且只能在服务器端撤销。

（2）对敏感接口进行权限验证，只有授权用户才能访问

### **1.设计用户权限系统**

### **用户角色**：定义不同的用户角色，如管理员、普通用户、访客等。

### **权限级别**：为每个角色分配不同的权限级别，例如读取、写入、删除等。

### **权限表**：创建一个数据库表来存储用户与他们的角色和权限的关联。

### **2.认证机制**

### **用户认证**：在用户尝试访问敏感接口之前，确保用户已通过身份验证。这通常通过用户名和密码、JWT等方式实现。

### **会话管理**：一旦用户通过身份验证，服务器会为用户创建一个会话或令牌，并在客户端存储它。

### **权限验证**

### **请求拦截：**在服务器端，对所有敏感接口请求进行拦截

### **检查令牌：**验证请求中是否包含有效的令牌。

### **解析令牌：**如果令牌有效，解析令牌以获取用户信息（如用户ID）。

### **查询权限：**使用用户ID从权限表中查询用户的角色和权限。

### **权限比对：**比对用户请求的接口所需的权限与用户实际的权限，确保用户有足够的权限来执行该操作。

### **权限控制策略**

### **白名单策略**：只允许具有特定权限的用户访问敏感接口。

### **黑名单策略**：禁止具有特定权限的用户访问敏感接口。

### **基于角色的访问控制（RBAC）**：根据用户的角色来授予或拒绝访问权限。

### **日志和监控**

### **记录请求日志**：记录所有敏感接口请求的详细信息，包括请求者、请求时间、请求内容等。

### **监控异常行为**：监控并报警任何可疑或异常的行为，如未经授权的访问尝试、大量失败的登录尝试等。

### **安全性考虑**

### **令牌安全性**：确保令牌是安全的，使用HTTPS来传输令牌，并定期更换令牌以防止泄露。

### **最小权限原则**：为用户分配最小的必要权限，以减少潜在的安全风险。

### **定期审计**：定期审查权限表和访问日志，确保没有不必要的权限和异常行为。

### **使用安全框架和库**：利用现有的安全框架和库来简化权限验证和管理的实现。

### **客户端和API文档**

### **明确接口权限**：在API文档中明确说明每个接口所需的权限级别。

### **客户端验证**：在客户端也进行一定的权限验证，但请注意，客户端验证不能作为唯一的安全措施，因为客户端代码可以被绕过或篡改。

7.2 数据安全

（1）使用HTTPS加密传输敏感信息

使用HTTPS加密传输敏感信息确保了在互联网上进行数据传输时，数据的内容得到保护，防止被恶意用户窃取、篡改或伪造。HTTPS是基于HTTP协议的安全传输协议，它通过使用SSL（Secure Sockets Layer）或TLS（Transport Layer Security）协议对网络通信进行加密和身份验证，从而确保数据在传输过程中的安全性。

**8. 性能优化 （丁弘扬）**

8.1 前端性能优化

（1）使用CDN加速静态资源加载

CDN加速静态资源加载的原理是通过将网站的内容复制到离用户最近的CDN服务器上，使用户在访问时可以更快地获取网站内容。CDN服务商在全球部署了大量的服务器节点，这些节点可以根据用户的地理位置和网络条件来选择最优的节点来提供内容。当用户请求静态资源时，CDN将首先检查本地缓存中是否有这些资源。如果有，则直接从缓存中提供；如果没有，则从源站获取这些资源并缓存到本地，以供后续请求使用。

（2）图片懒加载和压缩

图片懒加载是一种延迟加载图片的技术，只在图片即将进入可视区域时才开始加载。与传统的立即加载所有图片的方式不同，懒加载可以提高页面的加载性能，尤其对于包含大量图片的页面更为有效。懒加载的实现方式有多种，其中一种常见的做法是在img标签中添加loading="lazy"属性，这适用于现代浏览器。另一种实现方式是通过JavaScript监测页面的滚动事件，当图片即将进入可视区域时再进行加载。

懒加载的作用主要有以下几点：

1. 减少初始页面加载时间：随着页面的增大，加载所有图片可能导致较长的初始加载时间。图片懒加载允许浏览器仅加载可视区域内的图片，从而加快了页面的首次加载。
2. 节省带宽和服务器资源：当页面中包含大量图片时，仅加载用户实际查看的部分可以减少不必要的带宽消耗和服务器资源。
3. 优化用户体验：用户在滚动页面时，仅当图片进入可视区域时才加载，提高了用户体验，尤其在移动设备上更为显著。

图片压缩是减小图片文件大小的过程，以便在网络中更快地传输和存储。压缩图片可以节省带宽、减少加载时间，并降低服务器存储成本。图片压缩的方法有多种，包括有损压缩和无损压缩。有损压缩会在一定程度上降低图片质量以减小文件大小，而无损压缩则可以在不损失质量的情况下减小文件大小。

（3）使用Vue.js的虚拟DOM提高页面渲染效率

Vue.js 是一种构建用户界面的渐进式框架，它通过其独特的响应式系统和组件化结构，使得开发变得简单且高效。其中，Vue.js 的一个核心特性就是其虚拟 DOM（Virtual DOM）的实现，它极大地提高了页面渲染的效率。

虚拟 DOM 是 Vue.js 用来优化真实 DOM 操作的一种技术。当我们需要更新页面上的某个元素时，Vue.js 并不是直接操作这个元素（真实 DOM），而是先更新其在 JavaScript 对象中的一个副本（虚拟 DOM）。然后，Vue.js 会对比这个副本和上一次渲染时生成的虚拟 DOM，找出需要变更的部分，最后将这些变更一次性地应用到真实 DOM 上。这种方式避免了大量不必要的 DOM 操作，从而提高了页面渲染的效率。

8.2 后端性能优化

（1）使用缓存技术减轻数据库压力

**本地缓存**：本地缓存（如Redis、Memcached等）可以将常用数据存储在内存中，使得应用程序能够更快地访问这些数据。当应用程序需要数据时，它会首先检查本地缓存中是否存在所需的数据。如果存在，则直接从缓存中获取数据，无需访问数据库；如果不存在，则从数据库中获取数据并将其存储在缓存中，以备后续使用。

**分布式缓存**：对于大型分布式系统，本地缓存可能无法满足需求。此时，可以使用分布式缓存（如Redis Cluster、Memcached集群等）来存储数据。分布式缓存将数据分散存储在多个节点上，提高了系统的可扩展性和容错性。同时，它还可以根据数据的访问频率和重要性来动态调整缓存策略，进一步提高缓存的利用率和效率。

**数据库查询缓存**：一些数据库管理系统（如MySQL等）提供了查询缓存功能。查询缓存可以缓存SQL查询的结果，以便在相同的查询再次执行时直接从缓存中获取结果，而无需重新执行查询。然而，需要注意的是，在高并发场景下，查询缓存可能会导致性能问题，因为它需要处理大量的并发访问和更新操作。因此，在使用查询缓存时需要根据实际情况进行配置和优化。

**对象关系映射（ORM）缓存**：在使用ORM框架（如MyBatis等）时，可以配置ORM缓存来减少数据库访问。ORM缓存可以将对象数据存储在内存中，使得在需要访问这些对象时可以直接从缓存中获取，而无需从数据库中加载。这可以大大减少数据库的访问次数，提高系统的响应速度。

（2）异步处理订单和支付请求

#### **订单处理**

#### **接收订单请求**：当用户提交订单时，系统接收订单信息并将其存储在数据库中（或缓存中）。

#### **触发异步任务**：使用一个消息队列或任务调度器来触发一个异步任务，该任务负责处理订单的后续逻辑。

#### **处理异步任务**：异步任务处理器从队列中获取订单信息，并执行相应的业务逻辑。处理完成后，它可以将结果存储回数据库，并通过事件通知系统将结果通知给相关的客户端。

#### **支付处理**

#### **接收支付请求**：当用户选择支付方式并提交支付请求时，系统接收支付信息并将其与订单关联起来。

#### **触发支付网关请求**：与订单处理类似，系统可以触发一个异步任务来调用支付网关的API进行支付处理。这个异步任务可以负责与支付网关进行通信、处理支付结果等。

#### **处理支付结果**：支付网关将支付结果返回给异步任务处理器。处理器根据支付结果更新订单状态（如已支付、支付失败等），并通过事件通知系统将结果通知给相关的客户端。

（3）水平扩展应用服务器和数据库服务器

**应用服务器水平扩展**：

应用服务器用于运行各种应用程序，提供业务逻辑处理、运算和用户界面展示等功能。水平扩展应用服务器主要通过增加服务器数量来提高系统的并发处理能力和可靠性。具体来说，当系统面临高并发访问时，可以通过添加更多的应用服务器来分摊负载，确保每台服务器都能高效处理请求，从而提高整体性能。同时，多台服务器可以互为冗余，一旦某台服务器出现故障，其他服务器可以接管其工作，保证系统的可用性和稳定性。

**数据库服务器水平扩展**：

数据库服务器是一种专门用于存储、管理和提供数据的计算机系统。水平扩展数据库服务器主要通过增加数据库节点数量来提高系统的处理能力和可靠性。具体来说，可以采用分布式数据库架构，将数据分散存储在多个数据库节点上，通过数据复制和分区等技术实现数据的高可用性和负载均衡。当某个数据库节点出现故障时，其他节点可以接管其工作，保证数据的持续可用。此外，水平扩展数据库服务器还可以根据业务需求动态调整数据库节点的数量，以满足不断变化的业务负载需求。

**9. 扩展性考虑 （王攀）**

9.1 前端扩展性

（1）使用组件化开发提高页面的可复用性

a.组件化设计：将页面拆分为多个独立的组件，每个组件负责实现 特定的功能或展示特定的内容。组件可以是按钮、表单、列表、 轮播图等，每个组件都是独立的、可复用的单元。

b.组件化开发工具：使用现有的组件化开发工具或框架，如React、 Vue等，来构建页面。这些工具提供了组件化开发的基础架构和 支持，可以更方便地创建、管理和复用组件。

c.组件库：建立组件库，将常用的组件收集整理并进行分类，以便 在需要时快速引用和使用。组件库可以包括UI组件、布局组件、 功能性组件等，提供了丰富的组件资源供开发人员选择和使用。

d.模板化设计：设计通用的模板和布局，将页面的结构和样式进行 模块化，使得不同页面可以共享相同的模板和布局，提高页面的 一致性和可维护性。

e.组件通信：采用合适的组件通信方式，如props和events（React）、 props和$emit（Vue）等，实现组件之间的数据传递和事件处理， 保持组件的独立性和可复用性。

f.单一职责原则：确保每个组件只负责实现一个功能或呈现一个特 定的内容，遵循单一职责原则，使得组件更容易被理解、测试和 维护。

（2）使用路由管理页面跳转，方便添加新页面

a.选择适合的路由库**：**在项目中选择适合的路由库，如React中的 React Router、Vue中的Vue Router等。这些路由库提供了路由 管理的基础功能和API，可以方便地实现页面跳转和状态管理。

b.配置路由：在项目中配置路由，定义页面之间的路由关系和跳 转规则。通常需要设置路由路径和对应的组件，以及可能的 参数传递和路由守卫等。

c.页面跳转：在代码中使用路由库提供的API进行页面跳转。可以 通过编程式导航或声明式导航的方式实现页面跳转，具体方法取 决于所选择的路由库和项目需求。

d.添加新页面：当需要添加新页面时，只需要在路由配置中添加对 应的路由路径和组件即可。无需修改其他页面或代码，便实现了 新页面的添加和跳转。

e.路由参数传递：在需要传递参数的情况下，可以通过路由参数、 查询参数、路由状态等方式进行参数传递，实现页面之间的数据 交互和状态管理。

f.路由守卫：根据需要设置路由守卫，用于控制页面的访问权限和 行为。可以在路由跳转前、跳转后或路由变化时执行相关逻辑， 实现页面的安全控制和业务逻辑处理。

9.2 后端扩展性

（1）使用模块化设计，方便添加新功能模块

a.拆分功能模块：首先，将项目中的功能按照业务逻辑和功能特点 进行拆分，划分为独立的模块。每个模块负责实现特定的功能或 业务逻辑，保持模块之间的高内聚低耦合。

b.模块化组织代码：将拆分后的功能模块组织成独立的代码文件或 文件夹，使用合适的命名规范和文件结构进行组织。可以采用单 文件组件、模块化文件夹结构等方式，使得每个模块都能够独立 开发、测试和维护。

c.模块间通信：确定模块之间的通信方式和接口规范。可以通过事 件总线、状态管理器、发布订阅模式等方式实现模块间的数据传 递和通信，保持模块之间的解耦合和灵活性。

d.模块化开发工具：选择适合的模块化开发工具或框架，如Vue、 React等，利用其提供的模块化开发特性和工具支持，方便地创 建、组织和扩展功能模块。

e.新增功能模块：当需要添加新的功能模块时，只需在项目中新增 对应的模块文件或文件夹，并按照模块化的设计原则进行开发和 实现。无需修改其他模块或代码，便实现了新功能模块的添加和 集成。

f.模块复用和拓展：设计和开发模块时，考虑到模块的复用和拓展 性，使得模块可以在不同的项目中重复使用，并且可以方便地进 行扩展和定制，满足不同项目和业务需求

g.模块文档和示例：为每个功能模块编写文档和示例，说明模块的 功能、用法和接口规范，方便其他开发人员理解和使用。这有助 于提高团队协作效率和代码质量。

h.模块测试和质量保证：为每个功能模块编写单元测试和集成测 试，确保模块的功能和性能符合预期。同时，定期进行代码审查 和质量评估，提高代码质量和稳定性。

i.模块版本管理：采用合适的版本管理工具和流程，对功能模块进 行版本控制和发布管理。确保不同版本之间的兼容性和迁移性， 以及新功能模块的平稳升级和更新。

j.模块性能优化：针对功能模块的性能和资源占用情况，进行必要 的优化和调整。可以采用代码分割、懒加载等技术，提高页面加 载速度和性能表现，提升用户体验。

k.模块安全性保护：对功能模块的安全性进行评估和防护，确保模 块不受恶意攻击和漏洞利用。可以采用安全编码规范、数据加密 等技术，保护用户数据和系统安全。

（2）使用微服务架构，将功能拆分为独立的服务，提高系统的 灵活性和可维护性

a.服务拆分：将系统按照业务功能划分为独立的服务。每个服务都 可以独立部署、扩展和维护，降低了服务之间的耦合性，提高了 系统的灵活性和可维护性。

b.独立部署：每个微服务都可以独立部署，不受其他服务的影响。 这意味着可以根据需要对单个服务进行更新、扩展或缩减，而不 必影响到整个系统的运行。

c.技术栈多样性：每个微服务可以使用适合其需求的最佳技术栈。 这样可以使团队更专注于他们擅长的技术领域，并且可以灵活选 择适合特定服务的技术，提高了开发效率和系统的可维护性。

d.易于扩展：微服务架构使得系统可以更容易地扩展和缩减，根据 系统负载和需求变化灵活调整各个服务的规模和资源分配。这样 可以更好地应对系统的高峰时段和增长需求，提高了系统的可扩 展性。

e.故障隔离：由于每个微服务都是独立部署和运行的，因此系统的 某个服务出现故障不会影响到其他服务的正常运行。这种故障隔 离的特性有助于提高系统的稳定性和可靠性。

f.团队自治：每个微服务可以由一个小团队负责开发、测试和运维， 这种团队自治的方式可以提高团队的责任感和主动性，加速项目 的开发和迭代速度

g.服务发现与治理：微服务架构中存在大量的服务实例，需要有效 地进行服务发现和治理。使用服务注册与发现机制，如Consul、 Etcd、Zookeeper等，可以帮助服务实例自动注册和发现，并提 供负载均衡、服务路由等功能，以确保服务之间的通信和协作。

h.监控与告警：由于微服务架构中存在大量的服务实例和服务之间 的复杂调用关系，需要建立有效的监控和告警系统，及时发现并 解决潜在的问题。使用监控工具，如Prometheus、Grafana等， 可以实时监控服务的运行状态和性能指标，并设置告警规则，及 时响应异常情况。

i.容错与恢复：微服务架构中的服务实例可能由于网络故障、资源 耗尽等原因而出现故障，需要具备容错和恢复能力。使用断路器、 重试机制、熔断器等技术，可以提高系统对故障的容忍能力，并 实现自动恢复和降级处理。

**10. 运维策略 （王攀）**

10.1 监控与日志

（1）使用监控工具实时监测系统的运行状态和性能指标

a.Prometheus：Prometheus 是一个开源的监控和警报工具包，适用 于记录任何可观察性的数字数据。它可以监控 CPU 使用率、内 存占用、磁盘使用量、网络流量等系统资源指标，以及应用程序 的请求响应时间、错误率等应用性能指标。

b.Grafana：Grafana 是一个开源的数据可视化和监控平台，通常与 Prometheus 等数据源一起使用。它可以根据收集的数据生成各 种类型的图表和仪表板，包括折线图、柱状图、饼图等，帮助用 户实时监测系统的运行状态和性能指标。

c.InfluxDB：InfluxDB 是一个开源的时间序列数据库，专门用于存储 和查询时间序列数据。它可以存储各种类型的监控数据，如系统 指标、应用性能指标、日志数据等，并提供灵活的查询语言和高 效的数据存储方式。

d.ELK Stack：ELK Stack 包括 Elasticsearch、Logstash 和 Kibana 三 个开源工具，用于日志收集、存储、搜索和可视化。它可以实时 监控系统日志、分析日志数据、检测异常和故障，帮助用户了解 系统的运行情况和性能表现。

f.Zabbix：Zabbix 是一个企业级的监控解决方案，提供了广泛的监 控功能，包括服务器性能、网络设备、数据库性能等。它可以实 时监控系统的运行状态、收集各种指标数据，并提供图形化的监 控报表和警报功能。

（2）记录系统运行日志，便于故障排查和性能优化

a.日志级别设置：根据日志的重要性和信息量，设置不同级别的日 志，如DEBUG、INFO、WARNING、ERROR、CRITICAL等。DEBUG 级别用于调试信息，INFO级别用于一般信息，WARNING级别用 于警告信息，ERROR级别用于错误信息，CRITICAL级别用于严重 错误信息。

b.日志格式规范：定义统一的日志格式，包括时间戳、日志级别、 模块名称、线程ID等关键信息。可以选择常见的日志格式，如 JSON格式、日志标准格式（RFC3339）等，以便后续日志分析和 处理。

c.异常信息记录：捕获系统中的异常信息，并记录到日志中。包括 异常类型、异常消息、异常堆栈信息等，有助于定位问题并进行 及时排查。

d.性能指标记录：记录系统的性能指标和关键指标，如请求响应时 间、数据库查询时间、内存占用、CPU利用率等。可以帮助评估 系统的性能瓶颈和优化方向。

e.业务操作记录：记录用户的操作行为和业务流程，包括用户请求、 请求参数、请求结果等。有助于跟踪用户行为和业务流程，发现 潜在问题并进行优化。

f.定期日志轮转：定期对日志文件进行轮转和归档，避免日志文件 过大导致占用过多的磁盘空间。可以设置按大小、按时间、按数 量等方式进行轮转，保留一定的历史日志用于故障排查和审计。

g.集中化日志管理：将日志集中管理到统一的日志服务器或日志平 台，方便查看和分析。可以选择开源的日志管理工具，如ELK Stack、Fluentd、Graylog等，也可以选择云端日志服务，如AWS CloudWatch、Google Cloud Logging等。

10.2 自动化部署

（1）使用CI/CD工具实现自动化构建和部署流程

a.选择合适的CI/CD工具：首先需要选择适合项目需求的CI/CD工 具，如Jenkins、GitLab CI、Travis CI、CircleCI等。这些工具提供 了丰富的功能和灵活的配置选项，能够满足不同规模和需求的项 目。

b.配置代码仓库集成：将代码仓库与CI/CD工具进行集成，实现代 码的自动触发和持续集成。可以设置Webhook或轮询等方式， 监控代码仓库的变化，并在代码提交或分支合并时触发自动构建 流程。

c.编写构建脚本：根据项目的需求和构建环境，编写构建脚本或配 置文件。构建脚本可以包括编译代码、运行测试、生成文档、打 包发布等步骤，确保代码在构建过程中的正确性和稳定性。

d.配置构建任务：在CI/CD工具中创建构建任务，并配置构建脚本 和相关参数。可以设置触发条件、构建触发器、构建步骤、构建 环境等选项，确保构建任务能够按照预期执行并生成期望的结果。

e.配置部署流程：根据项目的部署需求，配置部署流程和目标环境。 可以选择自动部署或手动部署、单机部署或多机集群部署等方式， 确保部署流程能够自动化和可靠化。

f.自动化测试与验证：在部署流程中集成自动化测试和验证步骤， 确保部署的软件版本符合预期并能够正常工作。可以包括功能测 试、性能测试、安全测试等方面的验证，确保软件质量和用户体 验。

g.持续监控与反馈：在CI/CD工具中设置监控和告警规则，监控构 建和部署过程中的关键指标和异常情况。及时发现并响应问题， 确保软件交付的稳定性和可靠性。

（2）使用容器化技术实现应用的快速部署和扩展

a.选择容器化平台：首先需要选择适合项目需求的容器化平台，如 Docker、Kubernetes、Docker Swarm等。这些平台提供了容器的 创建、部署、管理和编排功能，能够满足不同规模和需求的项目。

b.编写Dockerfile：为应用编写Dockerfile，定义应用的运行环境和 依赖关系。Dockerfile中包括基础镜像的选择、软件包的安装、 代码的复制等步骤，确保容器能够正确地运行应用。

c构建Docker镜像：使用Docker命令或CI/CD工具，构建应用的 Docker镜像。在构建过程中，将Dockerfile和应用代码打包成镜 像，并上传到镜像仓库，以便后续部署和扩展使用。

d编写容器编排配置：如果使用Kubernetes等容器编排工具，需要 编写容器编排配置文件，定义应用的部署和扩展规则。包括Pod 的定义、服务的暴露、副本集的配置等，确保应用能够在集群中 稳定运行。

e部署应用容器：使用容器编排工具或手动操作，部署应用容器到 容器集群中。根据容器编排配置文件，创建和管理应用的Pod副 本，确保应用能够在集群中高可用和可扩展。

f.监控和管理容器：使用容器监控工具或集成监控系统，监控和管 理应用容器的运行状态和性能指标。包括CPU利用率、内存占用、 网络流量等方面的监控，及时发现并解决潜在问题。

g自动化运维和扩展：根据实际需求，设置自动化运维和扩展策略， 自动化处理容器的启动、停止、伸缩等操作。可以根据负载情况 和资源利用率，自动调整容器的数量和配置，提高系统的弹性和可靠性。

**11. 维护策略 （刘易）**

11.1 定期更新

（1）定期检查和更新系统的依赖库和组件版本：

* 前端方面,定期检查并更新Vue.js, ElementUI等框架和UI库的版本,确保使用最新的特性、性能优化和安全补丁。
* 后端方面,跟进Spring Boot, Spring Cloud, MyBatis等关键组件的版本更新,修复已知安全漏洞,提升系统性能。
* 对于Node.js中间层,也要留意Node.js运行时和npm包的版本更新

（2）定期备份数据库，确保数据的安全性和可恢复性：

* 根据数据的重要性和变更频率,制定每日、每周、每月的自动化增量和全量备份计划。
* 将备份文件存储在独立的服务器或云存储,防止单点故障。
* 定期对备份的数据进行恢复测试,确保备份的有效性,并优化恢复流程。

11.2 Bug修复

（1）及时响应用户反馈和系统报警，修复已知Bug：

* 建立完善的用户反馈渠道,包括App内反馈、客服系统、社交媒体账号等,鼓励用户提供Bug反馈和改进建议。
* 对自动化异常监控和报警系统(如Spring Boot Admin)产生的告警信息,要及时分析和处理。
* 根据Bug的严重程度和影响范围,制定修复优先级,并设置合理的解决时限。
* Bug修复后,要进行充分的回归测试,确保没有引入新的缺陷。

（2）定期进行安全漏洞扫描和修复，确保系统的安全性：

* 使用OWASP ZAP等专业安全扫描工具,定期对系统进行全面的安全扫描。
* 关注主流组件和库的安全通告,评估已知漏洞对系统的影响。
* 根据漏洞的危害性,制定修复计划,并及时通过升级组件版本、修改配置、增加安全校验等方式完成漏洞修复。
* 加强对开发人员的安全培训,提高安全编码意识,减少代码层面的安全缺陷。

**12. 风险管理 （刘易）**

12.1 高并发

（1）使用负载均衡和分布式架构提高系统的并发处理能力：

* 使用Nginx等反向代理服务器,对多个后端服务实例进行负载均衡,提高并发处理能力。
* 利用Spring Cloud的服务发现、配置管理、熔断等特性,实现微服务架构,便于水平扩展。
* 对关键服务（如订单、支付）进行独立扩展,避免单个服务成为瓶颈。
* 采用分布式数据库方案（如ShardingSphere）,实现数据的分片存储和读写分离。

（2）使用缓存技术和异步处理减轻服务器压力：

* 使用Redis等内存缓存,对热点数据（如商品信息、用户session）进行缓存,减轻数据库压力。
* 利用消息队列(如Kafka、RabbitMQ)异步处理非实时性的任务,如订单状态通知、数据统计分析等。
* 对于高并发的写请求,可采用缓存+数据库的最终一致性方案,先写缓存再异步同步到数据库。

12.2 安全漏洞

（1）定期进行安全漏洞扫描和修复：

* 在代码合并和发布前,要进行严格的代码审查,检查是否引入安全漏洞。
* 将安全测试纳入开发流程,对认证授权、数据校验、敏感信息传输等关键场景进行充分的安全测试。
* 定期进行外部安全审计,发现潜在的架构和配置层面的安全风险。

（2）提高开发人员的安全意识，加强代码审查和漏洞修复：

* 对外部传输的敏感数据采用HTTPS加密,并使用足够强度的加密算法。
* 对存储的用户隐私数据,如密码、手机号等,进行不可逆加密或者脱敏处理。
* 严格限制内网服务器的外网访问,防止敏感数据泄露。

**附加 上周的其他实验任务如下：**

对比书上各种软件体系结构风格和视图特点，思考自己项目属于哪种设计风格？（张勇）

对于外卖系统项目，考虑到其复杂性和需求的多样性，可以选择一种结合了多种软件体系结构风格的混合体系结构。结合分层结构、微服务架构、事件驱动架构和客户端-服务器结构等多种特点，以实现系统的高性能、高可用性和高可扩展性。

分层结构：外卖系统可以采用分层结构来组织其各个模块和组件。比如可以将系统分为表示层（Presentation Layer）、业务逻辑层（Business Logic Layer）和数据访问层（Data Access Layer）。这样可以将系统的不同功能模块进行逻辑上的分离，降低耦合度，提高可维护性和可扩展性。

微服务架构：作为一种新兴的软件体系结构风格，微服务架构也可以在外卖系统中发挥重要作用。将系统拆分为多个小型、自治的服务，每个服务都负责一个特定的业务功能，通过定义清晰的接口进行通信。这样可以实现系统的松耦合、可扩展和可维护。

事件驱动架构：外卖系统中涉及到大量的异步操作，比如订单状态的变化、库存的更新等。采用事件驱动架构可以更好地处理这些异步事件，通过发布-订阅模式实现服务之间的解耦，提高系统的响应性和可扩展性。

客户端-服务器结构：外卖系统通常包含多个客户端应用（Web、移动应用等）和服务器端应用。采用客户端-服务器结构可以将系统的核心业务逻辑和数据存储放在服务器端，而将与用户交互的界面和逻辑放在客户端，从而实现系统的分布式部署和管理。

针对上述软件体系结构风格的外卖系统项目，可以分别从不同视图的特点进行思考：

分层结构视图：

逻辑视图：分层结构将系统划分为不同的层次，每个层次有其特定的责任和功能。逻辑视图可以展示各个层次之间的交互关系和数据流动。

开发视图：开发视图可以展示各个层次的开发工作分配情况，每个团队负责开发和维护特定的层次，通过清晰的接口定义实现层与层之间的交互。

部署视图：部署视图可以展示系统各个层次的部署情况，比如表示层部署在Web服务器上，业务逻辑层和数据访问层部署在应用服务器上，数据库层部署在数据库服务器上。

微服务架构视图：

逻辑视图：微服务架构将系统拆分为多个小型、自治的服务，每个服务都有其独立的数据模型和业务逻辑。逻辑视图可以展示各个微服务之间的关系和依赖。

开发视图：开发视图可以展示各个微服务的开发团队和技术栈，每个团队负责开发和维护一个或多个微服务，通过清晰的接口定义实现服务之间的通信。

部署视图：部署视图可以展示各个微服务的部署情况，每个微服务可以独立部署和扩展，通过服务发现和负载均衡实现服务的高可用性和负载均衡。

事件驱动架构视图：

逻辑视图：事件驱动架构通过发布-订阅模式来处理系统中的异步事件，逻辑视图可以展示系统中涉及的各种事件和事件处理逻辑。

开发视图：开发视图可以展示各个事件处理器的开发团队和技术栈，每个团队负责开发和维护特定的事件处理器，通过事件总线进行事件的发布和订阅。

部署视图：部署视图可以展示各个事件处理器的部署情况，通过事件总线实现事件处理器之间的解耦，提高系统的可扩展性和灵活性。

客户端-服务器结构视图：

逻辑视图：客户端-服务器结构将系统划分为客户端应用和服务器端应用，逻辑视图可以展示客户端与服务器之间的交互关系和数据传输。

开发视图：开发视图可以展示客户端和服务器端的开发团队和技术栈，分别负责开发和维护客户端和服务器端的应用程序。

部署视图：部署视图可以展示客户端和服务器端的部署情况，客户端应用部署在用户设备上，服务器端应用部署在服务器上，通过网络进行通信。

2.网上搜索最新的软件体系结构资料，如MVC、Kruchten 4+1视图等。（丁弘扬）

Kruchten 4+1视图是对逻辑架构进行描述，最早由 Philippe Kruchten 提出，他在1995年的《IEEE Software》上发表了题为《The 4+1 View Model of Architecture》的论文，引起了业界的极大关注，并最终被RUP采纳，现在已经

成为架构设计的结构标准。包括：逻辑视图，开发视图，处理视图，物理视图，场景视图。

逻辑视图：用于描述系统的功能需求，即系统给用户提供哪些服务；以及描述系统软件功能拆解后的组件关系、组件约束和边界，反映系统整体组成与系统如何构建的过程。

开发视图：开发视图关注软件开发环境下实际模块的组织，反映系统开发实施过程。

处理视图：又称过程视图、运行视图。用于描述系统软件组件之间的通信时序，数据的输入输出。

物理视图：开发出的软件系统，最终是要运行在物理或软件环境上。物理环境可能是服务器、PC机、移动终端等物理设备；软件环境可以是虚拟机、容器、进程或线程。物理视图关注“目标程序及其依赖的运行库和系统软件”最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。

场景视图：即4+1中的1。4+1中的4个视图都是围绕着场景视图为核心的。它用于描述系统的参与者与功能用例间的关系，反映系统的最终需求和交互设计。

MVC：模型-视图-控制器模式，也称为MVC模式，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。MVC被独特的发展起来用于映射传统的输入、处理和输出功能在一个逻辑的图形化用户界面的结构中。它把软件系统分为三个基本部分：模型（Model）负责存储系统的中心数据，视图（View）将信息显示给用户，控制器（Controller）处理用户输入的信息。负责从视图读取数据，控制用户输入，并向模型发送数据，是应用程序中处理用户交互的部分。负责管理与用户交互，交互控制。

MVC特点：

MVC重要特点就是两种分离：

视图和数据模型的分离：使用不同的视图对相同的数据进行展示；分离可视和不可视的组件，能够对模型进行独立测试。因为分离了可视组件减少了外部依赖利于测试。

视图和表现逻辑(Controller)的分离：Controller是一个表现逻辑的组件，并非一个业务逻辑组件。MVC可以作为表现模式也可以作为建构模式，意味这Controller也可以是业务逻辑。分离逻辑和具体展示，能够对逻辑进行独立测试。

MVC优点：耦合性低；重用性高；生命周期成本低；部署块；可维护性高；有利软件工程化管理。

MVC缺点：没有明确的定义；不适合小型，中等规模的应用程序；增加系统结构和实现的复用性；视图与控制器间的过于紧密的连接；视图对模型数据的低效率访问；一般高级的界面工具或构造器不支持模式。

分层模式：这种模式可以用来构建能分解为子任务组的结构化程序，每个子任务都处于特定的抽象级别。每层为下一更高层提供服务。一个通用信息系统常见的4层如下：呈现层（UI-用户界面层），应用层（service-服务层），业务逻辑层（domain-领域层），数据访问层（persistence-持久化层）

CS模式：这个模式包含两个部分：一个服务端+多个客户端。服务端组件提供给多个客户端组件服务。客户端请求服务，服务端提供相应的服务给客户端。除此之外，服务端不间歇地监听来自客户端的服务请求。

主从模式：这种模式由两部分组成：主人（master）和奴隶（slaves）。主组件将工作分配给特定的从组件，并根据从组件返回的结果计算最终结果。

管道过滤器模式：这个模式可被用于构建生成和处理数据流的系统。每个处理步骤都包含在一个过滤器组件中。被处理的数据需通过管道进行传递。这些管道可用于缓冲或同步目的。

经济人模式：这个模式用于使用架构的组件来构建的分布式系统。这些组件可以通过远程服务调用相互交互。代理组件负责协调组件之间的通信。服务端将其能力（服务和特性）发布给代理。客户端向经纪人请求服务，然后经纪人将客户重定向到其注册的对应服务。

点对点模式：在这种模式中，单个组件被称为同级点。 同级点可以既作为客户端，向其它同级点请求服务，又作为服务器向其它同级点提供服务。 一个同级点既可以充当客户端或服务器或两者兼而有之，并且可以随着时间动态地改变其角色。

事件总线模式：这个模式主要用于处理事件，有4个主要的组件：事件源，事件监听器，频道，事件总线。事件源将消息发布到事件总线上的特定频道。监听器订阅特定频道。监听器会收到发布到他们之前订阅的频道的消息。

黑板模式：这种模式对于没有确定性解决策略的问题是有帮助的。黑板模式由3个主要组件组成。黑板：一个包含来自解决方案空间对象的结构化全局内存。知识源：具有自我表达的专用模块。控制组件：选择，配置，执行模块。所有组件都可以访问黑板。 组件可能产生添加到黑板的新数据对象。 组件在黑板上查找特定类型的数据，并可能通过与现有知识源的模式匹配找到这些数据。

解释模式：此模式用于设计解释用专用语言编写的程序的组件。 它主要指定如何解释执行程序代码，称为用特定语言编写的句子或表达式。 基本思想是为语言的每个符号设置一个类。

3. 参阅课本和网上资料，研究经典软件体系结构案例KWIC。

An Introduction to Software Architecture，4.1节

On-the-Criteria-To-Be-Used-in-Decomposing-Systems-into-Modules（Example System 1）

<http://www.cs.cmu.edu/~ModProb/index.html> （刘易和王攀）

KWIC，即“Keyword in Context”（上下文中的关键字），是一种索引技术，用于组织和呈现文本中的信息，以便用户可以快速找到他们感兴趣的关键字及其上下文。KWIC索引通过将每个关键字放在一个中心位置，并显示关键字前后的文本来实现这一点。这种索引方法特别有用，因为它允许用户在不阅读整个文档的情况下快速识别与特定关键字相关的上下文。索引通常按字母顺序排列，使得用户可以轻松地找到特定的关键字。

案例分析：

案例一：An Introduction to Software Architecture, 4.1 节

关注点:

该案例主要介绍了 KWIC 系统的基本功能和设计约束，并提出了一个基于管道-过滤器架构的解决方案。

架构:

输入 -> 循环移位 -> 字母化 -> 输出  
每个阶段都是一个独立的模块，数据按顺序流经各个模块进行处理。

优点:

简单易懂，易于实现。

模块化程度高，便于维护和测试。

缺点:

灵活性较差，难以应对需求变化，例如添加新的功能或修改排序规则。

效率可能不高，因为数据需要依次通过所有模块。

案例二：On-the-Criteria-To-Be-Used-in-Decomposing-Systems-into-Modules (Example System 1)

关注点:

该案例主要探讨了 KWIC 系统的模块化分解原则，并提出了两种基于数据流和功能分解的方案。

方案一 (数据流):

Line Storage (数据存储) -> Input (输入) -> Circular Shift (循环移位) -> Alphabetizer (排序) -> Output (输出)  
数据存储模块用于存放输入数据，其他模块围绕数据存储进行操作。

方案二 (功能分解):

Master Control (主控模块) -> Input (输入) -> Circular Shifter (循环移位模块) -> Alphabetizer (排序模块) -> Output (输出)  
主控模块负责协调各个功能模块的执行。

优点:

方案一数据流清晰，方案二功能明确。

比案例一更具灵活性，更容易扩展和修改。

缺点:

方案一模块之间耦合度较高。

方案二 Master Control 模块可能过于复杂。

案例三：[http://www.cs.cmu.edu/~ModProb/index.html](http://www.cs.cmu.edu/~ModProb/index.html" \t "https://cloud.dify.ai/app/e12e14e9-b4e6-4068-b870-66177004fa2b/_blank)

ModProb KWIC 系统特点:

基于 ModProb 框架: 利用 ModProb 平台的模块化特性和概率推理能力。

数据表示: 使用 ModProb 的知识库表示 KWIC 索引和相关信息。

推理引擎: 使用 ModProb 的推理引擎执行查询和检索操作。

实现方式:

数据输入模块: 将文本数据转换为 ModProb 知识库中的事实和规则。

循环移位模块: 可以通过规则或函数实现，生成所有可能的循环移位，并将其存储在知识库中。

排序模块: 利用 ModProb 的推理引擎根据指定的排序规则进行排序，例如按字母顺序或按行号排序。

输出模块: 从知识库中检索排序后的结果，并将其格式化为用户友好的输出形式。

优点:

模块化和可扩展: 继承 ModProb 平台的优点，KWIC 系统易于扩展和修改。

概率推理: 可以利用 ModProb 的概率推理能力进行更复杂的查询和检索，例如模糊匹配、概率排序等。

知识重用: KWIC 索引可以与其他知识库集成，实现知识共享和重用。

缺点:

复杂性: ModProb 平台本身具有一定的复杂性，学习曲线较陡峭。

性能: 对于大规模数据，ModProb 的性能可能不如专门设计的 KWIC 系统。