JLU Emshop 电商系统 - 综合项目报告

吉林大学 2024级卓越工程师班 面向对象程序设计课程

一、需求分析

（1）描述系统功能

1.1 系统概述

JLU Emshop是一个基于C/S架构的完整电商交易系统,采用面向对象设计思想,

实现了商品浏览、购物车管理、订单处理、支付结算、优惠券系统等完整的电商业务流程。系统支持多用户并发访问,具备完善的权限管理和数据持久

化能力。

1.2 核心功能模块

【用户管理系统】

- 用户注册登录:安全的用户认证机制,支持密码加密存储

- 会话管理:基于Token的会话控制,支持多终端登录管理

- 个人信息管理:用户资料修改、密码管理、账户安全设置

- 收货地址管理:多地址管理、默认地址设置、地址验证

- 角色权限管理:普通用户/管理员角色区分,权限分级控制

【商品管理系统】

- 商品浏览:分页展示、分类筛选、多维度排序

- 商品搜索:关键词搜索、价格区间筛选、模糊匹配

- 商品详情:完整的商品信息展示(名称、价格、库存)

- 库存管理:实时库存监控、库存预警、库存同步机制

- 商品分类:多级分类体系、分类导航、分类统计

【购物车系统】

- 购物车操作:添加、删除、修改商品数量

- 实时计算:自动计算总价、优惠金额、运费

- 批量管理:支持批量删除、全选操作、商品选择

- 持久化存储:购物车数据持久化保存,跨设备同步

- 库存验证:购物车结算前库存检查,防止超卖

【订单处理系统】

- 订单创建:从购物车快速生成订单,支持立即购买

- 订单管理:查看订单列表、订单详情、订单历史

- 状态流转:完整的订单生命周期管理

\* pending(待确认) → confirmed(已确认) → paid(已支付)

\* 支持cancelled(已取消)和refunded(已退款)状态

- 订单搜索:按状态、时间、金额筛选订单

【支付系统】

- 多支付方式:支付宝、微信支付、银行卡、信用卡

- 退款处理:订单退款、退款审核、退款金额计算

【优惠券系统】

- 优惠券类型:固定金额券、折扣券、满减券

- 优惠券管理:创建、发放、使用、过期管理

- 用户优惠券:我的优惠券、可用优惠券筛选、优惠券兑换

- 自动计算:结算时自动应用最优惠券

- 优惠活动:限时优惠、新人专享

【管理员功能】

- 用户管理:查看用户列表、用户状态管理、角色分配

- 商品管理:商品上下架、库存调整、价格修改

- 订单管理:订单审核、退款审批

（2）分析重点难点

2.1 技术架构难点

【C/S架构实现】

- 挑战:实现稳定的客户端-服务器通信

- 难点:网络异常处理、断线重连、消息队列管理

- 解决:采用Netty框架实现高性能TCP通信,自定义消息协议

【跨语言集成】

- 挑战:Java服务端与C++核心业务逻辑集成

- 难点:JNI接口设计、内存管理、类型转换

- 解决:采用标准JNI规范,实现8194行C++核心代码的JNI封装

【多客户端支持】

- 挑战:同时支持Qt桌面客户端和未来的Web/移动端

- 难点:接口标准化、数据格式统一

- 解决:定义统一的JSON数据交换格式,RESTful API设计

2.2 业务逻辑难点

【并发控制】

- 挑战:多用户同时操作同一商品库存

- 难点:防止超卖、库存一致性保证

- 解决:数据库行级锁、乐观锁版本号控制、库存预扣机制

【事务管理】

- 挑战:订单创建涉及多表操作(订单表、订单详情、库存、优惠券)

- 难点:保证数据一致性、回滚机制

- 解决:使用MySQL InnoDB事务、ACID特性保证

【优惠券叠加】

- 挑战:多种优惠策略叠加计算

- 难点:折扣优先级、互斥规则、最优方案选择

- 解决:实现策略模式,设计优惠券计算引擎

【订单状态流转】

- 挑战:订单状态变更的合法性校验

- 难点:状态机设计、非法状态转换拦截

- 解决:定义完整的状态转换规则,状态机模式实现

2.3 性能优化难点

【数据库查询优化】

- 挑战:商品列表分页查询、订单历史查询性能

- 难点:大数据量下的查询效率

- 解决:合理设计索引、分页参数限制、查询条件优化

【连接池管理】

- 挑战:数据库连接资源管理

- 难点:连接泄漏检测、连接复用

- 解决:实现智能连接池(初始5个,最大20个连接)

【内存管理】

- 挑战:C++ JNI代码的内存安全

- 难点:避免内存泄漏、野指针

- 解决:采用RAII原则、智能指针、自动资源管理

（3）技术难点说明

3.1 JNI技术难点

【接口设计】

难点描述:

Java与C++之间的接口设计需要考虑类型映射、内存管理、异常传递。

Java的String、Object等类型需要转换为C++的std::string、json对象。

解决方案:

- 统一使用JSON字符串作为数据交换格式

- C++端使用nlohmann/json库进行JSON处理

- 定义标准的响应格式:{success, message, data, error\_code}

- 使用JNI工具函数封装类型转换逻辑

实现代码示例:

```cpp

// JNI接口定义

JNIEXPORT jstring JNICALL Java\_emshop\_EmshopNativeInterface\_login

(JNIEnv \*env, jclass, jstring jUsername, jstring jPassword) {

// 类型转换

const char\* username = env->GetStringUTFChars(jUsername, nullptr);

const char\* password = env->GetStringUTFChars(jPassword, nullptr);

// 调用C++业务逻辑

std::string result = UserService::login(username, password);

// 释放资源

env->ReleaseStringUTFChars(jUsername, username);

env->ReleaseStringUTFChars(jPassword, password);

// 返回JSON字符串

return env->NewStringUTF(result.c\_str());

}

```

【内存管理】

难点描述:

JNI调用中涉及Java堆和C++堆的交互,需要正确管理内存生命周期。

GetStringUTFChars获取的字符串必须用ReleaseStringUTFChars释放。

解决方案:

- 使用RAII原则,确保资源自动释放

- 定义智能指针包装类管理JNI资源

- 统一的异常处理机制,确保异常时资源释放

3.2 网络通信难点

【消息协议设计】

难点描述:

需要设计一个高效、可扩展的应用层协议,支持命令路由、参数传递、

响应格式统一。

解决方案:

- 采用文本行协议:每条消息以换行符分隔

- 命令格式:METHOD PARAM1 PARAM2 ...

- 支持引号包围参数,处理空格和特殊字符

- JSON响应格式统一

协议示例:

```

客户端请求:

LOGIN admin admin123

GET\_PRODUCTS electronics 1 10

CREATE\_ORDER 1 "北京市朝阳区xxx" COUPON100

服务端响应:

{"success":true,"message":"登录成功","data":{"user\_id":1,"username":"admin"}}

{"success":true,"message":"获取商品成功","data":{"products":[...],"total":50}}

```

【会话管理】

难点描述:

需要维护客户端连接状态,区分不同用户,实现会话隔离。

解决方案:

- 使用ConcurrentHashMap<ChannelId, UserSession>存储会话

- 用户登录成功后创建会话对象

- 连接断开时自动清理会话

- 支持基于会话的权限验证

3.3 数据库设计难点

【数据库范式】

难点描述:

需要平衡范式设计和查询性能,避免过度规范化导致查询复杂。

解决方案:

- 核心表采用第三范式设计

- 适当数据冗余提升查询效率(如订单表保存商品名称快照)

- 使用视图简化复杂查询

- 关键字段建立索引

【并发控制】

难点描述:

库存扣减、订单创建等操作需要保证数据一致性。

解决方案:

```sql

-- 库存扣减使用悲观锁

SELECT stock\_quantity FROM products WHERE product\_id = ? FOR UPDATE;

UPDATE products SET stock\_quantity = stock\_quantity - ?

WHERE product\_id = ? AND stock\_quantity >= ?;

-- 订单创建使用事务

START TRANSACTION;

INSERT INTO orders (...) VALUES (...);

INSERT INTO order\_items (...) VALUES (...);

UPDATE products SET stock\_quantity = stock\_quantity - ? WHERE ...;

COMMIT;

```

【字符集处理】

难点描述:

需要支持中文商品名、评价等,避免乱码问题。

解决方案:

- 数据库字符集统一使用utf8mb4

- 排序规则使用utf8mb4\_unicode\_ci

- 连接时指定字符集:SET NAMES utf8mb4

- C++代码统一使用UTF-8编码

3.4 UI设计难点

【跨平台适配】

难点描述:

Qt应用需要在不同分辨率、不同操作系统上良好显示。

解决方案:

- 使用Qt的布局管理器(QVBoxLayout、QHBoxLayout、QGridLayout)

- 定义相对尺寸,避免硬编码像素值

- 使用QSS样式表实现主题切换

- 测试不同DPI设置下的显示效果

【异步操作】

难点描述:

网络请求是异步的,需要避免UI阻塞,提供良好的用户反馈。

解决方案:

- 使用Qt的信号槽机制实现异步通信

- 网络操作在独立线程执行

- 显示加载动画提示用户等待

- 实现超时处理和错误提示

二、系统设计

（1）功能模块划分

1.1 整体架构分层

【表示层 (Presentation Layer)】

技术选型:Qt 6.9.1 C++

职责:

- 用户界面展示和交互

- 用户输入验证和反馈

- 数据格式化展示

- 主题样式管理

主要模块:

- LoginDialog:登录注册界面

- MainWindow:主窗口框架

- ProductsTab:商品浏览页面

- CartTab:购物车页面

- OrdersTab:订单管理页面

- CouponsTab:优惠券页面

- ProfileTab:个人中心

- AdminTab:管理员控制台

【通信层 (Communication Layer)】

技术选型:Netty 4.x Java

职责:

- TCP网络通信

- 会话管理和维护

- 消息路由和分发

- 协议编解码

主要组件:

- EmshopNettyServer:Netty服务器主类

- EmshopServerHandler:业务处理器

- SessionManager:会话管理器

- CommandRouter:命令路由器

【业务层 (Business Layer)】

技术选型:C++ 17 (JNI)

职责:

- 核心业务逻辑处理

- 数据验证和转换

- 事务控制和协调

- 业务规则实现

服务模块(共8个Service):

- UserService:用户认证和管理(552行代码)

- ProductService:商品管理(约1500行)

- CartService:购物车服务(约900行)

- OrderService:订单处理(约1800行)

- CouponService:优惠券系统(约600行)

- AddressService:地址管理(约300行)

- ReviewService:评价系统(约400行)

- PaymentService:支付处理(约500行)

【持久层 (Persistence Layer)】

技术选型:MySQL 8.0

职责:

- 数据持久化存储

- 数据查询和更新

- 事务管理

- 数据完整性约束

数据访问组件:

- DatabaseManager:数据库连接管理

- ConnectionPool:连接池(初始5个,最大20个)

- SQLExecutor:SQL执行器

- TransactionManager:事务管理器

1.2 核心功能模块详细划分

【模块1:用户认证模块】

功能清单:

- F1.1 用户注册:用户名、密码、邮箱、手机号

- F1.2 用户登录:用户名/邮箱登录、密码验证

- F1.3 会话管理:Token生成、会话保持、自动登出

- F1.4 密码管理:密码修改、找回密码

- F1.5 权限验证:角色检查、权限拦截

接口定义:

```cpp

class UserService {

public:

json register(const std::string& username,

const std::string& password,

const std::string& email);

json login(const std::string& username,

const std::string& password);

json logout(long userId);

json getUserInfo(long userId);

json updateUserInfo(long userId, const json& userInfo);

};

```

【模块2:商品管理模块】

功能清单:

- F2.1 商品列表:分页查询、分类筛选、排序

- F2.2 商品搜索:关键词搜索、价格区间、综合排序

- F2.3 商品详情:商品信息、库存状态、用户评价

- F2.4 商品维护:添加商品、修改商品、上下架(管理员)

- F2.5 库存管理:库存查询、库存调整、低库存预警

数据结构:

```cpp

struct Product {

long product\_id;

std::string name;

double price;

int stock\_quantity;

std::string category;

std::string description;

std::string image\_url;

double rating;

int review\_count;

std::string status; // active/inactive

};

```

【模块3:购物车模块】

功能清单:

- F3.1 添加商品:添加到购物车、数量设置

- F3.2 查看购物车:商品列表、小计、总计

- F3.3 修改数量:增加、减少、直接输入

- F3.4 删除商品:单个删除、批量删除

- F3.5 选择商品:全选、反选、部分选择

- F3.6 清空购物车:清除所有商品

业务规则:

- 同一商品在购物车中只保留一条记录,累加数量

- 添加时检查库存,数量不能超过库存

- 价格使用添加时的价格,不随商品价格变动而变动

- 购物车数据持久化,登录后自动加载

【模块4:订单管理模块】

功能清单:

- F4.1 创建订单:从购物车创建、立即购买

- F4.2 订单列表:全部订单、状态筛选、时间排序

- F4.3 订单详情:订单信息、商品明细、物流信息

- F4.4 订单操作:取消订单、确认收货、删除订单

- F4.5 订单跟踪:物流查询、配送状态

- F4.6 订单统计:销售报表、订单趋势(管理员)

订单状态机:

```

[pending待确认]

↓ 确认订单

[confirmed已确认]

↓ 支付成功

[paid已支付]

↓ 商家发货

[completed已完成]

任意状态 → [cancelled已取消]

paid状态 → [refunding退款中] → [refunded已退款]

```

【模块5:支付模块】

功能清单:

- F5.1 支付方式:支付宝、微信、银行卡、信用卡

- F5.2 支付处理:金额验证、支付调用、结果回调

- F5.3 支付查询:支付状态、交易流水

- F5.4 退款处理:退款申请、退款审核、退款到账

支付流程:

```

1. 用户选择支付方式

2. 验证支付金额与订单金额一致

3. 调用第三方支付接口(模拟)

4. 接收支付结果回调

5. 更新订单状态为已支付

6. 扣减商品库存

7. 清空购物车已结算商品

8. 发送支付成功通知

```

【模块6:优惠券模块】

功能清单:

- F6.1 优惠券模板:创建模板、设置规则

- F6.2 优惠券发放:批量发放、定向发放

- F6.3 用户优惠券:我的优惠券、可用筛选

- F6.4 优惠券使用:结算时选择、自动应用

- F6.5 优惠券管理:过期处理、使用统计

优惠券类型:

```cpp

enum class CouponType {

FIXED\_AMOUNT, // 固定金额券:满X减Y

PERCENTAGE, // 折扣券:X折

FREE\_SHIPPING, // 包邮券

GIFT // 赠品券

};

struct Coupon {

long coupon\_id;

std::string code;

CouponType type;

double value; // 金额或折扣比例

double min\_amount; // 最低消费金额

std::string start\_date;

std::string end\_date;

int total\_quantity;

int used\_quantity;

std::string status;

};

```

【模块7:管理员模块】

功能清单:

- F8.1 用户管理:用户列表、状态设置、角色分配

- F8.2 商品管理:商品审核、库存管理、价格调整

- F8.3 订单管理:订单审核、发货操作、退款审批

- F8.4 优惠券管理:创建活动、发放优惠券

- F8.5 数据统计:销售报表、用户分析、商品排行

权限控制:

```cpp

bool checkAdminPermission(long userId) {

// 查询用户角色

json result = executeQuery(

"SELECT role FROM users WHERE user\_id = ?",

{userId}

);

if (result["success"].get<bool>() &&

!result["data"].empty()) {

std::string role = result["data"][0]["role"];

return (role == "admin");

}

return false;

}

```

（2）类图及说明

2.1 核心类设计

【基础服务类 BaseService】

```

┌────────────────────────────────────┐

│ BaseService │

├────────────────────────────────────┤

│ # dbManager: DatabaseManager& │

│ # logger: Logger │

├────────────────────────────────────┤

│ + BaseService(DatabaseManager&) │

│ + virtual ~BaseService() │

│ + virtual getServiceName(): string│

│ # executeQuery(...): json │

│ # executeUpdate(...): json │

│ # logInfo(message: string) │

│ # logError(message: string) │

│ # createResponse(...): json │

└────────────────────────────────────┘

△

│ 继承

┌─────┴─────┬─────────┬─────────┐

│ │ │ │

┌────▽────┐ ┌───▽────┐ ┌──▽─────┐ ┌▽─────────┐

│ User │ │Product │ │Cart │ │Order │

│ Service │ │Service │ │Service │ │Service │

└─────────┘ └────────┘ └────────┘ └──────────┘

```

设计说明:

- BaseService提供公共功能:数据库访问、日志记录、响应格式化

- 所有业务服务继承BaseService,复用基础能力

- 使用模板方法模式,定义业务处理流程框架

【用户服务类 UserService】

```

┌──────────────────────────────────────────┐

│ UserService │

├──────────────────────────────────────────┤

│ - sessionManager: SessionManager& │

├──────────────────────────────────────────┤

│ + register(username, password, email) │

│ + login(username, password): json │

│ + logout(userId): json │

│ + getUserInfo(userId): json │

│ + updateUserInfo(userId, info): json │

│ + changePassword(userId, old, new): json │

│ + getUserAddresses(userId): json │

│ + addUserAddress(userId, ...): json │

│ - validateUsername(username): bool │

│ - hashPassword(password): string │

│ - generateToken(userId): string │

│ - verifyPassword(input, stored): bool │

└──────────────────────────────────────────┘

```

关键方法说明:

- register:用户注册,包括用户名唯一性检查、密码加密

- login:用户登录,验证凭据、生成会话Token

- hashPassword:使用SHA256对密码进行哈希

- generateToken:生成会话Token,格式"sess\_{userId}\_{timestamp}"

【商品服务类 ProductService】

```

┌──────────────────────────────────────────┐

│ ProductService │

├──────────────────────────────────────────┤

│ - cache: ProductCache │

├──────────────────────────────────────────┤

│ + getProductList(category, page, size) │

│ + searchProducts(keyword, ...): json │

│ + getProductDetail(productId): json │

│ + addProduct(product): json │

│ + updateProduct(productId, data): json │

│ + deleteProduct(productId): json │

│ + checkStock(productId): json │

│ + updateStock(productId, qty, op): json │

│ + getLowStockProducts(threshold): json │

│ - buildSearchQuery(conditions): string │

│ - cacheProduct(productId, data) │

│ - invalidateCache(productId) │

└──────────────────────────────────────────┘

```

设计要点:

- 使用缓存减少数据库查询

- searchProducts支持多条件组合查询

- updateStock使用乐观锁防止超卖

【购物车服务类 CartService】

```

┌──────────────────────────────────────────┐

│ CartService │

├──────────────────────────────────────────┤

│ + addToCart(userId, productId, qty) │

│ + getCart(userId): json │

│ + updateCartItemQuantity(...) │

│ + removeFromCart(userId, productId) │

│ + clearCart(userId): json │

│ + getCartSummary(userId): json │

│ + updateCartSelected(userId, ...) │

│ - calculateTotalPrice(cartItems): double │

│ - validateStockAvailability(...): bool │

└──────────────────────────────────────────┘

```

核心逻辑:

- addToCart:如果商品已在购物车,累加数量;否则新增记录

- getCartSummary:计算购物车总价、商品总数

- validateStockAvailability:检查购物车所有商品库存

【订单服务类 OrderService】

```

┌──────────────────────────────────────────┐

│ OrderService │

├──────────────────────────────────────────┤

│ - stateMachine: OrderStateMachine │

├──────────────────────────────────────────┤

│ + createOrderFromCart(userId, ...) │

│ + createOrderDirect(userId, ...) │

│ + getOrderList(userId): json │

│ + getOrderDetail(orderId): json │

│ + updateOrderStatus(orderId, status) │

│ + cancelOrder(userId, orderId): json │

│ + payOrder(orderId): json │

│ + shipOrder(orderId, tracking): json │

│ + completeOrder(orderId): json │

│ + trackOrder(orderId): json │

│ - generateOrderNo(): string │

│ - lockStock(productId, quantity): bool │

│ - unlockStock(productId, quantity) │

│ - validateStateTransition(from, to) │

└──────────────────────────────────────────┘

```

状态转换验证:

```cpp

bool OrderService::validateStateTransition(

const std::string& fromState,

const std::string& toState) {

static const std::map<std::string,

std::set<std::string>> transitions = {

{"pending", {"confirmed", "cancelled"}},

{"confirmed", {"paid", "cancelled"}},

{"paid", {"shipped", "refunding"}},

{"shipped", {"delivered", "refunding"}},

{"delivered", {"completed", "refunding"}},

{"refunding", {"refunded", "paid"}}

};

auto it = transitions.find(fromState);

if (it != transitions.end()) {

return it->second.count(toState) > 0;

}

return false;

}

```

【数据库管理类 DatabaseManager】

```

┌──────────────────────────────────────────┐

│ DatabaseManager (Singleton) │

├──────────────────────────────────────────┤

│ - instance: DatabaseManager\* │

│ - connectionPool: ConnectionPool │

│ - config: DatabaseConfig │

│ - mutex: std::mutex │

├──────────────────────────────────────────┤

│ + getInstance(): DatabaseManager& │

│ + initialize(config): bool │

│ + getConnection(): Connection\* │

│ + releaseConnection(Connection\*) │

│ + executeQuery(sql, params): ResultSet │

│ + executeUpdate(sql, params): int │

│ + beginTransaction(): bool │

│ + commit(): bool │

│ + rollback(): bool │

│ - DatabaseManager() // 私有构造 │

│ - createConnectionPool() │

│ - validateConnection(Connection\*) │

└──────────────────────────────────────────┘

```

单例模式实现:

```cpp

DatabaseManager& DatabaseManager::getInstance() {

static DatabaseManager instance;

return instance;

}

```

连接池设计:

- 初始连接数:5个

- 最大连接数:20个

- 连接超时:30秒

- 空闲连接回收:5分钟无使用自动关闭

2.2 类之间的关系

【依赖关系】

```

Qt客户端 ──依赖→ Netty服务端 ──依赖→ JNI层 ──依赖→ C++业务层

↓

DatabaseManager

↓

MySQL数据库

```

【聚合关系】

```

MainWindow (主窗口)

├── ProductsTab (商品页)

├── CartTab (购物车页)

├── OrdersTab (订单页)

├── CouponsTab (优惠券页)

├── ProfileTab (个人中心)

└── AdminTab (管理员页)

```

【组合关系】

```

Order (订单)

├──■ OrderItem (订单项1)

├──■ OrderItem (订单项2)

└──■ OrderItem (订单项N)

■ 表示强关联,订单删除时级联删除订单项

```

（3）数据库定义

3.1 数据库设计原则

【设计原则】

1. 规范化设计:核心表采用第三范式(3NF)

2. 适度冗余:订单表保存商品快照,提高查询性能

3. 字符集统一:使用utf8mb4支持emoji和特殊字符

4. 索引优化:为高频查询字段建立索引

5. 外键约束:保证数据完整性和一致性

【命名规范】

- 表名:小写单词,复数形式,如users、products、orders

- 字段名:小写下划线分隔,如user\_id、product\_name

- 主键:表名\_id,如user\_id、product\_id

- 外键:引用表名\_id,如在orders表中引用user\_id

- 时间戳:统一使用created\_at、updated\_at

3.2 核心表结构

【表1:用户表 users】

```sql

CREATE TABLE users (

user\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '用户ID',

username VARCHAR(50) UNIQUE NOT NULL COMMENT '用户名',

password VARCHAR(255) NOT NULL COMMENT '密码(加密)',

email VARCHAR(100) UNIQUE COMMENT '邮箱',

phone VARCHAR(20) COMMENT '手机号',

role VARCHAR(20) DEFAULT 'user' COMMENT '角色:user/admin',

status VARCHAR(20) DEFAULT 'active' COMMENT '状态:active/banned/deleted',

points\_balance INT DEFAULT 0 COMMENT '积分余额',

avatar\_url VARCHAR(500) COMMENT '头像URL',

last\_login\_time DATETIME COMMENT '最后登录时间',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP COMMENT '注册时间',

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_username (username),

INDEX idx\_email (email),

INDEX idx\_status (status),

INDEX idx\_created (created\_at)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='用户表';

```

字段说明:

- password:存储SHA256加密后的密码

- role:普通用户为'user',管理员为'admin'

- status:active激活、banned禁用、deleted已删除

- points\_balance:用户积分,可用于兑换优惠券

【表2:商品分类表 categories】

```sql

CREATE TABLE categories (

category\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '分类ID',

name VARCHAR(100) NOT NULL COMMENT '分类名称',

parent\_id BIGINT DEFAULT 0 COMMENT '父分类ID,0表示顶级分类',

level INT DEFAULT 1 COMMENT '层级:1顶级,2二级,3三级',

sort\_order INT DEFAULT 0 COMMENT '排序序号',

icon\_url VARCHAR(500) COMMENT '分类图标',

status VARCHAR(20) DEFAULT 'active' COMMENT '状态',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_parent (parent\_id),

INDEX idx\_level (level),

INDEX idx\_sort (sort\_order)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='商品分类表';

```

分类层级设计:

```

电子数码 (level=1, parent\_id=0)

├── 手机通讯 (level=2, parent\_id=1)

│ ├── 智能手机 (level=3, parent\_id=2)

│ └── 手机配件 (level=3, parent\_id=2)

└── 电脑办公 (level=2, parent\_id=1)

├── 笔记本 (level=3, parent\_id=3)

└── 台式机 (level=3, parent\_id=3)

```

【表3:商品表 products】

```sql

CREATE TABLE products (

product\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '商品ID',

name VARCHAR(200) NOT NULL COMMENT '商品名称',

category\_id BIGINT NOT NULL COMMENT '分类ID',

price DECIMAL(10,2) NOT NULL COMMENT '价格',

stock\_quantity INT DEFAULT 0 COMMENT '库存数量',

sales\_count INT DEFAULT 0 COMMENT '销量',

description TEXT COMMENT '商品描述',

image\_url VARCHAR(500) COMMENT '主图URL',

rating DECIMAL(3,2) DEFAULT 0 COMMENT '平均评分(0-5)',

review\_count INT DEFAULT 0 COMMENT '评价数量',

brand VARCHAR(100) COMMENT '品牌',

model VARCHAR(100) COMMENT '型号',

status VARCHAR(20) DEFAULT 'active' COMMENT '状态:active/inactive',

is\_hot BOOLEAN DEFAULT FALSE COMMENT '是否热销',

is\_new BOOLEAN DEFAULT FALSE COMMENT '是否新品',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_category (category\_id),

INDEX idx\_price (price),

INDEX idx\_sales (sales\_count),

INDEX idx\_rating (rating),

INDEX idx\_status (status),

FULLTEXT idx\_name\_desc (name, description),

FOREIGN KEY (category\_id) REFERENCES categories(category\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='商品表';

```

设计要点:

- price使用DECIMAL(10,2)精确存储金额

- rating存储平均评分,每次新增评价时更新

- FULLTEXT全文索引支持中文商品搜索

- is\_hot和is\_new标记用于首页推荐

【表4:购物车表 cart】

```sql

CREATE TABLE cart (

cart\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '购物车项ID',

user\_id BIGINT NOT NULL COMMENT '用户ID',

product\_id BIGINT NOT NULL COMMENT '商品ID',

quantity INT NOT NULL DEFAULT 1 COMMENT '数量',

selected BOOLEAN DEFAULT TRUE COMMENT '是否选中',

price DECIMAL(10,2) NOT NULL COMMENT '加入时价格',

added\_at DATETIME DEFAULT CURRENT\_TIMESTAMP COMMENT '添加时间',

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

UNIQUE KEY uk\_user\_product (user\_id, product\_id),

INDEX idx\_user (user\_id),

INDEX idx\_product (product\_id),

FOREIGN KEY (user\_id) REFERENCES users(user\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='购物车表';

```

业务规则:

- uk\_user\_product唯一索引保证同一用户不会有重复商品

- price记录加入购物车时的价格,不随商品价格变动

- selected字段支持部分商品结算

【表5:订单主表 orders】

```sql

CREATE TABLE orders (

order\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '订单ID',

order\_no VARCHAR(32) UNIQUE NOT NULL COMMENT '订单编号',

user\_id BIGINT NOT NULL COMMENT '用户ID',

total\_amount DECIMAL(10,2) NOT NULL COMMENT '商品总额',

discount\_amount DECIMAL(10,2) DEFAULT 0 COMMENT '优惠金额',

shipping\_fee DECIMAL(10,2) DEFAULT 0 COMMENT '运费',

final\_amount DECIMAL(10,2) NOT NULL COMMENT '实付金额',

status VARCHAR(20) DEFAULT 'pending' COMMENT '状态',

payment\_method VARCHAR(50) COMMENT '支付方式',

payment\_time DATETIME COMMENT '支付时间',

shipping\_address TEXT COMMENT '收货地址(JSON)',

shipping\_time DATETIME COMMENT '发货时间',

tracking\_number VARCHAR(100) COMMENT '物流单号',

completed\_time DATETIME COMMENT '完成时间',

cancelled\_time DATETIME COMMENT '取消时间',

remark VARCHAR(500) COMMENT '订单备注',

coupon\_code VARCHAR(50) COMMENT '使用的优惠券',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_user (user\_id),

INDEX idx\_order\_no (order\_no),

INDEX idx\_status (status),

INDEX idx\_created (created\_at),

INDEX idx\_payment\_time (payment\_time),

FOREIGN KEY (user\_id) REFERENCES users(user\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='订单主表';

```

订单编号生成规则:

```cpp

std::string OrderService::generateOrderNo() {

// 格式:ORD + YYYYMMDD + 时分秒 + 6位随机数

// 示例:ORD20250115143022123456

auto now = std::chrono::system\_clock::now();

auto time\_t = std::chrono::system\_clock::to\_time\_t(now);

struct tm tm;

localtime\_s(&tm, &time\_t);

char buffer[32];

snprintf(buffer, sizeof(buffer),

"ORD%04d%02d%02d%02d%02d%02d%06d",

tm.tm\_year + 1900, tm.tm\_mon + 1, tm.tm\_mday,

tm.tm\_hour, tm.tm\_min, tm.tm\_sec,

rand() % 1000000);

return std::string(buffer);

}

```

【表6:订单详情表 order\_items】

```sql

CREATE TABLE order\_items (

item\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '订单项ID',

order\_id BIGINT NOT NULL COMMENT '订单ID',

product\_id BIGINT NOT NULL COMMENT '商品ID',

product\_name VARCHAR(200) NOT NULL COMMENT '商品名称快照',

product\_image VARCHAR(500) COMMENT '商品图片快照',

price DECIMAL(10,2) NOT NULL COMMENT '单价快照',

quantity INT NOT NULL COMMENT '数量',

subtotal DECIMAL(10,2) NOT NULL COMMENT '小计',

is\_reviewed BOOLEAN DEFAULT FALSE COMMENT '是否已评价',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

INDEX idx\_order (order\_id),

INDEX idx\_product (product\_id),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id) ON DELETE CASCADE,

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='订单详情表';

```

快照设计说明:

- 订单创建时保存商品名称、图片、价格快照

- 即使商品信息后续修改,订单记录保持不变

- ON DELETE CASCADE:订单删除时自动删除订单项

【表7:用户地址表 user\_addresses】

```sql

CREATE TABLE user\_addresses (

address\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '地址ID',

user\_id BIGINT NOT NULL COMMENT '用户ID',

receiver\_name VARCHAR(50) NOT NULL COMMENT '收货人',

phone VARCHAR(20) NOT NULL COMMENT '联系电话',

province VARCHAR(50) NOT NULL COMMENT '省份',

city VARCHAR(50) NOT NULL COMMENT '城市',

district VARCHAR(50) NOT NULL COMMENT '区县',

detail\_address VARCHAR(200) NOT NULL COMMENT '详细地址',

postal\_code VARCHAR(10) COMMENT '邮政编码',

is\_default BOOLEAN DEFAULT FALSE COMMENT '是否默认地址',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_user (user\_id),

INDEX idx\_default (is\_default),

FOREIGN KEY (user\_id) REFERENCES users(user\_id) ON DELETE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='用户地址表';

```

地址管理规则:

- 每个用户最多10个地址

- 设置新的默认地址时,自动取消其他地址的默认状态

- 用户删除时级联删除其所有地址

【表8:优惠券模板表 coupons】

```sql

CREATE TABLE coupons (

coupon\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '优惠券ID',

name VARCHAR(100) NOT NULL COMMENT '优惠券名称',

code VARCHAR(50) UNIQUE NOT NULL COMMENT '优惠券代码',

type VARCHAR(20) NOT NULL COMMENT '类型:fixed/percentage/free\_shipping',

value DECIMAL(10,2) NOT NULL COMMENT '面值或折扣率',

min\_amount DECIMAL(10,2) DEFAULT 0 COMMENT '最低消费金额',

total\_quantity INT NOT NULL COMMENT '发行总量',

used\_quantity INT DEFAULT 0 COMMENT '已使用数量',

per\_user\_limit INT DEFAULT 1 COMMENT '每人限领',

start\_date DATETIME NOT NULL COMMENT '生效时间',

end\_date DATETIME NOT NULL COMMENT '失效时间',

description VARCHAR(500) COMMENT '使用说明',

status VARCHAR(20) DEFAULT 'active' COMMENT '状态',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

INDEX idx\_code (code),

INDEX idx\_type (type),

INDEX idx\_dates (start\_date, end\_date),

INDEX idx\_status (status)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='优惠券模板表';

```

优惠券类型说明:

- fixed:固定金额券,value表示减免金额

- percentage:折扣券,value表示折扣百分比(如95表示95折)

- free\_shipping:包邮券,value为0

【表9:用户优惠券表 user\_coupons】

```sql

CREATE TABLE user\_coupons (

user\_coupon\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

user\_id BIGINT NOT NULL COMMENT '用户ID',

coupon\_id BIGINT NOT NULL COMMENT '优惠券ID',

order\_id BIGINT COMMENT '使用的订单ID',

status VARCHAR(20) DEFAULT 'unused' COMMENT '状态:unused/used/expired',

received\_at DATETIME DEFAULT CURRENT\_TIMESTAMP COMMENT '领取时间',

used\_at DATETIME COMMENT '使用时间',

expired\_at DATETIME NOT NULL COMMENT '过期时间',

INDEX idx\_user (user\_id),

INDEX idx\_coupon (coupon\_id),

INDEX idx\_status (status),

INDEX idx\_expired (expired\_at),

FOREIGN KEY (user\_id) REFERENCES users(user\_id),

FOREIGN KEY (coupon\_id) REFERENCES coupons(coupon\_id),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='用户优惠券表';

```

状态转换:

```

unused(未使用) → used(已使用)

unused(未使用) → expired(已过期)

```

【表10:商品评价表 product\_reviews】

```sql

CREATE TABLE product\_reviews (

review\_id BIGINT PRIMARY KEY AUTO\_INCREMENT COMMENT '评价ID',

product\_id BIGINT NOT NULL COMMENT '商品ID',

user\_id BIGINT NOT NULL COMMENT '用户ID',

order\_id BIGINT NOT NULL COMMENT '订单ID',

rating INT NOT NULL COMMENT '评分(1-5)',

content VARCHAR(1000) COMMENT '评价内容',

images VARCHAR(1000) COMMENT '评价图片(JSON数组)',

is\_anonymous BOOLEAN DEFAULT FALSE COMMENT '是否匿名',

helpful\_count INT DEFAULT 0 COMMENT '有用数',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

INDEX idx\_product (product\_id),

INDEX idx\_user (user\_id),

INDEX idx\_order (order\_id),

INDEX idx\_rating (rating),

INDEX idx\_created (created\_at),

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

FOREIGN KEY (user\_id) REFERENCES users(user\_id),

FOREIGN KEY (order\_id) REFERENCES orders(order\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='商品评价表';

```

3.3 助表

【表11:系统通知表 notifications】

```sql

CREATE TABLE notifications (

notification\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

user\_id BIGINT NOT NULL COMMENT '用户ID',

type VARCHAR(50) NOT NULL COMMENT '通知类型',

title VARCHAR(200) NOT NULL COMMENT '标题',

content TEXT COMMENT '内容',

related\_id BIGINT COMMENT '关联ID(订单ID等)',

is\_read BOOLEAN DEFAULT FALSE COMMENT '是否已读',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

INDEX idx\_user (user\_id),

INDEX idx\_type (type),

INDEX idx\_read (is\_read),

INDEX idx\_created (created\_at),

FOREIGN KEY (user\_id) REFERENCES users(user\_id)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='系统通知表';

```

通知类型:

- order\_status:订单状态变更通知

- promotion:促销活动通知

- coupon:优惠券到账通知

- system:系统公告

【表12:操作日志表 operation\_logs】

```sql

CREATE TABLE operation\_logs (

log\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

user\_id BIGINT COMMENT '操作用户ID',

operation VARCHAR(100) NOT NULL COMMENT '操作类型',

module VARCHAR(50) NOT NULL COMMENT '模块',

details TEXT COMMENT '操作详情(JSON)',

ip\_address VARCHAR(50) COMMENT 'IP地址',

user\_agent VARCHAR(500) COMMENT '用户代理',

created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,

INDEX idx\_user (user\_id),

INDEX idx\_operation (operation),

INDEX idx\_module (module),

INDEX idx\_created (created\_at)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='操作日志表';

```

3.4 数据库视图

【视图1:订单详情视图】

```sql

CREATE VIEW v\_order\_details AS

SELECT

o.order\_id,

o.order\_no,

o.user\_id,

u.username,

o.total\_amount,

o.discount\_amount,

o.final\_amount,

o.status,

o.payment\_method,

o.created\_at,

COUNT(oi.item\_id) as item\_count,

GROUP\_CONCAT(

CONCAT(oi.product\_name, '×', oi.quantity)

SEPARATOR ', '

) as product\_summary

FROM orders o

LEFT JOIN users u ON o.user\_id = u.user\_id

LEFT JOIN order\_items oi ON o.order\_id = oi.order\_id

GROUP BY o.order\_id;

```

【视图2:商品详情视图】

```sql

CREATE VIEW v\_product\_details AS

SELECT

p.product\_id,

p.name,

p.price,

p.stock\_quantity,

p.sales\_count,

p.rating,

p.review\_count,

c.name as category\_name,

c.parent\_id as parent\_category\_id

FROM products p

LEFT JOIN categories c ON p.category\_id = c.category\_id

WHERE p.status = 'active';

```

3.5 存储过程

【存储过程1:库存扣减】

```sql

DELIMITER $$

CREATE PROCEDURE UpdateProductStock(

IN p\_product\_id BIGINT,

IN p\_quantity INT,

IN p\_operation VARCHAR(10),

OUT p\_result INT

)

BEGIN

DECLARE current\_stock INT;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SET p\_result = -1;

ROLLBACK;

END;

START TRANSACTION;

-- 锁定商品行

SELECT stock\_quantity INTO current\_stock

FROM products

WHERE product\_id = p\_product\_id

FOR UPDATE;

IF p\_operation = 'subtract' THEN

IF current\_stock < p\_quantity THEN

SET p\_result = 0; -- 库存不足

ROLLBACK;

ELSE

UPDATE products

SET stock\_quantity = stock\_quantity - p\_quantity

WHERE product\_id = p\_product\_id;

SET p\_result = 1; -- 成功

COMMIT;

END IF;

ELSEIF p\_operation = 'add' THEN

UPDATE products

SET stock\_quantity = stock\_quantity + p\_quantity

WHERE product\_id = p\_product\_id;

SET p\_result = 1; -- 成功

COMMIT;

ELSE

SET p\_result = -2; -- 无效操作

ROLLBACK;

END IF;

END$$

DELIMITER ;

```

（4）UI设计

4.1 UI设计原则

【设计理念】

1. 用户友好:简洁直观的界面,降低学习成本

2. 一致性:统一的视觉风格和交互模式

3. 响应式:适应不同屏幕尺寸和分辨率

4. 反馈及时:操作结果实时反馈,避免用户等待焦虑

5. 容错性:友好的错误提示,支持操作撤销

【视觉规范】

- 主色调:渐变蓝(#4facfe → #00f2fe)

- 辅助色:灰色系(#f5f7fa背景,#2c3e50文字)

- 字体:微软雅黑,10pt正文,12pt标题

- 圆角:8px统一圆角半径

- 阴影:适度阴影增强层次感

- 间距:8px基准间距,16px模块间距

4.2 主要界面设计

【界面1:登录注册界面 LoginDialog】

功能要点:

- 用户名/密码输入框,支持Tab键切换

- 密码框显示/隐藏切换按钮

- "记住密码"复选框,下次自动填充

- 登录按钮为主操作,注册按钮为次要操作

- 显示友好的错误提示(用户名密码错误、网络异常等)

QSS样式示例:

```css

QDialog {

background-color: #f5f7fa;

}

QLineEdit {

background-color: #ffffff;

border: 2px solid #dfe6e9;

border-radius: 8px;

padding: 8px 12px;

color: #2c3e50;

font-size: 10pt;

}

QLineEdit:focus {

border-color: #3498db;

background-color: #f8f9fa;

}

QPushButton#loginButton {

background: qlineargradient(x1:0, y1:0, x2:1, y2:1,

stop:0 #4facfe, stop:1 #00f2fe);

color: white;

border: none;

border-radius: 8px;

padding: 10px 30px;

font-size: 11pt;

font-weight: 500;

}

QPushButton#loginButton:hover {

background: qlineargradient(x1:0, y1:0, x2:1, y2:1,

stop:0 #3d9bef, stop:1 #00d9e5);

}

```

【界面2:主窗口 MainWindow】

窗口组成:

- 顶部导航栏:显示用户名、通知中心、设置、退出按钮

- 左侧导航:商品、购物车、订单、优惠券、个人中心(管理员增加管理员页)

- 右侧内容区:QStackedWidget实现多页面切换

- 状态栏:显示连接状态、操作提示

导航切换信号槽:

```cpp

// MainWindow.cpp

void MainWindow::setupNavigationBar() {

connect(ui->btnProducts, &QPushButton::clicked,

this, [this]() {

ui->stackedWidget->setCurrentWidget(productsTab);

updateNavigationStyle(ui->btnProducts);

});

connect(ui->btnCart, &QPushButton::clicked,

this, [this]() {

ui->stackedWidget->setCurrentWidget(cartTab);

updateNavigationStyle(ui->btnCart);

cartTab->refreshCart(); // 刷新购物车数据

});

// ... 其他导航按钮

}

void MainWindow::updateNavigationStyle(QPushButton\* activeBtn) {

// 重置所有按钮样式

for (auto btn : navigationButtons) {

btn->setProperty("active", false);

btn->style()->unpolish(btn);

btn->style()->polish(btn);

}

// 激活当前按钮

activeBtn->setProperty("active", true);

activeBtn->style()->unpolish(activeBtn);

activeBtn->style()->polish(activeBtn);

}

```

【界面3:商品浏览页 ProductsTab】

功能实现:

- QGridLayout网格布局展示商品卡片

- 自定义ProductCard小部件封装商品信息

- 支持关键词搜索、分类筛选、多维度排序

- 分页加载,每页显示20个商品

- 商品卡片悬停显示详细信息预览

商品卡片类:

```cpp

class ProductCard : public QWidget {

Q\_OBJECT

public:

ProductCard(const QJsonObject& product, QWidget\* parent = nullptr);

long getProductId() const { return productId; }

QString getProductName() const { return productName; }

double getPrice() const { return price; }

signals:

void addToCartClicked(long productId, int quantity);

void viewDetailClicked(long productId);

private:

void setupUI();

void loadProductImage();

long productId;

QString productName;

double price;

int stock;

QLabel\* imageLabel;

QLabel\* nameLabel;

QLabel\* priceLabel;

QLabel\* ratingLabel;

QPushButton\* addToCartBtn;

};

```

【界面4:购物车页 CartTab】

交互逻辑:

- 复选框控制商品选中/取消,实时更新总价

- 数量增减按钮,最小值1,最大值受库存限制

- 超出库存时显示红色警告提示

- 删除商品弹出确认对话框

- 结算按钮显示已选商品数量

- 选择优惠券打开优惠券选择对话框

购物车刷新实现:

```cpp

void CartTab::refreshCart() {

// 发送获取购物车请求

QString request = "GET\_CART";

networkClient->sendRequest(request);

}

void CartTab::onCartDataReceived(const QJsonObject& response) {

if (!response["success"].toBool()) {

QMessageBox::warning(this, "错误",

response["message"].toString());

return;

}

// 清空现有列表

cartItemsLayout->removeAllWidgets();

cartItems.clear();

// 解析购物车数据

QJsonArray items = response["data"]["items"].toArray();

for (const QJsonValue& value : items) {

QJsonObject item = value.toObject();

CartItemWidget\* widget = new CartItemWidget(item, this);

connect(widget, &CartItemWidget::quantityChanged,

this, &CartTab::onItemQuantityChanged);

connect(widget, &CartItemWidget::deleteClicked,

this, &CartTab::onItemDeleteClicked);

connect(widget, &CartItemWidget::selectedChanged,

this, &CartTab::recalculateTotal);

cartItemsLayout->addWidget(widget);

cartItems.append(widget);

}

// 更新总价

recalculateTotal();

}

void CartTab::recalculateTotal() {

double totalPrice = 0.0;

int selectedCount = 0;

for (CartItemWidget\* item : cartItems) {

if (item->isSelected()) {

totalPrice += item->getSubtotal();

selectedCount++;

}

}

// 更新UI显示

ui->labelSelectedCount->setText(

QString("已选商品(%1)").arg(selectedCount));

ui->labelTotalPrice->setText(

QString("¥%1").arg(totalPrice, 0, 'f', 2));

// 应用优惠券折扣

double discountAmount = calculateCouponDiscount(totalPrice);

ui->labelDiscount->setText(

QString("-¥%1").arg(discountAmount, 0, 'f', 2));

// 运费计算(满299包邮)

double shippingFee = (totalPrice >= 299.0) ? 0.0 : 10.0;

ui->labelShippingFee->setText(

QString("¥%1").arg(shippingFee, 0, 'f', 2));

// 实付金额

double finalAmount = totalPrice - discountAmount + shippingFee;

ui->labelFinalAmount->setText(

QString("¥%1").arg(finalAmount, 0, 'f', 2));

// 更新结算按钮

ui->btnCheckout->setEnabled(selectedCount > 0);

ui->btnCheckout->setText(

QString("去结算(%1)").arg(selectedCount));

}

```

【界面5:订单列表页 OrdersTab】

状态筛选实现:

```cpp

void OrdersTab::onStatusFilterChanged(const QString& status) {

currentStatus = status;

currentPage = 1;

loadOrders();

}

void OrdersTab::loadOrders() {

QString request = QString("GET\_USER\_ORDERS %1 %2 %3")

.arg(currentStatus)

.arg(currentPage)

.arg(pageSize);

networkClient->sendRequest(request);

}

```

【界面6:支付对话框 PaymentDialog】

支付流程:

```cpp

void PaymentDialog::onConfirmPayment() {

// 1. 验证支付密码

QString password = ui->lineEditPassword->text();

if (password.isEmpty()) {

QMessageBox::warning(this, "提示", "请输入支付密码");

return;

}

// 2. 获取选择的支付方式

QString paymentMethod;

if (ui->radioAlipay->isChecked()) {

paymentMethod = "alipay";

} else if (ui->radioWechat->isChecked()) {

paymentMethod = "wechat";

} else if (ui->radioBankCard->isChecked()) {

paymentMethod = "bank\_card";

}

// 3. 发送支付请求

QString request = QString("PROCESS\_PAYMENT %1 %2 %3 {}")

.arg(orderId)

.arg(paymentMethod)

.arg(amount);

networkClient->sendRequest(request);

// 4. 显示加载动画

QProgressDialog\* progressDialog = new QProgressDialog(

"正在处理支付...", nullptr, 0, 0, this);

progressDialog->setWindowModality(Qt::WindowModal);

progressDialog->show();

}

void PaymentDialog::onPaymentResponse(const QJsonObject& response) {

if (response["success"].toBool()) {

QMessageBox::information(this, "成功", "支付成功!");

accept(); // 关闭对话框

emit paymentSuccessful(orderId);

} else {

QMessageBox::critical(this, "失败",

response["message"].toString());

}

}

```

（5）协议设计

5.1 应用层协议定义

【协议格式】

采用文本行协议,每条消息以换行符(\n)分隔

请求格式:

```

METHOD [PARAM1] [PARAM2] ... [PARAMN]\n

```

响应格式(JSON):

```json

{

"success": true/false,

"message": "操作结果描述",

"data": {

// 业务数据

},

"error\_code": "错误码(可选)",

"timestamp": 1705123456789

}\n

```

【命令示例】

```

// 用户登录

请求: LOGIN admin admin123

响应: {"success":true,"message":"登录成功","data":{"user\_id":1,"username":"admin","role":"admin","token":"sess\_1\_1705123456"}}

// 获取商品列表

请求: GET\_PRODUCTS electronics 1 10

响应: {"success":true,"message":"获取商品成功","data":{"products":[...],"total":45,"page":1,"page\_size":10}}

// 添加到购物车

请求: ADD\_TO\_CART 123 2

响应: {"success":true,"message":"添加成功","data":{"cart\_id":456}}

// 创建订单

请求: CREATE\_ORDER 1 "北京市朝阳区xxx" COUPON100 "请尽快发货"

响应: {"success":true,"message":"订单创建成功","data":{"order\_id":789,"order\_no":"ORD20250115143022123456","final\_amount":447.00}}

```

5.2 参数处理

【引号支持】

参数包含空格时使用双引号包围:

```

ADD\_ADDRESS 张三 "13812345678" "北京市" "朝阳区" "三里屯街道" "SOHO大厦1号楼"

```

解析算法:

```cpp

std::vector<std::string> parseCommand(const std::string& command) {

std::vector<std::string> parts;

bool inQuotes = false;

std::string current;

bool escaped = false;

for (char c : command) {

if (escaped) {

if (c == '\\' || c == '"') {

current += c;

} else {

current += '\\';

current += c;

}

escaped = false;

} else if (c == '\\') {

escaped = true;

} else if (c == '"') {

inQuotes = !inQuotes;

} else if (c == ' ' && !inQuotes) {

if (!current.empty()) {

parts.push\_back(current);

current.clear();

}

} else {

current += c;

}

}

if (!current.empty()) {

parts.push\_back(current);

}

return parts;

}

```

5.3 错误码系统

【错误码规范】

格式: `EXXXYY`

- XX: 模块代码(00-99)

- Y: 错误类型(0-9)

- YY: 具体错误序号(00-99)

模块代码分配:

- 00:通用

- 10:用户认证

- 20:商品管理

- 30:购物车

- 40:订单管理

- 50:支付系统

- 60:优惠券

- 70:促销活动

- 80:地址管理

- 90:系统管理

错误类型:

- 0x:系统级错误

- 1x:验证错误

- 2x:业务错误

- 3x:权限错误

- 4x:资源错误

常用错误码:

```

E00001 - 数据库连接失败

E00101 - 请求参数缺失

E10201 - 用户名或密码错误

E10302 - Token已过期

E20201 - 库存不足

E40202 - 订单状态错误

E50201 - 支付失败

```

（6）设计模式使用

6.1 单例模式 (Singleton)

【应用场景】

- DatabaseManager:数据库连接管理器

- ConfigLoader:配置加载器

- Logger:日志记录器

【实现代码】

```cpp

class DatabaseManager {

private:

static DatabaseManager\* instance;

static std::mutex mutex;

// 私有构造函数

DatabaseManager() {

// 初始化连接池

}

// 禁止拷贝构造和赋值

DatabaseManager(const DatabaseManager&) = delete;

DatabaseManager& operator=(const DatabaseManager&) = delete;

public:

static DatabaseManager& getInstance() {

// C++11保证线程安全

static DatabaseManager instance;

return instance;

}

Connection\* getConnection();

void releaseConnection(Connection\* conn);

};

```

6.2 工厂模式 (Factory)

【应用场景】

- ServiceFactory:创建各种业务服务实例

【实现代码】

```cpp

class ServiceFactory {

public:

enum class ServiceType {

USER,

PRODUCT,

CART,

ORDER,

COUPON

};

static std::shared\_ptr<BaseService> createService(

ServiceType type,

DatabaseManager& dbManager) {

switch (type) {

case ServiceType::USER:

return std::make\_shared<UserService>(dbManager);

case ServiceType::PRODUCT:

return std::make\_shared<ProductService>(dbManager);

case ServiceType::CART:

return std::make\_shared<CartService>(dbManager);

case ServiceType::ORDER:

return std::make\_shared<OrderService>(dbManager);

case ServiceType::COUPON:

return std::make\_shared<CouponService>(dbManager);

default:

throw std::invalid\_argument("Unknown service type");

}

}

};

// 使用示例

auto userService = ServiceFactory::createService(

ServiceFactory::ServiceType::USER,

DatabaseManager::getInstance()

);

```

6.3 策略模式 (Strategy)

【应用场景】

- 优惠券折扣计算:不同类型优惠券使用不同计算策略

【实现代码】

```cpp

// 折扣计算策略接口

class DiscountStrategy {

public:

virtual ~DiscountStrategy() = default;

virtual double calculate(double originalPrice,

double couponValue) const = 0;

};

// 固定金额折扣

class FixedAmountDiscount : public DiscountStrategy {

public:

double calculate(double originalPrice,

double couponValue) const override {

return std::max(0.0, originalPrice - couponValue);

}

};

// 百分比折扣

class PercentageDiscount : public DiscountStrategy {

public:

double calculate(double originalPrice,

double couponValue) const override {

double discountRate = couponValue / 100.0;

return originalPrice \* (1.0 - discountRate);

}

};

// 满减折扣

class ThresholdDiscount : public DiscountStrategy {

private:

double threshold;

double reduction;

public:

ThresholdDiscount(double threshold, double reduction)

: threshold(threshold), reduction(reduction) {}

double calculate(double originalPrice,

double couponValue) const override {

if (originalPrice >= threshold) {

return originalPrice - reduction;

}

return originalPrice;

}

};

// 使用策略

class CouponService : public BaseService {

private:

std::unique\_ptr<DiscountStrategy> strategy;

public:

void setStrategy(std::unique\_ptr<DiscountStrategy> s) {

strategy = std::move(s);

}

double applyDiscount(double originalPrice, double value) {

if (strategy) {

return strategy->calculate(originalPrice, value);

}

return originalPrice;

}

};

```

6.4 观察者模式 (Observer)

【应用场景】

- Qt信号槽机制天然支持观察者模式

- 订单状态变更通知

【实现代码】

```cpp

// Qt信号槽示例

class OrderService : public QObject {

Q\_OBJECT

signals:

void orderStatusChanged(long orderId, const QString& newStatus);

void orderCreated(long orderId, double amount);

void orderCancelled(long orderId);

public:

void updateOrderStatus(long orderId, const QString& status) {

// 更新数据库

// ...

// 发出信号

emit orderStatusChanged(orderId, status);

// 发送通知

notifyUser(orderId, status);

}

};

// 观察者

class NotificationManager : public QObject {

Q\_OBJECT

public slots:

void onOrderStatusChanged(long orderId, const QString& status) {

// 生成通知消息

QString message = QString("您的订单%1状态变更为:%2")

.arg(orderId).arg(status);

// 发送通知

sendNotification(message);

}

};

// 连接信号槽

connect(orderService, &OrderService::orderStatusChanged,

notificationMgr, &NotificationManager::onOrderStatusChanged);

```

6.5 模板方法模式 (Template Method)

【应用场景】

- BaseService定义通用业务处理流程

【实现代码】

```cpp

class BaseService {

protected:

DatabaseManager& dbManager;

// 模板方法:定义业务处理流程

json executeBusinessOperation(

const std::string& operation,

std::function<json()> businessLogic) {

try {

// 1. 前置验证

if (!validateOperation(operation)) {

return createErrorResponse("权限验证失败");

}

// 2. 开启事务

dbManager.beginTransaction();

// 3. 执行业务逻辑(由子类实现)

json result = businessLogic();

// 4. 提交事务

dbManager.commit();

// 5. 记录日志

logSuccess(operation, result);

return result;

} catch (const std::exception& e) {

// 回滚事务

dbManager.rollback();

// 记录错误

logError(operation, e.what());

return createErrorResponse(e.what());

}

}

// 钩子方法:子类可重写

virtual bool validateOperation(const std::string& op) {

return true;

}

public:

BaseService(DatabaseManager& db) : dbManager(db) {}

virtual ~BaseService() = default;

};

// 子类使用

class UserService : public BaseService {

public:

json register(const std::string& username,

const std::string& password) {

return executeBusinessOperation("REGISTER", [&]() {

// 具体业务逻辑

validateUsername(username);

std::string hashedPassword = hashPassword(password);

json result = executeUpdate(

"INSERT INTO users (username, password) VALUES (?, ?)",

{username, hashedPassword}

);

return createSuccessResponse("注册成功", result["data"]);

});

}

};

```

6.6 RAII模式 (Resource Acquisition Is Initialization)

【应用场景】

- 数据库连接自动释放

- JNI资源管理

【实现代码】

```cpp

// 数据库连接RAII包装

class ConnectionGuard {

private:

DatabaseManager& manager;

Connection\* connection;

public:

ConnectionGuard(DatabaseManager& mgr)

: manager(mgr), connection(nullptr) {

connection = manager.getConnection();

if (!connection) {

throw std::runtime\_error("Failed to get connection");

}

}

~ConnectionGuard() {

if (connection) {

manager.releaseConnection(connection);

}

}

Connection\* get() { return connection; }

// 禁止拷贝

ConnectionGuard(const ConnectionGuard&) = delete;

ConnectionGuard& operator=(const ConnectionGuard&) = delete;

};

// 使用示例

void UserService::queryUser(long userId) {

ConnectionGuard guard(dbManager);

Connection\* conn = guard.get();

// 使用连接查询

// ...

// 函数结束时自动释放连接

}

三、实现 - 编译环境与核心流程

三、实现

（1）编译环境

1.1 开发环境配置

【操作系统】

- Windows 10/11 64-bit

- 推荐内存:8GB以上

- 磁盘空间:10GB以上

【C++服务端环境】

编译器:

- MinGW-w64 GCC 15.2.0

- 支持C++17标准

- 64位编译

JDK配置:

- OpenJDK 21.0.1

- JAVA\_HOME环境变量设置

- JNI头文件路径:

\* C:\Program Files\Java\jdk-21\include

\* C:\Program Files\Java\jdk-21\include\win32

数据库驱动:

- MySQL Connector/C++ 8.0

- MySQL头文件:D:\MySQL\include

- MySQL库文件:D:\MySQL\lib\libmysql.dll

第三方库:

- nlohmann/json 3.12.0 (单头文件库)

- 路径:cpp/nlohmann\_json.hpp

编译命令:

```batch

g++ -std=c++17 -shared -O2 -DNDEBUG ^

"-IC:\Program Files\Java\jdk-21\include" ^

"-IC:\Program Files\Java\jdk-21\include\win32" ^

-I"D:\MySQL\include" ^

-o emshop\_native\_impl\_oop.dll ^

emshop\_native\_impl\_oop.cpp ^

services\UserService.cpp ^

services\ProductService.cpp ^

services\CartService.cpp ^

services\OrderService.cpp ^

services\CouponService.cpp ^

services\AddressService.cpp ^

services\ReviewService.cpp ^

libmysql.dll

```

编译选项说明:

- `-std=c++17`: 使用C++17标准

- `-shared`: 生成动态链接库(.dll)

- `-O2`: 二级优化

- `-DNDEBUG`: 禁用调试代码,提升性能

- `-I`: 指定头文件搜索路径

【Java服务端环境】

JDK版本:

- OpenJDK 21.0.1

- 支持Java 21新特性

构建工具:

- Apache Maven 3.9.6

- pom.xml配置文件管理依赖

核心依赖:

```xml

<!-- Netty网络框架 -->

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-all</artifactId>

<version>4.1.115.Final</version>

</dependency>

<!-- SLF4J日志框架 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.16</version>

</dependency>

<!-- Logback日志实现 -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.5.12</version>

</dependency>

<!-- Jackson JSON处理 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.18.2</version>

</dependency>

<!-- Gson JSON处理(备用) -->

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.11.0</version>

</dependency>

```

Maven编译命令:

```batch

mvn clean package

```

生成文件:

- target/emshop-server.jar (可执行JAR包)

- 包含所有依赖(通过maven-shade-plugin打包)

【Qt客户端环境】

Qt版本:

- Qt 6.9.1 (LTS版本)

- MinGW 64-bit编译器

Qt模块:

- Qt Core:核心功能

- Qt Widgets:GUI控件

- Qt Network:网络通信

- Qt Gui:图形界面

构建工具:

- CMake 3.30+

- 配置文件:CMakeLists.txt

编译步骤:

```batch

# 1. 配置项目

cmake -S qtclient -B qtclient/build -G "MinGW Makefiles" ^

-DCMAKE\_PREFIX\_PATH=D:/Qt/6.9.1/mingw\_64

# 2. 编译项目

cmake --build qtclient/build -- -j

# 3. 生成可执行文件

# 输出:qtclient/build/src/emshop\_qtclient.exe

```

CMakeLists.txt核心配置:

```cmake

cmake\_minimum\_required(VERSION 3.30)

project(emshop\_qtclient VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

# 查找Qt包

find\_package(Qt6 REQUIRED COMPONENTS Core Widgets Network)

# 添加可执行文件

add\_executable(emshop\_qtclient

src/main.cpp

src/core/ApplicationContext.cpp

src/network/NetworkClient.cpp

src/ui/MainWindow.cpp

src/ui/LoginDialog.cpp

# ...其他源文件

)

# 链接Qt库

target\_link\_libraries(emshop\_qtclient

Qt6::Core

Qt6::Widgets

Qt6::Network

)

```

【数据库环境】

MySQL版本:

- MySQL 8.0.40

- InnoDB存储引擎

- utf8mb4字符集

配置文件(my.ini):

```ini

[mysqld]

# 端口配置

port = 3306

# 字符集配置

character-set-server = utf8mb4

collation-server = utf8mb4\_unicode\_ci

# 默认存储引擎

default-storage-engine = INNODB

# 最大连接数

max\_connections = 500

# 查询缓存

query\_cache\_type = 1

query\_cache\_size = 64M

# InnoDB配置

innodb\_buffer\_pool\_size = 1G

innodb\_log\_file\_size = 256M

innodb\_flush\_log\_at\_trx\_commit = 2

```

数据库初始化:

```batch

# 1. 登录MySQL

mysql -u root -p

# 2. 创建数据库

CREATE DATABASE emshop CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci;

# 3. 导入初始化脚本

USE emshop;

SOURCE cpp/emshop\_database\_init.sql;

# 4. 导入测试数据

SOURCE cpp/init\_coupons.sql;

```

1.2 开发工具

【IDE环境】

C++开发:

- Visual Studio Code 1.95

- 扩展:C/C++、CMake Tools

- 或Visual Studio 2022 Community

Java开发:

- IntelliJ IDEA 2024.3 Community

- Maven Helper插件

- Lombok插件

Qt开发:

- Qt Creator 14.0.2

- 集成Qt Designer界面设计器

- 代码提示和调试支持

【版本控制】

Git:

- Git 2.47.1 for Windows

- GitHub Desktop (可选)

提交规范:

```

feat: 添加用户登录功能

fix: 修复购物车数量更新bug

docs: 更新README文档

style: 代码格式化

refactor: 重构订单服务

test: 添加单元测试

chore: 更新依赖版本

```

【调试工具】

C++调试:

- GDB (MinGW内置)

- VS Code调试配置:launch.json

Java调试:

- IntelliJ IDEA内置调试器

- 断点、变量监视、调用栈

Qt调试:

- Qt Creator调试器

- qDebug()输出调试信息

数据库工具:

- MySQL Workbench 8.0

- Navicat Premium (可选)

- DBeaver Community (可选)

网络抓包:

- Wireshark 4.4.2

- 分析TCP通信内容

（2）核心流程

2.1 系统启动流程

【服务端启动流程】

```

1. 加载配置文件(config.json)

├── 读取数据库配置

├── 读取服务器端口

└── 读取JNI库路径

2. 初始化JNI本地库

├── System.loadLibrary("emshop\_native\_impl\_oop")

├── 调用JNI\_OnLoad函数

└── 注册本地方法

3. 初始化数据库连接池

├── 创建初始连接(5个)

├── 验证数据库可用性

└── 启动连接监控线程

4. 初始化业务服务

├── UserService初始化

├── ProductService初始化

├── OrderService初始化

└── 其他服务初始化

5. 启动Netty服务器

├── 创建Boss和Worker线程组

├── 配置ServerBootstrap

├── 绑定端口8888

└── 等待客户端连接

6. 启动成功

└── 输出日志:"Server started on port 8888"

```

启动代码:

```java

public class EmshopNettyServer {

public static void main(String[] args) {

try {

// 1. 加载配置

ConfigLoader config = new ConfigLoader("config.json");

// 2. 加载JNI库

String jniPath = config.getJniLibraryPath();

System.load(jniPath);

logger.info("JNI library loaded: {}", jniPath);

// 3. 初始化系统

String initResult = EmshopNativeInterface.initializeService();

logger.info("System initialized: {}", initResult);

// 4. 启动Netty服务器

int port = config.getServerPort();

EmshopNettyServer server = new EmshopNettyServer(port);

server.start();

} catch (Exception e) {

logger.error("Failed to start server", e);

System.exit(1);

}

}

}

```

【客户端启动流程】

```

1. 初始化Qt应用程序

├── QApplication app(argc, argv)

├── 设置应用程序名称

└── 加载全局样式表

2. 创建应用上下文

├── ApplicationContext初始化

├── 加载配置信息

└── 初始化网络客户端

3. 显示登录对话框

├── LoginDialog loginDialog

├── 等待用户输入

└── 验证登录凭据

4. 连接服务器

├── NetworkClient::connectToServer()

├── TCP连接到localhost:8888

└── 等待连接成功

5. 登录成功后显示主窗口

├── MainWindow mainWindow

├── 加载用户信息

├── 初始化各个Tab页面

└── mainWindow.show()

6. 进入事件循环

└── return app.exec()

```

main.cpp:

```cpp

int main(int argc, char \*argv[]) {

QApplication app(argc, argv);

QApplication::setApplicationName("EmshopClient");

QApplication::setOrganizationName("JLU");

// 加载全局样式

app.setStyleSheet(loadStyleSheet());

// 创建应用上下文

ApplicationContext context;

// 显示登录对话框

LoginDialog loginDialog(context);

if (loginDialog.exec() != QDialog::Accepted) {

return 0; // 用户取消登录

}

// 显示主窗口

MainWindow mainWindow(context);

mainWindow.show();

return app.exec();

}

```

2.2 用户登录流程

客户端登录代码:

```cpp

void LoginDialog::onLoginClicked() {

QString username = ui->lineEditUsername->text().trimmed();

QString password = ui->lineEditPassword->text();

// 验证输入

if (username.isEmpty() || password.isEmpty()) {

QMessageBox::warning(this, "提示", "请输入用户名和密码");

return;

}

// 禁用按钮,显示加载状态

ui->btnLogin->setEnabled(false);

ui->btnLogin->setText("登录中...");

// 发送登录请求

QString request = QString("LOGIN %1 %2")

.arg(username).arg(password);

networkClient->sendRequest(request, [this](const QString& response) {

onLoginResponse(response);

});

}

void LoginDialog::onLoginResponse(const QString& response) {

// 解析JSON响应

QJsonDocument doc = QJsonDocument::fromJson(response.toUtf8());

QJsonObject obj = doc.object();

bool success = obj["success"].toBool();

// 恢复按钮状态

ui->btnLogin->setEnabled(true);

ui->btnLogin->setText("登录");

if (success) {

// 保存用户信息

QJsonObject data = obj["data"].toObject();

long userId = data["user\_id"].toVariant().toLongLong();

QString username = data["username"].toString();

QString role = data["role"].toString();

context.setUserId(userId);

context.setUsername(username);

context.setRole(role);

// 关闭对话框

accept();

} else {

// 显示错误信息

QString message = obj["message"].toString();

QMessageBox::warning(this, "登录失败", message);

}

}

```

服务端登录处理:

```java

case "LOGIN":

if (parts.length >= 3) {

String username = parts[1];

String loginResult = EmshopNativeInterface.login(username, parts[2]);

// 如果登录成功,保存会话信息

if (loginResult.contains("\"success\":true")) {

long userId = extractUserIdFromResponse(loginResult);

String role = extractRoleFromResponse(loginResult);

session = new UserSession(userId, username, role);

userSessions.put(ctx.channel().id(), session);

logger.info("User login successful - username={}, userId={}, role={}",

username, userId, role);

}

return loginResult;

}

break;

```

C++ JNI登录实现:

```cpp

JNIEXPORT jstring JNICALL Java\_emshop\_EmshopNativeInterface\_login

(JNIEnv \*env, jclass, jstring jUsername, jstring jPassword) {

const char\* username = env->GetStringUTFChars(jUsername, nullptr);

const char\* password = env->GetStringUTFChars(jPassword, nullptr);

try {

// 1. 查询用户

json result = executeQuery(

"SELECT user\_id, username, password, role, status "

"FROM users WHERE username = ?",

{username}

);

if (result["data"].empty()) {

return env->NewStringUTF(

R"({"success":false,"message":"用户不存在","error\_code":"E10201"})"

);

}

// 2. 验证密码

std::string storedPassword = result["data"][0]["password"];

std::string hashedInput = sha256(password);

if (hashedInput != storedPassword) {

return env->NewStringUTF(

R"({"success":false,"message":"密码错误","error\_code":"E10201"})"

);

}

// 3. 检查账户状态

std::string status = result["data"][0]["status"];

if (status != "active") {

return env->NewStringUTF(

R"({"success":false,"message":"账户已被禁用","error\_code":"E10204"})"

);

}

// 4. 生成Token

long userId = result["data"][0]["user\_id"];

std::string token = generateToken(userId);

// 5. 更新最后登录时间

executeUpdate(

"UPDATE users SET last\_login\_time = NOW() WHERE user\_id = ?",

{userId}

);

// 6. 构造响应

json response;

response["success"] = true;

response["message"] = "登录成功";

response["data"]["user\_id"] = userId;

response["data"]["username"] = result["data"][0]["username"];

response["data"]["role"] = result["data"][0]["role"];

response["data"]["token"] = token;

std::string responseStr = response.dump();

return env->NewStringUTF(responseStr.c\_str());

} catch (const std::exception& e) {

json errorResponse;

errorResponse["success"] = false;

errorResponse["message"] = e.what();

errorResponse["error\_code"] = "E00001";

std::string errorStr = errorResponse.dump();

return env->NewStringUTF(errorStr.c\_str());

} finally {

env->ReleaseStringUTFChars(jUsername, username);

env->ReleaseStringUTFChars(jPassword, password);

}

}

```

2.3 商品浏览流程

【商品列表加载流程】

```

1. 用户点击"商品"导航按钮

└── MainWindow切换到ProductsTab

2. ProductsTab初始化

├── 设置默认分类"全部"

├── 设置默认页码1

└── 调用loadProducts()

3. 发送GET\_PRODUCTS请求

└── GET\_PRODUCTS all 1 20

4. Netty服务器路由到JNI

└── EmshopNativeInterface.getProductList("all", 1, 20)

5. C++执行SQL查询

├── SELECT \* FROM products WHERE status='active'

├── 分页:LIMIT 20 OFFSET 0

└── 返回商品数组JSON

6. 客户端接收响应

├── 解析JSON数据

├── 创建ProductCard小部件

└── 添加到QGridLayout

7. 显示商品列表

└── 用户可以浏览、搜索、筛选

```

ProductsTab代码:

```cpp

void ProductsTab::loadProducts() {

// 构造请求

QString request = QString("GET\_PRODUCTS %1 %2 %3")

.arg(currentCategory)

.arg(currentPage)

.arg(pageSize);

// 显示加载动画

showLoadingIndicator();

// 发送请求

networkClient->sendRequest(request, [this](const QString& response) {

hideLoadingIndicator();

onProductsReceived(response);

});

}

void ProductsTab::onProductsReceived(const QString& response) {

QJsonDocument doc = QJsonDocument::fromJson(response.toUtf8());

QJsonObject obj = doc.object();

if (!obj["success"].toBool()) {

QMessageBox::warning(this, "错误", obj["message"].toString());

return;

}

// 清空现有商品

clearProductCards();

// 解析商品数据

QJsonObject data = obj["data"].toObject();

QJsonArray products = data["products"].toArray();

int total = data["total"].toInt();

// 创建商品卡片

int row = 0, col = 0;

const int columns = 4; // 每行4个商品

for (const QJsonValue& value : products) {

QJsonObject product = value.toObject();

ProductCard\* card = new ProductCard(product, this);

connect(card, &ProductCard::addToCartClicked,

this, &ProductsTab::onAddToCart);

connect(card, &ProductCard::viewDetailClicked,

this, &ProductsTab::onViewDetail);

productsLayout->addWidget(card, row, col);

col++;

if (col >= columns) {

col = 0;

row++;

}

}

// 更新分页控件

updatePagination(total);

}

```

================================================================================

JLU Emshop 电商系统 - 综合项目报告(第六部分)

三、实现(续) - 核心方法与功能实现

================================================================================

（3）核心方法

3.1 UserService核心方法

【用户注册方法】

```cpp

// UserService.cpp

std::string UserService::registerUser(

const std::string& username,

const std::string& password,

const std::string& phone,

const std::string& email) {

try {

// 1. 参数验证

if (username.length() < 3 || username.length() > 20) {

return buildErrorResponse("用户名长度必须在3-20个字符之间", "E10101");

}

if (password.length() < 6) {

return buildErrorResponse("密码长度至少6个字符", "E10102");

}

if (!isValidPhone(phone)) {

return buildErrorResponse("手机号格式不正确", "E10103");

}

if (!isValidEmail(email)) {

return buildErrorResponse("邮箱格式不正确", "E10104");

}

// 2. 检查用户名是否已存在

json existCheck = executeQuery(

"SELECT COUNT(\*) as count FROM users WHERE username = ?",

{username}

);

if (existCheck["data"][0]["count"].get<int>() > 0) {

return buildErrorResponse("用户名已被注册", "E10105");

}

// 3. 检查手机号是否已存在

json phoneCheck = executeQuery(

"SELECT COUNT(\*) as count FROM users WHERE phone = ?",

{phone}

);

if (phoneCheck["data"][0]["count"].get<int>() > 0) {

return buildErrorResponse("手机号已被注册", "E10106");

}

// 4. 加密密码

std::string hashedPassword = sha256(password);

// 5. 插入数据库

int affectedRows = executeUpdate(

"INSERT INTO users (username, password, phone, email, role, status, "

"register\_time, last\_login\_time) "

"VALUES (?, ?, ?, ?, 'customer', 'active', NOW(), NOW())",

{username, hashedPassword, phone, email}

);

if (affectedRows == 0) {

return buildErrorResponse("注册失败", "E10107");

}

// 6. 获取新插入的user\_id

json userIdQuery = executeQuery(

"SELECT user\_id FROM users WHERE username = ?",

{username}

);

long userId = userIdQuery["data"][0]["user\_id"].get<long>();

// 7. 构造成功响应

json response;

response["success"] = true;

response["message"] = "注册成功";

response["data"]["user\_id"] = userId;

response["data"]["username"] = username;

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E10199");

}

}

```

【密码加密方法】

```cpp

// UserService.cpp

std::string UserService::sha256(const std::string& input) {

// 使用标准库实现SHA-256哈希

// 这里简化展示,实际使用OpenSSL或类似库

unsigned char hash[SHA256\_DIGEST\_LENGTH];

SHA256\_CTX sha256;

SHA256\_Init(&sha256);

SHA256\_Update(&sha256, input.c\_str(), input.length());

SHA256\_Final(hash, &sha256);

std::stringstream ss;

for (int i = 0; i < SHA256\_DIGEST\_LENGTH; i++) {

ss << std::hex << std::setw(2) << std::setfill('0')

<< (int)hash[i];

}

return ss.str();

}

```

【生成Token方法】

```cpp

std::string UserService::generateToken(long userId) {

// 生成格式: sess\_{userId}\_{timestamp}\_{random}

auto now = std::chrono::system\_clock::now();

auto timestamp = std::chrono::system\_clock::to\_time\_t(now);

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(10000, 99999);

int randomNum = dis(gen);

std::stringstream ss;

ss << "sess\_" << userId << "\_" << timestamp << "\_" << randomNum;

return ss.str();

}

```

3.2 ProductService核心方法

【商品列表查询方法】

```cpp

// ProductService.cpp

std::string ProductService::getProductList(

const std::string& category,

int page,

int pageSize) {

try {

// 1. 参数验证

if (page < 1) {

return buildErrorResponse("页码必须大于0", "E20101");

}

if (pageSize < 1 || pageSize > 100) {

return buildErrorResponse("每页数量必须在1-100之间", "E20102");

}

// 2. 构造SQL查询

std::string sql;

std::vector<std::string> params;

if (category == "all" || category.empty()) {

// 查询所有分类

sql = "SELECT product\_id, product\_name, category, price, "

"stock, description, image\_url, created\_at "

"FROM products WHERE status = 'active' "

"ORDER BY created\_at DESC "

"LIMIT ? OFFSET ?";

params = {std::to\_string(pageSize), std::to\_string((page - 1) \* pageSize)};

} else {

// 查询指定分类

sql = "SELECT product\_id, product\_name, category, price, "

"stock, description, image\_url, created\_at "

"FROM products WHERE category = ? AND status = 'active' "

"ORDER BY created\_at DESC "

"LIMIT ? OFFSET ?";

params = {category, std::to\_string(pageSize),

std::to\_string((page - 1) \* pageSize)};

}

// 3. 执行查询

json result = executeQuery(sql, params);

// 4. 查询总数

std::string countSql;

std::vector<std::string> countParams;

if (category == "all" || category.empty()) {

countSql = "SELECT COUNT(\*) as total FROM products WHERE status = 'active'";

} else {

countSql = "SELECT COUNT(\*) as total FROM products "

"WHERE category = ? AND status = 'active'";

countParams = {category};

}

json countResult = executeQuery(countSql, countParams);

int total = countResult["data"][0]["total"].get<int>();

// 5. 构造响应

json response;

response["success"] = true;

response["message"] = "查询成功";

response["data"]["products"] = result["data"];

response["data"]["total"] = total;

response["data"]["page"] = page;

response["data"]["page\_size"] = pageSize;

response["data"]["total\_pages"] = (total + pageSize - 1) / pageSize;

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E20199");

}

}

```

【商品详情查询方法】

```cpp

std::string ProductService::getProductDetail(long productId) {

try {

// 1. 查询商品基本信息

json productResult = executeQuery(

"SELECT product\_id, product\_name, category, price, stock, "

"description, image\_url, created\_at, updated\_at "

"FROM products WHERE product\_id = ? AND status = 'active'",

{productId}

);

if (productResult["data"].empty()) {

return buildErrorResponse("商品不存在", "E20201");

}

// 2. 查询商品评价统计

json reviewStats = executeQuery(

"SELECT COUNT(\*) as review\_count, "

"AVG(rating) as avg\_rating "

"FROM reviews WHERE product\_id = ?",

{productId}

);

// 3. 查询最新评价(前3条)

json recentReviews = executeQuery(

"SELECT r.review\_id, r.rating, r.content, r.created\_at, "

"u.username "

"FROM reviews r "

"JOIN users u ON r.user\_id = u.user\_id "

"WHERE r.product\_id = ? "

"ORDER BY r.created\_at DESC LIMIT 3",

{productId}

);

// 4. 合并数据

json response;

response["success"] = true;

response["message"] = "查询成功";

response["data"]["product"] = productResult["data"][0];

response["data"]["review\_count"] = reviewStats["data"][0]["review\_count"];

response["data"]["avg\_rating"] = reviewStats["data"][0]["avg\_rating"];

response["data"]["recent\_reviews"] = recentReviews["data"];

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E20299");

}

}

```

【商品搜索方法】

```cpp

std::string ProductService::searchProducts(

const std::string& keyword,

int page,

int pageSize) {

try {

// 1. 构造LIKE查询

std::string searchPattern = "%" + keyword + "%";

// 2. 查询商品(匹配名称或描述)

json result = executeQuery(

"SELECT product\_id, product\_name, category, price, "

"stock, description, image\_url, created\_at "

"FROM products "

"WHERE (product\_name LIKE ? OR description LIKE ?) "

"AND status = 'active' "

"ORDER BY created\_at DESC "

"LIMIT ? OFFSET ?",

{searchPattern, searchPattern,

std::to\_string(pageSize),

std::to\_string((page - 1) \* pageSize)}

);

// 3. 查询总数

json countResult = executeQuery(

"SELECT COUNT(\*) as total FROM products "

"WHERE (product\_name LIKE ? OR description LIKE ?) "

"AND status = 'active'",

{searchPattern, searchPattern}

);

int total = countResult["data"][0]["total"].get<int>();

// 4. 构造响应

json response;

response["success"] = true;

response["message"] = "搜索成功";

response["data"]["products"] = result["data"];

response["data"]["total"] = total;

response["data"]["keyword"] = keyword;

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E20399");

}

}

```

3.3 CartService核心方法

【添加购物车方法】

```cpp

// CartService.cpp

std::string CartService::addToCart(

long userId,

long productId,

int quantity) {

try {

// 1. 参数验证

if (quantity <= 0) {

return buildErrorResponse("数量必须大于0", "E30101");

}

// 2. 检查商品是否存在

json productCheck = executeQuery(

"SELECT product\_id, product\_name, price, stock "

"FROM products WHERE product\_id = ? AND status = 'active'",

{productId}

);

if (productCheck["data"].empty()) {

return buildErrorResponse("商品不存在或已下架", "E30102");

}

int stock = productCheck["data"][0]["stock"].get<int>();

// 3. 检查库存

if (quantity > stock) {

return buildErrorResponse("库存不足", "E30103");

}

// 4. 检查购物车中是否已有该商品

json existingCart = executeQuery(

"SELECT cart\_id, quantity FROM cart "

"WHERE user\_id = ? AND product\_id = ?",

{userId, productId}

);

int affectedRows = 0;

if (!existingCart["data"].empty()) {

// 已存在,更新数量

int oldQuantity = existingCart["data"][0]["quantity"].get<int>();

int newQuantity = oldQuantity + quantity;

// 再次检查库存

if (newQuantity > stock) {

return buildErrorResponse("库存不足", "E30103");

}

affectedRows = executeUpdate(

"UPDATE cart SET quantity = ?, updated\_at = NOW() "

"WHERE cart\_id = ?",

{newQuantity, existingCart["data"][0]["cart\_id"].get<long>()}

);

} else {

// 不存在,插入新记录

affectedRows = executeUpdate(

"INSERT INTO cart (user\_id, product\_id, quantity, "

"created\_at, updated\_at) "

"VALUES (?, ?, ?, NOW(), NOW())",

{userId, productId, quantity}

);

}

if (affectedRows == 0) {

return buildErrorResponse("添加失败", "E30104");

}

// 5. 构造成功响应

json response;

response["success"] = true;

response["message"] = "添加成功";

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E30199");

}

}

```

【获取购物车列表方法】

```cpp

std::string CartService::getCartItems(long userId) {

try {

// 联表查询购物车和商品信息

json result = executeQuery(

"SELECT c.cart\_id, c.product\_id, c.quantity, "

"p.product\_name, p.price, p.stock, p.image\_url, "

"c.updated\_at "

"FROM cart c "

"JOIN products p ON c.product\_id = p.product\_id "

"WHERE c.user\_id = ? AND p.status = 'active' "

"ORDER BY c.updated\_at DESC",

{userId}

);

// 计算总价

double totalPrice = 0.0;

for (const auto& item : result["data"]) {

double price = item["price"].get<double>();

int quantity = item["quantity"].get<int>();

totalPrice += price \* quantity;

}

// 构造响应

json response;

response["success"] = true;

response["message"] = "查询成功";

response["data"]["items"] = result["data"];

response["data"]["total\_price"] = totalPrice;

response["data"]["item\_count"] = result["data"].size();

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E30299");

}

}

```

【更新购物车数量方法】

```cpp

std::string CartService::updateCartItem(

long userId,

long cartId,

int quantity) {

try {

// 1. 验证数量

if (quantity <= 0) {

return buildErrorResponse("数量必须大于0", "E30301");

}

// 2. 验证购物车项是否属于该用户

json cartCheck = executeQuery(

"SELECT c.cart\_id, c.product\_id, p.stock "

"FROM cart c "

"JOIN products p ON c.product\_id = p.product\_id "

"WHERE c.cart\_id = ? AND c.user\_id = ?",

{cartId, userId}

);

if (cartCheck["data"].empty()) {

return buildErrorResponse("购物车项不存在", "E30302");

}

int stock = cartCheck["data"][0]["stock"].get<int>();

// 3. 检查库存

if (quantity > stock) {

return buildErrorResponse("库存不足", "E30303");

}

// 4. 更新数量

int affectedRows = executeUpdate(

"UPDATE cart SET quantity = ?, updated\_at = NOW() "

"WHERE cart\_id = ? AND user\_id = ?",

{quantity, cartId, userId}

);

if (affectedRows == 0) {

return buildErrorResponse("更新失败", "E30304");

}

// 5. 构造成功响应

json response;

response["success"] = true;

response["message"] = "更新成功";

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E30399");

}

}

```

3.4 OrderService核心方法

【创建订单方法】

```cpp

// OrderService.cpp

std::string OrderService::createOrder(

long userId,

long addressId,

const std::vector<long>& cartIds,

long couponId,

const std::string& paymentMethod) {

Connection\* conn = nullptr;

try {

// 1. 获取数据库连接

conn = DatabaseManager::getInstance().getConnection();

// 2. 开启事务

conn->setAutoCommit(false);

// 3. 验证收货地址

json addressCheck = executeQuery(

"SELECT address\_id FROM addresses "

"WHERE address\_id = ? AND user\_id = ?",

{addressId, userId}

);

if (addressCheck["data"].empty()) {

conn->rollback();

DatabaseManager::getInstance().releaseConnection(conn);

return buildErrorResponse("收货地址不存在", "E40101");

}

// 4. 获取购物车商品信息

std::string cartIdsStr = joinIds(cartIds);

json cartItems = executeQuery(

"SELECT c.cart\_id, c.product\_id, c.quantity, "

"p.product\_name, p.price, p.stock "

"FROM cart c "

"JOIN products p ON c.product\_id = p.product\_id "

"WHERE c.cart\_id IN (" + cartIdsStr + ") "

"AND c.user\_id = ? AND p.status = 'active'",

{userId}

);

if (cartItems["data"].empty()) {

conn->rollback();

DatabaseManager::getInstance().releaseConnection(conn);

return buildErrorResponse("购物车为空", "E40102");

}

// 5. 计算订单金额并检查库存

double totalAmount = 0.0;

for (const auto& item : cartItems["data"]) {

int quantity = item["quantity"].get<int>();

int stock = item["stock"].get<int>();

double price = item["price"].get<double>();

if (quantity > stock) {

conn->rollback();

DatabaseManager::getInstance().releaseConnection(conn);

std::string msg = "商品[" + item["product\_name"].get<std::string>()

+ "]库存不足";

return buildErrorResponse(msg, "E40103");

}

totalAmount += price \* quantity;

}

// 6. 应用优惠券

double discountAmount = 0.0;

if (couponId > 0) {

json couponResult = applyCoupon(userId, couponId, totalAmount, conn);

if (couponResult["success"].get<bool>()) {

discountAmount = couponResult["discount\_amount"].get<double>();

}

}

double finalAmount = totalAmount - discountAmount;

// 7. 生成订单号

std::string orderNumber = generateOrderNumber();

// 8. 插入订单主表

int orderRows = executeUpdate(

"INSERT INTO orders (order\_number, user\_id, total\_amount, "

"discount\_amount, final\_amount, address\_id, coupon\_id, "

"payment\_method, order\_status, created\_at) "

"VALUES (?, ?, ?, ?, ?, ?, ?, ?, 'pending', NOW())",

{orderNumber, userId, totalAmount, discountAmount, finalAmount,

addressId, couponId, paymentMethod},

conn

);

// 9. 获取订单ID

json orderIdQuery = executeQuery(

"SELECT order\_id FROM orders WHERE order\_number = ?",

{orderNumber},

conn

);

long orderId = orderIdQuery["data"][0]["order\_id"].get<long>();

// 10. 插入订单详情并扣减库存

for (const auto& item : cartItems["data"]) {

long productId = item["product\_id"].get<long>();

int quantity = item["quantity"].get<int>();

double price = item["price"].get<double>();

std::string productName = item["product\_name"];

// 插入订单详情

executeUpdate(

"INSERT INTO order\_items (order\_id, product\_id, product\_name, "

"price, quantity, subtotal) "

"VALUES (?, ?, ?, ?, ?, ?)",

{orderId, productId, productName, price, quantity, price \* quantity},

conn

);

// 扣减库存

int stockRows = executeUpdate(

"UPDATE products SET stock = stock - ? "

"WHERE product\_id = ? AND stock >= ?",

{quantity, productId, quantity},

conn

);

if (stockRows == 0) {

conn->rollback();

DatabaseManager::getInstance().releaseConnection(conn);

return buildErrorResponse("扣减库存失败,库存不足", "E40104");

}

}

// 11. 删除购物车项

executeUpdate(

"DELETE FROM cart WHERE cart\_id IN (" + cartIdsStr + ") AND user\_id = ?",

{userId},

conn

);

// 12. 如果使用了优惠券,更新为已使用

if (couponId > 0) {

executeUpdate(

"UPDATE user\_coupons SET status = 'used', used\_at = NOW() "

"WHERE user\_id = ? AND coupon\_id = ?",

{userId, couponId},

conn

);

}

// 13. 提交事务

conn->commit();

DatabaseManager::getInstance().releaseConnection(conn);

// 14. 构造成功响应

json response;

response["success"] = true;

response["message"] = "订单创建成功";

response["data"]["order\_id"] = orderId;

response["data"]["order\_number"] = orderNumber;

response["data"]["final\_amount"] = finalAmount;

return response.dump();

} catch (const std::exception& e) {

if (conn != nullptr) {

conn->rollback();

DatabaseManager::getInstance().releaseConnection(conn);

}

return handleException(e, "E40199");

}

}

```

【生成订单号方法】

```cpp

std::string OrderService::generateOrderNumber() {

// 格式: yyyyMMddHHmmss + 6位随机数

auto now = std::chrono::system\_clock::now();

std::time\_t now\_c = std::chrono::system\_clock::to\_time\_t(now);

std::tm\* now\_tm = std::localtime(&now\_c);

char timeStr[15];

std::strftime(timeStr, sizeof(timeStr), "%Y%m%d%H%M%S", now\_tm);

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(100000, 999999);

int randomNum = dis(gen);

return std::string(timeStr) + std::to\_string(randomNum);

}

```

（4）核心功能实现

4.1 购物车功能

【购物车状态管理】

购物车采用数据库持久化存储,支持跨会话访问:

表结构:

```sql

CREATE TABLE cart (

cart\_id BIGINT AUTO\_INCREMENT PRIMARY KEY,

user\_id BIGINT NOT NULL,

product\_id BIGINT NOT NULL,

quantity INT NOT NULL DEFAULT 1,

created\_at DATETIME NOT NULL,

updated\_at DATETIME NOT NULL,

FOREIGN KEY (user\_id) REFERENCES users(user\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id),

UNIQUE KEY uk\_user\_product (user\_id, product\_id)

);

```

关键特性:

1. \*\*唯一约束\*\*:同一用户同一商品只能有一条记录

2. \*\*数量累加\*\*:重复添加时累加数量

3. \*\*库存检查\*\*:添加或更新时实时检查库存

4. \*\*自动清理\*\*:创建订单后自动删除对应购物车项

【购物车UI实现】

```cpp

// CartTab.cpp

void CartTab::loadCartItems() {

long userId = context.getUserId();

QString request = QString("GET\_CART %1").arg(userId);

networkClient->sendRequest(request, [this](const QString& response) {

QJsonDocument doc = QJsonDocument::fromJson(response.toUtf8());

QJsonObject obj = doc.object();

if (obj["success"].toBool()) {

QJsonObject data = obj["data"].toObject();

QJsonArray items = data["items"].toArray();

double totalPrice = data["total\_price"].toDouble();

// 清空现有项

clearCartItems();

// 创建购物车项控件

for (const QJsonValue& value : items) {

QJsonObject item = value.toObject();

CartItemWidget\* itemWidget = new CartItemWidget(item, this);

connect(itemWidget, &CartItemWidget::quantityChanged,

this, &CartTab::onQuantityChanged);

connect(itemWidget, &CartItemWidget::removeClicked,

this, &CartTab::onRemoveItem);

connect(itemWidget, &CartItemWidget::checkStateChanged,

this, &CartTab::updateTotalPrice);

cartLayout->addWidget(itemWidget);

cartItemWidgets.append(itemWidget);

}

// 更新总价显示

updateTotalPrice();

}

});

}

void CartTab::onCheckoutClicked() {

// 获取选中的购物车项

QVector<long> selectedCartIds;

for (CartItemWidget\* item : cartItemWidgets) {

if (item->isChecked()) {

selectedCartIds.append(item->getCartId());

}

}

if (selectedCartIds.isEmpty()) {

QMessageBox::warning(this, "提示", "请选择要结算的商品");

return;

}

// 显示结算对话框

CheckoutDialog dialog(selectedCartIds, context, this);

if (dialog.exec() == QDialog::Accepted) {

// 订单创建成功,刷新购物车

loadCartItems();

// 切换到订单页面

emit switchToOrdersTab();

}

}

```

4.2 订单状态机

【订单状态流转】

```

pending(待支付) ──> paid(已支付) ──> shipped(已发货) ──> completed(已完成)

│ │ │

└──> cancelled └──> refunding ──┴──> refunded(已退款)

(已取消) (退款中)

```

状态转换规则:

```cpp

// OrderService.cpp

bool OrderService::canTransitionTo(

const std::string& currentStatus,

const std::string& targetStatus) {

static const std::map<std::string, std::vector<std::string>> transitions = {

{"pending", {"paid", "cancelled"}},

{"paid", {"shipped", "refunding"}},

{"shipped", {"completed", "refunding"}},

{"refunding", {"refunded"}},

{"completed", {}},

{"cancelled", {}},

{"refunded", {}}

};

auto it = transitions.find(currentStatus);

if (it == transitions.end()) {

return false;

}

const auto& allowedTargets = it->second;

return std::find(allowedTargets.begin(), allowedTargets.end(), targetStatus)

!= allowedTargets.end();

}

```

【订单支付流程】

```cpp

std::string OrderService::payOrder(long userId, long orderId) {

try {

// 1. 验证订单

json orderCheck = executeQuery(

"SELECT order\_id, order\_status, final\_amount "

"FROM orders WHERE order\_id = ? AND user\_id = ?",

{orderId, userId}

);

if (orderCheck["data"].empty()) {

return buildErrorResponse("订单不存在", "E40201");

}

std::string status = orderCheck["data"][0]["order\_status"];

if (status != "pending") {

return buildErrorResponse("订单状态不正确", "E40202");

}

// 2. 模拟支付(实际应调用第三方支付接口)

double amount = orderCheck["data"][0]["final\_amount"].get<double>();

bool paymentSuccess = simulatePayment(amount);

if (!paymentSuccess) {

return buildErrorResponse("支付失败", "E40203");

}

// 3. 更新订单状态

int affectedRows = executeUpdate(

"UPDATE orders SET order\_status = 'paid', paid\_at = NOW() "

"WHERE order\_id = ? AND user\_id = ?",

{orderId, userId}

);

if (affectedRows == 0) {

return buildErrorResponse("更新订单状态失败", "E40204");

}

// 4. 构造成功响应

json response;

response["success"] = true;

response["message"] = "支付成功";

response["data"]["order\_id"] = orderId;

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E40299");

}

}

```

4.3 优惠券系统

【优惠券应用逻辑】

```cpp

// CouponService.cpp

json CouponService::applyCoupon(

long userId,

long couponId,

double orderAmount,

Connection\* conn) {

json result;

try {

// 1. 查询优惠券信息

json couponQuery = executeQuery(

"SELECT c.coupon\_id, c.discount\_type, c.discount\_value, "

"c.min\_purchase\_amount, c.max\_discount\_amount, c.valid\_until, "

"uc.status "

"FROM coupons c "

"JOIN user\_coupons uc ON c.coupon\_id = uc.coupon\_id "

"WHERE c.coupon\_id = ? AND uc.user\_id = ? AND uc.status = 'unused'",

{couponId, userId},

conn

);

if (couponQuery["data"].empty()) {

result["success"] = false;

result["message"] = "优惠券不可用";

return result;

}

auto coupon = couponQuery["data"][0];

// 2. 检查是否过期

std::string validUntil = coupon["valid\_until"];

if (isExpired(validUntil)) {

result["success"] = false;

result["message"] = "优惠券已过期";

return result;

}

// 3. 检查最低消费金额

double minAmount = coupon["min\_purchase\_amount"].get<double>();

if (orderAmount < minAmount) {

result["success"] = false;

result["message"] = "未达到最低消费金额";

result["min\_amount"] = minAmount;

return result;

}

// 4. 计算折扣金额

std::string discountType = coupon["discount\_type"];

double discountValue = coupon["discount\_value"].get<double>();

double maxDiscount = coupon["max\_discount\_amount"].get<double>();

double discountAmount = 0.0;

if (discountType == "percentage") {

// 百分比折扣

discountAmount = orderAmount \* discountValue / 100.0;

} else if (discountType == "fixed") {

// 固定金额折扣

discountAmount = discountValue;

}

// 限制最大折扣金额

if (maxDiscount > 0 && discountAmount > maxDiscount) {

discountAmount = maxDiscount;

}

// 确保折扣不超过订单金额

if (discountAmount > orderAmount) {

discountAmount = orderAmount;

}

result["success"] = true;

result["discount\_amount"] = discountAmount;

result["message"] = "优惠券应用成功";

return result;

} catch (const std::exception& e) {

result["success"] = false;

result["message"] = e.what();

return result;

}

}

```

【优惠券领取功能】

```cpp

std::string CouponService::claimCoupon(long userId, long couponId) {

try {

// 1. 检查优惠券是否存在

json couponCheck = executeQuery(

"SELECT coupon\_id, coupon\_name, total\_quantity, "

"claimed\_quantity, valid\_until "

"FROM coupons WHERE coupon\_id = ?",

{couponId}

);

if (couponCheck["data"].empty()) {

return buildErrorResponse("优惠券不存在", "E50101");

}

auto coupon = couponCheck["data"][0];

// 2. 检查是否已领取

json claimedCheck = executeQuery(

"SELECT user\_coupon\_id FROM user\_coupons "

"WHERE user\_id = ? AND coupon\_id = ?",

{userId, couponId}

);

if (!claimedCheck["data"].empty()) {

return buildErrorResponse("您已领取过该优惠券", "E50102");

}

// 3. 检查库存

int total = coupon["total\_quantity"].get<int>();

int claimed = coupon["claimed\_quantity"].get<int>();

if (claimed >= total) {

return buildErrorResponse("优惠券已领完", "E50103");

}

// 4. 插入用户优惠券记录

int insertRows = executeUpdate(

"INSERT INTO user\_coupons (user\_id, coupon\_id, status, claimed\_at) "

"VALUES (?, ?, 'unused', NOW())",

{userId, couponId}

);

// 5. 更新已领取数量

executeUpdate(

"UPDATE coupons SET claimed\_quantity = claimed\_quantity + 1 "

"WHERE coupon\_id = ?",

{couponId}

);

// 6. 构造成功响应

json response;

response["success"] = true;

response["message"] = "领取成功";

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E50199");

}

}

```

4.4 库存并发控制

【乐观锁实现】

使用MySQL的版本号机制实现乐观锁:

```sql

-- 添加版本号字段

ALTER TABLE products ADD COLUMN version INT NOT NULL DEFAULT 0;

-- 更新时检查版本号

UPDATE products

SET stock = stock - ?, version = version + 1

WHERE product\_id = ? AND stock >= ? AND version = ?;

```

C++实现:

```cpp

std::string ProductService::decreaseStock(

long productId,

int quantity,

Connection\* conn) {

const int MAX\_RETRY = 3;

int retryCount = 0;

while (retryCount < MAX\_RETRY) {

try {

// 1. 查询当前库存和版本号

json stockQuery = executeQuery(

"SELECT stock, version FROM products WHERE product\_id = ?",

{productId},

conn

);

if (stockQuery["data"].empty()) {

return buildErrorResponse("商品不存在", "E20301");

}

int stock = stockQuery["data"][0]["stock"].get<int>();

int version = stockQuery["data"][0]["version"].get<int>();

// 2. 检查库存

if (stock < quantity) {

return buildErrorResponse("库存不足", "E20302");

}

// 3. 扣减库存(乐观锁)

int affectedRows = executeUpdate(

"UPDATE products SET stock = stock - ?, version = version + 1 "

"WHERE product\_id = ? AND stock >= ? AND version = ?",

{quantity, productId, quantity, version},

conn

);

if (affectedRows > 0) {

// 成功

json response;

response["success"] = true;

response["message"] = "扣减库存成功";

return response.dump();

} else {

// 版本号冲突,重试

retryCount++;

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

}

} catch (const std::exception& e) {

return handleException(e, "E20399");

}

}

return buildErrorResponse("扣减库存失败,请重试", "E20303");

}

```

【悲观锁实现】

使用SELECT FOR UPDATE实现悲观锁:

```cpp

std::string ProductService::decreaseStockWithLock(

long productId,

int quantity,

Connection\* conn) {

try {

// 1. 加锁查询(悲观锁)

json lockQuery = executeQuery(

"SELECT stock FROM products WHERE product\_id = ? FOR UPDATE",

{productId},

conn

);

if (lockQuery["data"].empty()) {

return buildErrorResponse("商品不存在", "E20301");

}

int stock = lockQuery["data"][0]["stock"].get<int>();

// 2. 检查库存

if (stock < quantity) {

return buildErrorResponse("库存不足", "E20302");

}

// 3. 扣减库存

int affectedRows = executeUpdate(

"UPDATE products SET stock = stock - ? WHERE product\_id = ?",

{quantity, productId},

conn

);

if (affectedRows == 0) {

return buildErrorResponse("扣减库存失败", "E20303");

}

json response;

response["success"] = true;

response["message"] = "扣减库存成功";

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E20399");

}

}

四、相关技术讨论

四、相关技术讨论

（1）持久层

1.1 数据库连接管理

【DatabaseManager单例模式】

DatabaseManager采用线程安全的单例模式管理数据库连接池:

```cpp

// DatabaseManager.h

class DatabaseManager {

private:

static DatabaseManager\* instance;

static std::mutex instanceMutex;

sql::Driver\* driver;

std::string host;

std::string user;

std::string password;

std::string database;

std::queue<sql::Connection\*> connectionPool;

std::mutex poolMutex;

int poolSize;

int maxPoolSize;

// 私有构造函数

DatabaseManager();

public:

// 禁止拷贝和赋值

DatabaseManager(const DatabaseManager&) = delete;

DatabaseManager& operator=(const DatabaseManager&) = delete;

// 获取单例实例

static DatabaseManager& getInstance();

// 连接池管理

sql::Connection\* getConnection();

void releaseConnection(sql::Connection\* conn);

void closeAllConnections();

// 配置管理

void initialize(const std::string& host, const std::string& user,

const std::string& password, const std::string& database);

};

```

实现:

```cpp

// DatabaseManager.cpp

DatabaseManager\* DatabaseManager::instance = nullptr;

std::mutex DatabaseManager::instanceMutex;

DatabaseManager& DatabaseManager::getInstance() {

if (instance == nullptr) {

std::lock\_guard<std::mutex> lock(instanceMutex);

if (instance == nullptr) { // Double-checked locking

instance = new DatabaseManager();

}

}

return \*instance;

}

DatabaseManager::DatabaseManager()

: driver(nullptr), poolSize(0), maxPoolSize(10) {

driver = get\_driver\_instance();

}

void DatabaseManager::initialize(const std::string& host,

const std::string& user,

const std::string& password,

const std::string& database) {

this->host = host;

this->user = user;

this->password = password;

this->database = database;

// 预创建5个连接

for (int i = 0; i < 5; i++) {

try {

sql::Connection\* conn = driver->connect(host, user, password);

conn->setSchema(database);

conn->setClientOption("charset", "utf8mb4");

std::lock\_guard<std::mutex> lock(poolMutex);

connectionPool.push(conn);

poolSize++;

} catch (sql::SQLException& e) {

std::cerr << "Failed to create connection: " << e.what() << std::endl;

}

}

}

sql::Connection\* DatabaseManager::getConnection() {

std::unique\_lock<std::mutex> lock(poolMutex);

// 如果池中有可用连接,直接返回

if (!connectionPool.empty()) {

sql::Connection\* conn = connectionPool.front();

connectionPool.pop();

poolSize--;

// 检查连接是否有效

if (!conn->isValid()) {

delete conn;

// 创建新连接

conn = driver->connect(host, user, password);

conn->setSchema(database);

conn->setClientOption("charset", "utf8mb4");

}

return conn;

}

// 池中无可用连接,且未达到最大连接数,创建新连接

if (poolSize < maxPoolSize) {

sql::Connection\* conn = driver->connect(host, user, password);

conn->setSchema(database);

conn->setClientOption("charset", "utf8mb4");

poolSize++;

return conn;

}

// 等待其他线程释放连接(最多等待5秒)

auto waitStart = std::chrono::steady\_clock::now();

while (connectionPool.empty()) {

lock.unlock();

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

lock.lock();

auto elapsed = std::chrono::steady\_clock::now() - waitStart;

if (elapsed > std::chrono::seconds(5)) {

throw std::runtime\_error("获取数据库连接超时");

}

}

sql::Connection\* conn = connectionPool.front();

connectionPool.pop();

poolSize--;

return conn;

}

void DatabaseManager::releaseConnection(sql::Connection\* conn) {

if (conn == nullptr) return;

std::lock\_guard<std::mutex> lock(poolMutex);

// 检查连接是否有效

if (conn->isValid()) {

connectionPool.push(conn);

poolSize++;

} else {

delete conn;

}

}

void DatabaseManager::closeAllConnections() {

std::lock\_guard<std::mutex> lock(poolMutex);

while (!connectionPool.empty()) {

sql::Connection\* conn = connectionPool.front();

connectionPool.pop();

delete conn;

}

poolSize = 0;

}

```

关键特性:

1. \*\*双检锁单例\*\*:线程安全的延迟初始化

2. \*\*连接池\*\*:复用连接,减少创建开销

3. \*\*连接验证\*\*:获取连接时检查有效性

4. \*\*超时机制\*\*:等待连接超时则抛出异常

5. \*\*自动释放\*\*:使用RAII模式确保连接释放

1.2 RAII资源管理

【ConnectionGuard RAII包装器】

```cpp

// ConnectionGuard.h

class ConnectionGuard {

private:

sql::Connection\* connection;

bool released;

public:

explicit ConnectionGuard()

: connection(DatabaseManager::getInstance().getConnection()),

released(false) {}

~ConnectionGuard() {

if (!released && connection != nullptr) {

DatabaseManager::getInstance().releaseConnection(connection);

}

}

// 禁止拷贝

ConnectionGuard(const ConnectionGuard&) = delete;

ConnectionGuard& operator=(const ConnectionGuard&) = delete;

// 移动构造

ConnectionGuard(ConnectionGuard&& other) noexcept

: connection(other.connection), released(other.released) {

other.connection = nullptr;

other.released = true;

}

sql::Connection\* get() const { return connection; }

void release() {

if (!released) {

DatabaseManager::getInstance().releaseConnection(connection);

released = true;

}

}

};

```

使用示例:

```cpp

std::string UserService::getUserInfo(long userId) {

try {

// RAII自动管理连接生命周期

ConnectionGuard guard;

sql::Connection\* conn = guard.get();

std::unique\_ptr<sql::PreparedStatement> pstmt(

conn->prepareStatement("SELECT \* FROM users WHERE user\_id = ?")

);

pstmt->setInt64(1, userId);

std::unique\_ptr<sql::ResultSet> res(pstmt->executeQuery());

// ... 处理结果

// guard析构时自动释放连接

return result.dump();

} catch (const std::exception& e) {

// 异常情况下也会自动释放连接

return handleException(e, "E10001");

}

}

```

1.3 预编译语句

【PreparedStatement使用】

使用预编译语句防止SQL注入并提升性能:

```cpp

json BaseService::executeQuery(

const std::string& sql,

const std::vector<std::string>& params,

sql::Connection\* conn) {

bool useExternalConn = (conn != nullptr);

if (!useExternalConn) {

conn = DatabaseManager::getInstance().getConnection();

}

try {

// 创建预编译语句

std::unique\_ptr<sql::PreparedStatement> pstmt(

conn->prepareStatement(sql)

);

// 绑定参数

for (size\_t i = 0; i < params.size(); i++) {

pstmt->setString(i + 1, params[i]);

}

// 执行查询

std::unique\_ptr<sql::ResultSet> res(pstmt->executeQuery());

// 获取元数据

sql::ResultSetMetaData\* meta = res->getMetaData();

int columnCount = meta->getColumnCount();

// 构造JSON结果

json result;

result["success"] = true;

result["data"] = json::array();

while (res->next()) {

json row;

for (int i = 1; i <= columnCount; i++) {

std::string columnName = meta->getColumnName(i);

if (res->isNull(i)) {

row[columnName] = nullptr;

} else {

std::string value = res->getString(i);

row[columnName] = value;

}

}

result["data"].push\_back(row);

}

if (!useExternalConn) {

DatabaseManager::getInstance().releaseConnection(conn);

}

return result;

} catch (sql::SQLException& e) {

if (!useExternalConn) {

DatabaseManager::getInstance().releaseConnection(conn);

}

json error;

error["success"] = false;

error["message"] = e.what();

error["sql\_state"] = e.getSQLState();

error["error\_code"] = e.getErrorCode();

throw std::runtime\_error(error.dump());

}

}

```

预编译语句的优势:

1. \*\*防止SQL注入\*\*:参数自动转义

2. \*\*性能优化\*\*:语句缓存,执行计划复用

3. \*\*类型安全\*\*:明确的参数类型绑定

4. \*\*代码清晰\*\*:SQL和数据分离

（2）SQL技术

2.1 复杂查询

【多表联查示例1:订单详情查询】

```sql

-- 查询订单完整信息(包含用户、地址、商品、优惠券)

SELECT

o.order\_id,

o.order\_number,

o.total\_amount,

o.discount\_amount,

o.final\_amount,

o.order\_status,

o.created\_at,

o.paid\_at,

o.shipped\_at,

u.user\_id,

u.username,

u.phone,

a.address\_id,

a.receiver\_name,

a.receiver\_phone,

a.province,

a.city,

a.district,

a.detail\_address,

c.coupon\_id,

c.coupon\_name,

c.discount\_type,

c.discount\_value,

GROUP\_CONCAT(

CONCAT\_WS('|',

oi.product\_id,

oi.product\_name,

oi.price,

oi.quantity,

oi.subtotal

) SEPARATOR ';;'

) AS order\_items

FROM orders o

LEFT JOIN users u ON o.user\_id = u.user\_id

LEFT JOIN addresses a ON o.address\_id = a.address\_id

LEFT JOIN coupons c ON o.coupon\_id = c.coupon\_id

LEFT JOIN order\_items oi ON o.order\_id = oi.order\_id

WHERE o.order\_id = ?

GROUP BY o.order\_id;

```

C++实现:

```cpp

std::string OrderService::getOrderDetail(long userId, long orderId) {

try {

json result = executeQuery(

"SELECT o.order\_id, o.order\_number, o.total\_amount, "

"o.discount\_amount, o.final\_amount, o.order\_status, "

"o.created\_at, o.paid\_at, o.shipped\_at, "

"u.username, u.phone, "

"a.receiver\_name, a.receiver\_phone, a.province, a.city, "

"a.district, a.detail\_address, "

"c.coupon\_name, c.discount\_type, c.discount\_value "

"FROM orders o "

"LEFT JOIN users u ON o.user\_id = u.user\_id "

"LEFT JOIN addresses a ON o.address\_id = a.address\_id "

"LEFT JOIN coupons c ON o.coupon\_id = c.coupon\_id "

"WHERE o.order\_id = ? AND o.user\_id = ?",

{orderId, userId}

);

if (result["data"].empty()) {

return buildErrorResponse("订单不存在", "E40301");

}

// 查询订单商品

json itemsResult = executeQuery(

"SELECT product\_id, product\_name, price, quantity, subtotal "

"FROM order\_items WHERE order\_id = ?",

{orderId}

);

// 合并数据

json response;

response["success"] = true;

response["message"] = "查询成功";

response["data"]["order"] = result["data"][0];

response["data"]["items"] = itemsResult["data"];

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E40399");

}

}

```

【多表联查示例2:商品评价统计】

```sql

-- 查询商品评价统计(包含评分分布、用户评价)

SELECT

p.product\_id,

p.product\_name,

p.price,

p.stock,

COUNT(DISTINCT r.review\_id) AS total\_reviews,

AVG(r.rating) AS avg\_rating,

SUM(CASE WHEN r.rating = 5 THEN 1 ELSE 0 END) AS five\_star\_count,

SUM(CASE WHEN r.rating = 4 THEN 1 ELSE 0 END) AS four\_star\_count,

SUM(CASE WHEN r.rating = 3 THEN 1 ELSE 0 END) AS three\_star\_count,

SUM(CASE WHEN r.rating = 2 THEN 1 ELSE 0 END) AS two\_star\_count,

SUM(CASE WHEN r.rating = 1 THEN 1 ELSE 0 END) AS one\_star\_count,

COUNT(DISTINCT o.order\_id) AS total\_orders

FROM products p

LEFT JOIN reviews r ON p.product\_id = r.product\_id

LEFT JOIN order\_items oi ON p.product\_id = oi.product\_id

LEFT JOIN orders o ON oi.order\_id = o.order\_id AND o.order\_status = 'completed'

WHERE p.product\_id = ?

GROUP BY p.product\_id;

```

【子查询示例:用户订单统计】

```sql

-- 查询用户订单统计信息

SELECT

u.user\_id,

u.username,

u.phone,

(SELECT COUNT(\*) FROM orders WHERE user\_id = u.user\_id) AS total\_orders,

(SELECT COUNT(\*) FROM orders

WHERE user\_id = u.user\_id AND order\_status = 'pending') AS pending\_orders,

(SELECT COUNT(\*) FROM orders

WHERE user\_id = u.user\_id AND order\_status = 'paid') AS paid\_orders,

(SELECT COALESCE(SUM(final\_amount), 0) FROM orders

WHERE user\_id = u.user\_id AND order\_status IN ('paid', 'shipped', 'completed'))

AS total\_spending,

(SELECT COUNT(\*) FROM user\_coupons

WHERE user\_id = u.user\_id AND status = 'unused') AS available\_coupons

FROM users u

WHERE u.user\_id = ?;

```

2.2 存储过程

【订单自动取消存储过程】

```sql

DELIMITER $$

CREATE PROCEDURE auto\_cancel\_unpaid\_orders()

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE v\_order\_id BIGINT;

DECLARE v\_product\_id BIGINT;

DECLARE v\_quantity INT;

-- 声明游标

DECLARE order\_cursor CURSOR FOR

SELECT order\_id FROM orders

WHERE order\_status = 'pending'

AND created\_at < DATE\_SUB(NOW(), INTERVAL 30 MINUTE);

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- 开启事务

START TRANSACTION;

OPEN order\_cursor;

read\_loop: LOOP

FETCH order\_cursor INTO v\_order\_id;

IF done THEN

LEAVE read\_loop;

END IF;

-- 1. 更新订单状态为已取消

UPDATE orders SET order\_status = 'cancelled' WHERE order\_id = v\_order\_id;

-- 2. 恢复商品库存

UPDATE products p

INNER JOIN order\_items oi ON p.product\_id = oi.product\_id

SET p.stock = p.stock + oi.quantity

WHERE oi.order\_id = v\_order\_id;

-- 3. 如果使用了优惠券,恢复优惠券状态

UPDATE user\_coupons uc

INNER JOIN orders o ON uc.user\_id = o.user\_id AND uc.coupon\_id = o.coupon\_id

SET uc.status = 'unused', uc.used\_at = NULL

WHERE o.order\_id = v\_order\_id AND o.coupon\_id IS NOT NULL;

END LOOP;

CLOSE order\_cursor;

-- 提交事务

COMMIT;

SELECT CONCAT('自动取消了 ', ROW\_COUNT(), ' 个超时未支付订单') AS result;

END$$

DELIMITER ;

```

调用存储过程:

```cpp

std::string OrderService::autoCancelUnpaidOrders() {

try {

ConnectionGuard guard;

sql::Connection\* conn = guard.get();

std::unique\_ptr<sql::Statement> stmt(conn->createStatement());

std::unique\_ptr<sql::ResultSet> res(

stmt->executeQuery("CALL auto\_cancel\_unpaid\_orders()")

);

std::string result;

if (res->next()) {

result = res->getString("result");

}

json response;

response["success"] = true;

response["message"] = result;

return response.dump();

} catch (const std::exception& e) {

return handleException(e, "E40499");

}

}

```

【优惠券过期检查存储过程】

```sql

DELIMITER $$

CREATE PROCEDURE expire\_coupons()

BEGIN

-- 更新已过期的用户优惠券状态

UPDATE user\_coupons uc

INNER JOIN coupons c ON uc.coupon\_id = c.coupon\_id

SET uc.status = 'expired'

WHERE uc.status = 'unused'

AND c.valid\_until < NOW();

SELECT CONCAT('过期了 ', ROW\_COUNT(), ' 张优惠券') AS result;

END$$

DELIMITER ;

```

2.3 触发器

【订单状态变更触发器】

```sql

-- 订单状态变更时记录到审计表

DELIMITER $$

CREATE TRIGGER trg\_orders\_status\_change

AFTER UPDATE ON orders

FOR EACH ROW

BEGIN

IF OLD.order\_status != NEW.order\_status THEN

INSERT INTO order\_status\_audit (

order\_id,

old\_status,

new\_status,

changed\_at

) VALUES (

NEW.order\_id,

OLD.order\_status,

NEW.order\_status,

NOW()

);

END IF;

END$$

DELIMITER ;

```

审计表定义:

```sql

CREATE TABLE order\_status\_audit (

audit\_id BIGINT AUTO\_INCREMENT PRIMARY KEY,

order\_id BIGINT NOT NULL,

old\_status VARCHAR(20) NOT NULL,

new\_status VARCHAR(20) NOT NULL,

changed\_at DATETIME NOT NULL,

FOREIGN KEY (order\_id) REFERENCES orders(order\_id),

INDEX idx\_order\_id (order\_id),

INDEX idx\_changed\_at (changed\_at)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

```

【库存变更触发器】

```sql

-- 商品库存变更时记录

DELIMITER $$

CREATE TRIGGER trg\_products\_stock\_change

AFTER UPDATE ON products

FOR EACH ROW

BEGIN

IF OLD.stock != NEW.stock THEN

INSERT INTO stock\_change\_log (

product\_id,

old\_stock,

new\_stock,

change\_amount,

changed\_at

) VALUES (

NEW.product\_id,

OLD.stock,

NEW.stock,

NEW.stock - OLD.stock,

NOW()

);

END IF;

END$$

DELIMITER ;

```

2.4 索引优化

【单列索引】

```sql

-- 用户表索引

CREATE INDEX idx\_username ON users(username);

CREATE INDEX idx\_phone ON users(phone);

CREATE INDEX idx\_email ON users(email);

CREATE INDEX idx\_status ON users(status);

-- 商品表索引

CREATE INDEX idx\_category ON products(category);

CREATE INDEX idx\_status ON products(status);

CREATE INDEX idx\_created\_at ON products(created\_at);

-- 订单表索引

CREATE INDEX idx\_user\_id ON orders(user\_id);

CREATE INDEX idx\_order\_number ON orders(order\_number);

CREATE INDEX idx\_order\_status ON orders(order\_status);

CREATE INDEX idx\_created\_at ON orders(created\_at);

```

【复合索引】

```sql

-- 购物车复合索引

CREATE UNIQUE INDEX uk\_user\_product ON cart(user\_id, product\_id);

-- 订单商品复合索引

CREATE INDEX idx\_order\_product ON order\_items(order\_id, product\_id);

-- 用户优惠券复合索引

CREATE INDEX idx\_user\_coupon\_status ON user\_coupons(user\_id, coupon\_id, status);

-- 订单状态和时间复合索引

CREATE INDEX idx\_status\_created ON orders(order\_status, created\_at);

```

【覆盖索引】

```sql

-- 覆盖索引(无需回表查询)

CREATE INDEX idx\_product\_list ON products(category, status, product\_id, product\_name, price, stock);

-- 查询只访问索引

EXPLAIN SELECT product\_id, product\_name, price, stock

FROM products

WHERE category = 'electronics' AND status = 'active';

```

查询计划分析:

```

+----+-------------+----------+-------+-------------------+------------------+---------+-------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+----------+-------+-------------------+------------------+---------+-------+------+-------------+

| 1 | SIMPLE | products | ref | idx\_product\_list | idx\_product\_list | 206 | const | 100 | Using index |

+----+-------------+----------+-------+-------------------+------------------+---------+-------+------+-------------+

```

（3）多线程技术

3.1 Netty线程模型

【Reactor模式】

Netty采用主从Reactor多线程模型:

```

[Boss线程组(acceptor)]

|

接受客户端连接(ServerSocketChannel)

|

v

+------------------+-------------------+

| | |

[Worker线程1] [Worker线程2] [Worker线程3]

| | |

处理I/O读写 处理I/O读写 处理I/O读写

执行业务逻辑 执行业务逻辑 执行业务逻辑

```

Netty服务器配置:

```java

public class EmshopNettyServer {

private final int port;

// Boss线程组:处理客户端连接

private EventLoopGroup bossGroup;

// Worker线程组:处理I/O操作

private EventLoopGroup workerGroup;

public EmshopNettyServer(int port) {

this.port = port;

// 通常Boss线程数为1,Worker线程数为CPU核心数×2

this.bossGroup = new NioEventLoopGroup(1);

this.workerGroup = new NioEventLoopGroup();

}

public void start() throws InterruptedException {

try {

ServerBootstrap bootstrap = new ServerBootstrap();

bootstrap.group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 128)

.childOption(ChannelOption.SO\_KEEPALIVE, true)

.childOption(ChannelOption.TCP\_NODELAY, true)

.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel ch) {

ChannelPipeline pipeline = ch.pipeline();

// 添加处理器

pipeline.addLast(new LineBasedFrameDecoder(8192));

pipeline.addLast(new StringDecoder(StandardCharsets.UTF\_8));

pipeline.addLast(new StringEncoder(StandardCharsets.UTF\_8));

pipeline.addLast(new EmshopServerHandler());

}

});

// 绑定端口并启动服务器

ChannelFuture future = bootstrap.bind(port).sync();

logger.info("Emshop server started on port {}", port);

// 等待服务器socket关闭

future.channel().closeFuture().sync();

} finally {

shutdown();

}

}

public void shutdown() {

bossGroup.shutdownGracefully();

workerGroup.shutdownGracefully();

}

}

```

【ChannelHandler线程安全】

```java

@ChannelHandler.Sharable // 标记为可共享的Handler

public class EmshopServerHandler extends SimpleChannelInboundHandler<String> {

private static final Logger logger = LoggerFactory.getLogger(EmshopServerHandler.class);

// 线程安全的会话存储(ConcurrentHashMap)

private static final ConcurrentHashMap<ChannelId, UserSession> userSessions

= new ConcurrentHashMap<>();

@Override

protected void channelRead0(ChannelHandlerContext ctx, String message) {

// 每个连接的读取操作在Worker线程中串行执行

// 同一个Channel的所有I/O操作在同一个EventLoop线程中执行

logger.debug("Received message from {}: {}", ctx.channel().remoteAddress(), message);

// 处理请求(可能调用JNI,访问数据库)

String response = processRequest(ctx, message);

// 发送响应

ctx.writeAndFlush(response + "\n");

}

@Override

public void channelActive(ChannelHandlerContext ctx) {

logger.info("Client connected: {}", ctx.channel().remoteAddress());

}

@Override

public void channelInactive(ChannelHandlerContext ctx) {

logger.info("Client disconnected: {}", ctx.channel().remoteAddress());

// 移除会话

userSessions.remove(ctx.channel().id());

}

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

logger.error("Exception in channel {}", ctx.channel().remoteAddress(), cause);

ctx.close();

}

}

```

Netty线程模型特点:

1. \*\*单线程单Channel\*\*:每个Channel的所有I/O操作在同一个EventLoop线程中执行

2. \*\*无锁设计\*\*:避免了多线程同步的开销

3. \*\*异步非阻塞\*\*:高效处理大量并发连接

4. \*\*线程池复用\*\*:减少线程创建销毁的开销

3.2 Qt多线程

【QThread基本使用】

```cpp

// NetworkWorker.h

class NetworkWorker : public QObject {

Q\_OBJECT

public:

explicit NetworkWorker(QObject\* parent = nullptr);

public slots:

void sendRequest(const QString& request);

signals:

void responseReceived(const QString& response);

void errorOccurred(const QString& error);

private:

QTcpSocket\* socket;

};

// NetworkClient.h

class NetworkClient : public QObject {

Q\_OBJECT

public:

explicit NetworkClient(QObject\* parent = nullptr);

~NetworkClient();

void connectToServer(const QString& host, quint16 port);

void sendRequest(const QString& request);

signals:

void connected();

void disconnected();

void responseReceived(const QString& response);

private:

QThread\* workerThread;

NetworkWorker\* worker;

};

// NetworkClient.cpp

NetworkClient::NetworkClient(QObject\* parent)

: QObject(parent) {

// 创建工作线程

workerThread = new QThread(this);

// 创建工作对象

worker = new NetworkWorker();

worker->moveToThread(workerThread);

// 连接信号和槽

connect(workerThread, &QThread::finished,

worker, &QObject::deleteLater);

connect(this, &NetworkClient::requestSent,

worker, &NetworkWorker::sendRequest);

connect(worker, &NetworkWorker::responseReceived,

this, &NetworkClient::responseReceived);

// 启动线程

workerThread->start();

}

NetworkClient::~NetworkClient() {

// 停止线程

workerThread->quit();

workerThread->wait();

}

void NetworkClient::sendRequest(const QString& request) {

emit requestSent(request);

}

```

【线程安全的信号槽】

Qt的信号槽机制自动处理跨线程通信:

```cpp

// 在主线程(UI线程)

void MainWindow::onLoginClicked() {

QString username = ui->lineEdit->text();

// 发送请求(跨线程到Worker线程)

networkClient->sendRequest("LOGIN " + username);

}

// 在Worker线程接收响应

void NetworkWorker::onDataReceived() {

QString response = socket->readAll();

// 发送信号(跨线程回到主线程)

emit responseReceived(response);

}

// 回到主线程更新UI

void MainWindow::onResponseReceived(const QString& response) {

// 安全地更新UI控件

ui->textBrowser->append(response);

}

```

连接类型:

```cpp

// 自动连接(推荐,Qt自动判断)

connect(worker, &Worker::finished, this, &MainWindow::onFinished);

// 直接连接(同一线程)

connect(worker, &Worker::finished, this, &MainWindow::onFinished, Qt::DirectConnection);

// 队列连接(跨线程)

connect(worker, &Worker::finished, this, &MainWindow::onFinished, Qt::QueuedConnection);

// 阻塞队列连接(跨线程,阻塞等待)

connect(worker, &Worker::finished, this, &MainWindow::onFinished, Qt::BlockingQueuedConnection);

```

3.3 线程同步

【C++ std::mutex】

```cpp

class ThreadSafeCache {

private:

std::unordered\_map<long, Product> cache;

mutable std::mutex cacheMutex;

public:

void put(long productId, const Product& product) {

std::lock\_guard<std::mutex> lock(cacheMutex);

cache[productId] = product;

}

bool get(long productId, Product& product) const {

std::lock\_guard<std::mutex> lock(cacheMutex);

auto it = cache.find(productId);

if (it != cache.end()) {

product = it->second;

return true;

}

return false;

}

void clear() {

std::lock\_guard<std::mutex> lock(cacheMutex);

cache.clear();

}

};

```

【读写锁(std::shared\_mutex)】

```cpp

class ProductCache {

private:

std::unordered\_map<long, Product> cache;

mutable std::shared\_mutex cacheMutex;

public:

// 读操作(多个线程可以同时读)

bool get(long productId, Product& product) const {

std::shared\_lock<std::shared\_mutex> lock(cacheMutex);

auto it = cache.find(productId);

if (it != cache.end()) {

product = it->second;

return true;

}

return false;

}

// 写操作(独占锁)

void put(long productId, const Product& product) {

std::unique\_lock<std::shared\_mutex> lock(cacheMutex);

cache[productId] = product;

}

// 写操作(独占锁)

void remove(long productId) {

std::unique\_lock<std::shared\_mutex> lock(cacheMutex);

cache.erase(productId);

}

};

```

【Java ConcurrentHashMap】

```java

public class SessionManager {

// 线程安全的HashMap

private final ConcurrentHashMap<ChannelId, UserSession> sessions

= new ConcurrentHashMap<>();

public void addSession(ChannelId channelId, UserSession session) {

sessions.put(channelId, session);

}

public UserSession getSession(ChannelId channelId) {

return sessions.get(channelId);

}

public void removeSession(ChannelId channelId) {

sessions.remove(channelId);

}

public int getActiveSessionCount() {

return sessions.size();

}

public void forEachSession(BiConsumer<ChannelId, UserSession> action) {

sessions.forEach(action);

}

}

```

（4）容器技术

4.1 STL容器

【std::vector动态数组】

```cpp

// 商品列表

std::vector<Product> products;

// 添加元素

products.push\_back(Product{1, "iPhone 15", 5999.0});

products.emplace\_back(2, "MacBook Pro", 12999.0);

// 预分配空间

products.reserve(100);

// 遍历

for (const auto& product : products) {

std::cout << product.name << ": " << product.price << std::endl;

}

// 排序

std::sort(products.begin(), products.end(),

[](const Product& a, const Product& b) {

return a.price < b.price;

});

// 查找

auto it = std::find\_if(products.begin(), products.end(),

[](const Product& p) {

return p.id == 1;

});

```

【std::map有序映射】

```cpp

// 商品分类统计

std::map<std::string, int> categoryCount;

categoryCount["electronics"] = 150;

categoryCount["books"] = 200;

categoryCount["clothing"] = 300;

// 遍历(按key排序)

for (const auto& [category, count] : categoryCount) {

std::cout << category << ": " << count << std::endl;

}

// 查找

if (categoryCount.find("electronics") != categoryCount.end()) {

int count = categoryCount["electronics"];

}

```

【std::unordered\_map哈希映射】

```cpp

// 用户会话缓存

std::unordered\_map<long, UserSession> sessionCache;

// 插入

sessionCache[1001] = UserSession{1001, "admin", "administrator"};

sessionCache.emplace(1002, UserSession{1002, "user1", "customer"});

// 查找(O(1)平均复杂度)

auto it = sessionCache.find(1001);

if (it != sessionCache.end()) {

UserSession& session = it->second;

}

// 删除

sessionCache.erase(1001);

// 清空

sessionCache.clear();

```

【std::set有序集合】

```cpp

// 商品ID集合

std::set<long> favoriteProducts;

// 插入

favoriteProducts.insert(1);

favoriteProducts.insert(5);

favoriteProducts.insert(3);

// 遍历(自动排序)

for (long productId : favoriteProducts) {

std::cout << productId << std::endl; // 输出:1 3 5

}

// 查找

if (favoriteProducts.count(5) > 0) {

std::cout << "Product 5 is in favorites" << std::endl;

}

```

【std::queue队列】

```cpp

// 订单处理队列

std::queue<Order> orderQueue;

// 入队

orderQueue.push(Order{1001, "ORD20240101001"});

orderQueue.push(Order{1002, "ORD20240101002"});

// 出队

while (!orderQueue.empty()) {

Order order = orderQueue.front();

orderQueue.pop();

// 处理订单

processOrder(order);

}

```

4.2 Qt容器

【QList动态列表】

```cpp

// 商品列表

QList<Product> products;

// 添加元素

products.append(Product{"iPhone", 5999.0});

products << Product{"MacBook", 12999.0};

// 访问元素

Product first = products.first();

Product last = products.last();

Product at = products.at(0);

// 遍历

for (const Product& p : products) {

qDebug() << p.name << p.price;

}

// 查找

int index = products.indexOf(targetProduct);

bool contains = products.contains(targetProduct);

// 移除

products.removeAt(0);

products.removeOne(targetProduct);

```

【QMap有序映射】

```cpp

// 用户信息映射

QMap<long, QString> userMap;

userMap.insert(1001, "Alice");

userMap.insert(1002, "Bob");

userMap[1003] = "Charlie";

// 遍历

QMapIterator<long, QString> it(userMap);

while (it.hasNext()) {

it.next();

qDebug() << it.key() << ":" << it.value();

}

// C++11 range-based for

for (auto it = userMap.begin(); it != userMap.end(); ++it) {

qDebug() << it.key() << ":" << it.value();

}

// 查找

if (userMap.contains(1001)) {

QString name = userMap.value(1001);

}

```

【QHash哈希映射】

```cpp

// 商品缓存

QHash<long, Product> productCache;

productCache.insert(1, Product{"iPhone", 5999.0});

productCache[2] = Product{"MacBook", 12999.0};

// 查找

if (productCache.contains(1)) {

Product p = productCache.value(1);

}

// 移除

productCache.remove(1);

// 清空

productCache.clear();

// 大小

int size = productCache.size();

bool empty = productCache.isEmpty();

```

4.3 Java集合

【ArrayList动态数组】

```java

// 商品列表

ArrayList<Product> products = new ArrayList<>();

// 添加元素

products.add(new Product(1, "iPhone", 5999.0));

products.add(0, new Product(2, "MacBook", 12999.0)); // 插入到指定位置

// 访问元素

Product first = products.get(0);

// 遍历

for (Product p : products) {

System.out.println(p.getName() + ": " + p.getPrice());

}

// 使用Stream API

products.stream()

.filter(p -> p.getPrice() > 1000)

.forEach(p -> System.out.println(p.getName()));

// 排序

products.sort(Comparator.comparing(Product::getPrice));

// 删除

products.remove(0);

products.removeIf(p -> p.getPrice() < 100);

```

【HashMap哈希映射】

```java

// 用户会话映射

HashMap<Long, UserSession> sessionMap = new HashMap<>();

// 添加

sessionMap.put(1001L, new UserSession(1001, "admin"));

sessionMap.putIfAbsent(1002L, new UserSession(1002, "user1"));

// 查找

UserSession session = sessionMap.get(1001L);

UserSession orDefault = sessionMap.getOrDefault(1003L, defaultSession);

// 遍历

for (Map.Entry<Long, UserSession> entry : sessionMap.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

// Lambda遍历

sessionMap.forEach((userId, session) -> {

System.out.println(userId + ": " + session.getUsername());

});

// 删除

sessionMap.remove(1001L);

```

【ConcurrentHashMap线程安全映射】

```java

// 线程安全的会话存储

ConcurrentHashMap<ChannelId, UserSession> sessions = new ConcurrentHashMap<>();

// 原子操作

sessions.putIfAbsent(channelId, newSession);

sessions.computeIfAbsent(channelId, k -> createSession(k));

sessions.computeIfPresent(channelId, (k, v) -> updateSession(v));

// 批量操作

sessions.forEach((channelId, session) -> {

if (session.isExpired()) {

sessions.remove(channelId);

}

});

// 统计

long activeCount = sessions.values().stream()

.filter(UserSession::isActive)

.count();

```

五、难点问题讨论

（1）JNI内存管理难点

1.1 问题描述

JNI(Java Native Interface)是Java调用C++代码的桥梁。在本项目中,Netty服务器通过JNI调用C++业务逻辑层,涉及大量的内存管理问题。

主要难点:

1. \*\*字符串编码转换\*\*:Java String(UTF-16) ↔ C++ std::string(UTF-8)

2. \*\*内存泄漏风险\*\*:JNI对象引用未正确释放

3. \*\*异常处理\*\*:C++异常无法直接传递到Java层

4. \*\*生命周期管理\*\*:Java对象和C++对象的生命周期不一致

1.2 解决方案

【字符串安全转换】

```cpp

// 错误示例:容易造成内存泄漏

JNIEXPORT jstring JNICALL Java\_emshop\_login(JNIEnv \*env, jclass, jstring jUsername) {

const char\* username = env->GetStringUTFChars(jUsername, nullptr);

// 如果这里抛出异常,username不会被释放!

std::string result = processLogin(username);

env->ReleaseStringUTFChars(jUsername, username);

return env->NewStringUTF(result.c\_str());

}

// 正确示例:使用RAII确保释放

class JStringGuard {

private:

JNIEnv\* env;

jstring jstr;

const char\* cstr;

public:

JStringGuard(JNIEnv\* e, jstring js)

: env(e), jstr(js), cstr(env->GetStringUTFChars(js, nullptr)) {}

~JStringGuard() {

if (cstr) {

env->ReleaseStringUTFChars(jstr, cstr);

}

}

const char\* c\_str() const { return cstr; }

};

JNIEXPORT jstring JNICALL Java\_emshop\_login(JNIEnv \*env, jclass, jstring jUsername) {

JStringGuard username(env, jUsername); // 自动管理生命周期

try {

std::string result = processLogin(username.c\_str());

return env->NewStringUTF(result.c\_str());

} catch (const std::exception& e) {

// 异常也会正确释放字符串

json errorResponse;

errorResponse["success"] = false;

errorResponse["message"] = e.what();

return env->NewStringUTF(errorResponse.dump().c\_str());

}

}

```

【JNI异常处理机制】

```cpp

// C++异常无法直接传递到Java,需要转换为JSON错误响应

std::string handleJNIException(JNIEnv\* env, const std::exception& e, const std::string& errorCode) {

json response;

response["success"] = false;

response["message"] = e.what();

response["error\_code"] = errorCode;

// 可选:设置Java异常(但会中断执行)

// jclass exClass = env->FindClass("java/lang/RuntimeException");

// env->ThrowNew(exClass, e.what());

return response.dump();

}

JNIEXPORT jstring JNICALL Java\_emshop\_createOrder(

JNIEnv \*env, jclass,

jlong userId, jlong addressId, jlongArray cartIds) {

try {

// 转换Java数组到C++ vector

jsize len = env->GetArrayLength(cartIds);

jlong\* elements = env->GetLongArrayElements(cartIds, nullptr);

std::vector<long> cartIdVec(elements, elements + len);

// 释放数组元素

env->ReleaseLongArrayElements(cartIds, elements, JNI\_ABORT);

// 调用业务逻辑

std::string result = OrderService::getInstance().createOrder(

userId, addressId, cartIdVec, 0, "alipay"

);

return env->NewStringUTF(result.c\_str());

} catch (const sql::SQLException& e) {

std::string error = handleJNIException(env, e, "E40001");

return env->NewStringUTF(error.c\_str());

} catch (const std::exception& e) {

std::string error = handleJNIException(env, e, "E00001");

return env->NewStringUTF(error.c\_str());

}

}

```

【中文乱码问题】

```cpp

// 问题:MySQL返回的中文数据显示为乱码

// 原因:字符集配置不一致

// 解决方案1:数据库连接时指定字符集

sql::Connection\* conn = driver->connect(host, user, password);

conn->setSchema(database);

conn->setClientOption("charset", "utf8mb4"); // 关键!

// 解决方案2:查询时指定字符集

stmt->execute("SET NAMES utf8mb4");

stmt->execute("SET CHARACTER SET utf8mb4");

// 解决方案3:数据库表使用utf8mb4

CREATE TABLE products (

product\_id BIGINT PRIMARY KEY,

product\_name VARCHAR(100) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci,

description TEXT CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

// 解决方案4:配置文件统一设置

[mysqld]

character-set-server = utf8mb4

collation-server = utf8mb4\_unicode\_ci

[client]

default-character-set = utf8mb4

```

1.3 经验总结

1. \*\*始终使用RAII\*\*:自动管理JNI对象生命周期

2. \*\*统一字符集\*\*:全栈使用UTF-8/UTF-8MB4

3. \*\*异常转换\*\*:C++异常转为JSON错误响应

4. \*\*及时释放\*\*:GetStringUTFChars后必须ReleaseStringUTFChars

5. \*\*避免局部引用泄漏\*\*:大量JNI调用时使用DeleteLocalRef

（2）并发库存控制难点

2.1 问题描述

电商系统中,多个用户可能同时购买同一商品,如果不加控制,会出现超卖问题:

```

初始库存:10件

时刻T1: 用户A查询库存=10,准备购买5件

时刻T2: 用户B查询库存=10,准备购买8件

时刻T3: 用户A扣减库存,剩余5件

时刻T4: 用户B扣减库存,剩余2件 ← 错误!实际应该库存不足

结果:实际售出13件,超出库存10件

```

2.2 解决方案对比

【方案1:悲观锁(SELECT FOR UPDATE)】

```sql

START TRANSACTION;

-- 加锁查询

SELECT stock FROM products WHERE product\_id = 1 FOR UPDATE;

-- 检查库存

IF stock >= 5 THEN

-- 扣减库存

UPDATE products SET stock = stock - 5 WHERE product\_id = 1;

COMMIT;

ELSE

ROLLBACK;

END IF;

```

优点:

- 实现简单,逻辑清晰

- 完全避免超卖

- 适合高冲突场景

缺点:

- 并发性能差,锁等待时间长

- 可能造成死锁

- 数据库压力大

【方案2:乐观锁(版本号)】

```sql

-- 1. 读取当前版本号

SELECT stock, version FROM products WHERE product\_id = 1;

-- 假设读到:stock=10, version=5

-- 2. 更新时检查版本号

UPDATE products

SET stock = stock - 5, version = version + 1

WHERE product\_id = 1 AND version = 5 AND stock >= 5;

-- 3. 如果affected\_rows=0,说明版本号冲突,重试

```

优点:

- 无锁,并发性能好

- 不会造成死锁

- 数据库压力小

缺点:

- 需要额外版本号字段

- 冲突时需要重试

- 高冲突场景性能下降

【方案3:Redis分布式锁】

```java

public boolean decreaseStock(long productId, int quantity) {

String lockKey = "lock:product:" + productId;

String lockValue = UUID.randomUUID().toString();

try {

// 1. 获取分布式锁(超时30秒)

boolean locked = redisTemplate.opsForValue()

.setIfAbsent(lockKey, lockValue, 30, TimeUnit.SECONDS);

if (!locked) {

return false; // 获取锁失败

}

// 2. 查询库存

Integer stock = getStockFromRedis(productId);

// 3. 检查库存

if (stock < quantity) {

return false;

}

// 4. 扣减库存

redisTemplate.opsForValue().decrement("stock:" + productId, quantity);

// 5. 异步更新数据库

asyncUpdateDatabase(productId, quantity);

return true;

} finally {

// 释放锁(使用Lua脚本保证原子性)

String script =

"if redis.call('get', KEYS[1]) == ARGV[1] then " +

" return redis.call('del', KEYS[1]) " +

"else " +

" return 0 " +

"end";

redisTemplate.execute(new DefaultRedisScript<>(script, Long.class),

Collections.singletonList(lockKey), lockValue);

}

}

```

优点:

- 性能极高,支持高并发

- 跨服务器分布式锁

- 减轻数据库压力

缺点:

- 架构复杂,引入Redis

- 需要考虑数据一致性

- 锁超时问题

2.3 本项目采用方案

本项目采用\*\*乐观锁(版本号)\*\*方案:

```cpp

std::string ProductService::decreaseStock(long productId, int quantity) {

const int MAX\_RETRY = 3;

for (int retry = 0; retry < MAX\_RETRY; retry++) {

try {

ConnectionGuard guard;

sql::Connection\* conn = guard.get();

// 1. 查询当前库存和版本号

json result = executeQuery(

"SELECT stock, version FROM products WHERE product\_id = ?",

{productId}, conn

);

if (result["data"].empty()) {

return buildErrorResponse("商品不存在", "E20301");

}

int stock = std::stoi(result["data"][0]["stock"].get<std::string>());

int version = std::stoi(result["data"][0]["version"].get<std::string>());

// 2. 检查库存

if (stock < quantity) {

return buildErrorResponse("库存不足", "E20302");

}

// 3. 乐观锁更新

int affectedRows = executeUpdate(

"UPDATE products SET stock = stock - ?, version = version + 1 "

"WHERE product\_id = ? AND version = ? AND stock >= ?",

{quantity, productId, version, quantity},

conn

);

if (affectedRows > 0) {

// 成功

json response;

response["success"] = true;

response["message"] = "扣减库存成功";

return response.dump();

}

// 版本冲突,短暂休眠后重试

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

} catch (const std::exception& e) {

return handleException(e, "E20399");

}

}

// 重试次数用尽

return buildErrorResponse("扣减库存失败,请稍后重试", "E20303");

}

```

选择理由:

1. 项目规模不大,并发冲突概率低

2. 无需引入Redis,降低架构复杂度

3. 数据库层面保证一致性

4. 重试机制应对偶发冲突

（3）订单状态机复杂度

3.1 问题描述

订单状态转换规则复杂,需要严格控制状态流转:

```

正常流程:

pending → paid → shipped → completed

异常流程:

pending → cancelled (超时未支付)

paid → refunding → refunded (用户申请退款)

shipped → refunding → refunded (用户申请退货)

```

难点:

1. \*\*状态转换合法性检查\*\*

2. \*\*状态转换的原子性\*\*

3. \*\*状态转换的副作用\*\*(如库存回滚、优惠券恢复)

4. \*\*并发状态转换冲突\*\*

3.2 解决方案

【状态机模式】

```cpp

// OrderStateMachine.h

class OrderStateMachine {

private:

// 合法的状态转换映射

static const std::map<std::string, std::vector<std::string>> transitions;

public:

// 检查状态转换是否合法

static bool canTransition(const std::string& from, const std::string& to);

// 执行状态转换(包含副作用)

static std::string transitionTo(

long orderId,

const std::string& fromStatus,

const std::string& toStatus,

sql::Connection\* conn

);

private:

// 状态转换的副作用处理

static void handleCancelOrder(long orderId, sql::Connection\* conn);

static void handlePayOrder(long orderId, sql::Connection\* conn);

static void handleShipOrder(long orderId, sql::Connection\* conn);

static void handleCompleteOrder(long orderId, sql::Connection\* conn);

static void handleRefundOrder(long orderId, sql::Connection\* conn);

};

// OrderStateMachine.cpp

const std::map<std::string, std::vector<std::string>>

OrderStateMachine::transitions = {

{"pending", {"paid", "cancelled"}},

{"paid", {"shipped", "refunding"}},

{"shipped", {"completed", "refunding"}},

{"refunding", {"refunded"}},

{"completed", {}}, // 终态

{"cancelled", {}}, // 终态

{"refunded", {}} // 终态

};

bool OrderStateMachine::canTransition(

const std::string& from,

const std::string& to) {

auto it = transitions.find(from);

if (it == transitions.end()) {

return false;

}

const auto& allowed = it->second;

return std::find(allowed.begin(), allowed.end(), to) != allowed.end();

}

std::string OrderStateMachine::transitionTo(

long orderId,

const std::string& fromStatus,

const std::string& toStatus,

sql::Connection\* conn) {

// 1. 检查转换合法性

if (!canTransition(fromStatus, toStatus)) {

json error;

error["success"] = false;

error["message"] = "不允许的状态转换: " + fromStatus + " -> " + toStatus;

error["error\_code"] = "E40401";

return error.dump();

}

try {

// 2. 开启事务

conn->setAutoCommit(false);

// 3. 更新订单状态(乐观锁)

std::unique\_ptr<sql::PreparedStatement> pstmt(

conn->prepareStatement(

"UPDATE orders SET order\_status = ?, updated\_at = NOW() "

"WHERE order\_id = ? AND order\_status = ?"

)

);

pstmt->setString(1, toStatus);

pstmt->setInt64(2, orderId);

pstmt->setString(3, fromStatus);

int affected = pstmt->executeUpdate();

if (affected == 0) {

conn->rollback();

json error;

error["success"] = false;

error["message"] = "订单状态已改变,请刷新后重试";

error["error\_code"] = "E40402";

return error.dump();

}

// 4. 执行副作用

if (toStatus == "cancelled") {

handleCancelOrder(orderId, conn);

} else if (toStatus == "paid") {

handlePayOrder(orderId, conn);

} else if (toStatus == "shipped") {

handleShipOrder(orderId, conn);

} else if (toStatus == "completed") {

handleCompleteOrder(orderId, conn);

} else if (toStatus == "refunded") {

handleRefundOrder(orderId, conn);

}

// 5. 提交事务

conn->commit();

json response;

response["success"] = true;

response["message"] = "状态更新成功";

return response.dump();

} catch (const std::exception& e) {

conn->rollback();

json error;

error["success"] = false;

error["message"] = e.what();

error["error\_code"] = "E40499";

return error.dump();

}

}

void OrderStateMachine::handleCancelOrder(long orderId, sql::Connection\* conn) {

// 1. 恢复商品库存

std::unique\_ptr<sql::PreparedStatement> pstmt(

conn->prepareStatement(

"UPDATE products p "

"INNER JOIN order\_items oi ON p.product\_id = oi.product\_id "

"SET p.stock = p.stock + oi.quantity "

"WHERE oi.order\_id = ?"

)

);

pstmt->setInt64(1, orderId);

pstmt->executeUpdate();

// 2. 恢复优惠券

pstmt.reset(conn->prepareStatement(

"UPDATE user\_coupons uc "

"INNER JOIN orders o ON uc.user\_id = o.user\_id AND uc.coupon\_id = o.coupon\_id "

"SET uc.status = 'unused', uc.used\_at = NULL "

"WHERE o.order\_id = ? AND o.coupon\_id IS NOT NULL"

));

pstmt->setInt64(1, orderId);

pstmt->executeUpdate();

}

void OrderStateMachine::handleRefundOrder(long orderId, sql::Connection\* conn) {

// 1. 恢复商品库存

handleCancelOrder(orderId, conn);

// 2. 记录退款信息(实际需调用支付接口)

std::unique\_ptr<sql::PreparedStatement> pstmt(

conn->prepareStatement(

"INSERT INTO refund\_records (order\_id, refund\_amount, refund\_reason, refund\_time) "

"SELECT order\_id, final\_amount, '用户申请退款', NOW() "

"FROM orders WHERE order\_id = ?"

)

);

pstmt->setInt64(1, orderId);

pstmt->executeUpdate();

}

```

3.3 经验总结

1. \*\*使用状态机模式\*\*:明确定义状态转换规则

2. \*\*乐观锁防并发\*\*:WHERE条件检查当前状态

3. \*\*事务保证原子性\*\*:状态更新和副作用在同一事务

4. \*\*记录状态变更历史\*\*:使用审计表追踪状态流转

5. \*\*异步任务处理\*\*:超时取消订单使用定时任务

（4）网络异常处理

4.1 问题描述

TCP长连接可能遇到的问题:

1. \*\*客户端突然断开\*\*(进程崩溃、网络中断)

2. \*\*服务端崩溃\*\*

3. \*\*网络抖动导致丢包\*\*

4. \*\*请求超时\*\*

5. \*\*半包/粘包\*\*

4.2 解决方案

【Netty心跳检测】

```java

public class EmshopServerInitializer extends ChannelInitializer<SocketChannel> {

@Override

protected void initChannel(SocketChannel ch) {

ChannelPipeline pipeline = ch.pipeline();

// 心跳检测:60秒没有读操作触发IdleStateEvent

pipeline.addLast(new IdleStateHandler(60, 0, 0, TimeUnit.SECONDS));

// 自定义心跳处理器

pipeline.addLast(new HeartbeatHandler());

// 行分隔符解码器(解决半包/粘包)

pipeline.addLast(new LineBasedFrameDecoder(8192));

pipeline.addLast(new StringDecoder(StandardCharsets.UTF\_8));

pipeline.addLast(new StringEncoder(StandardCharsets.UTF\_8));

// 业务处理器

pipeline.addLast(new EmshopServerHandler());

}

}

public class HeartbeatHandler extends ChannelInboundHandlerAdapter {

private static final Logger logger = LoggerFactory.getLogger(HeartbeatHandler.class);

@Override

public void userEventTriggered(ChannelHandlerContext ctx, Object evt) {

if (evt instanceof IdleStateEvent) {

IdleStateEvent event = (IdleStateEvent) evt;

if (event.state() == IdleState.READER\_IDLE) {

// 60秒没有读到数据,认为客户端已断开

logger.warn("Client {} idle timeout, closing connection",

ctx.channel().remoteAddress());

ctx.close();

}

} else {

ctx.fireUserEventTriggered(evt);

}

}

}

```

【Qt客户端重连机制】

```cpp

class NetworkClient : public QObject {

Q\_OBJECT

private:

QTcpSocket\* socket;

QTimer\* reconnectTimer;

QString host;

quint16 port;

int reconnectAttempts;

const int MAX\_RECONNECT\_ATTEMPTS = 5;

public:

NetworkClient(QObject\* parent = nullptr)

: QObject(parent),

socket(new QTcpSocket(this)),

reconnectTimer(new QTimer(this)),

reconnectAttempts(0) {

connect(socket, &QTcpSocket::connected,

this, &NetworkClient::onConnected);

connect(socket, &QTcpSocket::disconnected,

this, &NetworkClient::onDisconnected);

connect(socket, QOverload<QAbstractSocket::SocketError>::of(&QTcpSocket::errorOccurred),

this, &NetworkClient::onError);

connect(reconnectTimer, &QTimer::timeout,

this, &NetworkClient::attemptReconnect);

}

void connectToServer(const QString& h, quint16 p) {

host = h;

port = p;

reconnectAttempts = 0;

socket->connectToHost(host, port);

}

private slots:

void onConnected() {

qDebug() << "Connected to server";

reconnectAttempts = 0;

reconnectTimer->stop();

emit connected();

}

void onDisconnected() {

qDebug() << "Disconnected from server";

emit disconnected();

// 开始重连

if (reconnectAttempts < MAX\_RECONNECT\_ATTEMPTS) {

int delay = std::min(1000 \* (1 << reconnectAttempts), 30000); // 指数退避

reconnectTimer->start(delay);

} else {

emit reconnectFailed();

}

}

void onError(QAbstractSocket::SocketError error) {

qDebug() << "Socket error:" << socket->errorString();

emit errorOccurred(socket->errorString());

}

void attemptReconnect() {

reconnectAttempts++;

qDebug() << "Reconnect attempt" << reconnectAttempts << "/" << MAX\_RECONNECT\_ATTEMPTS;

socket->connectToHost(host, port);

}

signals:

void connected();

void disconnected();

void errorOccurred(const QString& error);

void reconnectFailed();

};

```

【请求超时处理】

```cpp

void NetworkClient::sendRequest(const QString& request,

std::function<void(const QString&)> callback) {

// 生成请求ID

QString requestId = QUuid::createUuid().toString();

// 创建超时定时器

QTimer\* timeoutTimer = new QTimer(this);

timeoutTimer->setSingleShot(true);

timeoutTimer->setInterval(30000); // 30秒超时

connect(timeoutTimer, &QTimer::timeout, [this, requestId]() {

// 超时处理

pendingRequests.remove(requestId);

json error;

error["success"] = false;

error["message"] = "请求超时";

error["error\_code"] = "E00002";

QString errorJson = QString::fromStdString(error.dump());

emit requestTimeout(requestId, errorJson);

});

// 保存回调

pendingRequests[requestId] = {callback, timeoutTimer};

// 发送请求(带请求ID)

QString fullRequest = QString("%1|%2").arg(requestId).arg(request);

socket->write(fullRequest.toUtf8() + "\n");

socket->flush();

// 启动超时定时器

timeoutTimer->start();

}

void NetworkClient::onResponseReceived(const QString& response) {

// 解析响应(格式: requestId|responseData)

QStringList parts = response.split('|', Qt::SkipEmptyParts);

if (parts.size() < 2) {

return;

}

QString requestId = parts[0];

QString responseData = parts.mid(1).join('|');

// 查找对应的回调

auto it = pendingRequests.find(requestId);

if (it != pendingRequests.end()) {

// 停止超时定时器

it->second.timeoutTimer->stop();

it->second.timeoutTimer->deleteLater();

// 调用回调

it->second.callback(responseData);

// 移除请求

pendingRequests.erase(it);

}

}

```

（5）Qt信号槽调试难度

5.1 问题描述

Qt的信号槽机制虽然强大,但调试困难:

1. \*\*连接失败静默\*\*:connect返回false但无明显错误提示

2. \*\*槽函数未执行\*\*:信号发出但槽没有响应

3. \*\*跨线程信号死锁\*\*:BlockingQueuedConnection使用不当

4. \*\*循环信号\*\*:信号A触发槽B,槽B又emit信号A

5.2 解决方案

【连接检查】

```cpp

// 错误示例:忽略返回值

connect(button, &QPushButton::clicked, this, &MainWindow::onButtonClicked);

// 正确示例:检查连接是否成功

bool connected = connect(button, &QPushButton::clicked,

this, &MainWindow::onButtonClicked);

if (!connected) {

qCritical() << "Failed to connect button signal!";

}

// 使用断言(Debug模式)

Q\_ASSERT(connect(button, &QPushButton::clicked,

this, &MainWindow::onButtonClicked));

```

【调试输出】

```cpp

class MainWindow : public QMainWindow {

Q\_OBJECT

signals:

void dataUpdated(int value);

private slots:

void onDataUpdated(int value) {

qDebug() << Q\_FUNC\_INFO << "value =" << value;

// Q\_FUNC\_INFO输出: void MainWindow::onDataUpdated(int) value = 42

}

private:

void emitSignal() {

int value = 42;

qDebug() << Q\_FUNC\_INFO << "Emitting dataUpdated with value =" << value;

emit dataUpdated(value);

}

};

```

【避免循环信号】

```cpp

class MainWindow : public QMainWindow {

Q\_OBJECT

private:

bool isUpdating; // 标志位

private slots:

void onSlider1Changed(int value) {

if (isUpdating) return; // 防止递归

isUpdating = true;

slider2->setValue(value); // 这会触发onSlider2Changed

isUpdating = false;

}

void onSlider2Changed(int value) {

if (isUpdating) return;

isUpdating = true;

slider1->setValue(value); // 这会触发onSlider1Changed

isUpdating = false;

}

};

// 或使用blockSignals

void onSlider1Changed(int value) {

slider2->blockSignals(true);

slider2->setValue(value);

slider2->blockSignals(false);

}

```

六、收获和展望

（1）项目收获

1.1 技术能力提升

【面向对象程序设计】

- 掌握了面向对象的三大特性:封装、继承、多态

- 学会使用设计模式解决实际问题(单例、工厂、策略等)

- 理解了SOLID原则在项目中的应用

- 能够设计清晰的类层次结构

【C/S架构理解】

- 深入理解客户端-服务器通信模式

- 掌握TCP/IP协议栈的应用层开发

- 学会设计应用层通信协议

- 了解了长连接、短连接的使用场景

【数据库设计与应用】

- 掌握E-R图建模和数据库范式

- 学会设计合理的表结构和索引

- 熟练使用SQL进行增删改查

- 理解事务、锁、触发器等高级特性

【多线程编程】

- 理解Netty的Reactor多线程模型

- 掌握Qt的信号槽跨线程通信

- 学会使用互斥锁、读写锁

- 了解线程安全容器的使用

【跨语言开发(JNI)】

- 掌握JNI的基本使用方法

- 理解Java和C++的内存模型差异

- 学会处理跨语言的异常和编码问题

- 积累了JNI内存管理经验

1.2 工程能力提升

【项目规划能力】

- 学会分析需求并划分功能模块

- 掌握了任务分解和时间估算

- 理解了迭代开发的重要性

【代码质量意识】

- 养成了良好的命名习惯

- 学会编写清晰的注释

- 重视代码复用和模块化

- 了解了代码规范的重要性

【调试能力】

- 掌握了断点调试技巧

- 学会使用日志定位问题

- 积累了常见错误的排查经验

- 提高了阅读错误信息的能力

【文档编写能力】

- 学会编写技术文档

- 掌握了Markdown语法

- 理解了文档对项目的重要性

1.3 团队协作能力

【版本控制】

- 熟练使用Git进行版本管理

- 掌握分支、合并、冲突解决

- 学会编写规范的commit message

【沟通能力】

- 学会清晰描述技术问题

- 理解了需求沟通的重要性

- 提高了文字表达能力

（2）项目不足与改进

2.1 当前不足

【功能层面】

1. \*\*支付功能未对接真实接口\*\*:目前仅模拟支付,实际应对接支付宝/微信支付

2. \*\*缺少商家端\*\*:目前只有客户端,应增加商家管理后台

3. \*\*无实时通知\*\*:订单状态变更无法实时推送给用户

4. \*\*搜索功能简单\*\*:仅支持关键词匹配,无分词和相关性排序

5. \*\*图片管理不完善\*\*:商品图片存储在本地,应使用OSS对象存储

【性能层面】

1. \*\*无缓存机制\*\*:商品信息等热点数据应使用Redis缓存

2. \*\*数据库连接池过小\*\*:初始连接数仅5个,高并发时可能不足

3. \*\*无负载均衡\*\*:单点服务器,无法水平扩展

4. \*\*日志未分级\*\*:所有日志混在一起,不利于问题排查

【安全层面】

1. \*\*密码加密算法简单\*\*:SHA256无加盐,应使用bcrypt

2. \*\*无HTTPS支持\*\*:数据明文传输,存在安全隐患

3. \*\*无SQL注入防护测试\*\*:虽然使用了预编译语句,但未系统测试

4. \*\*无访问频率限制\*\*:容易被暴力破解或DOS攻击

【用户体验】

1. \*\*UI设计单调\*\*:界面较为简陋,缺少视觉设计

2. \*\*无离线功能\*\*:断网后无法使用

3. \*\*错误提示不友好\*\*:部分错误提示过于技术化

4. \*\*无国际化支持\*\*:界面硬编码中文

2.2 改进计划

【短期改进(1-2个月)】

1. 对接真实支付接口(支付宝沙箱)

2. 开发Web管理后台(Vue.js + Element UI)

3. 引入Redis缓存热点数据

4. 添加Logback日志分级

5. 优化UI界面设计

【中期改进(3-6个月)】

1. 实现WebSocket实时通知

2. 引入Elasticsearch实现全文搜索

3. 使用阿里云OSS存储图片

4. 实现HTTPS加密通信

5. 添加访问频率限制中间件

【长期规划(6-12个月)】

1. 微服务改造(Spring Cloud)

2. 容器化部署(Docker + Kubernetes)

3. 实现秒杀功能

4. 开发移动端APP(Flutter)

5. 引入大数据分析(推荐系统)

（3）未来展望

3.1 技术方向

【后端技术】

- 学习Spring Boot / Spring Cloud微服务架构

- 深入研究分布式系统(一致性、CAP理论)

- 掌握消息队列(RabbitMQ / Kafka)

- 学习容器技术(Docker / Kubernetes)

- 研究云原生架构

【前端技术】

- 学习现代前端框架(React / Vue 3)

- 掌握TypeScript语言

- 学习前端工程化(Webpack / Vite)

- 研究响应式设计和移动端开发

【数据库技术】

- 深入学习MySQL调优

- 了解NoSQL数据库(MongoDB / Cassandra)

- 学习数据库中间件(MyCat / ShardingSphere)

- 研究时序数据库(InfluxDB)

【运维技术】

- 学习Linux系统管理

- 掌握CI/CD流程(Jenkins / GitLab CI)

- 学习监控系统(Prometheus + Grafana)

- 了解日志收集(ELK Stack)

3.2 职业规划

【近期目标(1年内)】

- 参与1-2个真实商业项目

- 贡献开源项目

- 考取相关技术认证

- 深入学习某个技术方向

【中期目标(3年内)】

- 成为某个技术领域的专家

- 具备独立设计系统架构的能力

- 能够带领小团队完成项目

- 发表技术文章分享经验

【长期目标(5年内)】

- 成为资深技术专家或架构师

- 具备全栈开发能力

- 对某个技术方向有深入研究

- 具备技术决策能力

3.3 项目演进路线

【Version 1.0(当前)】

- ✅ 基础功能完整

- ✅ C/S架构实现

- ✅ 数据库持久化

- ✅ 简单UI界面

【Version 2.0(计划中)】

- 📋 Web管理后台

- 📋 Redis缓存层

- 📋 支付宝对接

- 📋 实时通知

- 📋 UI美化

【Version 3.0(展望)】

- 🔮 微服务架构

- 🔮 全文搜索

- 🔮 秒杀功能

- 🔮 推荐系统

- 🔮 移动端APP

【Version 4.0(愿景)】

- 🌟 云原生部署

- 🌟 AI智能客服

- 🌟 大数据分析

- 🌟 国际化支持

- 🌟 开放平台API

（4）总结

本项目从需求分析、系统设计、编码实现到测试部署,完整经历了软件开发的全生命周期。通过这个项目:

1. \*\*技术层面\*\*:掌握了C++、Java、Qt等多种技术栈,理解了C/S架构、数据库设计、多线程编程等核心概念。

2. \*\*工程层面\*\*:学会了模块划分、接口设计、代码组织等软件工程实践,养成了良好的编程习惯。

3. \*\*思维层面\*\*:培养了系统化思考问题的能力,学会了权衡取舍,理解了"没有银弹"的道理。

4. \*\*态度层面\*\*:体会到了细节的重要性,明白了"代码是写给人看的,顺便让机器执行"。

虽然项目还存在诸多不足,但这正是学习和成长的空间。技术的道路没有终点,只有不断学习,才能不断进步。

希望通过持续迭代改进,将这个项目打造成一个功能完善、架构优雅、性能优异的电商系统,为将来的职业发展奠定坚实基础。

最后,感谢所有帮助过我的老师、同学和朋友。正是因为你们的支持和鼓励,我才能完成这个项目。

\*\*路漫漫其修远兮,吾将上下而求索!\*\*

JLU Emshop 电商系统综合报告完成

总计八个部分

感谢您的阅读!