**面向对象程序设计实践（C++）**

**课程实验报告**

|  |  |
| --- | --- |
| 题目 | 电商交易平台设计与实现 |
| 姓名 | 江雨璇 |
| 学号 | 2023212796 |
| 班级 | 2023211302 |

**目录**

[1 前言 3](#_Toc176)

[2 总体设计 4](#_Toc16217)

[2.1 系统架构 4](#_Toc24819)

[3 详细设计 9](#_Toc31951)

[3.1 用户管理子系统详细设计 9](#_Toc6897)

[3.2 子系统商品管理详细设计 10](#_Toc20574)

[3.3 子系统购物车详细设计 11](#_Toc4559)

[3.4 子系统数据处理详细设计 11](#_Toc13506)

[3.5 子系统间关系 13](#_Toc12485)

[3.6 数据库说明 13](#_Toc32281)

[3.7 接口协议说明 14](#_Toc16945)

[4 实现 17](#_Toc24858)

[5 总结与展望 22](#_Toc13437)

[6 网络版实现说明 23](#_Toc5355)

# 前言

1.1 实验目的

1. 实现一个基于C++的电商平台系统

2. 掌握面向对象编程的核心概念和实践

3. 学习用户管理、商品管理和购物车功能的实现

4. 理解多态性和继承在系统设计中的应用

1.2 实验环境

1.2.1 硬件环境

- 处理器：任意支持C++的处理器

- 内存：建议4GB以上

1.2.2 软件环境

- 操作系统：Windows 10

- 开发工具：Visual Studio 2019或更高版本

- 编译器版本：支持C++11及以上标准

1.3 实验内容概述

本实验实现了一个简单的电商平台系统，包含用户管理、商品管理和购物车功能。系统支持两种用户角色（消费者和商家），实现了商品的增删改查、购物车管理、订单结算等功能。

# 总体设计

## 系统架构

2.1 系统架构

系统采用面向对象的设计方法，主要包含以下几个核心模块：

- 用户管理模块（User）

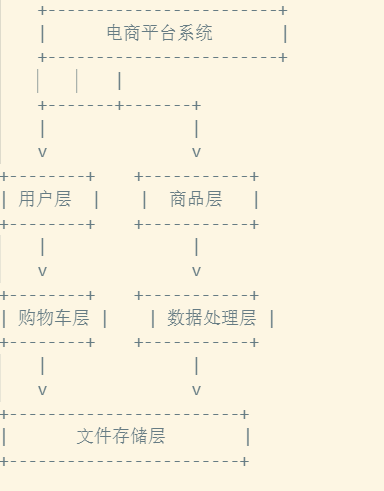
- 商品管理模块（Product）

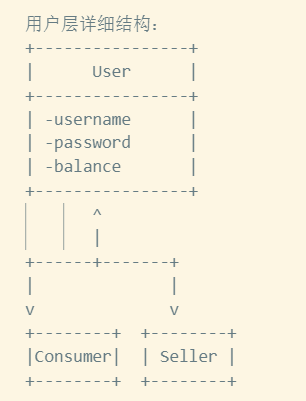
- 购物车模块（ShoppingCart）

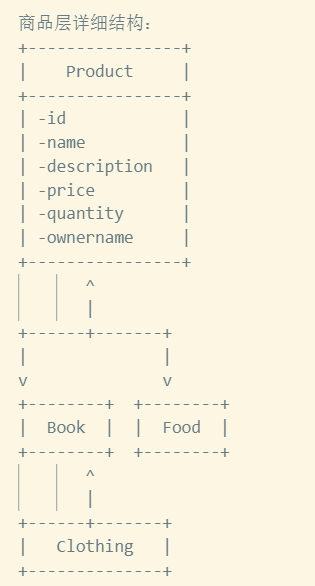
- 数据处理模块（DataHandler）

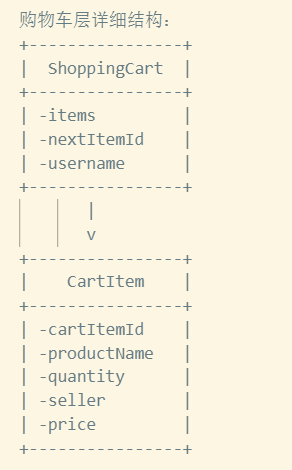
- 工具类模块（Utility）

类层次结构：









类关系说明：

1. 继承关系：

- User <- Consumer/Seller

- Product <- Book/Food/Clothing

2. 组合关系：

- ShoppingCart 包含 CartItem 对象

- Product 关联 User（通过 ownername）

3. 依赖关系：

- ShoppingCart 依赖 User（结算时）

- DataHandler 依赖 User 和 Product（数据管理）

主要属性说明：

1. User类：

- username：用户名

- password：密码

- balance：账户余额

2. Product类：

- id：商品ID

- name：商品名称

- description：商品描述

- price：价格

- quantity：库存数量

- ownername：商品所有者

3. CartItem类：

- cartItemId：购物车项ID

- productName：商品名称

- quantity：数量

- seller：卖家

- price：价格

4. ShoppingCart类：

- items：购物车项列表

- nextItemId：下一个购物车项ID

- username：用户名

2.2 功能模块划分

1. 用户管理功能

- 用户注册

- 用户登录

- 密码修改

2. 商品管理功能

- 商品信息查看

- 商品添加

- 商品修改

- 商品删除

- 商品搜索

- 商品筛选

- 商品折扣

3. 购物车功能

- 添加商品到购物车

- 查看购物车

- 修改购物车商品数量

- 删除购物车商品

- 结算功能

2.2 功能模块划分

1. 用户管理功能

- 用户注册

- 用户登录

- 密码修改

2. 商品管理功能

- 商品信息查看

- 商品添加

- 商品修改

- 商品删除

- 商品搜索

- 商品筛选

- 商品折扣

3. 购物车功能

- 添加商品到购物车

- 查看购物车

- 修改购物车商品数量

- 删除购物车商品

- 结算功能

2.3 类图设计

系统主要包含以下类：

- User（基类）

- Consumer（消费者类）

- Seller（商家类）

- Product（商品基类）

- Book（图书类）

- Food（食品类）

- Clothing（服装类）

- ShoppingCart（购物车类）

- DataHandler（数据处理类）

2.4 主要数据结构设计

1. 用户数据结构

- 用户名（string）

- 密码（string）

- 余额（double）

- 用户类型（string）

2. 商品数据结构

- 商品ID（int）

- 商品名称（string）

- 商品描述（string）

- 价格（double）

- 数量（int）

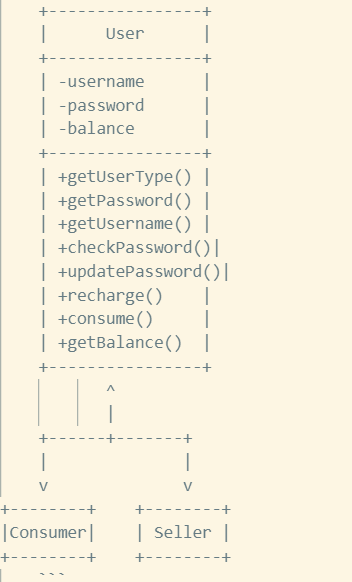
- 商品类型（string）

- 所有者（string）

# 详细设计

分子系统给出详细设计，每个子系统用到的类，类之间的的关系（说明+图示）

## 用户管理子系统详细设计



3.1.2 类关系说明

- User类作为基类，定义了用户的基本属性和方法

- Consumer和Seller类继承自User类，分别实现消费者和商家的特定功能

- 所有用户共享基本的账户管理功能（充值、消费、密码修改等）

3.1.3 关键方法说明

- getUserType()：获取用户类型（消费者/商家）

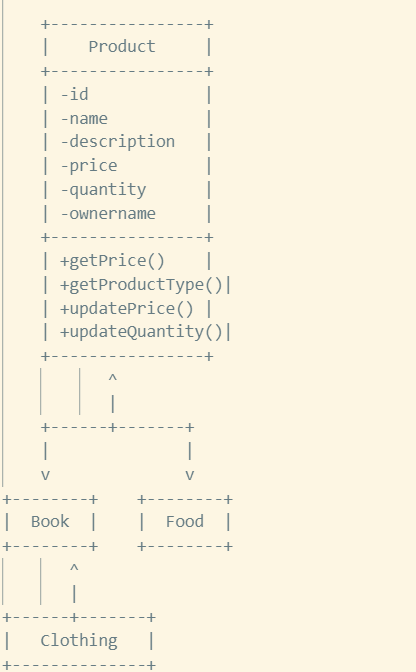
- checkPassword()：验证用户密码

- recharge()：账户充值

- consume()：账户消费

## 子系统商品管理详细设计

3.2.1 类设计



3.2.2 类关系说明

- Product类作为基类，定义了商品的基本属性和方法

- Book、Food、Clothing类继承自Product类，实现不同类型商品的特定功能

- 每种商品类型可以有自己的价格计算策略

3.2.3 关键方法说明

- getPrice()：获取商品价格（可被子类重写实现不同的价格策略）

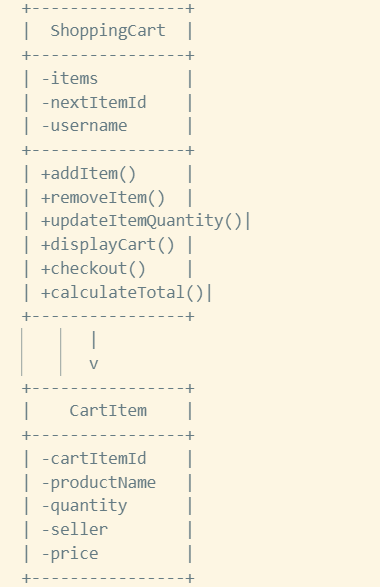
- getProductType()：获取商品类型

- updatePrice()：更新商品价格

- updateQuantity()：更新商品库存

## 子系统购物车详细设计

3.3.1 类设计



3.3.2 类关系说明

- ShoppingCart类管理购物车整体功能

- CartItem类表示购物车中的单个商品项

- ShoppingCart通过vector容器管理多个CartItem对象

3.3.3 关键方法说明

- addItem()：添加商品到购物车

- removeItem()：从购物车移除商品

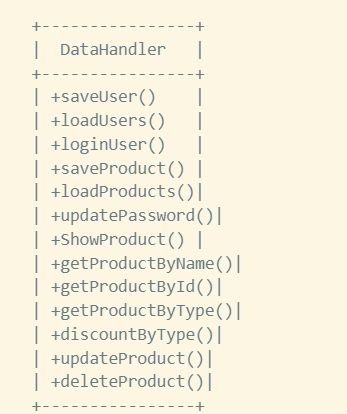
- updateItemQuantity()：更新商品数量

- checkout()：结算购物车

- calculateTotal()：计算总金额

## 子系统数据处理详细设计

3.4.1 类设计



3.4.2 类关系说明

- DataHandler类负责所有数据的持久化操作

- 与User类和Product类交互，处理用户和商品数据的存储和读取

- 提供数据查询和更新接口

3.4.3 关键方法说明

- saveUser()/loadUsers()：用户数据的保存和加载

- saveProduct()/loadProducts()：商品数据的保存和加载

- loginUser()：用户登录验证

- getProductByXXX()：多种商品查询方式

- updateProduct()/deleteProduct()：商品信息更新和删除

3.4.2 类关系说明

- DataHandler类负责所有数据的持久化操作

- 与User类和Product类交互，处理用户和商品数据的存储和读取

- 提供数据查询和更新接口

3.4.3 关键方法说明

- saveUser()/loadUsers()：用户数据的保存和加载

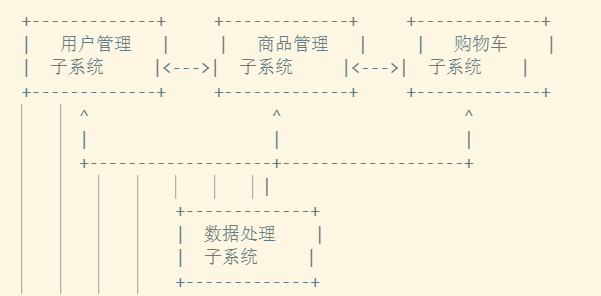
- saveProduct()/loadProducts()：商品数据的保存和加载

- loginUser()：用户登录验证

- getProductByXXX()：多种商品查询方式

- updateProduct()/deleteProduct()：商品信息更新和删除

## 子系统间关系



3.5.1 交互说明

- 用户管理子系统为其他子系统提供用户身份验证和权限控制

- 商品管理子系统为购物车子系统提供商品信息

- 购物车子系统在结算时需要与用户管理子系统交互

- 数据处理子系统为所有其他子系统提供数据持久化支持

## 数据库说明

3.6.1 文件存储格式

1. 用户数据文件（users.txt）

```

格式：username password balance userType

字段说明：

- username：字符串，用户名

- password：字符串，密码

- balance：浮点数，账户余额

- userType：字符串，用户类型（Consumer/Seller）

```

2. 商品数据文件（products.txt）

```

格式：id name description price quantity type ownername

字段说明：

- id：整数，商品ID

- name：字符串，商品名称

- description：字符串，商品描述

- price：浮点数，商品价格

- quantity：整数，库存数量

- type：字符串，商品类型（Book/Food/Clothing）

- ownername：字符串，商品所有者

3. 购物车数据文件（cart\_[username].txt）

```

格式：cartItemId productName quantity seller price

字段说明：

- cartItemId：整数，购物车项ID

- productName：字符串，商品名称

- quantity：整数，购买数量

- seller：字符串，卖家用户名

- price：浮点数，商品单价

```

3.6.2 数据关系说明

1. 用户-商品关系

- 一个用户（Seller）可以拥有多个商品

- 一个商品只能属于一个用户

- 关系通过商品的ownername字段维护

2. 用户-购物车关系

- 一个用户（Consumer）拥有一个购物车

- 购物车通过用户名关联

- 关系通过购物车文件名（cart\_[username].txt）维护

3. 商品-购物车关系

- 一个商品可以出现在多个购物车中

- 一个购物车可以包含多个商品

- 关系通过CartItem类维护

## 接口协议说明

3.7.1 子系统间通信协议

1. 用户认证协议

```

请求格式：AUTH|username|password

响应格式：AUTH\_RESP|status|userType

字段说明：

- status：认证状态（SUCCESS/FAILED）

- userType：用户类型（Consumer/Seller）

```

2. 商品操作协议

```

查询请求：PROD\_QUERY|type|value

查询响应：PROD\_RESP|status|data

更新请求：PROD\_UPDATE|id|field|value

更新响应：UPDATE\_RESP|status|message

字段说明：

- type：查询类型（BY\_NAME/BY\_ID/BY\_TYPE）

- value：查询值

- field：更新字段

- status：操作状态

- data：商品数据

- message：响应消息

```

3. 购物车操作协议

```

添加请求：CART\_ADD|productName|quantity

删除请求：CART\_REMOVE|cartItemId

更新请求：CART\_UPDATE|cartItemId|quantity

结算请求：CART\_CHECKOUT

响应格式：CART\_RESP|status|message|data

字段说明：

- status：操作状态

- message：响应消息

- data：操作结果数据

```

3.7.2 数据持久化协议

1. 文件读写协议

```

读取请求：FILE\_READ|filename

读取响应：FILE\_DATA|status|content

写入请求：FILE\_WRITE|filename|content

写入响应：FILE\_RESP|status|message

字段说明：

- filename：文件名

- content：文件内容

- status：操作状态

- message：响应消息

```

3.7.3 协议实现说明

1. 底层实现

- 所有协议基于文本格式实现

- 使用管道符（|）分隔字段

- 使用换行符分隔消息

2. 错误处理

- 所有响应都包含状态字段

- 错误信息通过message字段传递

- 支持基本的错误恢复机制

3. 安全性

- 用户密码在传输前进行加密

- 文件操作进行权限验证

- 关键操作需要用户认证

# 实现

4.1 核心代码实现

```cpp

// 用户登录实现

User\* DataHandler::loginUser(const std::string& username, const std::string& password) {

std::vector<User\*> users = loadUsers();

for (User\* user : users) {

if (user->getUsername() == username && user->checkPassword(password)) {

std::cout << "欢迎, " << username << "! 登录成功！" << std::endl;

return user;

}

}

std::cout << "用户名或密码错误！" << std::endl;

return NULL;

}

// 商品搜索实现

std::vector<Product\*> DataHandler::getProductByName(const std::string& name) {

std::vector<Product\*> products = loadProducts();

std::vector<Product\*> productsResult;

for (Product\* product : products) {

if (product->getName().find(name) != std::string::npos) {

productsResult.push\_back(product);

}

}

if (productsResult.empty()) {

std::cout << "没有找到相关商品！" << std::endl;

}

return productsResult;

}

// 商品折扣实现

void DataHandler::discountByType(const std::string& type, double discountRate, std::string ownername) {

std::vector<Product\*> products = loadProducts();

bool found = false;

for (Product\* product : products) {

if (product->getProductType() == type &&

(product->getOwnername().find(ownername) != std::string::npos)) {

double oldPrice = product->getPrice();

double newPrice = oldPrice \* discountRate;

product->updatePrice(newPrice);

found = true;

}

}

if (!found) {

std::cout << "没有找到类型为 " << type << " 的商品！" << std::endl;

return;

}

// 更新文件

std::ofstream file("products.txt", std::ios::trunc);

if (file.is\_open()) {

for (Product\* product : products) {

file << product->getId() << "\t"

<< product->getName() << "\t"

<< product->getDescription() << "\t"

<< product->getPrice() << "\t"

<< product->getQuantity() << "\t"

<< product->getOwnername() << "\t"

<< product->getProductType() << "\n";

}

file.close();

std::cout << "已成功更新类型为 " << type << " 的商品价格！" << std::endl;

}

}

```

4.2 关键算法实现

```cpp

// 购物车结算实现

bool ShoppingCart::checkout(Consumer\* buyer, std::vector<User\*>& allUsers) {

if (items.empty()) {

std::cout << "购物车为空，无法结算！" << std::endl;

return false;

}

// 计算每个卖家的总金额

std::map<std::string, double> sellerTotals;

for (const auto& item : items) {

double itemTotal = item.quantity \* item.price;

sellerTotals[item.seller] += itemTotal;

}

// 计算总金额

double totalAmount = calculateTotal();

// 检查买家余额是否足够

if (buyer->getBalance() < totalAmount) {

std::cout << "余额不足！" << std::endl;

std::cout << "订单总额: ￥" << totalAmount << std::endl;

std::cout << "当前余额: ￥" << buyer->getBalance() << std::endl;

std::cout << "还差: ￥" << totalAmount - buyer->getBalance() << std::endl;

return false;

}

// 显示订单预览

displayOrderPreview();

// 确认购买

std::cout << "\n确认要购买吗？" << std::endl;

std::cout << "1. 确认购买" << std::endl;

std::cout << "2. 取消购买" << std::endl;

std::cout << "请选择(1/2): ";

int choice;

std::cin >> choice;

if (choice != 1) {

std::cout << "已取消购买！" << std::endl;

return false;

}

// 执行交易

std::cout << "\n正在处理交易..." << std::endl;

// 扣除买家余额

for (User\* user : allUsers) {

if (user->getUsername() == buyer->getUsername() &&

user->getUserType() == "Consumer") {

user->consume(totalAmount);

std::cout << " 已扣除买家 " << buyer->getUsername()

<< " 余额 ￥" << totalAmount << std::endl;

break;

}

}

// 给每个卖家转账

bool allTransfersSuccessful = true;

for (const auto& sellerTotal : sellerTotals) {

bool sellerFound = false;

for (User\* user : allUsers) {

if (user->getUsername() == sellerTotal.first &&

user->getUserType() == "Seller") {

user->recharge(sellerTotal.second);

std::cout << " 已向卖家 " << sellerTotal.first

<< " 转账 ￥" << sellerTotal.second << std::endl;

sellerFound = true;

break;

}

}

if (!sellerFound) {

std::cout << "× 错误：未找到卖家 " << sellerTotal.first << std::endl;

allTransfersSuccessful = false;

}

}

// 处理交易结果

if (allTransfersSuccessful) {

// 更新用户文件

std::ofstream file("users.txt", std::ios::out | std::ios::trunc);

if (!file.is\_open()) {

std::cerr << "错误：无法打开用户文件进行更新！" << std::endl;

// 回滚交易

for (User\* user : allUsers) {

if (user->getUsername() == buyer->getUsername() &&

user->getUserType() == "Consumer") {

user->recharge(totalAmount);

break;

}

}

for (const auto& sellerTotal : sellerTotals) {

for (User\* user : allUsers) {

if (user->getUsername() == sellerTotal.first &&

user->getUserType() == "Seller") {

user->consume(sellerTotal.second);

break;

}

}

}

std::cout << "× 交易失败：无法更新用户数据！" << std::endl;

std::cout << " 已回滚所有资金转账" << std::endl;

return false;

}

// 写入更新后的用户数据

for (User\* user : allUsers) {

file << user->getUsername() << " "

<< user->getBalance() << " "

<< user->getUserType() << " "

<< user->getPassword() << "\n";

}

file.close();

// 清空购物车

items.clear();

saveToFile();

std::cout << "\n 交易成功完成！" << std::endl;

std::cout << " 总支付金额: ￥" << totalAmount << std::endl;

return true;

}

else {

// 交易失败，退还买家金额

for (User\* user : allUsers) {

if (user->getUsername() == buyer->getUsername() &&

user->getUserType() == "Consumer") {

user->recharge(totalAmount);

break;

}

}

// 更新用户文件

std::ofstream file("users.txt", std::ios::out | std::ios::trunc);

if (!file.is\_open()) {

std::cerr << "警告：无法更新用户文件，但资金已退还！" << std::endl;

return false;

}

// 写入更新后的用户数据

for (User\* user : allUsers) {

file << user->getUsername() << " "

<< user->getBalance() << " "

<< user->getUserType() << " "

<< user->getPassword() << "\n";

}

file.close();

std::cout << "\n× 交易失败！" << std::endl;

std::cout << " 已退还金额: ￥" << totalAmount << std::endl;

std::cout << " 当前余额: ￥" << buyer->getBalance() << std::endl;

return false;

}

}

```

# 总结与展望

5.1 实验总结

1. 成功实现了电商平台的基本功能

2. 掌握了面向对象编程的核心概念

3. 学会了多态性和继承的应用

4. 理解了数据持久化的实现方式

5.2 遇到的问题及解决方案

1. 中文编码问题

- 解决方案：使用UTF-8编码，确保控制台正确显示中文

2. 内存管理问题

- 解决方案：使用智能指针和RAII原则管理内存

3. 数据持久化问题

- 解决方案：使用文件系统进行数据存储

5.3 改进建议

1. 添加数据库支持，提高数据存储效率

2. 实现图形用户界面，提升用户体验

3. 添加商品评价功能

4. 实现订单管理系统

5. 添加支付系统集成

5.4 心得体会

通过本次实验，我深入理解了面向对象编程的思想，学会了如何设计一个完整的系统。在实现过程中，我体会到了代码复用、模块化设计的重要性，也认识到了良好的代码结构和清晰的接口设计对系统维护的重要性。

# 网络版实现说明

8.1 网络架构

8.1.1 整体架构

```

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

| 客户端 | | 服务器 |

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

| Client类 |<--->| Server类 |

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

| |

v v

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

| 业务逻辑层 | | 业务逻辑层 |

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

```

8.1.2 通信流程

- 客户端通过TCP Socket与服务器建立连接

- 使用JSON格式进行数据交换

- 采用请求-响应模式进行通信

8.2 客户端实现

8.2.1 Client类设计

```

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

| Client |

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

| -serverIP\_ |

| -port\_ |

| -clientSocket\_ |

| -connected\_ |

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

| +Connect() |

| +Disconnect() |

| +SendMessage() |

| +Login() |

| +GetProducts() |

| +SearchProducts()|

| +AddToCart() |

| +GetCart() |

| +RemoveFromCart()|

| +CreateOrder() |

| +PayOrder() |

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

```

8.2.2 核心功能实现

1. 连接管理

```cpp

bool Client::Connect() {

clientSocket\_ = socket(AF\_INET, SOCK\_STREAM, 0);

// 配置服务器地址

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(port\_);

inet\_pton(AF\_INET, serverIP\_.c\_str(), &serverAddr.sin\_addr);

// 建立连接

return connect(clientSocket\_, (sockaddr\*)&serverAddr, sizeof(serverAddr)) != SOCKET\_ERROR;

}

```

2. 消息发送

```cpp

Message Client::SendMessage(const Message& msg) {

// 将Message对象转换为JSON

json j;

j["type"] = static\_cast<int>(msg.type);

j["data"] = msg.data;

std::string msgStr = j.dump();

// 发送数据

send(clientSocket\_, msgStr.c\_str(), msgStr.length(), 0);

// 接收响应

char buffer[4096];

int bytesReceived = recv(clientSocket\_, buffer, sizeof(buffer), 0);

// 解析响应

Message response;

json respJson = json::parse(std::string(buffer, bytesReceived));

response.type = static\_cast<MessageType>(respJson["type"].get<int>());

response.data = respJson["data"].get<std::string>();

return response;

}

```

8.3 通信协议

8.3.1 消息类型

```cpp

enum class MessageType {

LOGIN\_REQUEST, // 登录请求

LOGIN\_RESPONSE, // 登录响应

GET\_PRODUCTS, // 获取商品列表

SEARCH\_PRODUCTS, // 搜索商品

ADD\_TO\_CART, // 添加到购物车

GET\_CART, // 获取购物车

REMOVE\_FROM\_CART, // 从购物车移除

CREATE\_ORDER, // 创建订单

PAY\_ORDER // 支付订单

};

```

8.3.2 数据格式

1. 登录请求

```json

{

"type": 0,

"data": {

"username": "用户名",

"password": "密码"

}

}

```

2. 商品操作

```json

{

"type": 2,

"data": {

"keyword": "搜索关键词"

}

}

```

3. 购物车操作

```json

{

"type": 4,

"data": {

"productId": "商品ID",

"quantity": 数量

}

}

```

8.4 安全性考虑

1. 连接安全

- 使用TCP协议保证数据传输可靠性

- 实现连接状态管理

- 异常断开处理

2. 数据安全

- 密码传输加密

- 数据格式验证

- 错误处理机制

3. 会话管理

- 用户登录状态维护

- 会话超时处理

- 并发连接控制

8.5 扩展性设计

1. 模块化设计

- 网络层与业务逻辑分离

- 消息处理模块化

- 协议可扩展

2. 错误处理

- 网络异常处理

- 数据解析异常处理

- 业务逻辑异常处理

3. 性能优化

- 连接池管理

- 数据缓存机制

- 异步处理支持

8.6 服务器端实现

8.6.1 Server类设计

```

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

| Server |

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

| -port\_ |

| -running\_ |

| -serverSocket\_ |

| -clientThreads\_|

| -currentUser |

| -dataHandler\_ |

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

| +Start() |

| +Stop() |

| +HandleClient()|

| +ProcessMessage()|

| +HandleLogin() |

| +HandleGetProducts()|

| +HandleSearchProducts()|

| +HandleAddToCart()|

| +HandleGetCart()|

| +HandleRemoveFromCart()|

| +HandleCreateOrder()|

| +HandlePayOrder()|

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

```

8.6.2 核心功能实现

1. 服务器启动

```cpp

bool Server::Start() {

serverSocket\_ = socket(AF\_INET, SOCK\_STREAM, 0);

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(port\_);

serverAddr.sin\_addr.s\_addr = INADDR\_ANY;

bind(serverSocket\_, (sockaddr\*)&serverAddr, sizeof(serverAddr));

listen(serverSocket\_, SOMAXCONN);

running\_ = true;

while (running\_) {

SOCKET clientSocket = accept(serverSocket\_, nullptr, nullptr);

if (clientSocket != INVALID\_SOCKET) {

clientThreads\_.emplace\_back(&Server::HandleClient, this, clientSocket);

}

}

return true;

}

```

2. 客户端处理

```cpp

void Server::HandleClient(SOCKET clientSocket) {

char buffer[8096];

while (running\_) {

int bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);

if (bytesReceived <= 0) break;

std::string msgStr(buffer, bytesReceived);

Message receivedMsg;

json j = json::parse(msgStr);

receivedMsg.type = static\_cast<MessageType>(j["type"].get<int>());

receivedMsg.data = j["data"].get<std::string>();

Message response = ProcessMessage(receivedMsg, clientSocket);

json jResponse;

jResponse["type"] = response.type;

jResponse["data"] = response.data;

send(clientSocket, jResponse.dump().c\_str(), jResponse.dump().length(), 0);

}

closesocket(clientSocket);

}

```

3. 消息处理

```cpp

Message Server::ProcessMessage(const Message& msg, SOCKET clientSocket) {

Message response;

switch (msg.type) {

case MessageType::LOGIN\_REQUEST:

response.data = HandleLogin(msg.data);

response.type = MessageType::LOGIN\_RESPONSE;

break;

case MessageType::GET\_PRODUCTS:

response.data = HandleGetProducts();

response.type = MessageType::PRODUCTS\_RESPONSE;

break;

// ... 其他消息类型处理

}

return response;

}

```

8.7 服务器端业务处理

8.7.1 用户认证

```cpp

std::string Server::HandleLogin(const std::string& data) {

json request = json::parse(data);

std::string username = request["username"];

std::string password = request["password"];

User\* user = dataHandler\_.loginUser(username, password);

if (user != nullptr) {

json response;

response["status"] = "success";

response["username"] = user->getUsername();

response["userType"] = user->getUserType();

response["balance"] = user->getBalance();

currentUser = user;

return response.dump();

}

return json{

{"status", "error"},

{"message", "用户名或密码错误"}

}.dump();

}

```

8.7.2 商品管理

```cpp

std::string Server::HandleGetProducts() {

std::vector<Product\*> products = dataHandler\_.ShowProduct();

json response;

if (products.empty()) {

response["status"] = "error";

response["message"] = "没有商品信息";

} else {

response["status"] = "success";

for (const auto& product : products) {

json productJson;

productJson["id"] = product->getId();

productJson["name"] = product->getName();

productJson["description"] = product->getDescription();

productJson["price"] = product->getPrice();

productJson["quantity"] = product->getQuantity();

productJson["ownername"] = product->getOwnername();

response["products"].push\_back(productJson);

}

}

return response.dump();

}

```

8.7.3 购物车操作

```cpp

std::string Server::HandleAddToCart(const std::string& data) {

json cartData = json::parse(data);

std::string productIdStr = cartData["productId"];

int quantity = cartData["quantity"];

Product\* product = dataHandler\_.getProductById(std::stoi(productIdStr));

if (product != nullptr) {

std::string username = currentUser->getUsername();

ShoppingCart cart(username);

cart.addItem(product->getName(), quantity,

product->getOwnername(), product->getPrice());

return json{

{"status", "success"},

{"message", "Product added to cart successfully"}

}.dump();

}

return json{

{"status", "error"},

{"message", "未找到该商品"}

}.dump();

}

```

8.8 并发处理

8.8.1 多线程管理

- 使用std::thread管理客户端连接

- 每个客户端连接在独立线程中处理

- 使用vector存储线程对象

8.8.2 线程安全

- 使用互斥锁保护共享资源

- 避免数据竞争

- 正确处理线程同步

8.8.3 资源管理

- 正确关闭socket连接

- 及时清理线程资源

- 处理异常情况