面向对象程序设计与实践（C++）：电商交易平台设计与实现实验报告

毛子恒
2019211397
北京邮电大学 计算机学院

日期：2021 年 7 月 2 日

1 概览

1.1 任务描述

使用 C++ 语言，基于面向对象的程序设计方法，设计并实现一个简单的电商交易平台，提供用户管理、商品管理、交易管理等功能。

1.2 开发环境

- macOS Big Sur 11.3
- Apple clang version 12.0.5
- cmake version 3.19.1
- QMake version 3.1
- Clion 2021.1.1
- Qt Creator 4.14.2
- Visual Studio Code 1.56.2

2 功能需求

用户管理和商品管理功能

- 用户注册和登录：支持新用户注册平台账号，已注册用户用平台账号登录平台，要求已注册用户的信息长久保留。
- 修改账户密码：支持登录后对用户账号的密码修改。
- 余额管理：支持用户账号中余额的查询、充值、消费等。
- 添加商品：支持商家添加新商品，要求已添加的商品信息长久保留。
- 展示平台商品信息：支持针对不同类型用户、无论登录与否均展示平台商品信息。
- 搜索平台商品信息：支持依据某种条件对平台商品进行筛选，并展示筛选结果。
- 商品信息管理：支持商家对其商品的信息进行管理。

- 账户至少需要账号名、密码、账户余额、账户类型（商家/消费者）等内容。
- 把所有的用户账户信息写入文件（要求使用文件存储各类信息），注册新账户的时候，要求注册的账户名不能已经存在于文件中，即账户名唯一。
- 至少设计一层继承体系（用户基类-用户子类）。设计一个用户基类，然后让商家类和消费者类等用户子类继承它，具体的用户是用户子类的实例对象。用户基类为抽象类，不能实例化，至少具有一个纯虚函数 `getUserType()` 用于指示用户类别。
- 电商平台上至少有三类商品，每类商品中至少有三个具体的商品，每个具体的商品至少包含商品描述、商品原价、商品剩余量等数据。所有的商品信息需要存储在文件或数据库中，不能写在代码中。
- 至少设计一层继承体系（商品基类-商品子类）。设计一个商品基类，然后让商品子类继承它，具体的商品是商品子类的实例对象。商品基类请至少具有一个虚函数 `getPrice()` 用于计算具体商品的价格。
- 支持对同一品类下所有产品打折的活动。
- 支持一定的错误场景处理能力。

交易管理功能

- 购物车管理：支持消费者向购物车添加、删除指定数量的商品，也支持消费者修改当前购物车商品的拟购买数量。
- 订单生产：选择购物车的商品生成订单，计算并显示订单总金额。
- 网上支付：消费者使用余额支付订单，支付成功后，消费者被扣除的余额应转至商家余额中。
- 当订单生成后，处于未支付状态时，应将对应数量的商品冻结，不能参与新订单的产生，避免商品被超额售卖。
- 支持一定的错误场景处理能力。

网络版交易平台功能

- 设计网络版交易平台客户端。
- 要求采用 C/S 结构，客户端与服务器系统之间使用 socket 进行通信。
- 支持一定的错误场景处理能力。

其他附加功能

- 客户端支持美观的 GUI。
- 采用 SQLite 数据库维护用户、商品和订单信息。
- 客户端和服务端采用 `json` 格式通信，并且采用JWT鉴权。

3 模块介绍

3.1 模块划分

各模块及其关系如图 1。

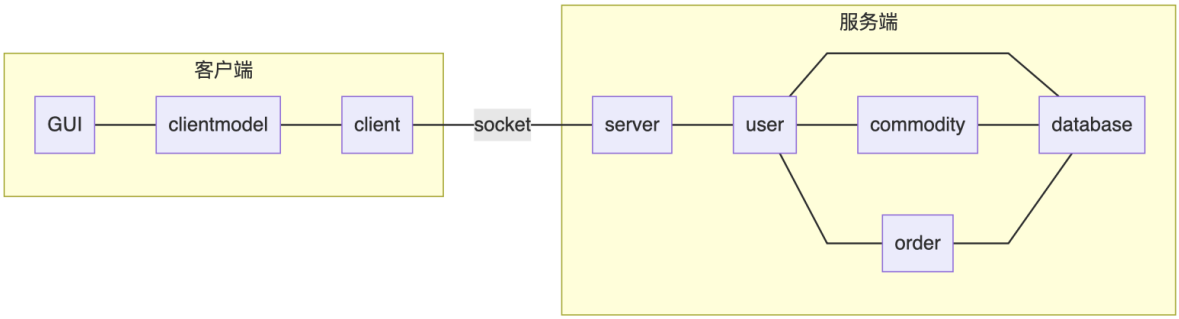


图 1: 模块关系图

电商管理平台采用 C/S 架构，有客户端和服务端两部分组成。客户端部分，GUI 采用 QML 实现，`clientmodel` 模块为集成到 QML 的数据结构，其中定义了一些用于信息展示的类和可供调用的接口。`client` 模块负责生成报文，并且通过 socket 与服务端通信。

服务端部分，`server` 模块与客户端通信、解析报文和生成回复报文、调用相应的处理函数；`user` 模块负责处理用户鉴权，以及用户管理操作；`commodity` 模块负责处理商品管理操作；`order` 模块负责处理订单管理操作；`database` 负责数据库管理，包括表创建、信息增删和查询等操作。

3.2 服务端

根据功能需求，服务端会对如下 25 种请求进行响应：

1. 用户登录。
2. 用户注册。
3. 用户登出。
4. 获取用户信息。
5. 修改用户密码。
6. 增加用户余额。
7. 减少用户余额。
8. 获取所有商品信息。
9. 获取本用户的商品信息。
10. 获取商品详情。
11. 通过筛选条件获取商品信息。
12. 设置商品信息。
13. 添加商品。
14. 删除商品。

15. 商品打折。
16. 获取购物车信息。
17. 添加商品到购物车。
18. 修改购物车中商品数量。
19. 从购物车中删除商品。
20. 添加订单。
21. 获取本用户的订单。
22. 获取订单详情。
23. 取消订单。
24. 支付订单。
25. 退款/确认退款订单。

服务端的功能主要在 `Server`、`UserManage`、`CommodityManage`、`OrderManage` 和 `Database` 类中实现，这些类的关系如图 2所示。

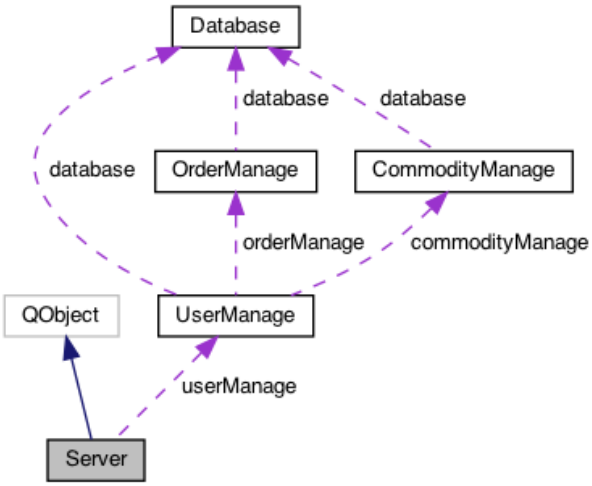


图 2: 服务端类关系图

3.2.1 用户管理

请求 1 至 7 属于用户管理功能，在 `UserManage` 类中实现。该类中实现的功能除了用户注册、登录、登出和用户信息管理之外，还有用户鉴权功能，因此该类中也定义了其他所有请求的接口，在进行鉴权后再调用其他类中相应的处理函数。此外，对于部分参数复杂的请求，`UserManage` 类负责解析 `json` 格式的请求参数。该类也实现了将回复信息打包成 `json` 格式，以及返回报错信息的工作。

仅举一例，对于以下查询商品信息的请求：

```
1  QString queryCommodity(const QJsonObject &info, QJsonArray &ret) const;
```

查询的参数在 `info` 变量中，查询结果保存在 `ret` 中，函数的返回值为报错信息，如果返回字符串为空，表示操作没有错误。

调用此函数时，首先解析 `info` 中的查询参数，之后根据需要进行鉴权操作，再调用 `CommodityManage` 类中的相应查询方法。

用户注册、登录与登出 用户注册时传入用户名、密码和用户类型，服务端会首先检查用户名是否重复，如果不重复，则会在数据库和另一个文本文件中同时添加一个记录，保存用户信息。

用户登录时传入用户名和密码，服务端进行校验后返回一个凭据，之后进行特定用户的请求时都需要附加这个凭据，凭据中包含用户名，以验证用户身份。凭据采用 JWT 格式，`UserManage` 类中只对 `json` 格式的 Payload 部分进行处理，其余部分的生成和解析由 `Server` 类负责，将在稍后说明。

用户类采用用户基类-用户子类继承体系，其主要结构如下：

```
class User
{
protected:
    int balance;

public:
    enum USER_TYPE
    {
        MERCHANT, CUSTOMER
    };

    User(int _balance) : balance(_balance) {}
    virtual USER_TYPE getUserType() const = 0;
    int getBalance() const { return balance; }
    void addBalance(int num) { balance += num; }
    virtual ~User() = default;
};

class Customer : public User
{
public:
    Customer(int _balance) : User(_balance) {}
    USER_TYPE getUserType() const override { return CUSTOMER; };
};

class Merchant : public User
{
public:
    Merchant(int _balance) : User(_balance) {}
    USER_TYPE getUserType() const override { return MERCHANT; };
};
```

`User` 类为用户基类，`Customer` 和 `Merchant` 类为 `User` 类的子类，分别表示消费者和商家。消费者类中的购物车功能被省略，将在稍后说明。

用户象在登录时被创建，登出时被销毁，为了支持多用户的同时登录，`UserManage` 类中保

存用户名向用户对象指针的映射，形如 `QMap<QString, QSharedPointer<User>> map`。由于访问 `User` 对象的唯一方式就是通过用户名映射到对应的对象，所以 `User` 类中不需要保存用户名的成员。所有用户的信息保存在数据库中，当用户登录时，从数据库中取出用户信息，以此创建用户对象。

对数据库的管理操作在 `Database` 类中实现。数据库采用 SQLite，利用 `QSqlDatabase` 类，实现对数据库文件的增删和修改操作。此外，为了满足功能需求中文件相关的要求，另外增加一个保存用户名的文件，用于检验用户名的唯一性。

利用 `QSqlQuery` 类执行 SQL 语句，同时输出调试信息。

数据库中 `user` 表的格式如下：

```
username TEXT PRIMARY KEY NOT NULL
passwd TEXT NOT NULL
type INT NOT NULL
balance INT NOT NULL
```

分别表示用户名、密码、类型、余额。由于涉及到精度问题，余额采用整数表示，以分为单位。

用户信息查询与修改 获取用户信息、修改密码、增减余额操作都需要在用户已登录的状态下进行，所以需要用凭据鉴权后执行操作。修改密码会直接修改数据库中的信息，增减余额操作会同时对用户对象中的余额变量和数据库中的余额信息进行修改。

3.2.2 商品管理

请求 8 至 15 属于商品管理功能，在 `CommodityManage` 类中实现，其中，除了请求 8、10、11、15 之外，其余操作都需要鉴权。

商品添加 商品添加请求只能由商家用户发起。商品信息保存在数据库中，`commodity` 表的格式如下：

```
id INT PRIMARY KEY NOT NULL
name TEXT NOT NULL
intro TEXT NOT NULL
type INT NOT NULL
belong TEXT NOT NULL
price INT NOT NULL
amount INT NOT NULL
```

分别表示商品 ID、名称、介绍、类型、所属商家、价格、数量。价格也以分为单位。与用户不同，通过商品 ID 来区分不同的商品，ID 在创建新商品时自动分配，商品所属商家即为发起创建商品操作的用户。

商品信息修改 商品信息修改请求只能由商家用户发起，并且只能修改此用户拥有的商品。可以修改商品的名称、介绍、价格和数量。直接从数据库中修改该商品的表项。

商品删除 商品删除请求只能由商家用户发起，并且只能删除此用户拥有的商品。删除时直接从数据库中删除该商品的表项。

商品信息查询 商品信息查询有四种情况：查询所有商品、按照筛选条件搜索商品、查询某个商品的详情、查找某个用户拥有的商品。前三种情况不需要鉴权，最后一种情况需要鉴权。

其中筛选条件包括：商品名关键词、商品类型、商品最小价格和最大价格、是否有货。

商品类采用商品基类-商品子类继承体系，其主要结构如下：

```
class Commodity
{
protected:
    int id;
    QString name;
    QString intro;
    QString belong;
    int price;
    int amount;

public:
    enum Type
    {
        CLOTHES, BOOKS, ELECTRONIC
    };

    Commodity(int _id, QString _name, QString _intro, QString _belong, int
    ↪ _price, int _amount) : id(_id), name(std::move(_name)),
    ↪ intro(std::move(_intro)), belong(std::move(_belong)), price(_price),
    ↪ amount(_amount) {}
    virtual int getCommodityType() const = 0;
    virtual int getDiscount() const = 0;
    int getId() const { return id; }
    const QString &getName() const { return name; }
    const QString &getIntro() const { return intro; }
    const QString &getBelong() const { return belong; }
    int getPrice() const { return price; }
    int getAmount() const { return amount; }
    virtual ~Commodity() = default;
};

class Clothes : public Commodity
{
private:
    static int discount;

public:
```

```
35      Clothes(int _id, QString _name, QString _intro, QString _belong, int
        ↪  _price, int _amount) : Commodity(_id, std::move(_name),
        ↪  std::move(_intro), std::move(_belong), _price, _amount) {}
36      static void setDiscount(int _discount) { discount = _discount; }
37      int getDiscount() const override { return discount; }
38      int getCommodityType() const override { return CLOTHES; }
39  };
40
41  class Books : public Commodity
42  {
43  private:
44      static int discount;
45
46  public:
47      Books(int _id, QString _name, QString _intro, QString _belong, int _price,
        ↪  int _amount) : Commodity(_id, std::move(_name), std::move(_intro),
        ↪  std::move(_belong), _price, _amount) {}
48      static void setDiscount(int _discount) { discount = _discount; }
49      int getDiscount() const override { return discount; }
50      int getCommodityType() const override { return BOOKS; }
51  };
52
53  class Electronic : public Commodity
54  {
55  private:
56      static int discount;
57
58  public:
59      Electronic(int _id, QString _name, QString _intro, QString _belong, int
        ↪  _price, int _amount) : Commodity(_id, std::move(_name),
        ↪  std::move(_intro), std::move(_belong), _price, _amount) {}
60      static void setDiscount(int _discount) { discount = _discount; }
61      int getDiscount() const override { return discount; }
62      int getCommodityType() const override { return ELECTRONIC; }
63  };
```

`Commodity` 类为商品基类，`Clothes`、`Books` 和 `Electronic` 类为服饰、书籍和电子产品商品子类。

由于商品信息保存在数据库中，在数据库中查询到商品后创建商品对象（或者商品对象的列表），这些对象用于在函数之间传递商品信息，在查询操作完成后，商品对象被销毁。

商品打折 商品打折操作可以由任何人发起，支持对某一类的所有商品打折，打折的倍率存储在对应类的一个静态成员中，由于精度的限制，由整数存储，默认为 100，表示原价。

3.2.3 购物车管理

请求 16 至 19 属于购物车管理功能，所有操作都需要鉴权。仅有消费者用户有购物车，购物车在消费者用户对象中保存，当用户登出时，购物车的内容会和用户对象一起销毁。

包含购物车功能的消费者类的定义如下：

```
class Customer : public User
{
private:
    QMap<int, int> cart;

public:
    Customer(int _balance) : User(_balance) {}
    USER_TYPE getUserType() const override { return CUSTOMER; };
    int getItemNum(int id) const;
    void addToCart(int id, int number);
    void removeFromCart(int id, int number);
    void eraseFromCart(int id);
    const QMap<int, int> &getCart() const { return cart; }
};
```

添加商品到购物车和修改商品数量时，如果数量上限/下限超出允许的范围，则会数量会自动被约束到对应的范围，不会产生报错。

3.2.4 订单管理

请求 20 至 25 属于订单管理功能，在 `OrderManage` 类中实现，所有操作都需要鉴权。

订单创建 订单创建请求只能由消费者用户发起。消费者选取一些商品 ID，服务端首先判断 ID 是否在购物车内，如果在，则根据购物车中该商品的数量创建订单，从将对应商品的数量减去购物车中商品的数量。

订单信息保存在数据库中，`orders` 表的格式如下：

```
id INT PRIMARY KEY NOT NULL
mainid INT NOT NULL
status INT NOT NULL
total INT NOT NULL
buyer TEXT NOT NULL
seller TEXT NOT NULL
content TEXT NOT NULL
```

分别表示订单 ID、主订单 ID、订单状态、总额、买家用户名、买家用户名、订单内容。

系统中实现的订单包含一个 ID 和主订单 ID，当消费者创建订单时，会根据订单中商品归属的商家，分成多个子订单，一个子订单中的商品都属于同一个商家，这些子订单的共同构成一个主订单。主订单的 ID 并不连续，由子订单的最小 ID 决定。

订单有五种状态：未支付、已取消、已支付、正在退款、已退款。默认创建订单时，订单处于未支付状态。

订单内容为商品 ID、商品数量二元组的列表，由于生成订单后订单内容就不发生变化，所以在数据库中以字符串保存，格式为“`商品 ID，商品数量；商品 ID，商品数量；...`”。

订单信息查询 订单信息查询有两种情况：查询某个用户的订单、查询某个订单的详情。查询用户订单时，只会返回订单 ID 和订单总额。查询订单详情时，会根据用户类型区分，如果用户为消费者，则按主订单 ID 查询，否则按子订单 ID 查询，以保证用户只能查询到与自己有关的订单信息。

订单类的主要结构如下：

```
class Order
{
public:
    enum Status
    {
        UNPAID, CANCELLED, PAID, REFUNDING, REFUNDED
    };

private:
    int id;
    int mainId;
    Status status;
    int total;
    QString buyer;
    QString seller;
    QList<QPair<int, int>> content;

public:
    Order(int _id, int _mainId, int _status, int _total, QString _buyer,
    ↪  QString _seller, const QString &_content);
    int getId() const { return id; }
    int getMainId() const { return mainId; }
    int getStatus() const { return status; }
    int getTotal() const { return total; }
    const QString &getBuyer() const { return buyer; }
    const QString &getSeller() const { return seller; }
    const QList<QPair<int, int>> &getContent() const { return content; }
};
```

与商品类类似，订单类在数据库查询后创建，并且在操作完成后销毁。

订单取消 该请求只能由消费者发起，将未支付的订单改为已取消。

订单支付 该请求只能由消费者发起，首先确认消费者余额充足，从消费者用户余额中扣除订单对应的金额，并且将未支付的订单改为已支付。

订单退款 该请求只能由消费者发起，将已支付的订单改为退款中。

确认订单退款 该请求只能由商家发起，首先确认商家余额充足，并且订单中的商品仍然存在，之后从商家用户余额中扣除子订单对应的金额，将订单中的商品返还给商家，将退款中的订单

改为已退款。

3.2.5 网络通信与报文解析

`Server` 类的功能是通过 UDP 协议与客户端通信，将收到的报文解析为 `json` 格式，并且将返回的 `json` 格式报文转化为字节流。

网络通信 服务器将 socket 绑定值本地的某个端口，并且监听此端口，当有报文到达时调用 `messageHandler` 处理函数。

报文解析 收到 UDP 报文为字节流，需要将其转化成 `json` 格式，报文的 `type` 字段指明了请求的类型，`payload` 字段指明了请求的内容，服务端根据请求的类型调用相应的处理函数，例如，对于登录请求，调用 `userLoginHandler` 函数。

JWT 鉴权 JWT 分为三个部分，分别为 Header、Payload 和 Signature，用户在登录时，如果登录成功，`UserManage` 类的函数会返回 `json` 格式的 Payload 部分，`Server` 类的 `jwtEncoding` 函数需要负责加上 Header 部分，利用密钥生成 Signature 部分，将这三部分通过 Base64 编码后连接成完整的 JWT 字符串，附在回复报文中。

之后，每当收到的请求需要鉴权，就在请求的内容中查找 `token` 字段，利用 `jwtVerify` 函数验证 Signature 是否正确，再利用利用 `jwtGetPayload` 函数并且将 `body` 转化为 `json` 格式，完成鉴权。

请求处理和回复报文构建 对于每种请求，服务端都需要从请求的内容中提取出需要的信息，进行（可选的）鉴权，之后调用 `UserManage` 类中相应的函数。

`UserManage` 类中的函数返回值为 `QString` 类型，如果操作没有产生错误，则该字符串为空，否则该字符串的内容是报错信息。`Server` 根据请求的不同类型构造回复报文，报文的 `status` 字段表示请求是否成功，`payload` 字段根据请求的类型给出请求的结果。

构造后的回复报文为 `json` 格式，需要转化成字节流，之后通过原本的 UDP 连接发送回客户端。

3.3 客户端

3.3.1 网络通信与报文解析

`Client` 类负责通过指定端口和服务器通信，该类保存服务端生成的凭据，将其附加在报文中，并且设置 `type` 字段，最后将报文转换成字节流。

发送报文后，客户端监听 socket 直至回复报文到达。收到报文后，客户端转换报文为 `json` 格式。

3.3.2 客户端数据结构

客户端中定义了三个数据结构 `UserModel`、`CommodityModel` 和 `OrderModel`，用于集成到 QML 中，展示数据。

此外，`ClientModel` 类定义了供 QML 调用的接口，包含3.2节中规定的 25 个操作，以及可供 QML 访问的数据成员。如下：

```
class UserModel : public QObject
{
Q_OBJECT
    Q_PROPERTY(QString username READ username CONSTANT)
    Q_PROPERTY(int type READ type CONSTANT)
    Q_PROPERTY(int balance READ balance CONSTANT)

public:
    UserModel(QString _username, int _type, int _balance) :
    ↪ m_username(std::move(_username)), m_type(_type), m_balance(_balance)
    ↪ {}
    QString username() const { return m_username; }
    int type() const { return m_type; }
    int balance() const { return m_balance; }

private:
    QString m_username;
    int m_type;
    int m_balance;
};

class CommodityModel : public QObject
{
Q_OBJECT
    Q_PROPERTY(int id READ id CONSTANT)
    Q_PROPERTY(QString name READ name CONSTANT)
    Q_PROPERTY(QString intro READ intro CONSTANT)
    Q_PROPERTY(int type READ type CONSTANT)
    Q_PROPERTY(int price READ price CONSTANT)
    Q_PROPERTY(int discount READ discount CONSTANT)
    Q_PROPERTY(int amount READ amount CONSTANT)
    Q_PROPERTY(int order READ order WRITE setOrder NOTIFY orderChanged)

public:
    CommodityModel(int _id, QString _name, QString _intro, int _type, int
    ↪ _price, int _discount, int _amount) : m_id(_id),
    ↪ m_name(std::move(_name)), m_intro(std::move(_intro)), m_type(_type),
    ↪ m_price(_price), m_discount(_discount), m_amount(_amount) {}
    CommodityModel(int _id, QString _name, int _type, int _price, int
    ↪ _discount, int _amount) : m_id(_id), m_name(std::move(_name)),
    ↪ m_type(_type), m_price(_price), m_discount(_discount),
    ↪ m_amount(_amount), m_order(0) {}
```

```
35      CommodityModel(int _id, QString _name, int _type, int _amount, int _order)
        ↪  : m_id(_id), m_name(std::move(_name)), m_type(_type),
        ↪  m_amount(_amount), m_order(_order) {}
36      int id() const { return m_id; }
37      QString name() const { return m_name; }
38      QString intro() const { return m_intro; }
39      int type() const { return m_type; }
40      int price() const { return m_price; }
41      int discount() const { return m_discount; }
42      int amount() const { return m_amount; }
43      int order() const { return m_order; }
44      void setOrder(int _order)
45      {
46          m_order = _order;
47          emit orderChanged();
48      }
49
50  Q_SIGNALS:
51      void orderChanged();
52
53  private:
54      int m_id;
55      QString m_name;
56      QString m_intro;
57      int m_type;
58      int m_price;
59      int m_discount;
60      int m_amount;
61      int m_order;
62  };
63
64  class OrderModel : public QObject
65  {
66  Q_OBJECT
67      Q_PROPERTY(int id READ id CONSTANT)
68      Q_PROPERTY(int mainid READ mainid CONSTANT)
69      Q_PROPERTY(int status READ status CONSTANT)
70      Q_PROPERTY(int total READ total CONSTANT)
71      Q_PROPERTY(QString buyer READ buyer CONSTANT)
72      Q_PROPERTY(QString seller READ seller CONSTANT)
73
74  public:
75      OrderModel(int _id, int _mainid, int _status, int _total, QString _buyer,
        ↪  QString _seller) : m_id(_id), m_mainid(_mainid), m_status(_status),
        ↪  m_total(_total), m_buyer(std::move(_buyer)),
        ↪  m_seller(std::move(_seller)) {}
76      OrderModel(int _id, int _status, int _total) : m_id(_id),
        ↪  m_status(_status), m_total(_total) {}
77      OrderModel(int _id, int _total) : m_id(_id), m_total(_total) {}
78      int id() const { return m_id; }
79      int mainid() const { return m_mainid; }
```

```
80      int status() const { return m_status; }
81      int total() const { return m_total; }
82      QString buyer() const { return m_buyer; }
83      QString seller() const { return m_seller; }
84
85  private:
86      int m_id;
87      int m_mainid;
88      int m_status;
89      int m_total;
90      QString m_buyer;
91      QString m_seller;
92  };
93
94  class ClientModel : public QObject
95  {
96  Q_OBJECT
97      Q_PROPERTY(bool userActive READ userActive NOTIFY userActiveChanged)
98      Q_PROPERTY(UserModel * user READ user NOTIFY userChanged)
99      Q_PROPERTY(QList<CommodityModel *> commodity READ commodity NOTIFY
        ↪ commodityChanged)
100     Q_PROPERTY(int commoditySize READ commoditySize NOTIFY
        ↪ commoditySizeChanged)
101     Q_PROPERTY(CommodityModel * singleCommodity READ singleCommodity NOTIFY
        ↪ singleCommodityChanged)
102     Q_PROPERTY(QList<OrderModel *> order READ order NOTIFY orderChanged)
103     Q_PROPERTY(OrderModel * mainOrder READ mainOrder NOTIFY mainOrderChanged)
104
105 public:
106     ClientModel(QObject * parent, quint16 _port) : QObject(parent),
        ↪ m_userActive(0), m_user(nullptr), m_commoditySize(0),
        ↪ m_singleCommodity(nullptr), m_mainOrder(nullptr), client(this, _port)
        ↪ {}
107     virtual ~ClientModel() = default;
108     bool userActive() const { return m_userActive; }
109     UserModel * user() const { return m_user; }
110     QList<CommodityModel *> commodity() const { return m_commodity; }
111     int commoditySize() const { return m_commoditySize; }
112     CommodityModel * singleCommodity() const { return m_singleCommodity; }
113     QList<OrderModel *> order() const { return m_order; }
114     OrderModel * mainOrder() const { return m_mainOrder; }
115
116     Q_INVOKABLE QString userLogin(QString username, QString password);
117     Q_INVOKABLE QString userRegister(QString username, QString password, int
        ↪ type);
118     Q_INVOKABLE void userLogout();
119     Q_INVOKABLE QString userGetInfo();
120     Q_INVOKABLE QString userSetPasswd(QString password);
121     Q_INVOKABLE QString userAddMoney(QString money);
122     Q_INVOKABLE QString userSubMoney(QString money);
123     Q_INVOKABLE QString itemGetAll();
```

```
124     Q_INVOKABLE QString itemGetMine();
125     Q_INVOKABLE QString itemGetDetail(int id);
126     Q_INVOKABLE QString itemGetFilter(QString name, int type, QString
        ↪ min_price, QString max_price, int remain);
127     Q_INVOKABLE QString itemSet(int id, QString name, QString intro, QString
        ↪ price, QString amount);
128     Q_INVOKABLE QString itemAdd(QString name, QString intro, int type, QString
        ↪ price, QString amount);
129     Q_INVOKABLE QString itemDelete(int id);
130     Q_INVOKABLE QString itemDiscount(int type, QString discount);
131     Q_INVOKABLE QString cartGet();
132     Q_INVOKABLE QString cartAdd(int id, QString number);
133     Q_INVOKABLE QString cartSet(int id, QString number);
134     Q_INVOKABLE QString cartDelete(int id);
135     Q_INVOKABLE QString orderAdd();
136     Q_INVOKABLE QString orderGetMine();
137     Q_INVOKABLE QString orderGetDetail(int id);
138     Q_INVOKABLE QString orderCancel(int id);
139     Q_INVOKABLE QString orderPay(int id);
140     Q_INVOKABLE QString orderRefund(int id);
141
142 Q_SIGNALS:
143     void userChanged();
144     void userActiveChanged();
145     void commodityChanged();
146     void singleCommodityChanged();
147     void commoditySizeChanged();
148     void orderChanged();
149     void mainOrderChanged();
150
151 private:
152     static QString checkStatus(const QJsonObject &ret);
153     bool m_userActive;
154     UserModel * m_user;
155     QList<CommodityModel *> m_commodity;
156     int m_commoditySize;
157     void clearCommodity();
158     CommodityModel * m_singleCommodity;
159     QList<OrderModel *> m_order;
160     void clearOrder();
161     OrderModel * m_mainOrder;
162     Client client;
163 };
```

`Q_PROPERTY` 指明的属性可以直接在 QML 中访问，`READ` 后指明了访问的方法名，当变量的值发生变化时，会激活 `NOTIFY` 后指明的信号。仅以 `singleCommodity` 对象为例：

```
1 Q_PROPERTY(CommodityModel * singleCommodity READ singleCommodity NOTIFY
  ↪ singleCommodityChanged)
2 CommodityModel * singleCommodity() const { return m_singleCommodity; }
```

```
3   void singleCommodityChanged();
4   CommodityModel * m_singleCommodity;
```

该属性对应 `m_singleCommodity` 私有成员，可以通过 `singleCommodity` 函数访问，当变量的值发生变化时，激活 `singleCommodityChanged` 信号。

`Q_INVOKABLE` 指明的方法可以直接在 QML 中调用。

`UserModel` 类中有用户名、类型、余额属性。`CommodityModel` 类中有商品 ID、名称、介绍、类型、价格、折扣、数量、所属订单属性。其中 `order` 属性在订单详情页用于展示商品所属的子订单 ID，在购物车页用于表示商品是否选中。`OrderModel` 类中有订单 ID、主订单 ID、订单状态、总额、买家、卖家属性。

`ClientModel` 中有一个 `UserModel` 对象 `user`，一个 `CommodityModel` 对象的列表 `commodity` 和一个单独的 `CommodityModel` 对象 `singleCommodity`，也有一个 `OrderModel` 对象的列表 `order` 和一个单独的 `OrderModel` 对象 `mainOrder`。`commodity` 用于展示商品列表，`singleCommodity` 用于商品详情页，`order` 用于订单列表，`mainOrder` 用于订单详情页。

3.3.3 可视化界面

可视化界面采用 QML 实现，其中所有的图标采用来自flaticon的免费 icon，使得界面风格现代化。另外，客户端内置开源的思源宋体作为字体。

界面采用 `StackView` 布局，即存在一个页面栈，当访问新页面时向栈中 `push` 一个页面，退出该页面时将该页面从栈中 `pop`。默认栈中只有一个主页页面。

我实现了如下 12 个页面：

1. 主页
2. 登录页
3. 用户页
4. 添加商品页
5. 修改商品页
6. 商品详情页
7. 搜索页
8. 搜索结果页
9. 购物车页
10. 打折页
11. 订单列表页
12. 订单详情页

此外，我还实现了一些供复用的文本、图标、输入框和对话框，提供了统一的视觉风格。

界面中的元素大部分都是网格化布局和相对定位结合，以满足在窗口大小可动态调整的基础上，在任何状态下都能够提供正确的显示效果。界面中的商品列表和订单列表均采用 `ScrollView` 和 `ListView` 配合，实现变长的可滚动的列表。

可视化界面的具体成果见4.3节的说明。

4 用户指南

4.1 安装依赖

所需环境：Qt。

选择系统对应的在线安装包，安装 Qt 5.15.2 和 Qt Creator 即可。

可以通过设置镜像源的方法加快下载速度。

4.2 编译项目

4.2.1 命令行编译

在 `server` 目录中执行以下命令来编译（以 macOS 为例）：

```
mkdir build
cd build
qmake ./server.pro -spec macx-clang CONFIG+=x86_64 CONFIG+=qtquickcompiler
make qmake_all
make
```

或者采用 CMake 来编译，修改 `CMakeLists.txt` 文件中 `CMAKE_PREFIX_PATH` 变量的前缀为本机 Qt 目录的路径，之后在 `server` 目录中执行以下命令：

```
mkdir build
cd build
cmake ..
make
```

在 `client` 目录中执行以下命令来编译（以 macOS 为例）：

```
mkdir build
cd build
qmake ./client.pro -spec macx-clang CONFIG+=x86_64 CONFIG+=qtquickcompiler
make qmake_all
make
```

4.2.2 在 Qt Creator 中编译

使用 Qt Creator 打开项目文件 `server.pro`，进入“项目”界面，在“编辑构建配置”后的下拉菜单中选择“Release”，设置“Build directory”的后缀为 `Phase3/server/build`。

之后使用 Release 模式构建并运行项目。

客户端操作类似。

4.3 使用手册

4.3.1 主页

启动服务端和客户端后，进入主页，如图 3所示。

图 3: 主页

主页上方四个图标分别是搜索页、订单页、购物车页和用户页的入口，中间为所有商品的列表，点击图标或者商品名称可以进入商品详情页。左下角为打折页的入口。

4.3.2 登录页

默认未登录情况下，在主页点击订单、购物车和用户页图标都会进入登录页，如图 4(a)所示。

登录页中间为用户名和密码输入框，点击左边的图标即可登录，如果登录失败，会弹出报错弹窗，如图 4(b)所示。如果登录成功，会自动退出登录页。

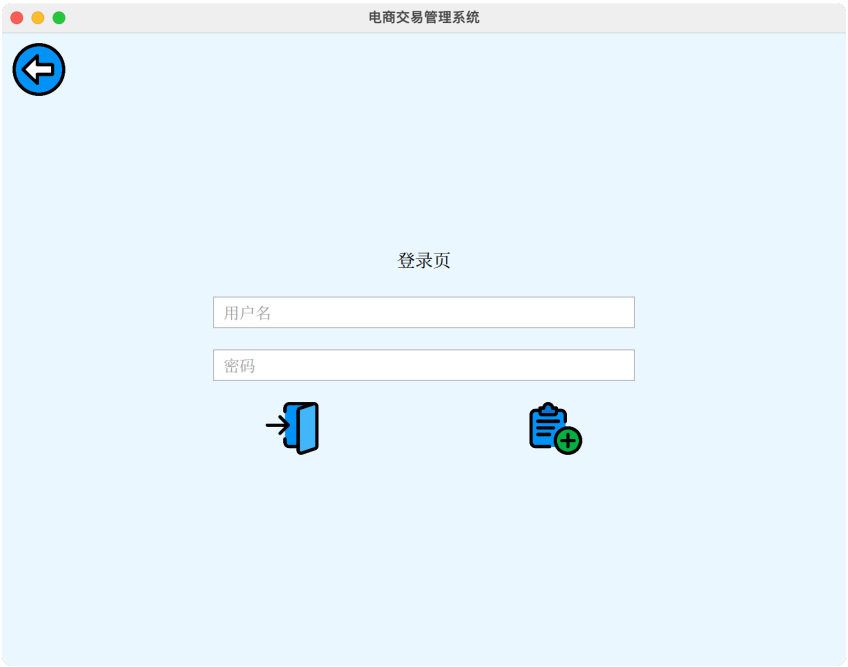
点击右边的图标即可注册，会弹出选择用户类型的弹窗，如图 4(c)所示。如果注册成功，会弹出注册成功弹窗，如图 4(d)所示。

4.3.3 用户页

登录成功后，在主页点击用户图标会进入用户页，根据用户类型的不同，用户页的显示内容会有所不同。

商家用户页如图 5(a)所示，页面上方显示用户信息，余额右边有增减余额的按钮，余额下方两个按钮分别表示修改密码和登出。页面下方显示商家拥有的商品，其中有增加商品、修改商品和删除商品的按钮。

消费者用户页如图 5(b)所示，其上半部分和商家用户页一致。



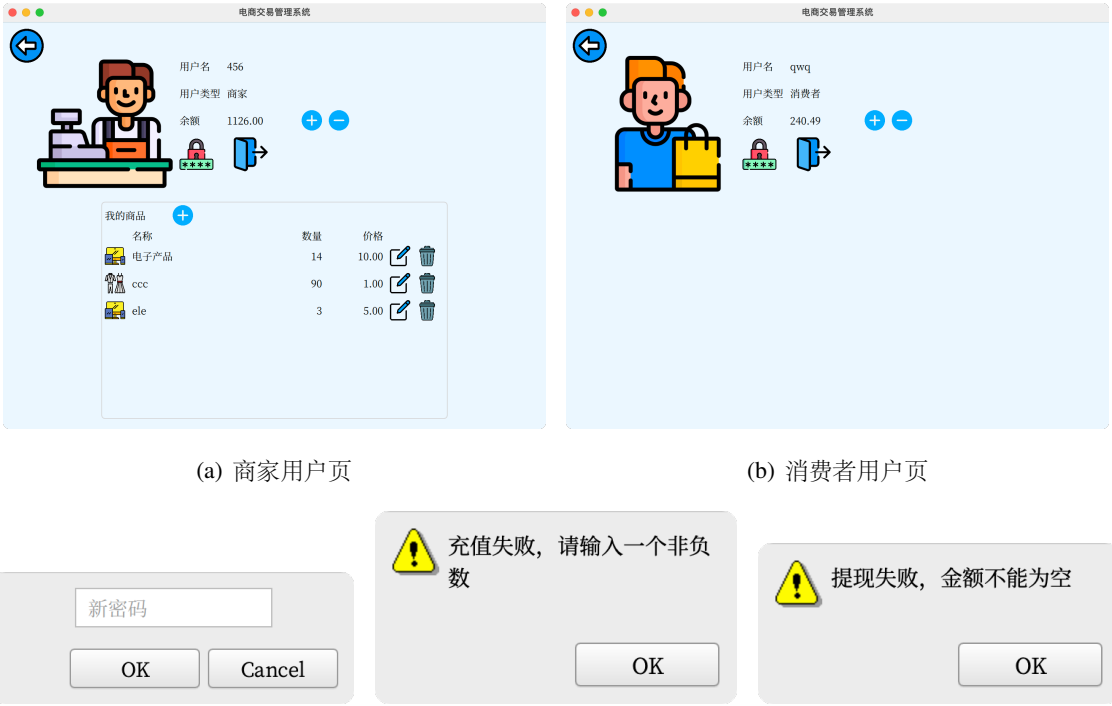
(a) 登录页

(b) 登录失败弹窗

(c) 选择用户类型弹窗

(d) 注册成功弹窗

图 4: 登录页



(a) 商家用户页　　(b) 消费者用户页

(c) 修改密码弹窗　　(d) 充值失败弹窗　　(e) 提现失败弹窗

图 5: 用户页

点击修改密码按钮，会弹出如**图 5(c)**所示的弹窗，输入新密码后点击确认，会弹出提示弹窗。

点击增/减余额按钮，会弹出类似的弹窗，在框中输入正整数后点击确认即可修改余额。如果修改成功，则页面上的余额会自动刷新。如果修改失败，会弹出如**图 5(d)**和**图 5(e)**的弹窗，显示错误提示信息。

点击登出按钮，用户会登出，并且自动退出用户页。

4.3.4 添加商品页

在用户页点击“我的商品”右方的 + 图标，进入添加商品页，如**图 6**所示。

在各输入框中输入商品信息，并且通过下拉菜单选择商品类型，点击蓝色对勾确认添加，如果添加成功，将自动退出本页面，并且新的商品会显示在用户页中，如果失败则会弹出报错信息。

点击红色叉取消添加。

4.3.5 修改商品页

在用户页的商品列表中点击编辑图标，即进入修改商品页，如**图 7**所示。

在该页面可以修改商品的名称、介绍、价格和数量，如果某个输入框留空表示不修改该属性。点击蓝色对勾确认修改，如果修改成功，将自动退出本页面，如果失败则会弹出报错信息。

点击红色叉取消修改。

图 6: 添加商品页

图 7: 修改商品页

4.3.6 商品详情页

在任何商品列表中点击商品图标或商品名称即可进入商品详情页，如图 **8**所示。

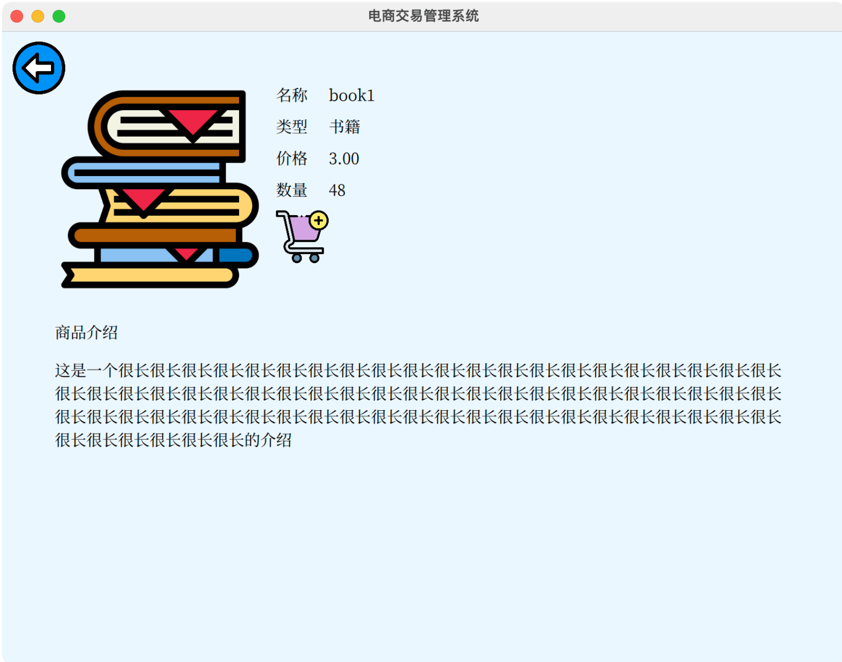


图 8: 商品详情页

该页面中主要显示商品的各项信息，另有一个添加至购物车按钮，点击后弹出弹窗，在框中输入需要添加的商品的数量，如果添加成功，则弹出成功提示窗口，否则弹出报错信息窗口。

4.3.7 搜索页

在主页中点击搜索按钮，进入搜索页，如图 **9**所示。

可以在输入框中输入名称、价格区间等筛选条件，也可以通过下拉菜单选择商品类型，或者选择忽略无货的商品。如果输入框为空，则表示不设该筛选条件。点击下方的搜索按钮进行搜索。

4.3.8 搜索结果页

在搜索页点击搜索按钮后，如果输入无误，则跳转到搜索结果页，如图 **10**所示，其中包含一个商品列表，即搜索的结果。

4.3.9 购物车页

在首页点击购物车按钮，即进入购物车页，如图 **11**所示，其中包含从商品详情页添加进购物车的商品。

购物车页中间为一个商品列表，每个商品前有一个复选框，表示是否选中该商品，选中的商品的总金额会在右下角动态显示。每个商品右边有修改商品数量和移出购物车按钮。

图 9: 搜索页

图 10: 搜索结果页

图 11: 购物车页

当点击修改商品数量按钮时，会弹出弹窗，在框中输入想要修改的数量，之后点击确认即可确认修改，如果修改成功，购物车页会显示新的商品数量，如果修改失败，则会弹出错误信息弹窗。

当点击移出购物车按钮时，会弹出确认弹窗，确认移出后，该商品会从商品列表中消失。

点击右下角的下单按钮，会弹出确认弹窗，确认下单后，则会用选中的商品创建订单，并且选中的商品会被移出购物车。

如果商品的实际数量少于购物车中的数量，则会按实际数量下单，如果购物车中的商品不存在，则订单中不会有该商品。

4.3.10 打折页

在主页中点击左下角的打折按钮，即进入打折页，如**图 12**所示。

打折页中包含一个商品类型的下拉菜单和一个输入框，在输入框中输入折扣信息，点击红色对勾即可开始打折。如果成功，则会自动退出本页面，否则会弹出错误信息弹窗。

4.3.11 订单列表页

在主页中点击订单按钮，即进入订单列表页，如**图 13**所示。

订单列表页中只显示与本用户有关的订单 ID 和订单金额。

4.3.12 订单详情页

在订单列表页中点击订单 ID 或者图标，即进入订单详情页，如**图 14**所示。

页面右上方显示主订单和子订单的有关信息，当主订单处于未支付状态时，显示支付和取消图标。

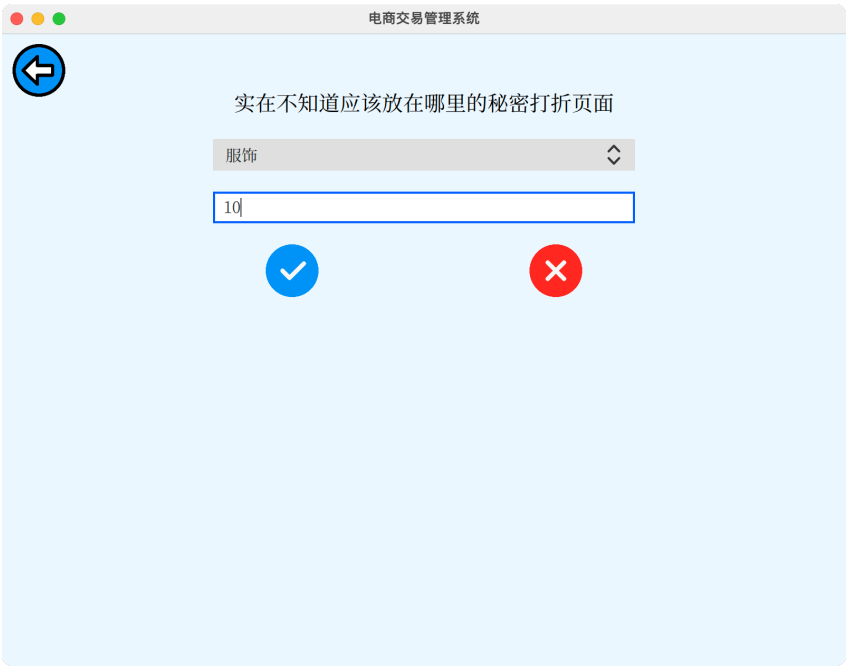


图 12: 打折页

图 13: 订单页

图 14: 订单详情页

页面下方显示订单中的商品信息。

点击支付图标，弹出确认弹窗，确认后即支付此订单，如果支付失败，则弹出错误信息，如**图 15(a)**所示。否则订单状态被刷新，如**图 15(c)**所示。

点击取消订单图标，弹出确认弹窗，确认后即取消此订单，如果取消失败，则弹出错误信息，否则订单状态被刷新，如**图 15(b)**所示。

订单支付完成后，如果点击子订单列表中的退款按钮，弹出确认弹窗，确认后即对该子订单发起退款申请，如果申请退款失败，则弹出错误信息，否则订单状态被刷新，如**图 15(d)**所示。

切换到商家账户，商家在该订单页面只能看到与自己有关的内容，不能看到其他商家的内容，如**图 15(e)**所示。点击子订单列表中的确认退款按钮，弹出确认弹窗，确认后即确认退款该子订单，如果退款失败，则弹出错误信息，否则订单状态被刷新，如**图 15(f)**所示。

5 测试结果

测试过程分为两步，首先对所有实现的功能进行测试，流程和4.3节的流程相同。之后对异常情况进行测试，考虑到的所有异常情况如下：

用户输入不合法 在客户端中，用户只能通过给定的输入框输入，在输入框中输入不合法的字符，包括在需要输入数字的地方输入字符、在需要输入整数的地方输入小数、输入数字的范围超出限制或输入字符串的长度超过限制等，测试结果显示所有的错误输入都能被正确处理。

数据库和用户名文件不同步 如前所述，用户信息会被同时写入到数据库和文本文件中，当二者不同步时，错误能被正确处理。

(a) 订单支付错误信息

(b) 已取消的订单

(c) 已支付的订单

(d) 正在退款的订单

(e) 商家的订单详情页

(f) 已退款的订单

图 15: 订单详情页

SQL 语句执行错误　通过人为修改数据库文件导致 SQL 语句执行错误，该错误能被正确处理。

UDP 报文错误　通过使用 Python 发送错误的 UDP 报文测试此类错误，包括 UDP 报文不完全、报文格式不正确、报文内容错误、凭据错误、非法修改权限外的值，这些错误都能被正确处理。

订单相关错误　包括添加进购物车的商品不存在/数量不足、支付时余额不足、退款时余额不足、退款时商品不存在等错误，这些错误都能被正确处理。

客户端和服务端通信错误　当客户端发出请求报文后，会被阻塞住并等待来自服务端的回复，当服务端下线时，客户端会停止在当前页面并且无法进行任何操作。

6　实验总结

6.1　实现的难点、遇到的问题和解决方案

调试信息输出　采用 Qt 内置的调试信息输出函数，将调试信息分为五个等级 Debug、Info、Warning、Critical、Fatal。在执行各个函数和关键操作之前都需要输出调试信息，UDP 传送的报文内容也需要输出。此外，在服务端的 `main.cpp` 中实现了一个函数 `messageHandler`，用于将调试信息格式化打印，并且给文字加上颜色。

在 `Database` 类中涉及到 SQLite 语句，为了对语句进行分析和调试，我实现了 `execDebug` 函数，在执行 SQLite 语句的同时打印语句的内容，当出现语句执行错误的时候可以快速检查出问题。

调试信息输出的效果如图 16。

```
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/user.cpp:82  验证 "qwq" 成功
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/order.cpp:75  通过主订单id查询订单
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:17  执行SQL语句 "SELECT * FROM orders WHERE mainid = :mainid"
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:20  :mainid : 28
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:532  查找成功
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/commodity.cpp:49  按id查找商品
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:17  执行SQL语句 "SELECT * FROM commodity WHERE id = :id"
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:20  :id : 8
[Debug] /Users/xqmmcqs/Documents/CPP_Work/Phase3/server/src/database.cpp:350  查找成功
```

图 16: 调试信息

订单的实现　程序在设计时考虑到订单可能需要退款，因此系统中实现的订单包含一个子 ID 和主订单 ID，当消费者创建订单时，会根据订单中商品归属的商家，将商品分成多个子订单，一个子订单中的商品都属于同一个商家，这些子订单的共同构成一个主订单。主订单的 ID 并不连续，由子订单的最小 ID 决定。支付、取消操作是针对主订单进行的，而退款操作是针对子订单进行的。进行主订单-子订单的划分也使得商家无法看到用户与其他商家交易的内容，保证了安全性。

当消费者查询订单时，是以主订单 ID 进行查询的，而当商家查询订单时，是以子订单 ID 进行查询的，由于主订单 ID 并不是主键，可能会有多个查询结果，这二者的在数据库中的查询方式也会有所不同。

由于订单的内容（也就是商品及其数量）是固定的，所以可以直接当做字符串来存储，订单的总金额也是只根据下单时刻的商品价值来决定，更加符合现实情况。

商品打折的处理 由于功能需求中的描述不甚明确，我采用我的理解实现了打折功能，即是类似全场促销的打折活动。打折功能没有权限限制，任何访客都可以进行打折，打折信息不写入数据库，当服务器重启后会消失。

此外，由于浮点数的精度限制，如果采用浮点数存储商品价格、折扣和订单金额，可能会出现较大的误差，客户端显示浮点数也可能出现长度过长的情况。于是我采用整数存储这些信息，折扣乘原价向下取整得到实际价格。

用户鉴权 用户登录之后，进行的操作不能超出其权限范围，例如消费者用户不能进行商品管理、不能查看和修改别人的订单、商家用户不能购物等等。而用户信息都存储在数据库中，识别一个用户的唯一凭证就是用户名。因此我的第一和第二版程序在用户登录后将用户名和签发人构成的 `json` 作为凭据返回，之后进行用户特定的操作只需要传入凭据就可以鉴权。

当设计网络版系统时，我采用当下流行的 JWT token，JWT 的实现细节已经在3.2.5节讨论过。JWT 凭据经过加密，密钥只保存在服务端，客户端难以自行生成凭据或者篡改凭据。因此大大增强了安全性。

JWT 凭据经 Base64 编码之后才能作为字符串加入报文中进行传输。这其中涉及到比较复杂的编码、解码和 `json` 对象处理操作。

错误信息返回 `UserManage` 类中的函数返回值为 `QString` 类型，如果操作没有产生错误，则该字符串为空，否则该字符串的内容是报错信息。`Server` 根据请求的不同类型构造回复报文，报文的 `status` 字段表示请求是否成功，`payload` 字段根据请求的类型给出请求的结果。客户端解析报文的 `status` 字段来判断操作是否成功，并且给予 QML 反馈，反馈机制类似，`ClientModel` 类中的函数的返回类型为 `QString`，当没有错误时，函数返回空串，否则函数返回错误信息。

QML 与 C++ 整合 QML 作为标记语言主要用于构建可视化界面，界面中的数据元素则可以使用 C++ 的对象，并且要求对象必须是 `QObject` 的子类，还要通过 `Q_PROPERTY` 指明可供访问的对象的属性、通过 `Q_INVOKABLE` 指明可供调用的对象的方法。

这一部分的难点主要有如下几个：每次对象的属性的值发生改变时，都要激活信号来提示 QML 刷新状态；通过 `ListView` 构建商品/订单列表，`ListView` 的属性 `ListModel` 是列表所展示的内容的模型，它对应的 C++ 对象是商品对象指针/订单对象指针的 `Qlist` 列表，这一部分在实现时涉及到复杂的处理，并且在更新列表时涉及到内存的释放和重新分配。

购物车管理部分，需要实现根据物品的选中/未选中来动态调整页面右下角的总金额，我通过内嵌一个 JavaScript 脚本来实现计算总金额。

界面布局和美术风格 可视化界面中所有的图标采用来自flaticon的免费 svg 格式 icon，在高分辨率下也不会出现图标模糊的情况，字体使用开源的思源宋体。

通过实现一些供复用的文本、图标、输入框和对话框，提供统一的视觉风格。界面中的元素大部分都是网格化布局和相对定位结合，以满足在窗口大小可动态调整的基础上，在任何状态下都能够提供正确的显示效果。界面中的商品列表和订单列表均采用 `ScrollView` 和 `ListView` 配合，实现变长的可滚动的列表。

6.2 工作总结

本次实验中我的收获主要有以下几个方面：

- 巩固了课堂上学习到的 C++ 面向对象知识，增强了 C++ 编程能力。
- 学习 Qt 框架的相关知识，增加对其的了解，初步掌握运用 Qt 开发 GUI 的能力。
- 初步掌握数据库的使用方法，熟悉 SQLite 的使用。
- 对面向对象编程有了更深的理解，增强面向对象架构和设计能力。
- 学习 Clion 和 Qt Creator 开发环境配置，学习 CMake 和 QMake 的使用方法。
- 在阅读 Qt 文档的过程中，增强信息检索能力和英文文献、文档阅读能力。

此外，我认为我的程序还有以下几点值得改进：

- 程序架构设计不成熟，函数封装层数过多，整体结构还需要精简。
- 程序各模块间功能划分不够明确，耦合程度仍然较高。
- 由于没有学习相关技术，没有进行完整的单元测试。

此次课程设计使我收获良多，积累了很多开发经验，对于其中的不足之处，我将在以后的学习中注重学习相关知识，争取从各方面提升自己的软件开发能力。