WeTalk 简易聊天室

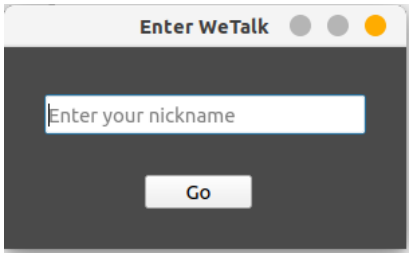
姓名：罗旭川　　学号：17307130162

一、项目基本聊天功能展示

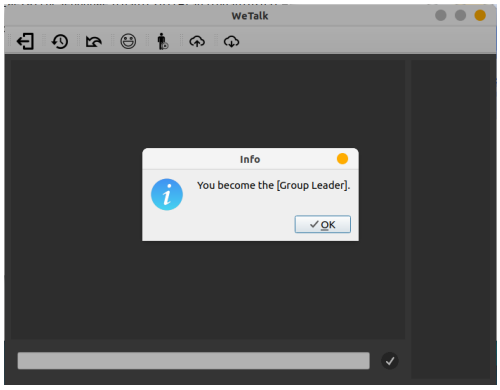
运行环境：Ubuntu 18.04、Python 3.6、PyQt5

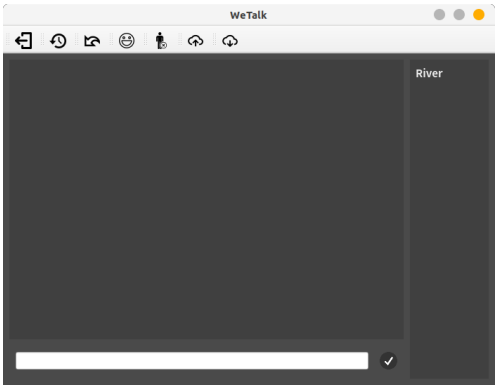
运行指令：

- 服务端启动：python server.py
- 客户端启动：python WeTalk.py



- 输入进入聊天室所用的昵称然后回车进入

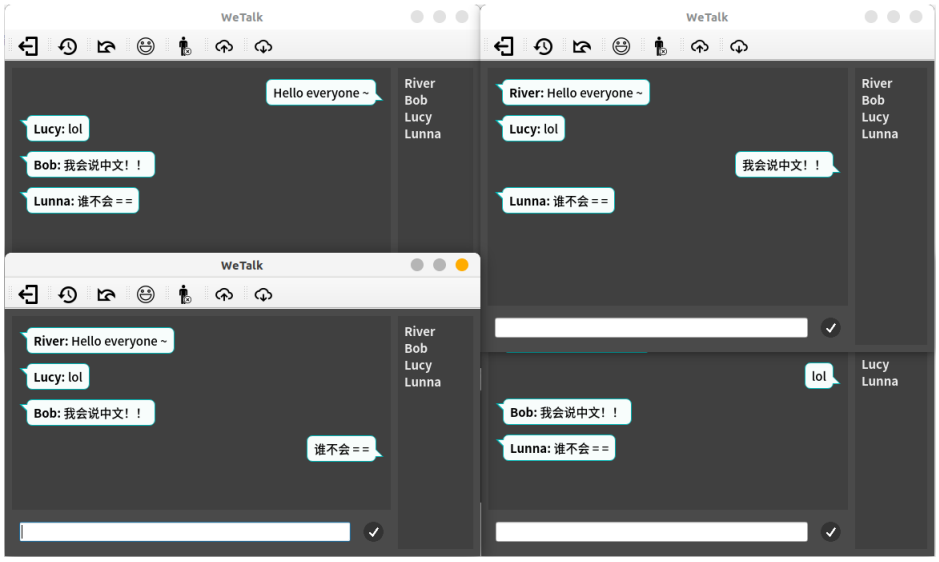




第一个进入聊天室的用户将会被提示成为群主(Group Leader)。

主界面主要分为四部分：上方工具栏，包含该聊天室的各种功能；中部消息框，用于显示用户发送的即时消息；右方用户栏，用于显示当前加入的所有用户的昵称；下方输入框，用于本用户输入并发送消息。

- 在输入框中输入信息开始愉快的聊天！



支持多人聊天。聊天消息会在消息框中渲染出气泡效果。支持中英文。

二、项目架构介绍

1. 服务端 1 + 2n 线程

- 主线程

  监听连接请求，每收到一个客户端连接则分配一个Server类对象服务之。每个Server类对象会启动两个线程，一个receiver线程和一个sender线程，分别负责信息的接收和信息的发送。

```
while True:
    conn, addr = s.accept()
    print(f'new connection: addr={addr}')
    _ = Server(conn, addr)
```

- receiver线程

  接收所连接的那个用户的所有请求。由于是端到端的连接，因此这里每个receiver只会接收到单个用户的请求信息，其他用户的请求信息会由它们专属的server对象的receiver线程接收和处理。

接收到消息后，首先根据消息的“头部”进行分类，具体的分类将在之后给出。大体分为两类：

- 群发请求

  receiver接收到群发信息后，对服务端本地的全局数据缓冲池进行更新。利用缓冲池的全局性来让其他用户的server能探测到该用户做出的改变。根据消息具体所属类型的不同，会更新服务端不同的数据缓冲池或做出其它操作。缓冲池类型将在介绍具体功能时分别给出。

- 单发请求

  receiver接收到单发信息后，对该server对象自己所维护的局部变量进行更新。对象内部变量的局部性让其它server探测不到该单发信息，从而实现该用户请求的隐蔽性。比如该用户下载文件的请求即为单发请求，sender只会将文件发给发出了请求的用户，而不是群发。

```
def update_buf(self, text):
  '''更新全局的数据缓冲池
  '''
  global msg_buf, msg_num, msg_lock, logIO_buf, logIO_num, logIO_lock
  ............
  mark = text[:mark_len]
  text = text[mark_len:]
  ............
```

receiver会将收到的字符串信息传入上述函数，该函数首先从字符串中截取头部mark，之后将通过mark的不同对不同的全局缓冲池或局部变量进行更新。

- sender线程

  - 群发信息

    需要群发给所有用户的信息，即为全局缓冲池中发生改变的信息。sender将这些改变发送给它所服务的客户端，这样一来n个sender线程的总体效果就是将全局缓冲池的变化“群发”给了n个不同的客户端。通过receiver改变全局变量，再通过sender将全局变量的改变群发出去，这就是该简易聊天室的基础原理。

```
    while True:
      try:
        with logIO_lock:    # 用户IO信息的群发，服务端并不需要知道有哪些用户，只需要转发用户进出的信息
          while self.__next_logIO < logIO_num:
            self.__send_UTF8(USER + logIO_buf[self.__next_logIO] + END)
            self.__next_logIO += 1

        with msg_lock:      # 群发消息的群发
          while self.__next_msg < msg_num:
            self.__send_UTF8(MSG + msg_buf[self.__next_msg] + END)
            self.__next_msg += 1
        .......
```

    我们以用户IO消息为例。logIO表示用户login和logout的信息。其中logIO_buf、logIO_num、logIO_lock为全局变量，分别表示logIO信息的存储缓冲池、logIO信息的当前接收数量、logIO_buf对应的互斥锁。self.next_logIO为该对象所维护的局部变量，表示该对象所负责的客户端已经接收的logIO数量。当self.next_logIO<logIO_num时，表示该客户端上的信息比服务端的旧，需要更新，于是就会发送需要发送的下一条新的logIO，直到客户端的信息数量和服务端一致。互斥锁的目的是防止sender线程和receiver线程对logIO_buf的访问冲突。

    以服务端的全局缓冲池作为标准，更新各个客户端的缓冲池，是本项目群发的具体实现方式。

- 单发信息

  单发消息较为简单，只需要通过检查局部变量是否发生了改变，若发生了改变，则发送给客户端即可。由于该局部变量只为该server所有，因此完全不会影响到其它server信息的发送。

2. 客户端 1 + 2 线程

- 主线程

GUI界面的响应线程，负责响应用户对程序界面的操作，并根据操作的具体内容进行相关函数的执行。本项目的GUI界面通过PyQt5 + html渲染实现。

- receiver线程

  与服务端receiver线程类似，该线程接收所有发送给该客户端的信息，并根据信息类型更改客户端本地的全局缓冲池或全局变量。这里使用全局变量是为了能让GUI模块探测到信息的变化，从而刷新界面。由于客户端不需要考虑对其它客户的影响，因此比较简单，全部改变通过维护本地全局变量来体现即可。

```
def update_buf(self, text):     # 转存数据到本地全局变量就好了
  '''更新客户端本地数据缓冲池
  '''
  global msg_buf, msg_num, msg_lock, logIO_buf, logIO_num, logIO_lock
  ............
  mark = text[:mark_len]
  text = text[mark_len:]
  ............
```

  receiver根据收到的信息头部mark来判断需要进行的更新操作，然后更新到全局变量中即可。

- refresher线程

  该线程专门用来探测客户端本地全局变量是否改变，若改变，则刷新相应的界面组件。实现方式是QTimer()，即定时探测。

```
  # 起refresher线程
  self.timer = QTimer()
  self.timer.timeout.connect(self.__refresh)
  self.timer.start(20)
```

  本项目设置为每20毫秒进行一次刷新，即调用self.__refresh()函数

  下面同样以logIO信息的刷新来简单说明一下刷新的实现方法：

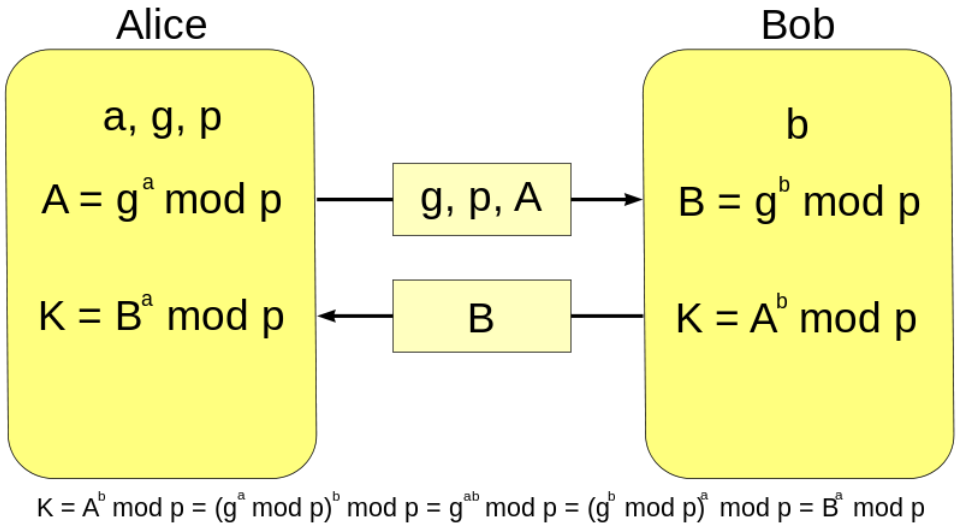
```
def __refresh(self):
  '''
  refresh线程（由Qtimer维护）
  '''
  global logIO_buf, logIO_num, logIO_lock
  ........
  try:
    with logIO_lock:
      while self.__next_logIO < logIO_num:
        opera = logIO_buf[self.__next_logIO]
        if opera[0] == '+':
          self.main_widget.userlist_widget.add(opera[1:])
```

```
            elif opera[0] == '-':
                self.main_widget.userlist_widget.delete(opera[1:])
            self.__next_logIO += 1
        ..........
```

和服务端运用的是类似的方法。GUI自己维护一个用来表示上一次刷新到的信息的位置self.__next_logIO，然后将其与全局变量logIO_num进行比较，由此判断是否需要刷新用户列表。若需要，则调用用户列表组件userlist_widget的相关函数进行用户的增减，从而实现界面的更新。
顺便一提，无论是客户端全局还是服务端全局实际上都只储存了用户进出的信息，真正的用户列表实际上只储存在了userlist_widget组件中。

3. 通过加密算法实现安全传输

- Diffie-Hellman 密钥交换



  - 客户端：

```
# 大素数p 原根g 秘钥a 公开秘钥A
p = 1945555039024054273
g = 5
a = None
A = None
# 共享秘钥K 加密时要用到的初始化向量IV
K = None
IV = None
```

客户端初始持有一个大素数及其原根。一般来讲，大素数要求达到100位以上才能保证安全性，这里仅做模拟，以一个18位的素数为例。5为该素数的一个原根。

```
        # 建立连接
        conn = socket.socket()
        conn.connect((self.host, self.port))
        # 开始交换秘钥
        a = random.randint(0, p - 1)
        A = pow(g, a, p)                # 计算公开秘钥A
        # 发送 g, p, A给服务端
        conn.sendall(bytes(KEY + str(g) + DIV + str(p) + DIV + str(A) + END, encoding='utf-8'))
```

在客户端刚刚与服务端建立连接之后、启动接收线程和界面之前，进行密钥交换流程。

首先客户端产生随机数$a(a < p)$，然后计算出客户端的公开秘钥$A = g^a \ mod \ p$，计算完成后将g、p、A发送给服务端。

- 服务端

```
# 公开秘钥B 共享秘钥K 初始化向量IV
self.__B = None
self.__K = None
self.__IV = None
```

服务端为每一个客户端都专门创建了相关的参数用于存储加密相关的秘钥和参数。

```
def __recv_key(self):
    recv_text = ''
    while True:
        recv_text += str(self.__conn.recv(4096), encoding='utf-8')
        index = recv_text.find(END)
        if index != -1:
            break
    start = recv_text.find(KEY)
    text = recv_text[start + mark_len: index]
    gpA = text.split(DIV, 2)
    g, p, A = int(gpA[0]), int(gpA[1]), int(gpA[2])
    b = random.randint(0, p - 1)
    self.__B = pow(g, b, p)         # 计算公开秘钥B
    self.__K = pow(A, b, p)         # 计算出共享秘钥K
    self.__IV = str(self.__K)[-16:]
    self.__K = str(self.__K)[:16]
```

收到客户端发送的g、p、A后，服务端计算其针对于该用户的公开秘钥$B = g^b \ mod \ p$和共享秘钥$K = A^b \ mod \ p$，计算完成后，服务端再将其计算出的公开秘钥B单发给客户端。

客户端收到后也能计算出共享秘钥$K = B^a \ mod \ p$。至此，客户端与服务端的密钥交换过程结束，它们拥有相同的共享秘钥K。之后的传输将用该共享秘钥K，采用AES算法进行加密。

由于AES算法要求向量IV和秘钥K都为16位串，因此这里分别截取后16位和前16位作为IV和K。

当且仅当客户端和服务端都得到了共享秘钥K后，客户端才开始初始化界面，然后发送Login消息。在此之前都会被阻塞。

- 使用AES加密传输内容

```
# 加密函数
def encrypt(text, K, IV):
    text = add_to_16(text)
    cryptos = AES.new(K.encode('utf-8'), AES.MODE_CBC, IV.encode('utf-8'))
    cipher_text = cryptos.encrypt(text)
    # 因为AES加密后的字符串不一定是ascii字符集的，输出保存可能存在问题，所以这里转为16进制字符串
    return b2a_hex(cipher_text)
```

加密采用的是AES中的CBC模式，其中还通过补零转16进制字符串的方法来防止传输出错。

```
# 解密后，去掉补足的空格用strip() 去掉
def decrypt(text, K, IV):
    cryptos = AES.new(K.encode('utf-8'), AES.MODE_CBC, IV.encode('utf-8'))
    plain_text = cryptos.decrypt(a2b_hex(text))
    return plain_text.rstrip('\0'.encode('utf-8'))
```

解密过程同理，要将多余的空格去除。这里解密我选择直接返回bytes序列，这是为了方面之后文件的传输。

- 效果

```
Recving Cipher: b'47e3a3d8ff9dd01aee43bdced01fc98640412073d0147bc781ea521e0e0794
64938a8dcc370efd1b936a78ea01ecede86c3db16bc1f7f63d659a399532dd4e340b38f3f0ef32c1
1950360ac7da92a8eb1674a40739afdcb5b39900f6af6c7c38'
River__DIV___这条信息（包括头部）加密的结果为上面的串
```

成功的实现了加密传输，之后的传输都会进行这样的加密。

4. 字符串标志位

通过设置必要的标志位来区分不同的信息类型。标志位又分为截断标志位和分类标志位。

- 截断标志位有2个：

```
END = '__END___'
DIV = '__DIV___'
```

END用来标志一条传输信息的结尾位置，用来预防出现截断发送或合并发送的情形，具体之后会讲。

DIV用来分隔一条传输信息中的两个不同部分，例如在一条群发消息中，用户名和消息内容就需要用DIV分隔开，因为它们必须储存在同一条信息中配对传输。

- 分类标志位有很多个：

```
MSG = '__MSG___'        # 消息
USER = '__USER__'       # 用户登录登出信息
LOGIN = '__LOGI__'      # 新登录信息
LOGOUT= '__LOGO__'      # 新登出信息
INIT = '__INIT__'       # 初始化信息
HIS = '__HIS___'        # 历史记录
ROLL = '__ROLL__'       # 撤回消息信息
NAME = '__NAME__'       # 重名信息
GRANT= '__GRANT_'       # 任免信息
KICK = '__KICK__'       # 踢人信息
UPLOAD= '__UPL___'      # 上传信息
FNAME ='__FNAME_'       # 新文件信息
DOWNLOAD='__DOWNL_'     # 下载信息
FILE = '__FILE__'       # 文件信息
KEY = '__KEY___'        # 公开秘钥信息
FACE = '__FACE__'       # 表情信息
mark_len = 8
```

分类标志位会加在发送信息的头部（即为前面提到的mark），用于区分该信息的类型。

例如一个叫River的用户发送一条Hello的消息，则该消息的传输形式如下：

```
__MSG___River__DIV___Hello__END___
```

其他标志位的具体含义将在下面介绍具体功能时相应给出。

注意到本项目中所有的标志位都规定为了8位，这样是为了方便字符串处理。

5. 三种不同的传输函数包装方式

```
def send_UTF8(text):                    # 负责发送utf-8可编码类的消息
  global conn, K, IV
  try:
    conn.sendall(MyCrypto.encrypt(text, K, IV)) # 加密发送
  except Exception as err:
    print(f'[send ERROR] {err}')

def sendFile(filename, filebytes):   # 负责发送文件
  global conn, K, IV
  try:
    conn.sendall(MyCrypto.encrypt(UPLOAD + filename + DIV + str(len(filebytes)) + END, K, IV)) # 先发文
件名和大小
    conn.sendall(filebytes)          # 然后发送文件bytes序列
    return True
  except Exception as err:
    print(f'[sendFile ERROR] {err}')
    return False

def recv_bytes():                       # 负责接收加密的所有消息
  global conn, K, IV
  try:
    text = conn.recv(4096)
    print(f'Recving Cipher: {text}')
    return MyCrypto.decrypt(text, K, IV)   # 接收后解密
  except Exception as err:
    print(f'[recv_file ERROR] {err}')
```

为了方便使用，本项目中将SOCKET中的函数自行进行了包装。发送函数将文件的发送单独的分离出来，因为文件的发送不能采用utf-8编码，且文件的发送序列较长。接收函数统一包装为接收bytes序列的形式，这样方便后续处理。加密和解密过程也包装在这三个函数中了。

文件的发送这里采用的是先发送一个包含了文件名和文件长度的短消息后，在单独进行文件的发送。短消息的格式为：

```
__UPL___fileName__DIV___fileLength__END___
```

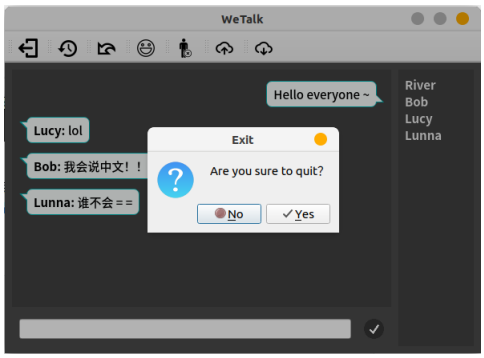
发送的长度将保证接收方完整地接收文件。

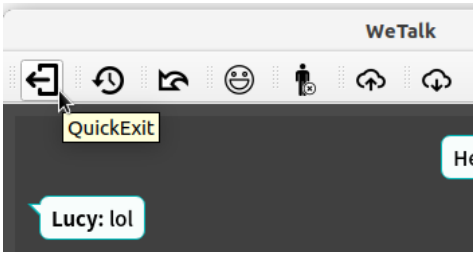
注意到这里还通过try-except捕获收发过程中可能出现的错误。

三、特殊功能介绍

1. 快速退出

- 效果展示





出于人性化考虑，当点击窗口右上角的关闭键退出群聊时，会弹出是否退出的询问框。当需要快速退出时，点击工具栏第一个QuickExit按键即可直接退出。后面将提到的群主踢人功能就是通过强制触发被踢用户的QuickExit功能实现的。

- 实现方法

```
app.exec_()
if nickname is not None:
    print('LOGOUT.')
    # 我走了
    send_UTF8(LOGOUT + nickname + END)
    conn.close()
```

点击按钮后，窗口关闭，应用退出，然后会执行上面语句。发送相应的LOGOUT字符串，然后关闭连接。

```
__LOGO__River__END___
```

服务端接收到上面字符串后，receiver线程提取出用户名River然后加入logIO_buf中，然后return，即结束了receiver线程。

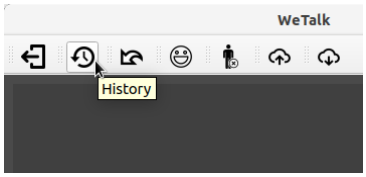
```
            if not self.__receiver.isAlive():
                if(len(threading.enumerate()) == 2): # 当所有用户都退出时，清空缓存池和缓存文件
                    self.__cleanBuf()
                self.__conn.close()
                return
```

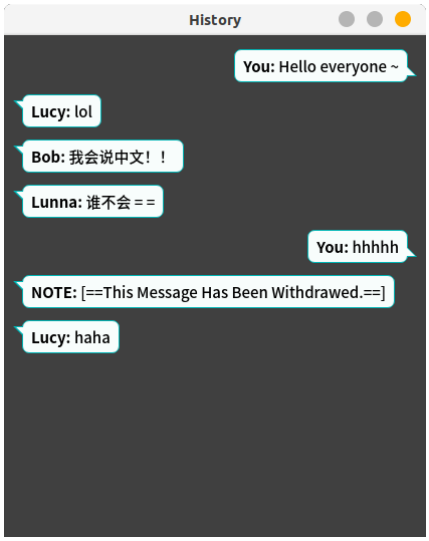
sender线程将logIO_buf中更新的内容群发出去，告诉其它用户River退出这一信息后，会执行上述代码。当发现receiver线程死掉了后，首先检查服务端一共还剩下多少个线程，若只剩下2个，说明自己是最后一个非主线程，则清空所有缓冲池，然后也结束自己，同时关闭conn连接。

通过上述交互，就实现了用户的干净退出。清除缓存池是因为，当自己是最后一个非主线程时，说明自己是最后一个退出的用户，该用户退出后聊天室将为空，即表示当次聊天结束。下一个进来的人不应该访问得到上一批人群聊的信息。这保障了聊天信息的隐蔽性。

2. 查看历史消息

- 效果展示





点击工具栏的History按键会弹出在加入聊天室之前，之前的人的聊了啥，即展示历史消息。历史消息中被撤回的消息会被替换。

- 实现方法

```
    # 请求历史记录
    send_UTF8(HIS + END)
    self.history_widget = HistoryWidget()
    self.history_ok = False
```

实际上历史记录的这个窗口是在用户登录后就创建的。在用户登录后，马上发送请求历史记录的信息，然后构建历史记录窗口。History按键只是简单地让历史记录窗口显示出来而已。

```
__HIS_____END___
```

这是一个单发请求。当服务端receiver接收到该请求后，将局部变量self.require_history的值置为True。然后sender会探测到该局部变量的变化，将历史记录单独发送给该客户：

```
        with msg_lock:
            if self.require_history is True:
                seq = str(msg_buf[:self.__next_msg])
                self.__send_UTF8(HIS + seq + END)
            self.require_history = False
```

msg_buf[:self.__next_msg]截取了该用户登录前的msg_buf直接转化为字符串传输。而从self.__next_msg开始的之后的信息，该用户则会通过服务端的群发接收到，即会显示在当前对话消息框中。这样的设计确保了信息不会遗漏。

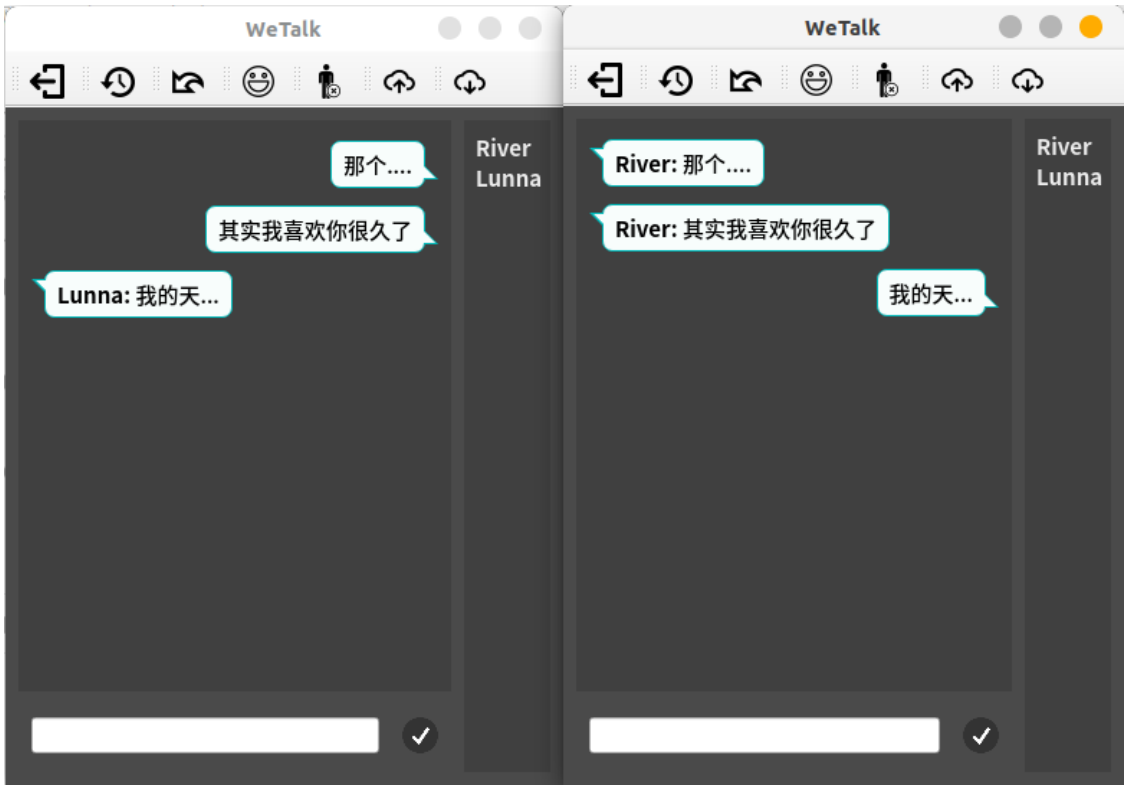
然后客户端接收服务端发来的历史记录：

```
    elif mark == HIS:
        global history_buf, history_lock
        if text == '[]':
            global no_history
            no_history = True
            return
        with history_lock:
            history_buf = eval(text)
        print(text)
```

判断出HIS头部后，简单的改变全局变量no_history和history_buf的值即可。之后refresh()线程就会根据更新渲染历史消息框的内容。这里维护了一个no_history全局变量用来处理没有历史记录的情形。此时会弹出告知无历史消息提示，而不是历史消息框。

3. 消息撤回

- 效果展示
  - 撤回前：



  - 撤回后：

点击工具栏第三个Rollback按键实现消息的撤回。如图所示，撤回将在别人的客户端和自己的客户端撤回该用户最后一条发送的消息。同时会将服务端msg_buf中的相应消息替换走，即从历史记录中也掩盖掉该条消息。

- 实现方法

```
def rollBack(self):
    '''发送撤回请求
    '''
    send_UTF8(ROLL + nickname + END)
```

Rollback按键直接触发上述函数，发送相关字符串形式的请求：

```
__ROLL__River__END___
```

服务端receiver接收到ROLL请求后，会更新roll_buf、roll_num，它们表示需要撤回消息的用户队列

```
    elif mark == ROLL:
        global roll_buf, roll_lock, roll_num
        with roll_lock:
            roll_buf.append(text)
            roll_num += 1
        with msg_lock:
            for i in range(len(msg_buf) - 1, -1, -1):
                if msg_buf[i].split(DIV, 1)[0] == text: # 覆盖该用户的最后一条信息，若删除的话会有下标麻烦
                    msg_buf[i] = 'NOTE' + DIV + '[==This Message Has Been Withdrawed.==]'
                    break
        self.__debugInfo('RollBack -> ' + text)
```

同时将服务端本地的msg_buf中撤回的消息替换掉。这里通过替换的方式而不是删除的方式是为了避免处理下标而带来的麻烦。

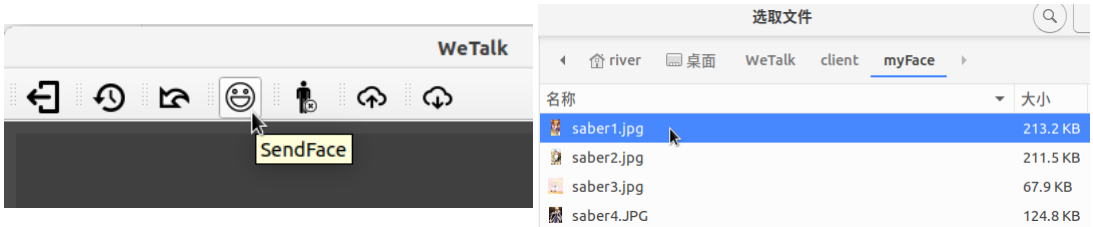
```
            with roll_lock:
                while self.__next_roll < roll_num:
                    self.__send_UTF8(ROLL + roll_buf[self.__next_roll] + END)
                    self.__next_roll += 1
```

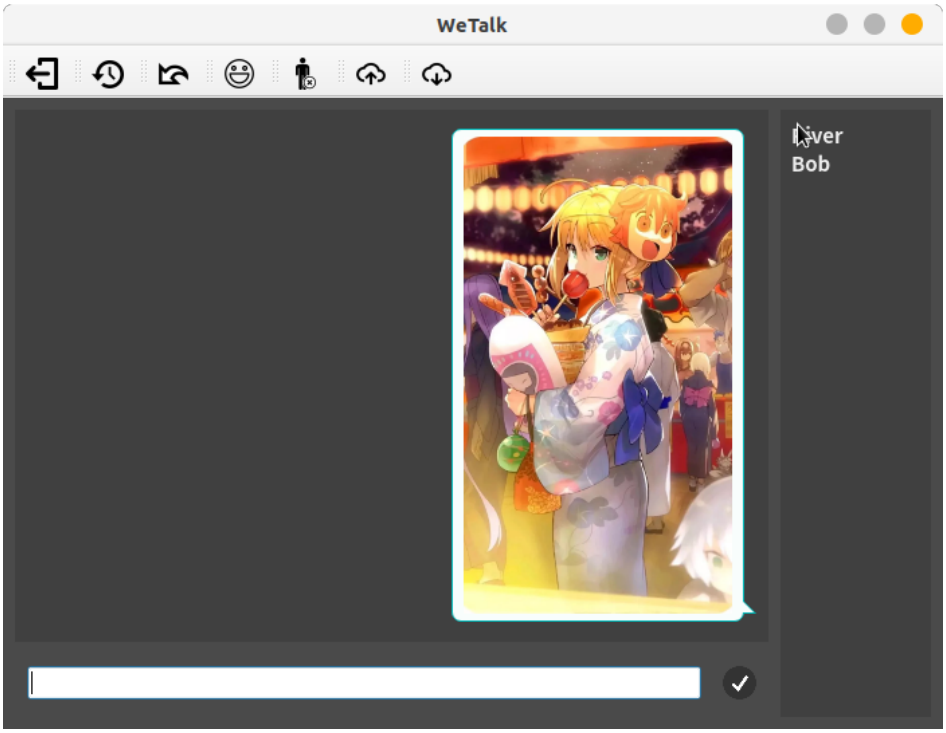
然后sender通过上述代码利用老办法将roll_buf中更新的内容群发到客户端，然后客户端将撤回请求接收到本地全局缓冲池roll_buf后，等待refresher更新即可。

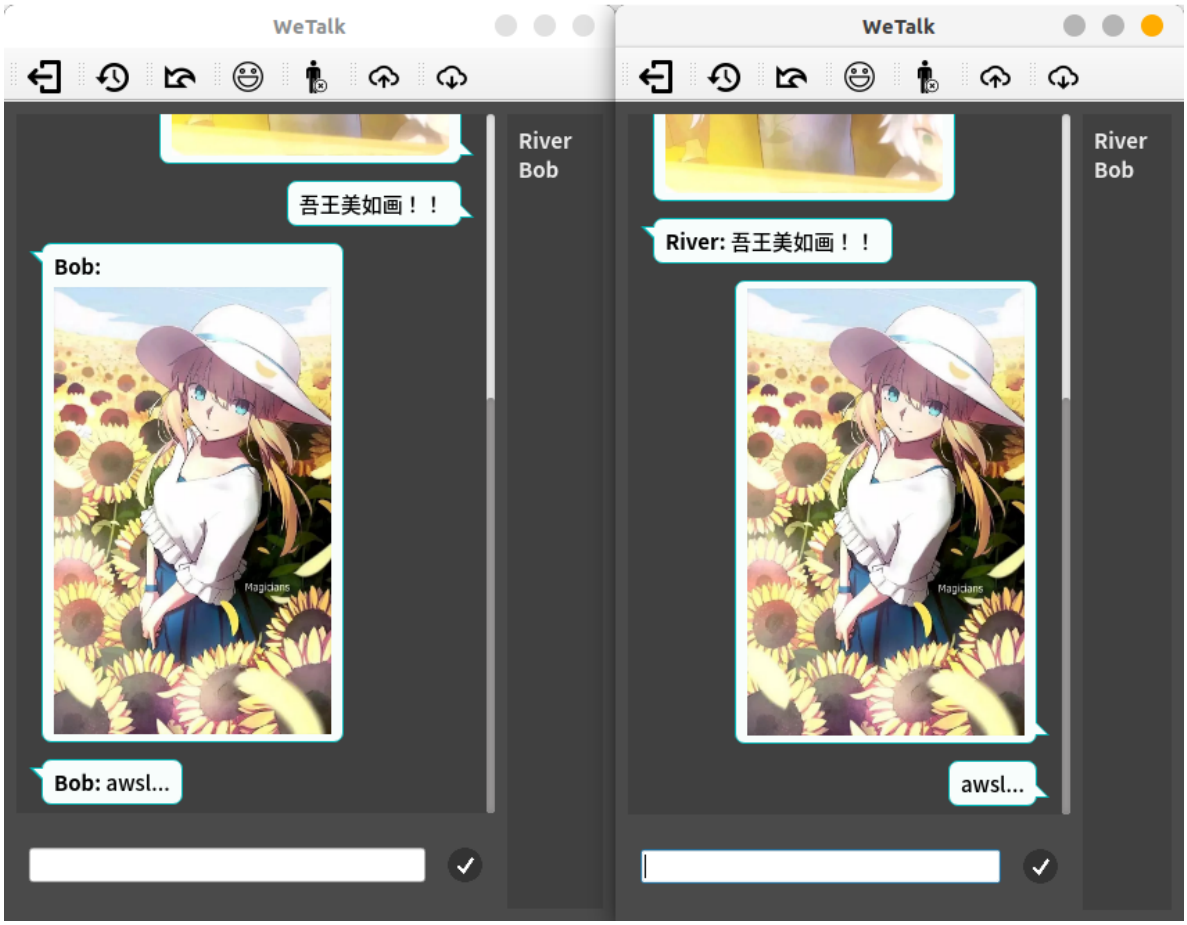
讲到这里，我们可以发现群发的操作其实几乎是一样的，无非是针对不同的消息头部、不同的全局缓冲池罢了。因此之后的群发操作若不是有特殊操作，我将不再赘述。

4. 表情/图片发送

- 效果展示







- 实现方法

  客户端从本地选择要上传的图片后，将图片序列发送至服务端，采用的适合发送文件一样的先发送一个包含用户名和图片大小的短信息，在发送图片序列，保证图片发送的完整性：

  ```
  __FACE__River__DIV___255350__END___
  ```

```
def sendFace(filebytes):
    global conn, K, IV, nickname
    try: # 先发用户名和图片大小
        conn.sendall(MyCrypto.encrypt(FACE + nickname + DIV + str(len(filebytes)) + END, K, IV))
        conn.sendall(filebytes)
        return True
    except Exception as err:
        print(f'[sendFace ERROR] {err}')
        return False
```

服务端接收到图片后，先保存到服务端本地的文件夹中，并且给图片赋予uuid生成的随机图片名，然后按照群发消息一样的套路，更新全局的face_buf、face_num。face_buf中存储的为待群发的图片，face_num表示服务端总共收到的图片数量。然后由sender检测并依次发送图片：

```
            with face_lock:
                while self.__next_face < face_num:
                    temp = face_buf[self.__next_face]
                    face_name = temp.split(DIV, 1)[1]
                    with open('./face_buf/' + face_name + '.jpg', 'rb') as f:
                        filebytes = f.read()

                    self.__send_UTF8(FACE + temp + DIV + str(len(filebytes)) + END)
                    self.__conn.sendall(filebytes)
                    self.__next_face += 1
```

图片发送到客户端也是一样的形式，即先发送短信息再发送图片。客户端收到群发而来的图片字节序列后，将其转为base64编码然后通过html渲染到界面上：

```
        while(bytes_received < bytes_total): # 按照长度接收图片
            data = conn.recv(bytes_total - bytes_received)
            bytes_received += len(data)
            facebytes += data
        s = base64.b64encode(facebytes).decode()
        with face_lock:
            face_buf.append(user_name + DIV + s)
            face_num += 1
```

转换成base64后，就直接将图片作为html代码用渲染消息一样的方式渲染到每一个客户端的界面上即可。

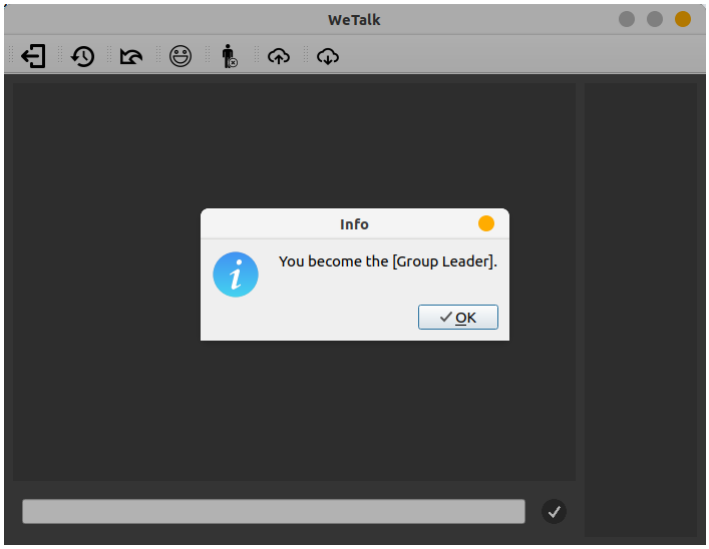
```
<img src=" .....base64序列....." width="200" alt="图像加载失败...">
```

上述这种直接渲染图片的方法无需将图片缓存到客户端，是一种比较常见、便捷的图片渲染方法。

5. 群主特权与轮转制

- 效果展示

    1. 第一个进入群聊的人会获得群主资格：

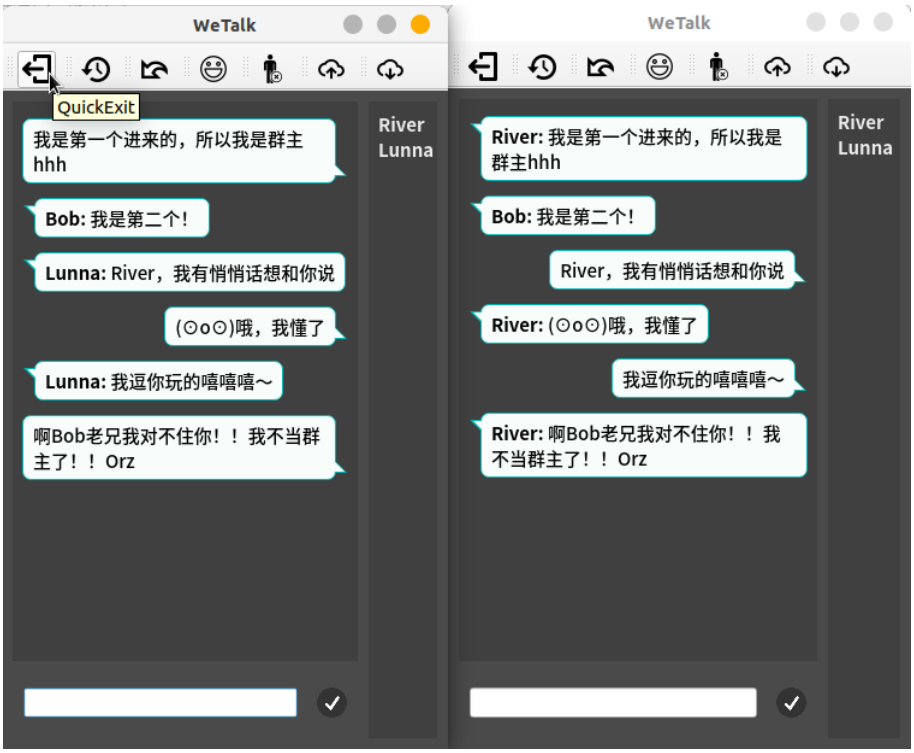

WeTalk
Info
You become the [Group Leader].
OK

2. 群主具有踢人权限：


WeTalk
我是第一个进来的，所以我是群主hhh
Bob: 我是第二个！
Lunna: River，我有悄悄话想和你说
(⊙o⊙)哦，我懂了
River
Bob
Lunna
WeTalk
River: 我是第一个进来的，所以我是群主hhh
我是第二个！
Lunna: River，我有悄悄话想和你说
River: (⊙o⊙)哦，我懂了
River
Bob
Lunna
WeTalk
River: 我是第一个进来的，所以我是群主hhh
Bob: 我是第二个！
River，我有悄悄话想和你说
River: (⊙o⊙)哦，我懂了
River
Bob
Lunna


WeTalk
我是第一个进来的，所以我是群主hhh
Bob: 我是第二个！
Lunna: River，我有悄悄话想和你说
Kick out someone
Enter his name:
Bob
Cancel
OK
River
Bob
Lunna
WeTalk
River: 我是第一个进来的，所以我是群主hhh
我是第二个！
Lunna: River，我有悄悄话想和你说
River: (⊙o⊙)哦，我懂了
River
Bob
Lunna
WeTalk
River: 我是第一个进来的，所以我是群主hhh
Bob: 我是第二个！
River，我有悄悄话想和你说
River: (⊙o⊙)哦，我懂了
River
Bob
Lunna

3. 群主退出后，将由下一个最早进入的人继承群主权限：



- 实现方法

```
# sender
        .......
        if groupLeader is None:
          with logIO_lock:
            groupLeader = self.__getFirstMember()

        if self.__nickname == groupLeader and self.become_leader is False:
          self.__send_UTF8(GRANT + END)
          self.become_leader = True
        .......
```

上述代码为服务端sender线程主循环中的一部分代码。该代码探测当前是否有群主，若没有，则通过logIO_buf找到第一个进入聊天室且还没有退出的人，将它任命为groupLeader，然后将任免信息单发给被任命的客户端：

```
__GRANT___END___
```

客户端的receiver线程收到任免信息后，将全局变量getLeaderPower设为True：

```
    elif mark == GRANT:
      global getLeaderPower
      getLeaderPower = True
```

这个为True的getLeaderPower将使能界面上的踢人功能。

群主在踢人窗口中输入要踢的用户的名字，然后发送踢人请求给客户端：

```
__KICK__Bob__END___
```

服务端收到KICK请求后，执行以下代码：

```
    elif mark == KICK: # 强制退出
      with logIO_lock:
        if not self.__isInUserList(text): # 若踢的人不在列表中，则什么事都不会发生
          return
        logIO_buf.append('-' + text)
        logIO_num += 1
      with kick_lock:
        kick_buf.append(text)
      self.__debugInfo('User- -> ' + text)
```

上述代码首先创建被踢人的Logout信息加入logio_buf中，用于群发出去，告诉所有用户这个人退出了；然后将被踢人的名字加入全局的kick_buf中。

```
          with kick_lock:
            if self.__nickname in kick_buf:
              kick_buf.remove(self.__nickname)
              self.__send_UTF8(KICK + END)
              return
```

服务端的sender然后不断扫描kick_buf，当发现它所服务的客户是被踢的对象之一时，则发送KICK给客户端用户强制关闭客户端；同时自己使用return来结束线程。当sender结束后，receiver也会随即探测到并结束，然后关闭连接，从而干净退出。群主其实还可以自己踢自己。

注意到这里的KICK信号虽然为单发信号，但还是维护了一个全局的kick_buf来实现。实际上通过局部变量也是可以实现的。这里只是两种方法二选一而已。

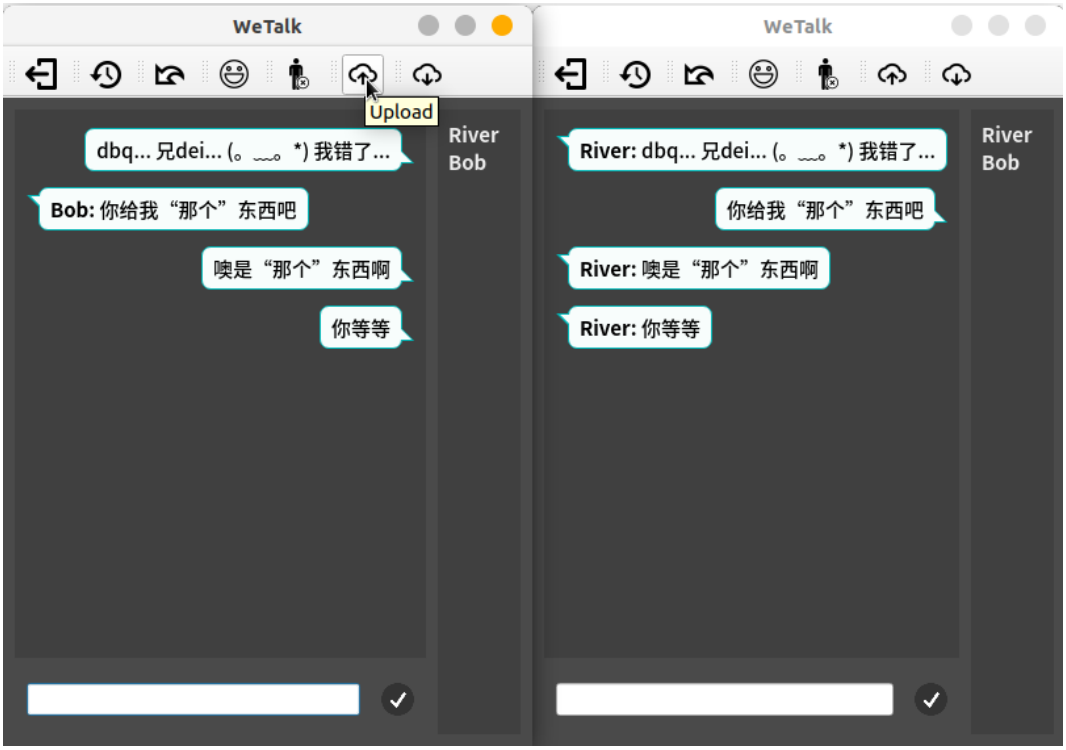
```
__KICK____END___
```

客户端收到上述的被踢指令，按照老办法，通过全局变量让refresher探测到，refresher探测到后，通过qApp.quit()强制退出。

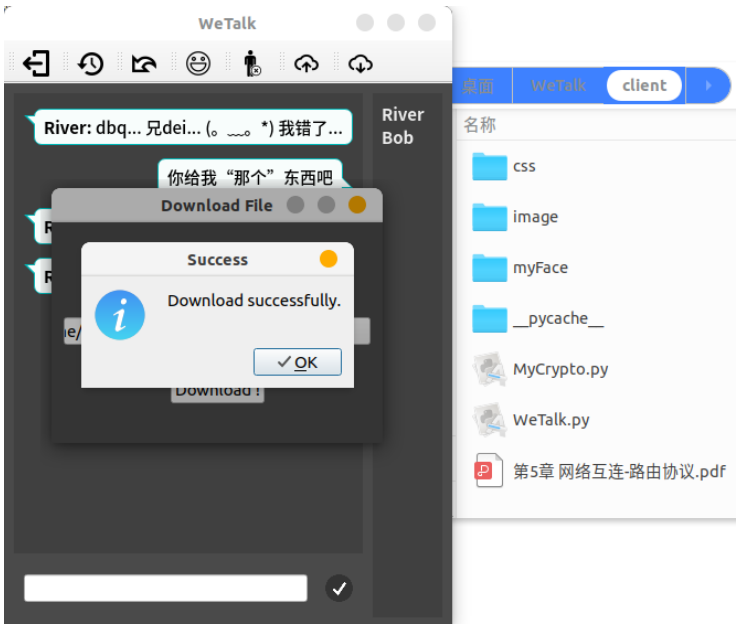
```
        if isKicked is True:
            QMessageBox.critical(self, 'Announce', f'You have been kicked out by the Group Leader.')
            qApp.quit()
```

6. 文件传输

- 效果展示
  - 上传：



- 上传后其他人可以找到并下载那个文件：



- 实现方法

```
def upLoad(self):
    filename, _ = QFileDialog.getOpenFileName(self,
                        "选取文件",
                        self.__cwd, # 起始路径
                        "All Files (*);;Text Files (*.txt)")   # 设置文件扩展名过滤,用双分号间隔
    if filename == "":
        print("取消选择")
        return
    with open(filename, 'rb') as f:                    # 将文件打开，读取字节流
        filebytes = f.read()
    if sendFile(os.path.basename(filename), filebytes):  # 忽略等待
        QMessageBox.information(self, 'Success', f'Upload successfully.')
    else:
        QMessageBox.critical(self, 'error', f'Upload fail.')
```

用户选择完要上传的文件后，客户端即能获取到要上传的文件路径。从文件路径中获取文件名filename，然后以'rb'模式打开文件获取文件字节流filebytes。

将文件名和文件长度先发送到服务端，服务端接收后就进入接收文件状态。

```
__UPL___filename__DIV___bytes_total__END___
```

```
        filebytes = b''
        bytes_received = 0
        while(bytes_received < bytes_total): # 按照长度接收文件
            data = self.__conn.recv(bytes_total - bytes_received)
            bytes_received += len(data)
            filebytes += data
```

接收文件时采用上述结构，逐段接收直至接收总长达到文件长度，从而保证接收的完整性。前面的图片发送具体实现也是如此。

服务端接收到上述文件后，根据文件名将文件保存在服务端本地的files_buf文件夹中，然后在全局的filename_buf中增添新上传的文件名。

```
    with open('./files_buf/' + filename, 'wb') as f:
        f.write(filebytes)
    with file_lock:
        filename_buf.append(filename)
        filename_num += 1
```

然后服务端sender根据全局的filename_buf、filename_num，利用相同的套路，将新上传的文件信息群发给所有用户。用户接收到新文件信息后也是用相同的套路储存在客户端本地全局缓冲池filename_buf中，之后refresher探测到新增的文件名，将其显示在下载界面上的可下拉选择器中供用户选择。

至此，服务端已经存储了上传的文件，所有客户端也获得了最新的已上传文件名列表。所以客户端就可以根据文件名列表选择想要的文件发送下载请求了：

```
__DOWNL_filenameChosen__END___
```

服务端receiver线程接收到上述请求后，首先分离出文件名，将其加入到局部列表变量self.require_files中。然后当sender线程探测到该局部变量列表不为空时，会逐个根据列表中的文件名将对应的文件从files_buf文件夹中打开，读取其字节流，然后以上传时同样的形式（先发长度后发文件）单发给请求的客户端。

客户端接收到上述文件后，结合之前选择的下载路径download_dir，将文件保存到本地即可。

四、细节处理

1. 保证接收消息的完整性和独立性

在进行代码调试的过程中，发现了socket API在接收过程中会出现以下两种情形：分段传输接收和合并接收

- 针对分段接收

  分段传输主要出现在传输很长的字符序列的情形，比如传输文件的情形。

  在接收函数conn.recv(n)中，n表示每次接收的最大字节数量。由于我们不可能将n设得无限大，因此当传输大文件时总会出现一次接收不完的情形。这是就需要通过循环接收的方式，将每次接收的片段累加起来，直到收到序列的结束标志END

- 针对合并接收

  当两个短的传输序列在很接近的时刻到达时，conn.recv(n)会将它们合并接收为一个字符串。

  这种情况将导致合并信息中的后一条会被当做前一条的内容，从而产生信息错误和信息遗失。针对这个问题，我们在每次接收到字符序列时，需要遍历找出所有的结束标志END，从而从每一次的接收序列中分离处所有的独立传输信息。

- 代码实现

```
def recv_proc(self):
    '''【接收工作线程】 接收服务端群发的更新数据，以此更新本地数据
    '''
    # 首先进行密钥交换
```

```
        self.__recv_key()

        END_bytes = bytes(END, encoding='utf-8')
        recvBytes = b''
        while True:
            recvBytes += recv_bytes()
            try:
                index = recvBytes.find(END_bytes)
                if index == -1:             # 针对不完整情况
                    continue
                while index != -1:          # 针对合并情况
                    self.update_buf(recvBytes[:index])
                    recvBytes = recvBytes[index + mark_len:]
                    index = recvBytes.find(END_bytes)
                recvBytes = b''
            except Exception as err:
                print(f'[update_buf ERROR] {err}')
```

上面为客户端receiver线程的函数实现，服务端的receiver类似。

当一次接收的序列recvBytes找不到结束标志END时，我们则通过continue等待下一次的接收序列，然后进行累加，直到找到结束标志为止。至此收到的序列至少包含一个独立的传输信息。

将上述序列通过循环不断搜索END标志然后切割分段，分出来的每一个段都是一个独立的传输信息。之后将每条独立信息输入self.update_buf()函数中进行相应的全局变量更新即可。

2. 用户允许中途加入

由于用户加入群聊有先后之分，之前加入的人可能会先进行聊天对话，而后加入的人就看不到前面的人的对话，于是就引入了之前提到的历史记录功能，供后加入的用户浏览前面用户的聊天内容；同样，后加入的人需要获取当前聊天室都已经有哪些人存在以及之前都传输了哪些文件。因此，在一个用户login时，服务端会对该用户单发之前的历史记录、当前的用户列表以及上传文件名列表。这样一来用户登录时就能接收到这些信息，然后更新历史记录框、用户列表和文件列表。

```
__INIT__userlist__DIV___filelist__END___
```

这条信息传输的是当前用户列表和文件名列表

```
__HIS___msglist__END___
```

这条信息传输的是历史记录

理论上这三个列表是可以一起传输的，但考虑到历史记录可能较多，因此将其分开来传输处理了。

3. 重名处理

处理重名问题理论上有两种办法：一是在用户登录时输入用户名时，就进行检测；二是在用户登录后已经进入聊天室后，对重名进行强制更改，然后告知用户。

考虑到检测用户名必须在连接建立之后才能实现，而在用户名输入界面就进行连接是不划算的（因为可能会反复登录退出），因此本项目中用的是第二种方法。

```
    if name in user_list:
        self.__nickname = self.name_change = name + '#'
        return name + '#'  # 若有重复用户名，则当前要增加的用户名变为 ...#
    self.__nickname = name # 记录本连接的用户名
```

服务端是按照上述代码来处理重名的，即简单的在新加入的重复用户名后加'#'。例如第一个进入的用户名为River，则之后以River命名的用户将会被重新赋予River#、River##....的新昵称。赋予新名称后，服务端会单发提示信息给相应的客户：

```
__NAME__newname__END___
```

客户端收到上述提示后就会弹出提示框告知用户。考虑到这是一个简易的灵活式聊天室，这样的设计是合理的。

4. 群聊结束后清空缓存

当所有人都退出聊天室后，说明本次聊天已经全部结束，此时将服务端的所有消息记录、用户记录、文件缓存、表情包缓存全部清空是更加合理的，因为下一个连接的用户应该被认为是发起了一个全新的群聊。

前面讲快速退出功能时已经提到过，服务端是通过获取剩余的线程数来判断本次群聊是否已经结束。当判断出群聊结束后，就开始执行清空缓存的函数：

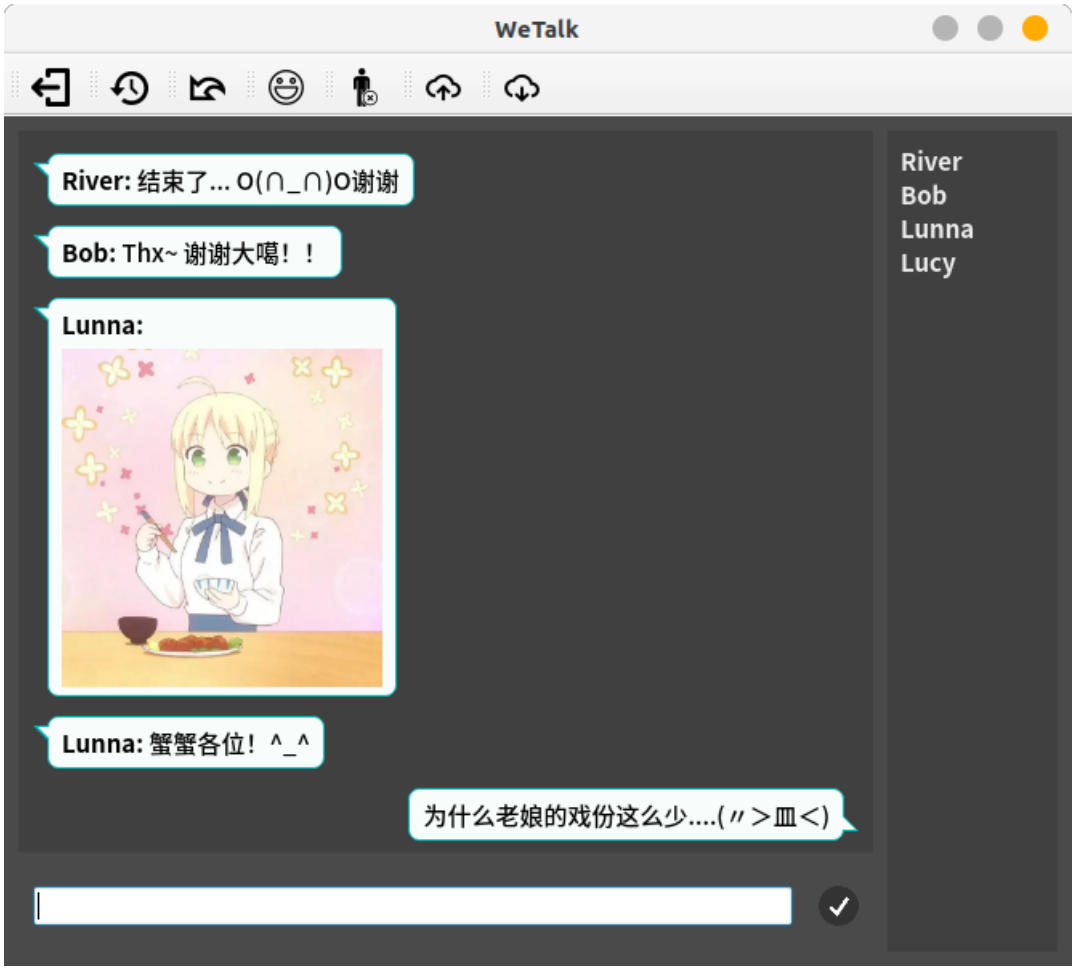
```
def __cleanBuf(self):
    '''清空缓存池
    '''
    global msg_buf, logIO_buf, roll_buf, kick_buf, filename_buf
    global groupLeader, msg_num, logIO_num, roll_num, filename_num
    shutil.rmtree('./files_buf')
    os.mkdir('./files_buf')
    shutil.rmtree('./face_buf')
    os.mkdir('./face_buf')
    msg_buf = []
    logIO_buf = []
    roll_buf = []
    kick_buf = []
    filename_buf = []
    groupLeader = None
    msg_num = 0
    logIO_num = 0
    roll_num = 0
    filename_num = 0
    print('NOTE: Clean ALL bufs done.')
```

清空操作非常简单，即清空所有的全局缓存变量以及服务端本地的文件缓存文件夹、表情包缓存文件夹即可。

5. 利用html进行消息框渲染

这可以说是我自认为的本项目中最得意的亮点了。单纯用PyQt5是难以实现消息框中聊天气泡效果的，因此需要借助静态html和css渲染来实现。而静态的html如何实现动态的聊天消息变化呢？在本项目中，这是通过在每个客户端中维护一个全局的html长代码串，当收到新消息或需要撤回旧消息时，就相应地更改这个html代码串，然后将更新完后的html重新加载到界面上即可。

五、项目总结



这是一个简易但又包含众多实用功能的聊天室，是通过严密的传输规定和接收逻辑来实现的，轻量而高效。由于代码的逻辑性较强，群发传输和单发传输存在着一定的固定方法，因此代码的维护和功能的增加都较为方便。若有时间，我将会继续为其加入其它实用的功能，如私聊等。