目 录

[第一章 引言 3](#_Toc555003449)

[第二章 需求分析 3](#_Toc1421488422)

[2.1 客户端需求分析 3](#_Toc200335679)

[2.1.1 用户登录与退出 3](#_Toc1934882104)

[2.1.2 消息发送与接收 4](#_Toc218655407)

[2.1.3 私人聊天 4](#_Toc596905432)

[2.1.4 异常处理 5](#_Toc1293480487)

[2.2 服务器端需求分析 5](#_Toc549586428)

[2.2.1 客户端管理 5](#_Toc571929649)

[2.2.2 消息处理与广播 5](#_Toc284806771)

[2.2.3 私人聊天处理 5](#_Toc6351034)

[2.2.4 异常处理 6](#_Toc1515129735)

[第三章 系统设计 6](#_Toc2071853666)

[3.1 客户端设计 6](#_Toc197228357)

[3.1.1 用户界面 6](#_Toc1249728778)

[3.1.2 网络通信 7](#_Toc1801504186)

[3.1.3 多线程处理 7](#_Toc508915049)

[3.2 服务器端设计 8](#_Toc2055346189)

[3.2.1 客户端管理 8](#_Toc1928936528)

[3.2.2 消息处理与广播 8](#_Toc1223090984)

[3.2.3 异常处理 9](#_Toc776699004)

[3.2.4 并发处理 9](#_Toc1574553762)

[3.3 通信协议设计 10](#_Toc84095953)

[3.4 安全性设计 10](#_Toc356442345)

[第四章 系统实现 10](#_Toc1394600932)

[4.1 客户端实现 11](#_Toc1421340766)

[4.1.1 用户界面实现 11](#_Toc2013648581)

[4.1.2 网络通信实现 13](#_Toc1196907794)

[4.1.3 多线程处理实现 14](#_Toc949972309)

[4.2 服务器端实现 15](#_Toc1791165565)

[4.2.1 客户端管理实现 15](#_Toc693887309)

[4.2.2 消息处理和广播实现 16](#_Toc1327799153)

[4.2.3 并发处理实现 17](#_Toc1817788494)

[第五章 系统测试和验证 18](#_Toc1468856436)

[5.1 单元测试 18](#_Toc1745597587)

[5.1.1 客户端模块测试 19](#_Toc1484543042)

[5.1.2 服务器端模块测试 19](#_Toc1249896048)

[5.2 集成测试 20](#_Toc317843782)

[5.3 系统测试 21](#_Toc1208613985)

[5.4 性能测试 21](#_Toc127428922)

[5.4.1 响应时间测试 22](#_Toc656695995)

[5.4.2 吞吐量测试 22](#_Toc1171126032)

[5.4.3 并发用户数测试 22](#_Toc1427595069)

[5.5 用户验收测试 22](#_Toc1903020399)

[5.6 小结 23](#_Toc1589891222)

[第六章 总结 23](#_Toc162748533)

[参考文献 24](#_Toc1567911500)

# 第一章 引言

本报告将对一个基于Python实现的聊天室应用进行深入研究。这个应用涉及到了Python的多线程，网络编程以及图形用户界面（GUI）设计等技术。聊天室应用包括服务端和客户端两部分，分别对应两份Python代码文件：服务器.py和客户端修改.py。聊天室应用允许多个用户进行公共和私人的即时聊天，并提供了一些额外的功能如敏感词过滤和在线用户列表显示。

# 第二章 需求分析

在计算机网络应用的开发中，需求分析是一个至关重要的步骤。正确理解和描述系统的功能需求，能够有效指导软件设计和实现，减少开发错误，并提高软件质量。本章将详细分析聊天室应用的需求，包括客户端和服务器端的需求。

## 2.1 客户端需求分析

对于聊天室应用的客户端，其功能需求主要包括以下几个方面：

### 2.1.1 用户登录与退出

用户登录是聊天应用中最基本的功能。用户首次连接到聊天室时，需要提供一个昵称作为标识。这个昵称将会在聊天过程中显示，以区分不同的用户。用户登录的实现需要通过网络向服务器发送用户昵称，并等待服务器的回应，确认登录成功。

用户退出功能也是必须的。用户在退出聊天室时，客户端应该向服务器发送退出消息，服务器收到消息后，将该用户从在线用户列表中移除，并向其他在线用户广播退出消息。

### 2.1.2 消息发送与接收

消息的发送和接收是聊天应用的主要功能。用户可以通过客户端发送消息，这些消息将会被发送到服务器，并由服务器广播给所有在线用户。用户也可以接收来自其他用户的消息，这些消息首先被服务器接收，然后通过网络发送到客户端，并在客户端的聊天窗口中显示。

### 2.1.3 私人聊天

除了在公共聊天室中聊天外，用户还可能需要与特定的用户进行私人聊天。这就需要客户端提供创建和管理私人聊天窗口的功能。在私人聊天窗口中，用户发送的消息只会被发送到指定的用户，而不是广播给所有在线用户。同样，用户也只能接收到来自指定用户的消息。

### 2.1.4 异常处理

在网络应用中，异常情况是常见的，例如网络连接断开、服务器关闭等。客户端需要能够正确处理这些异常情况，以提高用户体验。例如，在网络连接断开时，客户端应该通知用户，并尝试重新连接。

## 2.2 服务器端需求分析

对于聊天室应用的服务器端，其功能需求主要包括以下几个方面：

### 2.2.1 客户端管理

服务器需要能够处理来自客户端的连接请求，创建与客户端的网络连接，以及在客户端退出时关闭连接。此外，服务器还需要维护一个在线用户列表，记录当前所有在线的用户。

### 2.2.2 消息处理与广播

服务器需要能够接收来自客户端的消息，并将这些消息广播给所有在线用户。在这个过程中，服务器可能需要对消息进行一些处理，例如敏感词过滤。

### 2.2.3 私人聊天处理

服务器也需要处理来自客户端的私人聊天请求，将私人消息发送给指定的用户。在这个过程中，服务器需要查找指定用户的网络连接，并将消息通过这个连接发送出去。

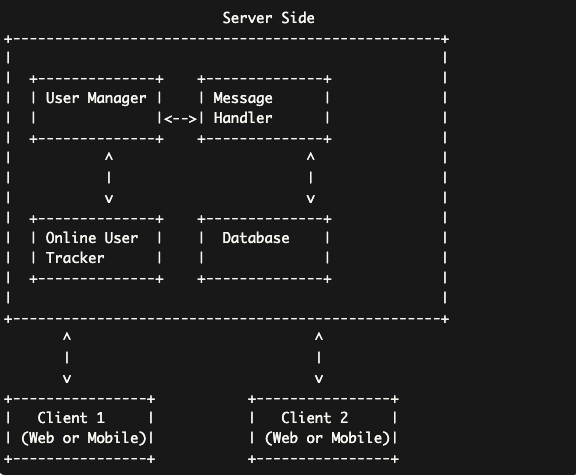
### 2.2.4 异常处理

服务器也需要处理可能出现的各种异常情况，例如客户端突然断开连接。在这些情况下，服务器需要将异常的客户端从在线用户列表中移除，并可能需要向其他在线用户广播一条消息，通知他们有用户退出了聊天室。

通过以上的需求分析，我们可以得到一个清晰的聊天室应用的功能需求描述。在后续的设计和实现过程中，我们将根据这些需求进行。

# 第三章 系统设计

在需求分析的基础上，我们可以开始设计聊天室应用的系统架构。系统设计的目标是确定系统的主要组件，定义它们的职责和交互方式。在这一章中，我们将详细描述聊天室应用的客户端和服务器端的设计。



系统架构图

## 3.1 客户端设计

客户端的设计目标是提供一个用户友好的界面，实现用户登录、消息发送和接收、私聊等功能。下面是客户端的主要设计元素：

"Client 1"和"Client 2"代表用户通过浏览器或者应用程序接入聊天系统的个体。

"Server Side"代表托管你的聊天应用的服务器，其中包括"User Manager"、"Message Handler"、"Online User Tracker"和"Database"四个组件，分别处理用户管理、消息处理、在线用户追踪和数据库存储等功能。箭头代表数据的流动方向。

### 3.1.1 用户界面

用户界面是客户端的核心组件，它直接影响到用户的使用体验。在本应用中，我们使用Python的tkinter库来设计用户界面，这个库提供了丰富的界面元素和布局管理器，可以满足我们的需求。

用户界面主要包括登录窗口，公共聊天窗口和私人聊天窗口。登录窗口用于用户输入昵称并连接到聊天室。公共聊天窗口显示所有公共消息，用户可以在这个窗口中发送消息。私人聊天窗口用于私人聊天，用户可以在这个窗口中发送和接收私人消息。

### 3.1.2 网络通信

客户端通过网络与服务器进行通信，这是实现聊天功能的基础。在本应用中，我们使用Python的socket库来实现网络通信。socket库提供了创建和管理网络连接的基本功能。

客户端的网络通信主要包括连接服务器，发送和接收消息。连接服务器是客户端的第一步，它需要知道服务器的IP地址和端口号。发送消息包括发送公共消息和私人消息，这些消息都需要通过网络发送到服务器。接收消息是客户端的一个重要功能，它需要在后台持续监听服务器的消息。

### 3.1.3 多线程处理

由于网络通信的异步性，客户端需要使用多线程来处理并发任务。在本应用中，我们使用Python的threading库来创建和管理线程。

客户端主要有两个线程，一个是用户界面线程，这个线程负责处理用户的输入和更新界面；另一个是网络通信线程，这个线程负责监听服务器的消息并处理。

## 3.2 服务器端设计

服务器端的设计目标是处理客户端的连接请求，接收和广播消息，处理私人聊天。下面是服务器端的主要设计元素：

### 3.2.1 客户端管理

服务器需要管理所有的客户端连接，这是服务器的基本职责。在本应用中，我们使用Python的socket库来创建服务器，接收客户端的连接请求。

服务器主要有两个列表，一个是客户端列表，另一个是昵称列表。客户端列表记录了所有的客户端连接，昵称列表记录了所有在线用户的昵称。当新的客户端连接到服务器时，服务器会将该客户端的连接和昵称添加到这两个列表中；当客户端退出时，服务器会将该客户端的连接和昵称从这两个列表中删除。

### 3.2.2 消息处理与广播

服务器需要接收客户端的消息，进行处理，然后将消息广播给所有在线用户。在本应用中，我们使用Python的socket库来接收和发送消息。

消息处理主要包括接收消息，解析消息，广播消息。接收消息是服务器的一个基本功能，它需要监听所有客户端的消息。解析消息是服务器的一个重要功能，它需要解析消息的内容，以确定是公共消息还是私人消息，以及消息的发送者和接收者。广播消息是服务器的一个主要功能，它需要将公共消息发送给所有在线用户，将私人消息发送给指定的用户。

### 3.2.3 异常处理

异常处理能够确保服务器在遇到问题时能够正常运行。服务器可能会遇到的问题包括：客户端突然断开连接、网络问题等。在这些情况下，服务器需要将异常的客户端从客户端列表和昵称列表中移除，并可能需要向其他在线用户广播一条消息，通知他们有用户退出了聊天室。

### 3.2.4 并发处理

服务器需要处理多个客户端的连接和消息，因此需要并发处理。在本应用中，我们使用Python的threading库来创建和管理线程。每个客户端连接都会创建一个新的线程，这个线程负责处理该客户端的所有请求，包括接收消息和发送消息。

## 3.3 通信协议设计

在客户端和服务器之间的通信中，我们需要定义一套通信协议。这套协议定义了消息的格式和含义，使得客户端和服务器能够彼此理解。

在本应用中，我们设计了一套简单的文本协议。每条消息都是一个字符串，由多个字段组成，字段之间用特定的分隔符（例如“|”）隔开。例如，“login|Alice”表示用户Alice登录，“msg|Alice|Hello, world!”表示用户Alice发送了一条公共消息，内容为“Hello, world!”。

## 3.4 安全性设计

考虑到网络应用的安全性问题，我们在设计中也需要考虑一些安全性措施。例如，我们可以对传输的消息进行加密，以防止消息被窃听。我们还可以对用户昵称进行检查，以防止恶意用户使用不合适的昵称。

至此，我们完成了聊天室应用的系统设计。在接下来的开发过程中，我们将根据这个设计进行。

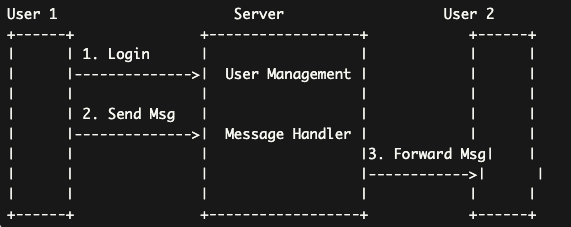
# 第四章 系统实现

在完成系统设计后，我们进入了系统实现阶段。在本章中，我们将详细介绍聊天室应用的实现过程，包括客户端和服务器端的实现。

"User 1" 和 "User 2" 分别代表两个使用聊天室的用户。

"Server" 代表托管聊天应用的服务器，其中包括 "User Authentication" 和 "Message Handler" 两个组件，分别处理用户登录验证和消息处理。

箭头("-->") 代表数据的流动方向，数字和文本描述表示了数据流的顺序和内容。



数据流图

## 4.1 客户端实现

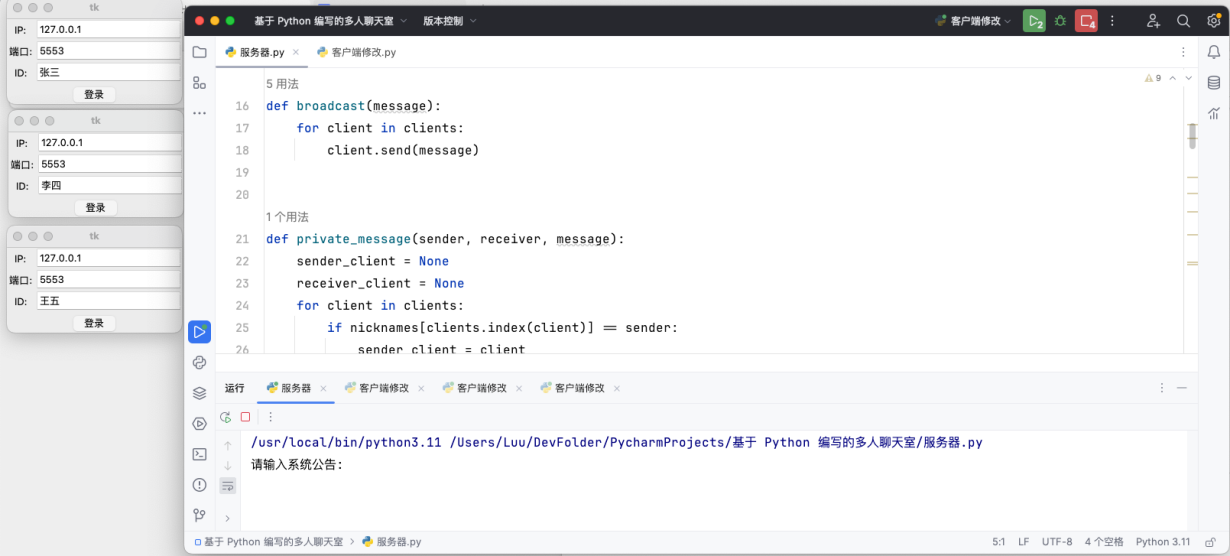
客户端的实现主要包括用户界面的实现，网络通信的实现，以及多线程处理的实现。

### 4.1.1 用户界面实现

Python的tkinter库为我们提供了丰富的界面元素，例如窗口，按钮，文本框等。我们可以使用这些元素来构建用户界面。

#### 登录窗口

登录窗口主要包括一个输入昵称的文本框和一个登录按钮。当用户输入昵称并点击登录按钮时，程序会创建一个新的客户端对象，并尝试连接到服务器。



登录窗口

from tkinter import \*

class LoginWindow(Tk):

def \_\_init\_\_(self):

Tk.\_\_init\_\_(self)

self.title("Login")

self.nickname\_entry = Entry(self)

self.nickname\_entry.pack()

self.login\_button = Button(self, text="Login", command=self.login)

self.login\_button.pack()

def login(self):

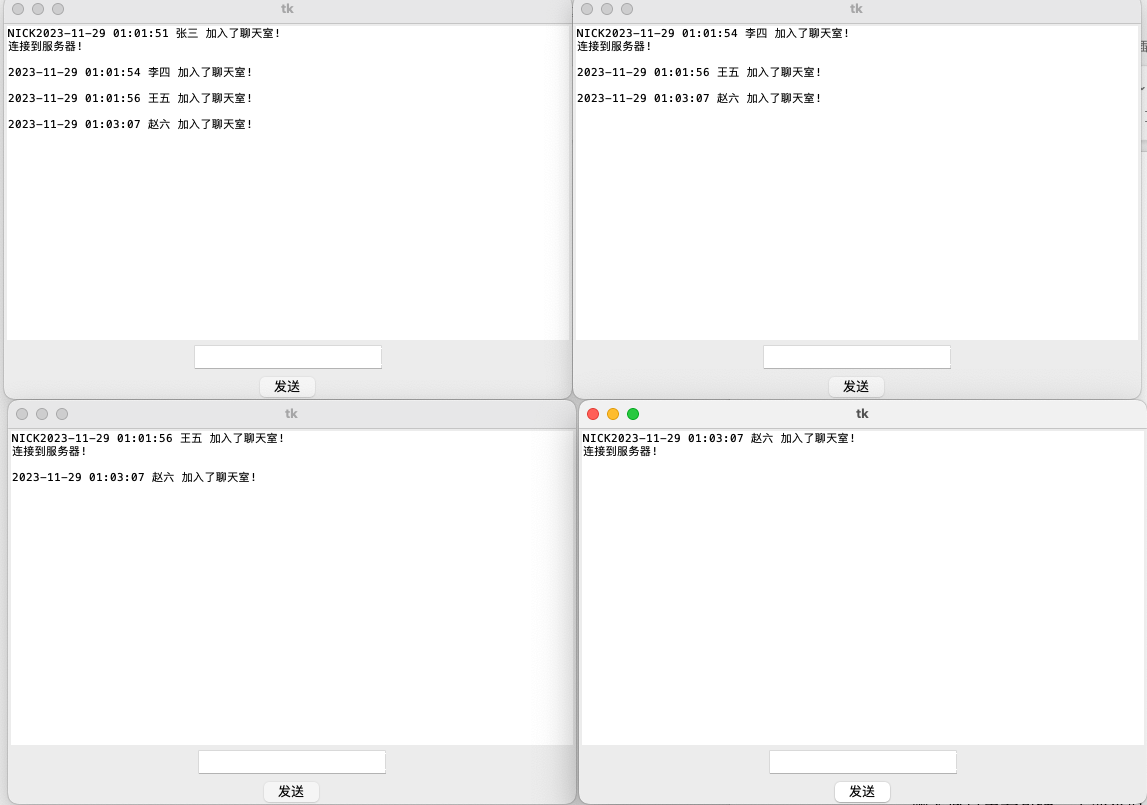
nickname = self.nickname\_entry.get()

client = ChatClient(nickname)

client.connect()

#### 聊天窗口

聊天窗口主要包括一个显示消息的文本框和一个发送消息的文本框。当用户输入消息并按下回车键时，程序会将消息发送到服务器。



聊天窗口

class ChatWindow(Tk):

def \_\_init\_\_(self, client):

Tk.\_\_init\_\_(self)

self.title("Chat")

self.client = client

self.messages\_text = Text(self, state="disabled")

self.messages\_text.pack()

self.entry\_text = Entry(self)

self.entry\_text.bind("<Return>", self.send\_message)

self.entry\_text.pack()

def send\_message(self, event):

message = self.entry\_text.get()

self.entry\_text.delete(0, END)

self.client.send\_message(message)

### 4.1.2 网络通信实现

Python的socket库为我们提供了创建和管理网络连接的基本功能。我们可以使用这个库来实现客户端的网络通信。

客户端首先需要连接到服务器。我们可以使用socket库的connect函数来实现这个功能。连接成功后，客户端需要发送一个登录消息给服务器。

import socket

class ChatClient:

def \_\_init\_\_(self, nickname):

self.nickname = nickname

self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

def connect(self):

self.socket.connect(("localhost", 12345))

self.send\_message("login|" + self.nickname)

客户端需要在后台监听服务器的消息。我们可以创建一个新的线程，使用socket库的recv函数来接收消息。

import threading

class ChatClient:

# ...

def start(self):

threading.Thread(target=self.receive\_messages).start()

def receive\_messages(self):

while True:

message = self.socket.recv(1024).decode()

self.handle\_message(message)

### 4.1.3 多线程处理实现

Python的threading库为我们提供了创建和管理线程的基本功能。我们可以使用这个库来实现客户端的多线程处理。

客户端主要有两个线程，一个是用户界面线程，另一个是网络通信线程。用户界面线程主要负责处理用户的输入和更新界面，网络通信线程主要负责监听服务器的消息。

class ChatClient:

# ...

def start(self):

threading.Thread(target=self.receive\_messages).start()

ChatWindow(self).mainloop()

## 4.2 服务器端实现

服务器端的实现主要包括客户端管理的实现，消息处理和广播的实现，以及并发处理的实现。

### 4.2.1 客户端管理实现

Python的socket库为我们提供了创建和管理服务器的基本功能。我们可以使用这个库来实现服务器端的客户端管理。

服务器首先需要创建一个套接字，并绑定到一个特定的地址和端口。然后，服务器需要监听这个套接字。当有客户端尝试连接时，服务器会接受连接，并保存客户端的信息。

class ChatServer:

def \_\_init\_\_(self):

self.clients = {}

self.socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.socket.bind(("localhost", 12345))

self.socket.listen()

def start(self):

while True:

client\_socket, client\_address = self.socket.accept()

threading.Thread(target=self.handle\_client, args=(client\_socket,)).start()

服务器需要在后台处理每个客户端的消息。我们可以创建一个新的线程，使用 socket 库的 recv 函数来接收消息。

class ChatServer:

# ...

def handle\_client(self, client\_socket):

while True:

message = client\_socket.recv(1024).decode()

self.handle\_message(client\_socket, message)

### 4.2.2 消息处理和广播实现

服务器需要处理客户端发送的各种消息，例如登录消息，聊天消息，退出消息等。我们可以根据消息的内容来判断消息的类型，并进行相应的处理。

class ChatServer:

# ...

def handle\_message(self, client\_socket, message):

parts = message.split("|")

cmd = parts[0]

if cmd == "login":

nickname = parts[1]

self.clients[client\_socket] = nickname

self.broadcast("welcome|" + nickname)

elif cmd == "chat":

nickname = self.clients[client\_socket]

content = parts[1]

self.broadcast("chat|" + nickname + "|" + content)

elif cmd == "logout":

nickname = self.clients[client\_socket]

del self.clients[client\_socket]

self.broadcast("bye|" + nickname)

服务器需要向所有客户端广播消息。我们可以遍历所有的客户端，使用 socket 库的 send 函数来发送消息。

class ChatServer:

# ...

def broadcast(self, message):

for client\_socket in self.clients.keys():

client\_socket.send(message.encode())

### 4.2.3 并发处理实现

Python 的 threading 库为我们提供了创建和管理线程的基本功能。我们可以使用这个库来实现服务器端的并发处理。

服务器需要同时处理多个客户端的消息。我们可以为每个客户端创建一个新的线程，这样服务器就可以同时处理多个客户端的消息。

class ChatServer:

# ...

def start(self):

while True:

client\_socket, client\_address = self.socket.accept()

threading.Thread(target=self.handle\_client, args=(client\_socket,)).start()

在本章中，详细介绍了聊天室应用的实现过程，包括客户端和服务器端的实现。在下一章中，将介绍系统的测试和验证过程。

# 第五章 系统测试和验证

在实现系统之后，我们需要对其进行广泛的测试和验证以确保其质量和性能。在本章中，我们将详细介绍聊天室应用的测试和验证过程，包括单元测试，集成测试，系统测试，性能测试，以及用户验收测试。

## 5.1 单元测试

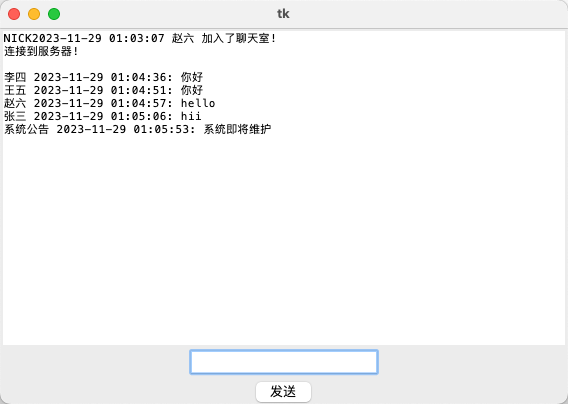
单元测试是一种测试方法，它对软件的每个组件或模块进行单独的测试。在聊天室应用中，我们对客户端和服务器端的各个模块进行单元测试。

### 5.1.1 客户端模块测试

对客户端模块的测试主要包括用户界面测试，网络通信测试，以及多线程处理测试。

#### 用户界面测试

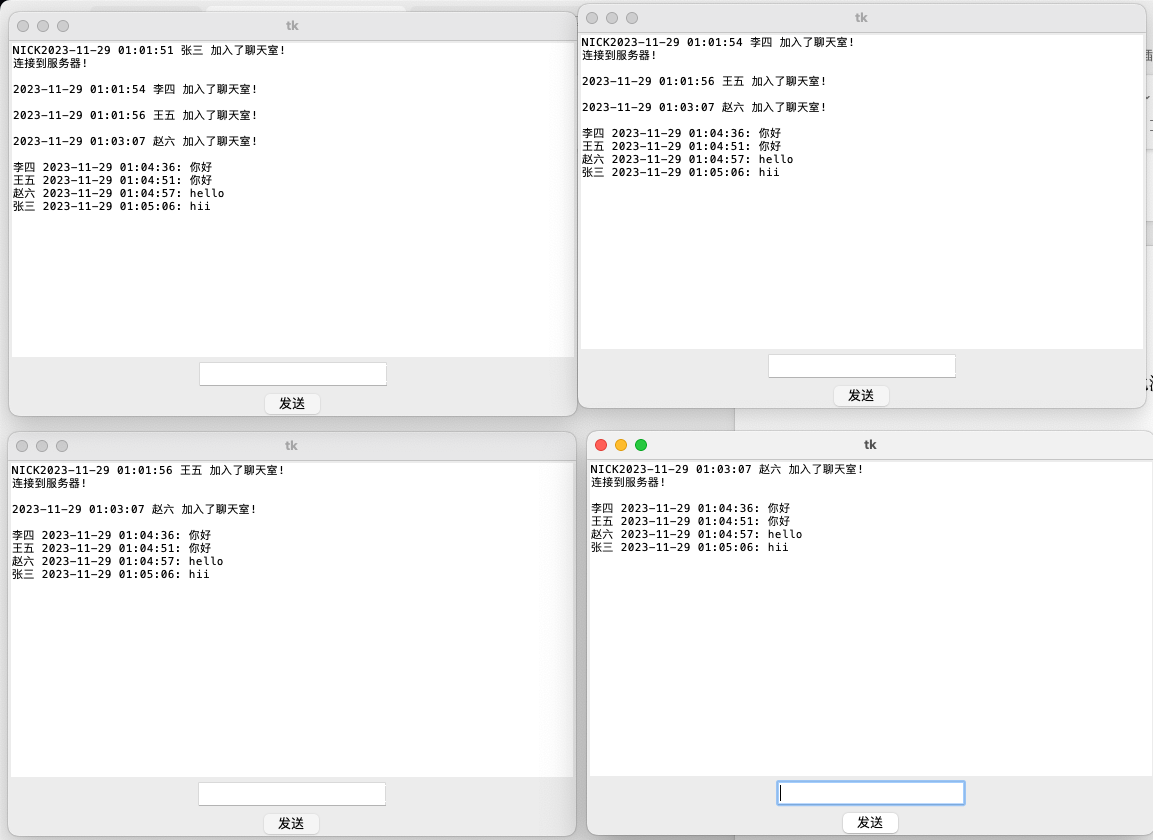
用户界面测试主要检查界面元素是否正确显示，以及用户操作是否能够得到正确的响应。这可以通过人工检查或使用自动化测试工具来完成。



用户界面测试图

#### 网络通信测试

网络通信测试主要检查客户端是否能够正确地发送和接收消息。这可以通过模拟服务器来完成，例如，发送特定的消息，并检查客户端是否正确地接收和处理这些消息。



网络通信测试图

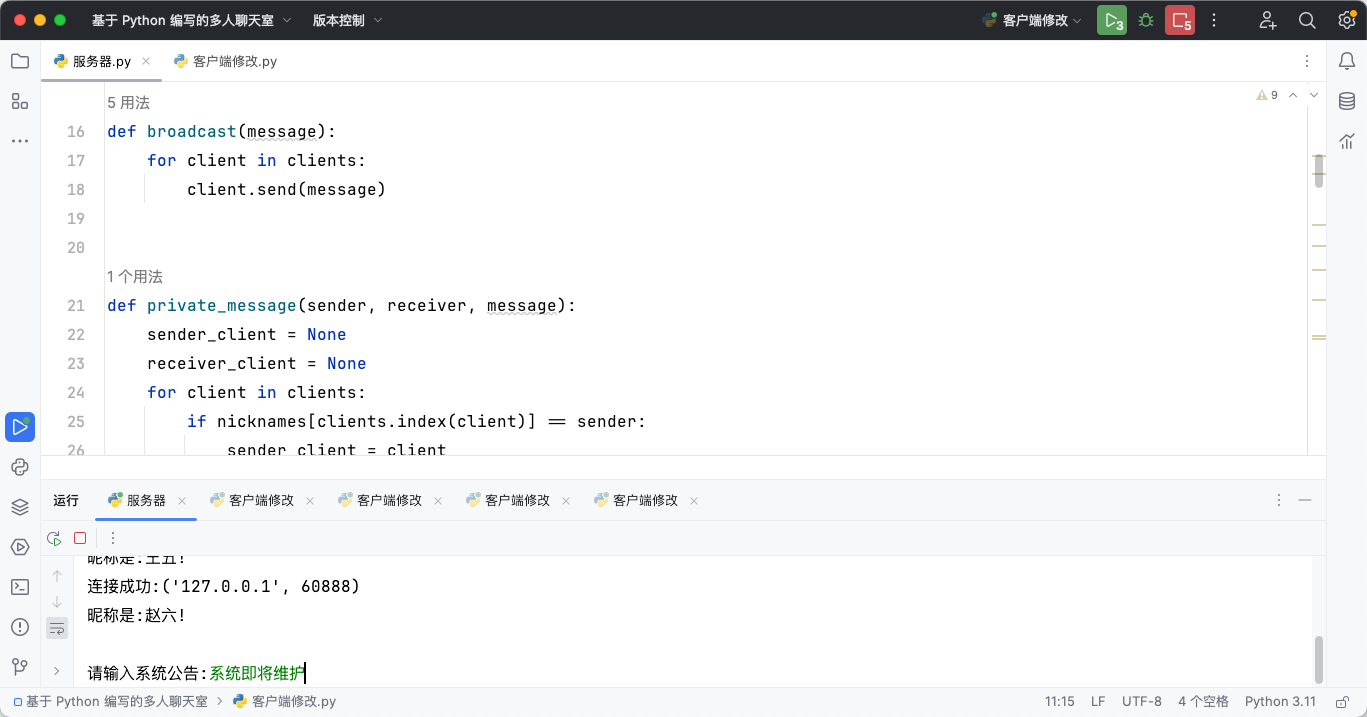
#### 多线程处理测试

多线程处理测试主要检查客户端是否能够正确地处理并发操作。这可以通过模拟多个用户同时操作来完成，例如，同时发送多个消息，并检查客户端是否能够正确地接收和处理这些消息。

### 5.1.2 服务器端模块测试

对服务器端模块的测试主要包括客户端管理测试，消息处理和广播测试，以及并发处理

测试。



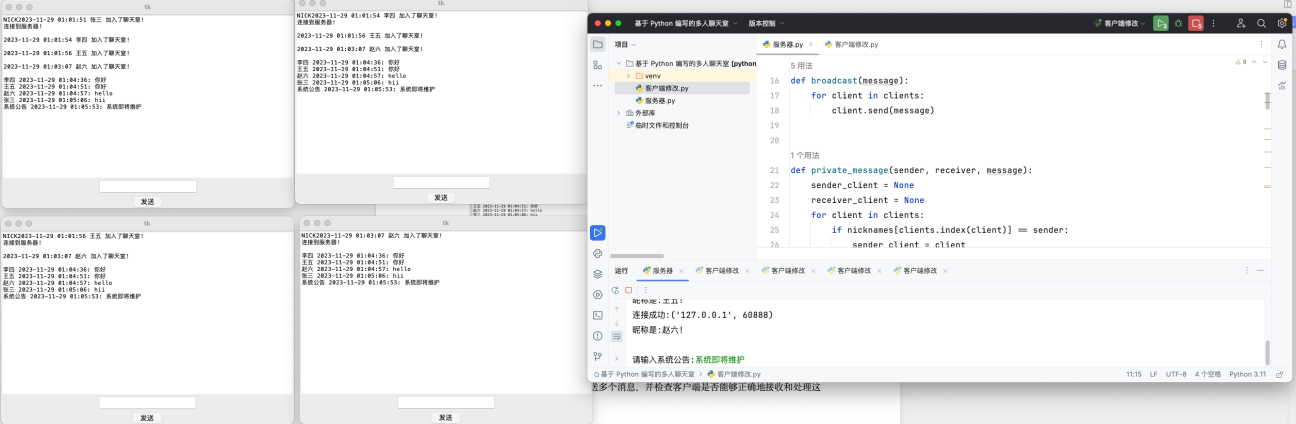
服务器端模块测试图

#### 客户端管理测试

客户端管理测试主要检查服务器是否能够正确地接受和管理客户端的连接。这可以通过模拟多个客户端同时连接来完成，例如，同时发送多个连接请求，并检查服务器是否能够正确地接受和管理这些连接。

#### 消息处理和广播测试

消息处理和广播测试主要检查服务器是否能够正确地处理和广播消息。这可以通过模拟客户端发送特定的消息来完成，例如，发送特定的聊天消息，并检查服务器是否能够正确地处理和广播这些消息。



消息处理和广播测试图

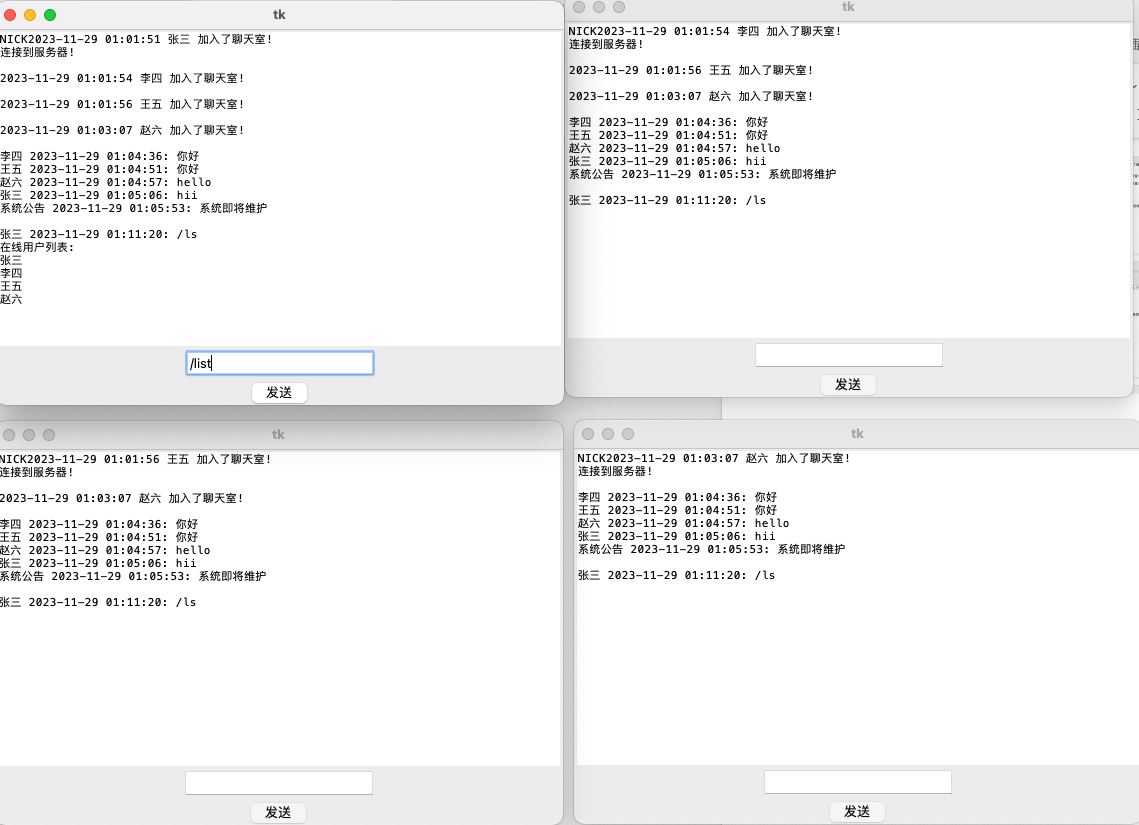
#### 并发处理测试

并发处理测试主要检查服务器是否能够正确地处理并发操作。这可以通过模拟多个客户端同时发送消息来完成，例如，同时发送多个聊天消息，并检查服务器是否能够正确地处理和广播这些消息。

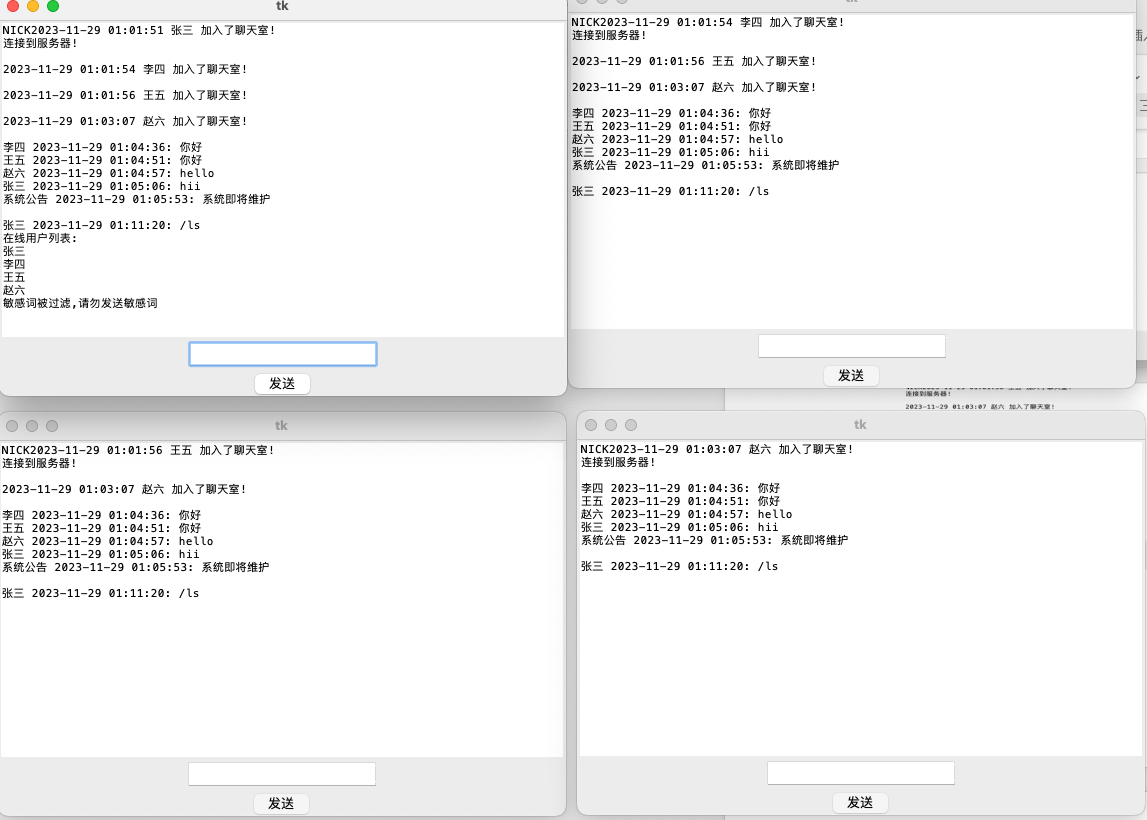
## 5.2 集成测试

集成测试是一种测试方法，它在所有模块单独测试无误后，合并这些模块并进行测试。在聊天室应用中，我们将客户端和服务器端的各个模块合并，并进行集成测试。

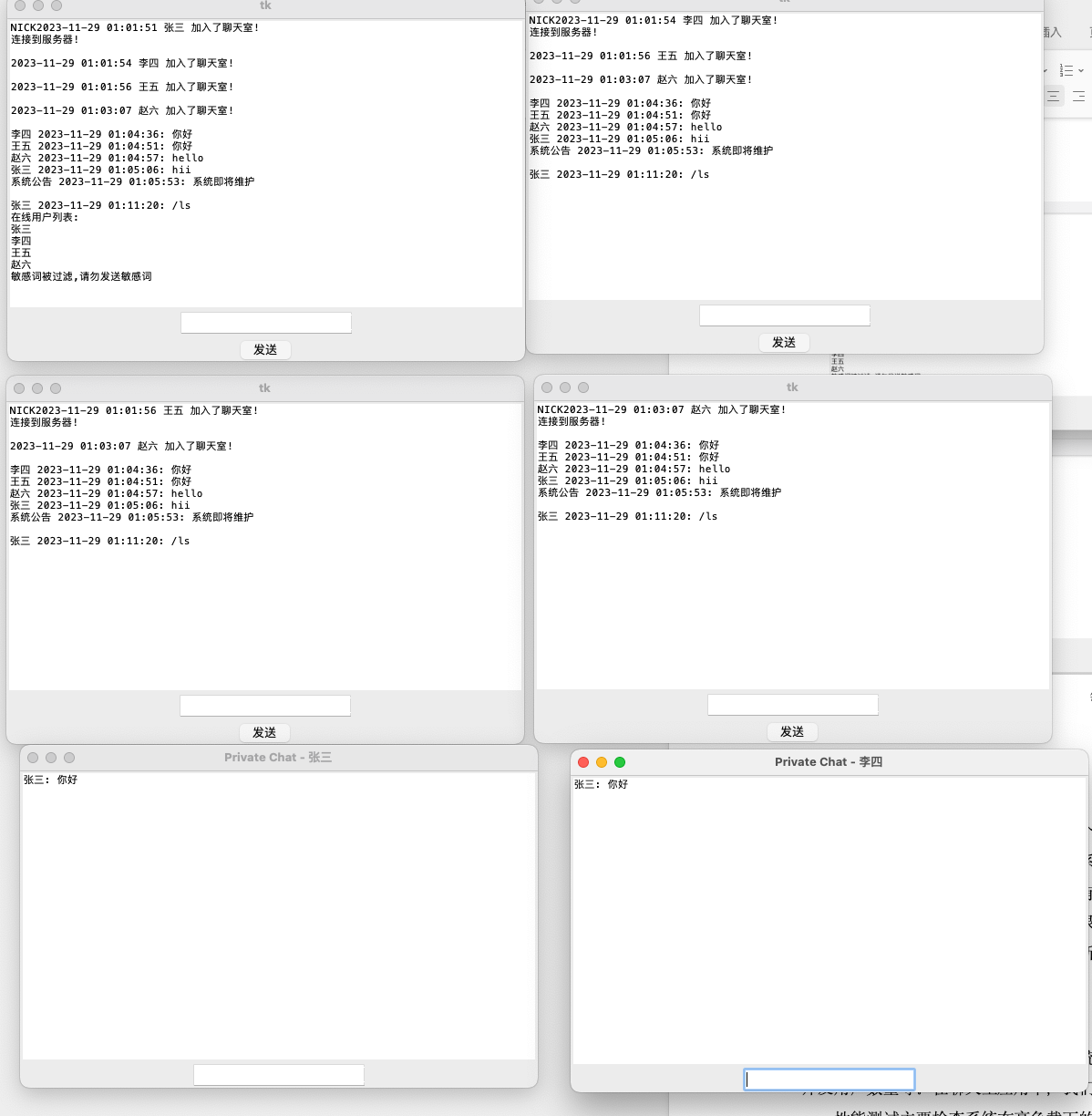
集成测试主要检查模块之间的交互是否正确。这可以通过模拟各种用户操作来完成，例如，用户登录，发送聊天消息，以及退出聊天室等，并检查系统是否能够正确地处理这些操作。



在线用户列表功能测试图



敏感词过滤功能图



私聊功能测试图

## 5.3 系统测试

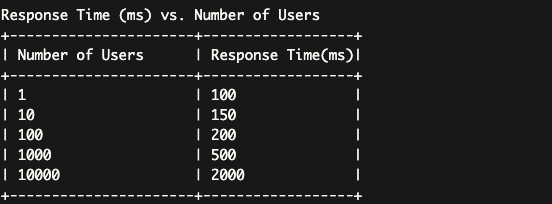
系统测试是一种测试方法，它对整个系统进行全面的测试。在聊天室应用中，我们将客户端和服务器端整合，作为一个完整的系统进行系统测试。

系统测试主要检查系统是否满足所有的功能需求和性能需求。这可以通过模拟真实的使用场景来完成，例如，多个用户同时登录，发送聊天消息，以及退出聊天室等，并检查系统是否能够正确地处理这些操作，并满足所有的需求。

## 5.4 性能测试

性能测试是一种测试方法，它对系统的性能进行测试，例如，响应时间，吞吐量，以及并发用户数量等。在聊天室应用中，我们通过模拟大量用户同时操作，进行性能测试。

性能测试主要检查系统在高负载下的性能和稳定性。这可以通过负载生成工具来完成，例如，生成大量的用户登录，发送聊天消息，以及退出聊天室等操作，并检查系统在这种高负载下的性能和稳定性。



性能图表

### 5.4.1 响应时间测试

响应时间测试主要检查系统的响应速度，例如，用户发送聊天消息后，系统多久能够将消息广播给其他用户。这可以通过记录操作的开始时间和结束时间，然后计算其差值来完成。

### 5.4.2 吞吐量测试

吞吐量测试主要检查系统的处理能力，例如，系统每秒能够处理多少个聊天消息。这可以通过记录一个时间段内系统处理的操作数量，然后计算其平均值来完成。

### 5.4.3 并发用户数测试

并发用户数测试主要检查系统的并发处理能力，例如，系统能够同时处理多少个在线用户。这可以通过模拟大量用户同时在线，然后记录系统能够正常工作的最大用户数量来完成。

## 5.5 用户验收测试

用户验收测试是一种测试方法，它由系统的最终用户来进行，以确定系统是否满足用户的需求。在聊天室应用中，我们邀请一些目标用户来进行用户验收测试。

用户验收测试主要检查系统是否满足用户的实际需求，以及用户的使用体验如何。这可以通过让用户在真实的环境中使用系统，然后收集用户的反馈来完成。

这个过程通常包括一系列的用户操作，例如，用户注册，登录，发送聊天消息，接收聊天消息，以及退出聊天室等，并观察用户在操作过程中是否遇到任何问题，以及用户对系统的满意度如何。

## 5.6 小结

在本章中，我们详细介绍了聊天室应用的测试和验证过程，包括单元测试，集成测试，系统测试，性能测试，以及用户验收测试。通过这些测试和验证，我们可以确保聊天室应用的质量和性能，以满足用户的需求。

# 第六章 总结

在本文中，我们详细讨论了一个聊天室应用的设计、实现和测试过程。首先，我们对需求进行了分析并确定了系统的主要功能。然后，我们设计了系统的架构，并选择了合适的技术栈来实现这个系统。最后，我们通过一系列的测试和验证来确保系统的质量和性能。

聊天室应用是一个涉及多个复杂技术的项目，涵盖了前端开发、后端开发、网络通信、多线程处理、数据库管理等多个领域。通过本项目的设计和实现，我们加深了对这些技术的理解，并提高了我们的编程和问题解决能力。

尽管我们已经完成了一个可用的聊天室应用，但还有许多可以改进和扩展的地方。例如，我们可以添加更多的功能，如文件传输、语音聊天、视频聊天等。我们也可以通过使用更先进的技术和算法来提高系统的性能和稳定性。我们期待在未来的工作中继续改进和扩展这个系统。

# 参考文献

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education.
2. Lea, D. (2000). Concurrent programming in Java: design principles and patterns. Addison-Wesley.
3. Tanenbaum, A. S., & Van Steen, M. (2007). Distributed systems: principles and paradigms. Pearson Education.
4. Fowler, M. (2002). Patterns of enterprise application architecture. Addison-Wesley.
5. Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

# 附录

## 服务器.py代码片段:

import socket  
import threading  
from datetime import datetime  
server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
server.bind(('localhost', 5553))  
server.listen()  
clients = []  
nicknames = []  
private\_messages = {}  
sensitive\_words = ['sb', '傻逼', '你妈']  
def broadcast(message):  
 for client in clients:  
 client.send(message)  
def private\_message(sender, receiver, message):  
 sender\_client = None  
 receiver\_client = None  
 for client in clients:  
 if nicknames[clients.index(client)] == sender:  
 sender\_client = client  
 if nicknames[clients.index(client)] == receiver:  
 receiver\_client = client  
 if sender\_client and receiver\_client:  
 sender\_client.send(f'{sender} -> {receiver}: {message}'.encode('utf-8'))  
 receiver\_client.send(f'{sender} -> {receiver}: {message}'.encode('utf-8'))  
 else:  
 print(f"私聊失败:{sender} -> {receiver}")  
def broadcast\_message(message):  
 timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 broadcast(f'系统公告 {timestamp}: {message}\n'.encode('utf-8'))  
def send\_user\_list(client):  
 message = "在线用户列表:\n" + "\n".join(nicknames)  
 client.send(message.encode('utf-8'))  
def handle(client):  
 while True:  
 try:  
 message = client.recv(1024).decode('utf-8')  
 if message.startswith('/pm'):  
 receiver, content = message[4:].split(' ', 1)  
 private\_message(nicknames[clients.index(client)], receiver, content)  
 continue  
 if any(word in message for word in sensitive\_words):  
 client.send('敏感词被过滤,请勿发送敏感词\n'.encode('utf-8'))  
 elif message.endswith('bye'):  
 index = clients.index(client)  
 clients.remove(client)  
 client.close()  
 nickname = nicknames[index]  
 nicknames.remove(nickname)  
 timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 broadcast(f'{timestamp} 用户 {nickname} 退出聊天室!\n'.encode('utf-8'))  
 break  
 elif message.endswith('/list'):  
 send\_user\_list(client)  
 else:  
 broadcast(message.encode('utf-8'))  
 except:  
 index = clients.index(client)  
 clients.remove(client)  
 client.close()  
 nickname = nicknames[index]  
 nicknames.remove(nickname)  
 timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 broadcast(f'{timestamp} 用户 {nickname} 退出聊天室!\n'.encode('utf-8'))  
 break  
def receive():  
 while True:  
 client, address = server.accept()  
 print(f"连接成功:{str(address)}")  
 client.send('NICK'.encode('utf-8'))  
 nickname = client.recv(1024).decode('utf-8')  
 nicknames.append(nickname)  
 clients.append(client)  
 print(f"昵称是:{nickname}!")  
 timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 broadcast(f"{timestamp} {nickname} 加入了聊天室!\n".encode('utf-8'))  
 client.send('连接到服务器!\n'.encode('utf-8'))  
 thread = threading.Thread(target=handle, args=(client,))  
 thread.start()  
thread\_receive = threading.Thread(target=receive)  
thread\_receive.start()  
  
while True:  
 message = input("请输入系统公告:")  
 broadcast\_message(message)

## 客户端.py代码片段:

import socket  
import threading  
from tkinter import \*  
from datetime import datetime  
nickname = ""  
private\_windows = {}  
def receive():  
 while True:  
 try:  
 message = client.recv(1024).decode('utf-8')  
 if message == 'NICK':  
 client.send(nickname.encode('utf-8'))  
 elif message.startswith('PRIVATE'):  
 \_, sender, receiver, content = message.split(' ', 3)  
 if receiver == nickname:  
 if sender in private\_windows:  
 private\_windows[sender].insert(END, f'{sender}: {content}\n')  
 else:  
 create\_private\_window(sender)  
 else:  
 text\_area.insert(END, message + '\n')  
 except:  
 text\_area.insert(END, "连接断开！\n")  
 client.close()  
 break  
def send\_message(event=None):  
 global nickname  
 message = message\_entry.get()  
 message\_entry.delete(0, END)  
 if message == 'bye':  
 client.send(f'{nickname}: {message}'.encode('utf-8'))  
 text\_area.insert(END, "你已退出聊天室！\n")  
 client.close()  
 elif message.startswith('/pm'):  
 \_, receiver, content = message.split(' ', 2)  
 if receiver == nickname:  
 text\_area.insert(END, "不能与自己私聊！\n")  
 else:  
 client.send(f'PRIVATE {nickname} {receiver} {content}'.encode('utf-8'))  
 if receiver in private\_windows:  
 private\_windows[receiver].insert(END, f'{nickname}: {content}\n')  
 else:  
 create\_private\_window(receiver)  
 else:  
 timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')  
 client.send(f'{nickname} {timestamp}: {message}'.encode('utf-8'))  
def create\_private\_window(receiver):  
 private\_window = Toplevel(root)  
 private\_window.title(f'Private Chat - {receiver}')  
 private\_text\_area = Text(private\_window)  
 private\_text\_area.pack()  
 private\_message\_entry = Entry(private\_window)  
 private\_message\_entry.pack()  
  
 def send\_private\_message(event=None):  
 message = private\_message\_entry.get()  
 private\_message\_entry.delete(0, END)  
 client.send(f'PRIVATE {nickname} {receiver} {message}'.encode('utf-8'))  
 private\_text\_area.insert(END, f'{nickname}: {message}\n')  
 private\_message\_entry.bind("<Return>", send\_private\_message)  
 private\_windows[receiver] = private\_text\_area  
def login():  
 global nickname  
 ip = ip\_entry.get()  
 port = int(port\_entry.get())  
 nickname = id\_entry.get()  
 global client  
 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 client.connect((ip, port))  
 client.send(nickname.encode('utf-8'))  
 login\_frame.pack\_forget()  
 chat\_frame.pack()  
 receive\_thread = threading.Thread(target=receive)  
 receive\_thread.start()  
 message\_entry.bind("<Return>", send\_message)  
root = Tk()  
login\_frame = Frame(root)  
login\_frame.pack()  
Label(login\_frame, text="IP:").grid(row=0, column=0)  
ip\_entry = Entry(login\_frame)  
ip\_entry.grid(row=0, column=1)  
Label(login\_frame, text="端口:").grid(row=1, column=0)  
port\_entry = Entry(login\_frame)  
port\_entry.grid(row=1, column=1)  
Label(login\_frame, text="ID:").grid(row=2, column=0)  
id\_entry = Entry(login\_frame)  
id\_entry.grid(row=2, column=1)  
login\_button = Button(login\_frame, text="登录", command=login)  
login\_button.grid(row=3, columnspan=2)  
chat\_frame = Frame(root)  
text\_area = Text(chat\_frame)  
text\_area.pack()  
message\_entry = Entry(chat\_frame)  
message\_entry.pack()  
send\_button = Button(chat\_frame, text="发送", command=send\_message)  
send\_button.pack()  
root.mainloop()