

**课 程 实验 报 告**

**题目： 网上书店管理系统**

**课程名称： 数据库课程设计**

**软件学院**

目录

[一、 概述 4](#_Toc4219)

[二、 需求分析 4](#_Toc3811)

[2.1. 功能需求 4](#_Toc8261)

[2.1.1. 供书目录与库存管理 5](#_Toc29948)

[2.1.2. 采购管理 6](#_Toc4146)

[2.1.3. 客户管理 6](#_Toc5807)

[2.1.4. 顾客订单与发货管理 6](#_Toc20046)

[2.1.5. 供应商管理 6](#_Toc3038)

[2.1.6. 网上浏览与查询 7](#_Toc5331)

[2.2. 非功能需求 7](#_Toc10297)

[2.2.1. 性能需求 7](#_Toc18852)

[2.2.2. 可靠性需求 7](#_Toc24385)

[2.2.3. 可用性需求 7](#_Toc19137)

[2.2.4. 安全性需求 8](#_Toc30059)

[2.2.5. 可维护性需求 8](#_Toc5980)

[2.2.6. 可扩展性需求 8](#_Toc20337)

[三、 数据库设计 8](#_Toc615)

[3.1. 系统的数据库结构设计 8](#_Toc30993)

[3.2. 数据库中的约束和触发器 16](#_Toc17120)

[3.3. 数据库的性能优化 16](#_Toc1730)

[3.4. 系统中的安全性设计 17](#_Toc5919)

[3.5. 数据库的扩展性和可维护性 17](#_Toc21040)

[3.6. 数据库管理与操作 17](#_Toc7154)

[3.7总结 17](#_Toc20350)

[四、 架构设计 18](#_Toc15698)

[4.1. 表现层（Presentation Layer） 18](#_Toc27033)

[4.2. 业务逻辑层（Business Logic Layer） 18](#_Toc5613)

[4.3. 数据层（Data Layer） 18](#_Toc1571)

[五、 前端设计 19](#_Toc31712)

[5.1 技术栈选择 19](#_Toc5567)

[5.2 组件化设计 19](#_Toc11342)

[5.3 响应式设计 20](#_Toc31691)

[5.4 状态管理 20](#_Toc21698)

[5.5 路由管理 20](#_Toc12486)

[5.6 页面展示 20](#_Toc28788)

[六、 后端配置 29](#_Toc3950)

[6.1. Maven 配置 30](#_Toc16467)

[6.2. 主应用类（BookstoreApplication） 30](#_Toc28583)

[6.3. CORS 配置（CorsConfig） 30](#_Toc20498)

[6.4. 安全配置（SecurityConfig） 31](#_Toc1608)

[6.5. 数据库连接配置（application.properties） 31](#_Toc13623)

[6.6. JWT认证过滤器（JwtAuthenticationFilter） 31](#_Toc27624)

[6.7. 总结 32](#_Toc5319)

[七、 后端各模块功能的实现 32](#_Toc11523)

[7.1. 图书管理模块 32](#_Toc32438)

[7.1.1. 控制器层（BookController） 32](#_Toc406)

[7.1.2. 模型层（Book） 34](#_Toc31614)

[7.1.3. 数据访问层（BookRepository） 35](#_Toc1758)

[7.1.4. 服务层（BookService） 35](#_Toc14185)

[7.2. 购物车管理模块 36](#_Toc13300)

[7.2.1. 控制器层（CartController） 36](#_Toc16700)

[7.2.2. 模型层（Cart 和 CartItem） 37](#_Toc18518)

[7.2.3. 数据访问层（CartRepository） 38](#_Toc16163)

[7.2.4. 服务层（CartService） 38](#_Toc11163)

[7.3. 用户管理与认证模块 39](#_Toc21515)

[7.3.1. 控制器层（CustomerController） 39](#_Toc12258)

[7.3.2. 服务层（CustomerService） 42](#_Toc212)

[7.3.3. JWT处理类（JwtUtil） 43](#_Toc29897)

[7.3.4. 自定义身份验证令牌（CustomAuthenticationToken） 43](#_Toc26619)

[7.4. 订单管理模块 44](#_Toc22241)

[7.4.1. 控制器层（OrderController） 44](#_Toc20397)

[7.4.2. 模型层（Order 和 OrderItem） 46](#_Toc30716)

[7.4.3. 数据访问层（OrderRepository 和 OrderItemRepository） 47](#_Toc23911)

[7.4.4. 服务层（OrderService） 48](#_Toc3793)

[7.5. 库存管理与采购订单模块 48](#_Toc8069)

[7.5.1. 控制器层 48](#_Toc24768)

[7.5.2 模型层 49](#_Toc23673)

[7.5.3. 数据访问层 50](#_Toc7873)

[7.5.4. 服务层 51](#_Toc7292)

[7.6. 供应商管理模块 52](#_Toc26309)

[7.6.1. 控制器层（SupplierController） 52](#_Toc18928)

[7.6.2. 模型层（Supplier） 53](#_Toc29961)

[7.6.3. 数据访问层（SupplierRepository） 54](#_Toc26078)

[7.6.4. 服务层（SupplierService） 54](#_Toc11572)

[八、 总结与体会 56](#_Toc31009)

[8.1 项目成果总结 56](#_Toc8504)

[8.2 个人体会 56](#_Toc28746)

# 概述

本任务旨在设计和实现一个小型数据库应用系统，以服务于一个网上书店的日常运营和网上购书业务。系统的设计涉及多个功能模块，包括供书目录与库存管理、采购管理、客户管理、顾客订单与发货管理、供应商管理及网上查询与浏览等。这些功能将通过客户端/服务器（C/S）模式来实现，以确保书店内部的日常业务能够高效管理；而顾客则可以通过互联网进行在线查询、浏览以及下单购书。为提升系统性能，业务规则尽量放置在中间层或数据库服务器端处理，从而有效地降低了客户端负载，提升了系统整体响应速度和稳定性。

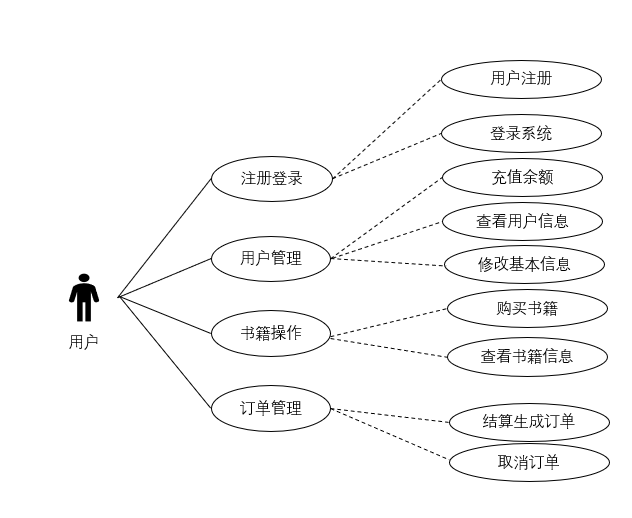
在实现过程中，我们采用了关系型数据库管理系统，通过设计多个数据库表、视图、存储过程和触发器来满足系统的需求。数据库设计的重点是保证数据的完整性、查询效率、事务的一致性以及安全性，确保书店运营管理的顺畅和顾客在线购物的便利。我们还采用了Spring Boot和Vue3用于实现前后端，进行了前后端的分离。

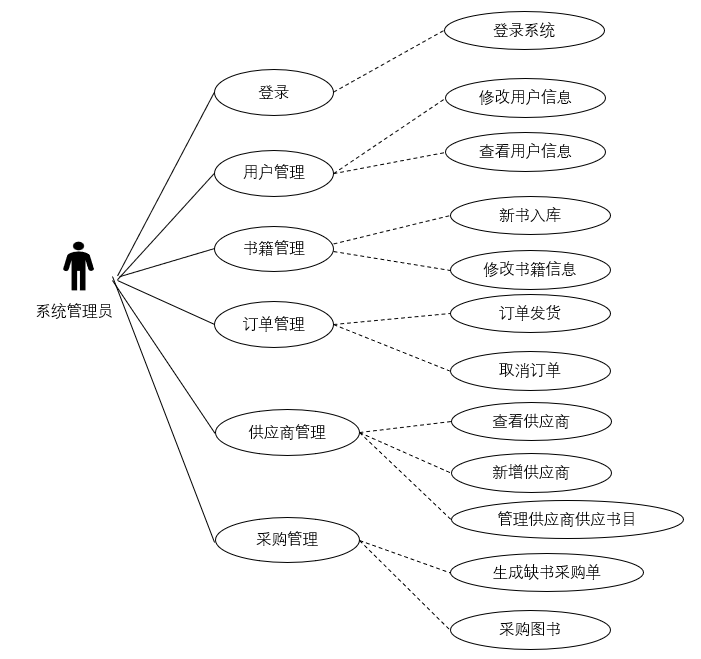
# 需求分析

本书店系统面向两个主要用户群体：书店管理人员和网上顾客。系统的核心功能涵盖了从书籍入库到顾客下单，再到发货的整个流程，并支持书店内部与顾客之间的互动。

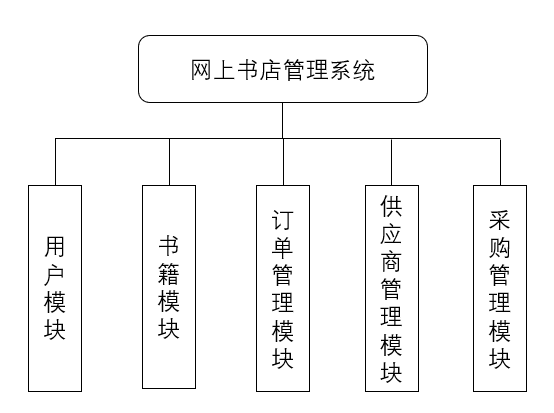
## 2.1. 功能需求

功能用例图如下图所示：





网上书店管理系统可以分为下面几个模块。



以下为各功能模块的详细需求分析：

### 2.1.1. 供书目录与库存管理

供书目录与库存管理是书店的核心业务之一。书店需要管理大量书籍信息，包括书号、书名、作者、出版社、价格、关键字、目录、封皮、存货量以及供应商信息等。此外，对于丛书的管理也是一个特殊需求，某个书号下可能包含多本书。库存管理要求精确记录每本书的存货数量，及时更新库存变化，并提供相应的库存查询功能。

* 关键需求：
  + 书籍信息记录（书号、书名、作者、出版社、价格等）。
  + 丛书的管理（一本书可能对应多本书）。
  + 书籍库存量的管理，库存变化的实时更新。
  + 供应商信息管理（多供应商情况）。

### 2.1.2. 采购管理

采购管理系统主要用于缺书登记和采购单管理。当书籍库存量低于预定阈值时，系统应自动生成缺书记录；同时，书店工作人员可以根据缺书记录生成采购单，采购单到货后，需及时更新库存信息，并通知顾客相关书籍的到货情况。

* 关键需求：
  + 缺书登记：包括书号、书名、出版社、数量、登记日期。
  + 自动生成缺书记录：当库存量低于设定限度时。
  + 采购单管理：包括采购单的创建、更新和删除。
  + 到货处理：根据采购单更新库存，并通知顾客。

### 2.1.3. 客户管理

客户管理模块主要负责记录和管理顾客的基本信息、账户余额和信用等级等。系统需要对客户的信用等级进行动态调整，根据顾客的余额和累计购买金额来提升或降低信用等级。此外，顾客的信息可由顾客自行在网上维护，如修改地址、邮箱等。

* 关键需求：
  + 客户信息记录：包括用户ID、密码、名称、地址、帐户余额、信用等级等。
  + 信用管理：动态调整顾客的信用等级。
  + 支付处理：顾客付款后更新余额，增加账户信用额度。

### 2.1.4. 顾客订单与发货管理

顾客订单管理模块处理顾客在网上下单的业务。每个订单需要记录订单号、订货日期、客户ID、订书数量、金额、发货地址等信息。系统必须保证顾客只能订购库存中已有的书籍，若库存不足时允许顾客预定。当订单生成后，系统会检查顾客的账户余额和信用等级，以确定是否可以发货，并根据顾客的信用等级进行发货处理。一个订单可以分次发货。

* 关键需求：
  + 顾客下单：生成订单，记录订书数量、金额和发货地址。
  + 订单检查：检查库存情况及顾客信用额度。
  + 发货处理：依据库存、余额和信用等级进行发货。

### 2.1.5. 供应商管理

供应商管理功能涉及供应商的基本信息及供应商的书目信息管理。书店需要记录供应商的名称、联系方式、所提供书籍信息等。此外，供应商应能够上传自己的书目信息，书店根据供应商提供的书籍信息进行库存更新。

* 关键需求：
  + 供应商信息管理：包括供应商的基本信息及其所提供的书籍。
  + 供应商书目信息：记录供应商供货的书籍信息。

### 2.1.6. 网上浏览与查询

为了方便顾客查找书籍，系统必须提供书籍信息的查询功能。顾客可以通过书号、书名、出版社、作者等关键字进行查询，并支持模糊查询。系统还需提供客户的基本信息查询，包括历史订单和发货信息等。

* 关键需求：
  + 支持按多种条件查询书目信息。
  + 支持模糊查询和精确查询。
  + 客户历史订单查询。

## 2.2. 非功能需求

非功能需求涉及系统的性能、可靠性、可用性、安全性、可维护性和可扩展性等方面。以下是网上书店项目的关键非功能需求：

### 2.2.1. 性能需求

响应时间：系统应保证在高负载情况下，页面加载和查询响应时间不超过2秒。

并发用户：系统应支持至少1000个并发用户同时在线操作。

事务处理：订单处理和库存更新等关键事务应在1秒内完成。

### 2.2.2. 可靠性需求

系统稳定性：系统应保证99.9%的正常运行时间，允许的年度停机时间不超过8.76小时。

数据准确性：系统操作应保证数据的一致性和准确性，错误率低于0.1%。

### 2.2.3. 可用性需求

用户界面：界面应直观易用，确保所有用户都能轻松地使用系统。

帮助和文档：提供在线帮助和用户手册，以便用户能够快速上手。

多语言支持：系统应支持至少两种语言，以满足不同地区用户的需求。

### 2.2.4. 安全性需求

数据加密：用户敏感数据（如密码、支付信息）应进行加密存储。

身份验证：系统应实现基于JWT的身份验证机制，确保用户身份的合法性。

权限控制：系统应根据用户角色（管理员、普通用户）实施不同的访问权限。

### 2.2.5. 可维护性需求

代码质量：代码应遵循行业标准，易于理解和维护。

日志记录：系统应记录详细的操作日志，以便于问题追踪和系统监控。

错误处理：系统应具备完善的错误处理机制，能够及时反馈并记录错误信息。

### 2.2.6. 可扩展性需求

模块化设计：系统应采用模块化设计，便于未来的功能扩展和维护。

API设计：系统应提供RESTful API，以便于与其他系统集成和扩展。

硬件和软件的兼容性：系统应能够在不同的硬件和软件环境中运行，包括不同的操作系统和数据库系统

这些非功能需求将确保网上书店系统不仅满足业务需求，而且在性能、安全性、可用性等方面达到行业标准。

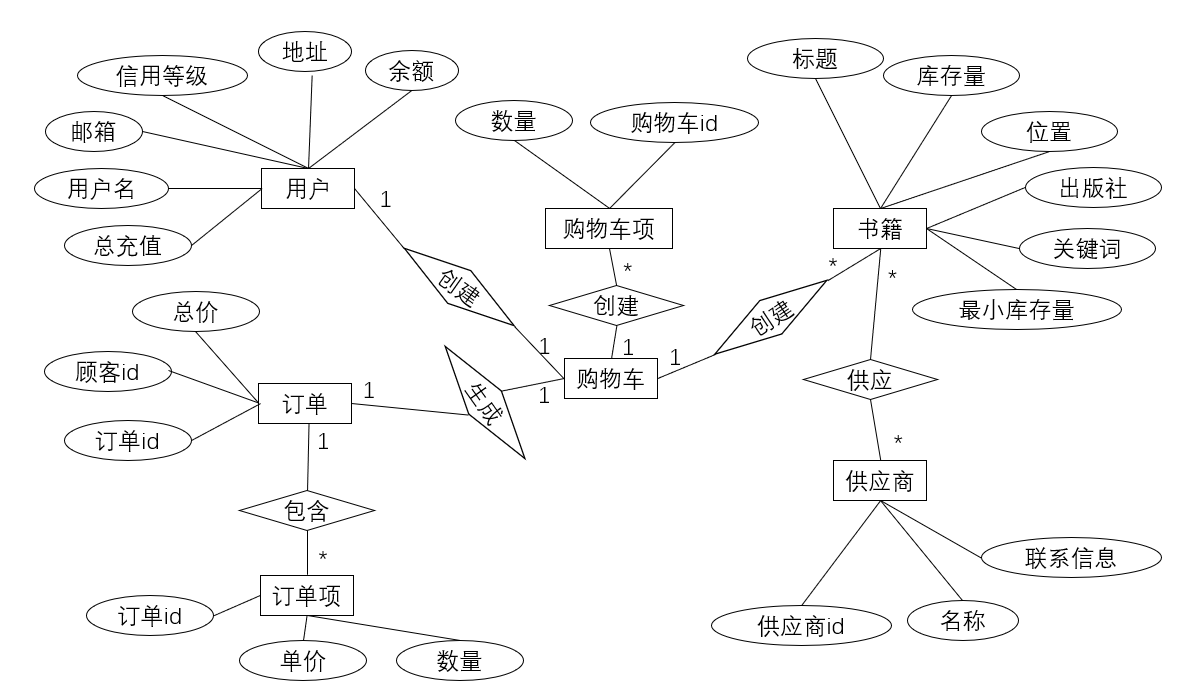
# 数据库设计

通过对需求的分析，我们构建了一个全面的数据库架构来实现书店的库存管理、采购管理、顾客管理、订单处理等功能。该系统采用了关系型数据库设计，合理地划分了不同的业务模块，使得各项功能能够高效地执行，同时保证数据的完整性和一致性。

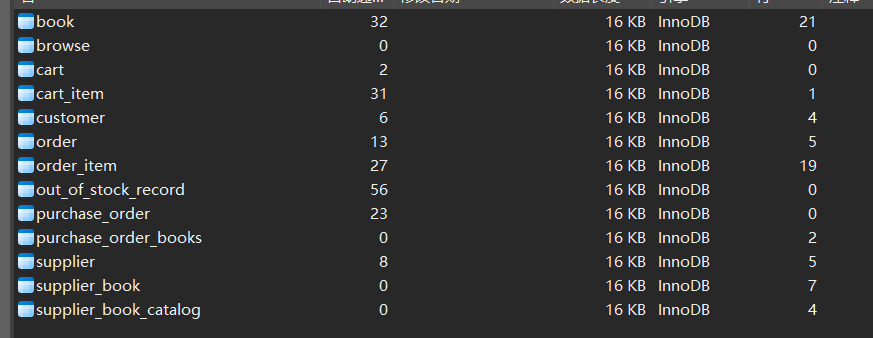
## 3.1. 系统的数据库结构设计

数据库的设计遵循“模块化、规范化”的原则，通过多个表格来管理书店的不同业务。每个表代表一个业务实体，例如书籍、顾客、订单、供应商等。同时，各个表格之间通过外键关联，确保数据之间的联系和一致性。

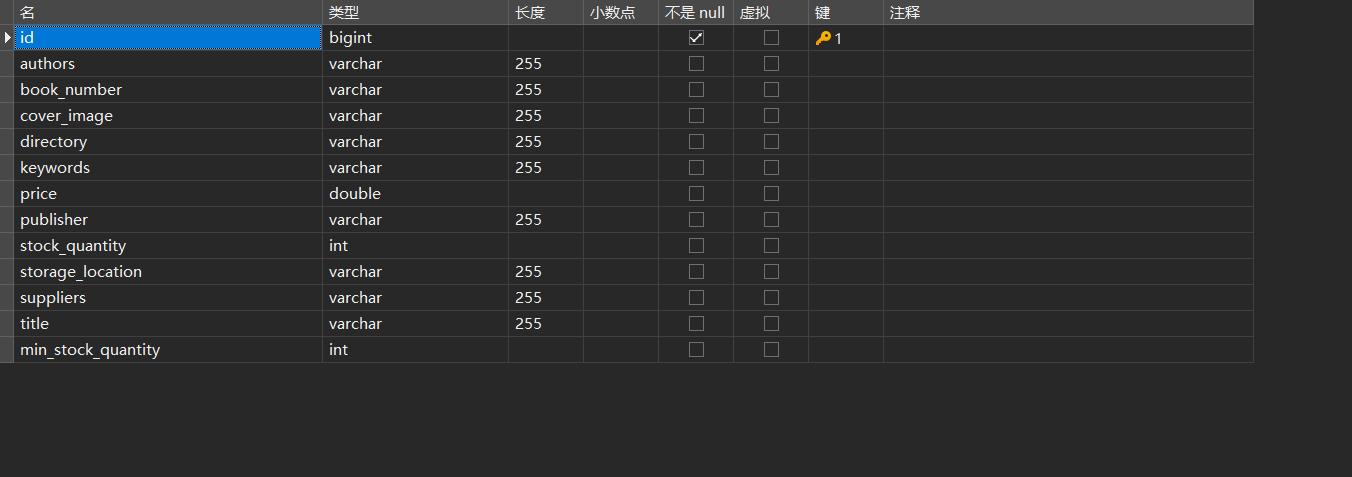
系统ER图如下所示：

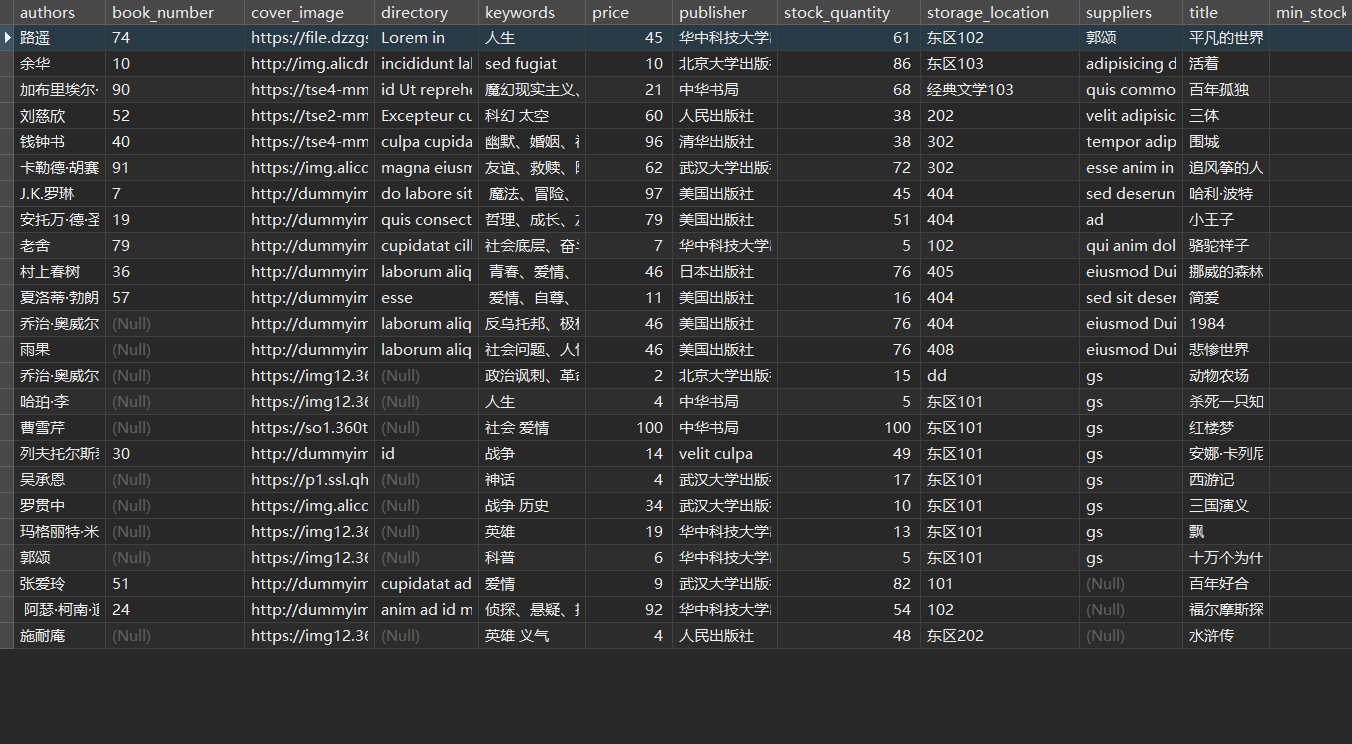


主要的数据库表包括：

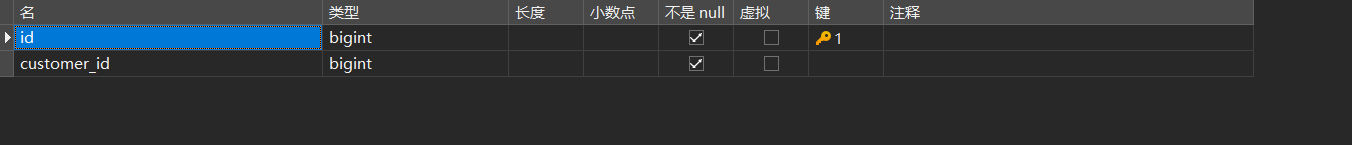


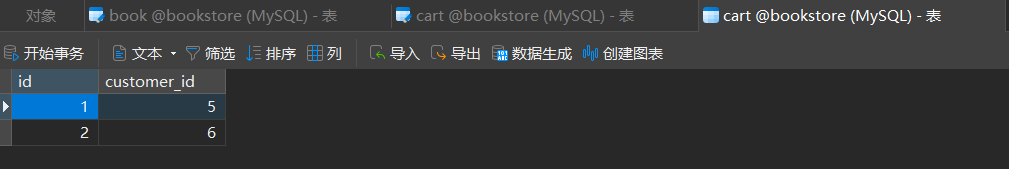
**书籍信息表（Book）**：该表存储所有书籍的基本信息，如书号、书名、作者、出版社、价格、存货量等。为了支持书店的库存管理，库存量是一个关键字段。当书籍被采购或者售出时，库存量会随之更新。





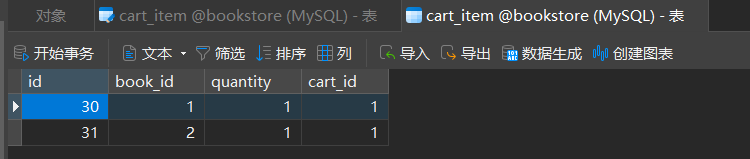
**购物车表（Cart）**：用于存储顾客的购物车信息，包括购物车ID和顾客ID。





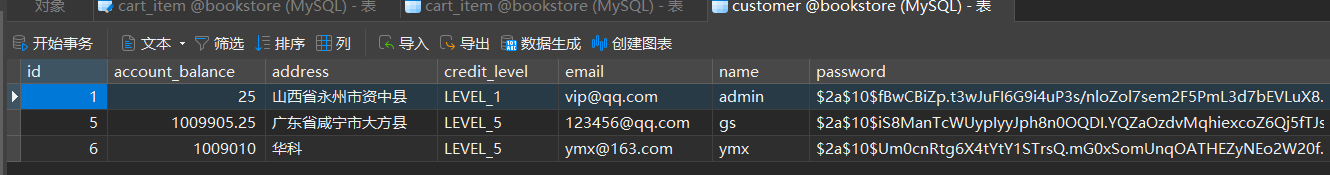
**购物车项表（Cart\_item）**：记录购物车中每项商品的信息，包括商品ID、数量和购物车ID。





**顾客信息表（Customer）**：记录顾客的个人信息，包括顾客的ID、用户名、密码、名称、地址、账户余额、信用等级等。该表对于书店的订单处理和顾客信用管理至关重要。



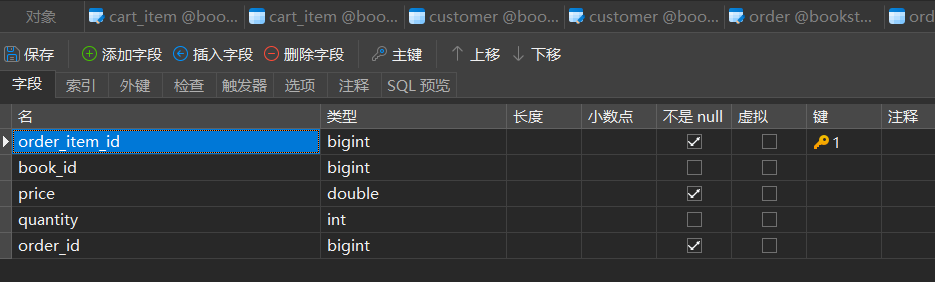


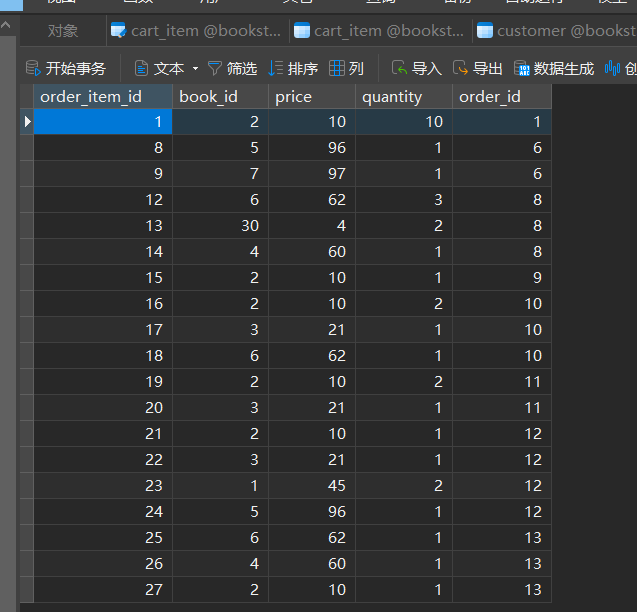
**订单表（Order）**：存储顾客下单的订单信息。每个订单包括订单号、顾客ID、订单状态、总金额等。订单表与库存表和顾客信息表有紧密的关系，确保只有当库存足够时，才能生成并确认订单。





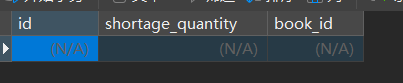
**订单项表（Order\_item）**：记录订单中每项商品的信息，包括订单项ID、书籍ID、价格、数量和订单ID。



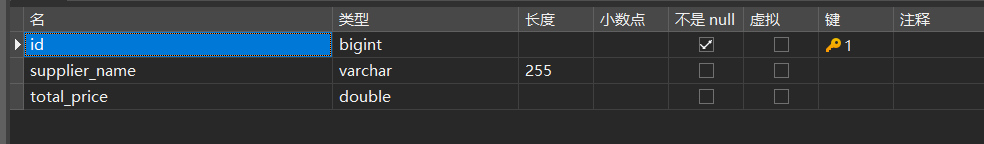


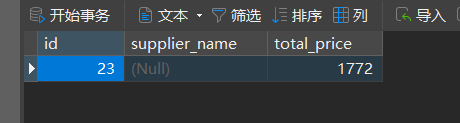
**缺货记录表（Out\_of\_stock\_record）**：用于记录库存不足的情况，包括缺货数量和书籍ID。



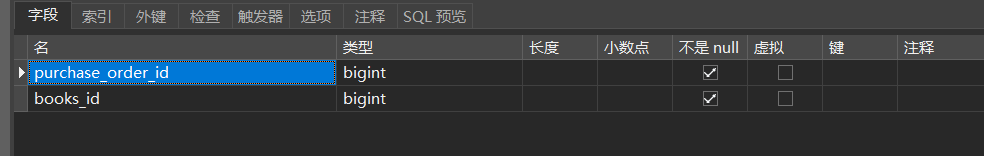


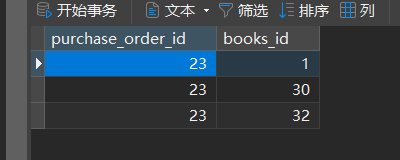
**采购订单表（Purchase\_order）**：记录书店的采购活动，包含了采购单号、供应商名称、总价格等信息。





**采购订单书籍表（Purchase\_order\_books）**：关联采购订单和书籍的信息，包括采购订单ID和书籍ID。



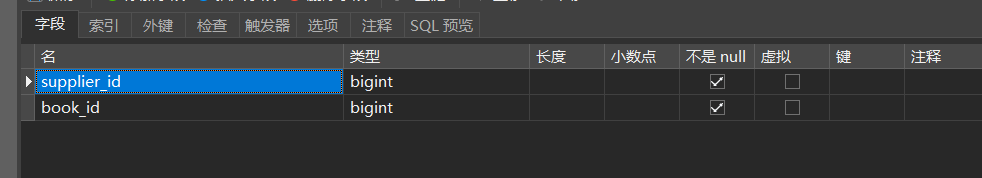


**供应商信息表（Supplier）**：供应商的基本信息及供货书籍的信息都存储在此表中。每个供应商提供的书籍信息可能与书店的库存管理直接相关，因此，供应商表对于缺书登记和采购单的生成至关重要。





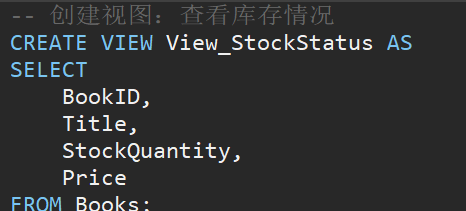
**供应商书籍表（Supplier\_book）**：关联供应商和书籍的信息，包括供应商ID和书籍ID。

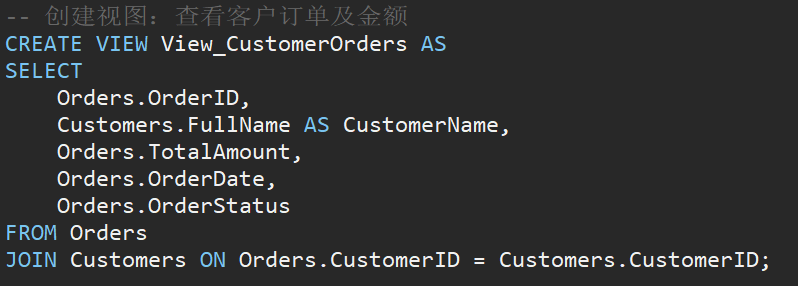




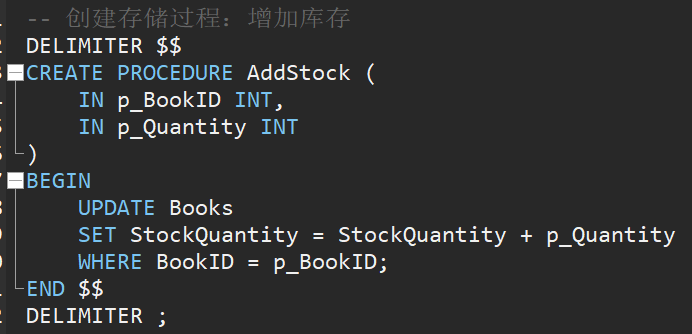
这些表格通过外键进行关联，确保数据的一致性和完整性。例如，订单表通过顾客ID与顾客信息表进行关联；采购订单表通过供应商ID与供应商信息表关联，以便准确记录每个订单的内容和采购需求。

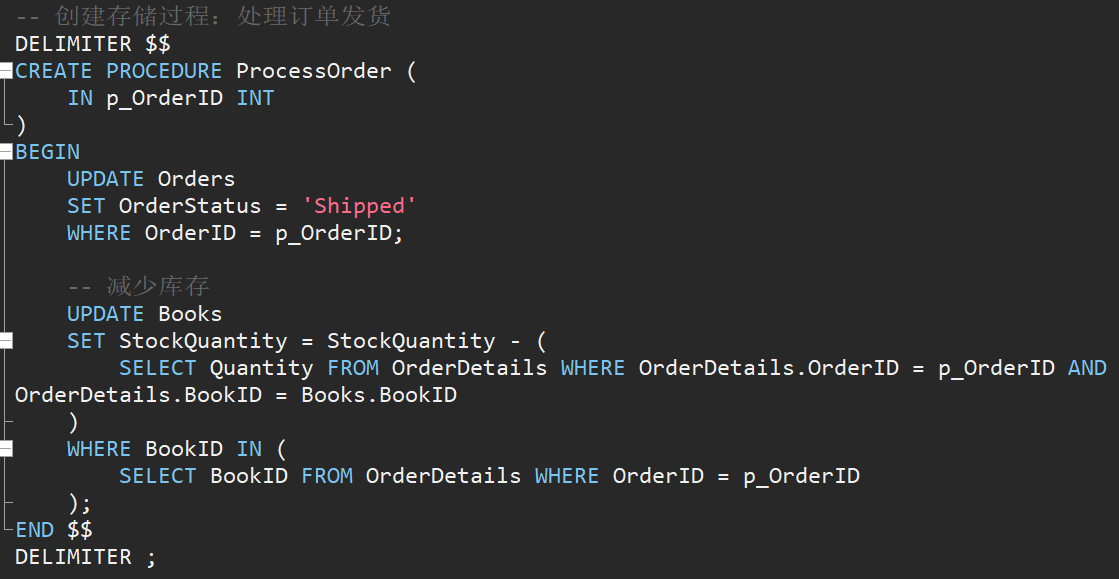
视图设计如下：





存储过程设计如下：

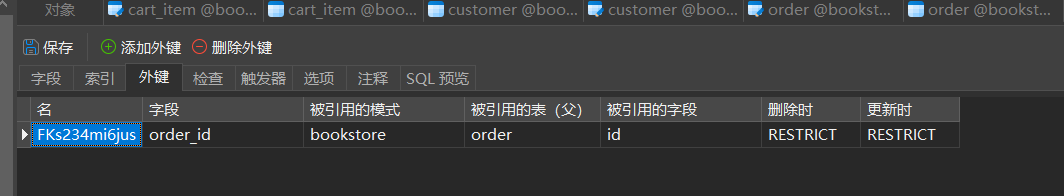




## 3.2. 数据库中的约束和触发器

为了确保数据的完整性和一致性，在数据库设计中，我们为各个表添加了多种约束和触发器。约束包括：

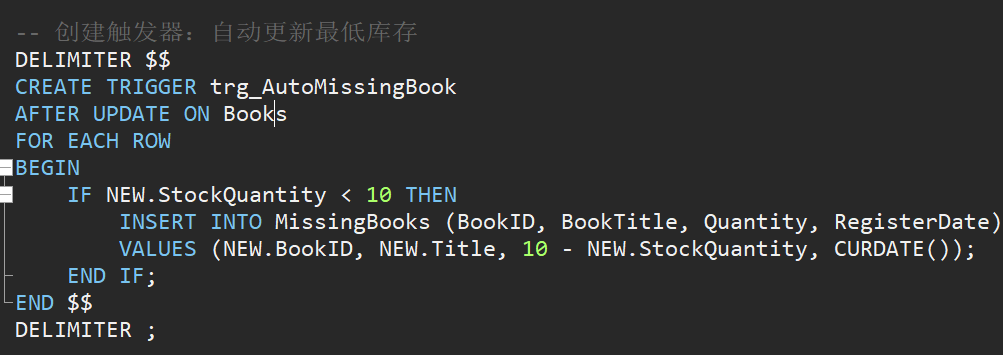
* 主键约束：每个表的主键字段如书籍ID、顾客ID、订单ID等都设置为主键，确保每条记录的唯一性。
* 外键约束：通过外键来保持表之间的关联，确保一条数据在另一个表中存在时才允许插入。例如，订单表中的顾客ID必须在顾客表中存在，采购记录中的书籍ID必须在书籍表中存在。例如order\_item中的外键order\_id。



* 检查约束：某些字段（如价格、库存量、余额等）需要确保符合一定的数值范围，我们通过设置检查约束来确保这些字段的合法性。

为了进一步提升系统的自动化和减少人为错误，系统还设计了触发器。触发器主要用于在某些事件发生时自动执行特定操作。

当订单的付款状态更新时，系统可以自动更新顾客的账户余额；当书籍的库存量更新时，系统会检查是否需要生成缺书记录或者更新库存警戒线。下图为自动更新最低库存触发器。



## 3.3. 数据库的性能优化

在数据库设计过程中，我们也充分考虑了系统性能的优化问题。以下是一些性能优化策略：

* 索引优化：为了提高查询速度，对于一些常用的查询条件字段（如书号、书名、作者、顾客ID、订单号等）设置了索引。索引可以大幅度提高查询效率，尤其是在涉及多表联接和大量数据检索时。
* 分区表设计：对于大数据量的表（如订单表和采购表），我们设计了分区表结构。通过按时间或者其他条件对表进行分区，能够有效提高查询速度并减少数据存储压力。
* 数据冗余：在某些业务场景下，考虑到查询效率和系统负载，设计了少量的数据冗余。例如，为了避免频繁的联接操作，顾客的信用等级和折扣信息可以存储在顾客表中，以便快速查询。

## 3.4. 系统中的安全性设计

在数据库的安全性设计方面，我们主要从以下几方面进行了考虑：

* **数据访问控制**：不同的用户（如管理员、普通顾客等）会有不同的权限，数据库的设计确保了数据访问权限的控制。例如，管理员可以访问和修改所有的表格，而普通顾客只能查询书目信息和下单，不能查看其他顾客信息。
* **加密存储**：为了保护顾客的敏感信息（如密码、账户余额等），数据库设计时对这些字段进行了加密存储处理。顾客的密码通过哈希算法进行存储，确保即使数据库被攻破，密码也无法泄露。
* **审计日志**：系统对每次数据库操作（如插入、删除、修改数据等）进行记录，并生成审计日志。通过审计日志，管理员可以追踪到每个操作的具体时间和操作人，有助于及时发现和解决潜在的安全问题。

## 3.5. 数据库的扩展性和可维护性

考虑到系统未来可能的扩展需求，数据库设计中保持了较高的可扩展性：

* **模块化设计**：每个业务模块如书籍管理、采购管理、客户管理等都独立设计，并通过外键和索引进行关联，方便后期的维护和扩展。
* **标准化和文档化**：数据库设计遵循一定的命名规范和数据类型标准，同时为每个表格和字段都提供了详细的注释，方便后续开发人员理解和维护。
* **备份与恢复策略**：为了保证数据的安全，系统设计了定期备份机制，同时提供了数据恢复功能。在发生系统故障或数据丢失时，可以快速恢复。

## 3.6. 数据库管理与操作

为了保证数据库的高效运行和维护，系统中还设计了多个管理功能。例如，数据库定期执行优化操作，清理不再使用的数据，归档历史数据等。同时，对于频繁执行的查询操作和数据更新，数据库管理员可以通过执行优化计划来提升性能，确保系统在高并发下的稳定性和响应速度。

## 3.7总结

本数据库系统的设计充分考虑了书店日常管理需求与顾客购物体验，围绕业务流程和数据流进行模块化设计。数据库通过表格结构实现了对书籍、客户、订单、供应商等各个业务实体的管理，并通过索引、分区、触发器等技术手段提升系统的性能和可维护性。同时，系统在安全性方面也进行了充分考虑，确保了顾客的隐私和数据的安全。总体来说，本次数据库设计满足了书店运营和顾客购物的多样化需求，并且为未来的扩展和维护奠定了良好的基础。

# 架构设计

本项目采用分层架构进行系统设计，这种架构方式不仅提高了系统的可维护性和可扩展性，而且通过模块化和分层管理，使得各个层次的职责更加明确，便于开发和后期的维护。

## 4.1. 表现层（Presentation Layer）

前端框架：使用 Vue3 框架构建用户友好的前端界面，利用其组件化特性，实现高度可复用的UI组件。

响应式设计：通过媒体查询和灵活的布局设计，确保用户在不同尺寸的设备上都能获得良好的阅读和操作体验。

动态页面展示：实现动态内容加载，减少页面刷新，提高用户体验。涵盖以下功能模块：

书目浏览：提供搜索、分类浏览、推荐等功能，帮助用户快速找到所需书籍。

订单管理：允许用户查看订单状态、历史订单，以及进行订单的支付、取消等操作。

客户信息查看：用户可以查看和编辑个人信息，包括地址、联系方式等。

购物车管理：实现购物车功能，包括添加、删除商品，以及结算等操作。

## 4.2. 业务逻辑层（Business Logic Layer）

核心数据处理：负责处理应用的核心业务逻辑，如用户认证、订单处理、库存管理等。

数据库交互：通过数据访问对象（DAO）模式，实现业务逻辑层与数据层的交互，确保数据操作的封装性和独立性。

业务规则实现：实现业务规则，如库存检查、价格计算、促销活动等，确保业务流程的准确性和一致性。

服务层：提供服务接口，供表现层调用，实现业务逻辑的集中管理和复用。

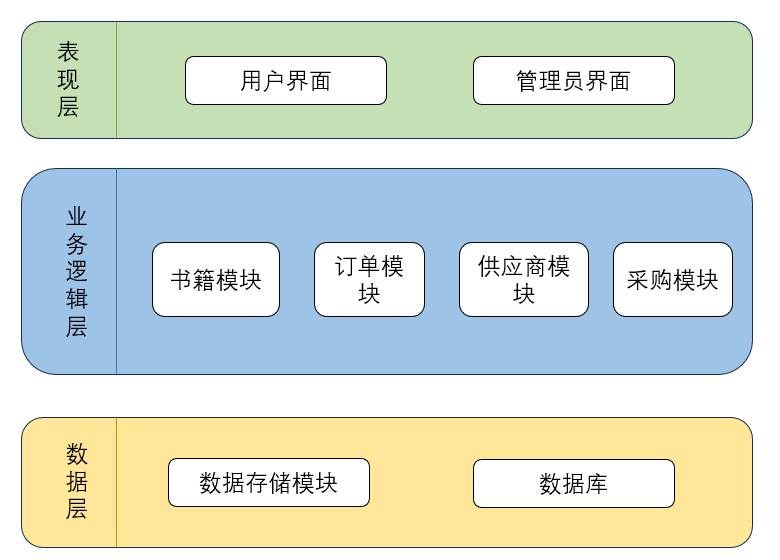
## 4.3. 数据层（Data Layer）

数据库设计：设计合理的数据库模式，确保数据的完整性和一致性，支持高效的数据存储和检索。

数据存储解决方案：选择合适的数据库技术，如关系型数据库或NoSQL数据库，根据数据特点和访问模式进行优化。

数据安全性：实施数据加密、备份和恢复策略，确保数据的安全性和可靠性。

数据访问性能：通过索引、查询优化等手段，提高数据访问性能，满足业务逻辑层的需求。



通过这样的架构设计，网上书店项目将能够提供一个稳定、高效、可扩展的在线购书平台，满足用户和书店的多样化需求。

# 前端设计

本项目的前端设计主要围绕Vue 3框架进行，利用其响应式和组件化的特性来构建一个高效、可维护的用户界面。以下是前端设计的关键点：

## 5.1 技术栈选择

Vue 3：作为前端核心框架，Vue 3提供了更小的打包体积、更快的虚拟DOM、组合式API等优势，使得开发更加灵活和高效。

JavaScript (ES6+)：使用现代JavaScript语法，提高代码的可读性和维护性。

CSS预处理器：增强CSS的可维护性和复用性。

Axios：用于处理前端与后端的HTTP请求，实现数据的获取和提交。

Vue Router：实现前端路由管理，支持页面的动态加载和导航。

Vuex：状态管理库，用于在组件间共享状态，特别是对于复杂的应用。

## 5.2 组件化设计

UI组件库：使用或开发一套UI组件库，包括按钮、输入框、下拉菜单等，确保界面的一致性和可复用性。

页面级组件：将每个页面或功能模块设计为一个独立的Vue组件，如书籍列表组件、订单详情组件等。

可复用组件：开发可在整个应用中复用的通用组件，如对话框、提示信息、分页控件等。

## 5.3 响应式设计

媒体查询：使用CSS媒体查询，确保界面在不同尺寸的设备上都能良好展示。

Flexbox和Grid布局：利用Flexbox和Grid布局，实现灵活且响应式的页面布局。

视口单位：使用视口宽度（vw）和视口高度（vh）等单位，使元素尺寸能够根据视口大小动态调整。

## 5.4 状态管理

Vuex的使用：通过Vuex管理全局状态，如用户认证信息、购物车内容等。

模块化状态管理：将状态管理模块化，便于不同功能模块独立管理自己的状态。

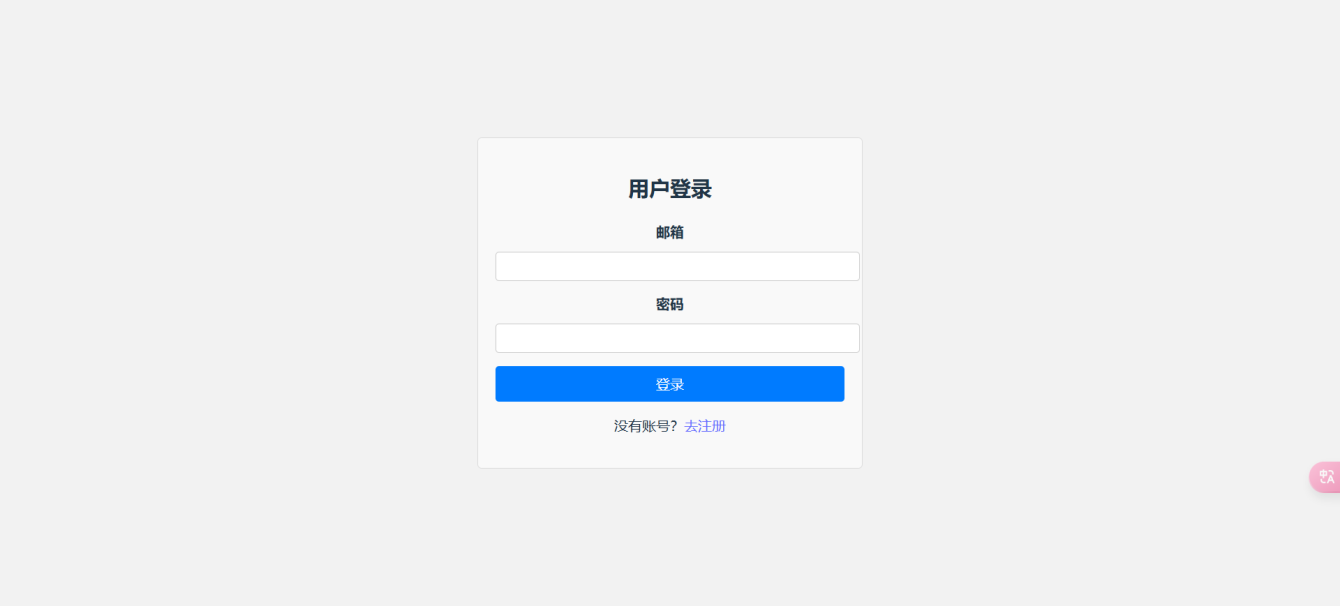
## 5.5 路由管理

动态路由：使用Vue Router实现动态路由，支持页面的懒加载和嵌套路由。

路由守卫：利用路由守卫进行权限控制，如未登录用户不能访问订单页面。

## 5.6 页面展示

### 5.6.1 用户登录/注册

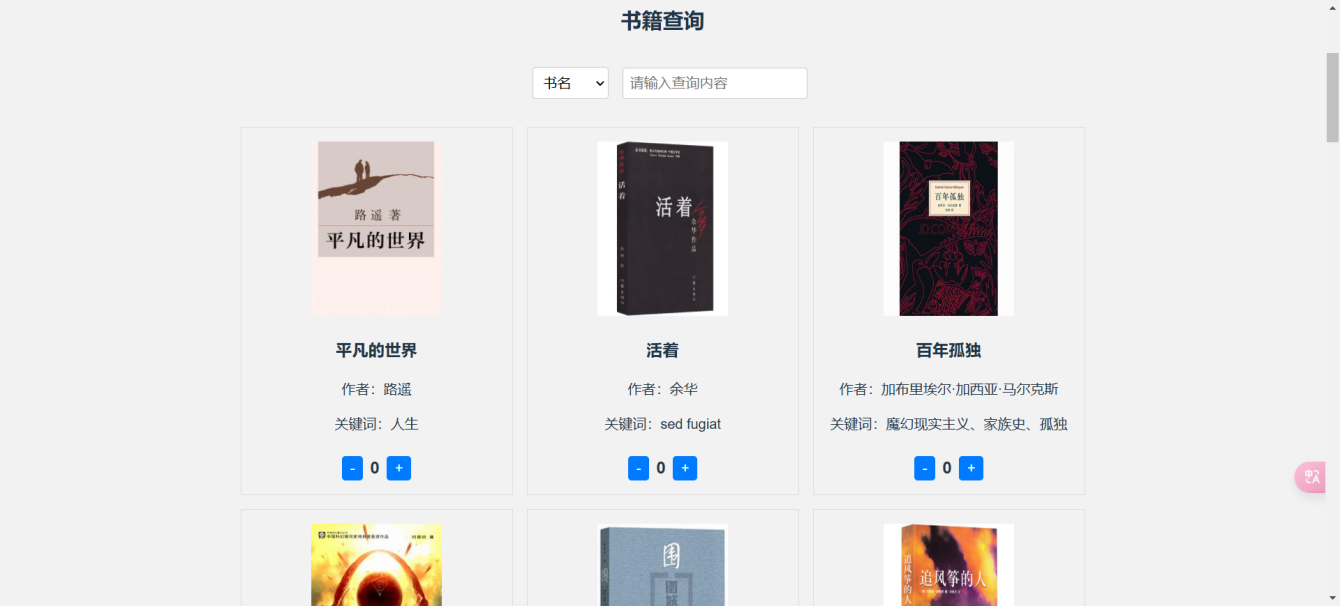


使用邮箱和密码进行登录。



输入用户名，邮箱，密码登录进行用户注册。邮箱不能重复。

### 5.6.2 书籍信息页面



用户可以进行书籍的模糊查询，并可选择书名，书号，作者，关键词等搜索。点击+或-可以进行书籍的购买或减少。点击单个书籍进行书籍详情界面。



在书籍详情界面会显示单个书籍更加详细的信息，如存储量。同时也可以进行书籍的购买，这里的购买量会同步购物车中的购买量。

### 5.6.3 购物车页面



在书籍总览页面和详情页面购买的书籍及数量存储在购物车中，这里可以进行删除和修改数量操作，总计金额为根据用户信用等级打折后的价格，点击结算即清空购物车，生成订单。

### 5.6.4 订单页面



结算后的订单，有已发货，未发货等状态，在未发货状态可以直接进行取消操作，余额将返回账户。

### 5.6.5 用户信息页面



用户的详细信息。并可以调整用户名，邮件，地址等基础信息。通过充值进行余额的提升。信用等级由累计充值的金额来自动判断。



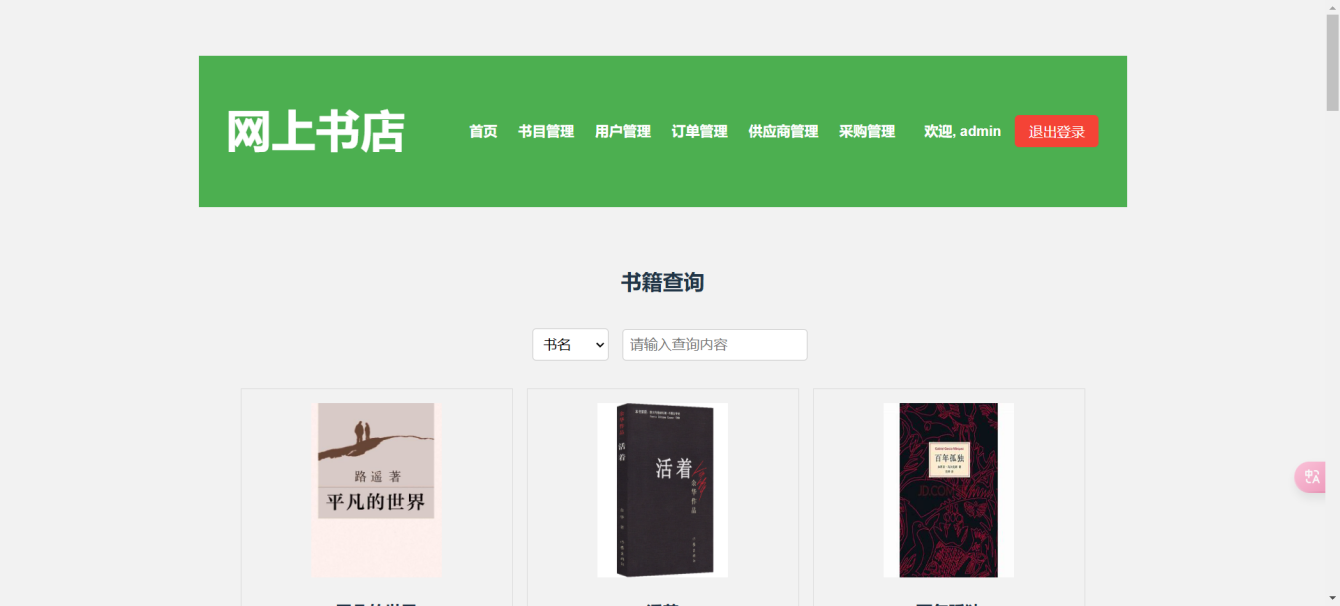


### 5.6.6 留言页面



用户可以在此处留言。

### 5.6.7 管理员首页



管理员首页。导航栏可以跳转至书目管理，用户管理，订单管理，供应商管理，采购管理。

### 5.6.8 书籍管理页面



对于库存中的书籍可以进行具体的信息的编辑和新书入库工作。

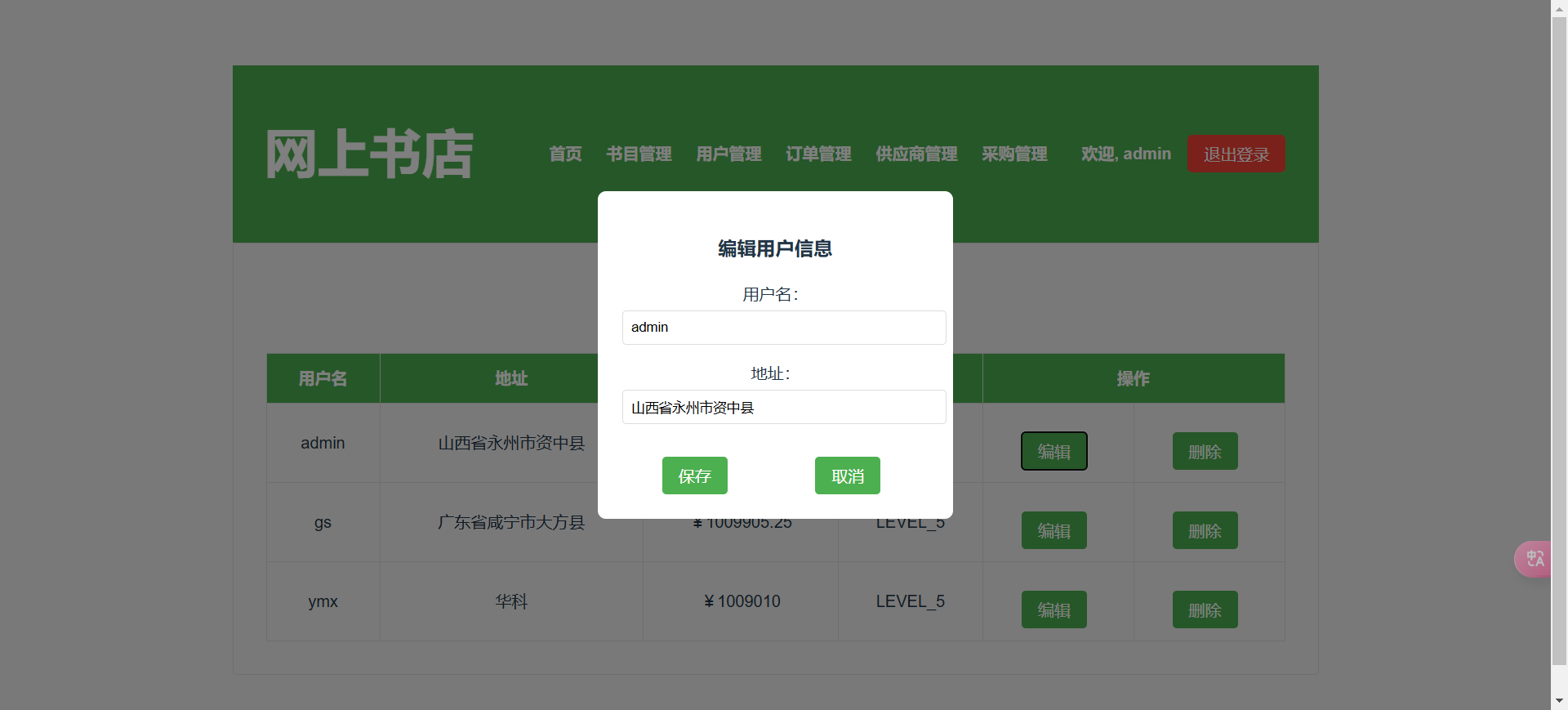




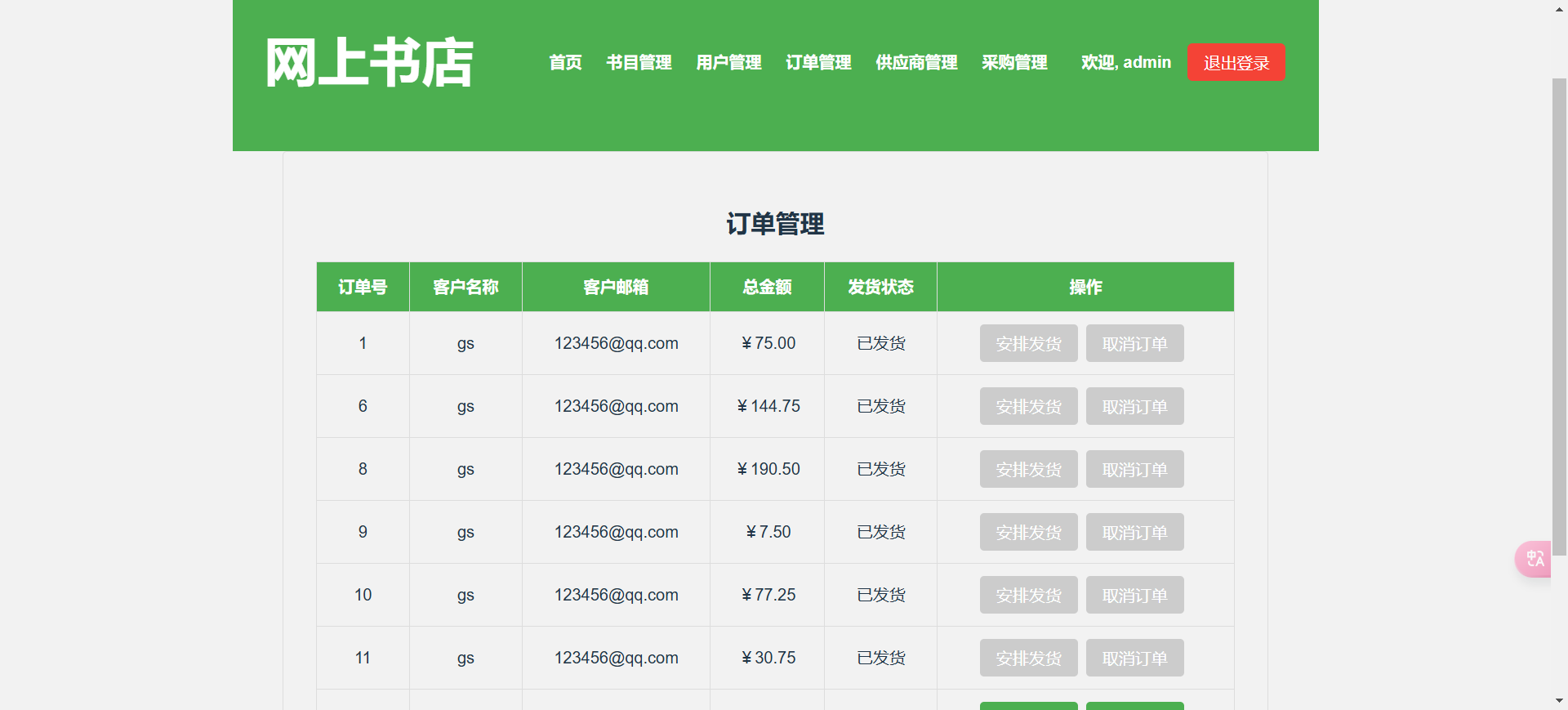
### 5.6.9 用户管理页面



可以调整用户的名称，地址等信息，并可以删除用户的账号。



### 5.6.10 订单管理页面



对于用户产生的订单，可以进行发货操作或者是取消订单。

### 5.6.11 供应商管理页面



增加，修改，删除供应商







对供应商供应的书目进行增加，删除操作。

### 5.6.11 采购管理页面



点击检查库存并生成采购单，可以自动检查缺货（低于最低库存量）的书籍信息，点击更新库存可以进行更新，点击删除可以忽略此次采购。

# 后端配置

基于之前的需求分析和数据库设计，现在聚焦于系统的后端实现。在后端开发过程中，我们使用了Spring Boot框架，配合Spring Security进行安全控制，Spring Data JPA进行数据库操作，以及JWT实现用户认证。

## 6.1. Maven 配置

本项目使用Maven作为构建工具，依赖项包括了Spring Boot核心库、Spring Data JPA、MySQL数据库驱动、JWT、Spring Security以及Spring Boot开发工具等。主要依赖有：

* Spring Boot Web Starter (spring-boot-starter-web)：用于构建Web应用，支持RESTful API。
* Spring Boot Data JPA (spring-boot-starter-data-jpa)：用于简化与数据库交互的开发工作，自动配置JPA，实现数据库操作。
* MySQL Driver (mysql-connector-java)：用于连接MySQL数据库。
* Spring Boot Validation (spring-boot-starter-validation)：用于验证输入数据的有效性（如表单提交时的字段验证）。
* Spring Security (spring-boot-starter-security)：用于控制应用的安全性，配置用户认证和授权。
* JWT依赖 (jjwt，jjwt-api，jjwt-impl等)：用于生成和解析JWT令牌，实现用户认证。
* Lombok：简化Java代码，自动生成getter/setter等方法。

Maven的pom.xml文件配置了项目所需的依赖，使得我们能够轻松集成这些技术，快速构建和管理项目。

## 6.2. 主应用类（BookstoreApplication）

@SpringBootApplicationpublic class BookstoreApplication {

public static void main(String[] args) {

SpringApplication.run(BookstoreApplication.class, args);

}

}

主应用类BookstoreApplication是Spring Boot应用的入口点。通过@SpringBootApplication注解，Spring Boot会自动配置所需的环境，并启动应用。这是典型的Spring Boot应用结构，启动时会加载所有配置并进行自动注入。

## 6.3. CORS 配置（CorsConfig）

在CorsConfig类中，我们配置了跨域资源共享（CORS），允许来自http://localhost:5173的请求进行跨域访问。这对于前后端分离的应用非常重要，确保前端应用可以顺利与后端通信，特别是在开发环境中，前端和后端通常会运行在不同的端口上。

* addAllowedOrigin("http://localhost:5173")：指定允许访问的前端地址。
* addAllowedHeader("\*")：允许所有的HTTP头。
* addAllowedMethod("\*")：允许所有HTTP方法（GET、POST、PUT、DELETE等）。
* setAllowCredentials(true)：允许客户端发送凭证（如cookies）。

## 6.4. 安全配置（SecurityConfig）

SecurityConfig类负责配置Spring Security以保护API的安全。主要功能包括：

* 禁用CSRF：在RESTful API中，通常不需要CSRF防护，因此使用csrf().disable()禁用。
* 权限控制：
  + /customers/register 和 /customers/login 允许匿名访问。
  + /customers/\*\* 路径需要普通用户角色（USER）权限。
  + /customers/admin/\*\* 路径需要管理员角色（ADMIN）权限。
  + 其他路径都需要认证。
* JWT过滤器：jwtAuthenticationFilter会在所有请求中间执行，用于验证JWT令牌是否有效。
* 密码加密：使用BCryptPasswordEncoder对用户的密码进行加密，增强系统的安全性。

## 6.5. 数据库连接配置（application.properties）

spring.datasource.url=jdbc:mysql://localhost:3306/bookstore?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

jwt.secret=041f65e9-176d-4e4c-aa68-ab3400f0d58c

数据库连接配置的application.properties文件中，包含了连接MySQL数据库的详细信息：

* spring.datasource.url：指定了MySQL数据库的连接URL，bookstore为数据库名称。
* spring.datasource.username 和 spring.datasource.password：提供数据库的用户名和密码。
* spring.jpa.hibernate.ddl-auto=update：此配置告诉Spring JPA在启动时自动更新数据库模式。
* jwt.secret：用于生成JWT令牌的密钥。这个密钥将用于加密和验证JWT令牌。

## 6.6. JWT认证过滤器（JwtAuthenticationFilter）

JwtAuthenticationFilter是系统中的一个核心组件，用于拦截请求并验证JWT令牌的有效性。其作用是：

* 提取JWT：从请求头中提取Authorization字段，获取JWT令牌。
* 验证JWT：使用预先设定的jwt.secret密钥验证JWT的合法性。
* 设置认证信息：如果JWT有效，则设置Authentication对象，表示当前请求的用户已通过认证。

该过滤器将会被添加到Spring Security的过滤链中，保证所有请求在进入Controller之前都经过JWT验证。

## 6.7. 总结

通过Spring Boot框架，可以非常方便地集成数据库、前端接口、验证机制等，同时保证系统的安全性。

* 使用Spring Data JPA进行数据库操作，简化了与数据库的交互。
* 使用JWT进行无状态的用户认证和授权控制，避免了传统的会话存储，确保了系统的高效性和扩展性。
* 配置Spring Security进行细粒度的访问控制，确保每个接口的权限得到严格控制。

# 后端各模块功能的实现

## 7.1. 图书管理模块

图书管理模块通过Spring Boot和Spring Data JPA实现了一个完整的图书管理功能，包括图书的基本管理、库存管理、供应商管理等。后端的设计注重业务逻辑的分层处理，使得每一层的职责清晰，代码易于维护和扩展。通过RESTful API接口，前端能够方便地进行图书的增删改查操作，确保了系统的灵活性和可扩展性。

### 7.1.1. 控制器层（BookController）

BookController是一个RESTful API的控制器，用于处理前端请求并将请求转发给BookService进行处理。所有的请求都以/api/books为基础路径，并通过HTTP方法映射到对应的业务功能。该控制器包含了多个操作接口：

**获取所有图书**：通过GET /api/books获取所有书籍的信息。

@GetMapping

public List<Book> getAllBooks() {

return bookService.getAllBooks();

}

这个方法调用BookService中的getAllBooks()，返回所有图书的列表。

**添加新图书**：通过POST /api/books添加一本新的图书。

@PostMapping

public ResponseEntity<Book> addBook(@RequestBody Book book) {

return ResponseEntity.ok(bookService.addBook(book));

}

该方法接收一个Book对象作为请求体，通过bookService.addBook()保存图书。

**更新图书信息**：通过PUT /api/books/{id}更新指定ID的图书信息。

@PutMapping("/{id}")

public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book book) {

return ResponseEntity.ok(bookService.updateBook(id, book));

}

**删除图书**：通过DELETE /api/books/{id}删除指定ID的图书。

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteBook(@PathVariable Long id) {

bookService.deleteBook(id);

return ResponseEntity.noContent().build();

}

**查询图书**：通过GET /api/books/{id}查询指定ID的图书。

@GetMapping("/{id}")

public ResponseEntity<Book> getBookById(@PathVariable Long id) {

return ResponseEntity.ok(bookService.getBookById(id));

}

**根据书号查询书籍**：通过GET /api/books/bookNumber/{bookNumber}根据书号查询图书。

@GetMapping("/bookNumber/{bookNumber}")

public ResponseEntity<List<Book>> getBooksByBookNumber(@PathVariable String bookNumber) {

return ResponseEntity.ok(bookService.getBooksByBookNumber(bookNumber));

}

**根据供应商查询书籍**：通过GET /api/books/supplier/{supplierId}根据供应商查询书籍。

@GetMapping("/supplier/{supplier}")

public ResponseEntity<List<Book>> getBooksBySupplier(@PathVariable Supplier supplier) {

return ResponseEntity.ok(bookService.getBooksBySupplier(supplier));

}

库存操作：

**新书入库**：通过POST /api/books/{id}/stock增加指定图书的库存。

@PostMapping("/{id}/stock")

public ResponseEntity<Book> addNewStock(@PathVariable Long id, @RequestParam int quantity) {

return ResponseEntity.ok(bookService.addNewStock(id, quantity));

}

**更新库存数量**：通过PUT /api/books/{id}/stock更新指定图书的库存数量。

@PutMapping("/{id}/stock")

public ResponseEntity<Book> updateStockQuantity(@PathVariable Long id, @RequestParam int quantity) {

return ResponseEntity.ok(bookService.updateStockQuantity(id, quantity));

}

供应商管理：

**添加供应商到图书**：通过POST /api/books/{id}/supplier/{supplierId}为指定图书添加供应商。

@PostMapping("/{id}/supplier/{supplierId}")

public ResponseEntity<Book> addSupplierToBook(@PathVariable Long id, @PathVariable Long supplierId) {

return ResponseEntity.ok(bookService.addSupplierToBook(id, supplierId));

}

**移除供应商**：通过DELETE /api/books/{id}/supplier/{supplierId}从图书中移除供应商。

@DeleteMapping("/{id}/supplier/{supplierId}")

public ResponseEntity<Book> removeSupplierFromBook(@PathVariable Long id, @PathVariable Long supplierId) {

return ResponseEntity.ok(bookService.removeSupplierFromBook(id, supplierId));

}

BookController通过调用BookService层来实现所有具体的业务逻辑，并将结果返回给客户端。控制器层注重于请求的接收和响应，而具体的业务处理则交给BookService。

### 7.1.2. 模型层（Book）

Book类是图书的实体类，它映射到数据库中的books表。每本图书的基本信息（如书号、书名、作者、出版社、价格、库存量等）都在该类中定义。关键字段包括：

图书基本信息：

bookNumber：书号。

title：书名。

authors：作者（逗号分隔）。

publisher：出版社。

price：价格。

keywords：关键字（最多10个）。

directory：目录。

coverImage：封面图。

stockQuantity：库存数量。

关联字段：

suppliers：通过@ManyToMany注解实现与供应商的多对多关联。每本书可能有多个供应商。

库存管理：

minStockQuantity：最低库存量，低于此数量时可以触发缺书登记等流程。

此类通过@Entity注解标识为JPA实体类，并与数据库中的books表进行映射。

### 7.1.3. 数据访问层（BookRepository）

BookRepository是Spring Data JPA提供的接口，它继承自JpaRepository<Book, Long>，提供了对Book实体的基本CRUD操作。它还提供了以下自定义查询方法：

findByBookNumber(String bookNumber)：根据书号查询书籍。

findBySuppliersContaining(Supplier supplier)：根据供应商查询书籍。

findByStorageLocation(String storageLocation)：根据库存存放位置查询书籍。

Spring Data JPA会根据方法名自动生成对应的SQL查询，因此不需要手动编写复杂的查询语句。

### 7.1.4. 服务层（BookService）

BookService层负责处理与图书相关的业务逻辑，包括CRUD操作、库存管理、供应商管理等。该层通过注入BookRepository和SupplierRepository，实现了以下功能：

获取图书列表：getAllBooks()获取所有图书信息。

添加新图书：addBook(Book book)将新的图书保存到数据库。

更新图书信息：updateBook(Long id, Book book)根据ID更新图书信息。

删除图书：deleteBook(Long id)删除指定ID的图书。

根据ID查询图书：getBookById(Long id)根据图书ID查询图书。

根据书号查询书籍：getBooksByBookNumber(String bookNumber)根据书号查询图书。

根据供应商查询书籍：getBooksBySupplier(Supplier supplier)根据供应商查询图书。

库存管理：

updateStockQuantity(Long id, int quantity)更新库存数量。

addNewStock(Long id, int quantity)增加库存量。

供应商管理：

addSupplierToBook(Long bookId, Long supplierId)将供应商添加到图书中。

removeSupplierFromBook(Long bookId, Long supplierId)将供应商从图书中移除。

此外，BookService还负责库存监控，通过checkStockAndGenerateOutOfStockRecords()检查库存并生成缺书记录。

## 7.2. 购物车管理模块

购物车管理模块的后端代码设计采用了Spring Boot的标准开发模式，基于Spring Data JPA进行数据存取，利用RESTful API暴露业务操作，符合前后端分离的设计理念。

* 控制器层（CartController）：通过RESTful API接口，暴露了购物车相关的操作接口，支持购物车的增删改查、结算等操作。
* 模型层（Cart 和 CartItem）：设计了购物车和购物车项模型，定义了购物车与书籍的关系，并通过calculateTotalAmount()方法提供了总金额计算功能。
* 数据访问层（CartRepository）：通过Spring Data JPA接口实现与数据库的交互，提供了查询购物车的功能。
* 服务层（CartService）：封装了购物车的核心业务逻辑，包括添加书籍、减少数量、删除书籍、清空购物车等功能。

### 7.2.1. 控制器层（CartController）

CartController负责暴露购物车相关的RESTful API接口，主要功能包括获取购物车内容、向购物车添加书籍、减少书籍数量、删除书籍和结算购物车等。每个API的路径是/api/carts，并且通过@RequestMapping注解定义了操作路径。详细功能如下：

**获取指定客户的购物车**：GET /api/carts/customer/{customerId}：根据客户ID获取客户的购物车。如果客户没有购物车，则会创建一个新的购物车。

@GetMapping("/customer/{customerId}")

public ResponseEntity<Cart> getCartByCustomerId(@PathVariable Long customerId) {

return ResponseEntity.ok(cartService.getCartByCustomerId(customerId));

}

**向购物车中添加书籍**：POST /api/carts/customer/{customerId}/add：将书籍添加到客户的购物车中。该请求包括书籍ID和数量参数。

@PostMapping("/customer/{customerId}/add")public ResponseEntity<Cart> addBookToCart(@PathVariable Long customerId,@RequestParam Long bookId,@RequestParam Integer quantity) {

return ResponseEntity.ok(cartService.addBookToCart(customerId, bookId, quantity));

}

**减少书籍数量**：POST /api/carts/customer/{customerId}/reduce：减少购物车中指定书籍的数量。如果数量减少至零，书籍会从购物车中删除。

@PostMapping("/customer/{customerId}/reduce")public ResponseEntity<Cart> reduceBookQuantity(@PathVariable Long customerId,@RequestParam Long bookId,

@RequestParam Integer quantity) {

return ResponseEntity.ok(cartService.reduceBookQuantity(customerId, bookId, quantity));

}

**删除购物车中的书籍**：DELETE /api/carts/customer/{customerId}/delete：从购物车中删除指定的书籍。

@DeleteMapping("/customer/{customerId}/delete")public ResponseEntity<Cart> deleteBookFromCart(@PathVariable Long customerId, @RequestParam Long bookId) {

return ResponseEntity.ok(cartService.deleteBookFromCart(customerId, bookId));

}

**结算购物车并创建订单**：POST /api/carts/customer/{customerId}/checkout：将购物车中的商品结算并生成订单。结算时会清空购物车并创建一个新的订单。

@PostMapping("/customer/{customerId}/checkout")public ResponseEntity<Order> checkoutCart(@PathVariable Long customerId) {

try {

Order order = orderService.createOrderFromCart(customerId);

return ResponseEntity.ok(order);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null); // 返回错误状态码

}

}

### 7.2.2. 模型层（Cart 和 CartItem）

Cart是购物车实体，CartItem是购物车中的每一项（即每一本书）。这两者构成了购物车的基本模型：

**Cart 类：**

Cart类与Customer是一对一关系，每个客户只能有一个购物车。

Cart包含多个CartItem，每个CartItem表示购物车中的一个书籍项。

Cart类还提供了calculateTotalAmount()方法，用于计算购物车的总金额（即所有书籍的总价）。

@OneToOne@JoinColumn(name = "customer\_id", nullable = false, unique = true)private Customer customer;

@OneToMany(mappedBy = "cart", cascade = CascadeType.ALL, orphanRemoval = true)private List<CartItem> items = new ArrayList<>(); // 购物车中的书籍列表

**CartItem 类：**

CartItem类是每一项书籍的数据，包含书籍ID、数量、以及该书籍所属的Cart（通过外键关系）。

CartItem与Cart的关系是多对一的，即多个CartItem对应一个Cart。

@ManyToOne@JoinColumn(name = "cart\_id", nullable = false)@JsonIgnoreprivate Cart cart;

### 7.2.3. 数据访问层（CartRepository）

CartRepository是Spring Data JPA提供的接口，继承了JpaRepository<Cart, Long>，用于对Cart实体进行数据库操作。通过定义findByCustomerId()方法，可以根据客户ID查询购物车。Spring Data JPA会根据方法名自动生成SQL查询，无需手动编写。

public interface CartRepository extends JpaRepository<Cart, Long> {

Optional<Cart> findByCustomerId(Long customerId); // 根据客户ID查询购物车

}

### 7.2.4. 服务层（CartService）

CartService是购物车的业务逻辑层，负责处理与购物车相关的操作。它通过注入CartRepository和CustomerRepository，提供了以下功能：

* 获取购物车：getCartByCustomerId(Long customerId)方法查询或创建购物车。
* 向购物车添加书籍：addBookToCart(Long customerId, Long bookId, Integer quantity)方法向购物车中添加书籍。
* 减少书籍数量：reduceBookQuantity(Long customerId, Long bookId, Integer quantity)方法减少购物车中书籍的数量。
* 删除书籍：deleteBookFromCart(Long customerId, Long bookId)方法从购物车中删除书籍。
* 清空购物车：clearCart(Long customerId)方法清空购物车中的所有书籍。

public Cart getCartByCustomerId(Long customerId) {

var cart = cartRepository.findByCustomerId(customerId)

.orElseGet(() -> createCartForCustomer(customerId));

if (cart == null) {

throw new RuntimeException("Cart not found for customer ID: " + customerId);

}

return cart;

}

CartService通过创建和操作Cart及CartItem对象，实现了购物车的增删改查功能。它将所有的操作集中在一个服务层，符合服务层设计的单一职责原则。

CartService还涉及到购物车的结算与订单生成。当顾客准备结算时，checkoutCart方法会调用OrderService来创建订单，并将购物车中的所有商品清空。结算后的购物车会转换为一个订单，并且购物车内容会被清空，确保顾客的购物车状态一致。

public ResponseEntity<Order> checkoutCart(@PathVariable Long customerId) {

try {

Order order = orderService.createOrderFromCart(customerId);

return ResponseEntity.ok(order);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null); // 返回错误状态码

}

}

## 7.3. 用户管理与认证模块

本模块设计了一个完整的用户管理与认证系统，包括用户注册、登录、信息更新、权限控制等功能：

* 使用Spring Security和JWT结合，实现了灵活的身份验证和授权控制。
* 用户密码通过BCrypt进行加密，增强了系统的安全性。
* 通过JWT Token在不同的请求之间传递用户身份信息，简化了会话管理。
* 在客户信息更新、余额操作等敏感操作时，实现了权限控制，确保用户只能操作自己的数据。

### 7.3.1. 控制器层（CustomerController）

CustomerController类是一个RESTful API控制器，提供了与用户（Customer）相关的各项功能接口，如用户信息的增删改查、登录、注册、账户余额更新等。具体功能如下：

**获取所有客户**：GET /customers，该接口返回所有客户的列表。

@GetMappingpublic List<Customer> getAllCustomers() {

return customerService.getAllCustomers();

}

**根据用户ID查询客户信息**：GET /customers/{id}，该接口根据用户ID返回该用户的详细信息。该接口具有权限控制，仅允许当前用户或管理员访问自己的信息。

@GetMapping("/{id}")public ResponseEntity<?> getCustomerById(@PathVariable Long id, HttpServletRequest request) {

Long currentUserId = getCurrentUserId(request);

if (!currentUserId.equals(id) && !currentUserId.equals(1L)) {

return ResponseEntity.status(403).body("You don't have permission to view this user.");

}

try {

Customer customer = customerService.getCustomerById(id);

return ResponseEntity.ok(customer);

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(e.getMessage());

}

}

**创建客户**：POST /customers，该接口允许创建新的用户。

@PostMappingpublic Customer createCustomer(@RequestBody Customer customer) {

return customerService.createCustomer(customer);}

**更新客户信息**：PUT /customers/{id}，该接口允许更新客户的名称、邮箱和地址等信息。用户仅能更新自己的信息，管理员可以更新任何用户的信息。

@PutMapping("/{id}")public ResponseEntity<?> updateCustomer(

@PathVariable Long id,

@RequestBody Map<String, Object> updates,

HttpServletRequest request) {

Long currentUserId = getCurrentUserId(request);

if (!currentUserId.equals(id) && !currentUserId.equals(1L)) {

return ResponseEntity.status(403).body("You don't have permission to update this user.");

}

try {

Customer updatedCustomer = customerService.updateCustomerFields(id, updates);

return ResponseEntity.ok(updatedCustomer);

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(e.getMessage());

}

}

**删除客户**：DELETE /customers/{id}，该接口允许管理员删除客户信息。管理员权限由ID为1的用户表示。

@DeleteMapping("/{id}")public ResponseEntity<?> deleteCustomer(@PathVariable Long id, HttpServletRequest request) {

Long currentUserId = getCurrentUserId(request);

if (!currentUserId.equals(1L)) {

return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Only the admin (ID: 1) can delete users.");

}

try {

customerService.deleteCustomer(id);

return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(e.getMessage());

}

}

**用户注册**：POST /customers/register，该接口允许用户注册。注册时，系统会对用户密码进行加密。

@PostMapping("/register")public ResponseEntity<?> register(@RequestBody Customer customer) {

try {

Customer registeredCustomer = customerService.register(customer);

return ResponseEntity.status(HttpStatus.CREATED).body(registeredCustomer);

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body(e.getMessage());

}

}

用户登录：POST /customers/login，该接口用于用户登录，验证用户的密码并返回JWT Token，用户凭此Token可以进行后续的授权操作。

@PostMapping("/login")public ResponseEntity<?> login(@RequestBody Customer customer) {

try {

String token = customerService.login(customer.getEmail(), customer.getPassword());

Claims claims = jwtUtil.parseJwt(token);

Customer loggedCustomer = customerService.getCustomerById(claims.get("userId", Long.class));

Map<String, Object> response = new HashMap<>();

response.put("token", "Bearer " + token);

response.put("user", loggedCustomer);

return ResponseEntity.ok(response);

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(e.getMessage());

}

}

**更新用户余额**：PUT /customers/{id}/updateBalance，该接口允许更新用户的账户余额和信用等级。用户可以充值，也可以提现。@PutMapping("/{id}/updateBalance")public ResponseEntity<?> updateBalance(

@PathVariable Long id,

@RequestParam Double amount,

HttpServletRequest request) {

Long currentUserId = getCurrentUserId(request);

if (!currentUserId.equals(id) && !currentUserId.equals(1L)) {

return ResponseEntity.status(HttpStatus.FORBIDDEN)

.body("You don't have permission to perform this action.");

}

try {

Customer updatedCustomer = customerService.updateBalance(id, amount);

Map<String, Object> response = new HashMap<>();

response.put("totalRecharge", updatedCustomer.getTotalRecharge());

response.put("accountBalance", updatedCustomer.getAccountBalance());

response.put("creditLevel", updatedCustomer.getCreditLevel().toString());

response.put("discount", updatedCustomer.getCreditLevel().getDiscount());

response.put("overdraftLimit", updatedCustomer.getCreditLevel().getOverdraftLimit());

return ResponseEntity.ok(response);

} catch (RuntimeException e) {

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body(e.getMessage());

}

}

### 7.3.2. 服务层（CustomerService）

CustomerService提供了用户信息管理的业务逻辑，包括创建、更新、删除用户、注册、登录、余额管理等：

* 注册用户：在注册过程中，用户的密码会经过BCryptPasswordEncoder加密后存储，以确保安全性。
* 登录功能：在用户登录时，系统会验证密码是否正确，成功后生成JWT Token，并返回给客户端。JWT Token中包含了用户ID和角色信息（如管理员、普通用户等）。
* 更新余额与信用等级：系统支持用户充值和提现操作，并根据累计充值金额调整用户的信用等级。

public String login(String email, String rawPassword) {

Optional<Customer> customerOpt = customerRepository.findByEmail(email);

if (customerOpt.isEmpty()) {

throw new RuntimeException("User with email " + email + " not found");

}

Customer customer = customerOpt.get();

if (passwordEncoder.matches(rawPassword, customer.getPassword())) {

throw new RuntimeException("Invalid password");

}

Map<String, Object> claims = new HashMap<>();

claims.put("userId", customer.getId());

claims.put("email", customer.getEmail());

claims.put("role", customer.getId() == 1 ? "ADMIN" : "USER");

return jwtUtil.createJwt(43200000, claims);

}

### 7.3.3. ****JWT处理类（JwtUtil）****

JwtUtil负责生成和解析JWT Token。它通过HMACSHA256加密算法签名生成JWT，并将用户信息（如ID和角色）存储在JWT的claims中。JWT的有效期为12小时。

* 生成JWT：createJwt()方法用于生成JWT Token。
* 解析JWT：parseJwt()方法用于解析JWT Token，返回其中的claims。
* 验证JWT：validateJwt()方法用于验证JWT是否有效，检查Token是否过期。

public String createJwt(long ttlMillis, Map<String, Object> claims) {

Date expiration = new Date(System.currentTimeMillis() + ttlMillis);

SecretKey key = Keys.hmacShaKeyFor(secretKey.getBytes(StandardCharsets.UTF\_8));

return Jwts.builder()

.setClaims(claims)

.setExpiration(expiration)

.signWith(key, SignatureAlgorithm.HS256)

.compact();

}

### 7.3.4. ****自定义身份验证令牌（CustomAuthenticationToken）****

CustomAuthenticationToken是一个自定义的身份验证令牌类，继承自AbstractAuthenticationToken。它用于表示身份验证的主体（如用户ID）和凭证（如密码）。根据用户ID，它会自动分配角色（管理员或普通用户）。

## 7.4. 订单管理模块

订单管理模块实现了订单的创建、查询、更新、删除、取消和发货等功能，涵盖了从购物车结算到订单状态管理的全流程。通过Spring Data JPA、Spring Boot的注解和依赖注入，系统能够高效地管理订单和订单项的生命周期。

* 功能完备：支持顾客下单、订单查询、订单取消、订单发货等操作。
* 权限控制：顾客只能操作自己的订单，管理员可以查看和管理所有订单。
* 库存与余额管理：在订单创建和取消时，库存和顾客账户余额会进行相应的更新，确保系统的数据一致性。

### 7.4.1. ****控制器层（OrderController）****

OrderController负责暴露订单管理的API接口，主要功能包括订单的创建、查询、更新、删除、取消、发货等操作。每个API接口路径都定义了对应的业务逻辑，如下：

**结算购物车并创建订单**：POST/api/orders/customer/{customerId}/checkout，该接口将购物车中的商品结算并生成订单。订单创建后会扣除用户账户余额，并更新库存。

@PostMapping("/customer/{customerId}/checkout")public ResponseEntity<?> checkoutCart(@PathVariable Long customerId) {

try {

Order order = orderService.createOrderFromCart(customerId);

return ResponseEntity.ok(order);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(e.getMessage());

}

}

**获取特定订单**：GET /api/orders/{orderId}，该接口根据订单ID返回订单的详细信息。

@GetMapping("/{orderId}")public ResponseEntity<Order> getOrderById(@PathVariable Long orderId) {

try {

Order order = orderService.getOrderById(orderId);

return ResponseEntity.ok(order);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null);

}

}

**获取某个顾客的所有订单**：GET /api/orders/customer/{customerId}，该接口返回某个顾客的所有订单。

@GetMapping("/customer/{customerId}")public ResponseEntity<List<Order>> getOrdersByCustomer(@PathVariable Long customerId) {

try {

List<Order> orders = orderService.getAllOrdersByCustomer(customerId);

return ResponseEntity.ok(orders);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null);

}

}

**更新订单**：PUT /api/orders/{orderId}，该接口用于更新订单状态等信息。

@PutMapping("/{orderId}")public ResponseEntity<Order> updateOrder(@PathVariable Long orderId, @RequestBody Order updatedOrder) {

try {

Order order = orderService.updateOrder(orderId, updatedOrder);

return ResponseEntity.ok(order);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null);

}

}

**删除订单**：DELETE /api/orders/{orderId}，该接口用于删除指定ID的订单。

@DeleteMapping("/{orderId}")public ResponseEntity<Void> deleteOrder(@PathVariable Long orderId) {

try {

orderService.deleteOrder(orderId);

return ResponseEntity.noContent().build();

} catch (RuntimeException e) {

return ResponseEntity.badRequest().build();

}

}

**取消订单**：DELETE /api/orders/{orderId}/cancel，该接口允许用户取消未发货的订单，并将退款退还到用户账户中。

@DeleteMapping("/{orderId}/cancel")public ResponseEntity<?> cancelOrder(@PathVariable Long orderId) {

try {

orderService.cancelOrder(orderId);

return ResponseEntity.ok("Order canceled successfully.");

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(e.getMessage());

}

}

**管理员查看所有订单**：GET /api/orders/admin/all，该接口供管理员查看所有订单。

@GetMapping("/admin/all")public ResponseEntity<List<Order>> getAllOrders() {

try {

List<Order> orders = orderService.getAllOrders();

return ResponseEntity.ok(orders);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(null);

}

}

**管理员安排单个订单发货**：PUT /api/orders/{orderId}/ship，该接口供管理员标记订单为已发货状态。

@PutMapping("/{orderId}/ship")public ResponseEntity<?> shipOrder(@PathVariable Long orderId) {

try {

Order shippedOrder = orderService.shipOrder(orderId);

return ResponseEntity.ok(shippedOrder);

} catch (RuntimeException e) {

return ResponseEntity.badRequest().body(e.getMessage());

}

}

OrderController提供了丰富的API接口来管理订单，包含了订单的所有主要操作，并且进行了权限控制，确保只有管理员可以进行一些敏感操作（如查看所有订单、发货操作）。

### 7.4.2. 模型层（Order 和 OrderItem）

Order类是订单实体，OrderItem是订单中的每一项商品。两者之间是一对多的关系，每个订单可以包含多个订单项。Order类还包含了订单的基本信息，如订单ID、顾客信息、订单项、总金额和订单状态等。

* Order 类：
  + customer：与顾客的多对一关系，表示订单属于哪个顾客。
  + items：与OrderItem的一对多关系，表示订单中的每一项商品。
  + totalAmount：订单总金额。
  + status：订单状态（如“PENDING”、“SHIPPED”等）。

@Entity@Table(name = "`order`")public class Order {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "customer\_id", nullable = false)

private Customer customer;

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)

@JsonManagedReference

private List<OrderItem> items;

private double totalAmount;

private String status = "PENDING"; // 默认状态为“PENDING”

}

* OrderItem 类：
  + bookId：订单项关联的书籍ID。
  + quantity：该书籍的数量。
  + price：该书籍的单价。
  + order：与Order的多对一关系，表示订单项属于哪个订单。

@Entitypublic class OrderItem {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "order\_id", nullable = false)

@JsonIgnore

private Order order;

private Long bookId;

private Integer quantity;

private double price;

}

### 7.4.3. ****数据访问层（OrderRepository 和 OrderItemRepository）****

OrderRepository和OrderItemRepository是Spring Data JPA提供的接口，用于操作Order和OrderItem实体。它们通过继承JpaRepository接口提供了基本的CRUD操作，并且可以通过方法名自动生成SQL查询。

OrderRepository： 提供了根据顾客ID查询所有订单的功能。

public interface OrderRepository extends JpaRepository<Order, Long> {

List<Order> findByCustomerId(Long customerId);

}

OrderItemRepository： 提供了基本的CRUD操作来管理订单项。

public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {}

### 7.4.4. ****服务层（OrderService）****

OrderService负责处理与订单相关的业务逻辑，主要包括创建订单、查询订单、更新订单状态、删除订单、取消订单、发货操作等。

**创建订单**：createOrderFromCart()方法从购物车生成订单，验证库存并扣除用户余额，创建订单并保存。

public Order createOrderFromCart(Long customerId) {

// 获取客户、购物车和订单项

// 验证库存并扣除余额

// 创建订单并保存

}

**获取订单**：getOrderById()和getAllOrdersByCustomer()方法提供了根据ID查询订单和查询某个顾客的所有订单的功能。

**更新订单**：updateOrder()方法允许更新订单的状态。

**删除订单**：deleteOrder()方法删除指定订单。

**取消订单**：cancelOrder()方法允许用户取消订单，并退款给用户账户，同时返还库存。

**管理员操作**：管理员可以查看所有订单，并安排订单发货（shipOrder()方法）。

## 7.5. 库存管理与采购订单模块

库存管理和采购订单模块设计了一个完整的缺书记录和采购单的生成与管理流程。系统通过以下几个方面实现了库存不足时的处理和库存补充的功能：

* 库存检查与缺书记录：定期检查库存，生成缺书记录。
* 采购单生成与管理：根据缺书记录生成采购单，并在采购单到货时更新库存。
* 采购单的生命周期管理：支持采购单的创建、更新、删除、到货处理和状态更新。
* 与库存的互动：库存更新、库存不足处理和书籍的入库处理确保了系统的库存一致性。

### 7.5.1. 控制器层

OutOfStockController是处理库存不足和缺书记录相关操作的控制器类，主要提供以下功能：

**检查库存并生成缺书记录**：POST /api/out-of-stock/check，该接口触发库存检查，检查库存是否低于最低库存量，如果是，则生成缺书记录。

@PostMapping("/check")public void checkStock() {

bookService.checkStockAndGenerateOutOfStockRecords();

}

**根据缺书记录生成采购单**：POST/api/out-of-stock/generate-purchase-order，该接口根据缺书记录生成采购单，确保及时补充库存。

@PostMapping("/generate-purchase-order")public PurchaseOrder generatePurchaseOrder() {

return bookService.generatePurchaseOrderFromOutOfStockRecords();

}

PurchaseOrderController负责管理采购单的相关操作，提供了对采购单的增删改查操作。具体功能包括：

**创建采购单**：POST /purchase-orders，用于创建新的采购单。采购单的生成是基于缺书记录生成的。

@PostMappingpublic PurchaseOrder createPurchaseOrder(@RequestBody PurchaseOrder purchaseOrder) {

return purchaseOrderService.createPurchaseOrder(purchaseOrder);

}

**更新采购单**：PUT /purchase-orders/{id}，允许更新采购单信息。

@PutMapping("/{id}")public PurchaseOrder updatePurchaseOrder(@PathVariable Long id, @RequestBody PurchaseOrder purchaseOrder) {

return purchaseOrderService.updatePurchaseOrder(id, purchaseOrder);

}

**删除采购单**：DELETE /purchase-orders/{id}，删除指定ID的采购单。

@DeleteMapping("/{id}")public void deletePurchaseOrder(@PathVariable Long id) {

purchaseOrderService.deletePurchaseOrder(id);

}

**获取所有采购单**：GET /purchase-orders，查询所有采购单。

@GetMappingpublic List<PurchaseOrder> getAllPurchaseOrders() {

return purchaseOrderService.getAllPurchaseOrders();

}

**更新采购单到货**：PUT /purchase-orders/{id}/receive，当采购单到货时，更新库存并删除采购单。

@PutMapping("/{id}/receive")public void receivePurchaseOrder(@PathVariable Long id) {

purchaseOrderService.markPurchaseOrderAsReceived(id);

}

### 7.5.2 模型层

OutOfStockRecord 类

OutOfStockRecord是库存不足时记录缺货信息的实体类，包含书籍和缺货数量。它与Book类之间通过ManyToOne关系连接，每条缺书记录对应一本书。

@Entity@Table(name = "out\_of\_stock\_record")public class OutOfStockRecord {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

private Book book; // 关联到具体的书籍

private Integer shortageQuantity; // 缺少的数量

// Getters and Setters

}

PurchaseOrder 类

PurchaseOrder类是采购单实体，包含了供应商信息、采购单总价、以及多个书籍信息。每个采购单可以包含多个书籍，体现了采购单与Book的多对多关系。

@Entitypublic class PurchaseOrder {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String supplierName;

private Double totalPrice;

@OneToMany

private List<Book> books = new ArrayList<>(); // 采购单中的书籍列表

// Getters and Setters

}

### **7.5.3. 数据访问层**

**PurchaseOrderRepository**：负责对PurchaseOrder实体的增删查改操作。

public interface PurchaseOrderRepository extends JpaRepository<PurchaseOrder, Long> {}

**OutOfStockRecordRepository**：负责对OutOfStockRecord实体的增删查改操作。

public interface OutOfStockRecordRepository extends JpaRepository<OutOfStockRecord, Long> {}

### 7.5.4. ****服务层****

PurchaseOrderService类处理与采购单相关的业务逻辑，主要包括采购单的创建、更新、删除和到货操作。BookService则提供了库存检查、缺书记录生成和采购单生成等功能。

**创建采购单**：createPurchaseOrder()方法根据缺书记录生成新的采购单，并保存到数据库。

public PurchaseOrder createPurchaseOrder(PurchaseOrder purchaseOrder) {

return purchaseOrderRepository.save(purchaseOrder);

}

**更新采购单到货**：markPurchaseOrderAsReceived()方法在采购单到货时，更新库存并删除该采购单。

public void markPurchaseOrderAsReceived(Long orderId) {

PurchaseOrder purchaseOrder = purchaseOrderRepository.findById(orderId)

.orElseThrow(() -> new RuntimeException("Purchase Order not found"));

for (Book book : purchaseOrder.getBooks()) {

book.setStockQuantity(book.getMinStockQuantity());

bookRepository.save(book);

}

purchaseOrderRepository.delete(purchaseOrder); // 删除采购单

}

**库存检查与缺书记录**：checkStockAndGenerateOutOfStockRecords()方法会遍历所有书籍，检查库存并生成缺书记录。

public void checkStockAndGenerateOutOfStockRecords() {

List<Book> books = bookRepository.findAll();

for (Book book : books) {

if (book.getStockQuantity() < book.getMinStockQuantity()) {

OutOfStockRecord record = new OutOfStockRecord();

record.setBook(book);

record.setShortageQuantity(book.getMinStockQuantity() - book.getStockQuantity());

outOfStockRecordRepository.save(record);

}

}

}

**根据缺书记录生成采购单**：generatePurchaseOrderFromOutOfStockRecords()方法根据缺书记录生成采购单，并返回生成的采购单。

public PurchaseOrder generatePurchaseOrderFromOutOfStockRecords() {

List<OutOfStockRecord> records = outOfStockRecordRepository.findAll();

PurchaseOrder purchaseOrder = new PurchaseOrder();

for (OutOfStockRecord record : records) {

purchaseOrder.getBooks().add(record.getBook());

}

// 计算采购单总价

double totalPrice = purchaseOrder.getBooks().stream()

.mapToDouble(book -> book.getPrice())

.sum();

purchaseOrder.setTotalPrice(totalPrice);

return purchaseOrderRepository.save(purchaseOrder);

}

## 7.6. 供应商管理模块

供应商管理模块实现了供应商信息的增删改查、书籍与供应商之间的关联管理等功能，涉及供应商与书籍的多对多关系。系统设计简洁、功能完备，能够高效地管理供应商和书籍的关系。

* 控制器层：提供了所有与供应商相关的API接口，包括供应商的CRUD操作、书籍的添加与删除、获取供应商提供的书籍列表等。
* 模型层：Supplier类与Book类之间建立了多对多的关系，使用@ManyToMany和@JoinTable注解来维护中间表供应商\_书籍。
* 数据访问层：SupplierRepository通过继承JpaRepository实现了对供应商的基本CRUD操作。
* 服务层：SupplierService处理了业务逻辑，包括供应商的创建、更新、删除，书籍的添加与删除，供应商提供的书籍查询等。

### 7.6.1. ****控制器层（SupplierController）****

SupplierController负责暴露供应商相关的API接口，提供了供应商的基本CRUD操作以及与书籍的关联操作。具体功能如下：

**获取所有供应商**：GET /suppliers，该接口返回所有供应商的列表。

@GetMappingpublic List<Supplier> getAllSuppliers() {

return supplierService.getAllSuppliers();

}

**创建供应商**：POST /suppliers，该接口用于创建新的供应商。

@PostMappingpublic Supplier createSupplier(@RequestBody Supplier supplier) {

return supplierService.createSupplier(supplier);

}

**更新供应商信息**：PUT /suppliers/{id}，该接口用于更新指定供应商的信息。

@PutMapping("/{id}")public Supplier updateSupplier(@PathVariable Long id, @RequestBody Supplier supplier) {

return supplierService.updateSupplier(id, supplier);

}

**删除供应商**：DELETE /suppliers/{id}，该接口用于删除指定ID的供应商。

@DeleteMapping("/{id}")public void deleteSupplier(@PathVariable Long id) {

supplierService.deleteSupplier(id);

}

**添加书籍到供应商目录**：POST /suppliers/{id}/catalog，该接口将书籍添加到指定供应商的书籍目录中。

@PostMapping("/{id}/catalog")public Supplier addBookToCatalog(@PathVariable Long id, @RequestParam Long bookId) {

return supplierService.addBookToCatalog(id, bookId);

}

**从供应商目录中删除书籍**：DELETE /suppliers/{id}/catalog，该接口从指定供应商的目录中删除书籍。

@DeleteMapping("/{id}/catalog")public Supplier removeBookFromCatalog(@PathVariable Long id, @RequestParam Long bookId) {

return supplierService.removeBookFromCatalog(id, bookId);

}

**获取供应商提供的书籍ID列表**：GET /suppliers/{id}/books，该接口返回指定供应商提供的所有书籍的ID列表。

@GetMapping("/{id}/books")public List<Long> getSupplierBookIds(@PathVariable Long id) {

return supplierService.getSupplierBookIds(id);

}

SupplierController暴露了所有必要的API接口，支持供应商的基本操作，并且提供了与书籍管理的互动功能。

### 7.6.2. ****模型层（Supplier）****

Supplier类是供应商实体，表示一个供应商的基本信息（如名称、联系方式等），以及该供应商提供的书籍。与Book之间的关系是多对多的，每个供应商可以提供多本书籍，每本书籍也可以由多个供应商提供。

* Supplier类：
  + books：通过@ManyToMany关系注解定义与Book类的关联，表示供应商提供的书籍列表。
  + @JoinTable：定义了中间表supplier\_book，用于存储供应商与书籍之间的多对多关系。

@Entity@JsonIgnoreProperties({"books"})public class Supplier {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String contactInfo;

@ManyToMany

@JoinTable(

name = "supplier\_book",

joinColumns = @JoinColumn(name = "supplier\_id"),

inverseJoinColumns = @JoinColumn(name = "book\_id")

)

private List<Book> books; // 供应商提供的书籍列表

// Getters and Setters

}

* Book类：
  + suppliers：与Supplier之间的多对多关系，每本书籍可以由多个供应商提供。

### 7.6.3. ****数据访问层（SupplierRepository）****

SupplierRepository是一个Spring Data JPA接口，继承自JpaRepository，提供了对Supplier实体的基本CRUD操作。它还提供了通过ID查询供应商以及相关书籍的功能。

public interface SupplierRepository extends JpaRepository<Supplier, Long> {}

### 7.6.4. ****服务层（SupplierService）****

SupplierService是供应商业务逻辑层，提供了与供应商和书籍相关的操作，主要功能如下：

**获取所有供应商**：getAllSuppliers()方法返回所有供应商的列表。

public List<Supplier> getAllSuppliers() {

return supplierRepository.findAll();

}

**创建供应商**：createSupplier()方法保存新的供应商。

public Supplier createSupplier(Supplier supplier) {

return supplierRepository.save(supplier);

}

**更新供应商信息**：updateSupplier()方法更新指定ID的供应商信息。

public Supplier updateSupplier(Long id, Supplier supplier) {

if (!supplierRepository.existsById(id)) {

throw new RuntimeException("Supplier not found");

}

supplier.setId(id);

return supplierRepository.save(supplier);

}

**删除供应商**：deleteSupplier()方法删除指定ID的供应商。

public void deleteSupplier(Long id) {

supplierRepository.deleteById(id);

}

**添加书籍到供应商目录**：addBookToCatalog()方法将指定书籍添加到供应商的书籍列表中，并保持供应商和书籍之间的关联。

public Supplier addBookToCatalog(Long supplierId, Long bookId) {

Supplier supplier = getSupplierById(supplierId);

Book book = getBookById(bookId);

supplier.getBooks().add(book);

book.getSuppliers().add(supplier);

supplierRepository.save(supplier);

bookRepository.save(book);

return supplier;

}

**从供应商目录中删除书籍**：removeBookFromCatalog()方法从供应商的书籍列表中移除指定书籍，并更新供应商和书籍之间的关系。

public Supplier removeBookFromCatalog(Long supplierId, Long bookId) {

Supplier supplier = getSupplierById(supplierId);

Book book = getBookById(bookId);

supplier.getBooks().remove(book);

book.getSuppliers().remove(supplier);

supplierRepository.save(supplier);

bookRepository.save(book);

return supplier;

}

**获取供应商提供的书籍ID列表**：getSupplierBookIds()方法返回供应商提供的书籍ID列表。

public List<Long> getSupplierBookIds(Long supplierId) {

Supplier supplier = getSupplierById(supplierId);

return supplier.getBooks().stream().map(Book::getId).toList();

}

获取单个供应商：getSupplierById()方法通过ID查找供应商。

public Supplier getSupplierById(Long id) {

return supplierRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Supplier not found"));

}

**获取单本书籍**：getBookById()方法通过ID查找书籍。

public Book getBookById(Long id) {

return bookRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Book not found"));

}

# 总结与体会

通过本次网上书店系统的设计与实现，我们深入理解并实践了数据库系统原理、软件工程、以及现代Web应用开发的相关知识。以下是我们在项目开发过程中的主要收获和体会：

## 8.1 项目成果总结

系统功能完整性：我们成功实现了网上书店的核心功能，包括供书目录与库存管理、采购管理、客户管理、订单处理、供应商管理以及网上浏览与查询等。

技术栈应用：通过Vue 3和Spring Boot的结合，我们实现了前后端分离的架构，提高了系统的可维护性和可扩展性。

数据库设计：我们设计了一个结构化和规范化的数据库，通过ER图、表单设计、约束、触发器等确保了数据的完整性和一致性。

安全性与性能：在系统设计中，我们特别关注了安全性和性能优化，包括数据加密、身份验证、权限控制以及数据库索引和查询优化。

用户体验：我们致力于提供用户友好的界面和流畅的交互体验，通过响应式设计和动态页面展示，满足了不同用户的需求。

## 8.2 个人体会

团队协作：在项目开发过程中，团队成员之间的沟通和协作至关重要。我们通过定期的会议和协作工具保持同步，确保了项目的顺利进行。

问题解决：面对开发中的挑战，如技术难题、性能瓶颈等，我们通过研究、讨论和实践，提升了问题解决的能力.

技术学习：项目中涉及的新技术和工具，如Vue 3的组合式API、Spring Boot的自动配置等，都是我们学习和掌握的新技能。

项目管理：通过这个项目，我们更加熟悉了项目管理的流程，包括需求分析、设计、开发、测试等。

创新思维：在满足基本需求的基础上，我们尝试了创新的功能设计，如自动化的库存管理和智能化的用户推荐系统，提升了系统的附加价值。

通过这个项目，我们不仅提升了自己的技术能力，也对网上书店的开发和数据库应用的设计和实现有了更深入的理解。我们相信，这次的经验将为我们未来的学习和工作打下坚实的基础。