计算机网络安全与管理

安全通信软件 safechat的设计

姓名	易俊泉
班级	软件92班
学号	2194411245
电话	18813517223
Email	1302190004@stu.xjtu.edu.cn
日期	2022-6-10

目录

安全通信软件 safechat 的设计

1 设计要求

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：

1、 独立完成

2、 具有完整的流程设计，报文格式等相关分析。

3、 具备自圆其说的安全性设计思考

2 设计分工

本次的课程由我和软件 92 班的徐礼祯同学共同设计完成。分工如下：

易俊泉：前端架构设计、WebSocket 客户端设计、消息发送、头像上传、密钥分配、公钥加密实现消息认证

徐礼祯：后端架构设计、WebSocket 服务端设计、注册登录、用户信息获取、AES 加密、Token 加密

3 设计原理

3.1 SHA-2

SHA-2，名称来自于安全散列算法 2（英语：Secure Hash Algorithm 2）的缩写，一种密码散列函数算法标准，由美国国家安全局研发[3]，由美国国家标准与技术研究院（NIST）在 2001 年发布。属于 SHA 算法之一，是 SHA-1 的后继者。其下又可再分为六个不同的算法标准，包括了：SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256。

3.2 RSA

RSA 加密算法是一种非对称加密算法，在公开密钥加密和电子商业中被广泛使用。RSA 是由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）在 1977 年一起提出的。当时他们三人都在麻省理工学院工作。RSA 就是他们三人姓氏开头字母拼在一起组成的。

对**极大整数做因数分解**的难度决定了 RSA 算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA 算法愈可靠。假如有人找到一种快速因数分解的算法的话，那么用 RSA 加密的信息的可靠性就会极度下降。但找到这样的算法的可能性是非常小的。今天只有短的 RSA 钥匙才可能被强力方式破解。到 2020 年为止，世界上还没有任何可靠的攻击 RSA 算法的方式。只要其钥匙的长度足够长，用 RSA 加密的信息实际上是不能被破解的。

用户通过如下过程生成密钥对

1、 选择两个随机的大素数p，q

2、 计算它们的乘积（系统的模）$n = p \times q$

3、 随机选取**加密密钥 e**：注意欧拉函数值$\phi(n) = (p-1)(q-1)$

4、 解下面的等式获得**解密密钥 d**

$$e.d = 1\ mod\ \phi(n)\ and\ \ 0 \le d \le n$$

5、 公开其公钥：$PU = \{e，n\}$，保留私钥$PR = \{d，n\}$

RSA 的使用如下：

1、 加密一条消息 M，发送方需要：获取公钥$PU = \{e,n\}$；计算$C = M^e\ mod\ n, where\ 0 \le M < n$

2、 解密 C，接收方需要：利用私钥$PR = \{d,n\}$；计算$M = C^d\ mod\ n$

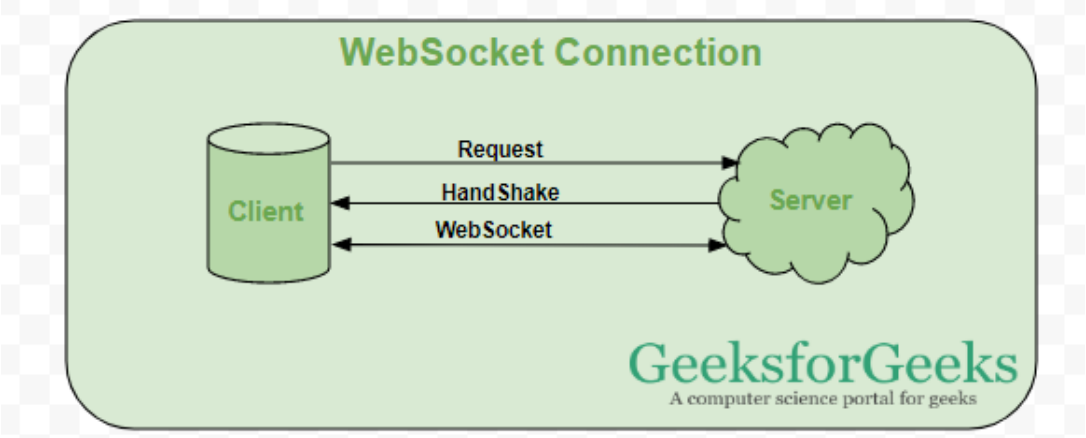
3、 必要的时候需要进行分块

3.3 WebSocket 协议

WebSocket 是双向的，在客户端-服务器通信的场景中使用的全双工协议，与 HTTP 不同，它以 ws://或 wss://开头。它是一个有状态协议，这意味着客户端和服务器之间的连接将保持活动状态，直到被任何一方（客户端或服务器）终止。在通过客户端和服务器中的任何一方关闭连接之后，连接将从两端终止。

以客户端-服务器通信为例，每当启动客户端和服务器之间的连接时，客户端-服务器进行握手随后创建一个新的连接，该连接将保持活动状态，直到被他们中的任何一方终止。建立连接并保持活动状态后，客户端和服务器将使用相同的连接通道进行通信，直到连接终止。

新建的连接被称为 WebSocket。一旦通信链接建立和连接打开后，消息交换将以双向模式进行，客户端-服务器之间的连接会持续存在。如果其中任何一方（客户端服务器）宕掉或主动关闭连接，则双方均将关闭连接。套接字的工作方式与 HTTP 的工作方式略有不同，状态代码 101 表示 WebSocket 中的交换协议。



图片 1WebSocket 连接示意

3.4 JWT

JWT 就是通过 JSON 形式作为 Web 应用中的令牌，用于在各方之间安全地将信息作为 JSON 对象传输。在数据传输过程中还可以完成数据加密，签名等相关处理。

3.4.1 基于 JWT 认证

1、首先，前端通过 Wb 表单将自己的用戶名和密码发送到后端的接口。这一过程一般是一个 HTTP POST 请求。

2、后端核对用戶名和密码成功后，将用戶的 id 等其他信息作为 JWT Payload(负载)，将其与头部分别进行 Base64 编码拼接后签名，形成一个 JWT(Token)。形成的 JWT 就是一个形同 11.Zzz.xx 的字符串。token head.payload.signature

3、后端将 JWT 字符串作为登录成功的返回结果返回给前端。前端可以将返回的结果保存在 localStorage 或 sessionStorage.上，退出登录时前端删除保存的 JWT 即可。

4、前端在每次请求时将 JWT 放入 HTTP Header 中的 Authorization 位。（解决 XSS 和 XSRF 问题）

5、后端检查 JWT 是否存在，如存在验证 JWT 的有效性。检查签名是否正确，检查 Token 是否过期，检查 Token 的接收方是否是自己（可选）

3.4.2 JWT 结构

jwt 生成的字符串包含有三部分

1、 **jwt 头信息部分 header**：标头通常由两部分组成：令牌的类型（即 JWT 所使用的签名算法，例如 HMAC、SHA256 或 RSA。它会使用 Base64 编码[1]组成 JWT 结构的第一部分。

2、 **在效载荷 Payload**：令牌的第二部分是有效负载，其中包含声明。声明是有关实体（通常是用戶）和其他数据的声明。同样的，它会使用 Ba$64 编码组成 JWT 结构的第二部分

3、 **签名哈希 Signature**：header 和 payload 都是结果 Base64 编码过的，中间用.隔开，第三部分就是前面两部分合起来做签名，密钥绝对自己保管好，签名值同样做 Base64 编码拼接在 JWT 后面。（签名并编码）

3.5 AES

高级加密标准（英语：Advanced Encryption Standard，缩写：AES），又称 Rijndael 加密法（荷兰语发音： [ˈrɛinda:l]，音似英文的“Rhine doll”），是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的 DES，已经被多方分析且广为全世界所使用。经过五年的甄选流程，高级加密标准由美国国家标准与技术研究院（NIST）于 2001 年 11 月 26 日发布于 FIPS PUB 197，并在 2002 年 5 月 26 日成为有效的标准。现在，高级加密标准已然成为对称密钥加密中最流行的算法之一。

严格地说，AES 和 Rijndael 加密法并不完全一样（虽然在实际应用中两者可以互换），因为 Rijndael 加密法可以支持更大范围的区块和密钥长度：AES 的区块长度固定为 128 比特，密钥长度则可以是 128，192 或 256 比特；而 Rijndael 使用的密钥和区块长度均可以是 128，192 或 256 比特。加密过程中使用的密钥是由 Rijndael 密钥生成方案产生。

大多数 AES 计算是在一个特别的有限域完成的。

AES 加密过程是在一个 4×4 的字节矩阵上运作，这个矩阵又称为“体（state）"，其初值就是一个明文区块（矩阵中一个元素大小就是明文区块中的一个 Byte）。（Rijndael 加密法因支持更大的区块，其矩阵的“列数（Row number）”可视情况增加）加密时，各轮 AES 加

[1] Base64 是一种编码，可以被翻译回原来的样子，并不是一种加密过程。

密循环（除最后一轮外）均包含 4 个步骤：

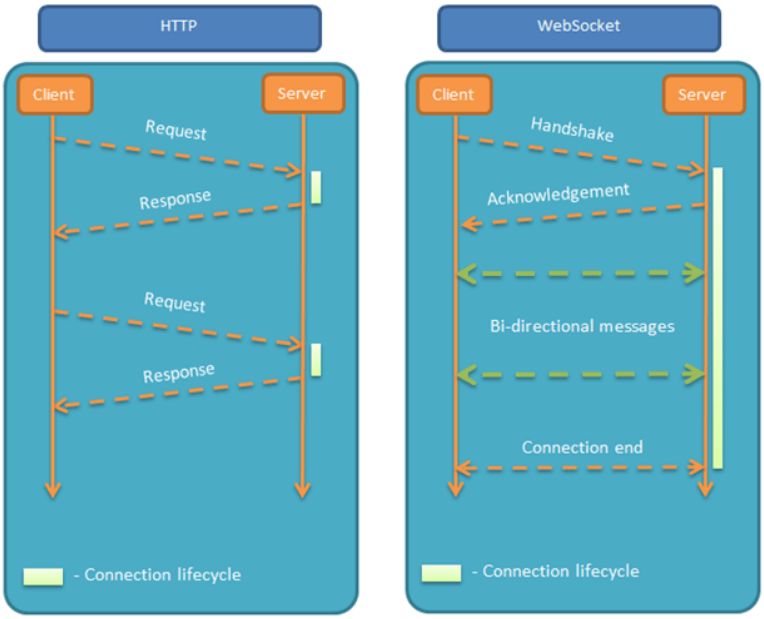
① AddRoundKey—矩阵中的每一个字节都与该次回合密钥（round key）做 XOR 运算；每个子密钥由密钥生成方案产生。
② SubBytes—透过一个非线性的替换函数，用查找表的方式把每个字节替换成对应的字节。
③ ShiftRows—将矩阵中的每个横列进行循环式移位。
④ MixColumns—为了充分混合矩阵中各个直行的操作。这个步骤使用线性转换来混合每内联的四个字节。最后一个加密循环中省略 MixColumns 步骤，而以另一个 AddRoundKey 取代。

4 整体设计方案

4.1 网络协议

本次设计中，我使用了 HTTP 协议处理一般的网络请求：如登录、注册、好友列表获取、个人信息获取、头像更新等功能。

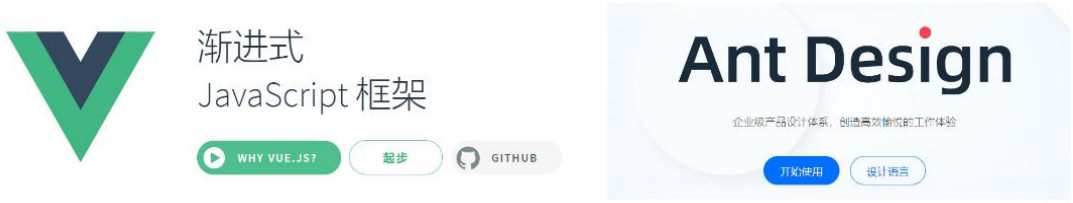
而好友之间点对点的通信，为了持续快速地沟通，我是用 WebSocket 协议来处理信息发送请求。



图片 2 WebSocket 与 HTTP

4.2 客户端技术选型

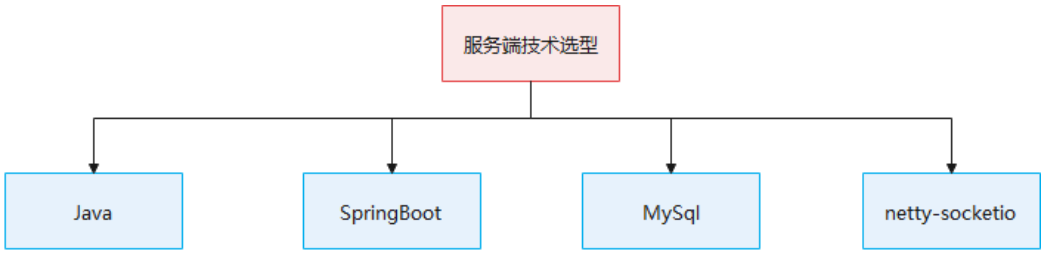
客户端负责的是与用户进行交互，因此在实用之外还需要考虑到界面美观整洁，以给用户带来良好的使用体验。因此，前端选择使用 vue + AntDesign 组件库进行界面构建。另一方面，由于需要建立 WebSocket 连接，发送 WebSocket 请求，因此需要引入 WebSocket 相关功能的实现。这里使用的是 socket.io 这一 NodeJS 第三方模块。



图片 3 Vue 与 AntDesign

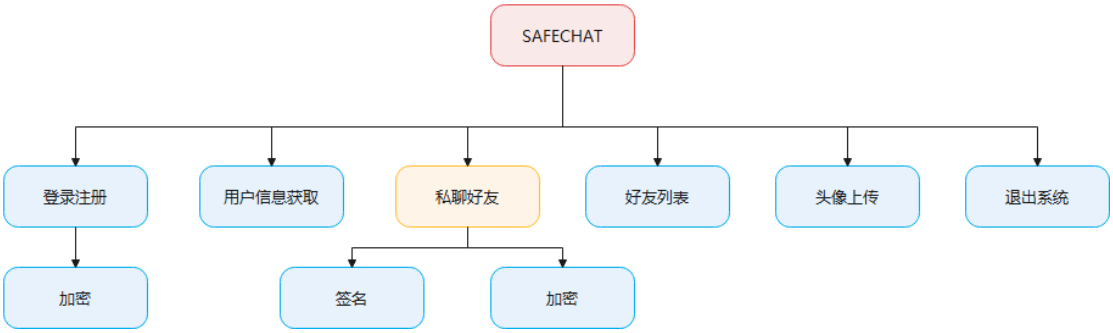
4.3 服务端技术选型

对于服务端，采用了 Java + SpringBoot 为大框架来进行服务端的开发。数据库采用的是经典的关系型数据库 MySql。同时为了建立 WebSocket 连接，处理 WebSocket 请求，选择了 socket.io 的一个 Java 移植版本 netty-socketio。netty-socketio 是一个开源的 Socket.io 服务器端的一个 java 的实现，它基于 Netty 框架，可用于服务端推送消息给客户端。



4.4 整体功能说明

本系统主要包含六个大的功能模块：登陆注册、用户信息获取、信息发送、好友列表显示、头像上传以及退出系统。**其中信息发送是本次课程设计最重要的部分，是安全通信的主要体现。**

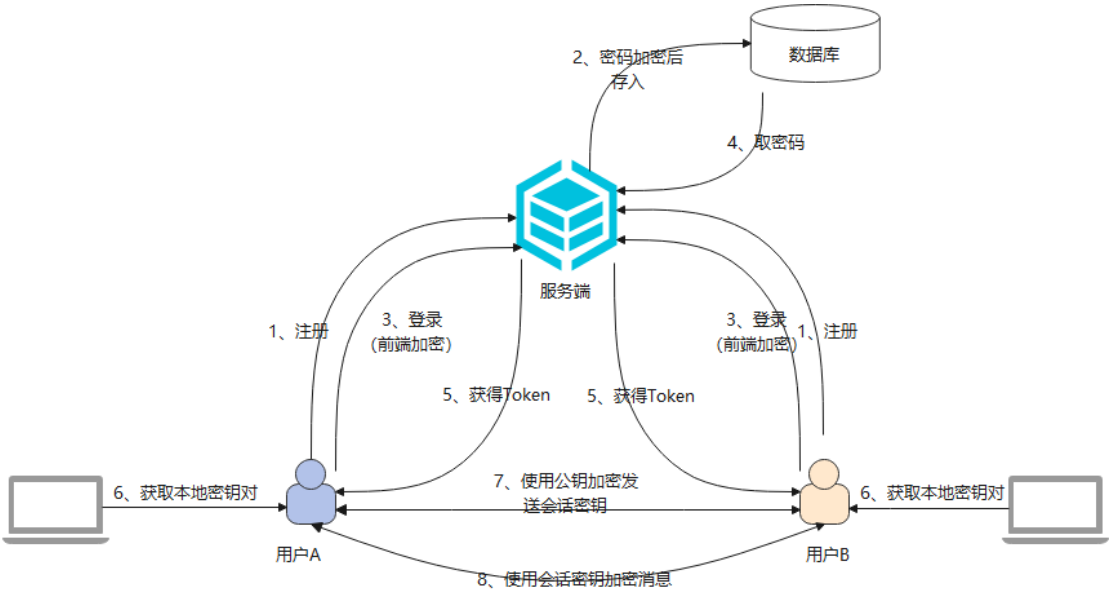


图片 4 整体功能说明

5 安全加密部分代码说明

5.1 整体安全加密功能设计

以两个人的通信为例，整体的设计图如下，具体说明见下文。



图片 5 整体安全加密功能设计图

5.2 HTTP 请求加密

5.2.1 Token 产生

```
1.  private static String sign(String userId,String password){
2.      Algorithm algorithm = Algorithm.HMAC256(password);
3.      String token = JWT.create()
4.              .withClaim(CLAIM_USERID_NAME,userId)
5.              .withExpiresAt(new Date(System.currentTimeMillis()+EXPIRED_TIM
    E/2))
6.              .sign(algorithm);
7.      return token;
8.  }
9.
10. /**
11. * 生成一个登录 token
12. * @param userId
13. * @param password
14. * @return
15. */
```

```
16. public static String loginSign(String userId,String password){
17.     String token = sign(userId,password);
18.     cache.putToken(token,token);
19.     return token;
20. }
```

每次登录产生 Token，并存储在前端的 localStorage 中，每次发送 HTTP 的 POST 和 GET 请求时加在 HTTP Header 中的 Authorization 位。（解决 XSS 和 XSRF 问题）

5.2.2 Token 认证

后端接收 HTTP 请求时需要认证 Token。

如此做可以认证发送 HTTP 请求的用户身份，适用于所有 HTTP 请求

```
1.  /**
2.  * 验证客户端传来 token 是否有效
3.  * 验证逻辑顺序如下：
4.  * 1. token 是否为空
5.  * 2. token 中账号是否存在
6.  * 3. 根据 token 中账号从数据库中获取真实密码等用户信息，并验证用户信息是否有效
7.  */
8.  public static void verifyToken(String clientToken, stu.software.chatroom.commo
    n.CommonService commonService){
9.      if(!StringUtils.hasText(clientToken)){
10.         //token 为空
11.         throw new RuntimeException("无登录令牌！");
12.     }
13.     //从客户端登录令牌中获取当前用户账号
14.     String userId = JWT.decode(clientToken).getClaim(CLAIM_USERID_NAME).asStri
    ng();
15.     if(!StringUtils.hasText(userId)){
16.         //token 中账号不存在
17.         throw new RuntimeException("登录令牌失效！");
18.     }
19.     //取出缓存中的登录令牌
20.     String cacheToken = cache.getToken(clientToken);
21.     if(!StringUtils.hasText(cacheToken)){
22.         //缓存中没有登录令牌
23.         throw new RuntimeException("登录令牌失效！");
24.     }
25.     User user = commonService.getUserById(userId);
26.     if(user==null){
27.         //用户不存在
28.         throw new RuntimeException("用户不存在！");
29.     }
30.     //验证 Token 有效性
```

```
    try{
        Algorithm algorithm = Algorithm.HMAC256(user.getU_pwd());
        JWTVerifier jwtVerifier = JWT.require(algorithm).withClaim(CLAIM_USERID_NAME,userId).build();//构建验证器
        jwtVerifier.verify(cacheToken);
    }catch(TokenExpiredException e){
        //令牌过期，刷新令牌
        String newToken = sign(userId,user.getU_pwd());
        cache.putToken(clientToken,newToken);
    }catch(Exception e){
        e.printStackTrace();
        //令牌验证未通过
        throw new RuntimeException("令牌错误！请登录。");
    }
}
```

5.3 注册密码加密

使用 SHA256 加密注册时用户使用的密码，数据库中存的是密文，这样可防止数据库被攻击导致密码泄露。

u_id	u_pwd	u_avatar	u_name
1	8d969eef6ecad3c2	https://note-image-130	mhy
2	8d969eef6ecad3c2	https://note-image-130	yjq
3	8d969eef6ecad3c2	https://note-image-130	用户1
4	8d969eef6ecad3c2	https://note-image-130	xlz

图片 6 注册密码加密

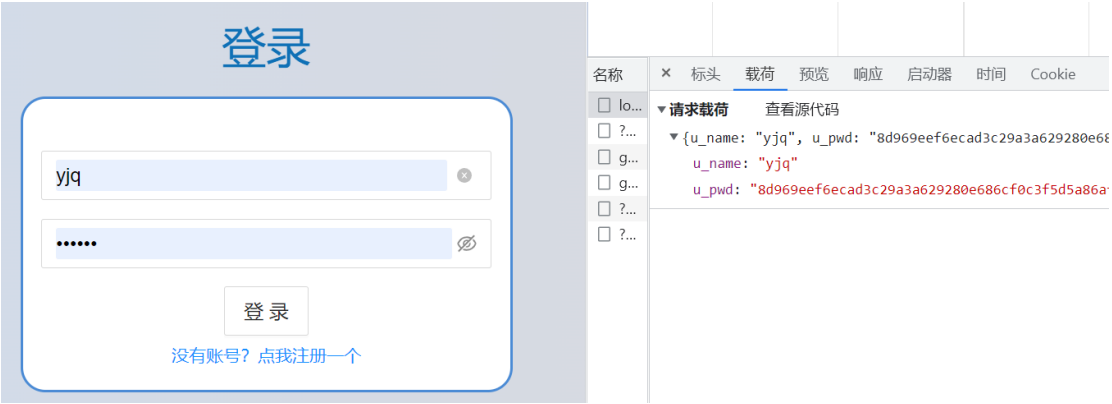
```
/**
* 利用 Apache 的工具类实现 SHA-256 加密
* @return str 加密后的报文
*/
public static String getSHA256Str(String str) {
    MessageDigest messageDigest;
    String encodeSir = str;
    try {
        messageDigest = MessageDigest.getInstance("SHA-256");
        byte[] hash = messageDigest.digest(str.getBytes(StandardCharsets.UTF_8));
        encodeSir = Hex.encodeHexString(hash);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
```

```
    }
    return encodeSir;
}
/**
* 通过该方法将密码加密
*/
public static String encodePwd(String u_pwd) {
    // 密码通过此方法解密并再加密
    return getSHA256Str(u_pwd);
}
```

5.4 登录密码加密

登录时，前端输入明文密码，使用 SHA256 加密该密码以后，再加数据发送到后端。后端根据该加密后的密码与数据库比对，从而验证用户身份。

此做法避免了前端请求数据被拦截导致密码泄露。



图片 7 登陆密码加密

```
import { sha256 } from 'js-sha256';

/**
 * 加密方法
 */
export function PASSWORD(str) {
    let encodedStr = str;
    encodedStr = sha256(encodedStr);
    return encodedStr;
}
const login = () => {
  post("/user/login", {
    u_name: u_name.value,
    u_pwd: PASSWORD(u_pwd.value),
  })
```

```
    .then((res) => {
      tip.success(res.message);
      let token = res.data;
      setLocalToken(token);
      router.push({ name: "Room", query: { usr: u_name.value } });
    })
    .catch((err) => {
      tip.error("账号密码错误！");
    });
};
```

5.5 密钥分配——使用 Keytool[2]

keytool 是个密钥和证书管理工具。它使用户能够管理自己的公钥/私钥对及相关证书，用于（通过数字签名）自我认证（用户向别的用户/服务认证自己）或数据完整性以及认证服务。它还允许用户储存他们的通信对等者的公钥（以证书形式）。

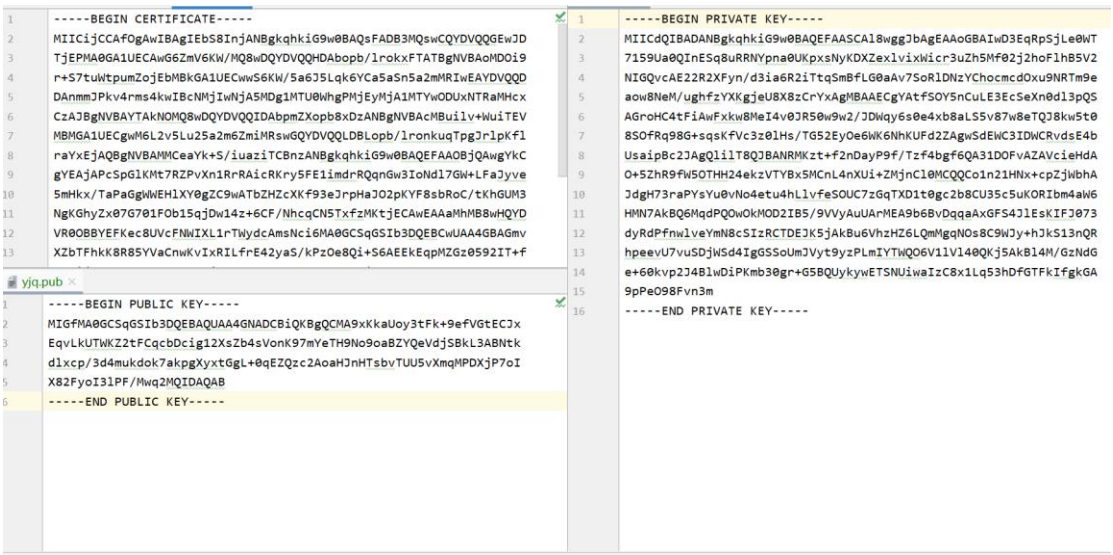
在计算机网络上，OpenSSL 是一个开放源代码的软件库包，应用程序可以使用这个包来进行安全通信，避免窃听，同时确认另一端连接者的身份。这个包广泛被应用在互联网的网页服务器上。

通过如下步骤可以产生证书和公钥

```
1. keytool -genkeypair -storetype PKCS12 -alias yjq - -keyalg RSA -keysize
1024 -dname "CN=易俊泉，OU=西安交通大学，O=软件学院，L=西安，ST=陕西，C=CN" -
keystore D:\mygit\大三下笔记\网安课设\safechat-server\src\main\resources\keys-
and-certs\yjq.keystore -keypass 123456 -storepass 123456 -validity 36500 -v
```

产生二进制文件 yjq.keystore，以上部分可由脚本生成。

经过 KeyStore 的相关操作生成公钥、证书和私钥。



图片 8 CA 分配的公私钥

当用户需要公钥和私钥时，只需要调用相关方法即可。

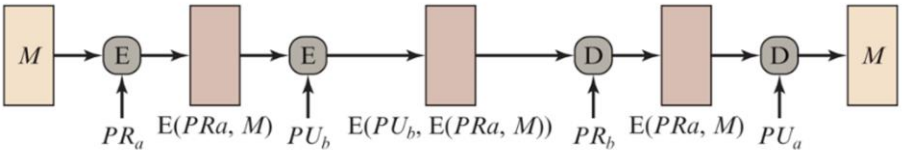
[2] 参考教程 https://blog.csdn.net/m0_59579040/article/details/124811147

```
1.  public static void genKeyPair(String name) throws Exception {
2.      //以 PKCS12 规格，创建 KeyStore
3.      KeyStore keyStore = KeyStore.getInstance("PKCS12");
4.      path = "keys-and-certs/" + name + ".keystore";
5.      //载入 jks 和该 jks 的密码 到 KeyStore 内
6.      keyStore.load(new FileInputStream(new ClassPathResource("keys-and-
    certs/yjq.keystore").getFile()), "123456".toCharArray());
7.
8.      // 要获取 key，需要提供 KeyStore 的别名 和该 KeyStore 的密码
9.      // 获取 keyStore 内所有别名 alias
10.     Enumeration<String> aliases = keyStore.aliases();
11.     String alias = null;
12.     alias = aliases.nextElement();
13.     char[] keyPassword = "123456".toCharArray();
14.     keyPairString.clear();
15.     //私钥
16.     privateKey = (PrivateKey) keyStore.getKey(alias, keyPassword);
17.     keyPairString.put("PR", new String(Base64.getEncoder().encode(privateKey.g
    etEncoded())));
18.     //证书
19.     Certificate certificate = keyStore.getCertificate(alias);
20.     //公钥
21.     publicKey = certificate.getPublicKey();
22.     keyPairString.put("PU", new String(Base64.getEncoder().encode(publicKey.ge
    tEncoded())));
23.
24. }
```

5.6 使用公钥加密保证消息认证和机密性[3]

A 和 B 进行通信，首先使用 A 的私钥对报文 M 进行加密——数字签名；然后 A 用 B 的公钥对上述结果进行加密——保证了保密性。

B 收到消息后，用 B 的私钥解密，再用 A 的公钥验证签名。



这里我使用 RSA 作为加密算法、SHA1WithRSA 作为签名算法，签名和加密的操作实现在类 RSAUtils.java 中。

[3] 参考教程 https://blog.csdn.net/m0_59579040/article/details/124811147.

5.6.1 签名

```
/**
* 私钥签名
* @param content 字符串
* @param priKey 私钥
* @return
* @throws Exception
*/
public static byte[] sign(String content, PrivateKey priKey) throws Exception
{
    Signature signature = Signature.getInstance(SIGALG);
    signature.initSign(priKey);
    signature.update(content.getBytes());
    return signature.sign();
}

/**
* 公钥验证签名
* @param content 字符串
* @param sign 签名
* @param pubKey 公钥
* @return 身份是否真实
* @throws Exception
*/
public static boolean verify(String content, byte[] sign, PublicKey pubKey) th
rows Exception {

    Signature signature = Signature.getInstance(SIGALG);
    signature.initVerify(pubKey);
    signature.update(content.getBytes());
    return signature.verify(sign);
}
```

5.6.2 加密解密

```
/**
* RSA 公钥加密
*
* @param content       加密字符串
* @param publicKey 公钥
* @return 密文
* @throws Exception 加密过程中的异常信息
*/
```

```
9.  public static String encrypt(String content, String publicKey) throws Exceptio
    n {
10.     //base64 编码的公钥
11.     byte[] decoded = Base64.getMimeDecoder().decode(publicKey);
12.     RSAPublicKey pubKey = (RSAPublicKey) KeyFactory.getInstance(KEYALG).genera
    tePublic(new X509EncodedKeySpec(decoded));
13.     System.out.println(pubKey.getAlgorithm());
14.     //RSA 加密
15.     Cipher cipher = Cipher.getInstance(KEYALG);
16.     cipher.init(Cipher.ENCRYPT_MODE, pubKey);
17.     String outStr = Base64.getEncoder().encodeToString(cipher.doFinal(content.
    getBytes("UTF-8")));
18.     return outStr;
19. }
20.
21. /**
22. * RSA 私钥解密
23. *
24. * @param content         加密字符串
25. * @param privateKey 私钥
26. * @return 明文
27. * @throws Exception 解密过程中的异常信息
28. */
29. public static String decrypt(String content, String privateKey) throws Excepti
    on {
30.
31.     //64 位解码加密后的字符串
32.     byte[] inputByte = Base64.getMimeDecoder().decode(content);
33.     //         //base64 编码的私钥
34.     byte[] decoded = Base64.getMimeDecoder().decode(privateKey);
35.     RSAPrivateKey priKey = (RSAPrivateKey) KeyFactory.getInstance("RSA").gener
    atePrivate(new PKCS8EncodedKeySpec(decoded));
36.     //RSA 解密
37.     Cipher cipher = Cipher.getInstance("RSA");
38.     cipher.init(Cipher.DECRYPT_MODE, priKey);
39.     String outStr = new String(cipher.doFinal(inputByte));
40.     return outStr;
41. }
```

5.7 使用 AES 加密消息

因为公钥加密的消息认证比较费时间，所以当两个用户建立消息通信时由一方产生会话密钥，使用公钥加密来传送会话密钥并认证身份。身份认证完成后，使用该会话密钥加密消息，

其中使用对称加密技术 AES 加密消息。

消息报文格式如下：

图片 9 报文格式

1、 id：报文标识 id;
2、 time：报文发送时间
3、 content：报文内容（加密）
4、 type：报文类型：会话密钥消息/公钥消息
5、 sender_name：发送者
6、 receiver_name：接收者
7、 sign：发送者签名。

加密过程如下：

```
1.  public final class AESUtils{
2.      private static final String ALGORITHM = "AES";
3.      public static String genAesSecret(){
4.          try {
5.              KeyGenerator kg = KeyGenerator.getInstance("AES");
6.              //下面调用方法的参数决定了生成密钥的长度，可以修改为 128, 192 或 256
7.              kg.init(256);
8.              SecretKey sk = kg.generateKey();
9.              byte[] b = sk.getEncoded();
10.             String secret = Base64.encodeBase64String(b);
11.             return secret;
12.         }
13.         catch (NoSuchAlgorithmException e) {
14.             e.printStackTrace();
15.             throw new RuntimeException("没有此算法");
16.         }
17.     }
18.     /**
19.      * 根据密钥对指定的明文 plainText 进行加密.
20.      *
21.      * @param plainBytes 明文
22.      * @param keyBytes   密码
23.      * @return 加密后的密文.
24.      * @since 0.0.8
```

```
25.      */
26.     public static byte[] encrypt(byte[] plainBytes, byte[] keyBytes) {
27.         try {
28.             SecretKey secretKey = getSecretKey(keyBytes);
29.             Cipher cipher = Cipher.getInstance(ALGORITHM);
30.             cipher.init(Cipher.ENCRYPT_MODE, secretKey);
31.             return cipher.doFinal(plainBytes);
32.         } catch (Exception e) {
33.             throw new RuntimeException(e);
34.         }
35.     }
36.
37.     /**
38.      * 根据密钥对指定的密文 cipherBytes 进行解密.
39.      *
40.      * @param cipherBytes 加密密文
41.      * @param keyBytes    秘钥
42.      * @return 解密后的明文.
43.      * @since 0.0.8
44.      */
45.     public static byte[] decrypt(byte[] cipherBytes, byte[] keyBytes) {
46.         try {
47.             SecretKey secretKey = getSecretKey(keyBytes);
48.
49.             Cipher cipher = Cipher.getInstance(ALGORITHM);
50.             cipher.init(Cipher.DECRYPT_MODE, secretKey);
51.             return cipher.doFinal(cipherBytes);
52.         } catch (Exception e) {
53.             throw new RuntimeException(e);
54.         }
55.     }
56.
57.     /**
58.      * 获取加密 key
59.      * @param keySeed seed
60.      * @return 结果
61.      * @since 0.0.8
62.      */
63.     private static SecretKey getSecretKey(byte[] keySeed) {
64.         try {
65.             // 避免 linux 系统出现随机的问题
66.             SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
67.             secureRandom.setSeed(keySeed);
68.             KeyGenerator generator = KeyGenerator.getInstance("AES");
```

```
            generator.init(secureRandom);
            return generator.generateKey();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

5.8 服务端加密

结合 RSA 与 AES 的加密如下：

先用公钥加密 RSA 发送对称加密使用的会话密钥，然后再用会话密钥进行 AES 对称加密通信。

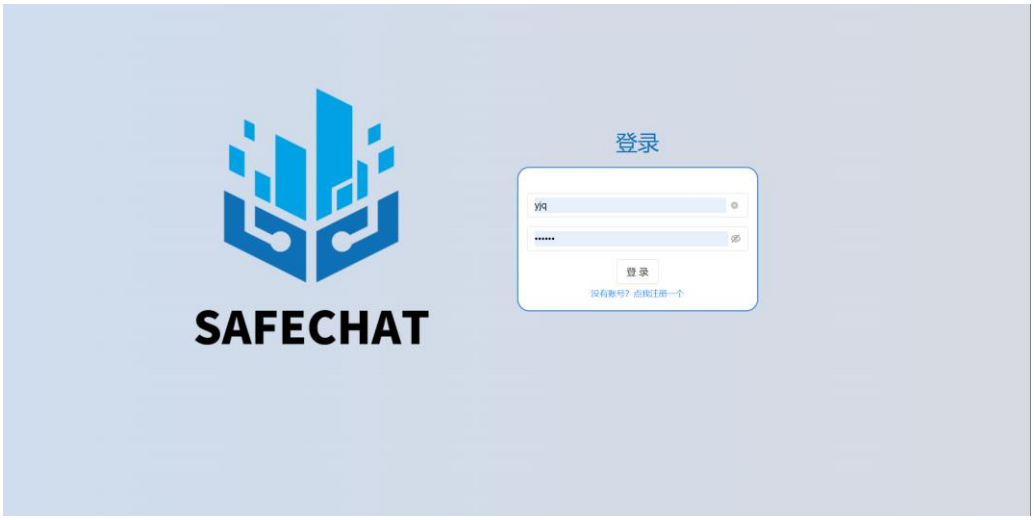
```
 // 监听客户端发送消息
socketIOServer.addEventListener(Constants.EVENT_MESSAGE_TO_SERVER, String.class, (client, data, ackSender) -> {
    String sender_name = getParamsByClient(client, "u_name");
    ObjectMapper mapper = new ObjectMapper();
    Message message = mapper.readValue(data, Message.class);
    String receiver_name = message.getReceiver_name();

    if (message.getType().equals(Constants.MASTER_MESSAGE)) {
        //使用公钥加密传送会话密钥
        if (AesKey.equals("")) {
            log.info("用户" + sender_name + "生成会话密钥");
            AesKey = AESUtils.genAesSecret();
            message.setContent(AesKey);
            log.info("用户" + sender_name + "使用用户" + sender_name + "的私钥对会话密钥进行签名");
            String sign = new String(RSAUtils.sign(message.getContent(), RSAUtils.getPrivateKey()), "ISO-8859-1");
            message.setSign(sign);
            String result = RSAUtils.encrypt(message.getContent(), publicKeyStringMap.get(receiver_name));
            log.info("使用用户" + receiver_name + "的公钥对会话密钥进行加密：" + result);
            message.setContent(result);
            sendMessageToFriend(message.getReceiver_name(), message);
        } else {
            return;
        }
    } else {
```

```
25.         //使用会话密钥发送消息
26.         byte[] bytes = AESUtils.encrypt(message.getContent().getBytes(), AesKe
    y.getBytes());
27.         String encrypt = new String(bytes, "ISO-8859-1");
28.         log.info("用户" + sender_name + "使用会话密钥加密消息");
29.         message.setContent(encrypt);
30.         sendMessageToFriend(message.getReceiver_name(), message);
31.     }
32. });
33. //
34. //GBK,  GB2312，UTF-8 等一些编码方式为多字节或者可变长编码，原来的字节数组就被改变
    了，再转回原来的 byte[]数组就会发生错误了。
35. //ISO-8859-1 通常叫做 Latin-1，Latin-1 包括了书写所有西方欧洲语言不可缺少的附加字
    符，其中 0~127 的字符与 ASCII 码相同，
36. // 它是单字节的编码方式，在来回切换时不会出现错误。
37.
38. // 监听客户端接收消息
39. socketIOServer.addEventListener("receive_triger", String.class, (client, data,
     ackSender) -> {
40.     ObjectMapper mapper = new ObjectMapper();
41.     Message message = mapper.readValue(data, Message.class);
42.     String sender_name = message.getSender_name();
43.     String receiver_name = message.getReceiver_name();
44.     if (message.getType().equals(Constants.MASTER_MESSAGE)) {
45.         log.info("收到来自" + sender_name + "发送给
    " + message.getReceiver_name() + "的消息： " + message.getContent());
46.         String result = RSAUtils.decrypt(message.getContent(), RSAUtils.getKey
    Pair().get("PR"));
47.         log.info("用户" + receiver_name + "使用用户" + receiver_name + "的私钥对
    消息进行解密：");
48.         message.setContent(result);
49.         log.info("用户" + receiver_name + "使用用户" + sender_name + "的公钥对消
    息进行验证签名");
50.         Boolean sign = (RSAUtils.verify(message.getContent(), message.getSign(
    ).getBytes("ISO-8859-1"), publicKeyMap.get(sender_name)));
51.         if (sign) {
52.             log.info("签名验证成功！身份无误");
53.         } else {
54.             throw new Exception("签名错误！");
55.         }
56.         receiveMessageFromFriend(message.getReceiver_name(), message);
57.     } else {
58.         log.info("收到来自" + sender_name + "发送给
    " + message.getReceiver_name() + "的消息： " + message.getContent());
```

```
59.         String text = new String(AESUtils.decrypt(message.getContent().getBytes("ISO-8859-1"), AesKey.getBytes()), "UTF-8");
60.         log.info("用户" + receiver_name + "使用会话密钥进行解密");
61.         message.setContent(text);
62.         receiveMessageFromFriend(message.getReceiver_name(), message);
63.     }
64. });
```

6 通信过程演示

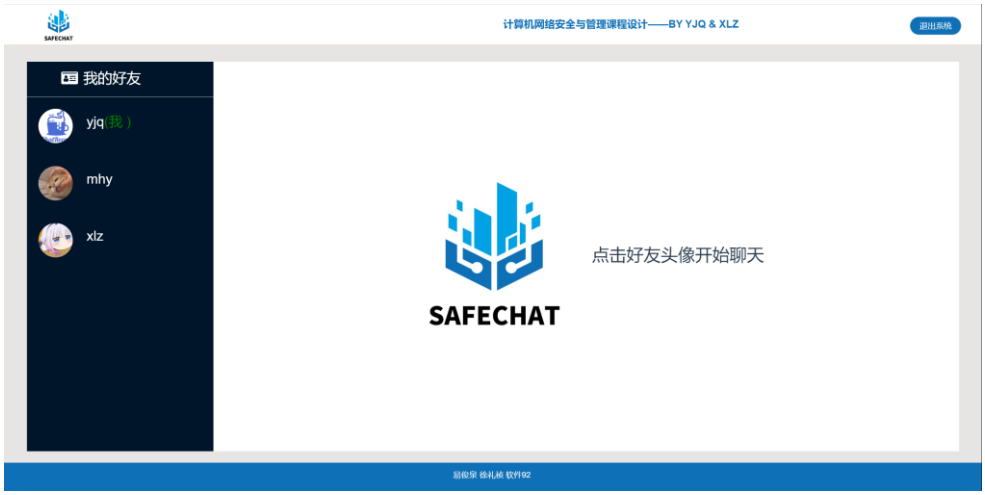
6.1 登录



图片 10 登录

6.2 进入主页面

可以看见我的好友列表



图片 11 好友列表

同时获取本地密钥库中的公私钥并将其加入公钥库。

```
s.s.c.w.impl.SocketIOServiceImpl      : 新建客户端连接: 127.0.0.1, 用户名 :yjq
o.s.web.method.HandlerMethod          : Arguments: [eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJVc2VySUQiOiIyIiwiZ
s.s.c.w.impl.SocketIOServiceImpl      : 密钥获取成功: MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDlRZIi5KIDNNqlRx6ua
s.s.c.w.impl.SocketIOServiceImpl      : 目前公钥库中有: 2 个公钥
```

6.3 选择好友进行私聊

选择好友进行私聊，进入聊天界面。

图片 12 私聊界面

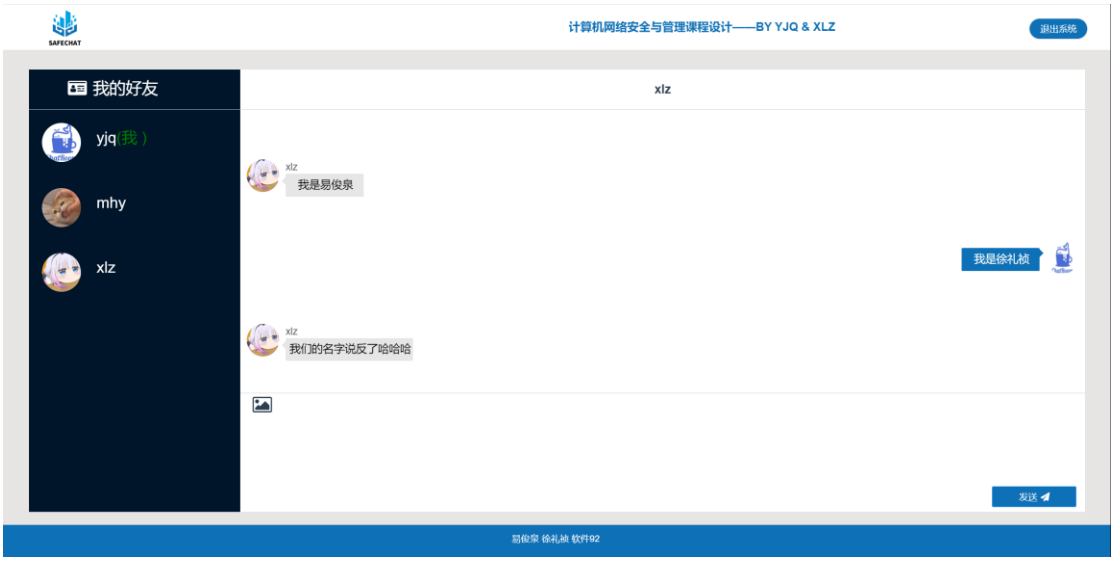
```
s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz生成会话密钥
s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz使用用户xlz的私钥对会话密钥进行签名
s.s.c.w.impl.SocketIOServiceImpl      : 使用用户yjq的公钥对会话密钥进行加密: aAyYwCyAwgRUggb//9/AQXUoS1A8y3M533
s.s.c.w.impl.SocketIOServiceImpl      : 收到来自xlz发送给yjq的消息: aAyYwCyAwgRUggb//9/AQXUoS1A8y3M533yXxuoY5U
s.s.c.w.impl.SocketIOServiceImpl      : 用户yjq使用用户yjq的私钥对消息进行解密:
s.s.c.w.impl.SocketIOServiceImpl      : 用户yjq使用用户xlz的公钥对消息进行验证签名
s.s.c.w.impl.SocketIOServiceImpl      : 签名验证成功! 身份无误
```

图片 13 利用公钥加密传递会话密钥

6.4 发送消息

在输入框中输入消息，点击发送，接收者和发送者的聊天框都会出现相应的消息。此消息是经过后端 AES 对称加密解密得到的。

图片 14 聊天视角 1



图片 15 聊天视角 2

```
s.s.c.w.impl.SocketIOServiceImpl      : 收到来自yjq发送给xlz的消息: Æ3 k    ?XAÛ?Ðz?âPr
s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz使用会话密钥进行解密

s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz使用会话密钥加密消息
s.s.c.w.impl.SocketIOServiceImpl      : 收到来自xlz发送给yjq的消息: 3ãA¹?ÔH´{£ ¡!õñÈ~
s.s.c.w.impl.SocketIOServiceImpl      : 用户yjq使用会话密钥进行解密

s.s.c.w.impl.SocketIOServiceImpl      : 用户yjq使用会话密钥加密消息
s.s.c.w.impl.SocketIOServiceImpl      : 收到来自yjq发送给xlz的消息: ª¢ ë.??8‹_ëí rçæ^
s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz使用会话密钥进行解密

s.s.c.w.impl.SocketIOServiceImpl      : 用户xlz使用会话密钥加密消息
s.s.c.w.impl.SocketIOServiceImpl      : 收到来自xlz发送给yjq的消息: <êê$??¾Ï‹vc\Dã6EÉØ^?HTSiÃ@s
s.s.c.w.impl.SocketIOServiceImpl      : 用户yjq使用会话密钥进行解密
```

图片 16 加密解密过程

7 设计总结

本次课程设计，花费我们的时间比较多，时间主要花在了算法的学习和工具类的使用上。在网上收集资料的时候，发现很多资料都是有问题的，这也导致我和徐礼祯同学花费了很多不必要的时间。但是随着算法的一步步完善，看着这个软件一步步地完成，还是很让人满足的。

从此次设计中，我学习到了计算机网络安全与管理的相关知识，将理论应用于实际。同时也增进了和同学之间的沟通。最后，十分感谢田暄老师的指导！

8 仓库地址

源代码详见 github 仓库（仓库有所迁移）：

后端地址：https://github.com/yijunquan-afk/safechat-server

前端地址： https://github.com/yijunquan-afk/safechat-client