计算机网络Project： 网络聊天室

17307130178 宁晨然 17307130244 虞舒甜

1.综述

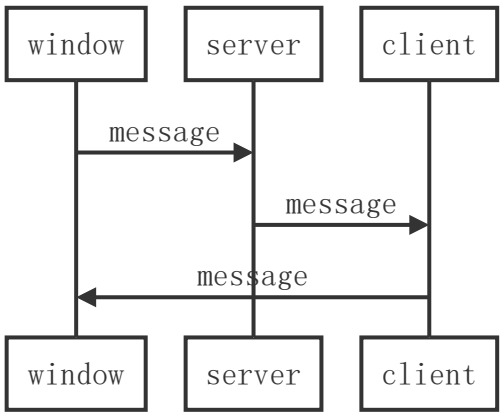
本实验由宁晨然和虞舒甜两人合作完成，实现了集登陆注册、群聊私聊、加群加好友等功能，以QT窗口前端做交互界面的网络聊天室。

2.实现

2.1 功能

- 用户登录
  - 用户名+密码形式登录
  - 数据库保留用户信息
- 用户注册
  - 用户名+密码注册
    - 用户名和密码内容安全检测
- 好友
  - 添加好友
  - 私聊
- 群聊
  - 加入群聊
  - 退出群聊
  - 新建群聊
  - 邀请好友进入群聊

2.2 基本结构

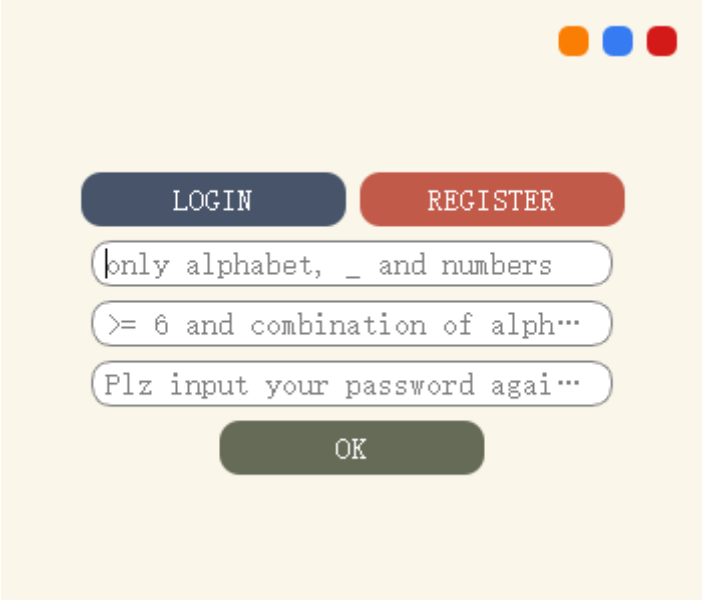


2.3 部分功能及界面展示

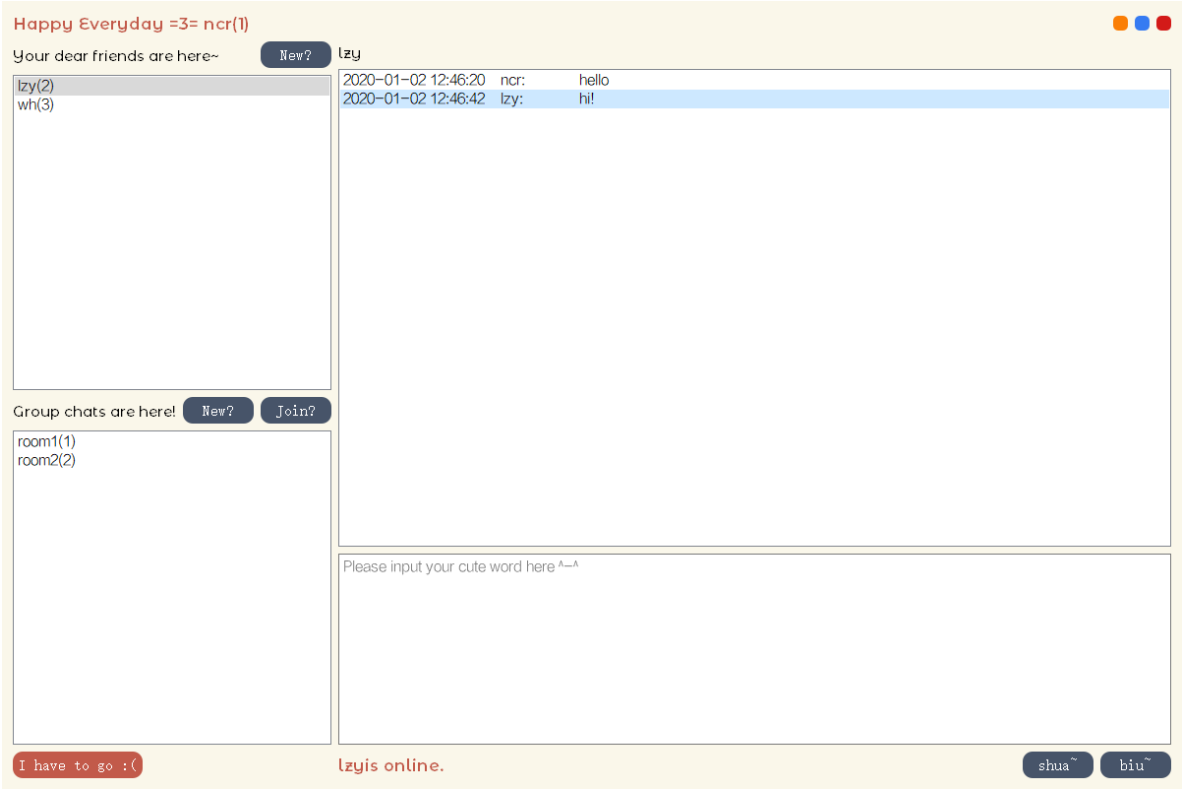
登录


LOGIN
REGISTER
Input your name here~
Input your password here~
OK

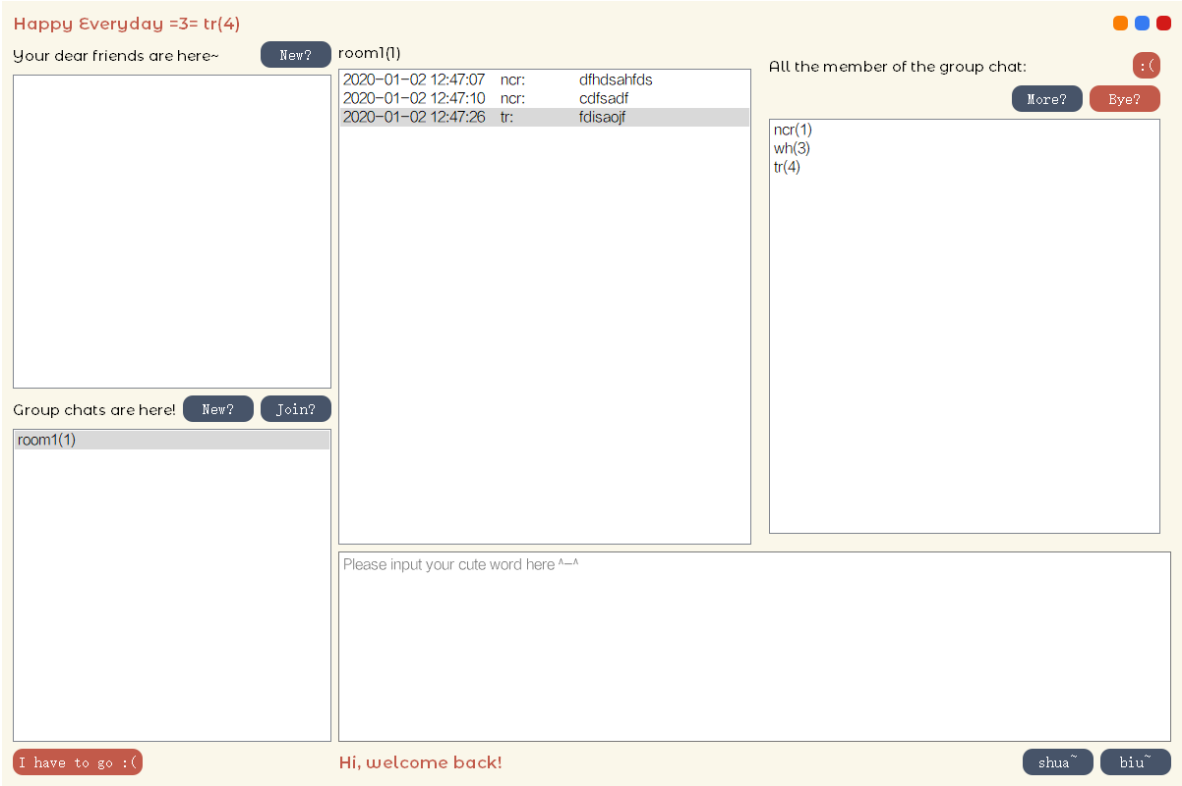
注册


LOGIN
REGISTER
only alphabet, _ and numbers
>= 6 and combination of alph…
Plz input your password agai…
OK

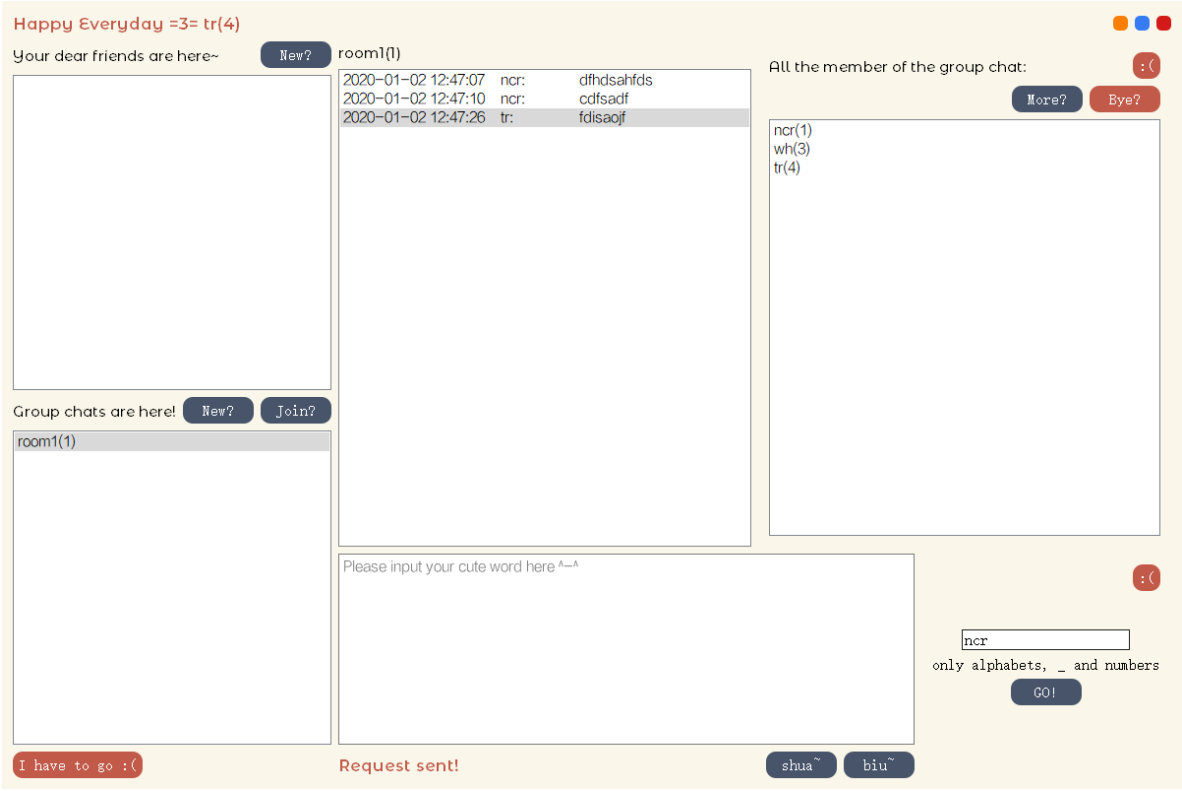
私聊

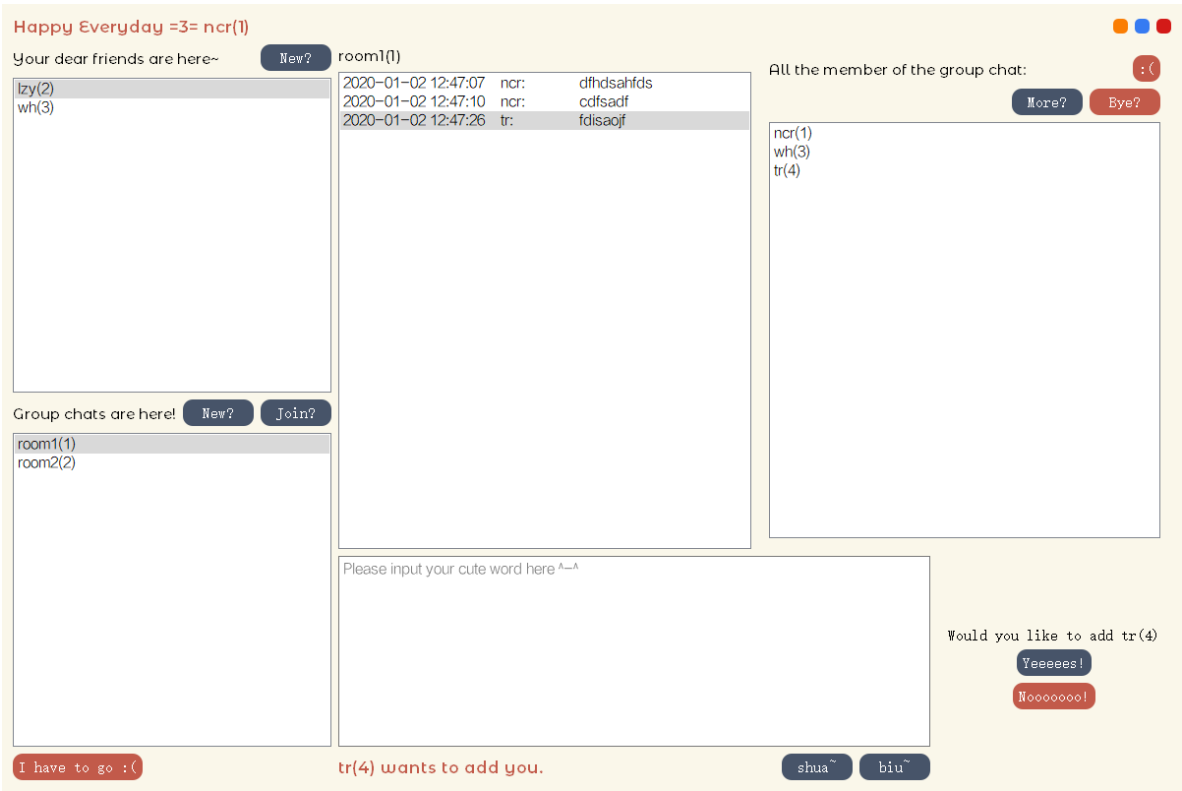

Happy Everyday =3= ncr(1)
Your dear friends are here~
New?
lzy
lzy(2)
wh(3)
2020-01-02 12:46:20 ncr: hello
2020-01-02 12:46:42 lzy: hi!
Group chats are here!
New?
Join?
room1(1)
room2(2)
Please input your cute word here ^—^
I have to go :(
lzyis online.
shua~
biu~

群聊

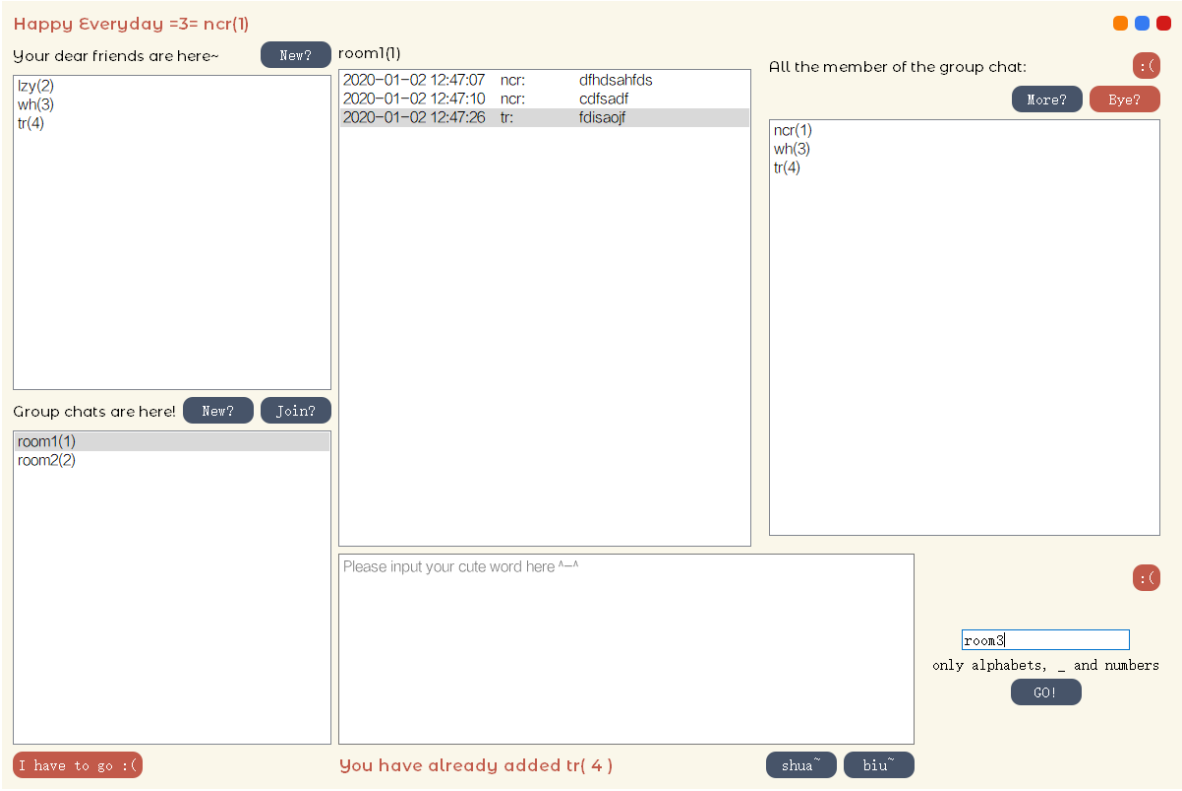

Happy Everyday =3= tr(4)
Your dear friends are here~
New?
room1(1)
2020-01-02 12:47:07 ncr: dfhdsahfds
2020-01-02 12:47:10 ncr: cdfsadf
2020-01-02 12:47:26 tr: fdisaojf
All the member of the group chat:
:(
More?
Bye?
ncr(1)
wh(3)
tr(4)
Group chats are here!
New?
Join?
room1(1)
Please input your cute word here ^—^
I have to go :(
Hi, welcome back!
shua~
biu~

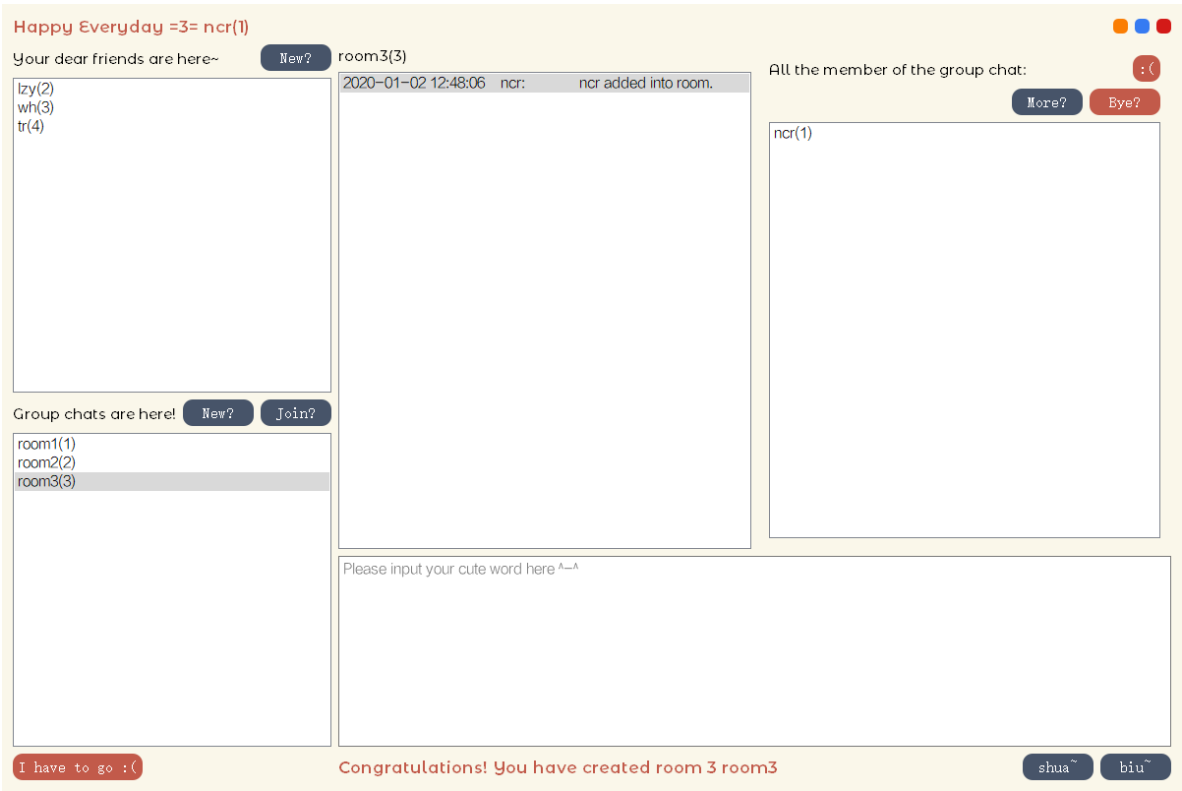
添加好友


Happy Everyday =3= tr(4)
Your dear friends are here~
New?
room1(1)
2020-01-02 12:47:07 ncr: dfhdsahfds
2020-01-02 12:47:10 ncr: cdfsadf
2020-01-02 12:47:26 tr: fdisaojf
All the member of the group chat:
:(
More?
Bye?
ncr(1)
wh(3)
tr(4)
Group chats are here!
New?
Join?
room1(1)
Please input your cute word here ^—^
:(
ncr
only alphabets, _ and numbers
GO!
I have to go :(
Request sent!
shua~
biu~

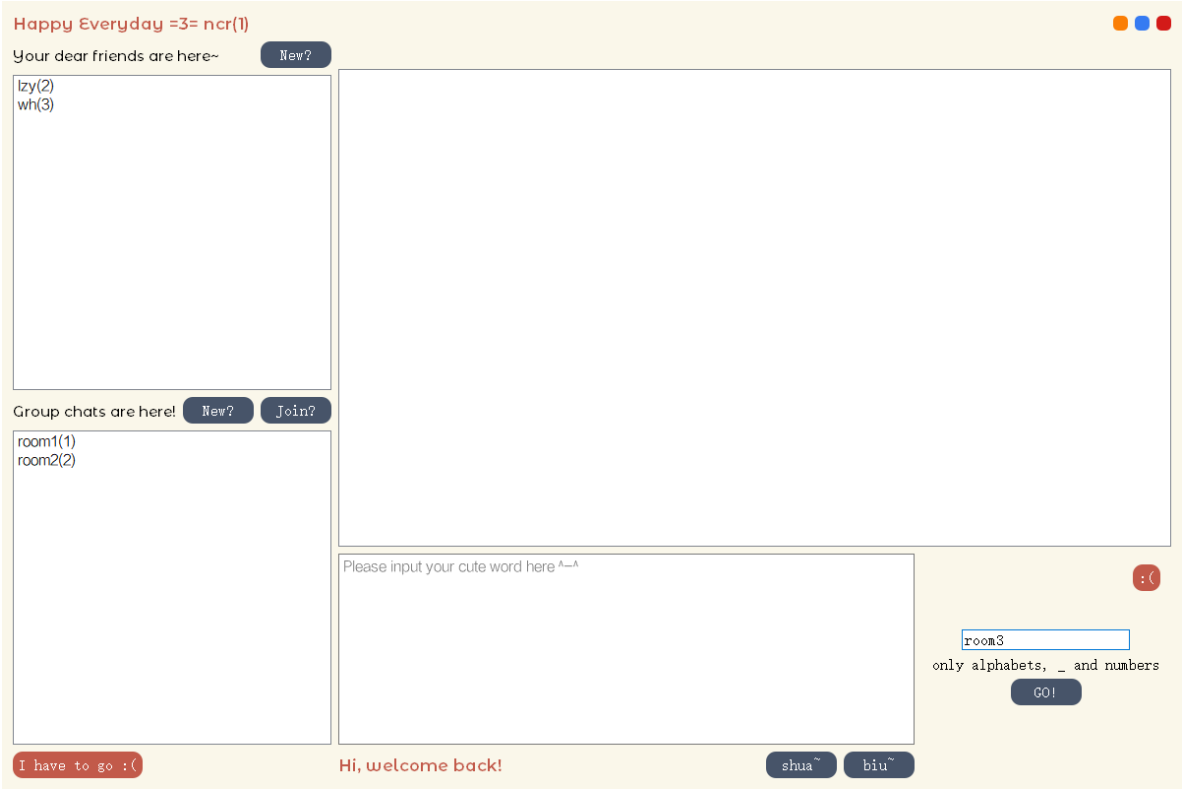

Happy Everyday =3= ncr(1)
Your dear friends are here~
New?
room1(1)
lzy(2)
wh(3)
2020-01-02 12:47:07 ncr: dfhdsahfds
2020-01-02 12:47:10 ncr: cdfsadf
2020-01-02 12:47:26 tr: fdisaojf
All the member of the group chat:
:(
More?
Bye?
ncr(1)
wh(3)
tr(4)
Group chats are here!
New?
Join?
room1(1)
room2(2)
Please input your cute word here ^—^
Would you like to add tr(4)
Yeeeees!
Nooooooo!
I have to go :(
tr(4) wants to add you.
shua~
biu~

创建群聊

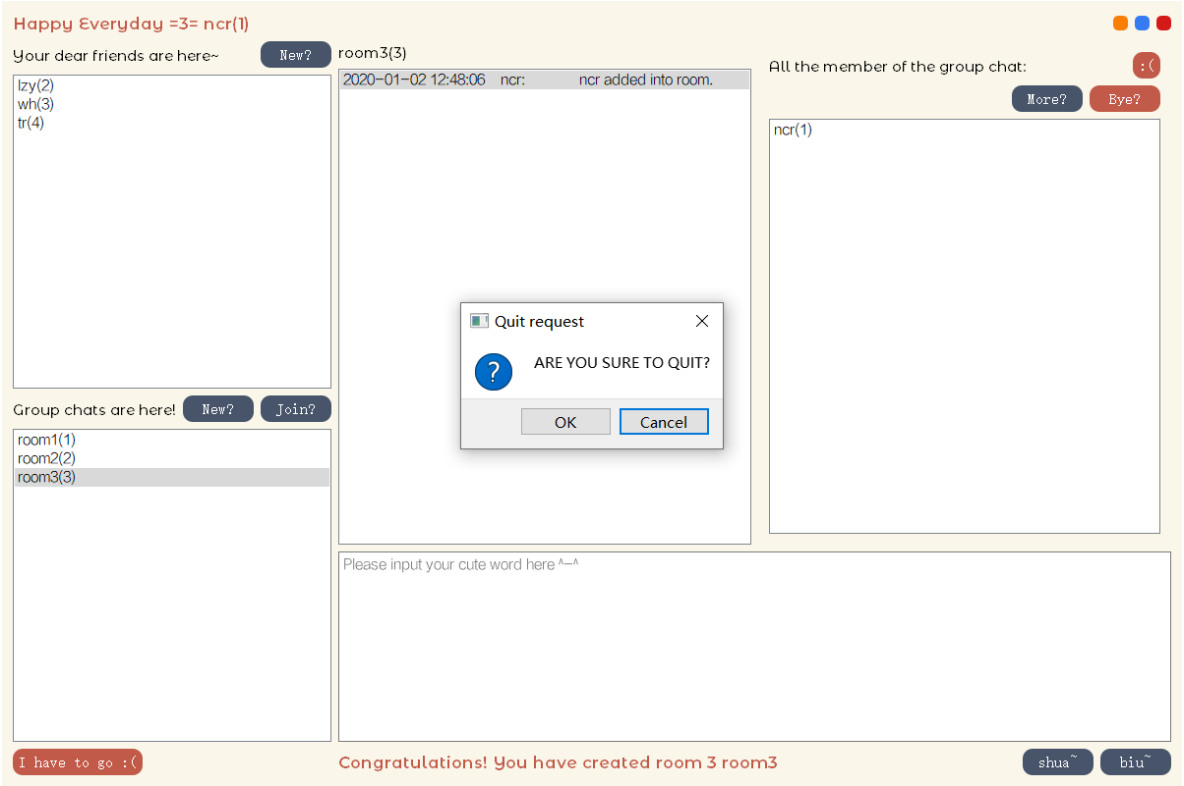

Happy Everyday =3= ncr(1)
Your dear friends are here~
New?
lzy(2)
wh(3)
tr(4)
Group chats are here!
New?
Join?
room1(1)
room2(2)
room1(1)
2020-01-02 12:47:07 ncr: dfhdsahfds
2020-01-02 12:47:10 ncr: cdfsadf
2020-01-02 12:47:26 tr: fdisaojf
Please input your cute word here ^—^
I have to go :(
You have already added tr(4)
shua~
biu~
All the member of the group chat:
:(
More?
Bye?
ncr(1)
wh(3)
tr(4)
:(
room3
only alphabets, _ and numbers
GO!


Happy Everyday =3= ncr(1)
Your dear friends are here~
New?
lzy(2)
wh(3)
tr(4)
Group chats are here!
New?
Join?
room1(1)
room2(2)
room3(3)
room3(3)
2020-01-02 12:48:06 ncr: ncr added into room.
Please input your cute word here ^—^
I have to go :(
Congratulations! You have created room 3 room3
shua~
biu~
All the member of the group chat:
:(
More?
Bye?
ncr(1)

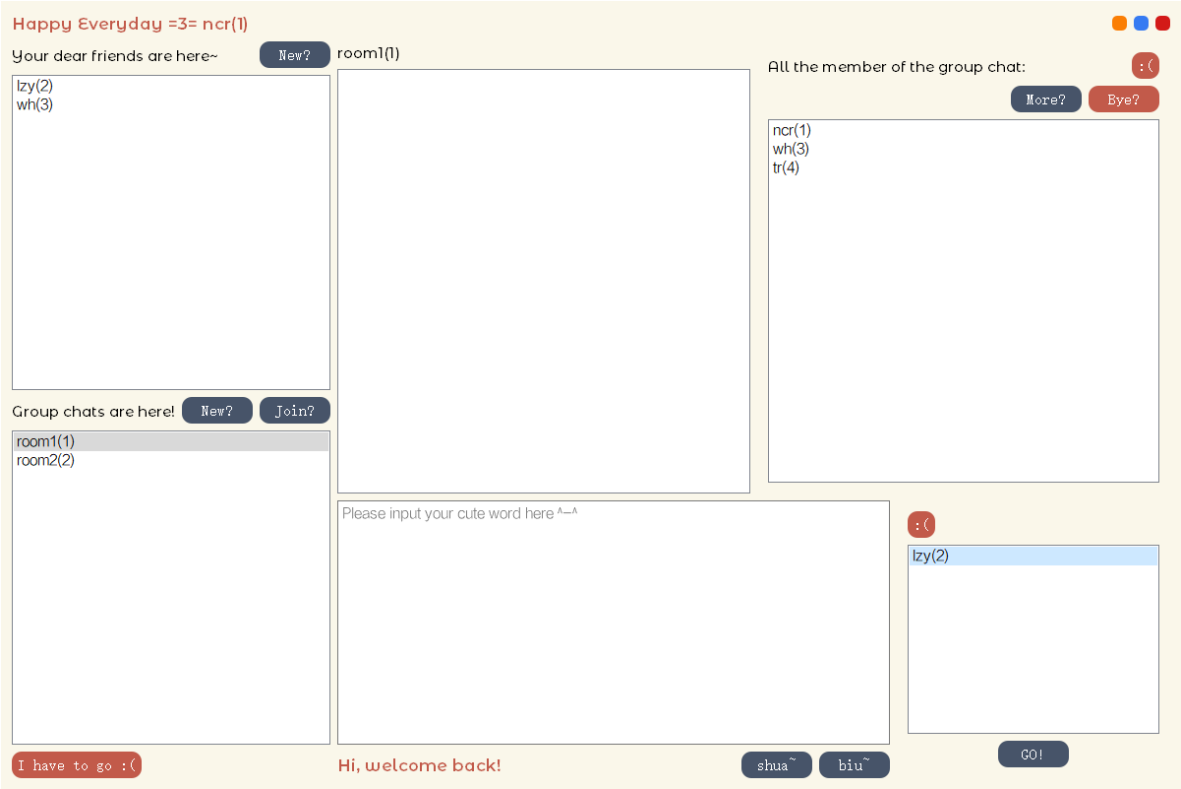
加入群聊



退出群聊



邀请好友加入群聊



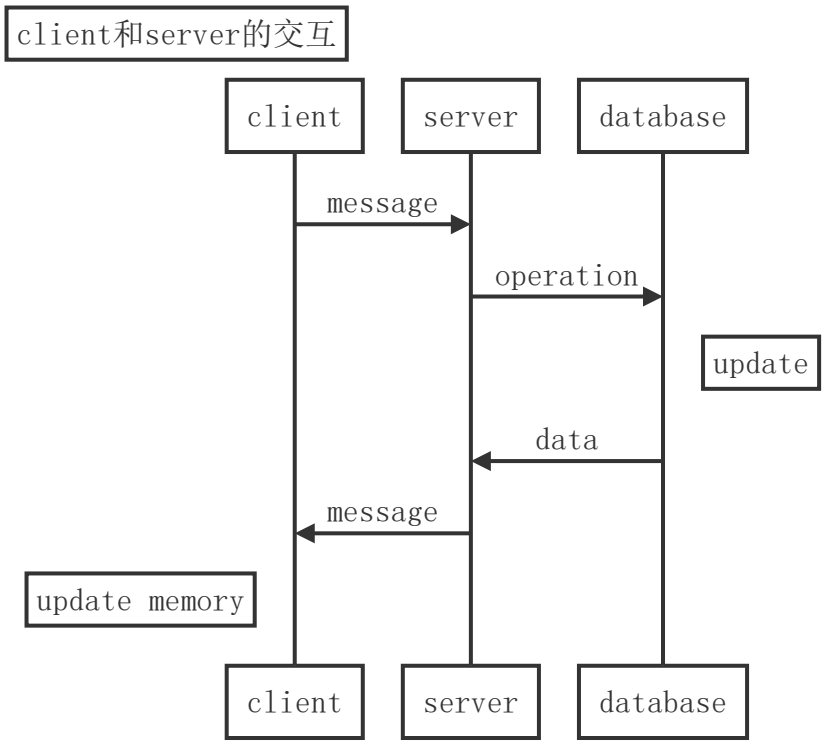
3.后端

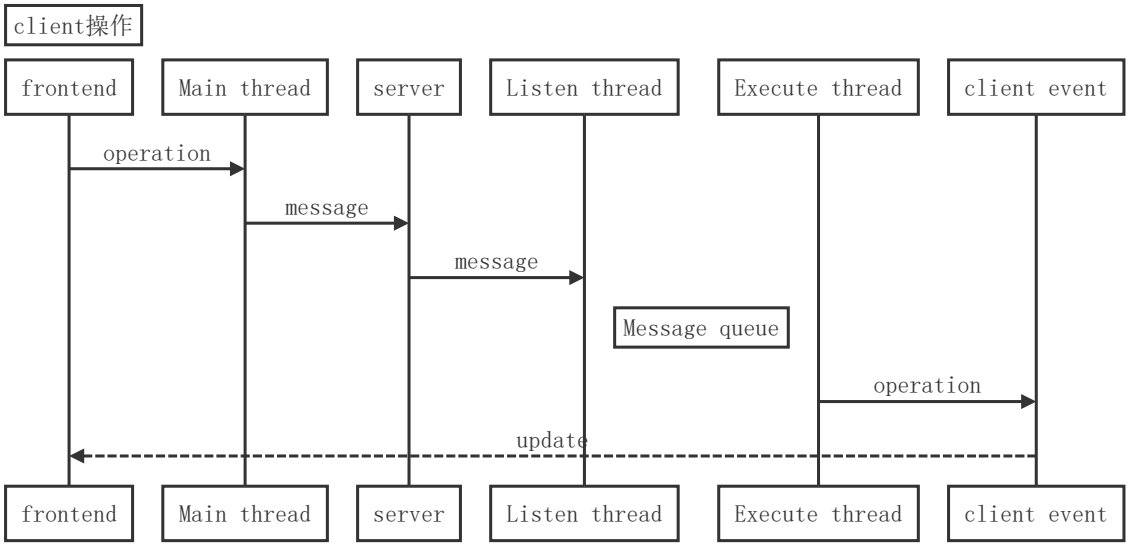
3.1 后端结构

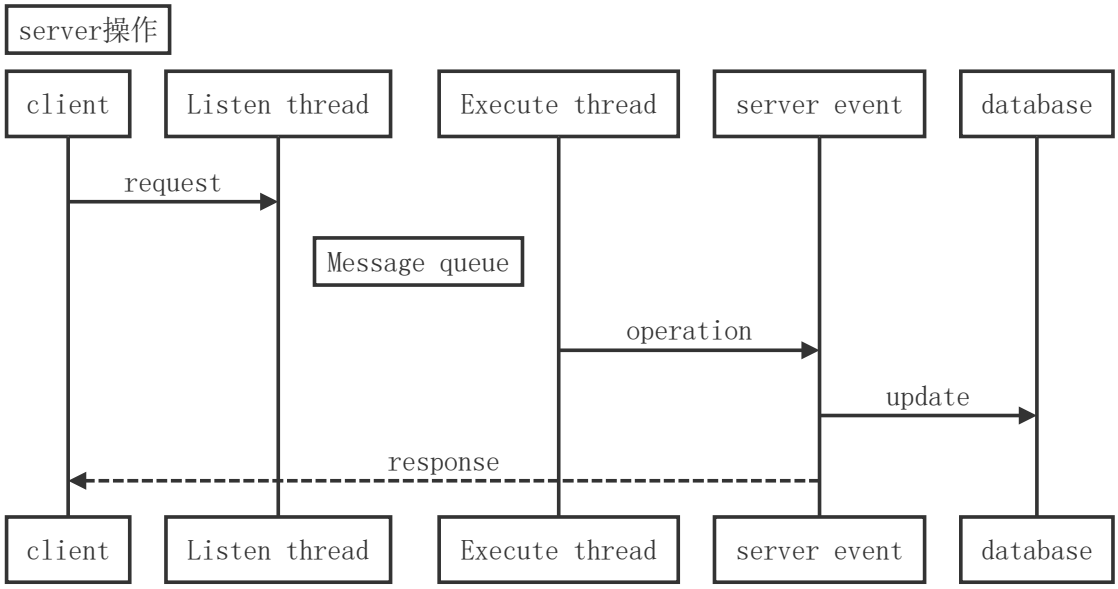
网络聊天室的结构分为：多个client和一个server。server和client采取**消息打包**传递方式通信。

- **client构成**：client本体/client_event处理器/client_memory
- **server构成**：server本体/server_event处理器/server_memory/sql数据库
- **message**：用于client和server之间的通信

3.2 后端示意图







3.3 后端详解

3.3.1 message协议

message的定义在`message.py`中，构建了一个server和client通信的协议。因为网络聊天室的通信属于应用层协议，所以类比了http的协议方式，主要的目的是为了将"指令"和"数据"分开。

message通信协议建立在**无连接**的基础上（或者是自动连接），server和client自动监听和接收，在message中包括了需要执行的指令、和指令需要的数据。

3.3.1.1 message结构：

- 分为method和entity两部分：method
- **method**：
  - 可以是server的response/ client的request/ server的failure信息等等
  - method是**指令集类型**，指令集定义在`class MessageType(enum.IntEnum)`。

```
# refer to HTTP
# method/state both use message type to present
# request
# |--method(4 Bytes)--|
# response
# |--state(4 Bytes)--|
# failure
# |--failure(4 Bytes)--|

# Entity body(not used in request)
# |--datatype(1 Byte)--|--datalen(4 Bytes)--|--data(N Bytes)--|
# ...
```

- **MessageType**:
  - 包含了server或者client需要的指令集（这里包括了部分，取了其中几个指令）
  - 一共有大约**60个**指令。涵盖了注册、发消息、错误检测等等各种指令。

```
class MessageType(enum.IntEnum):
    # Client Action:method 1-100
    # [username, password]
    login = 1
    # [username, password]
    register = 2
    # Server Action:state 101-200
    login_successful = 100
    register_successful = 101
    # [room_id, user_id, online]
    room_user_on_off_line = 112
    login_info = 113
    logout_successful = 114
    # Failure 201-300
    login_failed = 201
    username_taken = 202
    # err_msg:str
    general_failure = 203
    # msg:str
    general_msg = 204
```

- **Entity**：实体部分，既数据部分，此处采用只有**一个实体**
  - 实体结构为：`|--datatype(1 Byte)--|--datalen(4 Bytes)--|--data(N Bytes)--|`
  - 由于实体data可能有python各种类型的数据，比如dict/list/str等，但是消息传输的时候都是面向比特/字节的传输的“字符串”，所以需要对所有的数据进行**字节化**，可以成为序列化。收到消息后需要进行反序列化。
  - **序列化和反序列化**部分代码（`message.py`）：此处代码比较多，**有100行**，用于python各种数据的序列化和反序列化。因为不是重点而且重复比较多，贴上部分代码。

```
    def serialize_int(self, data):
        body = long_to_bytes(data)
        return bytes([VAR_TYPE_INVERSE['int']]) + pack('!L', len(body))
+ body
        def deserialize_bytes(self, bytes):
        return bytearray(bytes)
    def serialize_any(self, obj):
```

```
        if obj is None:
            return bytearray([0])
        type_byte = VAR_TYPE_INVERSE[type(obj).__name__]
        return self.serialize_by_type[type_byte](self,obj)

    def deserialize_int(self, bytes):
        return int.from_bytes(bytes, 'big')
    def deserialize_float(self, bytes):
        return unpack('!f', bytes)[0]
    def deserialize_str(self, bytes):
        return bytes.decode()
    def deserialize_list(self, bytes):
        byte_reader = ByteArrayReader(bytes)
        ret = []
        while (not byte_reader.empty()):
            body_type = byte_reader.read(1)[0]
            body = byte_reader.read(int.from_bytes(byte_reader.read(4),
byteorder='big'))
            body = self.deserialize_by_type[body_type](self, body)
            ret.append(body)
        return ret
```

3.3.1.2 message类的实现：

- 需要将method和entity结合起来，包括所有序列化、反序列化的方法。
- 封装在message类中：发送和接受的message类中的`serial`
- ```
  class Message:
      def __init__(self, message_type, data, serial = None):
          if serial == None:
              self.message_type = message_type
              self.entity = Entity_Body(data)
              self.serialize_message()
          else:
              self.serial = serial
              self.deserialize_message()
      def serialize_message(self):
          result = bytes([self.message_type.value])
          result += self.entity.serial
          self.serial = result
      def deserialize_message(self):
          ret = {}
          byte_reader = ByteArrayReader(self.serial)
          self.message_type = MessageType(byte_reader.read(1)[0])
          self.entity = Entity_Body(0, byte_reader.read_to_end())
  ```
- message这里没有过多限制消息大小、serial长度。

3.3.2 client

3.3.2.1 client本体类

采用线程池的方式分开所有多线程，使用生产者-消费者模型。因为client前端可能产生事件（比如鼠标点击更新数据库等等操作），而server可能不停的在给client发送消息，所以也需要监听，这样可能造成client处理不过来，所以必然需要使用多线程。
  ```

在client中可能会访问client本地的数据库，所以任何client事件都需要加锁机制，防止数据库被重复操作出错。

注意三个线程是分别的作用：

- client listening：监听server来的消息，放进消息队列
- client execute：监听消息队列，执行消息
- client main：阻塞，等待前端操作；执行前端操作

3.3.2.2 多线程的作用

- 这里使用的是线程池threading pool便于管理所有线程，共**三个线程**
- 监听listening和执行executing机制分开
- 并且有主函数线程用于前端操作
- 防止一个线程阻塞所有client操作
- **listen采用阻塞监听，使用消息队列交给execute；execute使用异步执行，从消息队列中取指令执行。**

3.3.2.3 client类函数

- `__init__()`：
  - 包括client的host/port/addr/bufsize等内容的初始化
  - 初始化后立刻进入主程序，**主程序会启动监听和执行，主程序线程阻塞**
  - **错误处理机制**：如果出错，会退出主程序，client自我销毁。
  - ```
        def __init__(self, thread_pool):
            self.HOST = '127.0.0.1'
            self.PORT = 8998
            self.ADDR = (self.HOST, self.PORT)
            self.BUFSIZE = 1024*1024
            self.sock = socket.socket()
            self.thread_pool = thread_pool
            self.lock = threading.Lock()
            self.qlock = threading.Lock()
            self.username = ""
            self.queue = Queue(maxsize=10)
            self.main(self.sock, self.ADDR)
        def main(self, sock, ADDR):
            try:
                sock.connect(ADDR)
                print('have connected with server')
                print("Try to login!")
                all_task = [self.thread_pool.submit(self.execute),
    self.thread_pool.submit(self.listening,(sock))]
                # self.handle()
                # 此处塞住，不会继续进行
            except Exception:
                print('error')
                return
                sock.close()
                self.thread_pool.shutdown()
                sys.exit()
    ```

- `listen()`：
  - 单独开一个线程持续监听是否有server发来的message。
    ```

    - 如果监听到message则放进client的**消息队列**queue中，不执行其他任何操作。
    - 采用**阻塞监听**的机制，一直循环监听server。
    - 只放队列，**防止漏听**消息。
    - ```
      def listening(self, sock):
          while True:
              # print("Listening")
              recv_data = sock.recv(self.BUFSIZE)
              if recv_data:
                  # print("Listener get data\n",len(recv_data))
                  self.queue.put(recv_data)
              else:
                  print("WARNING:Server closed connection.")
                  break
      ```

- `execute()`：
    - 单独开一个线程持续执行client中queue消息队列中的message。
    - 如果queue不为空，取出第一个消息：
        - 消息反序列化，分出method和entity中的data。
        - 请求client数据库锁
        - 执行method指令：交给client_event处理机制`handle_event`
        - 释放client数据库锁
    - 采用异步执行的机制，一直阻塞在消息队列中。
    - 异步执行可以防止listening的消息过多，执行不过来的情况
    - ```
      def execute(self):
          while True:
              print("Execute waiting.")
              # 消费者
              recv_data = self.queue.get()
              n = Message(0,0, recv_data)
              print("Execute get data\n", n.message_type)

              # 数据库访问需要锁
              self.lock.acquire()
              handle_event(n.message_type, n.entity.data)
              self.lock.release()

              # self.queue.task_done()
              print("Execute task done.\n")
              if n.message_type == MessageType.logout_successful:
                  break
      ```

3.3.2.4 client主程序调用的操作函数

- **线程**：这里的操作函数在client主线程中
- **作用**：用于前端操作，主要用于给server发送消息
- 一共实现了10个client对server的操作
-

函数	操作	传参 data
login	登陆	["login", username, password]
send	发送消息	["send", str, target_id, targer_type] target_type:1群聊，0私聊；target_id：朋友id或群聊id
register	注册新用户	["register", username, password]
logout	登出	["logout", 0]
join	加群	["join", room_name]
create	创建群	["create", room_name]
atr	拉人进群	["atr", friend_id, room_id]
quit	退群	["quit", room_id]
add	好友申请	["add", friend_name]
resolve	处理申请	["resolve", friend_name]
- 这些操作函数是**前端调用的API，作用是给前端调用、给server发送消息（请求）。**
- **主要函数代码**：(节选其中几个函数的代码)
- ```
      def login(self, data):
          d = data[1]
          m = Message(MessageType.login, [data[1], data[2]])
          self.username = m.entity.data[0]
          print(m.serial)
          self.sock.sendall(m.serial)
      def send(self, data):
          data = {'message': data[1], 'target_id': int(
              data[2]), 'target_type': int(data[3])}
          m = Message(MessageType.send_message, data)
          self.sock.sendall(m.serial)
      def add_friend_to_room(self, data):
          # data = ["atr", friend_id, room_id]
          if int(data[1]) not in memory_friends.friends.keys():
              print("No such friend")
              return
          if int(data[2]) not in memory_rooms.keys():
              print("No such room!")
              return
          data = [int(data[1]), int(data[2])]
          self.sock.sendall(Message(MessageType.add_friend_to_room,
  data).serial)

      def quit_room(self, data):
          # data = ["quit", room_id]
          room_id = int(data[1])
  ```
  ```

```
        self.sock.sendall(Message(MessageType.quit_room,
room_id).serial)
        pass

    def add_friend(self, data):
        # data = ["add",friend_name]
        friend_name = data[1]
        if friend_name in memory_friends.friends.values():
            print(friend_name, " already added.")
            return
        self.sock.sendall(Message(MessageType.add_friend,
friend_name).serial)

    def resolve_friend(self, data):
        # data = ["resolve",friend_name]
        global memory_request
        friend_id = memory_request.pop(data[1])
        if friend_id:
            self.sock.sendall(
                Message(MessageType.resolve_friend_request,
friend_id).serial)
        else:
            print(data[1], " didn't send you request.")
```

- 比如`send(self, data)`：
  - 作用：给server连接的其他在线client发送群聊或者私聊消息
  - 类型：target_type中为0就是私聊，1是群聊；target_id就是目标发送的人或者群id。
  - 步骤：
    1. 先将要发送的data打包成message
    2. `data = {'message': data[1], 'target_id': int(data[2]), 'target_type': int(data[3])}`
    3. `m = Message(MessageType.send_message, data)`
    4. 再将Message通过socket发送给server处理
  - 发送的内容是message的序列serial`self.sock.sendall(m.serial)`
- 又比如`add_friend`：
- ```
      def add_friend(self, data):
          # data = ["add",friend_name]
          friend_name = data[1]
          if friend_name in memory_friends.friends.values():
              print(friend_name, " already added.")
              return
          self.sock.sendall(Message(MessageType.add_friend,
  friend_name).serial)
  ```
  - 作用：给server连接的某个client发送好友申请（需要对方同意）
  - 类型：需要发送想要添加的用户名字friend_name
  - 步骤：
    1. 检查是否在该client的好友列表中了
    2. 如果已经有了，就不操作提醒用户已经添加了
    3. 如果没有，给server发送加好友申请的请求消息
  ```

- 其他的函数的内容大同小异，此处不予赘述。

3.3.3 client_event

前面client类中的main线程中，前端可能会有发送请求给server的各种操作，而对于execute线程，可能有也有各种需要处理的操作。此处的`client_event.py`中的所有函数，是用于执行server发送的指令和数据，在`execute`线程中处理。

- `client_event`用于处理server来的消息
- 位于`execute`线程
- 需要访问数据库，使用前需要加锁
- 统一使用`handle_event`的方式处理任何`message_type`
- 写了**248行代码**

3.3.3.1 注意

- `client_event`属于后端操作，处理server发来的指令修改本地数据库
- 后端操作完后，需要将指令递交给前端，让前端（显示/执行/变换）等操作
- `client_event`和前端交互接口使用异步执行，也是使用了queue，即：
  - **client_event执行完后立刻将需要前端操作的指令放入前端queue，前端异步执行**

3.3.3.2 handle_event：便于处理不同类型的指令的总处理器

- **为了方便的处理各种不同类型的指令**，定义client可能需要处理的所有指令（如下），定义在数组中，每次从中取值调用对应的函数。
- ```
  event_handler_map = {
      MessageType.login_successful: login_successful,
      MessageType.login_failed: login_failed,
      MessageType.login_info: login_info,
      MessageType.friend_on_off_line: friend_on_off_line,
      MessageType.register_successful: register_successful,
      MessageType.on_new_message: on_new_message,
      MessageType.server_kick: server_kick,
      MessageType.general_failure: general_failure,
      MessageType.logout_successful: logout_successful,
      MessageType.join_successful: join_successful,
      MessageType.create_successful: create_successful,
      MessageType.someone_inroom: someone_inroom,
      MessageType.someone_outroom: someone_outroom,
      MessageType.me_outroom: me_outroom,
      MessageType.incoming_friend_request: incoming_friend_request,
      MessageType.add_friend_result: add_friend_result,
  }

  def handle_event(event_type, parameters):
      event_handler_map[event_type](parameters)
  ```
- **类型如上，可以按照分类如下（最重要的部分！）：**
  - login相关：成功登陆/登陆失败/登陆获取的信息/注册成功/成功登出
  - friend相关：好友上下线/好友申请结果/新好友申请
  - room相关：加群成功/创群成功/退群成功/有人进群/有人退群
  - message相关：新消息
  - error相关：一般错误
  ```

3.3.3.3 重要函数解析

1. `on_new_message`:新消息！！

- 处理新消息的能力是client最重要的部分之一，涉及前后端、数据库多处操作。
- 步骤：
  - 发来的data包括了消息的信息：发送者id/发送者名字/目标id/目标类型/时间/（可选群聊名字）
  - **存入client数据库**：需要分清群聊还是私聊，存入client_memory（在后面会讲）
  - 如果是群聊，target_type =1，id就是群聊id，将该消息添加到client_memory的群聊数据库
  - 如果是私聊，target_type =0，id就是用户id，可能是自己发的消息/别人发的消息，区分放进数据库
  - **前端显示**：异步显示，放入前端处理的queue中，数据同parameters
- ```
  def on_new_message(parameters):
      # 收到新消息：1.存入client数据库 2.terminal显示 3.前端显示
      # message = {"message",
  'sender_id','sender_name','target_type','time','target_id','room_name'}
      print("her:", parameters)
      info = [parameters['time'], parameters['sender_name'],
  parameters['message']]
      c = chat(info)
      global memory_user_info
      user_id = memory_user_info[0]
      print("On new message")

      # 1. 存入client数据库
      if parameters['target_type'] == 0:
          # 私聊:分自己发的还是别人发的
          global memory_friends
          if parameters["sender_id"] == user_id:
              # 自己发的
              memory_friends.add_chat(parameters["target_id"], c)
          else:
              memory_friends.add_chat(parameters["sender_id"], c)
          # 2. terminal显示
          c.show()

      else:
          # 群聊
          global memory_rooms
          room_id = parameters["target_id"]
          # add_message 里面有show
          memory_rooms[room_id].add_message(c)

      # 前端
      Message_queue.put(["on_new_message", parameters])
  ```

2. `login_info`：登陆信息

- **作用**：当登陆成功后，server不仅会告知client成功登陆，还给client发送该用户在server端存储的所有信息

  比如用户名、好友列表、好友聊天记录、群聊、群聊记录等等信息，统一放进`login_info`发送过来。
  ```

- 这个包可能会很大，如果client在server端的消息过多可能发送不了这么多信息（待解决）。
- **步骤**：
  - 发送的数据是：{"rooms","friends"}
  - 对于rooms信息：
    - 有多个room信息，每个room信息里面包括：群聊id/群聊用户/聊天记录
    - 对于每个room信息解码后存放在client_memory中（后面会讲）
  - 对于friends信息：
    - 有多个friend信息，包括：好友id/好友名字/聊天记录
    - 对于每个friend信息解码后存放在client_memory
  - **前端显示**：后端操作好后，**放进前端处理队列**queue中
- ```
  def login_info(parameters):
      # parameters ['rooms',]
      print("Client login info")
      # print(parameters)
      # store rooms info
      rooms = parameters["rooms"]
      global memory_rooms
      for r in rooms:
          chat_history = []
          for message in r["chat_history"]:
              m = Entity_Body(0, message[0]).data
              info = [m["time"], m["sender_name"], m["message"]]
              c = chat(info)
              # print("Adding chatting")
              chat_history.append(c)
          # print("\tChat sort")
          if len(chat_history):
              chat_history.sort(key=lambda x: x.time)
          room_id = r["room_id"]
          # print("\tMemory rooms")
          memory_rooms[room_id] = croom(
              room_id, r["room_name"], r["room_users"], chat_history)
          memory_rooms[room_id].show_all()
      # print("\trooms done.")

      # parameters['friends']
      friends = parameters["friends"]
      # print("Friends:",friends)
      global memory_friends
      for friend in friends:
          chat_history = []
          for message in friend["chat_history"]:
              m = Entity_Body(0, message).data
              info = [m["time"], m["sender_name"], m["message"]]
              c = chat(info)
              chat_history.append(c)
          if len(chat_history):
              chat_history.sort(key=lambda x: x.time)
          memory_friends.add_friend(
              [friend["id"], friend["username"], chat_history])
      print("Friends:", memory_friends.friends)
      Message_queue.put(["login_info", "0"])
  ```
  ```

3. `join_successful`：**加群成功**

- **作用**：加入群聊成功，需要存放群聊信息，更新前端
- **步骤**：
  - parameters中包括群聊所有信息。
  - 如同login_info一样存放所有群信息到client_memory中。
  - 前端显示：放进前端处理队列queue中。
- ```
  def join_successful(parameters):
      print("Join successfully!")
      # 添加群聊所有信息，paramters里面存放所有信息
      r = parameters
      print(r)
      chat_history = []
      for message in r["chat_history"]:
          m = Entity_Body(0, message[0]).data
          info = [m["time"], m["sender_name"], m["message"]]
          c = chat(info)
          # print("Adding chatting")
          chat_history.append(c)
      # print("\tChat sort")
      if len(chat_history):
          chat_history.sort(key=lambda x: x.time)
      global memory_rooms
      room_id = r["room_id"]
      memory_rooms[room_id] = croom(
          room_id, r["room_name"], r["room_users"], chat_history)

      memory_rooms[room_id].show_all()
      Message_queue.put(["join_successful", [room_id, r["room_name"]]])
  ```

3.3.3.4 其他简单函数

- 其他函数实现逻辑比较简单，此处不予赘述。都基本符合统一的步骤规则。
- **步骤：**
  - **后端数据库client_memory更新**
  - **前端显示更改的数据内容**
- ```
  def login_successful(parameters):
      # 登陆成功后可以获得[user_id, username]
      print("Login successfully!")
      Message_queue.put(["login_success", "0"])
      global memory_user_info
      memory_user_info.append(parameters[0])
      memory_user_info.append(parameters[1])
      print(memory_user_info)
  def login_failed(parameters):
      # 登陆失败可能是用户名或者密码错误
      Message_queue.put(["login_failure", "0"])
      print("Login failed!")
      print("Wrong username or password.")
  def friend_on_off_line(parameters):
      # 好友上线
      # print(parameters)
      # 直接在系统中广播
      if parameters[0]:
  ```

```
        print(parameters[1], "is online.")
    else:
        print(parameters[1], "is offline.")
    Message_queue.put(["friend_on_off_line", parameters])
def create_successful(parameters):
    # parameters = [room_id, room_name]
    print("Create successfully!")
    room_id = parameters[0]
    room_name = parameters[1]
    global memory_rooms
    memory_rooms[room_id] = croom(room_id, room_name, [memory_user_info])
    Message_queue.put(["create_successful", [room_id, room_name]])
def someone_inroom(parameters):
    # info = [room_id, user_id, username]
    global memory_rooms
    memory_rooms[parameters[0]].add_user(parameters[1], parameters[2])
    Message_queue.put(["someone_inroom", parameters])
def someone_outroom(parameters):
    # info = [room_id, user_id]
    global memory_rooms
    memory_rooms[parameters[0]].delete_user(parameters[1])
    Message_queue.put(["someone_outroom", parameters])
def me_outroom(parameters):
    # parameters = room_id
    global memory_rooms
    memory_rooms.pop(parameters)
    print("Quit room successfully!")
    Message_queue.put(["me_outroom", parameters])
def incoming_friend_request(parameters):
    # parameters = [user_id, user_name]
    global memory_request
    if parameters[1] not in memory_request.keys():
        memory_request[parameters[1]] = parameters[0]
    print(parameters[1], " wants to add you.")
    Message_queue.put(["incoming_friend_request", [
                       parameters[0], parameters[1]]])
def add_friend_result(parameters):
    # parameters = [BOOL, [id,name]]
    friend = parameters[1]
    print("Here")
    print(parameters)
    global memory_friends
    memory_friends.add_friend(friend)
    if parameters[0]:
        print("Resolve successfully!")
    else:
        print("Request successfully!")
    Message_queue.put(["add_friend_result", [parameters[0],
                                             parameters[1][0], parameters[1]
[1]]])
def register_successful(parameters):
    print("Register successfully!")
    Message_queue.put(["register_success", "0"])
    # print("register done!")
def general_failure(parameters):
    print("General failure!")
    # print("Could be wrong username or wrong password!")
    Message_queue.put(["general_failure", parameters])
```

3.3.4 client_memory

`client_memory`存放了`client`所有的数据库，存放在计算机内存中（没有部署数据库），而是直接使用了python的数据结构类型存放，下面简单介绍数据结构。

原则：

- 所有client都有本地数据库，每次创建的时候自动创建数据库，可以认为是client附着的属性
- 所有数据库操作，比如在execute中的数据库更新操作，都需要申请锁，防止数据库乱套

3.3.4.1 `memory_user_info`

- 结构很简单就是一个list，存放用户名和用户id`[username,userid]`

3.3.4.2 `memory_friends`

- `memory_friends = friends()`
- **friends**类：
  - 包括好友的用户名/id/聊天记录
  - **方法**：初始化/添加好友/添加聊天记录
  - ```
    class friends:
        def __init__(self, friends=[]):
            self.friends = {friend[0]: friend[1] for friend in friends}
            self.chat_history = {friend[0]: [] for friend in friends}
        def add_friend(self, friend):
            if friend[0] not in self.friends.keys():
                self.friends[friend[0]] = friend[1]
                if len(friend) == 3:
                    self.chat_history[friend[0]] = friend[2]
                    print("Chat_history:", friend[2])
                else:
                    self.chat_history[friend[0]] = []
        def add_chat(self, friend_id, chat):
            self.chat_history[friend_id].append(chat)
    ```

3.3.4.3 `memory_rooms`

- **结构**：`{room_id:croom()}`
- **croom**类型：
  - 包括群聊的群聊id/群名/聊天记录
  - **方法**：初始化/添加用户/删除用户/添加聊天记录
  - 这里的添加和删除都有一定的**错误处理机制**：判断是否在群里
  - ```
    class croom:
        def __init__(self, room_id=0, roomname=0, users=[], chat_history=
    []):
            self.room_id = room_id
            self.roomname = roomname
            self.users = {i[0]: i[1] for i in users}
            self.chat_history = chat_history
        def add_user(self, user_id, username):
    ```

```
            if user_id not in self.users.keys():
                self.users[user_id] = username
        def delete_user(self, user_id):
            if user_id in self.users.keys():
                self.users.pop(user_id)
        def add_message(self, c):
            self.chat_history.append(c)
            c.show()
```

3.3.4.4 `Message_queue`

- 结构：`Message_queue = Queue()`用于前端和后端通信的**前端任务队列**
- 此处涉及的都是前端和后端通信，放在第四章讲解。

3.3.5 server

client可以有多个实例，连接同一个server，所以server肯定需要多线程来处理多个client的操作，并且还有监听线程来处理新建连接。这里采用`multiconn_server.py`

server的结构同样分为 **server数据库、server数据库操作、server类、server事件处理、server数据。**

3.3.5.1 server类

- server类的结构比较简单，主要在于多线程的处理
- **主线程**：循环监听是否有新建连接的client，如果有则建立子线程分配出来处理该client
- **对待client的线程：**
  - 子线程1监听：监听client发送的请求，监听后放入queue中
  - 子线程2执行：执行client queue中的命令
  - 子主线程阻塞
- **注意：**
  - 由于有多个对待client的实例，而server端数据库操作一定需要**加锁**，防止子线程混乱
  - 锁在server的主线程中。
  - 对待client的线程同样使用**handle_event**的机制，交给server_event操作。
  - **错误处理机制**：
    - 如果client建立的连接超时会自动断联
    - 如果client操作出错会断联
    - 如果子线程出错会关闭

```
class server:
    def __init__(self):
        print("Server is starting")
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind(('127.0.0.1', 8998))  # 配置soket，绑定IP地址和端口号
        self.sock.listen(5)  # 设置最大允许连接数，各连接和server的通信遵循FIFO原则
        self.lock = threading.Lock()
        print( "Server is listenting port 8001, with max connection 10")
        index = 0
        create_database()
        while True:  # 循环轮询socket状态，等待访问
            connection, address = self.sock.accept()
            print("Address is:",address)
```

```
            index += 1
            # 当获取一个新连接时，启动一个新线程来处理这个连接
            thread.start_new_thread(self.child_connection, (index, self.sock,
connection))
            if index > 10:
                break
        self.sock.close()

    def child_connection(self, index, sock, connection):
        q = Queue(maxsize=6)
        thread_pool = ThreadPoolExecutor(6)
        def listening(connection):
            while True:
                print("Server listening.")
                buf = connection.recv(1024)
                if buf:
                    print("\tServer listener get data")
                    q.put(buf)

        def execute(connection):
            while True:
                print("Server executing.")
                buf = q.get()
                n = Message(0,0,buf)
                self.lock.acquire()
                # print("\tServer lock acquired.")
                handle_event(connection,n.message_type,n.entity.data)
                self.lock.release()
                # print("\tServer lock released.")

                q.task_done()
                print("\tServer task done.")
                if n.message_type == MessageType.logout:
                    break
        try:
            print("begin connecion ", index)
            connection.settimeout(500)
            all_task = [thread_pool.submit(execute,
(connection)),thread_pool.submit(listening,(connection))]
            wait(all_task, return_when=FIRST_COMPLETED)
            print("All task done.\n")
            connection.close()
        except socket.timeout:
            # 如果建立连接后，该连接在设定的时间内无数据发来，则time out
            print('time out')
            connection.close()
        except :
            print("Unknown.")
            connection.close()
        print("closing connection %d" % index)  # 当一个连接监听循环退出后，连接可以关
掉
        thread_pool.shutdown()
        thread.exit_thread()
```

3.3.5.2 server_memory

这个memory是server端的数据库1，用于存放建立的client实例的socket连接

```
sc_to_user_id = {}
user_id_to_sc = {}
```

- `sc_to_user_id`使用来从client的socket获得用户id
- `user_id_to_sc`是用来从client的id获得用户socket
- 因为socket是用来发消息和接收消息的连接层面，所以非常重要

3.3.5.3 create_database

这个是server端的数据库2，用于存放所有用户的所有数据，利用sql语句建立。

- server的数据库建立在sql中，使用了sqlite3

`sql`语句建立几个table，用于规范化的存放所有的用户数据。

类别	内容
chat_history	id/user_id/target_id/target_type/data/sent
friends	from_user_id/to_user_id/accepted
rooms	id/room_name
room_user	room_id/user_id
users	id/username/password

```
DROP TABLE IF EXISTS "main"."chat_history";
CREATE TABLE "chat_history"
(
    "id" INTEGER NOT NULL,
    "user_id" INTEGER,
    "target_id" INTEGER,
    "target_type" TEXT,
    "data" Blob,
    "sent" INTEGER,
    PRIMARY KEY ("id" ASC)
);
-- ----------------------------
-- Table structure for friends
-- ----------------------------
DROP TABLE IF EXISTS "main"."friends";
CREATE TABLE "friends"
(
    "from_user_id" INTEGER NOT NULL,
    "to_user_id" INTEGER NOT NULL,
    "accepted" INTEGER,
    FOREIGN KEY ("from_user_id") REFERENCES users("id"),
    FOREIGN KEY ("to_user_id") REFERENCES users("id")
);
-- ----------------------------
-- Table structure for rooms
-- ----------------------------
DROP TABLE IF EXISTS "main"."rooms";
CREATE TABLE "rooms"
(
    "id" INTEGER NOT NULL,
```

```
    "room_name" TEXT,
    PRIMARY KEY ("id" ASC)
);
-- ----------------------------
-- Table structure for room_user
-- ----------------------------
DROP TABLE IF EXISTS "main"."room_user";
CREATE TABLE "room_user"
(
    "room_id" INTEGER NOT NULL,
    "user_id" INTEGER,
    FOREIGN KEY ("room_id") REFERENCES rooms("id")
);
-- ----------------------------
-- Table structure for users
-- ----------------------------
DROP TABLE IF EXISTS "main"."users";
CREATE TABLE "users"
(
    "id" INTEGER NOT NULL,
    "username" TEXT,
    "password" TEXT,
    PRIMARY KEY ("id" ASC)
);
```

3.3.5.4 data.py

作用：用于提供对database操作的接口API，为了获取数据库中的内容或者对数据库进行删减修改。

代码：代码比较多，**有338行**，定义了多个数据库接口函数，下面截取部分。

例如：

- 创建数据库/获取用户名/获取用户id/获取群聊记录/获取用户信息
- 添加用户/删除群聊/添加群聊记录
- 等等

```
conn = sqlite3.connect('database.db',
isolation_level=None,check_same_thread=False)

def get_cursor():
    return conn.cursor()


def commit():
    return conn.commit()


# create database
def create_database():
    with open("../data/create_database.sql", 'r', encoding="utf-8") as f:
        c = get_cursor()
        content = f.read()
        c.executescript(content)
        f.close()


# create user(register)
```

```
def add_user(info):
    # info = [username, password]
    c = get_cursor()
    ifused = c.execute('SELECT * from users where username=(?)',
                       [info[0]]).fetchall()
    if ifused:
        return False
    c.execute('INSERT into users (username,password) values (?,?)',
              [info[0], info[1]])
    return True


# create room
def add_room(info):
    # info = [user_id, roomname]
    c = get_cursor()
    ifused = c.execute('SELECT * from rooms where room_name=?',
                       [info[1]]).fetchall()
    if ifused:
        return False
    c.execute('INSERT into rooms (room_name) values (?)', [info[0]])
    res = c.execute('SELECT id from rooms where room_name=?',
                    [info[0]]).fetchall()[0][0]
    c.execute('INSERT into room_user (room_id, user_id) values(?,?)',
              [res, info[0]])
    return True

def create_room_from_user(info):
    # info = [user_id, roomname]
    c = get_cursor()
    c.execute('INSERT into rooms (room_name) values (?)', [info[1]])
    return c.lastrowid


def check_user(info):
    c = get_cursor()
    return c.execute('SELECT * from users where username=? and
password=?',info).fetchall()


# add friend to room
def add_friend_in_room(info):
    # info = [roomname, user_id]
    c = get_cursor()
    ifhave = c.execute('SELECT * from rooms where room_name=?',
                       [info[1]]).fetchall()
    if not ifhave:
        return False
    c.execute('INSERT into rooms (room_name) values (?)', [info[0]])
    res = c.lastrowid
    c = get_cursor()
    ifin = c.execute('SELECT * from room_user where values=(?,?)',
                     [res, info[1]]).fetchall()
    if ifin:
        return True
    c = get_cursor()
    c.execute('INSERT into room_user (room_id,user_id) values(?,?)',
              [res, info[1]])
```

```
    return True
```

3.3.6 server_event

最核心的部分就是Server_event，因为所有对于client发送来的消息（指令）都是通过这个操作。

- 进入server_event需要先获取操作数据库2的锁，操作完毕后释放
- 操作函数通过handle_event调配（同client_event的方法）

3.3.6.1 handle_event

- 为了操作的简便，同理可以直接传参调用指令数组
- 由于已经定义好了每个messagetype是做什么事情，只需要调用event_handler就可以调用响应函数
- ```
  event_handler_map = {
      MessageType.login: login,
      MessageType.send_message: send_message,
      MessageType.register: register,
      MessageType.logout:logout,
      MessageType.resolve_friend_request: resolve_friend_request,
      MessageType.add_friend: add_friend,
      MessageType.join_room: join_room,
      MessageType.create_room: create_room,
      MessageType.add_friend_to_room:add_friend_to_room,
      MessageType.quit_room:quit_room,
  }

  def handle_event(sc, event_type, parameters):
      event_handler_map[event_type](sc, parameters)
      print("\t task done")
  ```
- handle_event前需要进行加锁，之后释放，因为涉及数据库操作。

3.3.6.2 重要函数解析

server_event比client_event复杂很多，因为client的操作只涉及client本地数据库操作，前端是异步操作。server虽然没有前端显示，但是操作需要先更新server数据库，还要给相应的client发送响应数据。

这部分代码很多，很重要，有361行代码，下面详细解析重要函数。

函数	操作	参数
login	登陆一个用户	[username, password]
send_message	给某些用户发送消息	{'message','target_id','target_type'}
register	注册某个用户	[username,password]
logout	某个用户登出	0
join_room	申请加群聊	room_name
create_room	创建群聊	room_name
add_friend_to_room	拉人进群	[friend_id, room_id]
  ```

函数_room	操作用户退群	参数n_id
add_friend	申请加好友	friend_name
resolve_friend_request	处理好友申请	friend_id

1. login
    - 步骤：
        - 检查用户名密码是否合法，若登陆失败发送失败消息
        - 若重复登陆发送失败消息
        - 登陆成功后，发送登陆成功消息
        - 获取好友申请、所有好友列表、用户聊天记录（群聊和私聊）打包进入login_info并发送
    - ```
      def login(sc, parameters):
          # parameters = [username, password]
          print("In login")
          global user_id_to_sc
          # print(user_id_to_sc)
          user_info = check_user(parameters)
          # print("\tuser_name ",parameters[0])
          # 1.登陆失败
          if not user_info:
              # print("\tServer sending login_failed")
              sc.send(Message(MessageType.login_failed,0).serial)
              return

          # 2.重复登陆
          user_id = user_info[0][0]
          # global user_id_to_sc
          if user_id in user_id_to_sc:
              sc_old = user_id_to_sc[user_id]
              sc_old = user_id_to_sc[user_id]
              # print("\tServer sending server_kick")
              sc_old.sendall(Message(MessageType.server_kick,0).serial)
              sc_old.close()
              remove_sc_from_socket_mapping(sc_old)

          # 3.登陆
          # 绑定user和client
          sc_to_user_id[sc] = user_id
          user_id_to_sc[user_id] = sc
          # 3.1 登陆成功
          # print("\tServer sending login_successful")
          sc.sendall(Message(MessageType.login_successful,[user_id,
      parameters[0]]).serial)

          login_info = {}
          # 3.2 获取好友申请 [{'id','username'},...]
          # print("\tServer sending incoming_friend_request")
          for friend in get_pending_friend_request(user_id):
              friend = [friend["id"], friend["username"]]

       sc.sendall(Message(MessageType.incoming_friend_request,friend).serial)
      ```
      ```

```
    # 3.3 获取用户所有群 [{'room_id','room_name','
room_user','chat_history'},...]
    login_info['rooms'] = get_room_info(user_id)

    # 3.4 获取用户好友列表 [{'id','username'},...] 且广播好友上线信息
    login_info['friends'] = get_friends_all(user_id)
    # print("Server:",login_info['friends'])
    for friend in login_info['friends']:
        if friend['id'] in user_id_to_sc.keys():
            # print("\tServer sending friend %d on off line
",friend['id'])

 user_id_to_sc[friend['id']].sendall(Message(MessageType.friend_on_off_
line, [True, parameters[0]]).serial)

    # 3.5 获取用户聊天记录
    # login_info['messages'] = get_chat_from_friends(user_id)
    # print("\tServer sending login_info")
    # print(login_info)
    sc.sendall(Message(MessageType.login_info, login_info).serial)
```

2. send_message
    - 步骤：
        - 获取发送信息的用户信息
        - 打包发送的消息
        - 如果是私聊，判断是否是好友、是否在线，视情况发送消息或者失败消息（注意两边都要发送聊天记录）
        - 如果是群聊，判断用户是否在群，获取所有群内在线用户，广播消息。
    - ```
      def send_message(sc, parameters):
          # 1.获取当前发送信息的用户信息
          # parameters:{'message','target_id','target_type'}
          user_id = sc_to_user_id[sc]
          sender = get_user(user_id)

          # 2.message
          message = {"message": parameters['message'],
                     'sender_id': user_id,
                     'sender_name': sender['username'],
                     'target_type': parameters['target_type'],
                     'time': time.strftime("%Y-%m-%d %H:%M:%S",
      time.localtime()) }
          # print(message["message"])

          # 3.私聊
          if parameters['target_type'] == 0:
              # 3.1 检查是否是好友
              if parameters['target_id'] == user_id:
                  str = 'You can\'t send message to yourself.'
                  print("\tServer sending general_failure")
                  sc.sendall(Message(MessageType.general_failure,
      str).serial)
                  return
              if not is_friend_with(user_id, parameters['target_id']):
                  str = 'You are not friends with '+
      get_user(parameters['target_id'])['username']
      ```
      ```

```
            print("\tServer sending general_failure")
            sc.sendall(Message(MessageType.general_failure,
str).serial)
            return

        # 3.2 发送方添加聊天记录
        message['target_id'] = parameters['target_id']
        print("\tServer sending on_new_message")

 user_id_to_sc[user_id].sendall(Message(MessageType.on_new_message,
message).serial)
        add_to_chat_history(user_id, message['target_id'],
message['target_type'],Entity_Body(message).serial,True)

        # 3.3 接收方添加聊天记录
        message['target_id'] = user_id
        sent = False
        target = parameters['target_id']
        if target in user_id_to_sc:
            sent = True
            print("\tServer sending on_new_message")

 user_id_to_sc[parameters['target_id']].sendall(Message(MessageType.on_
new_message, message).serial)
        # print("sent = ",sent)
        add_to_chat_history(parameters['target_id'],
message['target_id'], message['target_type'],
Entity_Body(message).serial, sent)

    # 4. 群聊
    if parameters['target_type'] == 1:
        message['target_id'] = parameters['target_id']
        message['room_name'] = get_room(parameters['target_id'])
['room_name']
        # 4.1 判断用户是否在群里
        if not in_room(user_id, parameters['target_id']):
            # print("\tyou not in room")
            sc.sendall(Message(MessageType.general_failure, 'You
haven\'t in room.').serial)
            return

        # 4.2 获取群聊中所有用户
        users_id = get_room_members_id(parameters['target_id'])
        print(users_id)
        for user_id in users_id:
            sent = False
            if user_id in user_id_to_sc:
                # print("\tServer sending on_new_message to ",user_id)

 user_id_to_sc[user_id].sendall(Message(MessageType.on_new_message,
message).serial)
                sent = True
        add_to_chat_history(user_id, message['target_id'], 1
,Entity_Body(message).serial,sent)
```

3. join
    - 步骤:

- 检查是否已经有该房间，否则发送加入失败
- 添加用户进房间
- 添加成功就发送房间内所有信息，更新用户数据库
- 把用户添加在该群所有在线用户的本地数据库
- 群里发送加群成功的消息

```
def join_room(sc, parameters):
    # parameters = room_name
    print("Server joining room:",parameters)
    user_id = sc_to_user_id[sc]
    room_id = get_room_from_name(parameters)["id"]

    # 1.检查是否有该房间
    if not room_id:
        data = "Join failed!"
        sc.sendall(Message(MessageType.general_failure,data).serial)
        return

    info = [room_id,user_id]
    user_name = get_user(user_id)["username"]

    # print("\t",user_id, user_name)
    # print("\t",room_id, parameters)
    # 2.添加用户进房间
    if add_user_in_room(info):
        print("\Add successful.")
        # 2.1 添加成功 发送房间所有信息，更新用户数据库
        room = get_room_info_from_room_id(room_id)
        sc.sendall(Message(MessageType.join_successful,room).serial)

        # 2.2 把该用户添加到该群所有"在线"用户的本地数据库中
        info.append(user_name)
        # info = [room_id, user_id, username]
        for member_id in get_room_members_id(room_id):
            if member_id in user_id_to_sc:
                member_sc = user_id_to_sc[member_id]

member_sc.sendall(Message(MessageType.someone_inroom,info).serial)

        # 2.3 群里发送一条加群消息
        message = user_name + " added into room."
        M = {'message':message,'target_id':room_id, 'target_type':1}
        send_message(sc,M)

    else:
        print("\tjoin failed!")
        data = "Join failed!"
        sc.sendall(Message(MessageType.general_failure, data).serial)
```

4. create_room

- 步骤：
  - 检查房间是否已经存在，否则发送失败消息
  - server数据库创建一个房间
  - 添加用户到房间
  - 反馈该用户房间信息、房间内广播加群消息

```
```

- ```
  def create_room(sc, parameters):
      # parameters = room_name
      user_id = sc_to_user_id[sc]
      room_name = parameters
      print("Creating room:",parameters)

      # 1. 判断房间是否已经存在
      if get_room_from_name(room_name) != None:
          data = room_name + " already exisited."
          sc.sendall(Message(MessageType.general_failure,data).serial)
          return

      # 2. 客户端数据库中创建一个房间
      room_id = create_room_from_user([user_id, room_name])
      print("\t room_id:",room_id)

      # 3. 添加用户到房间
      if add_user_in_room([room_id, user_id]):

          # 3.1 反馈该用户房间信息
          data = [room_id,room_name]
          sc.sendall(Message(MessageType.create_successful,data).serial)

          # 3.2 房间内广播信息
          username = get_user(user_id)["username"]
          message =  username + " added into room."
          M = {'message':message,'target_id':room_id,'target_type':1}
          send_message(sc,M)

      else:
          data = "Create failed!"
          sc.sendall(Message(MessageType.general_failure,data).serial)
  ```

5. add_friend
   - 步骤：
     - 判断是否是好友，否则发送失败
     - server数据库记录好友请求
     - 给好友发送好友请求，等待对方消解
   - ```
     def add_friend(sc, parameters):
         # parameters = friend_name
         friend_name = parameters
         friend = get_user_from_name(parameters)
         user_id = sc_to_user_id[sc]
         user_name = get_user(user_id)["username"]
         if friend:
             friend_id = friend["id"]
             if not is_friend_with(user_id, friend_id):
                 # 1.客户端数据库记录好友请求
                 add_friend_request(user_id, friend_id)
                 # 2.给朋友发送好友请求
                 if friend_id in user_id_to_sc:

      user_id_to_sc[friend_id].sendall(Message(MessageType.incoming_friend_r
     equest, [user_id,user_name]).serial)
         else:
             data = "Theres no such friend"
     ```

```
            sc.sendall(Message(MessageType.general_failure, data).serial)
            print(data)
```

6. 其他函数代码见`server_event`中，不予赘述。

4.前端

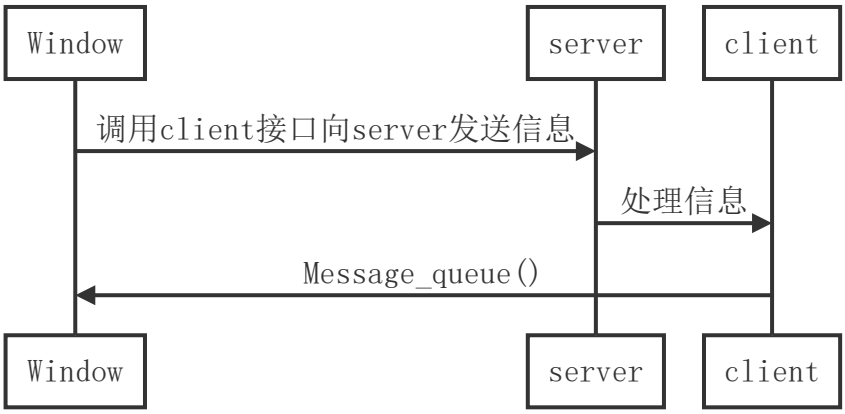
4.1 前端结构

前端主要由两个窗体结构构成：

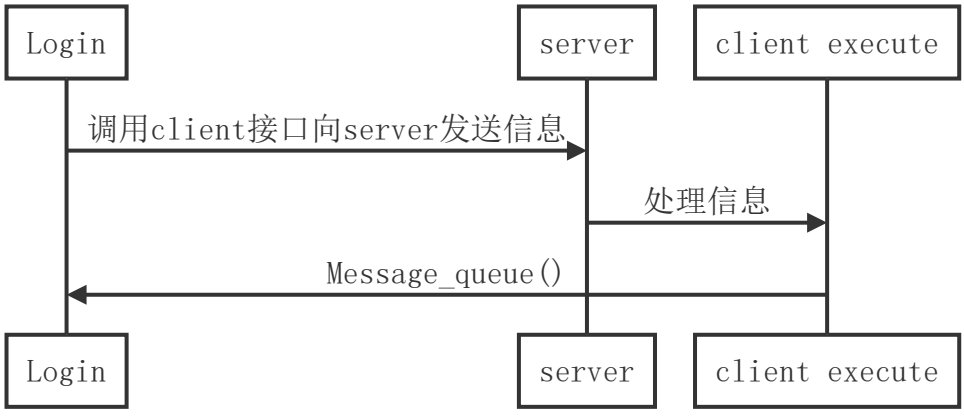
1. MainWindow: 聊天界面主窗口
2. Login: 登录窗口

4.2 前端示意图

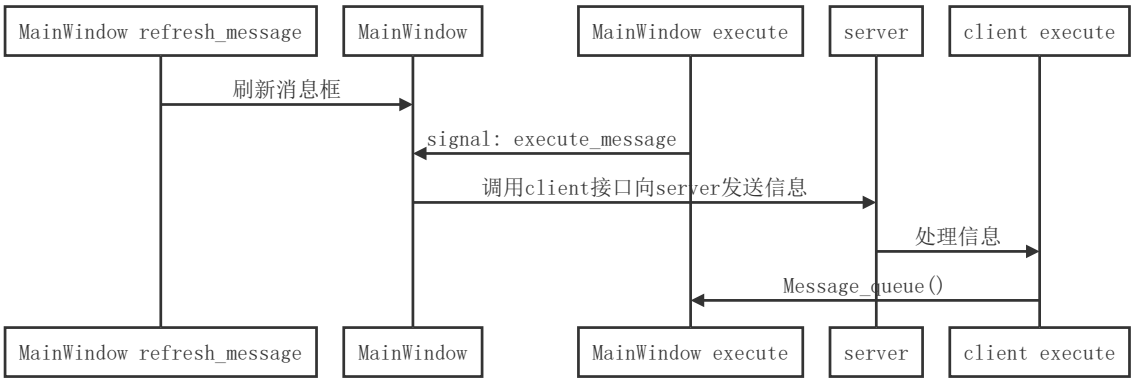
前端和后端的交互



Login窗口



MainWindow窗口



4.3 前端详解

前端使用pyqt进行编程，窗体使用类定义，功能函数以类接口的方式出现，并通过信号触发。

4.3.1 Login 类

Login窗口主要实现注册和登录的功能。窗口类定义在`window.py`的858行至1075行。

4.3.1.1 窗体主要成员

固定部件

login_button：QPushButton() 登录功能选择按钮

register_button：QPushButton() 注册功能选择按钮

account_input：QLineEdit() 用户名输入框

pwd_input：QLineEdit() 密码输入框

ok_button：QPushButton() 确认键

非固定部件

pwd_check()：QLineEdit() 密码确认输入框

其他功能成员

Ltag：int 功能选择标识，0为登录，1为注册

c：client 创建的client类

thread_pool：创建的线程池

```
# 在创建窗体时，将创建的client和线程池作为参数传入，赋值给类的成员。
self.c = client
self.thread_pool = thread_pool
```

4.3.1.2 主要功能函数`send()`

`send()`由`ok_button`触发。

在Login窗口只需要处理与登录注册有关的信息类型：login_success、login_failure、register_sucess、general_failure，并且不需要异步接收信息，因此不需要使用线程来处理信息。

```
    def send(self):
        username = self.account_input.text()
```

```
        pwd = self.pwd_input.text()
        data = ["login", username, pwd]
        if self.Ltag == 0:
            self.c.login(data)
            while True:
                if ~Message_queue.empty():
                    break
                time.sleep(0.1)
            recv_message = Message_queue.get()
            if recv_message[0] == "login_success":
                self.close()
                mw.mainlabel.setText("Happy Everyday =3= " + memory_user_info[1]
+ "(" + str(memory_user_info[0]) + ")")
                self.thread_pool.submit(mw.execute)
                self.thread_pool.submit(mw.refresh_message)
                mw.show()
            elif recv_message[0] == "login_failure":
                message = QMessageBox()
                message.about(self, 'LOGIN ERROR', 'Wrong password or wrong
username :(')
                self.account_input.clear()
                self.pwd_input.clear()
        else:
            namecheck = name_check(username)
            pwdcheck = password_check(pwd)
            pwd2 = self.pwd_check.text()
            if (not namecheck) or (not pwdcheck) or pwd != pwd2:
                message = QMessageBox()
                message.about(self, 'INPUT ERROR', 'Check the requirements for
name and password! ')
                self.account_input.clear()
                self.pwd_input.clear()
                self.pwd_check.clear()
            else:
                data = ["register", username, pwd]
                self.c.register(data)
                while True:
                    if ~Message_queue.empty():
                        break
                    time.sleep(0.1)

                recv_message = Message_queue.get()
                print(recv_message[0])
                if recv_message[0] == "register_success":
                    message = QMessageBox()
                    message.about(self, 'Register success', "Welcome! Let's
login!")
                    self.logincheck()
                    self.account_input.setText(username)
                elif recv_message[0] == "general_failure":
                    message = QMessageBox()
                    message.about(self, 'ERROR', 'I think you have already
registered!')
                    self.account_input.clear()
                    self.pwd_input.clear()
                    self.pwd_check.clear()
```

登录

首先通过`Ltag`判断当前选择功能，`Ltag`为0则为登录。

```
if self.Ltag == 0:
```

1. 发送消息

   调用client类中的对应接口函数即可。

   ```
    self.c.login(data)
   ```

2. 接收消息

   ```
   while True:
       if ~Message_queue.empty():
           break
       time.sleep(0.1)
   recv_message = Message_queue.get()
   ```

   消息的接收通过忙等来实现，窗口不停地检查`Message_queue`是否为空，非空的时候跳出循环，接收信息。

3. 判断消息类型并处理

   ```
   if recv_message[0] == "login_success":  # 登录成功
       self.close() # 关闭登录窗口
       mw.mainlabel.setText("Happy Everyday =3= " + memory_user_info[1] + "(" +
   str(memory_user_info[0]) + ")") # 显示登录信息
       self.thread_pool.submit(mw.execute) # 提交主窗口信息处理线程
       self.thread_pool.submit(mw.refresh_message) # 提交主窗口信息栏刷新线程
       mw.show() # 显示主窗口
   elif recv_message[0] == "login_failure": # 登录失败
       message = QMessageBox() # 弹窗显示登录失败
       message.about(self, 'LOGIN ERROR', 'Wrong password or wrong username :
   (')
       self.account_input.clear() # 清空输入框
       self.pwd_input.clear()
   ```

注册

1. 检查输入

   ```
   namecheck = name_check(username) # 检查用户名输入
   pwdcheck = password_check(pwd) # 检查密码输入
   pwd2 = self.pwd_check.text() # 二次密码输入
   if (not namecheck) or (not pwdcheck) or pwd != pwd2: # 如果输入错误
       message = QMessageBox() # 弹窗提示
       message.about(self, 'INPUT ERROR', 'Check the requirements for name and
   password! ')
       self.account_input.clear() # 清空输入框
       self.pwd_input.clear()
       self.pwd_check.clear()
   ```

```
def name_check(str): # 限制用户名只能由字母数字和下划线组成
    if len(str) == 0:
        return 0
    for i in range(len(str)):
        if not (str[i].isalpha() or str[i] == '_' or str[i].isdigit()):
            # 判断是否为字母数字下划线
            return 0
    return 1 and (not is_Chinese(str)) # 中文编码可能会被识别为两个普通字符

def password_check(str): # 限制密码长度≥6且只能由字母和数字组成
    if len(str) < 6:
        return 0
    return str.isalnum() and (not is_Chinese(str))
```

2. 发送信息

   同样调用接口。

```
self.c.register(data)
```

3. 接收消息

   同上登录功能，不做赘述。

4. 判断消息类型并处理

```
if recv_message[0] == "register_success": # 注册成功
    message = QMessageBox() # 弹窗提示
    message.about(self, 'Register success', "Welcome! Let's login!")
    self.logincheck() # 切换到登录功能
    self.account_input.setText(username)
elif recv_message[0] == "general_failure": # 注册失败
    message = QMessageBox() # 弹窗提示
    message.about(self, 'ERROR', 'I think you have already registered!')
    self.account_input.clear() # 清空输入
    self.pwd_input.clear()
    self.pwd_check.clear()
```

4.3.1.3 其他功能函数

`login_check()`和`register_check()`主要实现功能切换，在点击功能按钮时触发。

`login_check()`：`login_button`触发

```
def logincheck(self):
    self.Ltag = 0 # 设置功能标识
    self.login_button.setEnabled(False) # 设置登录键选中
    self.register_button.setEnabled(True) # 设置注册键未选中

    self.account_input.clear() # 清空输入
    self.pwd_input.clear()
    self.account_input.setPlaceholderText("Input your name here~")
    self.pwd_input.setPlaceholderText("Input your password here~")

    self.he.removeItem(self.he.itemAt(1)) # 删除二次输入密码框
    self.pwd_check.deleteLater()
```

`register_check()`：`register_button`**触发**

```
def registercheck(self):
    self.Ltag = 1 # 设置功能标识
    self.login_button.setEnabled(True) # 设置登录键未选中
    self.register_button.setEnabled(False) # 设置注册键选中
    # 创建二次输入密码框
    self.pwd_check = QLineEdit()
    self.pwd_check.setPlaceholderText("Plz input your password again ^-^ ")
    self.pwd_check.setEchoMode(QLineEdit.Password)
    self.pwd_check.setStyleSheet('QLineEdit{border:1px solid
gray;width:250px;border-radius:10px;padding:2px 4px;}')
    self.he.insertWidget(1, self.pwd_check)
    self.he.setStretchFactor(self.pwd_check, 4)
    # 清空输入
    self.account_input.clear()
    self.pwd_input.clear()
    self.account_input.setPlaceholderText("only alphabet, _ and numbers")
    self.pwd_input.setPlaceholderText(">= 6 and combination of alphabets and
numbers")
```

4.3.2 MainWindow 类

4.3.2.1 窗体主要成员

固定部件

friend_list：QListWidget() 好友列表/私聊列表

public_chat_list：QListWidget() 群聊列表

chat_history：QListWidget() 聊天记录显示

add_friends：QPushButton() 加好友功能

join_public_chat：QPushButton() 加入群聊功能

create_public_chat：QPushButton() 创建群聊功能

input：QTextEdit() 输入框

send：QPushButton() 发送信息

clear：QPushButton() 清空输入框

mainquit：QPushButton() 退出登录

message：QLabel() 消息提示栏

非固定部件

request_Widget：收到好友请求提示弹窗

add_Widget：加好友功能弹窗

create_Widget：创建群聊功能弹窗

join_Widget：加入群聊功能弹窗

pcf_widget：群聊成员列表弹窗

invite_Widget：邀请好友加入群聊弹窗

> 这里将所有弹窗和其中的所有部件整合成一个部件，便于后面的删除操作

其他功能成员

current_display：[聊天类型(0,1)，id]　标识目前显示的聊天界面，0表示私聊（id为好友id），1表示群聊（id为群聊id）

message_time：时钟，指示信息栏上一次刷新后的时间

execute_message：信号量，用于与信息接收线程通信

4.3.2.2 多线程

前端一共由三个线程组成：

1. MainWindow：主线程，与Client main实际上是一个线程
2. MainWindow execute：负责接收Client execute放入Message_queue的信息
3. MainWindow refresh_message： 负责刷新消息栏

MainWindow execute

```
def execute(self):
    while True:
        recv_message = Message_queue.get()
        print("Window get sth!")
        mw.execute_message.emit(recv_message)
```

该线程在一个无限的循环中不停地从`Message_queue`中获取`Client execute`发送的信息，一旦接收到信息，就会将接收到的信息直接通过`execute_message`信号传递给主线程。

```
execute_message = pyqtSignal(list)

self.execute_message.connect(self.mainexecute)
```

`execute_message`信号是一个带有list参数的信号。

在主窗口初始化时，将该信号与`mainexecute`函数绑定。当信号被激发时，会触发`mainexecute`函数，并且将list作为函数的参数传入。

```
def mainexecute(self, data):
    self.c.lock.acquire()
    self.message_dealer[data[0]](data[1])
    self.c.lock.release()
```

`mainexecute`会对参数进行第一步处理，将信息类型和其他参数分开。通过信息类型在`message_dealer`中寻找对应的处理函数并调用。

```
self.message_dealer = {
    "login_info": self.login_info,
    "friend_on_off_line": self.friend_on_off_line,
    "on_new_message": self.on_new_message,
    "general_failure": self.general_failure,
    "join_successful": self.join_successful,
    "create_successful": self.create_successful,
    "someone_inroom": self.someone_inroom,
    "someone_outroom": self.someone_outroom,
    "me_outroom": self.me_outroom,
    "incoming_friend_request": self.incoming_friend_request,
    "add_friend_result": self.add_friend_result,
    "logout_success": self.logout_success
}
```

MainWindow refresh_message

```
def refresh_message(self):
    while True:
        self.message_time = self.message_time + 1
        time.sleep(1)
        if self.message_time == 60:
            self.set_message_box("I'm the message box~")
```

该线程通过`message_time`来记录上一次更新`message_box`到现在的时长，每1s递增。当达到60s的时候，就更新`message_box`，避免同一信息停留时间过长。

```
def set_message_box(self, string):
    self.message.setText(string)
    self.message_time = 0
```

每次接收到`client execute`线程发来的信息，相应的处理函数都会调用`set_message_box()`，都会将`message_box`文本设为string，并且将`message_time`清零。

4.3.2.3 信息处理函数

```
self.message_dealer = {
    "login_info": self.login_info, # 表示已经录入登录信息
    "friend_on_off_line": self.friend_on_off_line, # 好友上下线
    "on_new_message": self.on_new_message, # 收到了新信息
    "general_failure": self.general_failure, # 系统错误
    "join_successful": self.join_successful, # 成功加入群聊
    "create_successful": self.create_successful, # 成功创建群聊
    "someone_inroom": self.someone_inroom, # 其他成员加入群聊
    "someone_outroom": self.someone_outroom, # 其他成员退出群聊
    "me_outroom": self.me_outroom, # 本人退出群聊
```

```
    "incoming_friend_request": self.incoming_friend_request, # 收到好友请求
    "add_friend_result": self.add_friend_result, # 添加好友结果
    "logout_success": self.logout_success # 退出成功
}
```

信息处理函数主要根据不同的信息类型，执行相应的操作。在这里不一一赘述，以`join_successful`为例进行分析。

```
def join_successful(self, data):
    # 解析参数
    room_id = data[0]
    room_name = data[1]
    # 修改当前展示聊天记录
    self.current_display = [1, room_id]
    # 刷新群聊列表
    self.refresh_pc()
    # 在群聊列表中选中当前群聊
    for i in range(self.public_chat_list.count()):
        m = self.public_chat_list.item(i)
        if m.data(ID) == room_id:
            m.setSelected(True)
            break
    # 刷新群聊信息
    self.refresh_ch(1, data[0])
    # 创建群聊成员弹窗
    self.create_pcf()
    # 设置消息栏
    self.set_message_box("Congratulations! You have joined room " + str(room_id)
+ " " + room_name)
    # 删除加入群聊功能弹窗
    self.delete_join_room()
```

4.3.2.4 弹窗实现

设计中一共有6个弹窗：request_Widget、add_Widget、create_Widget、join_Widget、pcf_widget、invite_Widget

每个弹窗都会有以下基本函数：

1. 创建函数
2. 功能按钮触发函数
3. 删除函数

接下来以request_Widget(收到好友请求提示弹窗)为例进行分析：

创建函数

```
def create_request(self, data):
    # 检查是否已经有弹窗 规定该位置只能出现一个弹窗
    # 如果有就删除
    self.create_check()
    # 创建弹窗
    self.request = QLabel()
    self.request.setText("Would you like to add " + data[1] + "(" + str(data[0])
+ ")")
    self.yesbut = QPushButton()
    self.yesbut.setText("Yeeeees!")
```

```
    self.ignorebut = QPushButton()
    self.ignorebut.setText("Noooooooo!")

    self.request_Widget = QWidget()
    lay = QVBoxLayout()
    lay.addStretch(1)
    lay.addWidget(self.request, 0, Qt.AlignCenter)
    lay.addWidget(self.yesbut, 0, Qt.AlignCenter)
    lay.addWidget(self.ignorebut, 0, Qt.AlignCenter)
    lay.addStretch(1)
    self.request_Widget.setLayout(lay)
    self.midlower.addWidget(self.request_Widget)
    # 绑定功能按钮触发函数
    self.yesbut.clicked.connect(lambda: self.requestaccepted(data[1]))
    self.ignorebut.clicked.connect(self.requestignored)
    # 美化界面设置
    self.yesbut.setStyleSheet('QPushButton{background-color:#475469;
color:#FFFFFF;'
                              'border-style:outset; border-radius:10px;
min-width:4em; padding:6px;}'
                              'QPushButton:pressed{color:grey; border-
style:inset;}')
    self.ignorebut.setStyleSheet('QPushButton{background-color:#C25A4A;
color:#FFFFFF;'
                              'border-style:outset; border-radius:10px;
min-width:4em; padding:6px;}'
                              'QPushButton:pressed{color:grey; border-
style:inset;}')
```

功能按钮触发函数

```
def requestaccepted(self, data):
    # 接受好友请求
    # 调用client接口函数直接发送信息给server
    self.c.resolve_friend(["resolve", data])

def requestignored(self):
    # 忽略好友请求
    # 删除弹窗
    self.delete_request()
```

删除函数

```
def delete_request(self):
    if self.midlower.count() != 1: # 检测当前弹窗是否存在
        self.midlower.removeItem(self.midlower.itemAt(1)) # 删除弹窗
        self.request_Widget.deleteLater()
```

4.3.2.5 信息更新

聊天室由于聊天内容信息、好友信息、群聊信息等更新需要根据不同的情况进行信息的刷新。

在设计中，每次的刷新都是根据后端传输的信息类型，判断需要刷新的信息。

需要进行刷新的内容以及它们对应的刷新函数为：

1. 好友列表（friend_list）：refresh_fl(self)

2. 群聊列表（public_chat_list）：refresh_pc(self)
3. 聊天记录（chat_history）：refresh_ch(self, type_of_chat, chat_id)
4. 群聊成员列表（pcf）：refresh_cm(self, chat_id)
5. 信息栏（message_box）：refresh_message(self)

下面以`refresh_ch(self, type_of_chat, chat_id)`为例简述流程。

```
def refresh_ch(self, type_of_chat, chat_id):
    # 清空原本的信息
    self.chat_history.clear()
    # 更新当前展示群聊记录
    self.current_display = [type_of_chat, chat_id]
    if type_of_chat == 1: # 群聊
        self.ch.setText(memory_rooms[chat_id].roomname + "(" +
str(memory_rooms[chat_id].room_id) + ")")
        # 从用户数据库中获取群聊聊天信息逐条加入
        for i in memory_rooms[chat_id].chat_history:
            self.chat_history.addItem(i.time+"\t"+i.username+":\t"+i.message)
    else:  # 私聊
        self.ch.setText(memory_friends.friends[chat_id])
        # 从用户数据库中获取私聊聊天信息逐条加入
        for i in memory_friends.chat_history[chat_id]:
            self.chat_history.addItem(i.time+"\t"+i.username+":\t"+i.message)
    # 如果聊天记录过多 保证显示在最后一行
    self.chat_history.setCurrentRow(self.chat_history.count()-1)
```

5.实验感想

本次的实验主要分三个模块完成：window、client、server。

每个模块都通过多线程实现了异步的操作，能够在处理操作的同时接受信息。在设计上client有三个线程，server针对每一个client都会创建三个子线程（一个主子线程和两个功能子线程），window也设计了三个线程。线程过多也导致了系统的结构比较复杂。

前端和后端之间、client和server之间信息的传递是借助了队列，采用了生产者消费者模型，确保了信息的传递不会出现丢失或者未处理的情况。

三个模块的信息处理都是采用词典来进行信息类型的匹配，易于理解也便于维护，在功能的添加、修改和删除上都比较方便。

实验中根据聊天室特点实现了message协议，实现对信息类型和参数的封装和解封装，应用在client和server的信息传递中。

这次的实验重点实现了一种协议、多线程管理操作、生产者消费者模型的实现等，提高了对应用层层面的理解。用python/sql/pyqt结合组成了前后端的系列操作，创建了一个完整的网络聊天系统。