# 基于AI导购功能的电商网站设计与实现

## 摘要

本文设计并实现了一个基于AI导购功能的电商网站，旨在解决传统电商平台用户选购决策困难、信息过载等问题。系统采用前后端分离架构，前端基于React框架开发，后端使用Node.js构建REST API服务，并集成DeepSeek大语言模型实现智能导购功能。系统的核心创新点是引入”沙漏模型”AI导购流程，通过需求表达、参数筛选、对比选择和决策推荐四个阶段，引导用户完成购买决策。系统实现了多模态搜索、个性化推荐、智能对比和售后服务等功能。测试表明，系统在提高用户购物体验、缩短决策时间方面具有积极效果，为AI技术在电商领域的应用提供了新思路。

**关键词**：电子商务 人工智能 大语言模型 智能导购 沙漏模型 个性化推荐

## ABSTRACT

This thesis designs and implements an e-commerce website with AI-powered shopping assistant functionality, aiming to solve problems such as decision-making difficulties and information overload on traditional e-commerce platforms. The system adopts a front-end and back-end separation architecture, with the front-end developed based on the React framework and the back-end built using Node.js to construct REST API services, while integrating the DeepSeek large language model to implement intelligent shopping guidance. The core innovation of the system is the introduction of the “Hourglass Model” AI shopping guidance process, which guides users to complete purchase decisions through four stages: demand expression, parameter filtering, comparison selection, and decision recommendation. The system implements multi-modal search, personalized recommendations, intelligent comparison, and after-sales service. Tests show that the system has positive effects in improving user shopping experience and shortening decision time, providing new ideas for the application of AI technology in the e-commerce field.

**KEY WORDS**: E-commerce, Artificial Intelligence, Large Language Model, Intelligent Shopping Assistant, Hourglass Model, Personalized Recommendation

## 目录

1. [绪论](#第1章-绪论) 1.1 [研究背景与意义](#X2c7e9b8f9781efbb2a6a4a7c7139706ed2d720d) 1.2 [国内外研究现状](#Xe434105f28de43e6620ddd8ba8024719a12d506) 1.3 [研究内容与目标](#X8ee81689dca2fc040d37328ab01be94b64dbb2e) 1.4 [论文结构安排](#X42be0e34395e43e9ce6fceb4e0fd078de1453d0)
2. [系统需求分析](#第2章-系统需求分析) 2.1 [业务需求分析](#X76657486b98d9685994e013004477d6bc21c316) 2.2 [功能需求分析](#X004e2f2482a06b2f2f3643f1f640bd81228c54d) 2.3 [非功能需求分析](#Xaccaebaf13c091021f4f9fd930961a4dd6731c8) 2.4 [用户角色分析](#X6cb597438c6549427832865cd0556da6029af5c)
3. [系统总体设计](#第3章-系统总体设计) 3.1 [系统架构设计](#Xe9ce35d2c4ef08a93d645ef66b9fc2ee389bfb2) 3.2 [技术选型](#X33282d66c0dc56eccfbfae4a7ec72a771afebbf) 3.3 [系统模块设计](#X1d1e4cb14bde9d931efe58a8a3d0553af58f62f) 3.4 [数据库设计](#X953dd44843a6e3ca90efee5ccc06d1a50cf981e) 3.5 [“沙漏模型”AI导购流程设计](#X19c148b542875af08029ec8de3d88a17ef60853)
4. [系统设计与实现](#第4章-系统详细设计与实现) 4.1 [开发环境与工具](#Xc2f57b60a01da2caac0b7147e4f8e7d84389808) 4.2 [系统前端设计与实现](#X645232ff166f609d3ad862474335ac1647b8b62) 4.3 [系统后端设计与实现](#Xe9173a8b5b7b1bf2457415638954c2060981aaa) 4.4 [AI导购功能设计与实现](#X97ee1dd332787e91111fb15aec508614212d67b) 4.5 [个性化推荐系统设计与实现](#Xca30f931fb1aef52d8c85eb3cb87f12ec97115e) 4.6 [多模态搜索设计与实现](#X4a1361a0ad27d81cfeea2c74bfdc29355029815) 4.7 [智能对比功能设计与实现](#Xbe37e73361bb80c5026bf3d6b8bf3a3e97fc337) 4.8 [智能售后服务设计与实现](#Xa1d275acb7cb53753a529f383aef873ab4cf04f)
5. [系统测试与评估](#第5章-系统测试与评估) 5.1 [测试环境与计划](#Xc4900c3bb7bb63ba4f3e29c206350f01ae8df06) 5.2 [功能测试](#X07438f455b36988746dc467706e09747dca5192) 5.3 [性能测试](#X4166bd80654af7186d8ac9343b0154469e4c11b) 5.4 [用户体验测试](#X86f91f91c5838cefebc48e94fe5877d736ba014) 5.5 [测试结果分析](#Xc49e07bf1eb23e5e62d16f020e22b4657b1a9ad)
6. [总结与展望](#第6章-总结与展望) 6.1 [项目总结](#Xe6af4419532bb46b4532a4f6e13032246c2a0e8) 6.2 [创新点总结](#Xd5ee3f714b210cb30990d2c019f2e1273342d40) 6.3 [不足与展望](#Xccbbea9386e1f21b32cb032e71873373ceae4b2)

参考文献

## 第1章 绪论

### 1.1 研究背景与意义

随着互联网技术的快速发展和电子商务的普及，网上购物已成为人们日常生活的重要部分。据中国互联网络信息中心（CNNIC）统计，我国网络购物用户规模已达数亿，占网民整体的比例持续上升。然而，随着电商平台商品种类的爆炸性增长，用户在面对海量商品时往往陷入”选择困难症”，表现为决策时间延长、选购体验降低以及购买后的认知失调等问题。

传统电商平台主要通过搜索引擎、筛选条件和基于协同过滤的推荐系统辅助用户购物，这些方法在帮助用户找到所需商品方面存在明显不足： 1. 用户需要具备较高的检索能力和专业知识 2. 难以处理复杂、模糊的需求表达 3. 缺乏个性化的购买指导和专业建议 4. 无法模拟线下导购的交互式体验

近年来，随着人工智能特别是大语言模型(LLM)技术的发展，将AI技术应用于电商领域的智能导购成为可能。大语言模型凭借其出色的语言理解、记忆与推理能力，可以像人类导购一样与用户进行自然语言交互，理解用户复杂的需求描述，提供个性化的商品推荐和专业购买建议，从而有效解决传统电商平台的用户决策困境。

本研究旨在设计和实现一个基于AI导购功能的电商网站，通过引入大语言模型DeepSeek并结合专门设计的”沙漏模型”购物决策流程，为用户提供智能化、个性化的导购服务。该系统的实现具有以下重要意义：

1. **理论意义**：探索大语言模型在电子商务领域的应用范式，为AI技术与垂直行业的融合提供实践经验。
2. **实践意义**：通过AI导购功能改善用户购物体验，提高用户满意度和平台转化率，降低退货率。
3. **社会意义**：降低用户在网购中的决策成本和时间成本，减轻信息过载带来的心理负担。
4. **产业意义**：推动电商行业的智能化升级，促进AI技术在商业场景的落地应用。

### 1.2 国内外研究现状

#### 1.2.1 智能导购研究现状

传统电商平台的导购功能主要通过搜索引擎、筛选条件和基于协同过滤的推荐系统实现。近年来，随着AI技术的发展，智能导购系统研究取得了进展：

**国外研究现状**： - Amazon推出了”Amazon Assistant”浏览器插件，提供价格跟踪和商品推荐功能。 - Google推出了”Google Shopping”服务，通过图像识别和个性化推荐提升购物体验。 - Pinterest的”Visual Search”技术允许用户通过图像寻找相似商品。 - 学术界方面，MIT的研究团队提出了结合强化学习和知识图谱的对话式推荐系统。

**国内研究现状**： - 阿里巴巴的”淘宝智能导购”系统集成了自然语言处理和用户行为分析技术。 - 京东的”京东AI购物助手”提供个性化推荐和价格预测功能。 - 小红书推出的”种草机器人”通过内容分析和社交推荐优化用户发现体验。 - 学术界方面，清华大学、中国科学院等机构在电商对话系统和多模态推荐系统方面取得了研究成果。

#### 1.2.2 大语言模型在电商领域的应用现状

大语言模型(LLM)作为近年来AI领域的重大突破，在电商领域的应用正处于探索阶段：

**国外应用现状**： - OpenAI的GPT系列模型被用于多个电商平台的客服和商品描述生成。 - Shopify集成了AI助手，帮助商家优化商品描述和营销文案。 - Amazon使用自研大语言模型优化其搜索引擎和推荐系统。 - eBay应用NLP技术改进其搜索算法和个性化推荐。

**国内应用现状**： - 阿里巴巴的”通义千问”被应用于平台的智能客服和内容生成。 - 京东使用自研大模型优化搜索和推荐效果。 - 美团将大语言模型应用于商品属性提取和搜索理解。 - 百度的文心大模型应用于其智能购物助手。

然而，目前大语言模型在电商导购领域的应用仍存在一些问题： 1. 缺乏专门针对购物决策流程的优化设计 2. 与电商平台的深度集成度不足 3. 个性化程度有限 4. 多模态能力不足 5. 专业领域知识欠缺

本研究将针对这些问题，提出基于”沙漏模型”的AI导购解决方案。

### 1.3 研究内容与目标

#### 1.3.1 研究内容

本研究的主要内容包括：

1. **电商平台需求分析**：分析用户在电商平台中的购物决策行为特点，明确智能导购系统的功能需求和非功能需求。
2. **AI导购”沙漏模型”设计**：设计针对电商决策的”沙漏模型”流程，包括需求表达、参数筛选、对比选择和决策推荐四个阶段。
3. **系统架构设计**：设计前后端分离的系统架构，实现电商平台与AI模型的集成。
4. **核心功能实现**：实现AI导购对话、多模态搜索、个性化推荐、智能对比和售后服务等功能模块。
5. **系统测试与评估**：对系统进行功能测试、性能测试和用户体验测试，评估系统的实用性和效果。

#### 1.3.2 研究目标

本研究的具体目标如下：

1. **技术目标**：
   * 设计并实现一个完整的基于AI导购功能的电商网站
   * 实现大语言模型DeepSeek与电商平台的融合
   * 实现”沙漏模型”AI导购流程
   * 实现多模态搜索和智能对比等功能
2. **用户体验目标**：
   * 提高用户购物决策效率，缩短决策时间
   * 增强用户购物体验的满意度
   * 降低购买后的认知失调和退货率
   * 提升用户对平台的忠诚度
3. **商业目标**：
   * 提高平台商品转化率
   * 提升平台用户留存率
   * 降低客服成本
   * 形成平台差异化竞争优势

### 1.4 论文结构安排

本论文的结构安排如下：

第1章为绪论，介绍了研究背景与意义、国内外研究现状、研究内容与目标。

第2章为系统需求分析，分析了系统的业务需求、功能需求、非功能需求和用户角色需求。

第3章为系统总体设计，包括系统架构设计、技术选型、系统模块设计、数据库设计和”沙漏模型”AI导购流程设计。

第4章为系统详细设计与实现，详细阐述了前端界面、后端服务、AI导购功能、个性化推荐系统、多模态搜索、智能对比功能和智能售后服务的设计与实现。

第5章为系统测试与评估，包括测试环境与方法、功能测试、性能测试、用户体验测试和测试结果分析。

第6章为总结与展望，总结研究工作的成果与创新点，分析不足，并对未来研究方向进行展望。

## 第2章 系统需求分析

### 2.1 业务需求分析

通过对现有电商平台用户购物行为的调研和分析，结合市场需求，本系统的业务需求主要包括以下几个方面：

#### 2.1.1 用户决策问题

现代电商平台面临的主要业务问题是用户在海量商品中的决策困难，具体表现在：

1. **信息过载**：商品数量庞大，用户难以有效筛选和比较。
2. **专业知识缺乏**：普通用户缺乏对商品技术参数的专业理解。
3. **决策疲劳**：长时间的选择和比较导致用户疲惫和放弃购买。
4. **认知失调**：购买后担心其他选择可能更好，导致满意度降低。

#### 2.1.2 业务目标

针对上述问题，本系统的业务目标包括：

1. **优化购物决策流程**：通过AI导购简化和优化用户的购物决策过程。
2. **提升用户体验**：提供个性化、智能化的购物体验，增强用户满意度。
3. **提高转化率**：缩短用户决策时间，提高商品购买转化率。
4. **增强用户忠诚度**：通过优质的购物体验提升用户对平台的黏性。
5. **降低退货率**：通过精准推荐和专业建议，减少购买后的后悔和退货行为。

#### 2.1.3 业务流程需求

系统需要支持以下核心业务流程：

1. **智能导购咨询**：用户能够通过自然语言与AI导购助手交流，表达需求并获取建议。
2. **产品发现与筛选**：AI助手根据用户需求推荐合适的商品，并支持多维度筛选。
3. **产品对比与评估**：系统支持多种方式的产品对比，帮助用户做出最佳选择。
4. **购买决策与确认**：系统提供决策支持，帮助用户确认最终购买选择。
5. **售后服务与反馈**：提供智能化的售后咨询和服务，处理用户反馈。

### 2.2 功能需求分析

根据业务需求分析，本系统的功能需求可归纳为以下几类：

#### 2.2.1 基础电商功能

1. **用户管理**：
   * 用户注册、登录、信息修改
   * 用户权限管理
   * 用户行为数据收集与分析
2. **商品管理**：
   * 商品分类与标签管理
   * 商品信息展示
   * 商品库存与价格管理
3. **订单管理**：
   * 购物车功能
   * 订单创建与支付
   * 订单状态跟踪
   * 订单历史查询

#### 2.2.2 AI导购核心功能

1. **智能对话导购**：
   * 自然语言交互界面
   * 用户需求理解与分析
   * 专业知识解答
   * 购买建议生成
   * 对话上下文记忆
2. **“沙漏模型”决策流程**：
   * 需求表达阶段引导
   * 参数筛选阶段辅助
   * 对比选择阶段支持
   * 决策推荐阶段建议
3. **个性化推荐**：
   * 基于用户画像的商品推荐
   * 基于上下文的实时推荐
   * 基于历史行为的兴趣推荐
   * 协同过滤与内容推荐结合

#### 2.2.3 高级功能

1. **多模态搜索**：
   * 文本搜索增强
   * 图像搜索
   * 语音搜索
   * 混合模态搜索
2. **智能对比分析**：
   * 商品参数对比
   * 价格历史对比
   * AI分析评测
   * 可视化对比结果
3. **智能售后服务**：
   * 售后问题智能解答
   * 操作指南生成
   * 退换货智能协助
   * 故障排查建议

### 2.3 非功能需求分析

#### 2.3.1 性能需求

1. **响应时间**：
   * 页面加载时间<3秒
   * API响应时间<1秒
   * AI回复响应时间<5秒
2. **并发能力**：
   * 支持合理数量用户同时在线
   * 保持系统在一定负载下稳定运行
3. **可扩展性**：
   * 支持用户和数据量增长
   * 模块化设计便于扩展

#### 2.3.2 安全需求

1. **数据安全**：
   * 用户隐私保护
   * 交易数据加密
   * 防SQL注入和XSS攻击
2. **用户认证**：
   * 安全的用户认证机制
   * 安全的密码存储
   * 访问权限控制
3. **系统安全**：
   * 防护常见安全攻击
   * 日志审计
   * 安全漏洞定期检测

#### 2.3.3 可用性需求

1. **用户友好性**：
   * 直观的用户界面
   * 清晰的操作流程
   * 适当的引导和提示
2. **可访问性**：
   * 支持多种设备访问
   * 适配不同屏幕尺寸
   * 网络环境优化
3. **容错性**：
   * 错误处理机制
   * 系统恢复能力
   * 用户操作撤销功能

### 2.4 用户角色分析

系统涉及的主要用户角色及其需求如下：

#### 2.4.1 普通用户

普通用户是系统的主要服务对象，其需求包括：

1. **功能需求**：
   * 浏览和搜索商品
   * 与AI导购交流咨询
   * 比较和选择商品
   * 完成购买流程
   * 查询订单和售后
2. **体验需求**：
   * 简单易用的界面
   * 个性化的推荐
   * 专业的购买建议
   * 高效的决策支持

#### 2.4.2 新用户

首次使用系统的用户，其特定需求包括：

1. **功能需求**：
   * 快速了解系统功能
   * 简单的注册流程
   * 基础的产品浏览
2. **体验需求**：
   * 直观的引导说明
   * 友好的新手教程
   * 简化的操作流程

#### 2.4.3 专业用户

具有较强专业知识和明确需求的用户，其需求包括：

1. **功能需求**：
   * 高级筛选功能
   * 详细的技术参数
   * 专业的对比分析
2. **体验需求**：
   * 精准的专业回答
   * 高效的信息获取
   * 深度的技术解析

#### 2.4.4 管理员

负责系统维护和管理的人员，其需求包括：

1. **功能需求**：
   * 用户管理
   * 商品管理
   * 订单管理
   * 系统监控
2. **体验需求**：
   * 高效的管理界面
   * 数据统计和分析
   * 便捷的操作流程

## 第3章 系统总体设计

### 3.1 系统架构设计

系统采用前后端分离的架构，以满足高可用和可扩展的需求。

#### 3.1.1 整体架构

系统的整体架构分为五层：

1. **表示层（前端）**：
   * 用户界面：包括Web界面和移动端适配界面
   * 前端交互：处理用户输入和界面响应
   * 前端缓存：优化用户体验和性能
2. **应用层（后端）**：
   * API网关：处理请求路由和负载均衡
   * 业务逻辑：实现核心功能和业务流程
   * 数据访问：与数据存储层交互
3. **智能服务层**：
   * AI模型服务：集成DeepSeek大语言模型
   * 推荐引擎：实现个性化推荐算法
   * 搜索引擎：支持多模态搜索功能
4. **数据存储层**：
   * 关系型数据库：存储结构化数据
   * NoSQL数据库：存储非结构化数据
   * 缓存系统：提高数据访问性能
5. **基础设施层**：
   * 服务器环境：支持系统运行的硬件和软件环境
   * 网络环境：确保系统连接和通信
   * 安全机制：保障系统和数据安全

#### 3.1.2 前端架构

前端采用基于React的单页应用（SPA）架构，主要组件包括：

1. **核心框架**：React.js作为核心UI框架
2. **状态管理**：React Context API管理应用状态
3. **路由管理**：React Router实现页面路由
4. **UI组件库**：自定义组件库，确保统一的用户体验
5. **API客户端**：负责与后端API通信
6. **AI交互模块**：处理与AI导购助手的实时对话

前端架构的特点是模块化和组件化，每个功能模块独立开发，通过组件组合形成完整页面。

#### 3.1.3 后端架构

后端采用基于Node.js的微服务架构，主要组件包括：

1. **API服务**：提供RESTful API供前端调用
2. **用户服务**：处理用户认证、授权和信息管理
3. **商品服务**：负责商品信息和库存管理
4. **订单服务**：处理订单创建、支付和状态管理
5. **AI集成服务**：与AI模型通信，处理自然语言理解和生成
6. **多模态服务**：处理图像、语音等多模态输入

后端架构采用了微服务设计理念，各服务之间通过API进行通信，保持松耦合，便于独立扩展和维护。

#### 3.1.4 AI服务架构

AI服务是本系统的核心创新点，其架构包括：

1. **模型接入层**：与DeepSeek大语言模型API集成
2. **上下文管理**：维护会话上下文和历史信息
3. **提示工程模块**：设计和优化AI提示策略
4. **知识增强模块**：补充领域知识和商品信息
5. **个性化适配层**：根据用户画像调整AI响应
6. **多模态处理**：支持图像识别和分析

AI服务架构采用了分层设计，确保与大语言模型的高效交互，同时支持业务逻辑的灵活扩展。

### 3.2 技术选型

系统的技术选型基于功能需求、非功能需求以及技术成熟度和生态系统支持等因素考虑。

#### 3.2.1 前端技术选型

前端开发采用以下技术栈：

1. **核心框架**：
   * **React.js**：选择React作为前端框架，因其组件化设计、虚拟DOM和丰富的生态系统
   * **React Router**：用于前端路由管理，实现单页应用导航
   * **React Context API**：用于状态管理，替代Redux简化开发
2. **UI与样式**：
   * **CSS-in-JS**：采用styled-components实现组件样式封装
   * **Tailwind CSS**：用于快速构建响应式UI
   * **FontAwesome**：提供丰富的图标资源
3. **功能组件**：
   * **Axios**：处理HTTP请求
   * **Socket.io-client**：实现实时通信功能
   * **React Markdown**：支持富文本和Markdown渲染
4. **开发工具**：
   * **Webpack**：模块打包
   * **Babel**：JavaScript编译器
   * **ESLint**：代码质量检查
   * **Prettier**：代码格式化

#### 3.2.2 后端技术选型

后端开发采用以下技术栈：

1. **运行环境与框架**：
   * **Node.js**：选择Node.js作为服务器环境，因其非阻塞I/O和事件驱动特性
   * **Express.js**：轻量级Web应用框架，用于构建RESTful API
   * **Axios**：用于服务器端HTTP请求
2. **数据存储**：
   * **MongoDB**：存储非结构化数据，如用户信息、商品详情等
   * **Redis**：用于缓存和会话管理
   * **Mongoose**：MongoDB对象模型工具
3. **API与通信**：
   * **RESTful API**：标准化API设计
   * **JWT**：用于身份验证和授权
   * **Socket.io**：实现实时通信
4. **开发工具**：
   * **Nodemon**：自动重启服务器
   * **Morgan**：HTTP请求日志记录
   * **Helmet**：增强API安全性
   * **Cors**：处理跨域资源共享

#### 3.2.3 AI技术选型

AI相关技术选型如下：

1. **大语言模型**：
   * **DeepSeek**：选择DeepSeek作为底层大语言模型，因其较强的理解能力和对中文的支持
   * **API集成方式**：采用RESTful API调用方式集成
2. **自然语言处理**：
   * **意图识别**：分析用户查询意图
   * **实体提取**：识别商品名称、属性等关键信息
   * **情感分析**：评估用户对推荐的满意度
3. **多模态处理**：
   * **图像识别**：识别用户上传图片中的商品
   * **文本生成**：生成商品描述和推荐理由
4. **推荐算法**：
   * **协同过滤**：基于用户行为的推荐
   * **内容推荐**：基于商品属性的推荐
   * **上下文感知推荐**：考虑对话上下文的推荐

### 3.3 系统模块设计

根据系统功能需求，将系统划分为多个功能模块，各模块之间相互协作，共同实现系统功能。

#### 3.3.1 模块划分

系统主要分为以下几个功能模块：

1. **用户管理模块**：
   * 用户注册与登录
   * 用户信息管理
   * 权限控制
   * 用户行为跟踪
2. **商品管理模块**：
   * 商品分类管理
   * 商品信息管理
   * 商品库存管理
   * 商品价格管理
3. **订单管理模块**：
   * 购物车管理
   * 订单生成与处理
   * 支付集成
   * 订单状态跟踪
4. **AI导购模块**：
   * 对话管理
   * 需求分析
   * 商品推荐
   * 专业解答
   * 沙漏流程管理
5. **搜索模块**：
   * 文本搜索
   * 图像搜索
   * 语音搜索
   * 多模态融合搜索
6. **推荐系统模块**：
   * 用户画像构建
   * 商品特征提取
   * 推荐算法实现
   * 推荐效果评估
7. **数据分析模块**：
   * 用户行为分析
   * 销售数据分析
   * AI导购效果分析
   * 数据可视化
8. **系统管理模块**：
   * 系统配置管理
   * 日志管理
   * 性能监控
   * 安全管理

#### 3.3.2 模块交互

模块间的主要交互关系如下：

|  |
| --- |
| 模块交互图 |

模块交互图

1. **用户界面与各模块的交互**：
   * 用户通过界面与各功能模块交互
   * 界面展示各模块处理结果
   * 收集用户反馈并传递给相应模块
2. **AI导购模块与其他模块的交互**：
   * 与用户管理模块交互获取用户信息
   * 与商品管理模块交互获取商品信息
   * 与推荐系统模块交互获取推荐结果
   * 与搜索模块交互执行搜索操作
3. **推荐系统与其他模块的交互**：
   * 与用户管理模块交互获取用户行为数据
   * 与商品管理模块交互获取商品特征
   * 与AI导购模块交互提供上下文感知推荐
4. **订单管理与其他模块的交互**：
   * 与用户管理模块交互验证用户身份
   * 与商品管理模块交互更新库存
   * 与支付系统交互处理支付

#### 3.3.3 关键模块详细设计

1. **AI导购模块设计**：

* AI导购模块是系统的核心，其内部结构包括：
  + **对话管理器**：管理用户与AI之间的对话流程
  + **上下文管理器**：维护对话上下文和历史信息
  + **意图识别器**：分析用户的意图和需求
  + **沙漏阶段控制器**：控制导购流程的阶段转换
  + **提示词生成器**：根据当前阶段生成适当的提示词
  + **响应生成器**：生成AI回复并格式化
  + **商品推荐集成器**：整合推荐系统的结果
* AI导购模块的工作流程如下：
  1. 接收用户输入
  2. 分析用户意图和需求
  3. 维护对话上下文
  4. 判断当前沙漏阶段
  5. 生成适当的提示词
  6. 调用大语言模型获取响应
  7. 处理响应并返回给用户

1. **推荐系统模块设计**：

* 推荐系统模块负责个性化商品推荐，其内部结构包括：
  + **数据收集器**：收集用户行为和产品数据
  + **用户画像构建器**：基于用户行为构建画像
  + **商品特征提取器**：提取商品的关键特征
  + **相似度计算器**：计算用户与商品的匹配度
  + **推荐策略选择器**：选择适合的推荐算法
  + **结果排序器**：对推荐结果进行排序
  + **推荐效果评估器**：评估推荐的效果
* 推荐系统的工作流程如下：
  1. 收集用户历史行为和实时行为
  2. 更新用户画像
  3. 根据用户画像和当前上下文选择推荐策略
  4. 计算商品与用户的匹配度
  5. 排序并生成推荐列表
  6. 评估并记录推荐效果

1. **多模态搜索模块设计**：

* 多模态搜索模块支持多种输入方式的商品搜索，其内部结构包括：
  + **文本处理器**：处理文本搜索查询
  + **图像处理器**：处理图像搜索查询
  + **语音处理器**：处理语音搜索查询
  + **模态融合器**：融合不同模态的搜索结果
  + **相关性排序器**：对搜索结果进行排序
  + **搜索增强器**：通过AI增强搜索结果的相关性
* 多模态搜索的工作流程如下：
  1. 接收用户多模态输入
  2. 根据输入类型调用相应处理器
  3. 提取关键特征和搜索关键词
  4. 执行搜索并获取初步结果
  5. 融合多模态结果并排序
  6. 返回最终的搜索结果列表

### 3.4 数据库设计

系统采用关系型数据库和NoSQL数据库相结合的方式，以满足不同类型数据的存储需求。

#### 3.4.1 数据库选型

系统数据库选型如下：

1. **MongoDB**：
   * 用于存储非结构化和半结构化数据
   * 存储对象：用户信息、商品详情、对话历史等
   * 优势：灵活的Schema，适合存储变化的数据结构
2. **Redis**：
   * 用于缓存和临时数据存储
   * 存储对象：会话信息、热门商品、缓存数据等
   * 优势：高性能、支持多种数据结构、适合实时数据

#### 3.4.2 核心数据模型设计

系统的核心数据模型包括：

|  |
| --- |
| 数据模型关系图 |

数据模型关系图

1. **用户模型**：

* User {  
   id: ObjectId,  
   username: String,  
   email: String,  
   password: String(加密),  
   profile: {  
   name: String,  
   avatar: String,  
   gender: String,  
   birthday: Date,  
   ...  
   },  
   preferences: {  
   categories: [String],  
   brands: [String],  
   priceRange: {min: Number, max: Number},  
   ...  
   },  
   createAt: Date,  
   updateAt: Date  
  }

1. **商品模型**：

* Product {  
   id: ObjectId,  
   name: String,  
   category: String,  
   brand: String,  
   price: Number,  
   discount: Number,  
   stock: Number,  
   images: [String],  
   description: String,  
   specifications: Object,  
   attributes: Object,  
   rating: Number,  
   reviews: [ObjectId],  
   createAt: Date,  
   updateAt: Date  
  }

1. **订单模型**：

* Order {  
   id: ObjectId,  
   userId: ObjectId,  
   items: [{  
   productId: ObjectId,  
   name: String,  
   price: Number,  
   quantity: Number  
   }],  
   totalPrice: Number,  
   status: String,  
   paymentMethod: String,  
   shippingAddress: Object,  
   createAt: Date,  
   updateAt: Date  
  }

1. **对话历史模型**：

* Conversation {  
   id: ObjectId,  
   userId: ObjectId,  
   messages: [{  
   role: String,  
   content: String,  
   timestamp: Date  
   }],  
   stage: String,  
   relatedProducts: [ObjectId],  
   createAt: Date,  
   updateAt: Date  
  }

1. **用户行为模型**：

* UserBehavior {  
   id: ObjectId,  
   userId: ObjectId,  
   type: String,  
   data: Object,  
   timestamp: Date  
  }

#### 3.4.3 数据库优化策略

为保证系统性能，采取以下数据库优化策略：

1. **索引优化**：
   * 为频繁查询的字段创建索引
   * 使用复合索引提高查询效率
   * 定期维护索引以保持性能
2. **数据分片**：
   * 针对大规模数据实施分片策略
   * 基于用户ID或时间范围进行分片
   * 确保分片均衡，避免热点问题
3. **缓存策略**：
   * 使用Redis缓存热门商品数据
   * 缓存常用查询结果
   * 实施合理的缓存失效策略
4. **数据库连接池**：
   * 管理数据库连接资源
   * 优化连接池参数
   * 监控连接状态和性能

### 3.5 “沙漏模型”AI导购流程设计

本系统的核心创新点是提出并实现了”沙漏模型”AI导购流程，该模型模拟了专业导购过程，旨在降低用户认知负担，提高购物决策效率。

#### 3.5.1 沙漏模型概述

“沙漏模型”将购物决策过程分为四个阶段，形成一个”沙漏”形状的流程：

|  |
| --- |
| 沙漏模型示意图 |

沙漏模型示意图

1. **需求表达阶段（沙漏上部）**：
   * 特点：开放性对话，广泛收集用户需求
   * 目标：了解用户的基本需求、使用场景和偏好
   * 交互方式：自然语言对话，开放性问题
2. **参数筛选阶段（沙漏颈部上段）**：
   * 特点：聚焦于具体参数和规格
   * 目标：确定关键筛选条件，缩小选择范围
   * 交互方式：结构化问题，参数确认
3. **对比选择阶段（沙漏颈部下段）**：
   * 特点：详细比较少数候选产品
   * 目标：帮助用户理解产品差异，权衡利弊
   * 交互方式：结构化对比，优劣分析
4. **决策推荐阶段（沙漏下部）**：
   * 特点：扩展决策考量，提供最终建议
   * 目标：消除决策疑虑，增强购买信心
   * 交互方式：个性化推荐，决策支持

这种”收缩-聚焦-扩展”的流程设计，符合用户的认知过程和决策路径，能够有效降低选择困难和认知负担。

#### 3.5.2 各阶段设计详情

**1. 需求表达阶段设计**

在需求表达阶段，系统主要通过开放性对话，引导用户表达购物需求。设计要点包括：

* **对话引导策略**：
  + 使用开放性问题鼓励用户自由表达
  + 探索性提问了解使用场景和预期目标
  + 建立融洽关系，增强用户信任感
* **信息收集重点**：
  + 产品类别（如手机、笔记本等）
  + 用途和使用场景
  + 大致预算范围
  + 特殊需求或偏好
* **用户意图识别**：
  + 分析用户表述中的关键词和实体
  + 识别明显和隐含的需求
  + 确定用户的知识水平和专业程度

示例对话：

用户: "我想买一个新手机"  
AI: "很高兴帮您选择新手机。能告诉我您主要用手机做什么？比如拍照、游戏或者办公？"  
  
用户: "主要是拍照，我喜欢旅行，希望能拍出好看的照片"  
AI: "了解了，拍照是您的主要需求。您对手机的预算大概是什么范围呢？另外，有没有特别喜欢或不喜欢的品牌？"

**2. 参数筛选阶段设计**

参数筛选阶段通过结构化交互，确定具体的筛选条件。设计要点包括：

* **参数确认策略**：
  + 提出具体、封闭性问题
  + 提供选项范围，降低用户思考负担
  + 解释专业术语，提供参数意义说明
* **核心参数筛选**：
  + 价格区间（精确预算）
  + 核心功能参数（如手机的处理器、内存）
  + 使用偏好相关参数（如拍照手机的摄像头规格）
  + 外观和设计偏好
* **参数优先级处理**：
  + 识别用户最关注的参数
  + 处理参数之间的冲突（如高性能与低价格）
  + 提供平衡建议

示例对话：

AI: "基于您的需求，我们需要关注手机的摄像头性能。您更看重广角拍摄还是夜景拍摄？"  
  
用户: "两者都重要，但可能夜景更重要一些"  
AI: "明白了。对于预算3000-5000元且夜景拍摄出色的手机，目前市场上有几款值得考虑。您对手机的电池续航有什么要求吗？"

**3. 对比选择阶段设计**

对比选择阶段聚焦于少数候选产品的详细比较。设计要点包括：

* **对比展示策略**：
  + 提供结构化的对比信息
  + 突出显示产品间的差异点
  + 针对用户关注点进行重点对比
* **对比内容组织**：
  + 基本信息对比（价格、品牌、型号）
  + 核心参数对比（性能指标、功能特点）
  + 优缺点分析
  + 适用场景说明
* **专业解读提供**：
  + 解释技术参数的实际意义
  + 分析参数差异对体验的影响
  + 提供专业评测见解

示例对话：

AI: "根据您的需求，我推荐这三款手机：华为P40、小米11 Ultra和iPhone 12。下面是它们的主要对比：  
  
1. 摄像头：  
 - 华为P40: 5000万像素主摄，拥有徕卡认证，夜景表现极佳  
 - 小米11 Ultra: 5000万像素三摄，搭配50倍变焦，全能型拍照  
 - iPhone 12: 1200万像素双摄，计算摄影能力强，色彩还原自然  
  
2. 性能：  
 - 华为P40: 麒麟990处理器，性能强劲但受制裁影响应用生态受限  
 - 小米11 Ultra: 骁龙888，安卓阵营顶级性能  
 - iPhone 12: A14仿生芯片，性能领先，优化出色  
  
您对哪款更感兴趣？或者想了解更多细节？"

**4. 决策推荐阶段设计**

决策推荐阶段帮助用户做出最终选择并增强购买信心。设计要点包括：

* **推荐策略**：
  + 基于用户需求和偏好给出明确建议
  + 解释推荐理由，增强说服力
  + 提供备选方案，避免单一推荐
* **决策支持**：
  + 解答用户最后的疑虑
  + 提供用户评价和专业评测信息
  + 分析性价比和长期使用价值
* **购后服务预告**：
  + 介绍保修和售后政策
  + 提供使用建议和注意事项
  + 承诺持续的购后支持

示例对话：

用户: "这几款都不错，你觉得哪个最适合我？"  
AI: "考虑到您注重夜景拍摄且经常旅行，我推荐华为P40。原因有三：  
  
1. 它的徕卡认证摄像头在夜景拍摄方面表现出色，旅行场景下能捕捉更多细节  
2. 电池续航能力强，适合旅行时长时间使用  
3. 它的防水等级达到IP68，旅行中更加安心  
  
不过需要注意的是，由于制裁原因，它不支持Google服务。如果您经常使用Google应用，可以考虑小米11 Ultra作为替代选择。  
  
您对这个推荐满意吗？或者还有其他考虑因素？"

#### 3.5.3 阶段转换策略

沙漏模型的阶段转换需要智能判断和平滑过渡，避免生硬的跳转体验。

**1. 阶段判断机制**

系统通过以下方式判断当前所处的阶段：

* **对话内容分析**：分析用户表达的具体程度和专业程度
* **已获取信息评估**：评估已获取的用户需求信息是否足够进入下一阶段
* **用户行为信号**：通过用户的响应方式判断其就绪程度
* **对话轮次计数**：考虑对话的进行程度

**2. 阶段转换触发条件**

各阶段的转换条件设计如下：

* **需求表达→参数筛选**：
  + 已获取基本需求和使用场景
  + 用户表达了明确的产品类别
  + 对话已进行2-3轮
* **参数筛选→对比选择**：
  + 已获取关键参数（如预算、功能需求）
  + 筛选后的商品数量≤5
  + 用户表达了希望看到具体商品
* **对比选择→决策推荐**：
  + 用户已经表达了对特定商品的兴趣
  + 用户提出了关于”哪个更好”的问题
  + 对话已深入到具体商品细节

**3. 平滑过渡策略**

为了使阶段转换自然，采用以下策略：

* **总结前阶段信息**：在转换前总结已获取的信息
* **解释下一步操作**：说明接下来将做什么
* **征求用户同意**：在重要转换点征求用户意见
* **保持对话连贯性**：避免生硬的话题转换

示例转换话术：

*从需求表达到参数筛选*： “非常感谢您分享了使用场景。基于您的需求，接下来我想了解一些具体参数，这样可以帮您筛选出最合适的产品。首先，您对预算有什么要求吗？”

*从参数筛选到对比选择*： “根据您提供的这些信息，我已经找到了几款符合您需求的产品。接下来我可以为您详细对比这些产品的优缺点，帮助您做出选择。”

*从对比选择到决策推荐*： “通过对这几款产品的对比，并考虑到您特别看重的点，我认为有一款产品可能最适合您。”

#### 3.5.4 沙漏模型优化设计

为进一步提升沙漏模型的效果，设计了以下优化机制：

**1. 上下文记忆增强**

* **长期记忆**：跨会话保存用户偏好和需求
* **短期记忆**：当前会话内的对话历史
* **选择性遗忘**：过滤无关或过时信息

**2. 阶段回退机制**

允许用户在需要时回到之前的阶段：

* **显式回退**：用户明确表示想要重新考虑
* **隐式回退**：系统检测到用户需求变化
* **部分回退**：仅重新考虑特定参数而不是全部

**3. 多模态交互增强**

在沙漏模型各阶段融入多模态交互：

* **需求表达阶段**：支持图片输入表达需求
* **参数筛选阶段**：可视化参数调整界面
* **对比选择阶段**：可视化对比表格和图片
* **决策推荐阶段**：多媒体产品展示

**4. 个性化调整**

根据用户特征调整沙漏模型：

* **专业用户**：可能跳过部分阶段，直接进入参数筛选
* **新手用户**：提供更多解释和引导
* **回头客**：基于历史偏好加速决策流程
* **浏览型用户**：提供更多探索性选项

通过以上设计，“沙漏模型”AI导购流程能够有效模拟专业导购的工作方式，从用户的模糊需求出发，逐步引导用户做出明确的购买决策，大幅提升购物体验和决策质量。

## 第4章 系统详细设计与实现

本章详细介绍系统的设计与实现，包括前后端开发、核心功能模块实现以及AI导购助手的具体实现。

### 4.1 开发环境与工具

#### 4.1.1 硬件环境

* 处理器：Intel Core i7-12700H
* 内存：16GB DDR4
* 存储：512GB NVMe SSD
* 操作系统：Windows 11 专业版

#### 4.1.2 软件环境与工具

* **前端开发**：
  + Node.js v16.14.0
  + React.js v18.2.0
  + Visual Studio Code 1.74
  + npm v8.3.1
* **后端开发**：
  + Node.js v16.14.0
  + Express.js v4.18.2
  + MongoDB v6.0
  + Mongoose v7.0.3
* **版本控制**：
  + Git v2.39.2
  + GitHub
* **API测试**：
  + Postman v10.13
  + Thunder Client (VS Code插件)
* **AI模型接入**：
  + DeepSeek API
  + Axios v1.3.4

### 4.2 系统前端设计与实现

系统前端基于React.js框架开发，采用组件化和模块化的设计理念。

#### 4.2.1 前端架构实现

前端项目结构如下：

src/  
├── assets/ # 静态资源文件  
├── components/ # 通用组件  
│ ├── common/ # 公共UI组件  
│ ├── layout/ # 布局组件  
│ └── features/ # 功能组件  
├── pages/ # 页面组件  
├── services/ # API服务  
├── utils/ # 工具函数  
├── hooks/ # 自定义hooks  
├── context/ # 上下文管理  
├── styles/ # 全局样式  
├── App.js # 应用入口  
└── index.js # 渲染入口

采用的主要设计模式包括：

1. **组件组合模式**：通过组合小组件构建复杂界面
2. **容器-展示组件模式**：分离数据逻辑和UI渲染
3. **Context模式**：使用React Context API管理全局状态
4. **自定义Hook模式**：封装复用逻辑

#### 4.2.2 关键页面设计与实现

**1. 首页设计与实现**

首页是用户进入系统的门户，设计遵循简洁直观的原则。

主要组件构成： - 顶部导航栏（Navbar） - 搜索栏（SearchBar） - AI助手入口（AIAssistant） - 商品分类导航（CategoryNav） - 商品推荐区（ProductRecommendation） - 底部信息栏（Footer）

关键代码实现：

// HomePage.jsx  
import React, { useEffect, useState } from 'react';  
import Navbar from '../components/layout/Navbar';  
import SearchBar from '../components/features/SearchBar';  
import AIAssistant from '../components/features/AIAssistant';  
import CategoryNav from '../components/features/CategoryNav';  
import ProductRecommendation from '../components/features/ProductRecommendation';  
import Footer from '../components/layout/Footer';  
import { fetchFeaturedProducts, fetchCategories } from '../services/api';  
  
const HomePage = () => {  
 const [featuredProducts, setFeaturedProducts] = useState([]);  
 const [categories, setCategories] = useState([]);  
   
 useEffect(() => {  
 const loadInitialData = async () => {  
 try {  
 const productsData = await fetchFeaturedProducts();  
 const categoriesData = await fetchCategories();  
   
 setFeaturedProducts(productsData);  
 setCategories(categoriesData);  
 } catch (error) {  
 console.error('Failed to load initial data:', error);  
 }  
 };  
   
 loadInitialData();  
 }, []);  
   
 return (  
 <div className="homepage">  
 <Navbar />  
 <div className="main-content">  
 <div className="search-assistant-container">  
 <SearchBar />  
 <AIAssistant />  
 </div>  
 <CategoryNav categories={categories} />  
 <ProductRecommendation products={featuredProducts} />  
 </div>  
 <Footer />  
 </div>  
 );  
};  
  
export default HomePage;

**2. AI导购助手对话页面设计与实现**

AI导购助手对话页面是本系统的核心功能页面，提供用户与AI导购助手的交互界面。

主要组件构成： - 对话历史显示区（ChatHistory） - 用户输入区（ChatInput） - 商品推荐展示区（ProductSuggestions） - 对话阶段指示器（StageIndicator） - 多模态输入工具栏（MultimodalTools）

关键代码实现：

// AIChatPage.jsx  
import React, { useState, useEffect, useRef } from 'react';  
import ChatHistory from '../components/features/ChatHistory';  
import ChatInput from '../components/features/ChatInput';  
import ProductSuggestions from '../components/features/ProductSuggestions';  
import StageIndicator from '../components/features/StageIndicator';  
import MultimodalTools from '../components/features/MultimodalTools';  
import { sendMessage, uploadImage } from '../services/aiService';  
import { useChatContext } from '../context/ChatContext';  
  
const AIChatPage = () => {  
 const {   
 messages,   
 addMessage,   
 suggestedProducts,  
 setSuggestedProducts,   
 currentStage,  
 setCurrentStage  
 } = useChatContext();  
   
 const [isLoading, setIsLoading] = useState(false);  
 const chatEndRef = useRef(null);  
   
 const handleSendMessage = async (text) => {  
 if (!text.trim()) return;  
   
 // Add user message to chat  
 addMessage({ role: 'user', content: text });  
 setIsLoading(true);  
   
 try {  
 // Send message to AI service  
 const response = await sendMessage(text, messages);  
   
 // Add AI response to chat  
 addMessage({ role: 'assistant', content: response.message });  
   
 // Update suggested products if any  
 if (response.products && response.products.length > 0) {  
 setSuggestedProducts(response.products);  
 }  
   
 // Update current stage if changed  
 if (response.stage && response.stage !== currentStage) {  
 setCurrentStage(response.stage);  
 }  
 } catch (error) {  
 console.error('Failed to send message:', error);  
 addMessage({   
 role: 'system',   
 content: '抱歉，发生了错误，请稍后再试。'   
 });  
 } finally {  
 setIsLoading(false);  
 }  
 };  
   
 const handleImageUpload = async (imageFile) => {  
 setIsLoading(true);  
   
 try {  
 const response = await uploadImage(imageFile);  
   
 addMessage({   
 role: 'user',   
 content: '上传了一张图片',   
 image: response.imageUrl   
 });  
   
 addMessage({   
 role: 'assistant',   
 content: response.analysis   
 });  
   
 if (response.products && response.products.length > 0) {  
 setSuggestedProducts(response.products);  
 }  
 } catch (error) {  
 console.error('Failed to upload image:', error);  
 addMessage({   
 role: 'system',   
 content: '图片上传失败，请稍后再试。'   
 });  
 } finally {  
 setIsLoading(false);  
 }  
 };  
   
 // Scroll to bottom when messages update  
 useEffect(() => {  
 chatEndRef.current?.scrollIntoView({ behavior: 'smooth' });  
 }, [messages]);  
   
 return (  
 <div className="ai-chat-page">  
 <div className="chat-container">  
 <StageIndicator currentStage={currentStage} />  
 <ChatHistory   
 messages={messages}   
 isLoading={isLoading}   
 />  
 <div ref={chatEndRef} />  
 <MultimodalTools onImageUpload={handleImageUpload} />  
 <ChatInput   
 onSendMessage={handleSendMessage}   
 disabled={isLoading}   
 />  
 </div>  
 {suggestedProducts.length > 0 && (  
 <ProductSuggestions products={suggestedProducts} />  
 )}  
 </div>  
 );  
};  
  
export default AIChatPage;

**3. 商品详情页设计与实现**

商品详情页展示商品的详细信息，并提供购买和咨询功能。

主要组件构成： - 商品图片展示（ProductGallery） - 商品基本信息（ProductInfo） - 商品规格参数（ProductSpecs） - 用户评价（ProductReviews） - 相关商品推荐（RelatedProducts） - AI助手咨询按钮（AskAIButton）

#### 4.2.3 核心组件设计与实现

**1. AI对话组件（ChatHistory）**

该组件负责显示用户与AI助手的对话历史。

关键代码实现：

// ChatHistory.jsx  
import React from 'react';  
import './ChatHistory.css';  
import UserMessage from './UserMessage';  
import AIMessage from './AIMessage';  
import SystemMessage from './SystemMessage';  
import LoadingIndicator from '../common/LoadingIndicator';  
  
const ChatHistory = ({ messages, isLoading }) => {  
 return (  
 <div className="chat-history">  
 {messages.length === 0 ? (  
 <div className="empty-chat">  
 <h3>欢迎使用AI购物助手</h3>  
 <p>请告诉我您想购买什么，我会帮您找到最合适的商品。</p>  
 <p>您也可以上传商品图片，我会分析并推荐相似商品。</p>  
 </div>  
 ) : (  
 messages.map((msg, index) => {  
 switch (msg.role) {  
 case 'user':  
 return <UserMessage key={index} message={msg} />;  
 case 'assistant':  
 return <AIMessage key={index} message={msg} />;  
 case 'system':  
 return <SystemMessage key={index} message={msg} />;  
 default:  
 return null;  
 }  
 })  
 )}  
 {isLoading && <LoadingIndicator />}  
 </div>  
 );  
};  
  
export default ChatHistory;

**2. 多模态搜索组件（MultimodalSearch）**

该组件支持文本、语音和图像多种搜索方式。

关键代码实现：

// MultimodalSearch.jsx  
import React, { useState } from 'react';  
import { BsSearch, BsMic, BsCamera } from 'react-icons/bs';  
import './MultimodalSearch.css';  
  
const MultimodalSearch = ({ onSearch, onImageSearch, onVoiceSearch }) => {  
 const [query, setQuery] = useState('');  
 const [isListening, setIsListening] = useState(false);  
 const [searchMode, setSearchMode] = useState('text'); // 'text', 'voice', 'image'  
   
 const handleTextSearch = (e) => {  
 e.preventDefault();  
 if (query.trim()) {  
 onSearch(query);  
 }  
 };  
   
 const handleImageUpload = (e) => {  
 const file = e.target.files[0];  
 if (file) {  
 onImageSearch(file);  
 }  
 };  
   
 const startVoiceRecognition = () => {  
 if ('webkitSpeechRecognition' in window) {  
 const recognition = new window.webkitSpeechRecognition();  
 recognition.lang = 'zh-CN';  
 recognition.continuous = false;  
   
 setIsListening(true);  
 setSearchMode('voice');  
   
 recognition.onresult = (event) => {  
 const transcript = event.results[0][0].transcript;  
 setQuery(transcript);  
 onVoiceSearch(transcript);  
 };  
   
 recognition.onerror = (event) => {  
 console.error('Speech recognition error', event.error);  
 setIsListening(false);  
 };  
   
 recognition.onend = () => {  
 setIsListening(false);  
 setSearchMode('text');  
 };  
   
 recognition.start();  
 } else {  
 alert('您的浏览器不支持语音识别功能');  
 }  
 };  
   
 return (  
 <div className="multimodal-search">  
 <form onSubmit={handleTextSearch} className="search-form">  
 <div className="search-input-wrapper">  
 <input  
 type="text"  
 value={query}  
 onChange={(e) => setQuery(e.target.value)}  
 placeholder="搜索商品..."  
 className="search-input"  
 />  
   
 <div className="search-buttons">  
 <button type="submit" className="search-btn">  
 <BsSearch />  
 </button>  
   
 <button   
 type="button"   
 className={`voice-btn ${isListening ? 'active' : ''}`}  
 onClick={startVoiceRecognition}  
 >  
 <BsMic />  
 </button>  
   
 <label className="image-btn">  
 <BsCamera />  
 <input  
 type="file"  
 accept="image/\*"  
 onChange={handleImageUpload}  
 style={{ display: 'none' }}  
 />  
 </label>  
 </div>  
 </div>  
 </form>  
   
 {searchMode === 'voice' && isListening && (  
 <div className="voice-indicator">  
 正在聆听...请说出您要搜索的内容  
 </div>  
 )}  
 </div>  
 );  
};  
  
export default MultimodalSearch;

**3. 沙漏流程状态指示器（HourglassIndicator）**

该组件以可视化方式显示AI导购助手的沙漏模型当前所处阶段。

关键代码实现：

// HourglassIndicator.jsx  
import React from 'react';  
import './HourglassIndicator.css';  
  
const stages = [  
 { id: 'need-expression', name: '需求表达', description: '告诉我您的需求' },  
 { id: 'parameter-filtering', name: '参数筛选', description: '确定关键参数' },  
 { id: 'comparison', name: '对比选择', description: '比较候选商品' },  
 { id: 'decision', name: '决策推荐', description: '最终推荐' }  
];  
  
const HourglassIndicator = ({ currentStage }) => {  
 return (  
 <div className="hourglass-indicator">  
 <div className="stage-indicator">  
 {stages.find(stage => stage.id === currentStage)?.name}  
 </div>  
 <div className="stage-description">  
 {stages.find(stage => stage.id === currentStage)?.description}  
 </div>  
 </div>  
 );  
};  
  
export default HourglassIndicator;

### 4.3 系统后端设计与实现

系统后端采用基于Node.js和Express框架的微服务架构，实现高可扩展性和模块化。整体架构设计如下图所示：

|  |
| --- |
| 后端架构设计图 |

后端架构设计图

后端架构主要分为五层：

1. **API网关层**：
   * 请求路由：将请求分发到对应的微服务
   * 认证鉴权：统一处理用户身份验证
   * 限流熔断：保护系统不被过载
   * 日志监控：记录请求信息
2. **微服务层**：
   * 用户服务：管理用户信息、认证授权
   * 商品服务：商品信息管理和搜索
   * 订单服务：订单流程和支付管理
   * AI服务：AI导购和推荐服务
   * 搜索服务：全文和多模态搜索
   * 推荐服务：个性化推荐引擎
3. **中间件层**：
   * 消息队列：处理异步任务和服务通信
   * 缓存服务：提高频繁访问数据的响应速度
   * 日志服务：集中存储和分析系统日志
   * 配置中心：统一管理服务配置
4. **数据访问层**：
   * 数据库访问：封装数据库操作
   * ORM/ODM：对象关系映射
   * 数据验证：确保数据一致性和完整性
   * 查询优化：优化数据库查询性能
5. **存储层**：
   * MongoDB：主要数据存储
   * Redis：缓存和会话存储
   * MinIO/S3：文件和图片存储
   * 时序数据库：存储监控和分析数据

这种分层架构具有以下优势： - 服务解耦：各服务独立开发和部署 - 技术灵活性：不同服务可选择最合适的技术栈 - 可扩展性：按需扩展特定服务 - 故障隔离：单一服务故障不影响整体系统

#### 4.2.2 数据库设计

系统采用MongoDB作为主要数据库，Redis作为缓存和会话存储。

**1. MongoDB数据模型**

MongoDB的文档模型适合存储复杂结构的数据，主要集合设计如下：

* **用户集合（Users）**：
* {  
   "\_id": "ObjectId",  
   "username": "String",  
   "email": "String",  
   "passwordHash": "String",  
   "profile": {  
   "name": "String",  
   "avatar": "String",  
   "gender": "String",  
   "birthday": "Date"  
   },  
   "preferences": {  
   "categories": ["String"],  
   "brands": ["String"],  
   "priceRange": { "min": "Number", "max": "Number" }  
   },  
   "shoppingHistory": ["ObjectId"],  
   "createdAt": "Date",  
   "updatedAt": "Date"  
  }
* **商品集合（Products）**：
* {  
   "\_id": "ObjectId",  
   "name": "String",  
   "category": "String",  
   "subCategory": "String",  
   "brand": "String",  
   "price": "Number",  
   "discount": "Number",  
   "stock": "Number",  
   "images": ["String"],  
   "description": "String",  
   "specifications": { "key": "value" },  
   "features": ["String"],  
   "ratings": {  
   "average": "Number",  
   "count": "Number"  
   },  
   "keywords": ["String"],  
   "createdAt": "Date",  
   "updatedAt": "Date"  
  }
* **订单集合（Orders）**：
* {  
   "\_id": "ObjectId",  
   "userId": "ObjectId",  
   "products": [  
   {  
   "productId": "ObjectId",  
   "quantity": "Number",  
   "price": "Number"  
   }  
   ],  
   "totalPrice": "Number",  
   "status": "String",  
   "paymentMethod": "String",  
   "shippingAddress": {  
   "name": "String",  
   "phone": "String",  
   "address": "String",  
   "city": "String",  
   "zipCode": "String"  
   },  
   "paymentStatus": "String",  
   "orderStatus": "String",  
   "createdAt": "Date",  
   "updatedAt": "Date"  
  }
* **对话历史集合（Conversations）**：
* {  
   "\_id": "ObjectId",  
   "userId": "ObjectId",  
   "sessionId": "String",  
   "messages": [  
   {  
   "role": "String",  
   "content": "String",  
   "timestamp": "Date",  
   "attachments": [  
   {  
   "type": "String",  
   "url": "String"  
   }  
   ]  
   }  
   ],  
   "context": {  
   "stage": "String",  
   "preferences": {"key": "value"},  
   "productIds": ["ObjectId"]  
   },  
   "createdAt": "Date",  
   "updatedAt": "Date"  
  }
* **用户行为集合（UserBehaviors）**：
* {  
   "\_id": "ObjectId",  
   "userId": "ObjectId",  
   "sessionId": "String",  
   "behaviors": [  
   {  
   "type": "String",  
   "target": "ObjectId",  
   "timestamp": "Date",  
   "details": {"key": "value"}  
   }  
   ],  
   "createdAt": "Date",  
   "updatedAt": "Date"  
  }

**2. Redis存储设计**

Redis主要用于以下场景：

* **会话管理**：
  + 键格式：session:{sessionId}
  + 值类型：Hash，存储会话数据
  + 过期时间：30分钟，用户活动后续期
* **频繁访问数据缓存**：
  + 热门商品：hotProducts:{category}
  + 用户购物车：cart:{userId}
  + 商品库存：stock:{productId}
* **限流控制**：
  + API请求计数：rateLimit:{ip}:{endpoint}
  + 登录尝试次数：loginAttempts:{userId}
* **实时分析**：
  + 页面访问统计：pageViews:{page}:{date}
  + 搜索热词：searchHotwords:{date}

#### 4.2.3 API设计

系统API遵循RESTful设计原则，主要分为以下几类：

**1. 用户相关API**

| 端点 | 方法 | 描述 |
| --- | --- | --- |
| /api/users/register | POST | 用户注册 |
| /api/users/login | POST | 用户登录 |
| /api/users/profile | GET | 获取用户信息 |
| /api/users/profile | PUT | 更新用户信息 |
| /api/users/preferences | GET | 获取用户偏好 |
| /api/users/preferences | PUT | 更新用户偏好 |

**2. 商品相关API**

| 端点 | 方法 | 描述 |
| --- | --- | --- |
| /api/products | GET | 获取商品列表 |
| /api/products/:id | GET | 获取商品详情 |
| /api/products/categories | GET | 获取商品分类 |
| /api/products/search | GET | 搜索商品 |
| /api/products/recommendations | GET | 获取推荐商品 |

**3. 订单相关API**

| 端点 | 方法 | 描述 |
| --- | --- | --- |
| /api/orders | POST | 创建订单 |
| /api/orders | GET | 获取订单列表 |
| /api/orders/:id | GET | 获取订单详情 |
| /api/orders/:id/status | PUT | 更新订单状态 |
| /api/cart | GET | 获取购物车 |
| /api/cart | POST | 添加商品到购物车 |

**4. AI导购相关API**

| 端点 | 方法 | 描述 |
| --- | --- | --- |
| /api/assistant/chat | POST | 发送消息 |
| /api/assistant/chat/:sessionId | GET | 获取历史对话 |
| /api/assistant/upload | POST | 上传图片 |
| /api/assistant/compare | POST | 请求商品对比 |
| /api/assistant/stage | GET | 获取当前阶段 |

**5. 搜索相关API**

| 端点 | 方法 | 描述 |
| --- | --- | --- |
| /api/search/text | GET | 文本搜索 |
| /api/search/image | POST | 图像搜索 |
| /api/search/voice | POST | 语音搜索 |
| /api/search/suggestions | GET | 获取搜索建议 |

API规范遵循以下原则： - 使用HTTP标准方法表示操作类型 - 使用URL路径表示资源 - 使用查询参数进行过滤和分页 - 统一的响应格式 - 适当的HTTP状态码表示请求结果

#### 4.2.4 身份认证与安全设计

系统安全性设计包含多层次防护策略：

**1. 身份认证机制**

采用基于JWT（JSON Web Token）的认证方案：

* **登录流程**：
  1. 用户提交凭证（用户名/密码）
  2. 服务器验证凭证
  3. 生成JWT令牌，包含用户ID和角色
  4. 返回令牌给客户端
* **认证流程**：
  1. 客户端请求携带令牌（Authorization头）
  2. 服务器验证令牌有效性和签名
  3. 解析令牌获取用户信息
  4. 进行访问控制决策
* **令牌管理**：
  + 访问令牌：短期有效（15分钟）
  + 刷新令牌：长期有效（7天）
  + 令牌刷新机制：使用刷新令牌获取新的访问令牌

**2. 授权与访问控制**

基于RBAC（基于角色的访问控制）模型：

* **角色设计**：
  + 普通用户：基本购物和查询权限
  + VIP用户：额外优惠和高级功能权限
  + 管理员：系统管理权限
* **权限检查**：
  + API级别：中间件验证请求权限
  + 资源级别：检查用户与资源的关系
  + 数据级别：过滤用户可访问的数据

**3. 数据安全**

* **敏感数据保护**：
  + 密码存储：使用bcrypt加盐哈希
  + 个人信息：加密存储敏感字段
  + 支付信息：使用第三方支付接口，不存储完整信息
* **数据传输安全**：
  + HTTPS/TLS：加密所有传输数据
  + API参数验证：防止不安全的参数

**4. 安全防护措施**

* **防止常见攻击**：
  + XSS防护：内容安全策略，输入验证和输出编码
  + CSRF防护：使用CSRF令牌，验证请求来源
  + 注入攻击防护：参数化查询，ORM/ODM框架
* **限流与防刷**：
  + IP级别限流：防止DDoS攻击
  + 账号级别限流：防止暴力破解
  + 敏感操作频率控制：如登录、发送验证码

#### 4.2.5 后端性能优化

为确保系统高性能运行，实施了多方面的优化：

**1. 数据库优化**

* **索引优化**：
  + 分析查询模式创建合适索引
  + 复合索引优化多字段查询
  + 避免过多索引导致的写入性能下降
* **查询优化**：
  + 优化查询结构，减少不必要的字段返回
  + 使用聚合管道优化复杂查询
  + 分页和限制结果集大小
* **连接池管理**：
  + 配置合适的连接池大小
  + 监控连接状态，及时释放空闲连接
  + 实现优雅的失败重连机制

**2. 缓存策略**

* **多级缓存**：
  + 内存缓存：服务内存中的LRU缓存
  + Redis缓存：分布式缓存层
  + CDN缓存：静态资源缓存
* **缓存策略**：
  + 时间策略：设置合适的TTL
  + 内容策略：缓存频繁访问、变化少的数据
  + 失效策略：主动失效与延迟双删
* **缓存穿透防护**：
  + 布隆过滤器：过滤不存在的Key
  + 空值缓存：缓存空结果
  + 请求合并：合并并发的相同请求

**3. 异步处理**

* **消息队列**：
  + 订单处理：异步处理订单流程
  + 通知发送：异步发送邮件和短信
  + 日志记录：异步写入日志
* **任务调度**：
  + 定时任务：定期更新缓存、统计数据
  + 延迟队列：处理需要延时执行的任务
  + 优先级队列：处理关键任务优先

**4. 服务监控与自动扩展**

* **健康检查**：
  + 服务健康状态监控
  + 数据库连接状态监控
  + 外部依赖监控
* **性能指标收集**：
  + 请求延迟统计
  + CPU和内存使用率
  + 数据库查询性能
* **自动扩展策略**：
  + 基于负载的扩展：CPU使用率超阈值触发扩展
  + 基于流量的扩展：请求量超阈值触发扩展
  + 定时扩展：在预期高峰期提前扩展

通过以上设计与实现，后端系统提供了高性能、安全可靠的服务支持，为AI导购功能的流畅运行提供了坚实基础。

### 4.3 AI导购助手实现

AI导购助手是本系统的核心功能，实现了基于DeepSeek大语言模型的智能购物指导。

#### 4.3.1 DeepSeek模型集成

系统与DeepSeek大语言模型的集成采用API调用方式，实现如下：

// services/ai/deepseekService.js  
const axios = require('axios');  
const config = require('../../config');  
  
// DeepSeek API配置  
const DEEPSEEK\_API\_URL = config.deepseek.apiUrl;  
const DEEPSEEK\_API\_KEY = config.deepseek.apiKey;  
  
// 封装DeepSeek API调用  
async function callDeepSeekAPI(messages, parameters = {}) {  
 try {  
 const response = await axios.post(  
 DEEPSEEK\_API\_URL,  
 {  
 model: 'deepseek-chat',  
 messages,  
 ...parameters  
 },  
 {  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': `Bearer ${DEEPSEEK\_API\_KEY}`  
 }  
 }  
 );  
   
 return response.data;  
 } catch (error) {  
 console.error('DeepSeek API error:', error.response?.data || error.message);  
 throw new Error('与AI模型通信失败');  
 }  
}  
  
// 生成对话响应  
async function generateChatResponse(userMessage, conversationHistory = []) {  
 // 整合会话历史  
 const messages = [  
 { role: 'system', content: getSystemPrompt() },  
 ...conversationHistory,  
 { role: 'user', content: userMessage }  
 ];  
   
 // 设置模型参数  
 const parameters = {  
 temperature: 0.7,  
 max\_tokens: 800,  
 top\_p: 0.95  
 };  
   
 // 调用API  
 const response = await callDeepSeekAPI(messages, parameters);  
   
 return {  
 message: response.choices[0].message.content,  
 usage: response.usage  
 };  
}  
  
// 系统提示词，定义AI导购助手的角色和行为  
function getSystemPrompt() {  
 return `你是一个专业的电商购物导购助手，你的主要职责是帮助用户选择最合适的商品。  
你将按照"沙漏模型"的流程引导用户完成购物决策，该模型分为四个阶段：  
1. 需求表达阶段：了解用户的基本需求、使用场景和偏好  
2. 参数筛选阶段：确定关键筛选条件，缩小选择范围  
3. 对比选择阶段：详细比较少数候选产品  
4. 决策推荐阶段：提供最终建议，消除决策疑虑  
  
你需要判断当前对话处于哪个阶段，并为用户提供相应的引导。  
你的回答应该简洁、专业、有针对性，避免过长的回复。  
当需要推荐具体商品时，你应该提供商品名称、价格、关键规格和推荐理由。  
如果用户的问题超出了你的能力范围，你应该礼貌地表示无法回答。`;  
}  
  
module.exports = {  
 generateChatResponse,  
 callDeepSeekAPI  
};

#### 4.3.2 沙漏模型流程实现

沙漏模型流程的实现涉及对话阶段管理、状态跟踪和转换策略实现。实现了根据对话内容智能判断当前阶段并提供相应的引导。关键代码摘录如下：

// services/ai/hourglassService.js  
const { generateChatResponse } = require('./deepseekService');  
const Conversation = require('../../models/Conversation');  
  
// 沙漏模型阶段定义  
const STAGES = {  
 NEED\_EXPRESSION: 'need\_expression',  
 PARAMETER\_FILTERING: 'parameter\_filtering',  
 COMPARISON: 'comparison',  
 DECISION: 'decision'  
};  
  
// 处理用户消息  
async function processMessage(message, conversationId, userId) {  
 // 获取或创建对话  
 let conversation = conversationId   
 ? await Conversation.findById(conversationId)   
 : new Conversation({  
 userId,  
 messages: [],  
 stage: STAGES.NEED\_EXPRESSION,  
 relatedProducts: []  
 });  
   
 // 添加用户消息到历史  
 conversation.messages.push({  
 role: 'user',  
 content: message,  
 timestamp: new Date()  
 });  
   
 // 获取当前对话阶段并拼装特定提示词  
 const currentStage = conversation.stage;  
 const stagePrompt = getStagePrompt(currentStage);  
 const extendedMessage = `${stagePrompt}\n用户消息: ${message}`;  
   
 // 获取对话历史并生成AI响应  
 const conversationHistory = conversation.messages  
 .slice(-10) // 仅使用最近10条消息  
 .map(msg => ({  
 role: msg.role,  
 content: msg.content  
 }));  
   
 const response = await generateChatResponse(extendedMessage, conversationHistory);  
   
 // 分析响应，判断阶段转换  
 const analysis = analyzeResponse(response.message, currentStage, message);  
   
 // 更新对话阶段和相关信息  
 if (analysis.newStage && analysis.newStage !== currentStage) {  
 conversation.stage = analysis.newStage;  
 }  
   
 conversation.messages.push({  
 role: 'assistant',  
 content: response.message,  
 timestamp: new Date()  
 });  
   
 if (analysis.productConditions) {  
 conversation.relatedProducts = analysis.productIds || [];  
 }  
   
 await conversation.save();  
   
 return {  
 message: response.message,  
 conversation\_id: conversation.\_id,  
 stage: conversation.stage,  
 productConditions: analysis.productConditions  
 };  
}  
  
// 分析响应，判断阶段转换和提取产品条件  
function analyzeResponse(response, currentStage, userMessage) {  
 // 阶段转换判断逻辑  
 let newStage = currentStage;  
   
 if (currentStage === STAGES.NEED\_EXPRESSION &&   
 (response.includes('预算') || response.includes('价格范围'))) {  
 newStage = STAGES.PARAMETER\_FILTERING;  
 } else if (currentStage === STAGES.PARAMETER\_FILTERING &&   
 response.includes('为您推荐以下')) {  
 newStage = STAGES.COMPARISON;  
 } else if (currentStage === STAGES.COMPARISON &&   
 response.includes('最适合您的是')) {  
 newStage = STAGES.DECISION;  
 }  
   
 // 提取产品筛选条件  
 const productConditions = extractProductConditions(response, userMessage);  
   
 return { newStage, productConditions };  
}

#### 4.3.3 多模态搜索实现

系统实现了多模态搜索功能，支持用户通过文本、图像和语音进行商品搜索。

**1. 图像搜索实现**

图像搜索功能通过调用大语言模型的图像分析能力，识别图片中的商品并进行搜索。实现过程如下：

* 图像上传和预处理
* 使用DeepSeek分析图像内容
* 提取关键特征和属性
* 基于提取的特征在商品数据库中搜索

关键代码摘录：

// 使用DeepSeek分析图像  
async function analyzeImageWithDeepSeek(imagePath) {  
 // 将图像转换为base64  
 const base64Image = await convertImageToBase64(imagePath);  
   
 // 创建包含图像的消息  
 const messages = [  
 {  
 role: 'system',  
 content: '你是一个专业的图像分析助手，请分析图像中的商品，并提供详细描述。'  
 },  
 {  
 role: 'user',  
 content: [  
 { type: 'text', text: '请分析这张图片中的商品，包括类别、品牌、型号等信息。' },  
 { type: 'image\_url', image\_url: { url: `data:image/jpeg;base64,${base64Image}` } }  
 ]  
 }  
 ];  
   
 // 调用API  
 const response = await callDeepSeekAPI(messages, { temperature: 0.5 });  
   
 // 提取分析结果  
 const analysis = response.choices[0].message.content;  
   
 // 提取产品条件  
 const productConditions = extractConditionsFromAnalysis(analysis);  
   
 return { result: analysis, productConditions };  
}  
  
// 图像搜索实现  
async function imageSearch(imagePath) {  
 // 分析图像获取商品特征  
 const analysis = await analyzeImageWithDeepSeek(imagePath);  
   
 // 基于提取的特征构建查询条件  
 const query = buildSearchQuery(analysis.productConditions);  
   
 // 搜索商品  
 const products = await Product.find(query).limit(20);  
   
 // 如果结果不足，使用更宽松的条件  
 if (products.length < 5 && analysis.productConditions?.category) {  
 const broadQuery = {   
 category: new RegExp(analysis.productConditions.category, 'i')   
 };  
 const additionalProducts = await Product.find(broadQuery)  
 .limit(20 - products.length);  
   
 products.push(...additionalProducts);  
 }  
   
 return {  
 results: products,  
 analysis: analysis.result,  
 total: products.length  
 };  
}

**2. 语音搜索实现**

语音搜索功能通过Web Speech API实现语音识别，并将识别结果传递给文本搜索服务。

// 前端语音识别实现  
function startVoiceRecognition() {  
 if ('webkitSpeechRecognition' in window) {  
 const recognition = new window.webkitSpeechRecognition();  
 recognition.lang = 'zh-CN';  
 recognition.continuous = false;  
   
 setIsListening(true);  
   
 recognition.onresult = (event) => {  
 const transcript = event.results[0][0].transcript;  
 setQuery(transcript);  
 onVoiceSearch(transcript);  
 };  
   
 recognition.onerror = (event) => {  
 console.error('Speech recognition error', event.error);  
 setIsListening(false);  
 };  
   
 recognition.onend = () => {  
 setIsListening(false);  
 };  
   
 recognition.start();  
 } else {  
 alert('您的浏览器不支持语音识别功能');  
 }  
}  
  
// 后端语音搜索处理  
async function voiceSearch(transcript) {  
 // 直接使用文本搜索处理转写结果  
 const searchResults = await textSearch(transcript, {}, 'relevance', 1, 20);  
   
 return {  
 results: searchResults.results,  
 total: searchResults.total,  
 transcript  
 };  
}

### 4.5 数据库实现

系统使用MongoDB作为主要数据库，实现了灵活的数据存储和查询功能。

#### 4.5.1 MongoDB模型实现

系统主要的数据模型包括用户模型、商品模型和对话历史模型。

**1. 用户模型**

用户模型存储用户账户信息、个人资料和偏好设置。

// models/User.js  
const mongoose = require('mongoose');  
const bcrypt = require('bcrypt');  
  
const userSchema = new mongoose.Schema({  
 username: {  
 type: String,  
 required: true,  
 unique: true,  
 trim: true  
 },  
 email: {  
 type: String,  
 required: true,  
 unique: true,  
 trim: true,  
 lowercase: true  
 },  
 password: {  
 type: String,  
 required: true  
 },  
 profile: {  
 name: String,  
 avatar: String,  
 gender: String,  
 birthday: Date,  
 phone: String,  
 address: String  
 },  
 preferences: {  
 categories: [String],  
 brands: [String],  
 priceRange: {  
 min: Number,  
 max: Number  
 },  
 features: [String]  
 },  
 createdAt: {  
 type: Date,  
 default: Date.now  
 },  
 updatedAt: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
// 密码加密  
userSchema.pre('save', async function(next) {  
 if (!this.isModified('password')) return next();  
   
 try {  
 const salt = await bcrypt.genSalt(10);  
 this.password = await bcrypt.hash(this.password, salt);  
 next();  
 } catch (error) {  
 next(error);  
 }  
});  
  
// 验证密码  
userSchema.methods.comparePassword = async function(candidatePassword) {  
 return bcrypt.compare(candidatePassword, this.password);  
};  
  
const User = mongoose.model('User', userSchema);  
  
module.exports = User;

**2. 商品模型**

商品模型存储商品基本信息、详细描述、规格参数和评价数据。

// models/Product.js  
const mongoose = require('mongoose');  
  
const productSchema = new mongoose.Schema({  
 name: {  
 type: String,  
 required: true,  
 index: true  
 },  
 category: {  
 type: String,  
 required: true,  
 index: true  
 },  
 brand: {  
 type: String,  
 required: true,  
 index: true  
 },  
 price: {  
 type: Number,  
 required: true  
 },  
 discount: {  
 type: Number,  
 default: 0  
 },  
 stock: {  
 type: Number,  
 required: true,  
 min: 0  
 },  
 images: {  
 type: [String],  
 default: []  
 },  
 description: {  
 type: String,  
 required: true  
 },  
 specifications: {  
 type: Map,  
 of: String  
 },  
 attributes: [{  
 name: String,  
 value: String  
 }],  
 rating: {  
 type: Number,  
 default: 0,  
 min: 0,  
 max: 5  
 },  
 reviews: [{  
 type: mongoose.Schema.Types.ObjectId,  
 ref: 'Review'  
 }],  
 createdAt: {  
 type: Date,  
 default: Date.now  
 },  
 updatedAt: {  
 type: Date,  
 default: Date.now  
 },  
 // 搜索优化字段  
 searchVector: {  
 type: String,  
 index: true  
 }  
});  
  
// 更新搜索向量  
productSchema.pre('save', function(next) {  
 // 将相关字段合并为搜索向量  
 this.searchVector = [  
 this.name,  
 this.brand,  
 this.category,  
 this.description,  
 ...(this.attributes || []).map(attr => `${attr.name} ${attr.value}`)  
 ].join(' ').toLowerCase();  
   
 next();  
});  
  
// 索引配置  
productSchema.index({ price: 1 });  
productSchema.index({ rating: -1 });  
productSchema.index({ brand: 1, category: 1 });  
productSchema.index({   
 name: 'text',   
 description: 'text',   
 brand: 'text',   
 category: 'text'   
}, {  
 weights: {  
 name: 10,  
 brand: 5,  
 category: 3,  
 description: 1  
 }  
});  
  
const Product = mongoose.model('Product', productSchema);  
  
module.exports = Product;

**3. 对话历史模型**

对话历史模型存储用户与AI导购助手的交互记录和相关状态。

// models/Conversation.js  
const mongoose = require('mongoose');  
  
const messageSchema = new mongoose.Schema({  
 role: {  
 type: String,  
 enum: ['user', 'assistant', 'system'],  
 required: true  
 },  
 content: {  
 type: String,  
 required: true  
 },  
 image: {  
 type: String  
 },  
 timestamp: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
const conversationSchema = new mongoose.Schema({  
 userId: {  
 type: mongoose.Schema.Types.ObjectId,  
 ref: 'User',  
 index: true  
 },  
 messages: [messageSchema],  
 stage: {  
 type: String,  
 enum: ['need\_expression', 'parameter\_filtering', 'comparison', 'decision'],  
 default: 'need\_expression'  
 },  
 relatedProducts: [{  
 type: mongoose.Schema.Types.ObjectId,  
 ref: 'Product'  
 }],  
 createdAt: {  
 type: Date,  
 default: Date.now  
 },  
 updatedAt: {  
 type: Date,  
 default: Date.now  
 },  
 lastActive: {  
 type: Date,  
 default: Date.now  
 }  
});  
  
// 更新时间  
conversationSchema.pre('save', function(next) {  
 this.updatedAt = new Date();  
 this.lastActive = new Date();  
 next();  
});  
  
// 索引配置  
conversationSchema.index({ lastActive: -1 });  
conversationSchema.index({ userId: 1, lastActive: -1 });  
  
const Conversation = mongoose.model('Conversation', conversationSchema);  
  
module.exports = Conversation;

#### 4.5.2 数据访问层实现

为了更好地封装数据访问逻辑，系统实现了一系列数据访问服务：

**1. 商品数据服务**

// services/product/productService.js  
const Product = require('../../models/Product');  
  
// 根据ID查找商品  
async function findProductById(productId) {  
 return Product.findById(productId);  
}  
  
// 查找特色商品  
async function findFeaturedProducts(limit = 8) {  
 return Product.find({ featured: true })  
 .sort({ rating: -1 })  
 .limit(limit);  
}  
  
// 按类别查找商品  
async function findProductsByCategory(category, page = 1, limit = 20) {  
 const skip = (page - 1) \* limit;  
   
 const products = await Product.find({ category })  
 .sort({ rating: -1 })  
 .skip(skip)  
 .limit(limit);  
   
 const total = await Product.countDocuments({ category });  
   
 return {  
 products,  
 total,  
 page,  
 pages: Math.ceil(total / limit)  
 };  
}  
  
// 根据条件查找商品  
async function findProductsByConditions(conditions, page = 1, limit = 20) {  
 if (!conditions || Object.keys(conditions).length === 0) {  
 return { products: [], total: 0, page, pages: 0 };  
 }  
   
 const skip = (page - 1) \* limit;  
 const query = buildQuery(conditions);  
   
 const products = await Product.find(query)  
 .sort({ rating: -1 })  
 .skip(skip)  
 .limit(limit);  
   
 const total = await Product.countDocuments(query);  
   
 return {  
 products,  
 total,  
 page,  
 pages: Math.ceil(total / limit)  
 };  
}  
  
// 构建查询条件  
function buildQuery(conditions) {  
 const query = {};  
   
 if (conditions.category) {  
 query.category = new RegExp(conditions.category, 'i');  
 }  
   
 if (conditions.brand || conditions.brands) {  
 const brands = conditions.brands || [conditions.brand];  
 query.brand = { $in: brands.map(b => new RegExp(b, 'i')) };  
 }  
   
 if (conditions.minPrice !== undefined && conditions.maxPrice !== undefined) {  
 query.price = {   
 $gte: conditions.minPrice,   
 $lte: conditions.maxPrice   
 };  
 } else if (conditions.minPrice !== undefined) {  
 query.price = { $gte: conditions.minPrice };  
 } else if (conditions.maxPrice !== undefined) {  
 query.price = { $lte: conditions.maxPrice };  
 }  
   
 return query;  
}  
  
module.exports = {  
 findProductById,  
 findFeaturedProducts,  
 findProductsByCategory,  
 findProductsByConditions  
};

**2. 对话历史服务**

// services/ai/conversationService.js  
const Conversation = require('../../models/Conversation');  
  
// 获取用户历史对话  
async function getUserConversations(userId, limit = 10) {  
 return Conversation.find({ userId })  
 .sort({ lastActive: -1 })  
 .limit(limit);  
}  
  
// 获取单个对话  
async function getConversationById(conversationId) {  
 return Conversation.findById(conversationId);  
}  
  
// 创建新对话  
async function createConversation(userId) {  
 const conversation = new Conversation({  
 userId,  
 messages: [],  
 stage: 'need\_expression'  
 });  
   
 await conversation.save();  
 return conversation;  
}  
  
// 添加消息到对话  
async function addMessageToConversation(conversationId, message) {  
 const conversation = await Conversation.findById(conversationId);  
   
 if (!conversation) {  
 throw new Error('对话不存在');  
 }  
   
 conversation.messages.push({  
 ...message,  
 timestamp: new Date()  
 });  
   
 conversation.lastActive = new Date();  
 await conversation.save();  
   
 return conversation;  
}  
  
// 更新对话阶段  
async function updateConversationStage(conversationId, stage) {  
 const validStages = ['need\_expression', 'parameter\_filtering', 'comparison', 'decision'];  
   
 if (!validStages.includes(stage)) {  
 throw new Error('无效的对话阶段');  
 }  
   
 return Conversation.findByIdAndUpdate(  
 conversationId,  
 { stage, lastActive: new Date() },  
 { new: true }  
 );  
}  
  
module.exports = {  
 getUserConversations,  
 getConversationById,  
 createConversation,  
 addMessageToConversation,  
 updateConversationStage  
};

## 第5章 系统测试与评估

本章对系统进行全面测试和评估，包括功能测试、性能测试和用户体验评估，以验证系统是否满足预期需求。

### 5.1 测试环境

#### 5.1.1 硬件环境

* 处理器：Intel Core i7-12700H
* 内存：16GB DDR4
* 存储：512GB NVMe SSD
* 网络：100Mbps宽带网络

#### 5.1.2 软件环境

* 操作系统：Windows 11 专业版
* 浏览器：Chrome 112、Firefox 112、Edge 112
* 开发工具：Visual Studio Code 1.74
* 测试工具：Postman 10.13, Chrome DevTools

### 5.2 功能测试

功能测试主要验证系统各个功能模块是否正常工作，测试结果如下：

#### 5.2.1 用户管理功能测试

| 测试项 | 测试内容 | 预期结果 | 实际结果 | 是否通过 |
| --- | --- | --- | --- | --- |
| 用户注册 | 注册新用户账号 | 成功创建账号并跳转到首页 | 与预期一致 | 是 |
| 用户登录 | 使用已注册账号登录 | 登录成功并展示用户信息 | 与预期一致 | 是 |
| 个人信息修改 | 修改用户资料 | 信息更新成功 | 与预期一致 | 是 |
| 密码修改 | 修改登录密码 | 密码更新成功 | 与预期一致 | 是 |
| 用户登出 | 退出登录状态 | 成功登出并返回登录页 | 与预期一致 | 是 |

#### 5.2.2 AI导购功能测试

| 测试项 | 测试内容 | 预期结果 | 实际结果 | 是否通过 |
| --- | --- | --- | --- | --- |
| 需求表达阶段 | 用户提出购买需求 | AI正确理解并询问细节 | 与预期一致 | 是 |
| 参数筛选阶段 | 用户与AI交互确定筛选条件 | AI根据用户反馈调整参数范围 | 与预期一致 | 是 |
| 对比选择阶段 | AI提供产品对比 | 展示2-5款商品对比信息 | 与预期一致 | 是 |
| 决策推荐阶段 | AI给出最终推荐 | 提供详细推荐理由 | 与预期一致 | 是 |
| 阶段转换 | AI自动判断并切换阶段 | 自然流畅地完成阶段转换 | 大部分情况流畅，个别复杂情况需优化 | 部分通过 |
| 上下文记忆 | 记住用户偏好和需求 | 在对话过程中持续使用已获取信息 | 与预期一致 | 是 |

#### 5.2.3 多模态搜索功能测试

| 测试项 | 测试内容 | 预期结果 | 实际结果 | 是否通过 |
| --- | --- | --- | --- | --- |
| 文本搜索 | 输入关键词搜索商品 | 返回相关商品列表 | 与预期一致 | 是 |
| 图像搜索 | 上传商品图片进行搜索 | 识别图片并返回相似商品 | 简单图片识别准确，复杂图片识别效果一般 | 部分通过 |
| 语音搜索 | 语音输入搜索关键词 | 识别语音并执行搜索 | 安静环境下识别准确 | 部分通过 |
| 搜索结果筛选 | 对搜索结果进行筛选排序 | 按照筛选条件调整结果 | 与预期一致 | 是 |

### 5.3 性能测试

性能测试主要评估系统在不同负载下的响应能力和稳定性。

#### 5.3.1 响应时间测试

使用Chrome DevTools进行响应时间测试，结果如下：

| 功能模块 | 平均响应时间(ms) | 最大响应时间(ms) | 最小响应时间(ms) | 备注 |
| --- | --- | --- | --- | --- |
| 首页加载 | 420 | 650 | 320 | 缓存启用状态 |
| 商品列表页 | 580 | 820 | 380 | 每页20个商品 |
| 商品详情页 | 510 | 750 | 350 | 包含推荐商品 |
| 用户登录 | 350 | 520 | 250 | - |
| AI对话响应 | 1500 | 2200 | 1000 | 取决于查询复杂度 |
| 图像搜索 | 1800 | 2500 | 1200 | 取决于图片大小 |

测试表明，常规页面加载和API请求响应时间在可接受范围内。AI相关功能的响应时间较长，但考虑到其复杂性，仍在合理范围内。

#### 5.3.2 并发性能测试

在开发环境中模拟不同用户并发访问系统，测试系统的稳定性：

| 并发用户数 | 平均响应时间(ms) | 请求成功率(%) | 系统CPU使用率(%) | 内存使用(MB) |
| --- | --- | --- | --- | --- |
| 10 | 620 | 100 | 25 | 850 |
| 20 | 780 | 100 | 35 | 980 |
| 50 | 1100 | 99.5 | 55 | 1200 |
| 100 | 1850 | 98 | 75 | 1450 |

在并发用户数为50以内时，系统表现良好。当并发数达到100时，响应时间开始显著增加，在实际部署中需要通过优化和扩展改进系统性能。

#### 5.3.3 AI模型性能测试

测试DeepSeek模型在不同场景下的响应性能：

| 测试场景 | 平均响应时间(s) | 响应质量评分(1-5分) |
| --- | --- | --- |
| 简单产品咨询 | 1.2 | 4.5 |
| 复杂需求分析 | 2.5 | 4.2 |
| 多参数比较 | 3.0 | 4.0 |
| 专业知识解答 | 2.2 | 4.3 |
| 图片分析 | 3.5 | 3.8 |

DeepSeek模型整体表现良好，尤其在简单咨询和专业知识解答方面表现突出。图片分析功能响应时间较长，准确性有待提高。

### 5.4 用户体验评估

为评估系统的实际使用体验，邀请了10名志愿者进行用户测试，并通过问卷收集反馈。

#### 5.4.1 用户测试方案

测试志愿者包括： - 电商平台常用用户：6人 - 不常使用电商的用户：3人 - 电商行业从业人员：1人

测试任务包括： 1. 使用AI导购助手购买指定类型商品 2. 尝试多模态搜索功能 3. 对比AI导购与传统导购的体验差异

#### 5.4.2 用户反馈分析

用户评分结果(1-5分)：

| 评估项目 | 平均分 | 标准差 |
| --- | --- | --- |
| 系统整体易用性 | 4.2 | 0.6 |
| AI导购流程自然度 | 4.0 | 0.7 |
| 推荐商品准确性 | 3.8 | 0.7 |
| 多模态搜索便捷性 | 4.1 | 0.5 |
| 系统响应速度 | 3.7 | 0.6 |
| 界面美观度 | 4.3 | 0.4 |
| 整体购物体验 | 4.1 | 0.5 |

用户主要正面反馈： - AI导购助手的对话自然流畅，能理解大部分需求 - 沙漏模型的流程设计帮助减轻了决策负担 - 多模态搜索特别是图像搜索功能非常实用 - 商品对比呈现方式直观清晰

主要改进建议： - AI响应时间可进一步优化 - 图像识别准确性有待提高 - 希望增加更多商品种类和参数 - 建议提供更个性化的推荐算法

### 5.5 测试结论

通过系统测试和评估，得出以下结论：

1. **功能完整性**：系统实现了预期的核心功能，能够满足电商平台的基本需求和AI导购的创新需求。个别功能如图像识别准确性和阶段转换智能性有待优化。
2. **性能表现**：系统在中等负载下表现良好，响应时间在可接受范围内。高并发场景下性能有所下降，未来可通过优化代码和增加服务器资源提升性能。
3. **用户体验**：用户对系统整体评价积极，特别是对AI导购流程和多模态搜索功能的评价较高。用户认为系统能有效减轻购物决策负担，提升购物体验。
4. **问题与不足**：主要问题集中在AI响应时间、图像识别准确性和高并发场景下的性能表现，这些将作为后续优化的重点方向。

总体而言，系统达到了设计目标，实现了基于沙漏模型的AI导购功能，有效解决了用户在电商平台面临的决策困难问题。用户反馈表明，系统的创新功能确实提升了购物体验和决策效率。

## 第6章 总结与展望

### 6.1 研究总结

本毕业设计围绕”基于AI导购功能的电商网站设计与实现”主题，完成了以下工作：

1. **提出了”沙漏模型”导购流程**：创新性地设计了模拟专业导购的四阶段购物决策流程，包括需求表达、参数筛选、对比选择和决策推荐阶段。该模型遵循用户认知过程，有效降低了决策负担。
2. **实现了基于大语言模型的AI导购助手**：成功集成DeepSeek大语言模型，并通过提示工程和上下文管理，赋予AI助手专业导购能力。实现了自然流畅的对话体验和精准的商品推荐。
3. **开发了多模态交互功能**：实现了文本、语音和图像多种搜索方式，为用户提供灵活便捷的商品查找体验。特别是图像搜索功能，提升了用户寻找相似商品的效率。
4. **构建了完整的电商系统**：从前端界面到后端服务，从数据存储到业务逻辑，实现了包含用户管理、商品管理、订单处理等在内的电商系统基础功能，为AI导购功能提供了支撑。
5. **进行了系统测试与用户评估**：通过功能测试、性能测试和用户体验评估，验证了系统的可用性和有效性。测试结果表明，系统达到了预期设计目标，用户评价整体积极。

研究结果显示，基于沙漏模型的AI导购功能能够有效解决用户在电商平台面临的信息过载和决策困难问题。通过引导用户从模糊需求到明确选择的过程，系统降低了用户的认知负担，提高了购物决策效率和满意度。

### 6.2 创新点

本设计的主要创新点包括：

1. **沙漏模型导购流程**：创新性地提出并实现了”沙漏模型”导购流程，这一模型符合用户认知过程，将购物决策分解为有序阶段，有效减轻了用户的选择困难。
2. **基于LLM的对话式导购**：不同于传统的基于规则或简单问答的导购系统，本系统利用大语言模型实现了深度对话式导购，能够理解复杂需求，提供个性化建议。
3. **多模态搜索与交互**：集成了文本、语音和图像多种搜索方式，特别是图像搜索功能，让用户能够通过上传图片快速找到相似商品，提升了用户体验。
4. **上下文感知的阶段转换**：系统能够智能判断当前对话所处阶段，实现平滑自然的阶段转换，避免了生硬的流程跳转，提供了连贯的用户体验。
5. **专业知识增强的商品推荐**：结合大语言模型的专业知识和商品信息，实现了基于专业分析的商品推荐，不仅推荐什么商品，还能解释为什么推荐，增强了用户信任。

### 6.3 不足与改进方向

尽管系统已经实现了预期功能，但仍存在一些不足，未来可从以下方面进行改进：

1. **AI响应性能优化**：
   * 优化模型调用方式，减少响应时间
   * 实现请求队列和缓存机制，提高并发处理能力
   * 探索模型量化和本地部署方案，降低API调用成本
2. **多模态功能增强**：
   * 提高图像识别准确性，支持更复杂场景的图片分析
   * 增强语音识别在嘈杂环境下的准确率
   * 增加视频分析等更丰富的多模态交互方式
3. **个性化推荐深化**：
   * 结合用户历史行为数据，实现更精准的个性化推荐
   * 引入协同过滤和内容推荐的混合算法
   * 开发用户兴趣模型，捕捉长期和短期偏好变化
4. **系统性能与扩展性**：
   * 优化数据库结构和查询效率
   * 实现系统水平扩展，提高高并发处理能力
   * 引入微服务架构，提升系统模块化和可维护性

### 6.4 未来展望

随着人工智能技术和电子商务的持续发展，基于AI的电商导购系统有着广阔的发展前景。未来可探索的方向包括：

1. **结合多模态大模型**：利用更先进的多模态大模型，实现更深度的图文理解和分析，提供更自然的商品展示和解读。
2. **引入AR/VR技术**：结合增强现实和虚拟现实技术，让用户能够在虚拟环境中试用商品，提供沉浸式购物体验。
3. **个性化AI导购助手**：为每个用户培养专属的AI导购助手，通过长期交互学习用户偏好，实现真正的个性化服务。
4. **情感计算与用户心理**：引入情感计算技术，识别和响应用户情绪变化，提供更人性化的购物体验。
5. **行业垂直化发展**：针对服装、电子产品、家居等不同行业，开发专业化的垂直领域AI导购系统，提供更专业的导购服务。

总之，本毕业设计在理论研究和实际应用之间架起了桥梁，通过”沙漏模型”和大语言模型技术，为电商行业提供了新的用户体验提升方案。未来随着技术的发展和应用的深入，AI导购系统将在提升用户购物体验、促进消费决策、推动电商创新等方面发挥更大作用。