

面向对象程序设计实践（C++）

物流管理系统设计与实现

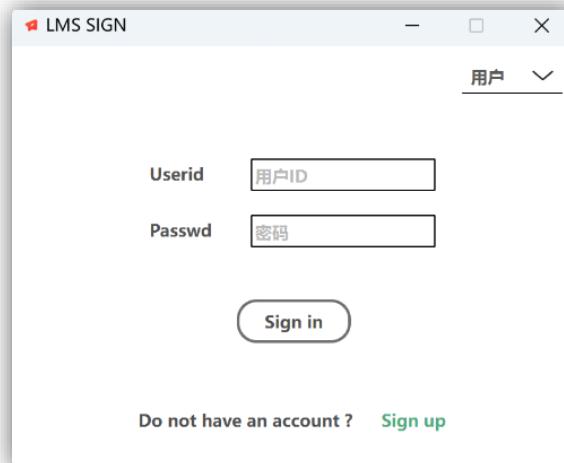
简旭峰 @ 2020212895

北京邮电大学 计算机学院

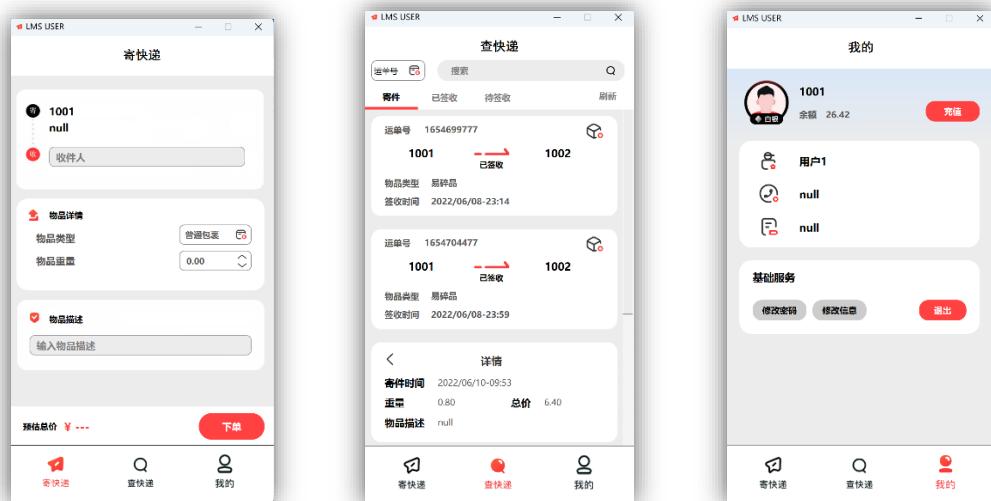
2022.04-2022.06

0、快照预览

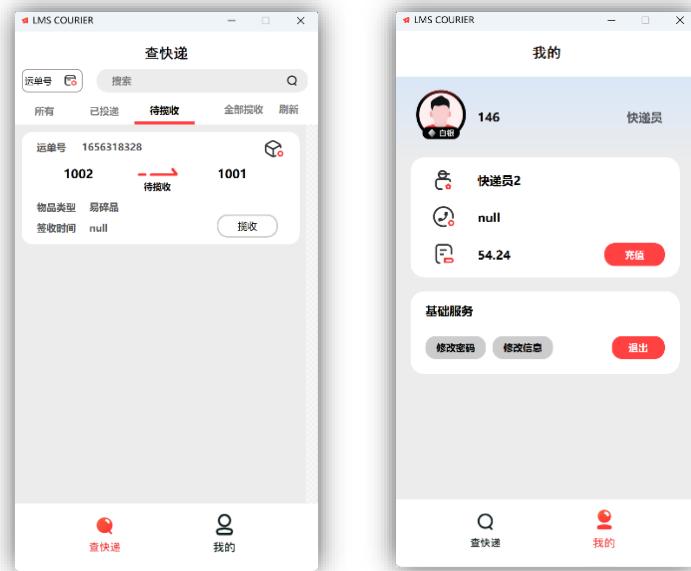
0.1、登录界面



0.2、用户界面

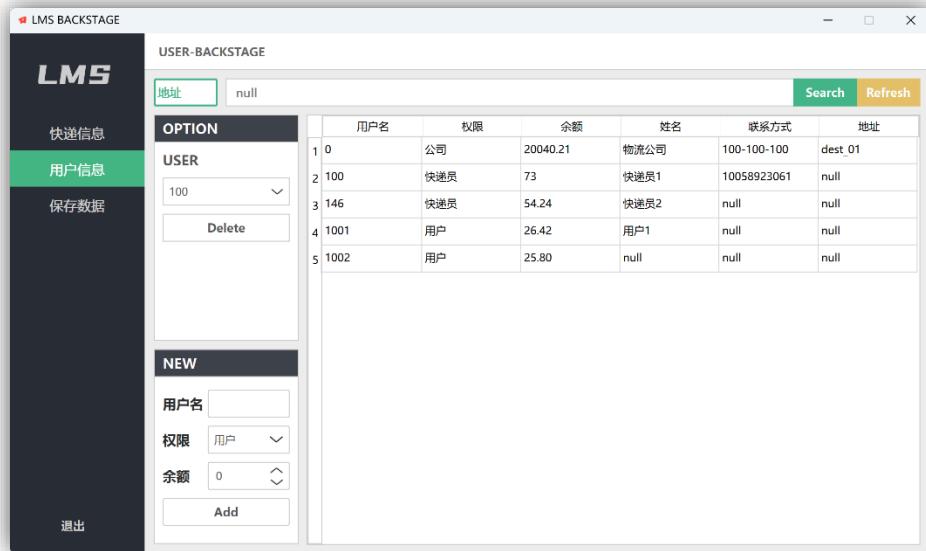


0.3、快递员界面



0.4、后台管理界面

运单号	运单状态	物品类型	重量/数量	揽收人	发件人	收件人	发件时间	收件时间	详情
1 1653385666	已签收	图书	16	100	1002	1001	2022/05/24-17:31	2022/05/25-07:32	Books
2 1654699777	已签收	易碎品	4.40	146	1001	1002	2022/06/08-22:32	2022/06/08-23:14	Items
3 1654704477	已签收	易碎品	5.60	146	1001	1002	2022/06/08-23:51	2022/06/08-23:59	null
4 16548227006	已签收	易碎品	6.40	159	1001	1002	2022/06/10-09:53	2022/06/10-09:55	null
5 16548227021	已签收	普通包裹	2.50	159	1001	1001	2022/06/10-09:53	2022/06/10-09:57	null
6 1654830379	待签收	易碎品	8.80	159	1001	1002	2022/06/10-10:49	null	null
7 1656304736	待签收	易碎品	9.60	146	1002	1001	2022/06/27-12:22	null	易碎品
8 1656304799	待签收	图书	10	未指定	1002	1001	2022/06/27-12:23	null	null
9 1656318328	待签收	易碎品	9.60	146	1002	1001	2022/06/27-16:08	null	null



0.5、服务端控制台

```

    问题   输出   调试控制台   终端   JUPYTER: VARIABLES
[Mon Jun 27 16:43:30 2022 CST] Current Dir: G:\Coding\QT\LMS\Ver-03\Server
[Mon Jun 27 16:43:30 2022 CST] USER Number : 5
[Mon Jun 27 16:43:30 2022 CST] PACKET Number : 9
[Mon Jun 27 16:43:30 2022 CST] <Message> Waiting For Connection...
[Mon Jun 27 16:43:30 2022 CST] <Message> Auto-Store Service Start
[Mon Jun 27 16:43:34 2022 CST] <Message> Connection 252 Establish
[Mon Jun 27 16:43:35 2022 CST] <Message> 252 Disconnect
[Mon Jun 27 16:43:41 2022 CST] <Message> Connection 244 Establish
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request A|10,admin,0
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request M|10|
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request M|8|0
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request H|7|
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request H|8|
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:41 2022 CST] 244 -> Request N|0|
[Mon Jun 27 16:43:41 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:43:48 2022 CST] 244 -> Request H|6|null
[Mon Jun 27 16:43:48 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:45:39 2022 CST] 244 -> Request P|||
[Mon Jun 27 16:45:39 2022 CST] 244 -> Exec return SUCCESS
[Mon Jun 27 16:45:48 2022 CST] 244 -> Request H|7|
[Mon Jun 27 16:45:48 2022 CST] 244 -> Exec return SUCCESS

```

目录

0、快照预览	1
0.1、登录界面	1
0.2、用户界面	1
0.3、快递员界面	2
0.4、后台管理界面	2
0.5、服务端控制台	3
1、概述	6
1.1、任务描述	6
1.2、开发环境	6
2、功能需求	7
2.1、物流业务管理子系统和用户管理子系统	7
2.2、快递员任务管理子系统	7
2.3、网络通信	7
3、总体框架	9
4、关键类设计	10
4.1、数据部分	10
4.1.1、User	10
4.1.2、PacketBase	10
4.1.3、UserControl	11
4.1.4、PacketControl	12
4.2、网络通信部分：	13
4.2.1、Socket	13
5、模块设计	14
5.1、概览	14
5.2、客户端	15
5.2.1、GUI	15
5.2.2、通信、请求与响应处理	16
5.2.3、断线重连	17
5.2.4、消息提醒弹窗	17
5.3、服务端	18
5.3.1、Socket 多线程	18
5.3.2、请求解析与执行	19
5.3.3、搜索与匹配	20
5.3.4、业务逻辑	21
5.3.5、自动存储	22
5.3.6、请求响应	23

6、使用指南	24
6.1、服务端	24
6.1.1、编译运行	24
6.1.2、运行状态	24
6.2、客户端登录	25
6.3、用户端	25
6.3.1、寄快递	25
6.3.2、查快递	26
6.3.3、收快递	27
6.3.4、个人信息	27
6.4、快递员端	28
6.4.1、查快递	28
6.4.2、揽收快递	28
6.4.3、个人信息	29
6.5、后台管理	30
6.5.1、快递管理	30
6.5.2、快递搜索	30
6.5.3、用户管理	31
6.5.4、用户搜索	31
6.5.5、存储数据	32
7、实验总结	33
7.1、问题及解决方案	33
7.2、经验教训	33

1、概述

1.1、任务描述

随着移动互联网的发展，上网购物、快递取货已经广泛地融入人们的生活。此次作业的任务是使用 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物流管理平台，提供物流管理、用户管理、员工管理等功能。

1.2、开发环境

Windows 11 22616.100

Qt Creator 6.1.1

Visual Studio Code 1.68.1

Cmake 3.22.0

g++ 10.3.0 C++17

2、功能需求

2.1、物流业务管理子系统和用户管理子系统

- 1、用户注册&登录：快递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。
- 2、修改账户密码：登录后对用户账号的密码修改。
- 3、余额管理：对用户账号中余额的查询、充值。
- 4、发送快递：用户申请发送快递到指定用户的手中，提交发送后系统为本次快递分配快递单号，本次快递状态变为待签收，并扣除用户的余额（每件快递 15 元），并将扣除的金额转到物流公司管理员的账号。
- 5、接受快递：用户可查看自己未签收的快递清单，可选择其中的一个或多个快递进行签收，签收意味着此物品的运送任务完成。
- 6、查询快递：用户查询自己发出的所有快递信息，以及接收的所有快递信息，可根据发送人/接收人、时间、快递单号查询快递。
- 7、物流业务管理：物流公司管理员可查看所有用户信息和所有的历史快递的详细信息，可根据用户、时间、快递单号进行查询。

2.2、快递员任务管理子系统

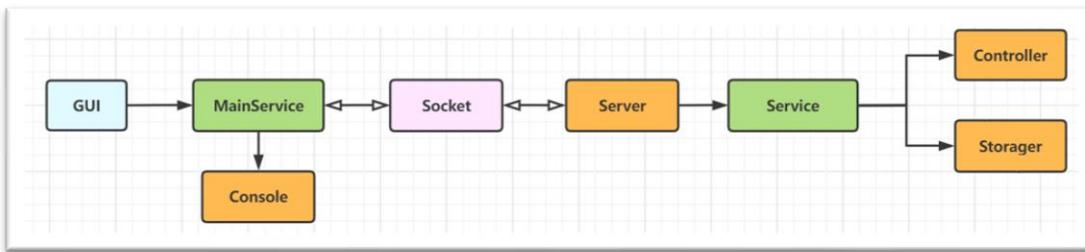
- 1、管理快递员：物流公司管理员添加和删除物流公司的快递员。即用户登录时可选不同身份，比如包括用户，快递员，管理员等，不同身份对应不同功能和权限。
- 2、快递分类：快递至少包含如下几类：易碎品（8 元/kg）、图书（2 元/本）、普通快递（5 元/kg）。
- 3、发送快递：用户发送快递到指定用户的手中，提交发送后系统为本次快递分配快递单号，快递进入待揽收状态，并扣除发件用户的余额（不同类型的快递具体计算），扣除金额转到物流公司管理员的账号。
- 4、快递员揽收：物流公司管理员对待揽收快递分配一名快递员，该快递员可查看其名下所有未揽收快递，并选择一个或多个进行揽收，被揽收的快递状态从待揽收变为待签收状态（无需模拟物品运输过程），并从物流公司管理员账号中将该快递费的 50% 转给快递员账号。
- 5、接受快递：收件用户查看自己未签收的快递，对自己的快递进行签收，签收意味着此物品的运送任务完成。
- 6、快递任务查询：快递员查询自己揽收和投递的所有快递信息，可根据发送人、接收人、时间、快递单号、以及快递状态查询快递信息。
- 7、物流业务管理：物流公司管理员可查看所有用户信息和所有的历史快递的详细信息，可根据用户、时间、快递单号进行查询。物流公司管理员可查看所有快递员及其揽收投递的快递的详细信息，可根据发件人、收件人、时间、快递单号进行查询。

2.3、网络通信

- 1、在版本一、二的基础上，将物流管理平台修改成网络版。

- 2、用户登录：用户通过客户端以账号密码登录平台。
- 3、用户发送快递：用户通过客户端向指定用户发送快递，数据发送到服务端等待处理。
- 4、物流管理：物流公司管理员通过客户端为未揽收快递分配快递员。
- 5、快递员揽收：快递员通过客户端揽收。
- 6、接受快递：用户通过客户端查看自己未签收的快递，对自己的快递进行签收，签收意味着此物品的运送任务完成。

3、总体框架



C/S	模块	功能
客户端	GUI	基于 QML 的 Qt 图形化界面
	MainService	响应 GUI 的点击操作，和服务器通信
	Console	消息弹窗提醒管理器
服务端	Socket	支持多线程的 socket 连接器
	Server	封装管理 socket 连接器和 Service 后台服务
	Service	后台服务主模块，解析执行来自客户端的请求
	Controller	底层数据控制，维护用户信息和包裹信息数据
	Storager	内存和文件的读取/写入通道

物流管理平台为 C/S 架构，客户端和服务端通过 **Socket** 建立 **TCP** 连接。客户端 **GUI** 通过 **点击事件** 传递 **MainService 基础服务** 完成数据请求和处理。服务端通过 **Service** 主服务完成对请求的解析和执行，通过 **Controller** 完成对底层数据的操作，通过 **Storager** 完成文件的读取和写入。

4、关键类设计

4.1、数据部分

4.1.1、User

User 类为用户类，为存储用户信息的基本单元，该类的成员函数主要为构造函数和所有变量的 **getter** 和 **setter** 方法，此处不做展示。

```

10  /* 用户类 */
11  class User{
12  private:
13      bool _signin;           //登录状态
14      int _id;                //用户名(ID,唯一)
15      int _auth;              //权限
16      std::string _name;       //姓名
17      std::string _passwd;     //密码
18      double _balance;        //余额
19      std::string _tel;        //联系方式
20      std::string _address;    //地址

```

4.1.2、PacketBase

PacketBase 为快递信息基类，包含快递的基本信息，所有具体快递类型需要继承该类。该类的成员函数主要为构造函数和所有变量的 **getter** 和 **setter** 方法，此处不做展示。

```

7   class PacketBase{
8  private:
9      size_t _id;           //运单号(包裹ID)
10     int _status;          //包裹状态
11     int _courier;         //揽收人
12     int _fromUser;        //寄件人
13     int _destUser;        //收件人
14     std::string _sendTime; //寄件时间
15     std::string _recvTime; //签收时间
16     std::string _describe; //包裹描述信息
17     int _type;             //包裹类型
18     double _count;         //包裹重量数量
19

```

三类具体快递类型均继承该基类进行设计。

```

105  /* 易碎品 */
106  class Breakable:public PacketBase{
107  public:
108      Breakable():PacketBase(){
109          setType((int)PacketType::BREAKABLE);
110      }
111      virtual double getPrice(){
112          return 8 * getCount();
113      }
114      virtual ~Breakable(){}
115  };
116

```

```

117 /* 图书 */
118 class Book:public PacketBase{
119 public:
120     Book():PacketBase(){
121         setType((int)PacketType::BOOK);
122     }
123     virtual double getPrice(){
124         return 2 * getCount();
125     }
126     virtual ~Book(){}
127 };
128
129 /* 普通快递 */
130 class Package:public PacketBase{
131 public:
132     Package():PacketBase(){
133         setType((int)PacketType::PACKAGE);
134     }
135     virtual double getPrice(){
136         return 5 * getCount();
137     }
138     virtual ~Package(){}
139 };
140

```

4.1.3、UserControl

UserControl 类为用户信息管理器，维护有用户信息指针列表和快递员信息指针列表。快递员信息列表是在读取或运行过程中通过用户列表的权限判断构建的备份，主要用于快递员的快速查找。此外还维护有存储器和控制台等。

```

92 /* 用户管理器 */
93 class UserControl{
94 private:
95     LOCK _lock;
96     Storager st; //存储器
97     Console sc; //控制台
98     std::vector<User*> _userList; //用户列表
99     std::vector<User*> _courierList; //快递员列表
100

```

UserControl 的成员函数主要为针对 User 列表的增删查改、对特定 User 信息的更改以及与文件的读写交互。

```

100 //字段匹配
101 bool user_matching(const User*,UserMatchField,std::string input) const;
102 public:
103     UserControl():_userList(),_courierList(){
104         _lock = LOCK::FREE;
105     }
106     LOCK* getLock(); //获取锁状态
107     void setLock(LOCK lv = LOCK::FREE); //设置锁状态
108     User* Signin(int userid,std::string passwd,int auth = (int)UserAuth::USER); //登录
109     int AddUser(User*); //添加用户
110     int DelUser(int userid); //删除用户
111     std::vector<User*> UserSearch(UserMatchField*,std::string*,int num); //用户搜索
112     User* UserLocate(int id); //单个用户定位
113     User* GetRoot(); //获取管理员指针
114     std::vector<User*>* getCourierList(); //获取快递员列表指针
115     return &_courierList;
116
117     void ReadIn(); //文件读入
118     int WriteBack(); //文件写入
119
120 };
121
122

```

4.1.4、PacketControl

PacketControl 类为包裹管理器，维护有包裹信息指针列表。此外还维护有存储器和控制台等。

```
142  /* 包裹管理器 */
143  class PacketControl{
144  private:
145      LOCK _lock;
146      Storager st;
147      Console sc;
148      std::vector<PacketBase*> _packetList; //包裹列表
```

PacketControl 的成员函数主要为针对 Packet 列表的增删查改、对特定 Packet 信息的更改以及与文件的读写交互。

```
152      //字段匹配
153      bool packet_matching(const PacketBase*,PacketMatchField,std::string input) const;
154  public:
155      PacketControl():_packetList(){
156          _lock = LOCK::FREE;
157      }
158      LOCK* getLock(){//获取锁状态
159          return &_lock;
160      }
161      void setLock(LOCK lv = LOCK::FREE){//设置锁状态
162          _lock = lv;
163      }
164      //包裹搜索
165      std::vector<PacketBase*> PacketSearch(PacketMatchField*,std::string* input,int num);
166      //单个包裹定位
167      PacketBase* PacketLocate(std::string id);
168      //设置包裹状态
169      int SetPacketStatus(size_t packetid,PacketStatus);
170      //添加包裹
171      int AddPacket(PacketBase*);
172      void ReadIn(); //文件读入
173      int WriteBack(); //文件写入
174      ~PacketControl();
175
176 };
```

4.2、网络通信部分：

4.2.1、Socket

Socket 类为自行封装的 Socket 连接器，主要维护有连接的 SOCKET 句柄和连接地址信息。该类的成员函数主要为 socket 连接、监听和数据接收。

```

17 /* Socket连接封装 */
18 class Socket{
19 private:
20     SOCKET _fd;           //SOCKET句柄
21     SOCKADDR_IN _addr;    //地址结构体
22 public:
23     void Listen(size_t bind_port);      //端口监听
24     SOCKET Accept(SOCKET server);      //连接监听
25     int RecvData(std::string&);        //数据接收
26     void Close();                     //关闭连接
27     void SetTimeOut(size_t send = SEND_TIME_OUT, size_t recv = RECV_TIME_OUT); //设置超时
28     SOCKET getFd();                  //获取句柄
29
30 };

```

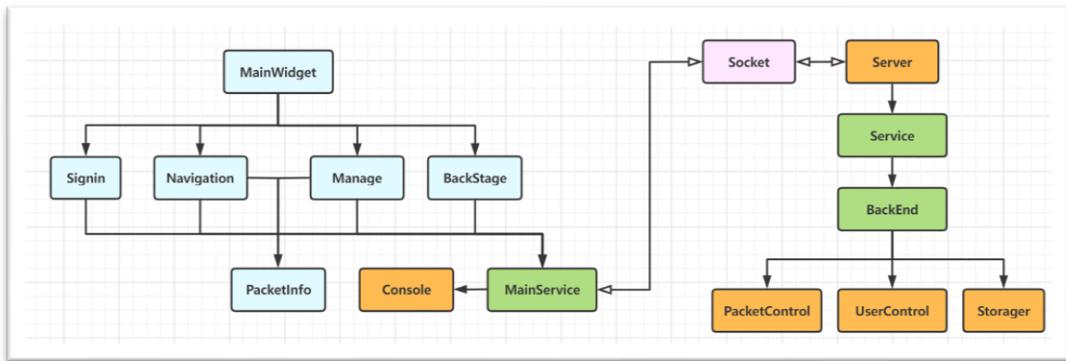


```

15 /* 端口监听 */
16 void Socket::Listen(size_t bind_port){
17     _fd = WSAsocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
18     //设置socket基础配置
19     _addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);          //任意地址
20     _addr.sin_family = AF_INET;                                //IPv4
21     _addr.sin_port = htons(bind_port);                         //端口
22     Console sc;
23     //端口绑定
24     if(bind(_fd, (SOCKADDR*)&_addr, sizeof(_addr)) != 0){
25         sc.err("Bind Port "+std::to_string(bind_port)+" Failed");
26         exit(-1);
27     }
28     sc.msg("Bind Port "+std::to_string(bind_port)+" Success");
29     if(listen(_fd, MAX_CONNECT) != 0){
30         sc.err("Listening Failed");
31         exit(-1);
32     }
33     sc.msg("Listening...");
34 }
35
36
37 /* 连接监听 */
38 SOCKET Socket::Accept(SOCKET server){
39     int length = sizeof(_addr);
40     _fd = accept(server, (SOCKADDR*)&_addr, &length); //新连接建立检测
41     return _fd;
42 }
```

5、模块设计

5.1、概览



C/S	模块	类	功能
客户端	GUI	MainWidget	主窗口
		Signin	登录界面
		Navigation	用户界面
		Manage	快递员界面
		PacketInfo	包裹信息组件
		BackStage	后台界面
	弹窗管理	Console	弹窗消息提醒管理器
	主服务	MainService	响应 GUI 点击操作，通信及数据处理
服务端	Socket	Socket	多线程 socket 连接器
	Server	Server	封装管理服务端
	主服务	Service	解析客户端请求
		BackEnd	执行客户端请求
	数据操作	UserController	用户信息管理器
		PacketController	包裹信息管理器
		Storager	内存和文件的读取/写入

物流管理平台采用 C/S 架构，由客户端和服务端组成。

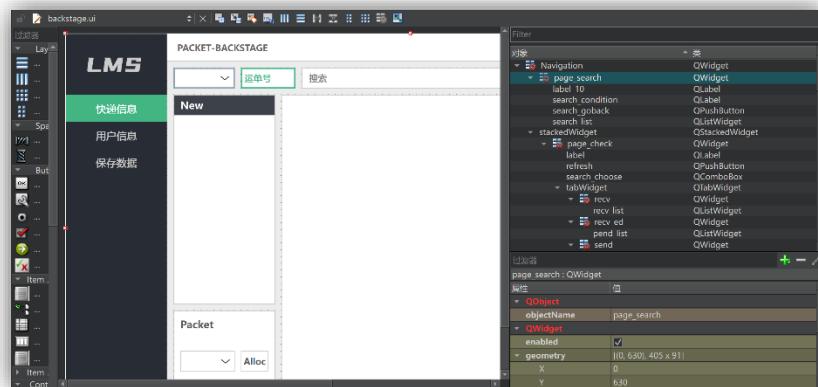
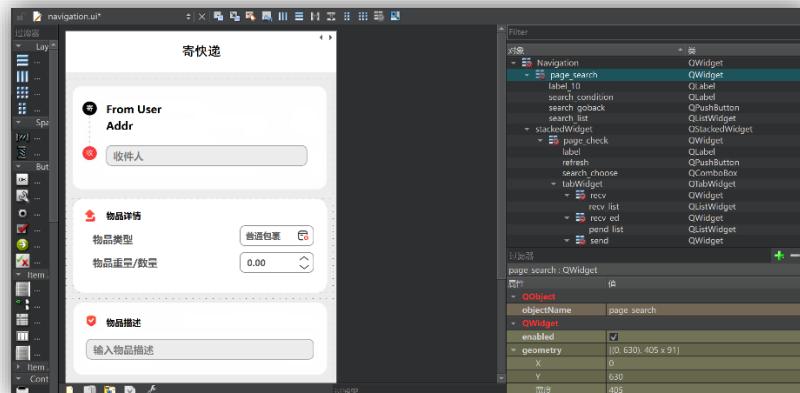
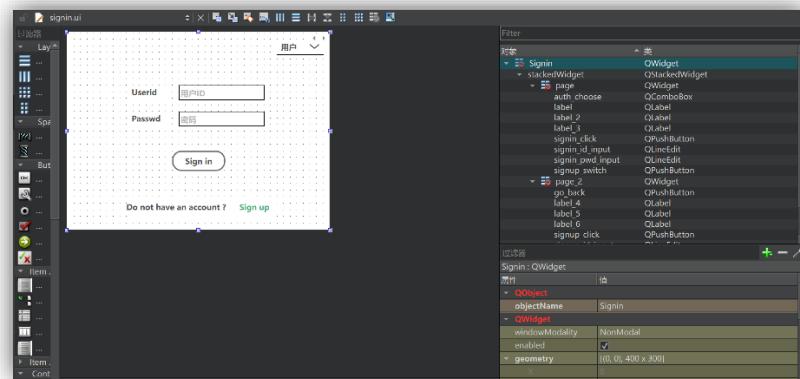
客户端分为 **GUI 界面、弹窗管理以及基础服务**。主 GUI 组件 **MainWidget** 管理三个子组件的显示与隐藏，子组件为 **Signin 登录界面**、**BackStage 后台界面**、**Navigation 用户界面** 以及 **Manage 快递员界面**。同时 **Navigation 用户界面** 和 **Manage 快递员界面** 还调用了 **PacketInfo 子组件** 进行包裹信息展示。图形界面和底层数据操作由 **MainService** 进行桥接，**GUI 组件** 通过 **点击事件** 调用 **MainService 层** 进行服务端通信、接收服务端的数据并处理。通过 **Console 模块** 进行弹窗消息提醒的统一管理。

服务端部分通过 **BackEnd 模块** 统一控制底层的 **Controller** 和 **Storager**。在通信方面，通过单独设计的 **Socket 模块** 进行多线程通信处理，通过 **Service 模块** 进行客户端请求内容的解析，并将解析结构传递给 **BackEnd 模块** 进行数据处理。**Server 模块** 作为 **Socket 模块** 和 **Service 模块** 的封装，主要负责 **Socket 模块** 和 **Service 模块** 的初始化和整个服务的启动。

5.2、客户端

5.2.1、GUI

客户端的 GUI 设计通过 **Qt Designer** 完成，主要基于 **QML**。客户端的设计灵感来源于 **顺丰快递小程序**，各按钮图标通过 **PS** 进行背景透明化，统一存储在 **Resources** 包中管理。登录、用户、快递员和后台管理为不同的 ui 文件，在各 GUI 类中存有 **MainService** 服务指针，各个点击事件通过相应槽函数或发出信号以告知 MainService 进行操作。



5.2.2、通信、请求与响应处理

客户端的网络通信、请求发送和响应处理集成在 MainService 类中完成。在客户端启动时，MainService 会同步初始化 Socket 连接，设置服务器地址以及请求应答超时时间。

```

12  /* 启动服务 */
13  void MainService::Start(){
14      //WSA初始化
15      WSADATA wsadata;
16      WSAStartup(0x202, &wsadata);
17      //socket初始化
18      _fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //TCP
19      _addr.sin_family = AF_INET; //IPv4
20      _addr.sin_addr.S_un.S_addr = inet_addr(SERVER_ADDRESS); //服务器地址
21      _addr.sin_port = htons(SERVER_PORT); //服务器端口
22      //设置超时
23      size_t time_out = SEND_TIME_OUT;
24      setsockopt(_fd, SOL_SOCKET, SO_SNDTIMEO, (char*)&time_out, sizeof(int));
25      time_out = RECV_TIME_OUT;
26      setsockopt(_fd, SOL_SOCKET, SO_RCVTIMEO, (char*)&time_out, sizeof(int));
27 }

```

客户端 socket 通信模式为阻塞式，向服务端发送请求后需同步等待服务端响应并解析响应内容。客户端默认请求超时重发次数为 5 次，是为了避免服务器可能返回的 BUSY 信号，提高程序操作效率。

```

54  /* 发送请求 */
55  int MainService::Request(QString req, QString& ans, int polling_times){
56      string request = req.toStdString();
57      sc.LogMsg("[SEND] "+request);
58      while(polling_times){
59          if(send(_fd, request.c_str(), request.length() + 1, 0) != SOCKET_ERROR){
60              break;
61          }
62          polling_times--;
63      }
64
65      int err;
66      char buf[1024];
67      QString res = "";
68      //循环数据接收
69      do{
70          memset(buf, 0, sizeof(buf));
71          //接收数据
72          err = recv(_fd, buf, sizeof(buf), 0);
73          res += QString::fromStdString(string(buf));
74      }while(err >= 1024);
75      if(err == SOCKET_ERROR){
76          return CONNECTION_LOST;
77      }
78      sc.LogMsg("[RECV] "+res.toStdString());
79      //响应数据解析
80      QStringList list = res.split("|");
81      int ret = list[0][0].toLatin1() - '0';
82      if((ReturnValue)ret != ReturnValue::SUCCESS){
83          return ret;
84      }
85      ans = list[1];
86      return SUCCESS;
87 }

```

5.2.3、断线重连

客户端设计有断线重连机制，当重发请求次数超过预设值时，客户端会判断与服务器失去连接，并弹出提示框判断用户是否选择重新连接。重连会再次向服务端发起登录请求，并更新客户端信息，如果是服务器宕机导致断线，则该函数返回 **CONNECTION_LOST**，表示链接失败。

```

64     //超时重连
65     while(polling_times == 0){
66         QMessageBox::StandardButton result =
67             QMessageBox::critical(nullptr,"ERROR","连接服务器超时",
68                                 QMessageBox::Retry|QMessageBox::Cancel,
69                                 QMessageBox::Retry
70             );
71         if(result == QMessageBox::Retry){
72             //连接重试
73             if(Connect(2)){
74                 if(_USER != nullptr){
75                     //重新登录
76                     if(Signin(_USER->getID(),_USER->getPasswd(),_USER->getAuth()) != SUCCESS){
77                         return SYSTEM_ERR;
78                     }
79                 }
80                 QMessageBox::information(nullptr,"TIPS","重连成功");
81                 return SUCCESS;
82             }
83         }else{
84             break;
85         }
86     }
87     if(polling_times == 0) return CONNECTION_LOST;

```

5.2.4、消息提醒弹窗

客户端设置有专用的 **Console** 类用户判断请求返回值并弹出含有指定内容的提示窗口。

```

13 bool Console::StatusCheck(int ret,QString not_found_msg,QString sys_err_msg,QString null_ptr_msg)
14 {
15     if(ret == SUCCESS) return true;
16     if(ret == NOT_FOUND){
17         QMessageBox::critical(nullptr,"ERROR",not_found_msg);
18     }else if(ret == SYSTEM_ERR){
19         QMessageBox::critical(nullptr,"ERROR",sys_err_msg);
20     }else if(ret == NULL_PTR){
21         QMessageBox::critical(nullptr,"ERROR",null_ptr_msg);
22     }else if(ret == TIME_OUT){
23         QMessageBox::information(nullptr,"TIPS","请求超时，服务器繁忙");
24     }
25     return false;
}

```

5.3、服务端

5.3.1、Socket 多线程

服务端采用 WSA 非阻塞式 socket 通信，通过自封装 Socket 类的 Accept 函数判断是否有新建立的 socket 连接。当新连接建立时，服务端将生成独立线程，并将相关服务的参数指针传递到线程函数中进行处理，主线程则继续进行连接监听。

```

109     Socket connect;
110     sc.msg("Waiting For Connection...");
111     /* 循环连接监听 */
112     while(true){
113         connect.Accept(_server.getFd());
114         if(connect.getFd() != INVALID_SOCKET){
115             HANDLE thread;           //新线程
116             void* temp[2]={(void*)&connect,(void*)this};    //构造参数指针数组
117             //新建线程
118             thread = CreateThread(nullptr,0,_recvData,&temp,0,nullptr);
119             CloseHandle(thread); //关闭线程句柄
120         }
121     }
122     _server.Close(); //关闭socket
123     WSACleanup(); //清空WSA
124 }
```

在线程函数中，将由 Service 类循环接收、解析并执行客户端请求并返回执行结果，服务端同样配置有特定次数的超时重发机制。当重发次数达到上限或是客户端主动关闭，此线程就会退出 socket 循环，清除线程内的残余数据并关闭此线程。

```

126 /* 新连接线程 */
127 DWORD Server::_recvData(LPVOID param){
128     /* 参数处理 */
129     Server* server = (Server*)((void**)param)[1];
130     Socket connect = *(Socket*)((void**)param)[0];
131     connect.SetTimeout(); //超时设置
132     string fd = std::to_string(connect.getFd());
133     server->sc.msg("Connection "+fd+" Establish");
134     string res;
135     //连接到主服务
136     Service svc(server);
137     while(connect.RecvData(res) == 0){
138         svc.sc.log(true,BOLDWHITE "%s" RESET" → Request %s\n",fd.c_str(),res.c_str());
139         //请求解析
140         int err = svc.Parse(res);
141         if(err == SUCCESS){
142             //请求执行
143             err = svc.Exec();
144             svc.sc.log(true,BOLDWHITE "%s" RESET" → Exec return %s\n", fd.c_str(),RetValText[err].c_str());
145         }else{
146             svc.sc.err(fd+" → Request Unknown return "+RetValText[err]);
147         }
148         //响应格式化
149         string ans = svc.Format(err);

152         while(times){
153             //长数据分段
154             string ans_copy = ans;
155             do{
156                 string buf;
157                 if(ans_copy.length() > 1024){
158                     buf = ans_copy.substr(0,1024);
159                     ans_copy = ans_copy.substr(1024);
160                 }else{
161                     buf = ans_copy;
162                     ans_copy = "";
163                 }
164                 //发回响应信息
165                 send_len = send(connect.getFd(),buf.c_str(),buf.length() + 1,0);
166             }while(ans_copy.length() > 0);
167             if(send_len > 0){
168                 break;
169             }
170             times--;
171         }
172         if(times == 0){
173             svc.sc.err("Connection Time Out");
174             break;
175         }
176     }
177     //关闭连接
178     server->sc.msg(std::to_string(connect.getFd())+" Disconnect");
179     connect.Close();
180     return 0;
181 }
```

5.3.2、请求解析与执行

在客户端与服务端的通信中，请求报文被设计为 [请求代号|参数|数据]，三个域由“|”隔开，多个参数和数据则由“，”隔开，在解析时通过 `find` 函数和自封装的 `Split` 函数实现请求的分离解析。

```

32  /* 请求解析 */
33  int Service::Parse(std::string request){
34      if(request.length() == 0) return NULL_PTR;
35      char function = request[0]; //请求类型
36      _func = (FUNCTION)function;
37      if(function < (char)FUNCTION::SIGN_IN || function > (char)FUNCTION::STORE_DATA){
38          return NOT_FOUND;
39      }
40      //三段分割
41      int pos1 = request.find(' ');
42      int pos2 = request.find(' ', pos1 + 1);
43      if(pos1 == string::npos || pos2 == string::npos){
44          return SYSTEM_ERR;
45      }
46      Clear();
47      string param = request.substr(pos1 + 1, pos2 - pos1 - 1);
48      string value = request.substr(pos2 + 1);
49      //内容分割
50      _param = sc.Split(param, ",");
51      _value = sc.Split(value, ",");
52      return SUCCESS;
53  }

```

如果该请求解析被接收，则将跳转到对该请求的执行，`Exec` 函数会通过请求类型进行具体实现函数的跳转。

```

55  //请求执行
56  int Service::Exec(){
57      switch(_func){
58          case FUNCTION::SIGN_IN:           return exec_sign_in();           //登录
59          case FUNCTION::SIGN_UP:          return exec_sign_up();          //注册
60          case FUNCTION::SIGN_OUT:         return exec_sign_out();         //登出
61          //
62          case FUNCTION::ADD_USER:         return exec_add_user();         //添加用户
63          case FUNCTION::DEL_USER:         return exec_del_user();         //删除用户
64          case FUNCTION::SET_USER:         return exec_set_user();         //设置用户信息
65          case FUNCTION::RECHARGE:         return exec_recharge();         //充值
66          case FUNCTION::CHECK_USER:        return exec_check_user();        //用户搜索
67          //
68          case FUNCTION::SEND_PACKET:       return exec_send_packet();       //发送包裹
69          case FUNCTION::RECV_PACKET:       return exec_recv_packet();       //签收包裹
70          case FUNCTION::COLLECT_PACKET:    return exec_collect_packet();    //揽收包裹
71          case FUNCTION::ALLOC_COURIER:     return exec_alloc_courier();     //分配快递员
72          case FUNCTION::CHECK_PACKET:      return exec_check_packet();      //包裹搜索
73          //
74          case FUNCTION::ALLOC_SWITCH:      return exec_alloc_switch();      //切换自动分配
75          case FUNCTION::AUTO_ALLOC:        return exec_auto_alloc();        //自动分配
76          case FUNCTION::STORE_DATA:        return exec_store_data();        //数据存储
77          default:                         return NOT_FOUND;
78      }
79  }

```

5.3.3、搜索与匹配

在业务逻辑中登录、搜索等涉及到不同字段的信息匹配搜索，因此在两个 Controller 类中分别封装了匹配判断函数和搜索函数。

以 UserController 为例，匹配函数通过传入的用户指针、匹配字段和数据进行匹配并返回结果。

```

111  bool UserControl::user_matching(const User* u,UserMatchField field,string input) const{
112      if(u == nullptr) return false;
113      switch(field){
114          case UserMatchField::ID:
115              return u->getID() == stoi(input);
116          case UserMatchField::AUTH:
117              return u->getAuth() == stoi(input);
118          case UserMatchField::BALANCE:
119              return u->getBalance() == stod(input);
120          case UserMatchField::PASSWD:
121              return u->getPasswd() == input;
122          case UserMatchField::NAME:
123              return u->getName().compare(0,input.length(),input) == 0;
124          case UserMatchField::TEL:
125              return u->getTel().compare(0,input.length(),input) == 0;
126          case UserMatchField::ADDRESS:
127              return u->getAddr().compare(0,input.length(),input) == 0;
128      }
129      return false;
130  }

```

对于多个字段的匹配，则由 UserSearch 函数通过 **for** 循环进行遍历控制。

```

148  /* 用户搜索 */
149  v<vector<User*>> UserControl::UserSearch(UserMatchField* farray,string* inputs,int num){
150      if(num == 0) return _userList;
151      vector<User*> list;
152      if(!sc.wait(&_lock)) return list;
153
154      setLock(LOCK::RUN);
155      //遍历搜索
156      auto itr = _userList.begin();
157      while(itr != _userList.end()){
158          bool flag = true;
159          //遍历字段
160          for(int i = 0; i < num && flag; i++){
161              setLock(LOCK::RUN);
162              flag = user_matching((*itr),farray[i],inputs[i]);
163          }
164          if(flag){
165              list.push_back((*itr));
166          }
167          itr++;
168      }
169      setLock();
170      //快速排序
171      sort(list.begin(),list.end(),[](User* u1,User* u2){return u1->getID() < u2->getID();});
172      return list;
173  }

```

5.3.4、业务逻辑

具体业务逻辑的实现流程为，Service 类完成请求的解析，将请求中的参数和数据以参数形式传递给 BackEnd，由 BackEnd 调用 UserController 和 PacketController 完成具体的数据提取、搜索、修改等操作。此处以发送包裹作为介绍。

在发送包裹时，BackEnd 需要首先检测用户的登录绑定状态，避免空指针错误。随后根据包裹类型新建相应用对象，并在填充该对象的完整信息前首先检查用户的余额情况。

```

39  /* 发送包裹 */
40  int BackEnd::SendPacket(User* CURUSER,int type,int dest,double count,string description){
41      if(CURUSER == nullptr)  return NULL_PTR;
42      static time_t pre_time = 0;
43      time_t t = time(NULL);
44      //操作间隔检测
45      if(t == pre_time){
46          return SYSTEM_ERR;
47      }else{
48          pre_time = t;
49      }
50      int ret;
51      //包裹对象初始化
52      PacketBase* pb;
53      switch((PacketType)type){
54          case PacketType::PACKAGE:
55              pb = new Package(); break;
56          case PacketType::BREAKABLE:
57              pb = new Breakable(); break;
58          case PacketType::BOOK:
59              pb = new Book(); break;
60      }
61      pb->setCount(count);           //更新数量/重量
62      double price = pb->getPrice(); //更新加个
63
64      //余额检测
65      if(CURUSER->getBalance() < price) return SYSTEM_ERR;
66      CURUSER->setBalance(CURUSER->getBalance() - price);
67      _ROOT->setBalance(_ROOT->getBalance() + price);

```

如果用户的余额满足要求，则通过传递的参数和当前时间进行包裹具体信息的填充，并将包裹加入 PacketController 维护的 PacketBase 动态数组。在返回前，还会检查 _AUTO_ALLOCATION 变量，判断是否需要自动随机分配快递员。

```

68      srand(time(nullptr));
69
70      //设置包裹信息
71      size_t id = t + CURUSER->getID();
72      pb->setId(id);
73      pb->setFromUser(CURUSER->getID());
74      pb->setDestUser(dest);
75      pb->setSendTime(sc.getTime());
76      pb->setDescription(description);
77      ret = _PACKETS.AddPacket(pb);
78      if(ret != SUCCESS) return ret;
79
80      //自动分配检测
81      if(_AUTO_ALLOCATION){
82          ret = RandomAlloc(to_string(id));
83      }
84      return ret;
85  }

```

对于快递分配，该系统内置了自动随机分配的开关，当开关激活时，新寄出的快递会直接进行自动随机分配。通过当前时间和静态变量 t 生成随机数，根据该随机数指定分配的快递员。

```

152 /* 随机分配 */
153 int BackEnd::RandomAlloc(string packetid){
154     static size_t t = 0;
155     vector<User*>& list = _USERS.getCourierList();
156     if(list->size() == 0)    return NULL_PTR; //空指针检测
157     //随机数生成
158     srand(time(nullptr) + (t++));
159     int id = rand()%list->size();
160     return AllocateCourier(packetid,list->at(id)->getID());
161 }

```

该系统内部检查功能有诸多**绑定设计**。在删除用户时，需要检测用户的登录状态，如果用户正在登录，则删除失败。如果删除快递员成功，则需要将该快递员所有未揽收快递的状态变更为未分配快递员状态以重新等待分配。

```

124 /* 删除用户 */
125 int BackEnd::DelUser(int id){
126     User* user = _USERS.UserLocate(id);
127     if(user == nullptr) return NOT_FOUND;
128     //快递员检测
129     if(user->getAuth() == (int)UserAuth::COURIER){
130         PacketMatchField f[] {PacketMatchField::COURIER,PacketMatchField::STATUS};
131         string inputs[] {to_string(id), "0"};
132         //搜索未揽收快递
133         vector<PacketBase*> list = _PACKETS.PacketSearch(f,inputs,2);
134         auto itr = list.begin();
135         //重置未揽收快递信息
136         while(itr != list.end()){
137             (*itr)->setCourier(0);
138             itr++;
139         }
140     }
141     return _USERS.DelUser(id);
142 }

```

5.3.5、自动存储

在 Server 类运行后，会在监听连接前启动独立线程进行间隔 3 分钟的自动数据保存，以更好的实现持久化。新线程的执行内容由 Lambda 表达式进行构建。

```

89     /* 自动保存线程 */
90     std::thread([this](){
91         sc.msg("Auto-Store Service Start");
92         int ret;
93         do{
94             Sleep(3 * 60 * 1000);           //间隔3min
95             ret = this->getBackEnd()->StoreData(); //存储数据
96             if(ret == TIME_OUT){          //锁超时重试
97                 Sleep(100);
98                 ret = this->getBackEnd()->StoreData();
99             }
100            if(ret == SUCCESS){
101                sc.msg("Data Store Success");
102            }else{
103                sc.log(true,BOLDWHITE"Data Store Time Out\n");
104            }
105        }while(ret==SUCCESS || ret == TIME_OUT);
106        sc.err("Storage Error : "+RetValText[ret]+". Exiting... ");
107        exit(-1);
108    }).detach();

```

5.3.6、请求响应

在 Service 类中 Exec 函数调用执行完成后，都会返回操作的执行结果代码。结果代码有 5 类，分别对应操作成功、超时、数据未找到、空指针错误以及系统错误。

```
76 enum ReturnValue{
77     SUCCESS,
78     TIME_OUT,
79     NOT_FOUND,
80     NULL_PTR,
81     SYSTEM_ERR,
82 };
```

通过该结果代码和存储在 Service 类变量_ans 中的返回数据，Format 函数将操作返回结果信息和数据信息以“|”拼接，作为响应信息发送回客户端。

```
81 /* 返回信息格式化 */
82 string Service::Format(int res){
83     //返回值代码+返回数据
84     string ret = BACK_MSG[res] + "|";
85     ret += _ans;
86     Clear(); //清空请求信息
87     return ret;
88 }
```

6、使用指南

6.1、服务端

6.1.1、编译运行

服务端环境为 Windows 11，采用 Cmake 进行辅助编译。该版本中设计了自动编译运行脚本，只需键入“`./auto.bat`”，就可以自动编译并运行程序。需要注意的是，可执行程序所在 bin 文件夹应与 data 数据文件夹处于同一文件目录下，否则可能造成数据文件读取失败。

```
PS G:\Coding\QT\LMS\Ver-03\Server> ./auto.bat

G:\Coding\QT\LMS\Ver-03\Server>mkdir build
子目录或文件 build 已经存在。

G:\Coding\QT\LMS\Ver-03\Server>cd build

G:\Coding\QT\LMS\Ver-03\Server\build>cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: G:/Coding/QT/LMS/Ver-03/Server/build

G:\Coding\QT\LMS\Ver-03\Server\build>make
Consolidate compiler generated dependencies of target LMS-Server
[100%] Built target LMS-Server

G:\Coding\QT\LMS\Ver-03\Server\build>cd ..
```

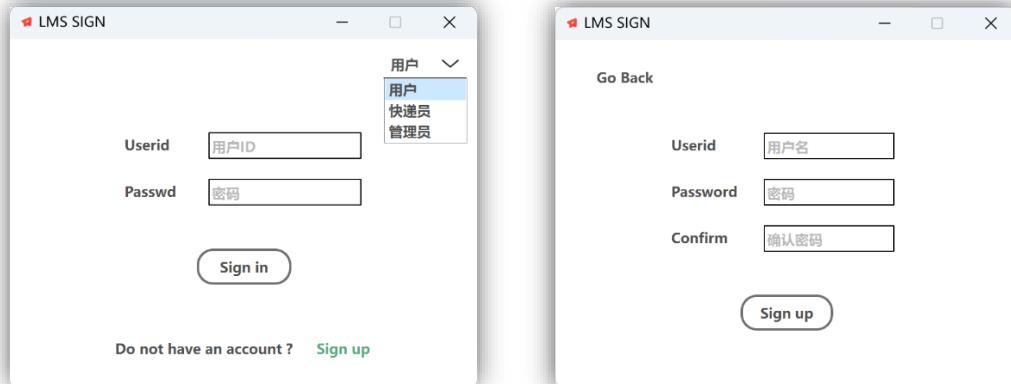
6.1.2、运行状态

服务端将绑定在 8201 端口运行，通过控制台可以看到各个客户端的连接建立和断开情况、各个客户端的请求内容以及当前时间。

```
G:\Coding\QT\LMS\Ver-03\Server\bin>LMS-Server
[Mon Jun 27 11:55:08 2022 CST] <Message> Bind Port 8201 Success
[Mon Jun 27 11:55:08 2022 CST] <Message> Listening...
[Mon Jun 27 11:55:08 2022 CST] <Message> Server Start
[Mon Jun 27 11:55:08 2022 CST] Current Dir: G:\Coding\QT\LMS\Ver-03\Server
[Mon Jun 27 11:55:08 2022 CST] USER Number : 5
[Mon Jun 27 11:55:08 2022 CST] PACKET Number : 6
[Mon Jun 27 11:55:08 2022 CST] <Message> Waiting For Connection...
[Mon Jun 27 11:55:08 2022 CST] <Message> Auto-Store Service Start
[Mon Jun 27 11:57:20 2022 CST] <Message> Connection 244 Establish
[Mon Jun 27 11:57:20 2022 CST] 244 -> Request A||10001,123,0
[Mon Jun 27 11:57:20 2022 CST] 244 -> Exec return NOT_FOUND
[Mon Jun 27 11:57:22 2022 CST] <Message> 244 Disconnect
[Mon Jun 27 11:57:22 2022 CST] <Message> Connection 264 Establish
[Mon Jun 27 11:57:22 2022 CST] 264 -> Request A||10001,1234,0
[Mon Jun 27 11:57:22 2022 CST] 264 -> Exec return NOT_FOUND
```

6.2、客户端登录

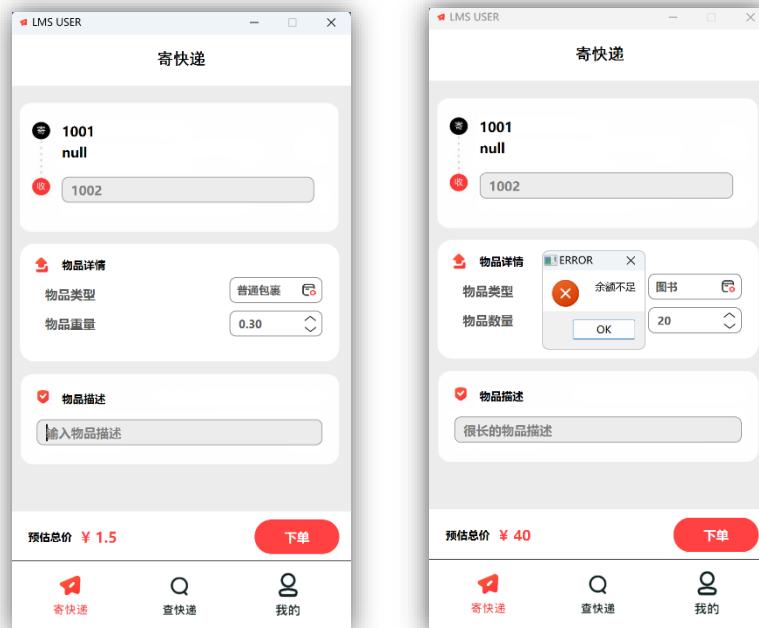
用户可以在登录界面选择登录的权限类型，可以点击注册按钮跳转到注册页面，点击返回按钮返回到登录界面。



6.3、用户端

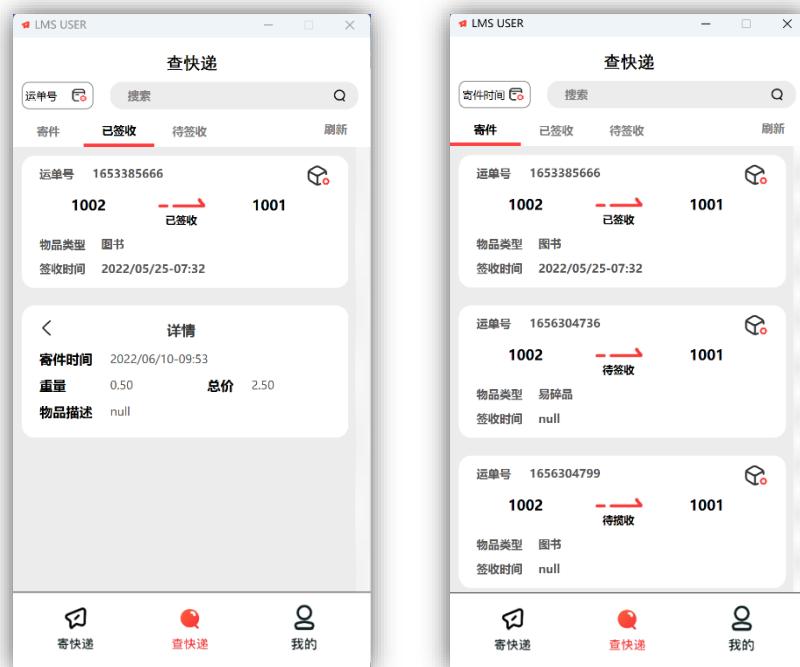
6.3.1、寄快递

点击底部导航栏上的寄快递选项，输入收件人的用户名（ID），选择物品类型和物品重量/数量，并填写物品描述。在底部会实时显示总价，点击下单即可寄出快递。如果输入内容不符合格式或是余额不足，系统会进行弹窗提醒。



6.3.2、查快递

点击底部导航栏的查快递选项，可以看到自己寄出的、已签收的、待签收的所有状态信息。点击快递组件右上角的按钮可以展开快递的详细信息。



在页面顶部，可以选择搜索字段，输入搜索内容后点击右侧搜索按钮就可以通过运单号、收件人、收件时间等搜索快递。涉及时间的搜索字段支持模糊匹配。



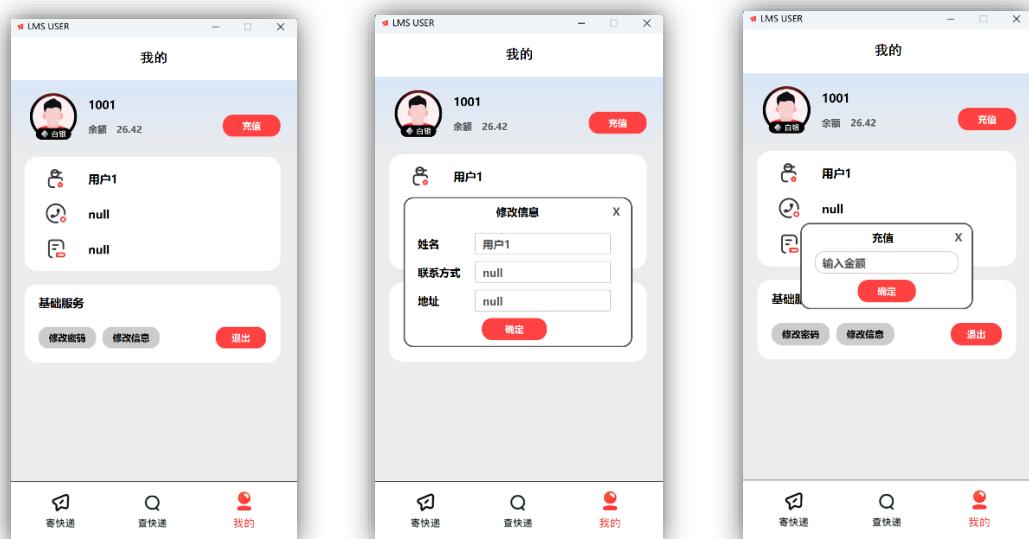
6.3.3、收快递

在查快递页面，用户可以点击待签收页面查看所有自己待签收的快递，并点击签收按钮进行签收。



6.3.4、个人信息

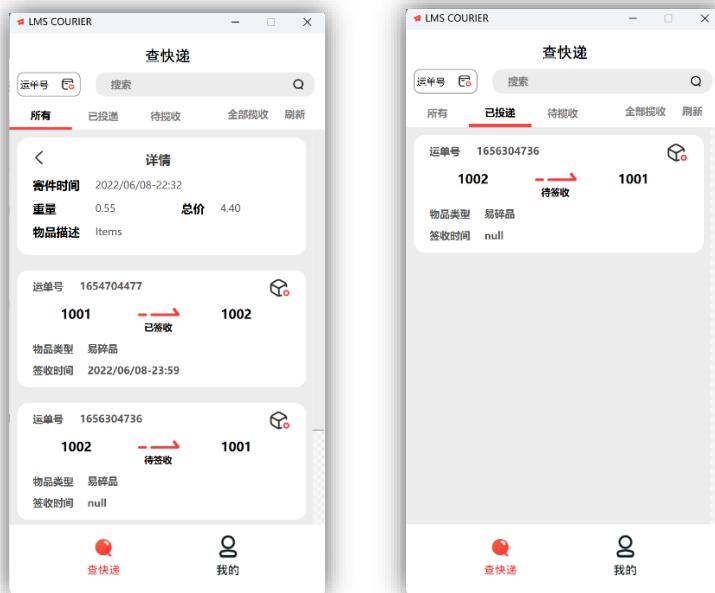
点击底部导航栏的我的选项，用户可以查看个人包括用户名、地址、电话、余额在内的所有信息。此外，可以点击上方的充值按钮进行余额充值，点击基础服务栏中的修改信息进行个人信息的更新，点击修改密码进行密码更新。



6.4、快递员端

6.4.1、查快递

快递员端的查快递功能与用户端基本框架相同。快递员可以查看所有自己投递的及待揽收的快递。



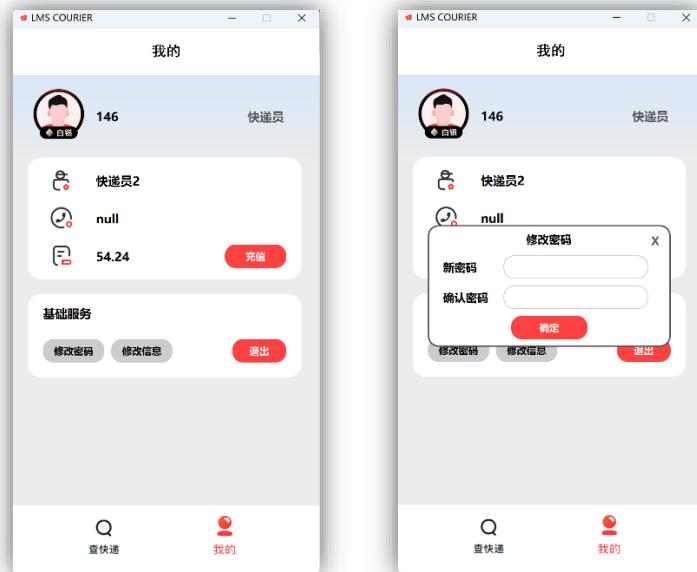
6.4.2、揽收快递

快递员可以在待揽收页面查看待揽收的快递项并分别揽收，也可以点击右上方的全部揽收进行一键操作。



6.4.3、个人信息

点击底部导航栏的我的选项，快递员可以查看个人包括用户名、电话、余额在内的所有信息。因为快递员端没有设置寄快递功能，因此略去了地址选项。同样的，快递员也可以点击按钮进行充值、修改信息和密码。



6.5、后台管理

6.5.1、快递管理

后台管理员可以在快递信息页面表格查看所有的快递信息。在左侧的 **New** 一栏中可以查看新寄出的、待分配快递员的所有运单。后台管理员可以点击每一项分别进行分配，也可以打开最底部的 **Auto Allocate** 按钮打开自动随机分配功能，此后所有新寄出的快递都将自动随机分配给快递员。

运单号	运单状态	物品类型	重量/数量	揽收人	发件人	收件人	发件时间	收件时间	详情
1 1653385666	已签收	图书	16	100	1002	1001	2022/05/24-17:31	2022/05/25-07:32	Books
2 1654699777	已签收	易碎品	4.40	146	1001	1002	2022/06/08-22:32	2022/06/08-23:14	Items
3 1654704477	已签收	易碎品	5.60	146	1001	1002	2022/06/08-23:51	2022/06/08-23:59	null
4 1654827005	已签收	易碎品	6.40	159	1001	1002	2022/06/10-09:53	2022/06/10-09:55	null
5 1654827021	已签收	普通包裹	2.50	159	1001	1001	2022/06/10-09:53	2022/06/10-09:57	null
6 1654830379	待签收	易碎品	8.80	159	1001	1002	2022/06/10-10:49	null	null
7 1656304736	待签收	易碎品	9.60	146	1002	1001	2022/06/27-12:22	null	易碎品
8 1656304799	待签收	图书	10	未指定	1002	1001	2022/06/27-12:23	null	null
9 1656318328	待签收	易碎品	9.60	146	1002	1001	2022/06/27-16:08	null	null

6.5.2、快递搜索

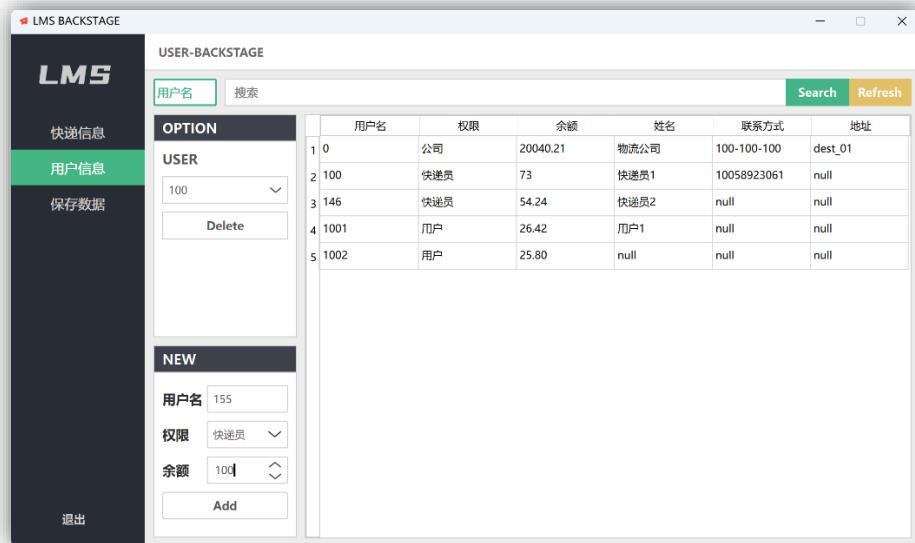
后台管理员可以在顶部的搜索条中选择搜索条件。第一栏是选择要指定的揽收人，第二栏则为选择匹配项目，在搜索框中输入搜索内容后，点击搜索按钮就可以查看相应的快递信息。点击最右侧的 **Refresh** 按钮则可以重新展示所有快递信息。

运单号	运单状态	物品类型	重量/数量	揽收人	发件人	收件人	发件时间	收件时间	详情
1 1654699777	已签收	易碎品	4.40	146	1001	1002	2022/06/08-22:32	2022/06/08-23:14	Items
2 1654704477	已签收	易碎品	5.60	146	1001	1002	2022/06/08-23:51	2022/06/08-23:59	null
3 1656304736	待签收	易碎品	9.60	146	1002	1001	2022/06/27-12:22	null	易碎品
4 1656318328	待签收	易碎品	9.60	146	1002	1001	2022/06/27-16:08	null	null

6.5.3、用户管理

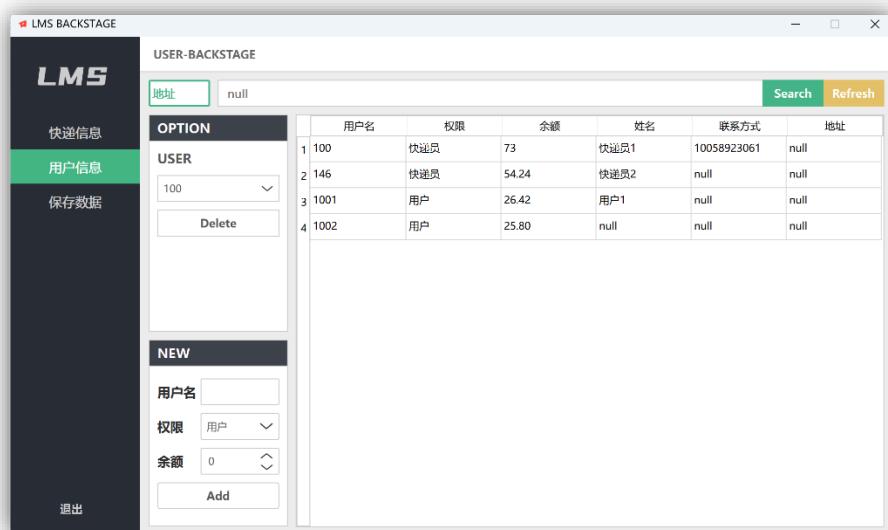
后台管理员可以在用户信息页面表格查看所有的用户信息。在左侧的 **OPTION** 栏中，可以选择对应的用户/快递员进行删除。删除用户涉及到一系列检查，如果用户正在登陆中，则该操作失败；如果用户是快递员，则所有该快递员未揽收的快递则将重新变为未分配快递员状态，等待分配至新的快递员。

在左侧的 **NEW** 一栏中，管理员可以指定用户名、权限和初始余额以添加用户，默认密码为 123。该操作的一处逻辑是让用户/快递员自行设定其他个人信息。



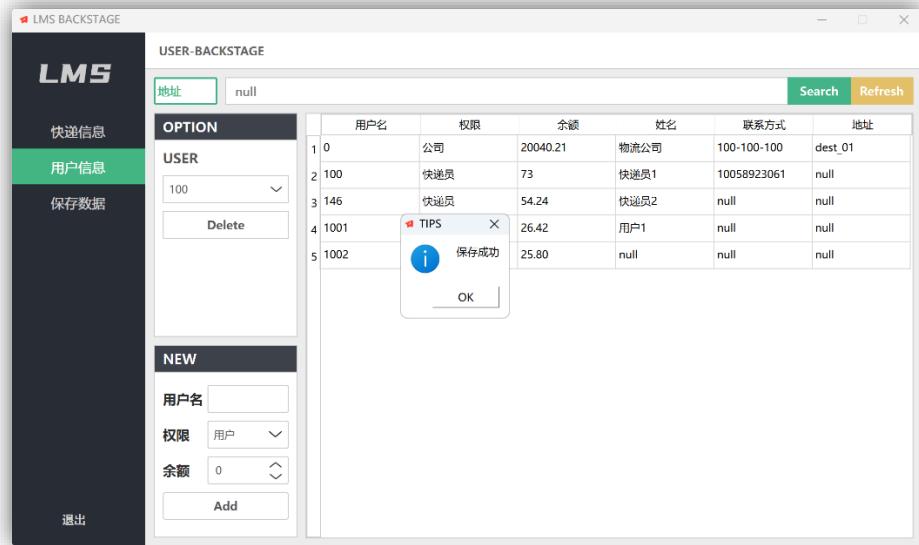
6.5.4、用户搜索

后台管理员可以在顶部的搜索条中选择搜索条件，在搜索框中输入搜索内容后，点击搜索按钮就可以查看相应的用户信息。点击最右侧的 Refresh 按钮则可以重新展示所有用户信息。



6.5.5、存储数据

在系统 3 分钟自动数据保存以外，后台管理员可以点击左侧导航栏的保存数据选项进行手动保存，该操作独立于自动保存外。



7、实验总结

7.1、问题及解决方案

实现过程中，考虑到完成多用户连接工作的功能，需要将 socket 连接变为非阻塞式，同时启动多线程进行独立管理。

因此，该系统利用 WSA 完成 socket 的非阻塞式连接，通过 accept 函数完成新连接的监听；在新连接建立时，通过传入服务端 socket 指针和 service 指针的结构体指针完成线程函数的参数传递。

另外，考虑到客户端异常退出时，服务端的该连接线程可能会占用资源，因此设置了 socket 连接的接收/发送数据超时时间，实现了客户端长时间无操作时服务端自动断开该连接，节约了服务器的线程资源。

数据方面，为防止多线程中的数据冲突，系统在两个 Controller 管理器中设置了锁标志，在每次检测中会先检测锁状态，如果有其他操作正在进行则等待 50ms 后再次请求。

7.2、经验教训

该网络版本由前版本二分离客户端与服务端而来。在初版设计时，对于模块的划分不够清晰，GUI 界面的点击事件通过直接操作底层控制器进行数据处理，导致 GUI 界面和数据服务的耦合度过高，因此在网络版本中对数据服务和 GUI 界面的点击事件函数进行了重构，各模块达到相对独立的状态，对于功能的增加和模块整合有了很大的帮助。

该版本设计中，所有的搜索操作依然对 vector 动态数组进行遍历搜索，如果数据量大，该方法效率会很低。未来的一个改进方向是采用二分查找、搜索树等查找算法或调用数据库进行数据管理，以提高数据操作的效率。

此外，该版本的锁设计实际上依然是单线程的处理模式，未来会通过研究多线程的锁机制以提高多线程数据处理的并发能力。