面向对象程序设计实践（C++）课程实验报告

姓名：郎波 学号：2023211048 班级：2023211314

目录

1.前言

代码仓库

https://github.com/QU4RTZ444/project

任务概述

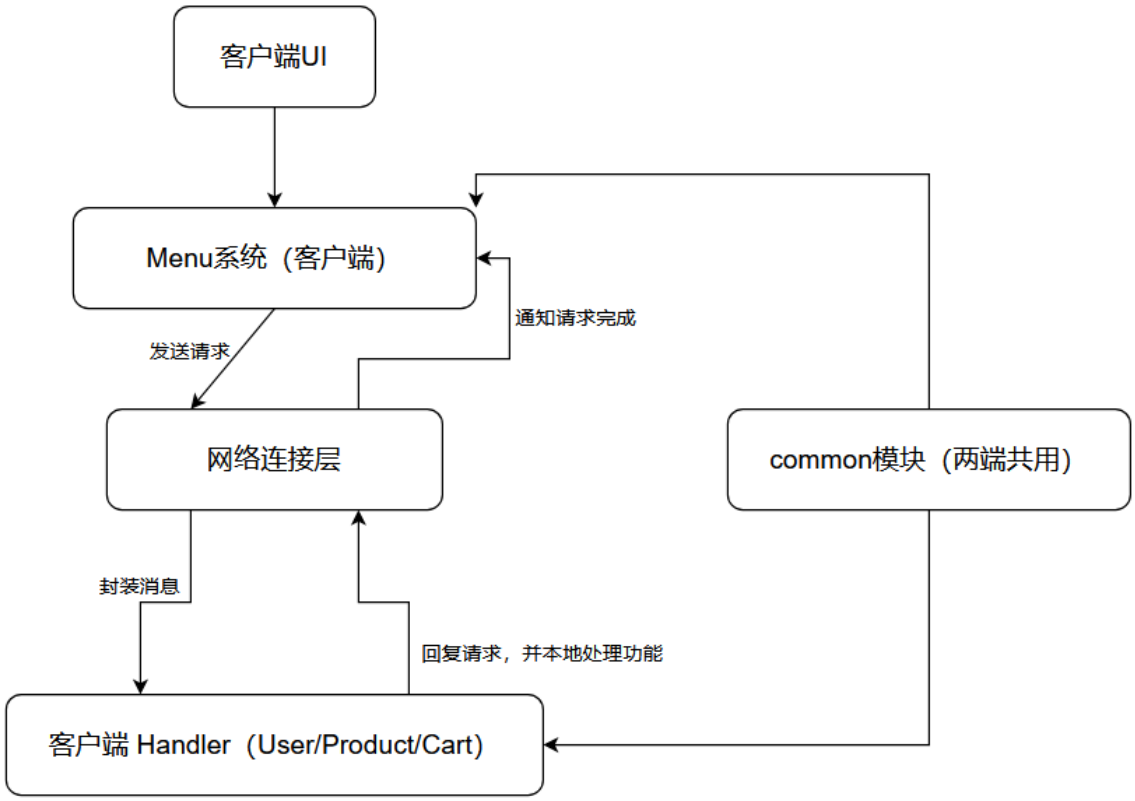
随着移动互联网普及，电商平台成为连接消费者与商家的重要载体。本实验要求使用 C++，采用面向对象思想，分三阶段逐步实现一个简化的电商交易平台：

- 阶段一（单机版）：实现账户管理与商品管理。
- 阶段二（单机版）：在阶段一基础上增加购物车、订单生成与支付流程。
- 阶段三（网络版）：采用 C/S 架构，使用 Socket 完成客户端—服务器通信，支持远程登录、浏览、下单与支付。

个人理解

- 三阶段要求功能递增但代码隔离，因此必须保持清晰的模块边界与接口；
- 错误处理、文件/数据库持久化、折扣策略、并发通信均为综合难点；
- 设计阶段需先绘制系统用例 & 类图，再分阶段演进，实现最小可行版本。
- 难点在于第三阶段的需求分离，需要第二阶段保持清晰的类划分和功能分模块，对于高耦合的类需要解耦合，否则无法下手

2.总体设计



以上为整体模块的示意图，项目大体划分为3个文件夹，server，client，common。 common目录下存放了一些全局通用的类，例如用户，商品等，还有网络工具和一些ui工具等。server文件夹存放了服务端需要做的功能，几种管理器，用于处理对应的请求，整体能够做到接受各种类型的请求消息，根据收到的消息调用对应的处理函数。client文件夹内部存放了客户端本地的界面显示，输入输出的处理函数还有对应各个功能的函数，负责向服务端发送请求以及接受对应的请求。

3.详细设计

3.1.用户管理子系统的详细设计

使用的类： User，Consumer，Merchant，UserManager。其中Consumer, Merchant类继承了User类，User类包含了一些公共的基本功能，修改密码等，还有通用的基本信息，密码，余额，账户名等 对于派生类重写了基类中的信息展示和标志能否购买或销售的函数，声明如下（仅包含重要逻辑，没有完整包含全部代码）：

```
// 抽象用户基类
class User {
protected:
    std::string username;     // 用户名
    std::string password;     // 密码
    double balance;          // 账户余额
    UserType userType;       // 用户类型

public:
    // 基本功能
    const std::string& getUsername() const { return username; }
    bool verifyPassword(const std::string& pwd) const { return password == pwd; }
    double getBalance() const { return balance; }
    void setBalance(double newBalance) { balance = newBalance; }
```

```
    UserType getUserType() const { return userType; }

    // 密码管理
    bool changePassword(const std::string& oldPassword, const std::string&
newPassword);
    void setPassword(const std::string& newPassword) { password = newPassword; }

    // 虚函数
    virtual void displayInfo() const = 0;
    virtual bool canSell() const = 0;
    virtual bool canBuy() const = 0;
};

// 消费者类
class Consumer : public User {
public:
    // 实现纯虚函数
    void displayInfo() const override {
        std::cout << "用户类型: 消费者\n用户名: " << username << "\n余额: " <<
balance << std::endl;
    }

    bool canSell() const override { return false; }
    bool canBuy() const override { return true; }
};

// 商家类
class Merchant : public User {
public:
    // 实现纯虚函数
    void displayInfo() const override {
        std::cout << "用户类型: 商家\n用户名: " << username << "\n余额: " << balance
<< std::endl;
    }

    bool canSell() const override { return true; }
    bool canBuy() const override { return true; }
};
```

关于UserManager类，其功能就是处理用户相关的逻辑，例如登录，注册，验证以及内容的保存等。下面来展开说一些函数的实现逻辑。 首先是用户注册，需要保证不出现用户名重复，具体方法是每次启动时会读取所有的用户信息，注册时会遍历所有的用户然后对比用户名是否存在重复，正确验证后就开始继续读取密码，并读入一些基本信息给新用户，用户输入时还要选择账号类型，完成所有的输入后向文件写入所有的信息，对于网络版需要加锁防止出现读写的冲突，同时客户端会先发送注册的请求，在客户端读取完所有的输入后发送给服务端，由服务端来完成逻辑的处理，接下来其他的函数的机制也是类似的之后不再赘述。 用户登录验证逻辑很简单，比对输入的密码即可，本次任务第三阶段由于时间关系以及个人计算机系统问题（后面会补充）对代码的平台进行了迁移因此代码差距较大，所以第三阶段对密码采用了明文处理，前两个阶段的密码是简单的哈希加密。 接下来是用户基本信息修改的相关功能包括修改密码和余额修改，UserManager类对于修改密码做的事是验证原始密码是否正确然后调用User类自身的修改密码功能，余额修改也是类似，这里还封装了UpdateUser方法用于更新用户信息，可以调用防止信息错误。 下面是类的定义：

```
class UserManager {
private:
    std::vector<std::unique_ptr<User>> users;
    std::string filename;
    std::mutex usersMutex;

    void loadUsers();
    void saveUsers();

public:
    UserManager(const std::string& filename);
    ~UserManager();

    bool registerUser(const std::string& username, const std::string& password,
UserType userType);
    User* authenticateUser(const std::string& username, const std::string&
password);
    bool userExists(const std::string& username);
    bool updateUser(const User& user);
    bool changePassword(const std::string& username, const std::string&
oldPassword, const std::string& newPassword);
};
```

对于这些类之间的关系已经比较清晰，不必赘述

3.2.商品管理子系统的详细设计

使用的类： Product, Clothing, Food, ProductManager。其中Clothing, Food类继承了Product类，Product类包含了一些公共的基本功能，商品名称，价格，库存，折扣等，功能主要有Getter，Setter，库存管理，声明如下（仅包含重要逻辑，没有完整包含全部代码，派生类仅展示一个，因为内容类似）：

```
class Product {
protected:
    int productId;
    std::string name;
    double price;              // 原价
    int stock;                 // 库存
    std::string merchantName;  // 商家名称
    double discount;           // 折扣
    int frozenStock;           // 冻结库存

public:
    // Getter方法
    int getProductId() const { return productId; }
    std::string getName() const { return name; }
    double getOriginalPrice() const { return price; }  // 获取原价
    virtual double getPrice() const;                   // 获取现价（考虑折扣）
    int getStock() const { return stock; }
    std::string getMerchantName() const { return merchantName; }
    double getDiscount() const { return discount; }
```

```
    int getFrozenStock() const { return frozenStock; } // 获取冻结库存
    virtual std::string getProductType() const = 0;

    // Setter方法
    void setPrice(double newPrice) { price = newPrice; }
    void setStock(int newStock) { stock = newStock; }
    void setDiscount(double newDiscount);

    // 库存管理
    bool reduceStock(int quantity);
    bool increaseStock(int quantity);

    // 冻结库存管理（为订单管理预留）
    bool freezeStock(int quantity);
    bool unfreezeStock(int quantity)

    // 为交易功能预留：检查是否可购买指定数量
    bool isAvailable(int quantity) const;

    // 检查是否有折扣
    bool hasDiscount() const { return discount < 1.0; }
};

/**
 * @brief 食品类
 * 继承自Product基类
 */
class Food : public Product {
public:
    Food(int id, const std::string& name, double price, int stock,
         const std::string& merchant, double discount = 1.0);

    /**
     * @brief 实现纯虚函数 - 返回商品类型
     * @return "食品"
     */
    std::string getProductType() const override { return "食品"; }

    /**
     * @brief 重写价格计算（食品可能有特殊折扣规则）
     * @return 实际价格
     */
    double getPrice() const override;
};
```

对于ProductManager类，其功能就是处理商品相关的逻辑，例如添加商品，删除商品，修改商品信息等。下面来展开说一些函数的实现逻辑。其中使用了ProductInfo结构体来传输商品信息，这样可以用于简化传输客户端无需依赖完整的Product类，同时仅传输结构体效率高，安全性也高，客户端只能做读取数据，需要修改信息再去通过服务器提供的接口。

```
struct ProductInfo {
    int id;
    std::string name;
    double price;
    int stock;
    std::string merchant;
    double discount;
};
```

ProductManager负责商品的完整生命周期管理。关键的设计特点如下：

- 工厂模式：createProduct()方法根据type参数创建不同类型的商品对象
- 线程安全：使用mutex保护共享数据，支持多客户端并发访问
- 持久化：自动加载和保存商品数据到文件
- 分页支持：支持大量商品的分页查询，提升性能
- 商家隔离：提供按商家查询的接口，支持商家管理自己的商品
- ID自动生成：维护nextProductId确保商品ID唯一性

对于库存冻结的机制在第三阶段还能做到处理并发订单处理，有效防止订单超额售卖，流程如下：

- 下单时：freezeStock() 冻结对应数量
- 支付成功：reduceStock() 真正扣减库存
- 取消订单：unfreezeStock() 释放冻结库存 整体大致代码如下：

```
class ProductManager {
private:
    std::vector<std::unique_ptr<Product>> products;
    std::string filename;// 文件名用于持久化存储
    mutable std::mutex productsMutex;
    int nextProductId;// 下一个商品ID，用于自动生成唯一ID

    void loadProducts();
    void saveProducts();
    void saveProductsToFile();

    std::unique_ptr<Product> createProduct(const std::string& type, int id, const
std::string& name,
     double price, int stock, const std::string& merchant, double discount = 1.0);

public:

    bool addProduct(const std::string& type, const std::string& name,
                    double price, int stock, const std::string& merchantName,
                    double discount = 1.0);

    bool modifyProduct(int productId, double newPrice = -1, int newStock = -1,
double newDiscount = -1);

    int setDiscountByType(const std::string& productType, double discount);
```

```
    // 修改返回类型：使用ProductInfo结构体代替Product对象
    std::vector<ProductInfo> getAllProducts() const;
    std::vector<ProductInfo> getProductsByPage(int page, int pageSize) const;
    std::vector<ProductInfo> searchProducts(const std::string& keyword) const;
    std::vector<ProductInfo> getProductsByType(const std::string& type) const;

    // 商家专用查询
    std::vector<ProductInfo> getProductsByMerchant(const std::string&
merchantName) const;
    std::vector<ProductInfo> getMerchantProductsByPage(const std::string&
merchantName, int page, int pageSize) const;
    int getMerchantProductCount(const std::string& merchantName) const;

    // 返回指针用于修改操作
    Product* getProductById(int productId);

    size_t getProductCount() const;
    int getTotalPages(int pageSize) const;// 获取总页数,用于翻页功能
};
```

商品数据采用了文本存储，格式为：

```
商品ID|商品类型|商品名称|原价|库存|商家名称|折扣
```

每次启动时加载全部商品到内存，修改时实时保存到文件，兼顾了查询性能和数据安全性。

3.3.订单管理子系统的详细设计

使用的类： Cart, CartManager, order相关的逻辑整合在Server类中直接处理，client.h中有OrderInfo, OrderItemInfo, CartItemInfo这些结构体用于为订单管理功能服务，CartItem由于不需要复杂的功能直接写成了结构体使用。 对于Cart类，包含了用户名称还有一个商品列表，商品列表使用了一个map来存储商品ID和CartItem结构体，方便快速查找和修改。整体效果就是一个用户对应一个购物车，购物车能够计算金钱等还包含了添加商品，删除商品，修改商品数量等功能。而CartItem结构体则包含了商品ID，商品名称，商品价格，商品数量等信息。两者的代码如下：

```
struct CartItem {
    int productId;           // 商品ID
    std::string productName; // 商品名称
    std::string productType; // 商品类型
    double originalPrice;    // 商品原价
    double currentPrice;     // 商品现价（考虑折扣）
    int quantity;           // 购买数量
    std::string merchantName; // 商家名称
    double discount;        // 折扣
    // 计算该项目的总价
    double getTotalPrice() const {return currentPrice * quantity;}
};

class Cart {
```

```
private:
    std::string username;
    std::map<int, CartItem> items; // productId -> CartItem

public:
    // 添加商品到购物车
    bool addItem(const CartItem& item);

    // 更新商品数量
    bool updateItemQuantity(int productId, int quantity);

    // 移除商品
    bool removeItem(int productId);

    // 清空购物车
    void clear();

    // 获取购物车中的所有商品
    std::vector<CartItem> getItems() const;

    // 获取购物车中商品的总数量
    int getTotalItemCount() const;

    // 获取购物车总价
    double getTotalPrice() const;

    // 检查购物车是否为空
    bool isEmpty() const;

    // 获取用户名
    std::string getUsername() const { return username; }
};
```

关于CartManager类由以下几点直接总结：

1. **多用户管理**：使用map<string, Cart>管理所有用户的购物车，以用户名为键实现用户隔离
2. **线程安全**：使用mutex保护共享数据，支持多客户端并发访问购物车
3. **持久化存储**：
    - 构造时自动加载历史购物车数据
    - 每次修改后自动保存到文件
    - 析构时确保数据完整保存
4. **统一接口**：为服务端提供统一的购物车操作接口，屏蔽底层实现细节
5. **懒加载机制**：用户购物车在首次访问时自动创建，避免内存浪费

代码展示：

```
class CartManager {
private:
    std::map<std::string, Cart> userCarts; // username -> Cart
    mutable std::mutex cartsMutex;
    std::string filename;

    void loadCarts();
    void saveCarts();

public:
    CartManager(const std::string& filename);
    ~CartManager();

    // 添加商品到用户购物车
    bool addItemToCart(const std::string& username, const CartItem& item);

    // 更新购物车中商品数量
    bool updateCartItem(const std::string& username, int productId, int quantity);

    // 从购物车移除商品
    bool removeItemFromCart(const std::string& username, int productId);

    // 清空用户购物车
    bool clearUserCart(const std::string& username);

    // 获取用户购物车
    Cart getUserCart(const std::string& username) const;

    // 获取用户购物车中的商品列表
    std::vector<CartItem> getUserCartItems(const std::string& username) const;

    // 获取用户购物车总价
    double getUserCartTotalPrice(const std::string& username) const;

    // 获取用户购物车商品数量
    int getUserCartItemCount(const std::string& username) const;
};
```

关于订单生成部分，这里直接从client还有server的对应部分抽离出相关代码来展示：

```
// 订单项目信息结构体
struct OrderItemInfo {
    int productId;
    std::string productName;
    std::string productType;
    double originalPrice;
    double currentPrice;
    int quantity;
    std::string merchantName;
    double discount;
```

```
    double getTotalPrice() const {
        return currentPrice * quantity;
    }
};

// 订单信息结构体
struct OrderInfo {
    int orderId;
    std::string orderTime;
    double totalAmount;
    std::string status;
    std::vector<OrderItemInfo> items;
};
```

使用结构体的原因不再次说明，下面还有创建订单的函数:

```
void Server::handleOrderCheckoutRequest(SOCKET clientSocket, const std::string&
data);//订单结算
```

对于订单结算的处理函数，主要逻辑是： 首先进行用户身份验证，检查用户是否已登录且为消费者类型，然后获取用户购物车内容并验证购物车非空。接下来计算订单总价并检查用户余额是否充足。

库存验证阶段会遍历购物车中的每个商品，检查商品是否存在以及库存是否充足。如果任何商品库存不足，会自动回滚之前已扣除的库存操作，确保数据一致性。通过库存验证后，实际扣除各商品库存并记录操作历史。

同时进行收入分配计算，按商家统计每个商家应得的收入金额和对应的商品清单。完成库存扣减后，从消费者账户扣除相应余额。

订单文件生成阶段，系统会生成唯一的订单ID和时间戳，然后为消费者创建个人订单文件记录购买信息，包括订单基本信息和商品明细。同时为每个涉及的商家分别创建订单文件，记录来自该订单的销售收入和商品信息。

最后进行数据持久化，清空用户购物车，保存所有用户和商品数据到文件，并向客户端返回订单创建成功的响应信息。

这种设计通过原子性操作保证了数据一致性，使用双文件记录机制让消费者和商家都能查看各自相关的订单信息，收入按商家自动分配并实时记录到对应文件中。订单文件采用统一格式，使用分隔符便于后续解析和查看。

3.4.Common目录下的工具类

在common目录下存放了一些全局通用的类，例如网络工具类，UI工具类等。下面是一些重要的工具类的设计。 Utils类采用静态方法设计，提供系统级的通用功能:

```
class Utils {
public:
    static void clearScreen();          // 清屏功能，跨平台兼容
    static void pauseScreen();          // 暂停等待用户输入
```

```
    static void showSeparator(const std::string& title = "");  // 显示分隔线
    static std::string formatMoney(double amount);  // 格式化金额显示
    static std::string getInput(const std::string& prompt);  // 带提示的输入
    static void showSuccess(const std::string& message);  // 成功消息
    static void showError(const std::string& message);    // 错误消息
    static void showInfo(const std::string& message);     // 信息消息
};
```

这些功能主要用于改善用户界面体验，提供统一的输出格式和交互方式。静态方法设计使得在任何地方都能方便调用，无需创建对象实例。关于网络部分放至3.6节说明。

3.5.数据库说明

本次实验第一二阶段使用了sqlite3作为数据库，第三阶段由于对代码进行了迁移使用了文本存储数据。数据库部分直接展示第二阶段的数据库设计，第一阶段的数据库设计与第二阶段相同，主要是增加了订单相关的表。

用户表（users）

```
CREATE TABLE IF NOT EXISTS users (
    username TEXT PRIMARY KEY,
    password_hash TEXT NOT NULL,
    user_type TEXT NOT NULL CHECK(user_type IN ('Consumer', 'Seller')),
    balance REAL DEFAULT 0.0 CHECK(balance >= 0)
);
```

- 用户名作为主键，确保唯一性
- 密码采用哈希存储，提高安全性
- 用户类型限制为消费者和商家两种
- 余额设置非负约束，防止账户透支

商品表（products）

```
CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    category TEXT NOT NULL CHECK(category IN ('图书', '食品', '服装')),
    description TEXT,
    price REAL NOT NULL CHECK(price >= 0),
    quantity INTEGER NOT NULL CHECK(quantity >= 0),
    seller_username TEXT,
    discount_rate REAL DEFAULT 1.0 CHECK(discount_rate > 0 AND discount_rate <=
1),
    FOREIGN KEY(seller_username) REFERENCES users(username)
);
```

- 自增ID作为主键，便于管理和引用
- 商品分类限定为图书、食品、服装三类

- 价格和库存量设置非负约束
- 折扣率限制在0-1之间，1表示无折扣
- 通过外键关联商家用户

购物车表（cart_items）

```
CREATE TABLE IF NOT EXISTS cart_items (
    username TEXT NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL CHECK(quantity > 0),
    PRIMARY KEY (username, product_id),
    FOREIGN KEY(username) REFERENCES users(username),
    FOREIGN KEY(product_id) REFERENCES products(id)
);
```

- 采用复合主键（用户名+商品ID），确保每个用户对每个商品只有一条记录
- 数量必须为正数
- 通过外键保证数据完整性

订单表（orders）

```
CREATE TABLE IF NOT EXISTS orders (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    buyer_username TEXT NOT NULL,
    total_amount REAL NOT NULL CHECK(total_amount > 0),
    status TEXT NOT NULL CHECK(status IN ('pending', 'paid', 'cancelled',
'failed')),
    create_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(buyer_username) REFERENCES users(username)
);
```

- 自增订单ID，便于订单追踪
- 订单状态包含待支付、已支付、已取消、支付失败四种状态
- 自动记录订单创建时间
- 总金额必须为正数

订单项目表（order_items）

```
CREATE TABLE IF NOT EXISTS order_items (
    order_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL CHECK(quantity > 0),
    price REAL NOT NULL CHECK(price >= 0),
    seller_username TEXT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY(order_id) REFERENCES orders(id),
    FOREIGN KEY(product_id) REFERENCES products(id),
```

```
    FOREIGN KEY(seller_username) REFERENCES users(username)
);
```

- 记录订单中每个商品的详细信息
- 保存下单时的商品价格，避免价格变动影响订单
- 记录商家信息，便于收入分配

商品库存锁定表（product_locks）

```
CREATE TABLE IF NOT EXISTS product_locks (
    product_id INTEGER NOT NULL,
    order_id INTEGER NOT NULL,
    locked_quantity INTEGER NOT NULL CHECK(locked_quantity > 0),
    lock_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (product_id, order_id),
    FOREIGN KEY(product_id) REFERENCES products(id),
    FOREIGN KEY(order_id) REFERENCES orders(id)
);
```

- 实现库存预扣机制，防止超卖
- 记录锁定时间，支持超时释放
- 用于处理订单支付过程中的库存管理

数据库设计特点

1. **完整性约束**：通过CHECK约束和外键约束保证数据的合法性和完整性
2. **并发控制**：通过库存锁定表实现订单处理的并发安全
3. **数据一致性**：订单和订单项目分表存储，支持事务操作
4. **历史记录**：保留订单创建时间和商品价格快照
5. **用户分离**：通过用户类型区分消费者和商家的不同权限

这种设计为第三阶段的网络版本提供了良好的数据结构基础，虽然最终改用文本文件存储，但数据模型保持了相同的逻辑结构。 对于第三阶段的文本存储，数据格式如下（给出运行中产生的例子）：

```
订单ID:1748966597
时间:2025-06-04 00:03:17
总金额:24
状态:已完成
商品明细:
  面包|数量:2|单价:12|小计:24|商家:B
---订单结束---
```

这种格式简单明了，便于后续解析和展示。每个订单以订单ID开头，包含时间、总金额、状态和商品明细，最后以"---订单结束---"标识结束。

```
A|3|1;苹果;食品;8.50;8.50;1;B;1.00|2;牛奶;食品;15.80;14.22;1;B;0.90|6;深入理解计算机系统;书籍;139.00;125.10;1;B;0.90
```

这种格式使用分号分隔每个商品项，商品项内部使用竖线分隔各个字段，便于解析和处理。每个字段的含义与数据库设计保持一致。 对于商品和用户信息的存储采用了二进制的格式存储，包含了相关的基本信息，这里不作展示。

3.6.接口协议说明

第三阶段的网络版本采用了基于TCP Socket的客户端-服务器通信架构，设计了完整的消息协议来处理各种业务请求。

消息协议设计

消息格式： 所有网络消息都采用统一的NetworkMessage结构体封装：

```
struct NetworkMessage {
    MessageType type;     // 消息类型（4字节）
    int length;           // 数据长度（4字节）
    std::string data;     // 消息数据（可变长度）
};
```

序列化格式： 消息在网络传输时按照以下格式序列化为字节流：

```
[消息类型:4字节][数据长度:4字节][消息数据:length字节]
```

消息类型分类

1. 连接管理类

- CONNECT_REQUEST (1) / CONNECT_RESPONSE (2)：建立连接
- DISCONNECT (3)：断开连接

2. 用户管理类

- REGISTER_REQUEST (13) / REGISTER_RESPONSE (14)：用户注册
- LOGIN_REQUEST (15) / LOGIN_RESPONSE (16)：用户登录
- LOGOUT_REQUEST (17) / LOGOUT_RESPONSE (18)：用户登出
- CHANGE_PASSWORD_REQUEST (19) / CHANGE_PASSWORD_RESPONSE (20)：修改密码

3. 商品浏览类

- PRODUCT_LIST_REQUEST (30) / PRODUCT_LIST_RESPONSE (31)：商品列表查看
- PRODUCT_SEARCH_REQUEST (32) / PRODUCT_SEARCH_RESPONSE (33)：商品搜索

4. 商家管理类

- MERCHANT_ADD_PRODUCT_REQUEST (36) / MERCHANT_ADD_PRODUCT_RESPONSE (37)：添加商品
- MERCHANT_MODIFY_PRODUCT_REQUEST (38) / MERCHANT_MODIFY_PRODUCT_RESPONSE (39)：修改商品

5. 购物车管理类

- CART_ADD_ITEM_REQUEST (50) / CART_ADD_ITEM_RESPONSE (51)：添加商品到购物车
- CART_VIEW_REQUEST (52) / CART_VIEW_RESPONSE (53)：查看购物车
- CART_UPDATE_ITEM_REQUEST (54) / CART_UPDATE_ITEM_RESPONSE (55)：更新购物车商品
- CART_REMOVE_ITEM_REQUEST (56) / CART_REMOVE_ITEM_RESPONSE (57)：移除购物车商品
- CART_CLEAR_REQUEST (58) / CART_CLEAR_RESPONSE (59)：清空购物车

6. 订单管理类

- ORDER_CHECKOUT_REQUEST (60) / ORDER_CHECKOUT_RESPONSE (61)：订单结算
- ORDER_LIST_REQUEST (62) / ORDER_LIST_RESPONSE (63)：订单列表查看

数据传输格式

请求数据格式： 不同类型的请求采用管道符(|)分隔的字符串格式：

- 登录请求：`用户名|密码`
- 注册请求：`用户名|密码|用户类型`
- 添加商品到购物车：`商品ID|数量`
- 商家添加商品：`商品类型|商品名称|价格|库存|折扣`

响应数据格式： 响应数据通常以状态码开头，后跟具体数据：

- 成功响应：`SUCCESS|具体数据`
- 错误响应：`ERROR|错误信息`

例如商品列表响应格式：

```
SUCCESS|商品数量|商品ID:商品名称:价格:库存:商家|商品ID:商品名称:价格:库存:商家|...
```

协议特点

1. 请求-响应模式： 每个请求都有对应的响应消息类型，客户端发送请求后等待服务端响应，确保通信的可靠性。

2. 类型安全： 通过枚举类MessageType定义所有消息类型，避免硬编码的魔数，提高代码可维护性。

3. 长度前缀： 消息头包含数据长度信息，解决TCP流式传输的粘包问题，确保消息边界正确识别。

4. 错误处理： 统一的错误响应格式，所有业务错误都通过ERROR前缀标识，便于客户端统一处理。

5. 扩展性： 消息类型采用整数编码，便于后续添加新的消息类型而不影响现有协议。

序列化实现

serialize()方法将NetworkMessage转换为字节流：

```
std::vector<char> NetworkMessage::serialize() const {
    // 按顺序写入：消息类型(4字节) + 数据长度(4字节) + 数据内容
}
```

deserialize()方法从字节流恢复NetworkMessage：

```
static NetworkMessage NetworkMessage::deserialize(const std::vector<char>& buffer)
{
    // 按顺序读取：消息类型 + 数据长度 + 数据内容
}
```

这种设计确保了客户端和服务端能够正确解析和处理各种消息，支持复杂的业务逻辑交互，同时保持了协议的简洁性和可扩展性。

4.实现

1.第一阶段的问题

第一阶段的开发比较顺利，没有特别麻烦的问题，设计好整体功能框架直接搭建即可，使用linux平台进行开发可以较方便使用第三方库（sqlite3），但是需要注意设计时预留接下来开发的空间。例如在查看商品时应该预留添加商品的功能，避免在功能扩展时大规模修改已有的代码，可能会出现新的bug，严重导致项目无法构建。

想法方面：这是第一次使用makefile相关的工具，相比于IDE的报错，直接看编译器给出的报错有时候会比较困难，一些容器的使用错误提示很难读懂需要借助其他工具。同时链接问题一直伴随着项目构建，使用了很多的头文件和源文件，链接时需要注意头文件的顺序以及源文件的链接顺序，否则会出现链接错误。这些错误想要查也非常难查，需要不断摸索。

经验：头文件必须要注意避免循环包含，否则会对项目构建带来非常大的阻碍，在使用linux开发的过程中也认识到了系统接口的差异，一阶段和二阶段的代码中有使用PlatformUtils类统一Windows和Linux的使用，例如密码的读取有使用屏蔽功能（只显示"*"），在Windows下使用了conio.h库的_getch()函数，而在Linux下使用了termios.h库的tcgetattr()和tcsetattr()函数来实现类似功能。对于Linux的使用过程中认识了信号相关的知识，于是也在代码中加入了信号处理，例如按下ctrl+c时能够正常退出程序而不是直接终止。

教训：对于功能的划分需要做的更细，最开始将很多的处理都丢在了menu_handler类中，在添加功能时逻辑混乱，最后拆分非常费劲，同时还要把改动同步到第二阶段，对于版本管理来说是一个很大的挑战，直接merge问题很大，最后选择了添加新分支手动修改结构处理。本来代码中有做跨平台的处理，在windows下构建出现了很大的问题，以后在项目编写时可以选择cmake和vcpkg来做跨平台的处理

2.第二阶段的问题

第二阶段的问题主要是由第一阶段留下的，代码结构不清晰导致难以下手，然后第二阶段又会给第三阶段的需求分离埋坑。在代码拆分完成后功能的实现就很容易了，添加新的数据表和新的类即可完成。

想法方面：在版本管理时面对merge能够更好处理了，结构改动多时可以选择手动merge。

经验：这次开发中添加新功能会有一些金钱没有正确同步之类的bug，直接查也没法简单看出，这时候结合vscode的调试功能还有添加调试打印信息进行分析就能很好定位错误，同时也认识到了应该构建release版本和debug版本，通过宏来进行调试信息的打印能够避免release版本中调试信息影响使用

教训：与第一阶段类似，一定要做好需求分块，不然在需求分离时会有很多麻烦

3.第三阶段的问题

第三阶段遇见的问题最麻烦，首先是前面提到的需求分离，仅仅基于第二阶段的成果继续添加网络功能就很容易出现新功能不知道加在哪的问题。我在最开始构建了基本的网络工具后连接了客户端和服务端，构建项目发现一直给我抛出空指针的错误，通过核心转储文件我定位了错误来源，最后发现由于最初实现时耦合度没有降下来，导致有些功能还是在本地处理，最后就会抛出空指针。苦恼了很久以后，重新编写自己的linux环境又出现问题，vscode远程连接组件无法正常连接，只能使用命令行，但是代码量很大使用vim不太现实，只能把代码迁移到windows上进行开发，这时候网络工具需要重构，还需要使用ide熟悉两个目标文件的构建，最后由于时间关系选择一部分妥协简化了实现。例如弃用了数据库，密码也没有继续处理等。但是这一次推进的方式更加合理，先构建双端的连接，再逐步添加功能，最后成功完成构建。

想法方面：这种代码量较大的项目架构是最重要的，遇到代码量大的文件时就应该思考功能是否应该拆分，编写时就需要注意降低接口的耦合度。

经验：这一次具体认识到了程序段错误崩溃后生成的核心转储文件是如何辅助错误定位的，最开始尝试程序内打印堆栈，但是效果不好。调试过程中也加强了分析堆栈调用的能力。

教训：整个任务规模比以前接触的内容大，算是认识到了C++工程的相关知识，也深刻认识到了C++这方面知识的难度和深度。我应该去强化一些相关工具的使用，学习更多相关知识。代码编写方面应该注重整体架构，合理的架构让开发不会过于凌乱，在功能迭代时能更加得心应手。