网络口罩预约系统项目实验报告

1、 Project 设计文档

① 系统结构:

本次项目设计在服务端和客户端都使用 python 语言进行，图形界面方面则使用了第三方库 pySide2 进行开发。

- 服务端：服务端程序的入口在 udp_server.py 内，主要负责启动图形界面。而服务端的主体逻辑都在 MainServer.py 中的 mainServer 类中实现，该类封装了用于 UDP 通信的监听套接字，全部的图形控件以及数据结构。
- 客户端：客户端程序的入口在 udp_client 内，主要负责打开套接字，启动登陆界面。登陆界面的全部逻辑位于 Welcome.py 内，在用户完成登陆操作后，Welcome.py 会创建 MainWindow 类进入程序的主界面
- 主要数据结构:

  服务端:

  admin：列表类型，保存所有的行政区名称

  adminDic：字典类型，最核心的数据结构，最外层的结构 key 值为行政区名，value 值对应一个新的字典，其中记录了关于该行政区的各项信息，主要包括“Usr“字段，记录所有申请该行政区内口罩预约的用户列表、”Activate“字段，记录是否启动预约，只可能有两个 value”是“或”否“，”Tatol“字段，该行政区最多可预约的口罩数量，本次项目中全部设置为 1000。

  dic：字典类型，模拟了数据库，记录所有注册用户的信息{用户名：密码}

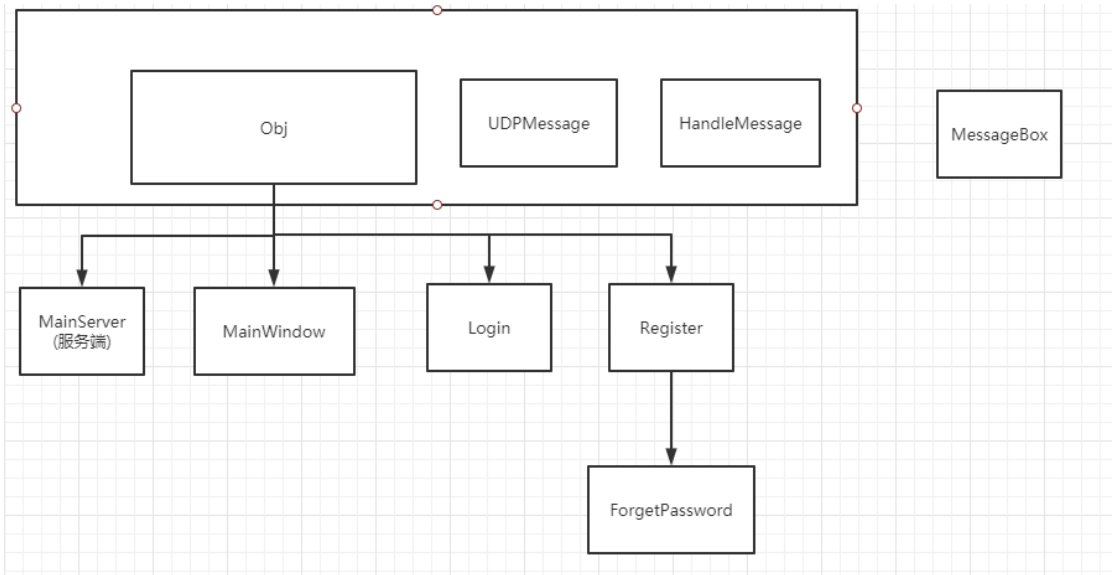
  onlineUser:字典类型，保存了所有当前在线用户的用户名、IP 地址以及端口号。

  客户端:

  admin：列表类型，同上

  adminDic：字典类型，同上
- ToolBox.py：本次项目实验中有关协议实现部分的细节大部分被封装在了这个文件中（除了服务端在生成 LoginSuccess 信号时，因为涉及到体量比较大的数据结构的传输，所以选择在类内部实现）

  本次项目所有自定义类的关系如下图所示:



其中 Obj 类、UDPMessage 类和 HandleMessage 类都位于 ToolBox.py 中，

Obj 类是本次实验中几乎所有实现类的基类、它主要封装了一些共有的基本信息，如 ui 资源的位置，udp 套接字，地址等。
UDPMessage 类包含两个静态方法，用于收发服务端和客户端的 UDP 数据报。
HandleMessage 类包含两个可供外部调用的静态方法，分别是 generate 方法和 read 方法，前者负责将数据封装在一个字典类中，后者负责分析收到的协议体。

② 应用程序协议

协议在实现过程中以 json 的格式进行传输，而协议内容分为以下几个部分：

- 消息头：也即消息类型，主要有以下几种

```
TYPE_LOGIN = 0x0  # 用户端登录请求
TYPE_REGISTER = 0x1  # 用户端注册请求
TYPE_FORGET_PASSWORD = 0x2  # 用户端忘记密码/修改密码请求
TYPE_QUIT = 0x3  # 用户端退出请求
TYPE_MESSAGE = 0x4  # 用户端发送消息（GroupMessage）请求
TYPE_BOOK = 0x5  # 用户端申请预约请求


TYPE_OK = 0x10  # 登陆通过响应
TYPE_FAIL = 0X20  # 登录失败（密码错误）响应
TYPE_REJECT = 0x30  # 登陆失败（重复登陆）响应
TYPE_LOGIN_SUCCESS = 0x40  # 登陆成功、每次更新行政区列表等核心数据结构时使用的消息头
TYPE_MESSAGE_BACK = 0x50  # 服务端发往客户端的消息
TYPE_BOOK_OK = 0x60  # 预约成功响应
TYPE_BOOK_FULL = 0x70  # 预约失败（预约名额已满）响应
TYPE_BOOK_REJECT = 0x80  # 预约失败（多次预约）响应
TYPE_BOOK_NOT_ACTIVATE = 0x90  # 预约失败（该行政区未启动预约）响应
TYPE_KICK = 0Xa0  # 用户被踢出预约队列通知
```

前半部分是客户端发往服务端的消息头，后半部分是客户端发往服务端的消息头。这些数据位于 json 格式的“Type”字段，每种消息头所代表的消息类型已在注释中给出。

- 消息体：

消息体的格式主要有消息头，也即消息类型所决定，每一部分的信息以统一的字段名进行区分，分别是“1”、“2”、“3”…在不同的消息头下，这些字段名都会携带不同意义的信息。
以 TYPE_MESSAGE 为例：

```
@staticmethod
def __generateMessage(str1=None, str2=None, str3=None, str4=None):
    message = {
        'Type': TYPE_MESSAGE,
        '1': str1,
        '2': str2
    }
```

“Type”字段位 TYPE_MESSAGE，“1”字段携带消息发送方的用户名，“2”字段携带消息正文。

③ 关键代码说明：

- UDPMessage 类：
  主要封装了收发消息的功能，代码如下图所示：

```
class UDPMessage:
    @staticmethod
    def sendUDPMessage(dataSocket, address, message):
        message = json.dumps(message).encode()
        dataSocket.sendto(message, address)

    @staticmethod
    def receiveUDPMessage(dataSocket):
        data,address = dataSocket.recvfrom(BUFFER_LENGTH)
        return data, address
```

  接受函数调用套接字接口获取数据，发送函数在发送前会用 json 对消息体经行编码。

- HandleMessage 类：
  如前文所述，主要有两个可供外部调用的方法，第一个是 read：
  调用方式：

```
data, address = UDPMessage.receiveUDPMessage(self.dataSocket)
ans = HandleMessage.read(data)
```

  将 UDPMessage 类返回的数据传入 read 函数。
  源代码：

```
class HandleMessage:
    @staticmethod
    def read(message):
        message.decode()
        message = json.loads(message)
        print(message)
        switch ={
            TYPE_OK: HandleMessage.__readOK,
            TYPE_FAIL: HandleMessage.__readFAIL,
            TYPE_REJECT: HandleMessage.__readREJECT,
            TYPE_LOGIN_SUCCESS: HandleMessage.__readLoginSuccess,
            TYPE_MESSAGE_BACK: HandleMessage.__readMessageBack,
            TYPE_LOGIN: HandleMessage.__readLogin,
            TYPE_REGISTER: HandleMessage.__readRegister,
            TYPE_FORGET_PASSWORD: HandleMessage.__readForgetPassword,
            TYPE_QUIT: HandleMessage.__readQuit,
            TYPE_MESSAGE: HandleMessage.__readMessage,
            TYPE_BOOK_OK: HandleMessage.__readBookOK,
            TYPE_BOOK_FULL: HandleMessage.__readBookFull,
            TYPE_BOOK_REJECT: HandleMessage.__readBookReject,
            TYPE_KICK: HandleMessage.__readKick,
            TYPE_BOOK_NOT_ACTIVATE: HandleMessage.__readBookNotActivate
        }
        ans = switch.get(message['Type'], HandleMessage.__default)(message)
        return ans
```

  对消息体的解码工作并不是由 UDPMessage 完成而是在 read 中实现的，所以仅仅还有一个参数，随后根据消息的“Type”字段，会继续调用不同的函数对消息正文进行分析。

这里以 TYPE_KICK 为例，以下是 read 函数所调用的__readKick 方法：
首先看一下 TYPE_KICK 的消息格式：

```
@staticmethod
def __generateKick(str1=None, str2=None, str3=None, str4=None):
    message = {
        'Type':TYPE_KICK,
        '1':str1
    }
    return message
```

“1”字段指示被提出的队列信息（行政区名）

```
@staticmethod
def __readKick(message):
    return "kick", '来自：' + "Server Admin" + '\n' + '正文：\n' + "您已被踢出 " + message['1'] + ' 的预约队列' + '\n' + '----------
```

该方法返回一个 kick 字符串来向应用程序返回消息类型，随后根据消息体内容组装字符串用于提示用户已被踢出队列。
在具体的实现中，由于服务端的处理需要涉及到许多内部数据结构的操作，所以服务端分析消息内容的工作主要在服务端应用程序内部独立进行而没有调用这些被封装的接口。这些接口主要供客户端应用程序进行调用。

generate 方法，
调用方法：

```
message = HandleMessage.generate(ans)
UDPMessage.sendUDPMessage(self.listenSocket, address, message)
```

其中 ans 是一个整数，指示了消息头的类型，generate 方法最多可以支持五个参数，后四个参数均为字符串，对应用消息体内不同的字段，在实际操作中，最多只用到了两个参数。
源代码：

```
@staticmethod
def generate(Type, str1=None, str2=None, str3=None, str4=None):
    switch = {
        TYPE_LOGIN: HandleMessage.__generateLogin,
        TYPE_REGISTER: HandleMessage.__generateRegister,
        TYPE_FORGET_PASSWORD: HandleMessage.__generateForgetPassword,
        TYPE_QUIT: HandleMessage.__generateQuit,
        TYPE_MESSAGE: HandleMessage.__generateMessage,
        TYPE_OK: HandleMessage.__generateOK,
        TYPE_FAIL: HandleMessage.__generateFAIL,
        TYPE_REJECT: HandleMessage.__generateREJECT,
        TYPE_LOGIN_SUCCESS: HandleMessage.__generateLoginSuccess,
        TYPE_MESSAGE_BACK: HandleMessage.__generateMessageBack,
        TYPE_BOOK: HandleMessage.__generateBook,
        TYPE_BOOK_OK: HandleMessage.__generateBookOK,
        TYPE_BOOK_FULL: HandleMessage.__generateBookFull,
        TYPE_BOOK_REJECT: HandleMessage.__generateBookReject,
        TYPE_KICK: HandleMessage.__generateKick,
        TYPE_BOOK_NOT_ACTIVATE: HandleMessage.__generateBookNotActivate
    }
    message = switch.get(Type, HandleMessage.__default)(str1, str2, str3, str4)
    print(message)
    return message
```

主要的实现思路和 read 函数大体一致，根据第一个参数的类型选择不同的函

数来调用生成不同的消息，这里以生成 TYPE_BOOK 消息为例：

```
@staticmethod
def __generateBook(str1=None, str2=None, str3=None, str4=None):
    message = {
        'Type':TYPE_BOOK,
        '1':str1,
        '2':str2
    }
    return message
```

“1”字段填入申请者的用户名，“2”字段填入申请的行政区名。

- 服务端和客户端之间的交互（基于多线程）：

服务端：

服务端在程序启动之初，会调用 mainServer 类的 launch 方法，该方法重写了 Obj 类的 launch 方法，启动了 UI 界面，并且开启一个监听线程用于接收客户端发来的请求。

```
def launch(self):
    thread = Thread(target=mainServer.__listen, args=(self,), daemon=True)
    thread.start()
    self.ui.show()

def __listen(self):
    self.ms.command.emit(self.ui.command, f"启动成功，正在端口{PORT}等待连接")
    while True:
        datagram, address = self.listenSocket.recvfrom(BUFFER_LENGTH)
        datagram = datagram.decode()
        datagram = json.loads(datagram)
        switch = {
            TYPE_LOGIN: self.login,
            TYPE_REGISTER: self.register,
            TYPE_FORGET_PASSWORD: self.forgetPassword,
            TYPE_QUIT: self.quit,
            TYPE_MESSAGE: self.groupMessage,
            TYPE_BOOK: self.book
        }

        case = switch.get(datagram["Type"], self.default)
        thread = Thread(target=case, args=(datagram, address))
        thread.start()
```

对于每一个发送到服务端的请求，监听线程会打开一个新的线程用于进行一些与消息头相对应的处理。

这里以 TYPE_LOGIN 为例：

```
def login(self, datagram, address):
    self.lock.acquire()
    username = datagram['1']
    password = datagram['2']
    result = self.dic.get(username, None)
    if result == password:
        if self.onlineUser.get(username, None) is None:
            userInfo = {username: address}
            self.onlineUser.update(userInfo)
            self.ms.command.emit(self.ui.command, self.onlineUser.__str__())
            ans = TYPE_LOGIN_SUCCESS
        else:
            ans = TYPE_REJECT
    else:
        ans = TYPE_FAIL

    if ans is TYPE_LOGIN_SUCCESS:
        message = self.generateLoginSuccessMessage()
    else:
        message = HandleMessage.generate(ans)
    self.lock.release()
    UDPMessage.sendUDPMessage(self.listenSocket, address, message)
```

Login 函数内需要操作一个服务端维护的“onlineUser“的字典类型，所以首先需要申请线程锁，随后对客户端发送的 username 和 password 进行比对，返回响应的结果。

客户端：

当客户端处与登录界面时，客户端应用程序工作在单线程的状态，一旦开启了客户端主界面，同样也会打开一个监听线程，用于处理服务端发回的响应，但这里没有对每一个响应另开启一个线程，而是全部在监听线程完成所有工作。以下是源代码：

```
def launch(self):
    thread = Thread (target=MainWindow.__listen, args=(self,), daemon=True)
    thread.start()
    self.ui.show()
    self.ui.exec_()

def __listen(self):
    try:
        while True:
            data, address = UDPMessage.receiveUDPMessage(self.dataSocket)
            ans = HandleMessage.read(data)
            if ans[0] == 'MessageBox':
                self.ms.receivedMessage.emit(self.ui.MessageBox, ans[1])
            elif ans[0] == 'update':
                self.ms.update.emit()
                self.lock.acquire()
                self.admin = ans[1].strip(',').split(',')
                self.adminDic = ans[2]
                self.lock.release()
            elif ans[0] == 'book':
                self.ms.receivedMessage.emit(self.ui.MessageBox, ans[2])
            elif ans[0] == 'kick':
                self.ms.receivedMessage.emit(self.ui.MessageBox, ans[1])
    except OSError:
        return
```

- 图形界面开发：

  由于本次实验采用了 pySide2 作为图形界面模块，所以做法大多参考了 pySide2 的官方文档，如在实现多线程下对等线程对主线程图形界面的变化时，主要通过 pySide2 自带的信号功能，完成。首先构建了一个 mySignal 类，其中定义了需要使用的信号类型。

```
class mySignal(QObject):
    receivedMessage = Signal(QTextBrowser, str)
    command = Signal(QTextBrowser, str)
    update = Signal()
```

这里主要有三种，第一个 receiveMessage 用于当监听端口收到聊天信息时需要更新主线程的聊天界面，第二个用于服务端更新控制台信息，第三个用户客户端接收到服务端更新的行政区信息后更新自己的主界面。

在使用这些信号时，需要在类内部声明成员变量，并调用 mySignal()方法初始化，随后将该函数绑定指定的信号处理函数，然后再子线程中对该信号的调用 emit 方法，发送信号。

除此之外，为了保证用户（无论是服务端还是客户端）所看到的信息全都是最新的，在 MainServer 和 MainWindow 类中还分别定义了 updateWindow()方法，在每次一核心数据结构被修改时都会调用这一方法对界面进行更新。

2、 用户使用手册

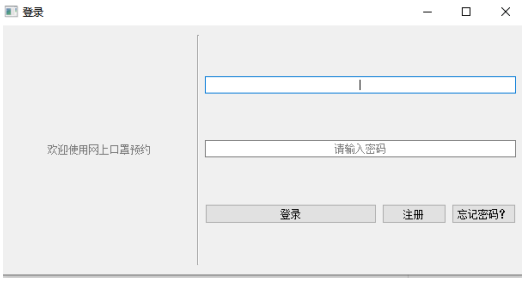
测试环境：Windows 10 python 版本 3.8

客户端：

① 启动应用程序：

由于开发的过程中使用了第三方库 PySide2，可能需要用户额外下载，所以这里推荐用户通过双击 launch_client.bat 的方法打开应用程序，对于 Windows 系统而言，如果环境变量支持 python 和 pip 的命令，那么 launch_client 会帮助用户从国内的镜像网站请求 PySide2 模块进行安装，随后再打开程序。

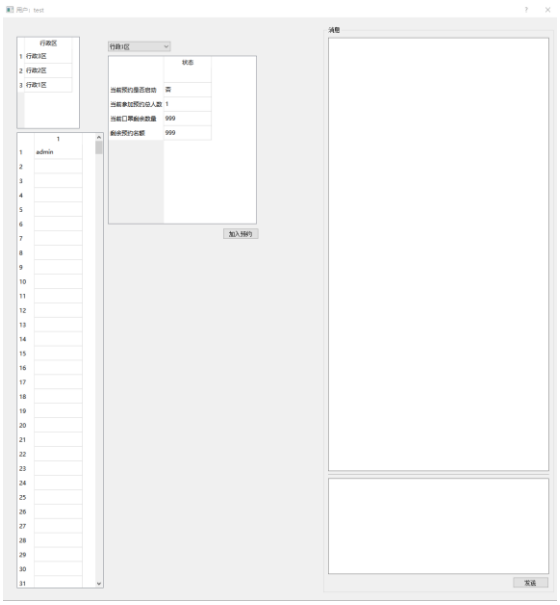
② 登陆界面



如图所示，输入用户名和密码即可登录，第一次使用时需要注册。“忘记密码？“选项在用户忘记自己的密码时可以点击重新设置自己的密码。

以下是注册界面和忘记密码界面

③ 进入主界面



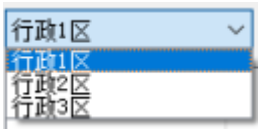
界面左侧为上方较小的表格显示当前可供选择的行政区列表，下方的条形表格显示当前的参与预约的用户列表。

界面中间部分上方的下拉选框可以修改当前选中的行政区，其下方的表格显示了关于当前选中行政区的具体信息，一旦改变下拉选框，其数据就会切换成与相对应的内容，最下方的按钮点击后可对当前选中的行政区发起预约申请。

界面右侧是消息界面，上方的文字框显示聊天信息，下方的输入框用于输入聊天内容，点击发送即可发出。

④ 发起预约申请

首先在下拉选框中选中想要预约的行政区



然后检查选框下方的详细信息，“当前预约是否启动“为”是“且剩余预约名额”大于 0 时，可以发起预约申请，直接点击下方的按钮即可。

如果预约成功，则会在右侧的消息界面显示申请结果：

预约成功：

列举预约失败的几种情况：

来自：Server Admin
正文：
对 行政1区 的预约失败
原因：
您已参与过预约

Server Admin
正文：
对 行政1区 的预约失败
原因：
该区域未启动口置预约

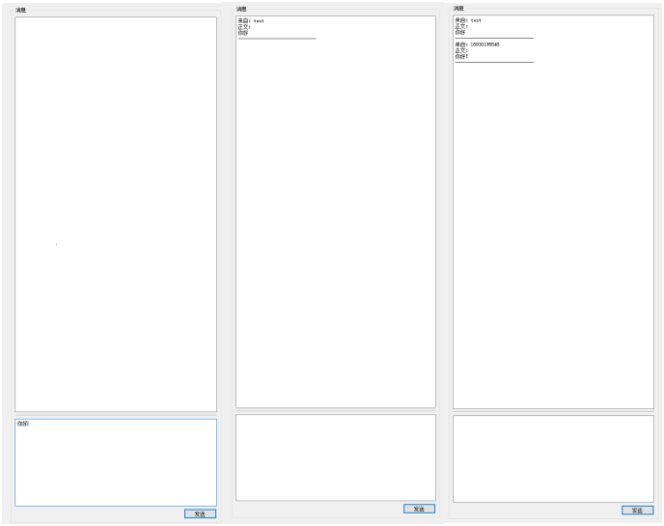
当您的一些不良举动被服务端管理员捕获，或者当前行政区的预约应意外情况被关闭时，您可能还会收到如下的信息：

来自：Server Admin
正文：
您已被踢出 行政1区 的预约队列

⑤ 消息界面

在右侧的消息界面您可以发送任何您想发送的内容，所有的内容都会被广播至包括管理员在内的所有在线用户，同样，您也会收到所有其余在线用户发送的消息，但管理员拥有对你发起私聊信息的权利。

需要注意的是您发送的每条信息的长度不可以超过 140 个中文字符，并且不能为空。输入后点击发送按钮即可



⑥ 退出应用程序

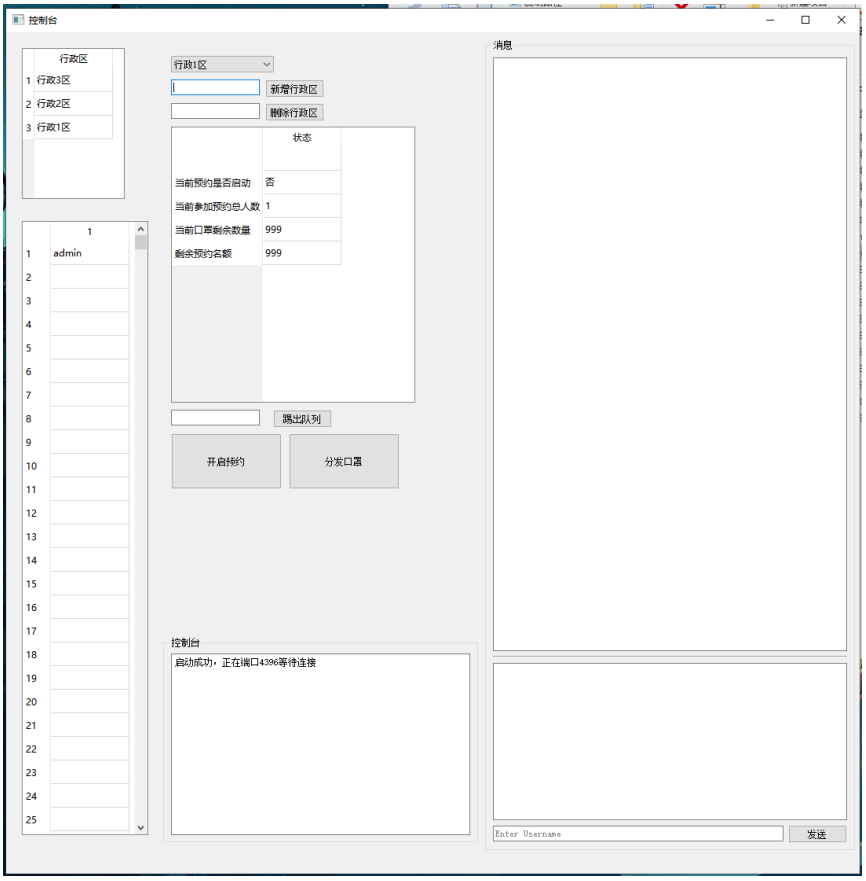
打开应用程序图形界面的同时也会打开一个命令行窗口，如果想要退出该程序，请务必点击图形界面的关闭按钮离开，等待 1 至 2 秒后命令行界面就会消失、请**务必不要以关闭命令行窗口的形式退出程序，**这样会导致服务端无法收到您离开的消息而导致无法登录。

服务端：

① 启动应用程序，同上，唯一不同的是需要点击 launch_server.bat

② 进入主界面

启动应用程序后会直接看到一个主界面，如下图所示



对于与客户端相同的部分这里就不多做赘述了，主要集中于几个服务端独有的部分：

删减行政区：在中间上方的两个可编辑框内可以输入想要删除或增加的行政区，随后点击相应的按钮即可

踢出某一用户：在屏幕中间的输入框中可以输入当前预约队列中的用户名，随后点击提出队列即可将该用户提出预约队列。

开启预约/分发口罩：点击这些按钮可以对当前下拉选框选中的行政区执行相应的操作。

控制台：服务端应用程序下方增设控制台功能，将会显示一些服务端用户操作时的提示或警告：如删除不存在的行政区、新增重名的行政区等。

③ 消息界面

主要功能和服务端保持大体一致，增设了向某一指定用户发送私聊信息的功能，只需在发送按钮左侧的可编辑栏内输入对应用户的用户名，如果对方用户在线则会发送发送栏中的内容。

注：如果在发送时上述可编辑栏为空，则消息将面向全体在线用户发送。