目录

安全通信软件的设计与实现

1 任务描述

结合所学安全机制设计实现一个简单的安全通信软件，包含机密性，消息认证等基本功能。并考虑其中涉及的密钥分配方式与机密性算法等相关问题的解决. 实现方法不限，使用机制不限。

要求：

- 独立完成
- 具有完整的流程设计，报文格式等相关分析。
- 具备自圆其说的安全性设计思考

2 项目简介

本项目是一个基于 C/S 架构的轻量级通信软件，名为“小鳄鱼聊天室”，旨在提供一个简洁、直观且流畅的在线交流平台。在 Java 环境下独立开发完成的本项目，实现了基本的文本聊天功能，并采用一系列安全协议，确保了客户端与服务器之间通信的加密，保障了用户数据的安全性。

主要功能：

- **用户注册与登录**：新用户可以通过注册功能创建自己的账号，而已注册用户可以通过登录功能进入聊天室。
- **添加与删除好友**：用户能够搜索其他用户并发送好友请求，一旦对方 接受，双方即成为好友。用户也可以管理自己的好友列表，包括删除 不再联系的好友。
- **实时文本聊天**：用户可以与好友进行单独的实时文本对话，支持发送 消息和接收即时回复，让沟通变得轻松而高效。

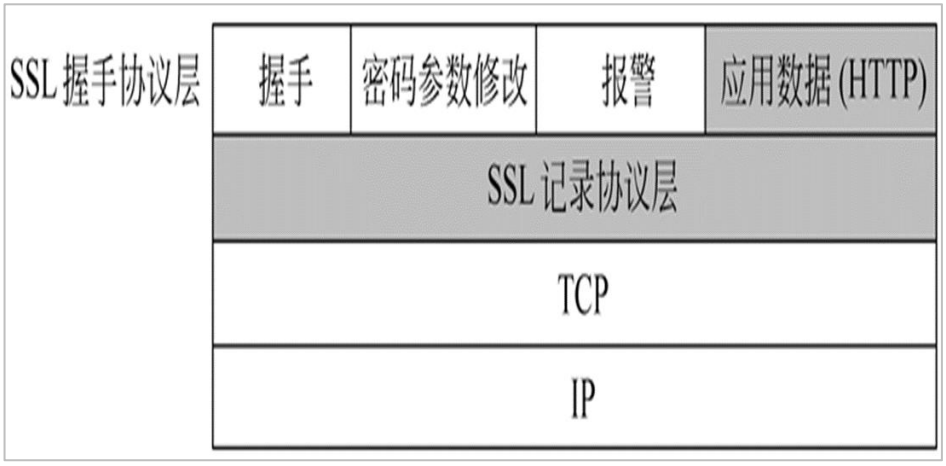
3 项目原理

本项目在设计和实现过程中，精心挑选了一系列成熟的技术和协议，以确保

软件的功能性、安全性和用户体验。下面详细介绍各项技术的选择和使用。

3.1 SSL\TLS 协议

为保障数据在客户端与服务器间传输的机密性和完整性，本项目采用了 SSL/TLS 协议。



SSL（Secure Sockets Layer）和 TLS（Transport Layer Security）是网络通信协议，用于在互联网上为数据传输提供安全和数据完整性保护。TLS 是 SSL 的后续版本，提供了更强的安全性，但两者在基本原理上是相似的。以下是 SSL/TLS 的工作原理：

1.握手过程

SSL/TLS 的握手过程是建立安全通信的关键，分为几个步骤：

客户端 Hello（ClientHello）

客户端开始 SSL/TLS 握手，发送一个 ClientHello 消息，其中包含客户端支持的 SSL/TLS 版本、加密套件列表以及一个客户端随机数（Client Random）。

服务器 Hello（ServerHello）

服务器回应一个 ServerHello 消息，选择一个客户端也支持的加密套件和 SSL/TLS 版本，并提供一个服务器随机数（Server Random）。

服务器证书（Server Certificate）

服务器发送它的证书给客户端，证书中包含了服务器的公钥。

密钥交换（Key Exchange）

客户端验证服务器的证书是否可信，然后使用证书中的公钥来加密一个预主

密钥（Pre-Master Secret）并发送给服务器。服务器用自己的私钥解密得到预主密钥。

握手完成

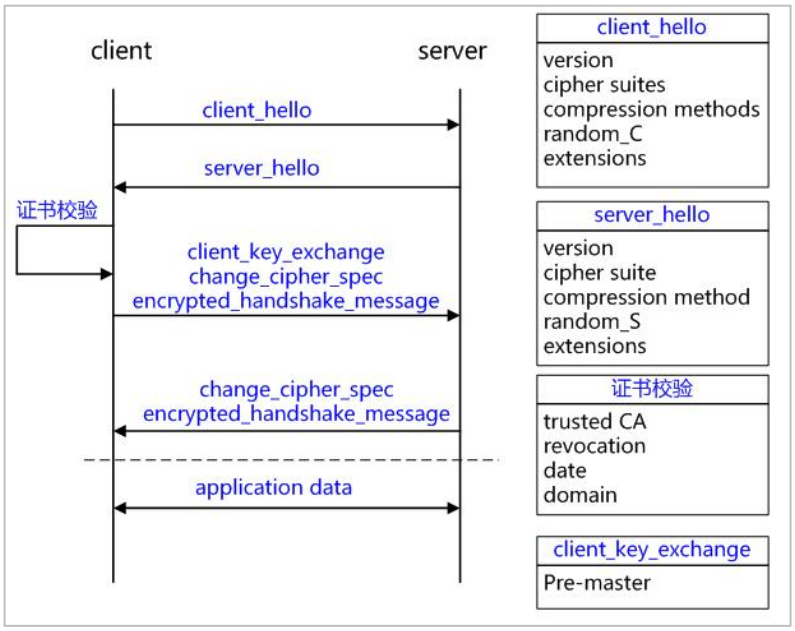
客户端和服务器都使用预主密钥和之前交换的随机数生成最终的会话密钥（Session Key）。接下来的通信将使用这个会话密钥进行加密，确保传输数据的机密性。

2.数据传输

在握手过程成功完成后，客户端和服务器就建立了一个安全的加密通道。所有传输的数据都将使用会话密钥进行对称加密。只有客户端和服务器才有对应的密钥来加密和解密数据，从而保证了传输过程中的安全性。

3.会话结束

一旦通信结束或者一方想要结束会话，会发送一个 close_notify 警告，随后关闭连接。之后的任何尝试重新连接都需要进行新的握手过程。



加密组件

SSL/TLS 的安全性基于以下几个加密组件：

a）对称加密

客户端和服务器使用相同的密钥来加密和解密数据，如 AES、3DES 等。

b）非对称加密

在握手阶段使用，如 RSA、ECC 等。服务器的公钥用来加密信息，私钥用来

解密。

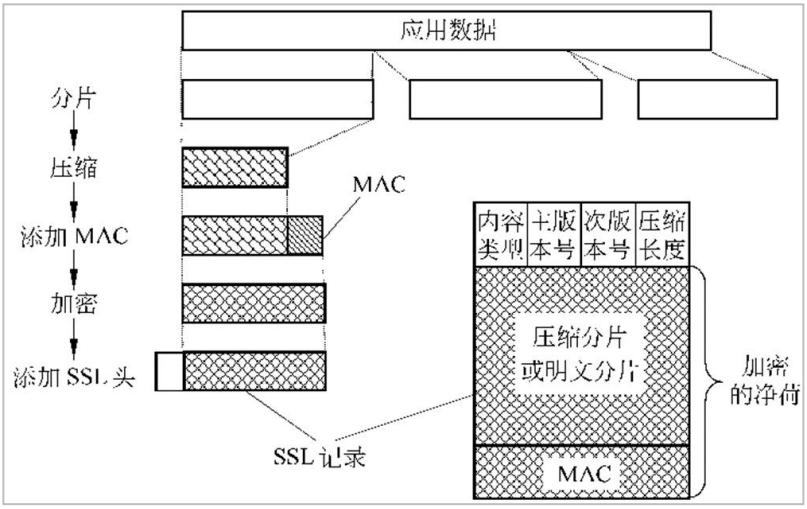
c)数字证书和签名

数字证书用来验证服务器的身份，通常由可信的证书颁发机构（CA）签发。数字签名保证了证书的真实性。

d)安全散列算法

如 SHA-256 等，用于验证数据的完整性，防止篡改。

e)SSL 记录格式



3.2 WebSocket 协议

考虑到实时通信的需求，本项目选用 WebSocket 协议作为客户端与服务器之间的通信协议。

WebSocket 是一个网络通信协议，提供了一种在客户端和服务器之间建立持久连接的方式，并能够实现全双工通信。这意味着服务器可以随时向客户端发送消息，客户端也可以随时向服务器发送消息，而不需要像传统的 HTTP 请求那样每次都建立一个新的连接。

1.握手过程

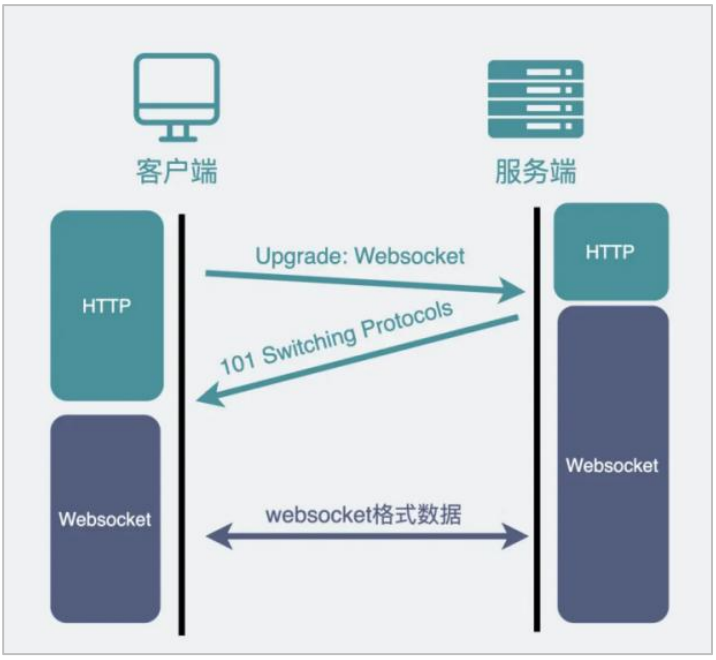
WebSocket 连接的建立始于一个 HTTP 请求，这个过程称为“握手”。客户端发送一个 HTTP 请求到服务器

这个 HTTP 请求告诉服务器，客户端希望将通信升级到 WebSocket。如果服务器支持 WebSocket 并同意升级，它将响应一个 HTTP 101 状态码，表示切换协

议，并包含确认的头信息。

2.数据帧和消息

一旦握手成功，客户端和服务器之间将建立 WebSocket 连接。数据通过称为“帧”的小片段来传输。WebSocket 定义了几种不同类型的帧，每种类型都有不同的用途。例如，有文本帧、二进制帧、关闭帧等。消息可能会分成多个帧进行发送，接收方需要将这些帧重新组装成完整的消息。



本次项目使用了 Java-WebSocket 开源代码库实现的 WebSocket 通信协议。

3.3 JavaFX 框架

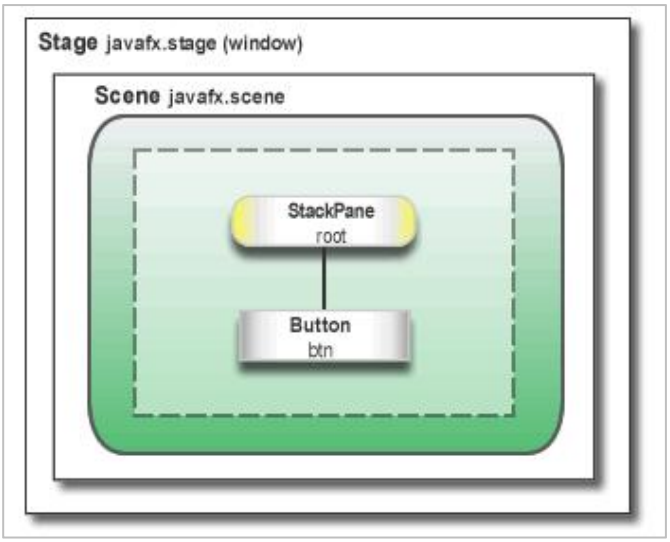
为了提供直观、流畅的用户界面，本项目选择 JavaFX 作为客户端界面开发框架。

JavaFX 是一个用于开发桌面应用和丰富互联网应用程序（RIA）的图形库。它提供了一系列先进的 UI 控件，并支持 3D 图形、多媒体内容处理以及各种动画效果。JavaFX 支持 FXML，允许开发者以声明性方式设计用户界面，与后端逻辑分离，提高了开发效率和界面的可维护性。JavaFX 的 CSS 样式支持使得定制化 UI 变得非常灵活，能够创造出既美观又符合品牌特色的用户界面。

JavaFX 使用硬件加速的图形管道进行渲染，称为 Prism。此外，为了完全加速图形的使用，它通过内部使用 DirectX 和 OpenGL 来利用软件或硬件渲染机制。

JavaFX 具有依赖于平台的 Glass 窗口化工具包层，用于连接到本机操作系统。它使用操作系统的事件队列来计划线程使用。此外，它还异步处理窗口、事件、计时器、媒体和 Web 引擎，支持媒体播放和 HTML/CSS。

JavaFX 应用程序的主要结构如下：



在这里，我们注意到两个主要容器：

• Stage 是应用程序的主要容器和入口点。它表示主窗口并作为 start（）方法的参数传递。

• Scene 是用于保存 UI 元素（如图像视图、按钮、网格、文本框）的容器。场景可以替换或切换到另一个场景。这表示分层对象的图，称为场景图。该层次结构中的每个元素都称为一个节点。单个节点有其 ID、样式、效果、事件处理程序和状态。此外，场景还包含布局容器、图像、媒体。以下是 JavaFX 的特点介绍：

1.线程

在系统级别，JVM 创建单独的线程来运行和呈现应用程序：

• Prism rendering thread——负责单独渲染场景图。

• Application thread——是任何 JavaFX 应用程序的主线程，所有活动节点和组件都附加到此线程。

2.生命周期

javafx.application.Application 类具有以下生命周期方法：

•init()——在创建应用程序实例后调用。此时，JavaFX API 尚未准备就绪，因此无法在此处创建图形组件。

• start(Stage stage)——所有图形组件都在此处创建，图形活动的主线从这里开始。

• stop()——在应用程序关闭之前调用;例如，当用户关闭主窗口时。在应用程序终止之前重写此方法。

• launch()方法启动 JavaFX 应用程序。

3.FXML

JavaFX 使用特殊的 FXML 标记语言来创建视图接口。这提供了一个基于 XML 的结构，用于将视图与业务逻辑分离。XML 在这里更合适，因为它能够非常自然地表示 Scene Graph 层次结构。

此外，使用 SceneBuilder 软件（或者 idea 的 SceneBuilder 插件）可以使用拖拽的方式自动生成.fxml 文件，更近进一步地简化了 UI 的开发。

最后，为了加载.fxml 文件，我们使用 FXMLLoader 类，它生成了场景层次结构的对象图。

3.4 策略模式

在安全机制的实现中，本项目采用了策略模式。策略模式允许在运行时选择算法或行为，使得服务端可以根据情况调用不同的算法处理客户端的请求。这种模式提高了代码的可扩展性和灵活性，便于未来对安全策略的更新和替换，确保软件能够适应不断变化的安全需求。

该设计模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。策略模式主要解决在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。当一个系统有许多许多类，而区分它们的只是他们直接的行为，我们就可以将这些算法封装成一个一个的类，任意地替换。实现关键是实现同一个接口。

优点：

1. 算法可以自由切换。
2. 避免使用多重条件判断。

3. 扩展性良好。

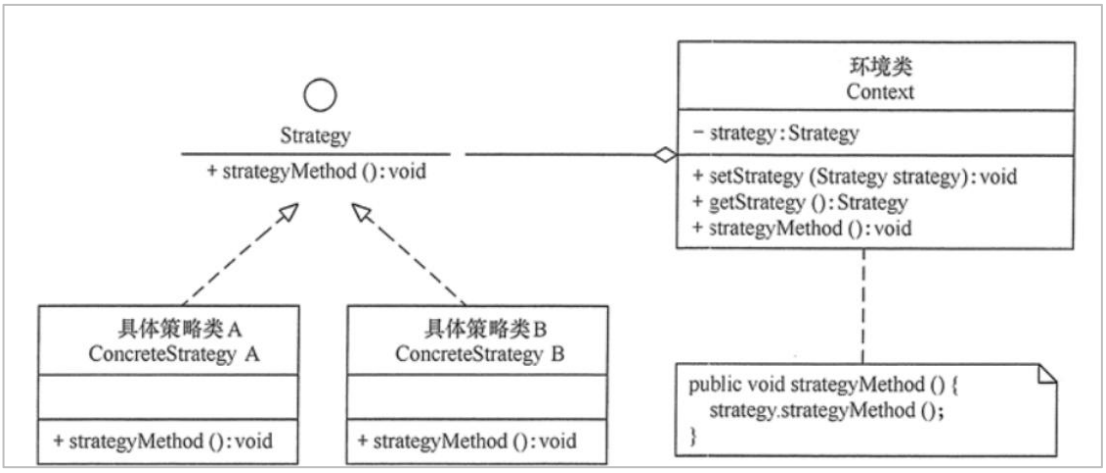
缺点：

1. 策略类会增多。
2. 所有策略类都需要对外暴露。

使用场景：

1. 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
2. 一个系统需要动态地在几种算法中选择一种。
3. 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

策略模式结构图：



可以看出策略模式有三个组成角色：

- 抽象策略(Strategy)类
- 具体策略(Concrete Strategy)类
- 环境(Context)类

策略模式通过将算法与使用算法的代码解耦，提供了一种动态选择不同算法的方法。客户端代码不需要知道具体的算法细节，而是通过调用环境类来使用所选择的策略。

3.5 数字签名

为了验证消息的来源和完整性，本项目在消息传输中实现了数字签名功能。

数字签名是一种电子签名，用于模拟传统的手写签名，但提供更高的安全性。它不仅能证明消息未被篡改，还能确认消息发送者的身份，是确保通信安全的重要技术。

数字签名的流程大致如下：

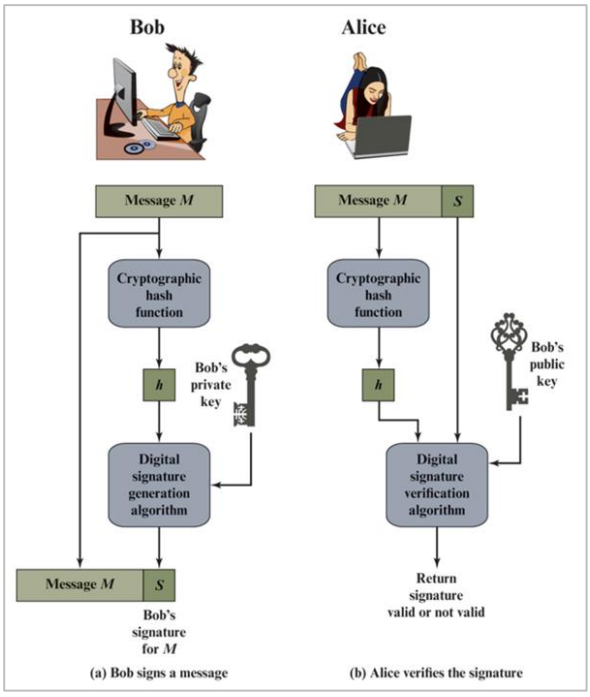
1. **消息摘要生成：**首先，使用哈希函数（如 SHA-256）对需要发送的消息进行处理，生成一个固定长度的消息摘要。这一步确保了即使是微小的消息变动，也会导致生成的摘要大幅不同，从而保障了数据的完整性。

2. **使用私钥签名：**发送方使用其私钥对消息摘要进行加密，生成数字签名。由于私钥是发送方独有的，这一步保证了签名的唯一性和不可否认性。

3. **附加数字签名到消息：**将生成的数字签名附加到原始消息上，一同发送给接收方。

4. **接收方验证签名：**接收方使用发送方的公钥对收到的数字签名进行解密，得到消息摘要。同时，接收方也使用相同的哈希函数对原始消息生成消息摘要。

5. **比对消息摘要：**接收方比较两个消息摘要是否一致。如果一致，说明消息未被篡改，并且确认了发送者的身份。



数字签名的性质：

- 能够验证签名者，签名日期和时间
- 能够认证被签名的消息内容

- 出现争执时能够被第三方仲裁

数字签名的要求：

- 签名必须依赖于正被签名的位模式
- 签名必须使用只有发送方知道的某些信息，以防止伪造和否认
- 生成数字签名比较容易
- 识别和验证数字签名比较容易
- 伪造数字签名在计算上是不可行的
- 保存数字签名的副本是可行的。

本项目中我使用私钥进行消息的签名，这不仅保证了消息的来源和完整性，还使得消息的身份验证和非抵赖性得到了保障。这种方法的采用，结合其他安全机制，为用户提供了一个安全可靠的通信平台。

3.6 消息认证

在本项目中，消息认证是确保通信安全的关键环节。通过使用公钥验证私钥签名的方法，我们能够确保消息的真实性和完整性。这种机制的实现主要基于非对称加密技术，以下是详细的说明：

A.消息认证的工作原理

1. **私钥签名**：发送方首先使用哈希函数对消息生成一个摘要（哈希值），然后使用自己的私钥对这个摘要进行加密，产生数字签名。这个数字签名随后会附加到原始消息上，一同发送给接收方。

2. **公钥验证**：接收方收到消息和数字签名后，首先使用发送方的公钥对数字签名进行解密，从而获取到消息摘要。接收方再对收到的原始消息应用相同的哈希函数，生成一个新的消息摘要。

3. **摘要比较**：接收方将从数字签名中解密得到的消息摘要与自己生成的消息摘要进行比较。如果两个摘要相同，那么说明消息在传输过程中未被篡改，且确实来自声称的发送方。

B.消息认证的关键特性

- **真实性**：通过公钥能够成功验证私钥签名，从而验证了消息的发送方确实是持有私钥的实体。这确保了消息的真实性。

- **完整性**：摘要比较可以确保消息自签名以来未被篡改。任何对消息内容的微小修改都会导致哈希值的巨大变化，从而在摘要比较时被发现。
- **不可抵赖性**：发送方一旦使用私钥对消息进行签名，便无法否认发送过该消息。因为只有发送方拥有私钥，所以能够用来生成该签名的只能是发送方本身。

在本项目的通信过程中，每条消息在发送前都会通过发送方的私钥进行签名。接收方在接收到消息后，使用发送方的公钥进行验证，以确保消息的安全性。这种基于公钥验证私钥签名的消息认证方法，不仅保障了消息内容的安全，也提升了整个通信系统的可信度。

3.7 单向口令

在本项目中，用户认证是一个关键的安全环节。为了加强认证过程的安全性，我们采用了单向口令技术，并结合了盐值（Salt）来提高系统的整体安全性。

单向口令技术是一种用户认证机制，它利用哈希函数的特性来确保口令的安全存储。在这种机制下，用户创建的口令不会以明文形式存储在任何地方。相反，当用户设置或输入口令时，系统会使用一个哈希函数将口令转换成一个哈希值，并将该哈希值存储在数据库中。由于哈希函数是单向的并且具有高度的碰撞阻力，即使数据库被未授权访问，攻击者也无法从哈希值逆推出原始的口令。

为了进一步提升安全性，我们在单向口令技术的基础上引入了盐值。盐值是一串随机生成的数据，它在哈希过程中与用户口令结合使用。将用户口令和盐值组合后再进行哈希运算，可以大幅增加破解难度。这是因为即使两个用户使用相同的口令，由于盐值的随机性，他们的哈希值将会不同，这样就大大减少了通过彩虹表等技术破解口令的可能性。

A.单向口令的存储流程：

1. **生成盐值**：当用户创建或修改口令时，系统会生成一个新的随机盐值。
2. **组合口令与盐值**：系统将用户的口令与生成的盐值在后台结合起来。
3. **生成哈希值**：应用哈希函数对组合后的数据进行处理，生成唯一的哈希值。
4. **存储哈希值和盐值**：将生成的哈希值和盐值一同存储在用户数据库中。值得注意的是，盐值是以明文形式存储的，因为其作用是为了在哈希过程中提供随

机性，而不是用来保密。

B.用户认证过程

1. **用户输入口令**：用户在登录时输入口令。

2. **检索盐值**：系统从数据库中检索与用户相关联的盐值。

3. **重复哈希过程**：系统将输入的口令与检索到的盐值结合，再次进行哈希运算。

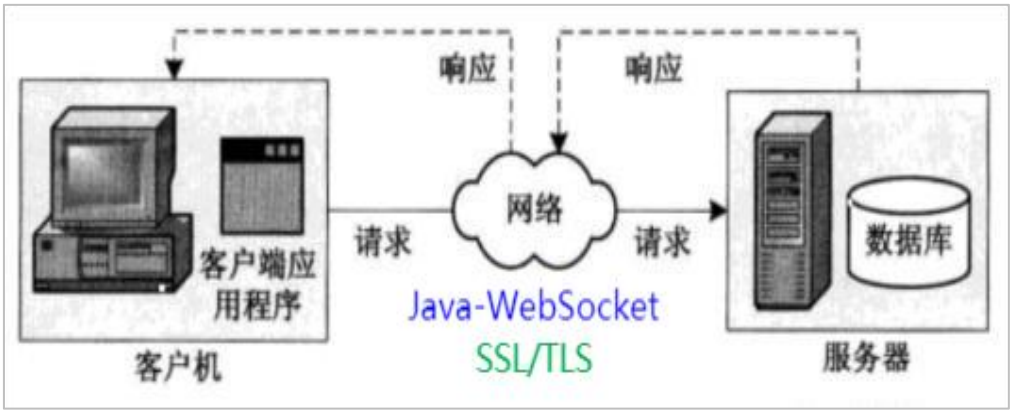
4. **验证哈希值**：系统比较计算出的哈希值与数据库中存储的哈希值是否一致。如果匹配，用户认证成功；否则，认证失败。

通过使用单向口令技术和盐值，我们确保了即使在数据库完整性被破坏的情况下，用户的口令仍然得到了保护。这种方法显著提升了用户数据的安全性，并为“小鳄鱼聊天室”项目的用户提供了一个更加安全的环境。

4 项目设计

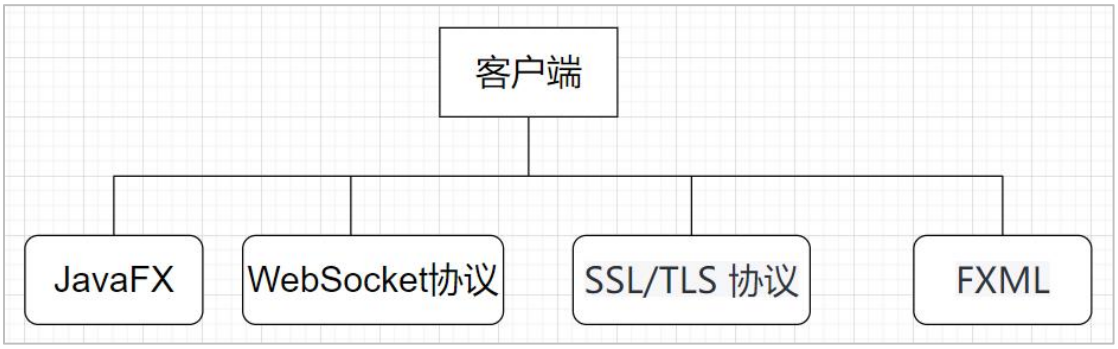
4.1 整体架构

本项目整体采用 C/S 架构，使用 Java-WebSocket 作为中间件进行客户端和服务端的连接。



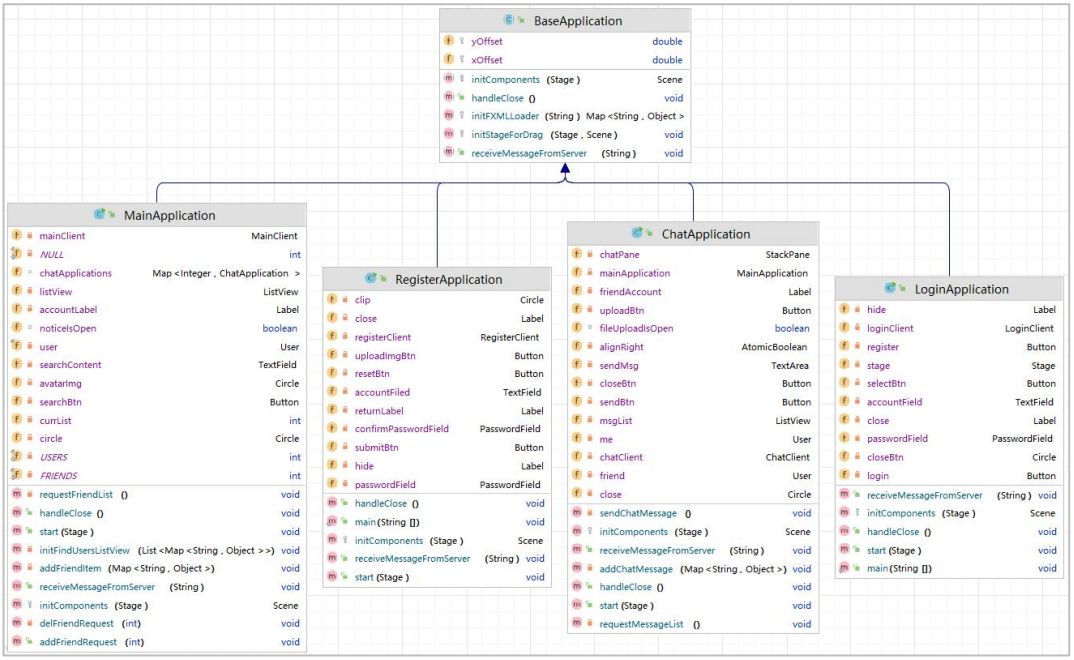
4.1.1 客户端架构

客户端技术选择如下：



1. 界面设计（Application）

泛化 JavaFX 的 Application 类，自定义一个 BaseApplication 抽象类，在里面定义一些适用于项目的方法。对于每一个程序的 UI 界面，均有一个 BaseApplication 的子类与之相对应，以下是 application 软件包下的 UML 类图：

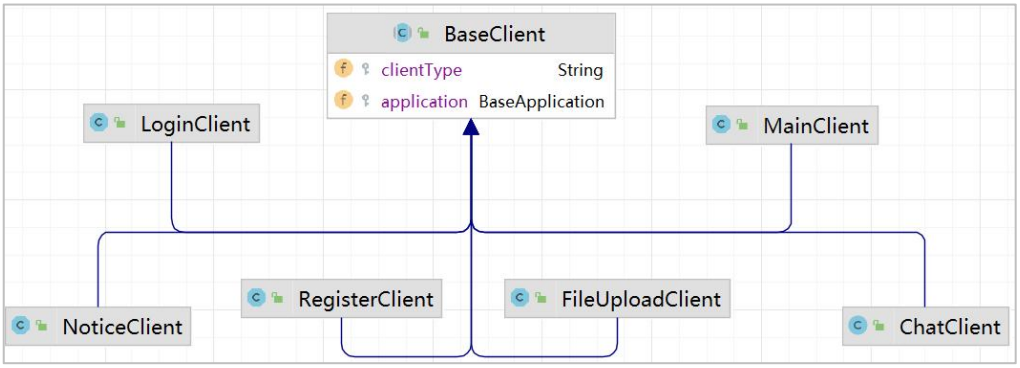


对于每一个 Application，均有一个 fxml 文件与之对应，Application 内部只需要加载该 fxml 文件即可生成 UI，我只需要在该类内部组件添加合适的监听器即可。

2. 通信设计（client）

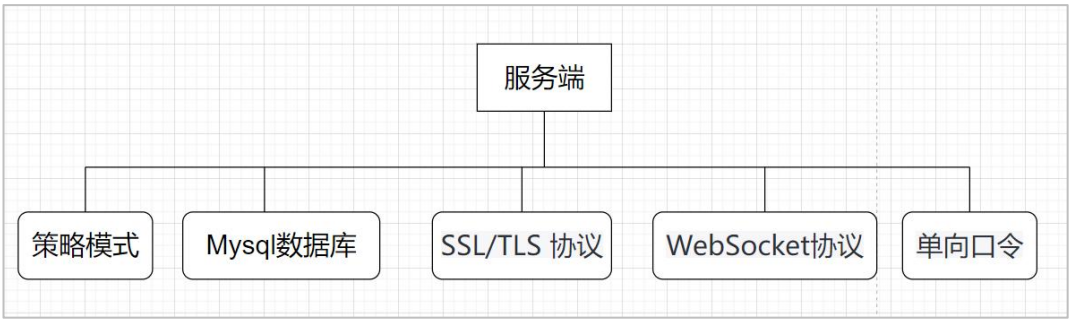
与 application 类似，泛化 WebSocketClient 类，自定义一个 BaseClient 抽象类，在里面实现最重要的 onMessage 方法，这个方法在接收到服务端发来的消息后，执行消息认证机制，然后立刻转发给 applicantion 下持有该类对象的 BaseApplication 的子类，子类作出相应的响应。

以下是 client 软件包下的 UML 类图：

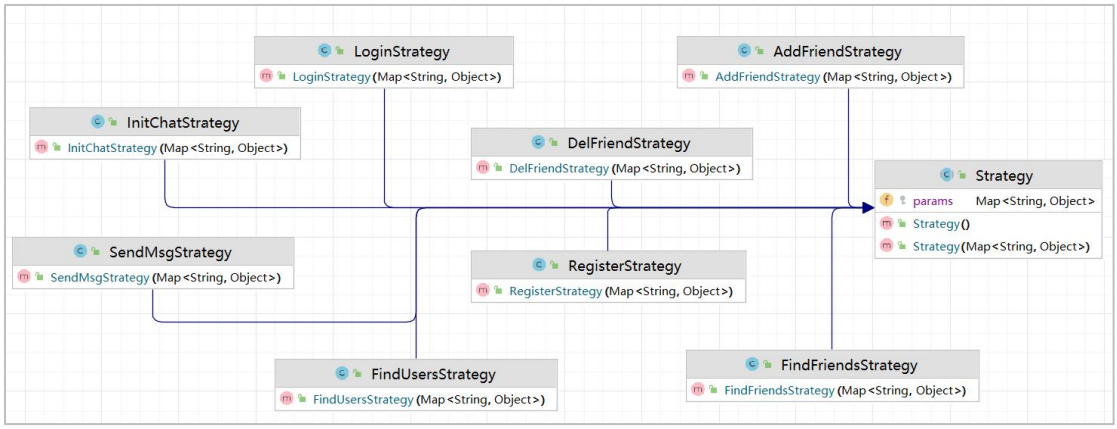


这样的设计使得客户端实现了松耦合，每一个 Client 子类均不需要自己实现具体方法（onOpen()、onClose()方法每个类在内部重载了，但是只做了打印操作，便于我观察程序的运行情况），这使得扩展客户端功能十分方便，主要的工作量在 application 软件包下。

4.1.2 服务端架构

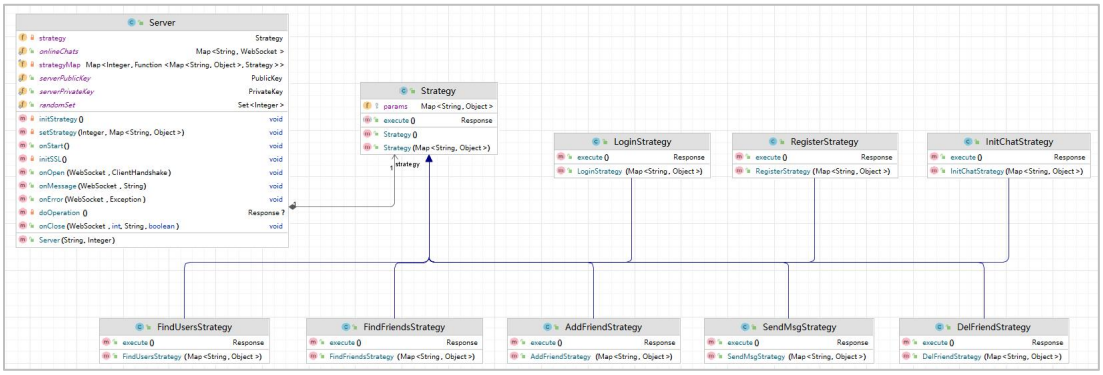


服务端主要采用策略模式，而对于每一个具体的策略类，都去调用 DAO 层访问数据库，并将结果封装进 Result 类里面返回。以下是 strategy 软件包下的 UML 类图：

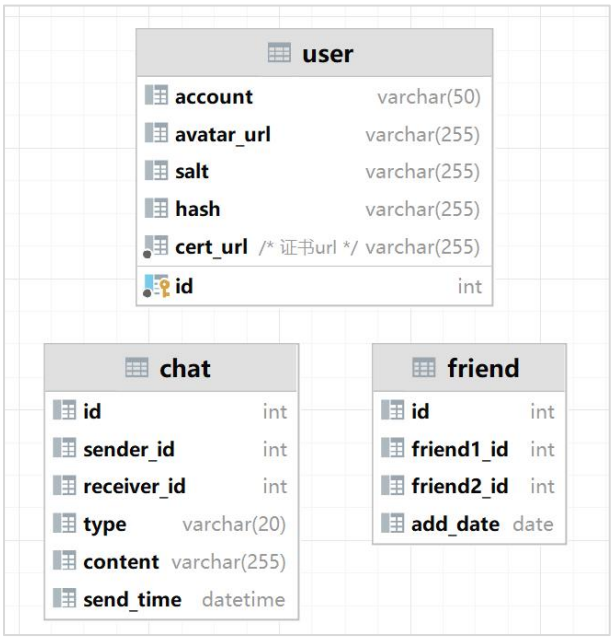


上述每一个子类均对应与一个客户端请求类型，Server 类作为上下文持有

Strategy 类的对象，根据请求类型执行不同的算法。Server 类图如下：



服务端数据库 E-R 图如下：



5 加密协议设计

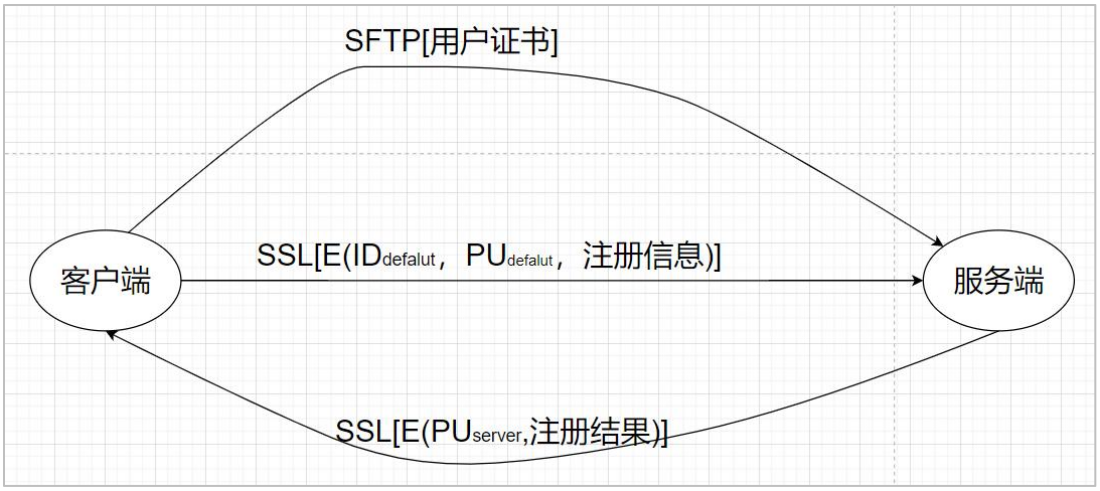
5.1 密钥分发设计

密钥分发是发生在用户注册过程中的，用户一旦注册完成，密钥即做好了分发。流程如下：

1. 生成一个包含公私钥对的密钥库文件（JKS）。
2. 使用 SFTP 将用户证书安全上传到服务器。
3. 利用客户端的默认密钥对中的私钥注册信息进行签名，并通过 SSL 发送到服务端。

4. 服务端用公钥验证信息，若认证通过，将信息存入数据库。

过程如图所示：



活动图描述如下：

5.2 协议过程

在通信协议过程中，客户端和服务端通过一系列安全措施确保消息的完整性、真实性和机密性，并防止重放攻击。这个过程大致如下：

1. **客户端开始流程：** 首先，客户端创建了一个包含随机数的请求消息。这个随机数的目的是确保每条消息都是唯一的，从而帮助防止重放攻击。

2. **客户端签名：** 客户端使用用户的私钥对这个请求消息进行数字签名。 然后，客户端将签名转换为 Base64 格式的字符串，并附加到原始消息的末尾。接着，整个消息（包括原始消息和签名）进行 Base64 编码。这个签名有助于服务端验证消息确实是由用户发出的，并且在传递过程中没有被篡改。

3. **加密与发送：**整个消息通过 SSL 协议加密后发送给服务端。这确保了消息在传输过程中的机密性。

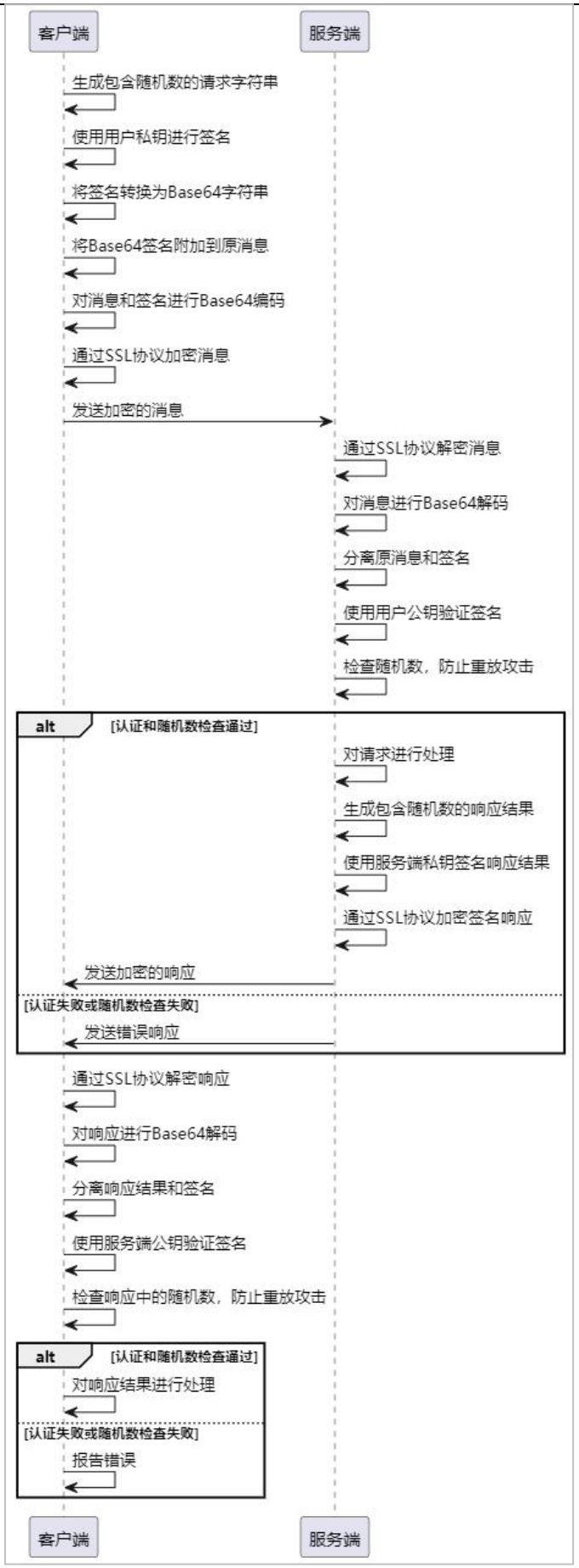
4. **服务端处理：** 服务端收到加密的消息后，先解密，然后对其进行 Base64 解码，从而恢复出原始的消息和签名。服务端使用客户端的公钥验证签名，同时检查消息中的随机数以确保这不是一条重放的消息。

5. **服务端响应：** 如果验证和随机数检查都通过，服务端处理请求并生成响应消息，这同样包含一个随机数。服务端使用自己的私钥对响应消息进行签名，然后通过 SSL 协议加密后发送给客户端。

6. **客户端接收响应：** 客户端收到加密的响应消息后，进行解密和 Base64 解码，然后使用服务端的公钥验证响应消息的签名，并检查响应中的随机数以确保这不是一条重放的响应。

7. **客户端响应：** 如果响应消息的签名和随机数检查都通过，客户端就会处理响应结果。如果任何一步检查失败，客户端会报告错误。

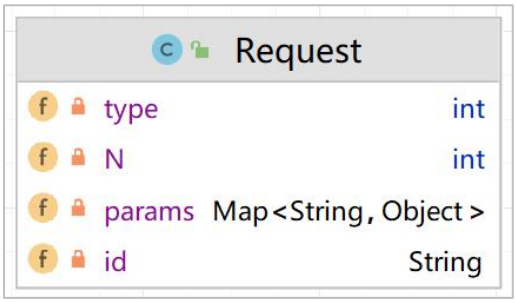
时序图描述如下：


客户端
服务端
生成包含随机数的请求字符串
使用用户私钥进行签名
将签名转换为Base64字符串
将Base64签名附加到原消息
对消息和签名进行Base64编码
通过SSL协议加密消息
发送加密的消息
通过SSL协议解密消息
对消息进行Base64解码
分离原消息和签名
使用用户公钥验证签名
检查随机数，防止重放攻击
alt
[认证和随机数检查通过]
对请求进行处理
生成包含随机数的响应结果
使用服务端私钥签名响应结果
通过SSL协议加密签名响应
发送加密的响应
[认证失败或随机数检查失败]
发送错误响应
通过SSL协议解密响应
对响应进行Base64解码
分离响应结果和签名
使用服务端公钥验证签名
检查响应中的随机数，防止重放攻击
alt
[认证和随机数检查通过]
对响应结果进行处理
[认证失败或随机数检查失败]
报告错误
客户端
服务端

5.3 报文设计

① 请求报文

请求报文由一请求体类对应，以下是该请求体的类图：



从而，通过数字签名后的报文格式如下：

id，type，N，params	间隙	数字签名

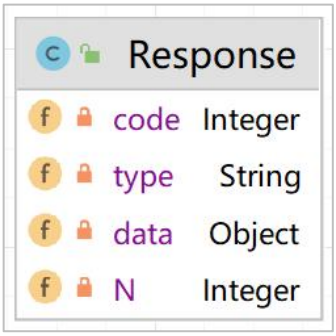
该报文还需通过 Base64 编码再经过服务器公钥加密后进行发送。

字段说明：

- id:标识用户公钥，便于服务端查找用户公钥，这里直接使用用户的用户名。
- type:标识请求类型。
- N:随机数，防止重放攻击。
- params:用户请求的参数。

② 响应报文

与请求报文类似，响应报文也由一响应体对应，类图如下：



从而，通过数字签名后的报文格式如下：

code，type，N，data	间隙	数字签名

该报文同意还需通过 Base64 编码再经过用户公钥加密后进行发送。

字段说明：

- code:服务端处理请求的结果，0 表示失败，1 表示成功。
- type:对应请求的类型。
- N:随机数，防止重放攻击。
- data:响应数据。

6 安全机制核心代码

6.1 密钥分发

在客户端的注册界面中，用户点击提交，客户端会自动化执行一系列操作：

```
boolean flag = KeyUtil.finishKeyFileOperation(account, password);
if (!flag){
   System.out.println("密钥文件操作失败");
   MsgDialog.showDialog(Alert.AlertType.ERROR, "密钥文件操作失败", "",false);
   return;
}else{
   String certPath = CONST.USER_FILE_PATH +account+".crt";
   try {
       FileUploader.sshSftpUploadCert(new File(certPath), account+".crt");
   } catch (Exception ex) {
       ex.printStackTrace();
       System.out.println("证书上传失败");
       MsgDialog.showDialog(Alert.AlertType.ERROR, "证书上传失败",
               "请检查网络连接或联系管理员",false);
       return;
   }
}
```

具体操作的代码如下：

```
public static boolean finishKeyFileOperation(String account,String
password){
   //生成密钥文件
   String
filename=generateKeyFile(account,password,password,"client-"+account,365
0);
   if (filename==null){
       return false;
   }
   System.out.println("filename:"+filename);
   //导入服务器证书
   boolean b = importCertificate(CONST.SERVER_CRT_PATH, filename, password,
"server");
   if (!b){
       return false;
   }
   //导出用户证书
   try {
       boolean b1 = exportCert(filename, "client-"+account, password,
CONST.USER_FILE_PATH +account+".crt");
       return b1;
   } catch (Exception e) {
       e.printStackTrace();
       return false;
   }
}
```

- 生成密钥文件

```
private static String generateKeyFile(String account,String
keystorePassword ,String keyPassword ,String alias,int validDate){
```

```
    String keystoreFileName = CONST.USER_FILE_PATH +account + ".jks";
    String command = String.format(
            "keytool -genkeypair -alias %s -keyalg RSA -keysize 2048
-validity %d -keystore %s " +
                    "-storepass %s -keypass %s -dname " +
                    "\"CN=%s, OU=MyUnit, O=MyOrganization, L=MyCity, S=MyState,
C=MyCountry\"",
            alias, validDate, keystoreFileName, keystorePassword, keyPassword,
alias);
    try {
        Process process=Runtime.getRuntime().exec(command);
        process.waitFor();
        if (process.exitValue()==0){
            System.out.println("generate succeed");
            return keystoreFileName;
        }else{
            System.err.println("fail");
            return null;
        }
    }catch (IOException|InterruptedException e){
        e.printStackTrace();
        return null;
    }
}
```

- 导入服务器证书

```
private static boolean importCertificate(String certPath, String
keystorePath, String keystorePassword, String alias) {
    try {
        // 构建 keytool 导入证书的命令
```

```
        String command = "keytool -import -file " + certPath +
                " -alias " + alias +
                " -keystore " + keystorePath +
                " -storepass " + keystorePassword
                + " -noprompt"; // 添加 -noprompt 选项来避免手动确认
        // 执行命令
        Process process = Runtime.getRuntime().exec(command);


        process.waitFor();
        int exitValue = process.exitValue();
        if (exitValue == 0) {
            System.out.println("Certificate was imported successfully.");
            return true;
        } else {
            System.out.println("There was an error importing the certificate.");
            return false;
        }
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
        return false;
    }
}
```

- 导出用户证书

```
private static boolean exportCert(String keystorePath, String alias, String
keystorePassword, String exportPath) throws Exception {
    FileInputStream is = new FileInputStream(keystorePath);
    KeyStore keystore = KeyStore.getInstance("JKS");
    keystore.load(is, keystorePassword.toCharArray());
    Certificate cert = keystore.getCertificate(alias);
```

```
    if (cert == null) {
        System.out.println("No certificate found for alias: " + alias);
        return false;
    }
    byte[] certBytes = cert.getEncoded();
    try{
        FileOutputStream fos = new FileOutputStream(exportPath);
        fos.write(certBytes);
        fos.close();
        return true;
    }catch (Exception e){
        e.printStackTrace();
        return false;
    }finally {
        is.close();
    }
}
```

- 上传用户证书方法

```
public static void sshSftpUploadCert(File file, String fileName) throws
Exception{
    Session session = null;
    Channel channel = null;
    JSch jsch = new JSch();
    if(CONST.UPLOAD_IMG_PORT <=0){
        //连接服务器，采用默认端口
        session = jsch.getSession(CONST.UPLOAD_IMG_USERNAME,
CONST.UPLOAD_IMG_URL);
    }else{
        //采用指定的端口连接服务器
```

```
        session = jsch.getSession(CONST.UPLOAD_IMG_USERNAME,
CONST.UPLOAD_IMG_URL ,CONST.UPLOAD_IMG_PORT);
    }
    //如果服务器连接不上，则抛出异常
    if (session == null) {
        throw new Exception("session is null");
    }
    //设置登陆主机的密码
    session.setPassword(CONST.UPLOAD_IMG_PASSWORD);//设置密码
    //设置第一次登陆的时候提示，可选值：(ask | yes | no)
    session.setConfig("StrictHostKeyChecking", "no");
    //设置登陆超时时间
    session.connect(30000);
    OutputStream outstream = null;
    try {
        //创建 sftp 通信通道
        channel = (Channel) session.openChannel("sftp");
        channel.connect(1000);
        ChannelSftp sftp = (ChannelSftp) channel;
        //进入服务器指定的文件夹
        sftp.cd(CONST.UPLOAD_CERT_PATH);
        //以下代码实现从本地上传一个文件到服务器，如果要实现下载，对换以下流就可以了
        outstream = sftp.put(fileName);
        outstream.write(fileToBytesArray(file));
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //关流操作
        if(outstream != null){
```

```
            outstream.flush();
            outstream.close();
        }
        if(session != null){
            session.disconnect();
        }
        if(channel != null){
            channel.disconnect();
        }
    }
}
```

6.2 单向口令

- 注册时生成盐值、计算哈希值并存储

```
public Response execute() {
    System.out.println("RegisterStrategy execute");
    String account= (String) params.get("account");
    String password= (String) params.get("password");
    String salt = Security.generateSalt(CONST.SALT_LENGTH);
    String hash=Security.hash(password,salt);
    if (hash==null){
        return new Response(0,0,null,null);
    }
    boolean res = UserDAO.register(account, hash, salt, (String)
params.get("avatarName"),(String) params.get("certName"));
    if(res){
        return new Response(0,1,null,null);
    }
```

```
    return new Response(0,0,null,null);
}
```

- 登录时将密码与盐值计算后与正确的哈希值进行对比

```
public Response execute() {
    System.out.println("LoginStrategy execute");
    User user = UserDAO.login((String)params.get("account"));
    if(user==null) return new Response(0,0,null,null);
    String password = (String)params.get("password");
    String slat= user.getSalt();
    String hash=user.getHash();
    String inputHash= Security.hash(password, slat);
    System.out.println(inputHash+","+hash);
    if(!hash.equals(inputHash))return new Response(0,0,null,null);
    user.setHash(null);
    user.setSalt(null);
    Map<String,Object> data=new HashMap<>();
    data.put("id",user.getId());
    data.put("account",user.getAccount());
    data.put("avatarUrl",user.getAvatarUrl());
    System.out.println(data);
    return new Response(0,1,null,data);
}
```

6.3 数字签名

- 客户端

```
public static String sign(String data, PrivateKey key) {
    try {
        // 创建签名实例并初始化
        Signature signature = Signature.getInstance("SHA256withRSA");
```

```
        signature.initSign(key);
        // 对数据进行签名
        signature.update(data.getBytes());
        byte[] signedData = signature.sign();
        // 将签名后的数据转换为Base64 字符串
        String signedDataBase64 =
Base64.getEncoder().encodeToString(signedData);
        // 将源数据和签名数据拼接为一个字符串
        String signed=data+SIGN_SPLIT+signedDataBase64;
        // 最后再次对整个字符串进行Base64 编码
        return Base64.getEncoder().encodeToString(signed.getBytes());
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

- 服务端

```
public static String sign(String data, PrivateKey key){
    try {
        Signature signature=Signature.getInstance("SHA256withRSA");
        signature.initSign(key);
        signature.update(data.getBytes());
        byte[] signed=signature.sign();
        String signedBase64= Base64.getEncoder().encodeToString(signed);
        String signedData=data+SIGN_SPLIT+signedBase64;
        return Base64.getEncoder().encodeToString(signedData.getBytes());
    } catch (Exception e) {
        e.printStackTrace();
        return null;
```

```
    }
}
```

6.3 消息认证

- 客户端

```
public static String verify(String fullMessageBase64, PublicKey publicKey)
{
    try {
        // 首先对整个消息进行Base64 解码
        String fullMessage = new
String(Base64.getDecoder().decode(fullMessageBase64));
        // 分离数据和签名
        int indexOfSignSplit = fullMessage.lastIndexOf(SIGN_SPLIT);
        if (indexOfSignSplit == -1) {
            // 如果没有找到分隔符，无法分离数据和签名
            return null;
        }
        String data = fullMessage.substring(0, indexOfSignSplit);
        String sign = fullMessage.substring(indexOfSignSplit +
SIGN_SPLIT.length());
        // 对签名进行Base64 解码
        byte[] signBytes = Base64.getDecoder().decode(sign);
        // 创建签名实例并用公钥初始化
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initVerify(publicKey);
        // 对数据进行更新
        signature.update(data.getBytes());
        // 验证签名
        boolean verified = signature.verify(signBytes);
```

```
        // 如果验证成功，返回原始数据，否则返回null
        return verified ? data : null;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

- 服务端

```
public static String verify(String fullMessageBase64){
    try {
        String fullMessage=new
String(Base64.getDecoder().decode(fullMessageBase64));
        // 分离数据和签名
        int indexOfSignSplit = fullMessage.lastIndexOf(SIGN_SPLIT);
        if (indexOfSignSplit == -1) {
            // 如果没有找到分隔符，无法分离数据和签名
            return null;
        }
        String data = fullMessage.substring(0, indexOfSignSplit);
        System.out.println("data="+data);
        String sign = fullMessage.substring(indexOfSignSplit +
SIGN_SPLIT.length());
        System.out.println("sign="+sign);
        // 从数据中获取公钥ID
        String publicKeyId = data.substring(data.indexOf("id=") + 3,
data.indexOf("&"));
        PublicKey publicKey = getPublicKey(publicKeyId);
        System.out.println("publicKeyId="+publicKeyId);
        System.out.println("publicKey="+publicKey);
```

```
        if (publicKey == null) {
            return null;
        }
        // 对签名进行Base64 解码
        byte[] signBytes = Base64.getDecoder().decode(sign);
        // 创建签名实例并用公钥初始化
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initVerify(publicKey);
        // 对数据进行更新
        signature.update(data.getBytes());
        // 验证签名
        boolean verified = signature.verify(signBytes);
        // 如果验证成功，返回原始数据，否则返回null
        return verified ? data : null;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

6.4 随机数挑战应答

- 客户端

客户端每次在发送消息之前生成一个随机数加入到请求体中，例如：

```
Request request=new Request();
request.setType(REGISTER);
request.setId(CLIENT_DEFAULT_ID);
request.setN(new SecureRandom().nextInt(1000000000));
Map<String, Object> params=new HashMap<>();
params.put("account",account);
```

```
params.put("password",password);
params.put("avatarName",avatarName);
params.put("certName",account+".crt");
request.setParams(params);
System.out.println(request.toString());
String signed=SignUtil.sign(request.toString(), ChatApp.defaultPrivateKey);
registerClient.send(signed);
```

客户端维护一个随机数集合，记录已经出现的随机数，当收到的响应中存在一个客户端遇到过的随机数，则不处理该响应并报告

```
int N= Integer.parseInt((String) Parse.parseResponse(verify).get("N"));
if (ChatApp.randomSet.contains(N)){
   System.out.println("重复消息");
   MsgDialog.showDialog(Alert.AlertType.ERROR,"重复消息","",false);
   return;
}else{
   ChatApp.randomSet.add(N);
}
```

- 服务端

服务端操作与客户端类似

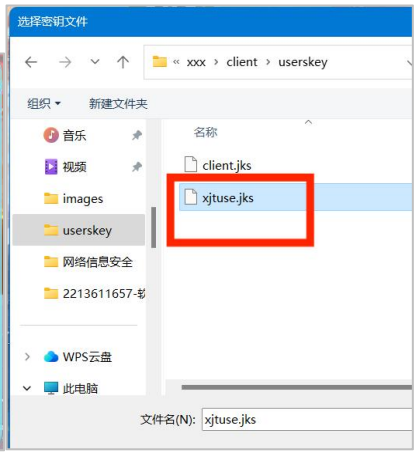
```
int N=Integer.parseInt((String) request.get("N"));
int random=new SecureRandom().nextInt(1000000000);
Response response=null;
if (randomSet.contains(N)){
   System.err.println("客户端请求随机数重复！");
   return;
}else{
   randomSet.add(N);
   int type= Integer.parseInt((String) request.get("type"));
   System.out.println(request);
```

```
    setStrategy(type, (Map<String, Object>) request.get("params"));
    response=doOperation();
    response.setType(String.valueOf(type));
}
```

7 运行结果展示

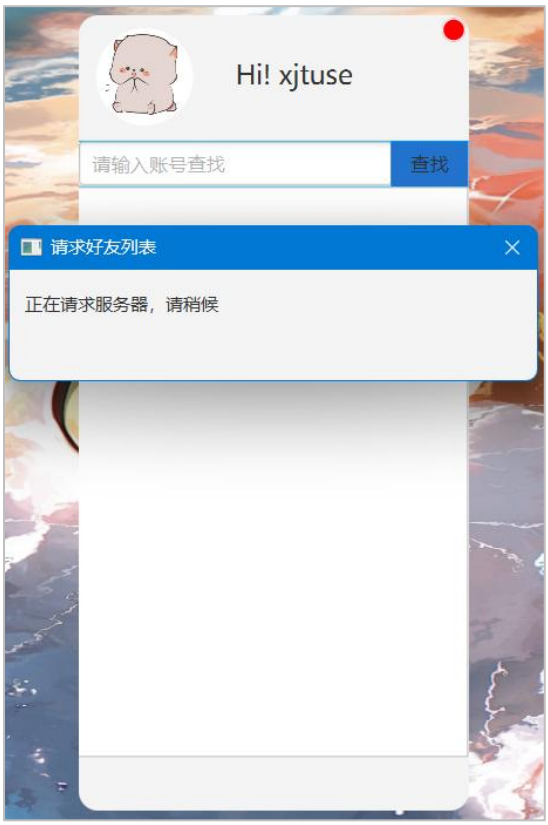
- 注册一个用户

- 登录



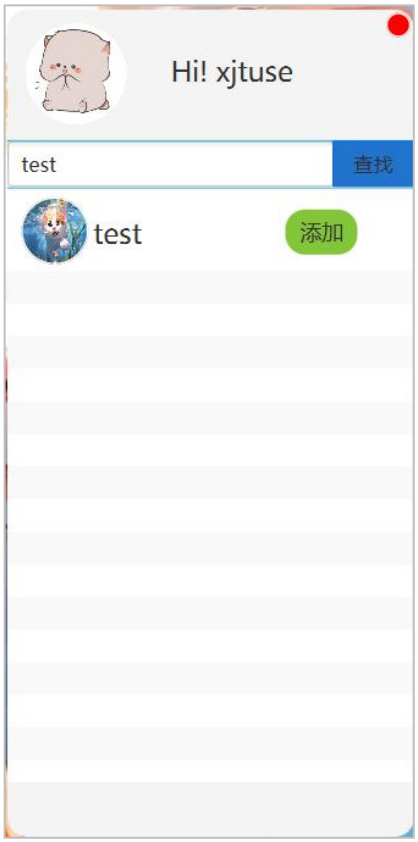
（选择个人密钥文件）

- 登录成功

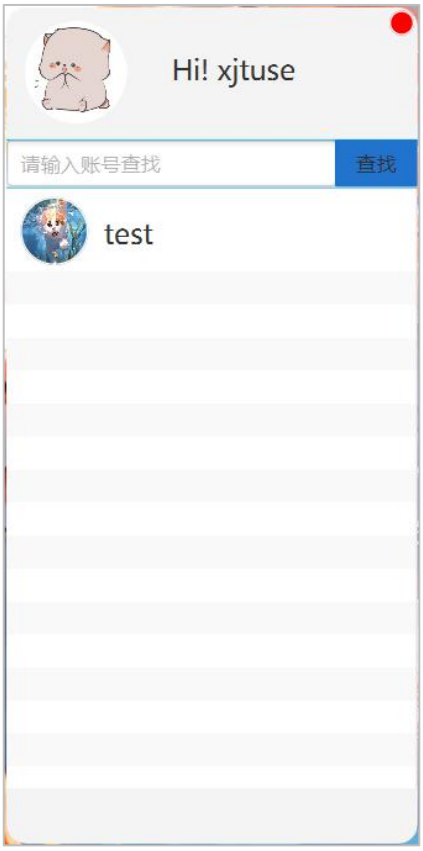


- 再注册一个用户：

- 然后在 xjtuse 用户中查找该用户

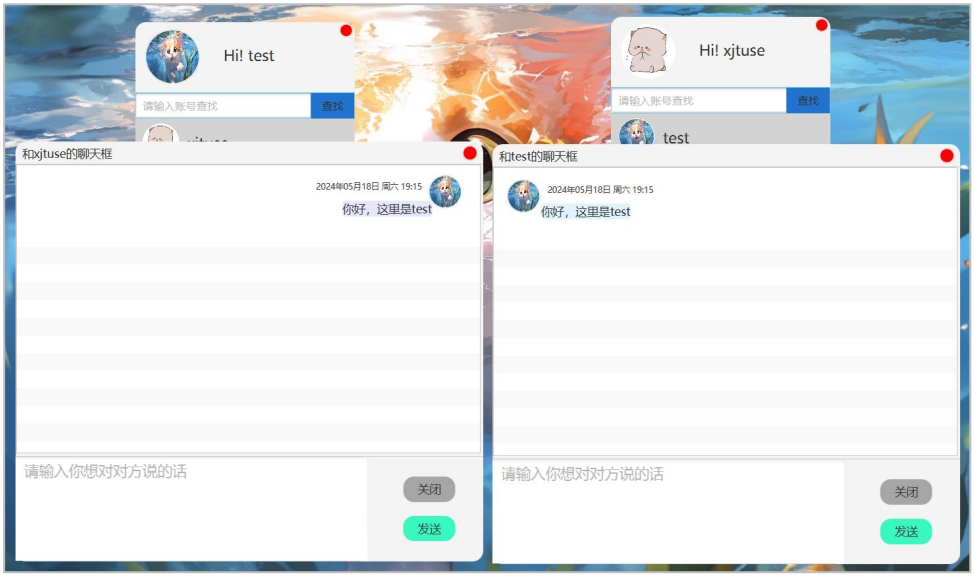


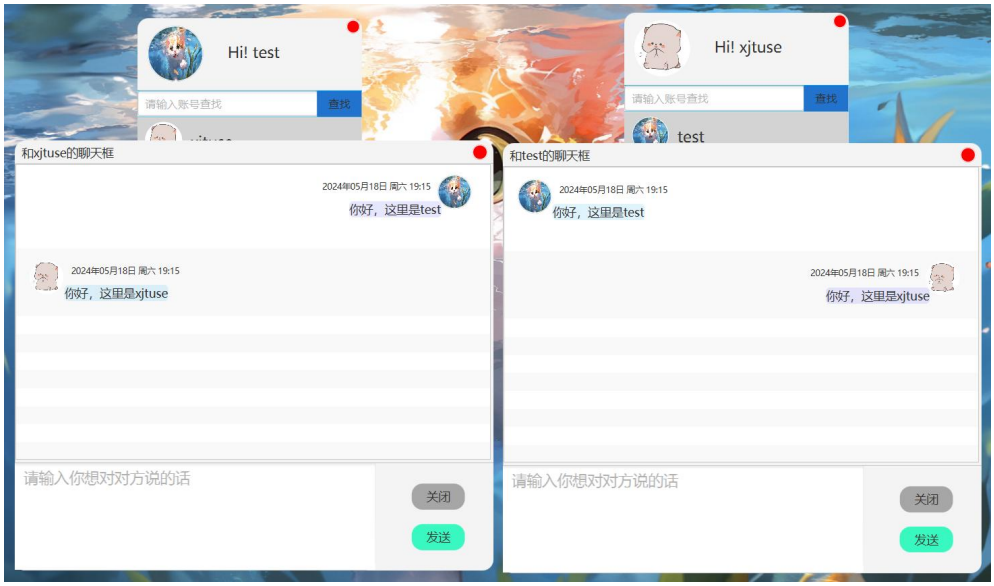
• 点击添加后，在两个用户界面中可以看到：



- 进入这两个用户的聊天窗口

- 互相发送消息





7 遇到的问题

1. 在项目设计初期，只是一股脑的在服务端接收客户端的消息后处理进行回复，后面发现消息类型多种多样，有登录请求、注册请求、添加（或删除）好友请求、对话请求等，这样会导致很长的 switch-case 语句出现，导致 server 端代码过长，不易于维护与扩展，后面受 web 项目中常用的 springboot 框架中的 controller 层的启发，web 前端向后端发送的消息是通过路径来进行定位到不同的 controller；而对于 C/S 架构，简化来看，server 端只是对 client 端发送的消息作出处理，并将处理后的结果数据返回给 client 端，而不同的消息类型对于与 server 端不同的处理操作，这类似于操作系统中的系统调用，用户程序调用系统调用陷入操作系统中，事先会将调用的系统调用号和调用参数存入寄存器中，内核程序读取这些寄存器然后执行，这种策略模式极大减小了 C/S 架构的代码耦合度，于是我便在我的项目中加入该设计模式，这也有利于我后期的开发与维护。
2. 在项目初期，我不知道消息报文的格式该如何设计比较好，后面我参考了浏览器中地址栏参数是采用‘&’符号进行分隔，于是我也这样设计，难点是如何解析这样格式的报文，特别是如何解析出 list（比如当请求好友列表时，必须使用数组或者 list 实现），后面我参考以前做过的 web 项目，设计一个从服务端向客户端返回的返回体，解决了这个问题。同样，在客户端也设计了一个请求体向服务端发送请求，这样有利于后期的维护。
3. 一开始我使用的是 Socket 类，并且在开发的初期，并未关闭 socket 的 io 流，在项目基本功能完成的时候，程序非常卡顿，后面逐步关闭 io 流发现程序无端报错，出现 IOException，后续一步一步调试后将报错解决，依旧发现程序运行卡顿，客户端和服务端之间的通信特别慢，怀疑是每通信一次都需要有 IO 流的打开与关闭，非常耗费计算机资源，于是改用 Java-WebSocket 进行项目的重构，好在服务端一开始被设计的耦合度比较小，重构起来没有遇到设计模式上面的困难。
4. JavaFX 在设计静态（不会随着程序的运行而改变样式）的 UI 上是非常方便的，但是在设计会随着程序运行而改变样式的 UI 时（比如聊天对话框自适

应聊天文本），需要使用 Controller 类在代码里面手动改变 UI 样式，这花费了我一个晚上查阅相关资料和调试代码。

5. 我最初不是使用的 SSL/TLS 协议，而是手动在代码中加入消息的加密和解密，这样要在代码中修改很多地方，不利于维护。后面使用 SSL/TLS 协议之后，代码改动小，但是在最初编写加密通信的测试小程序时，由于我对 SSL/TLS 协议的不熟悉，导致程序一直无法实现握手成功，后续花费一天的时间才解决。

6. 当测试多个用户同时使用客户端时，出现了线程之间的竞争，导致我服务端中出现资源竞争，特别是对 map 的资源竞争，后面我使用了 Collections.synchronizedMap()方法获取到了增删改查操作是原子操作的 map；此外，我还加入了 synchronized 语句，实现了 map 的互斥访问，保障了服务端多线程程序的正确执行。

7. 项目中并没有完成用户状态的表示，比如用户登录后，该用户的好友应该可以看到该用户上线了，离线后改用户的好友应该可以看到该用户处于离线状态。由于这个功能并不是主要功能，碍于时间原因，并未在项目中实现。

8. 登录时用户需要选择密钥库文件，但是这样的操作在某种程度上是不友好的。比如，一个用户登录之后，未防止其余用户在同一台电脑登录时查看到本人的密钥库文件，需要提取将文件拷贝走，这样的操作不是很方便。或者说，本软件的应用场景应该就是一个用户使用一台设备，而不是多个用户使用一台。

9. 由于我只有一台物理机（即使有一台阿里云服务器，但是该服务器的性能无法支持我使用 idea 进行远程开发），即使将用户证书上传至服务器上后，服务端使用的用户证书也是保存在本地的，也就是说，本项目中的密钥分发只是做了一个模拟，或者说是假想，并没有完完全全的实现。

10. 项目并未考虑到大量用户的情况，对此情况，我打算加入队列来优化程序，但是学业繁忙，时间有限，并未落实。

8 总结与展望

8.1 项目亮点

1. 本项目完成了一个聊天软件的基本功能，能够实现登录、注册、聊天、

好友管理这些核心的功能，效果良好。

2. 程序中有大量的异常处理语句，同样也会使用弹窗的方式反馈给用户，这提高了程序的鲁棒性。

3. 软件的架构设计耦合度低，代码可读性高，利于后续维护和功能的扩展。

4. 满足关键性的安全概念：

(1) 机密性

在项目中我在 WebSocket 协议中加入了 SSL/TLS 协议，使用公钥加密，私钥解密的方式保证了消息传输过程中的机密性，并且消息在发送之前进行了两次的 Base64 编码，这进一步提高了抵御攻击的能力。

还有一点就是密钥分发，在密钥分发的时候我使用了客户端的默认密钥进行消息的传输，以及用户证书是使用 sftp 进行发送（即使证书不需要保密），这保证了密钥分发过程是安全的。

(2) 完整性

在消息的发送之前，应用程序先对消息进行了数字签名，这使得一旦发送途中被篡改，该消息将无法通过接收方的验证，这样一来，完整性得到了保证。

(3) 真实性与可审计性

在消息发送之前，应用程序先对消息使用自己的私钥进行了数字签名，这使得这条消息只能由发送方发出，无法由别人发出，并且发送方不可抵赖。

此外，通过单向口令，即使数据库泄露，攻击者也无法获取用户的登录密码，这也提供了一定的安全性。

8.2 项目不足

1. 软件 UI 设计不够完美，动画过渡地不够流畅，字体和 UI 样式可能存在一些不和谐，而且 UI 主要是模仿腾讯 QQ，没有创新性。并且，虽然相比于 JavaSwing，JavaFX 的 UI 设计更加方便，但是 JavaFX 的性能不如 JavaSwing。

2. 软件并未考虑到大量请求的情况，并且由于设备有限，不太容易模拟出大量请求的测试。

8.3 展望

1. 改进 UI 设计

采用现代化的 UI 设计理念，使用流畅的动画效果，和谐的色彩搭配以及直观的用户交互设计。

2. 考虑大量请求的情况

优化服务器端的架构，消息队列来缓冲请求，使用负载均衡器分发用户请求到不同的服务器实例，以及利用缓存策略减轻数据库的压力。

3. 增加发送表情包的功能

在客户端 UI 中集成表情包选择器，允许用户快速挑选和发送表情。服务器端需要处理和传输表情数据，确保表情包在不同客户端间正确显示。

4. 增加文件上传功能

为了增强用户体验并提升软件的实用性，计划在后续版本中增加文件上传功能。允许用户在聊天中上传和发送各种类型的文件，比如文档、图片、音频和视频等，用户可以从本地选择文件，然后上传到服务器，接收方可以下载并查看这些文件。

5. 增加群聊功能

设计群聊的数据模型，实现群管理功能（如创建群、加入群、退出群等），以及优化消息广播机制以支持大量用户同时在线的场景。

6. 增加消息提示功能

实现桌面通知功能，当应用程序在后台运行时，新消息到来可以通过操作系统的通知中心提示用户。

7. 开发适配安卓系统的客户端

开发适用于 Android 设备的客户端应用，确保与现有的服务端兼容，并提供与桌面版相似的功能和用户体验。

8. 提高性能

进行代码审核和性能测试，优化算法和数据结构，减少资源消耗，以及考虑使用更高效的 C++ QT 框架对软件进行重写。