网络安全技术

实
验
报
告

学　　院 计算机学院

年　　级 2017

班　　级 1 班

学　　号 1711425

姓　　名 曹元议

2020 年 3 月 23 日

目录

一、实验目的.....1
二、实验内容.....1
2.1 DES 算法简述.....1
2.2 使用 socket 进行 TCP 通信过程概述（以 Java 为例）.....4
三、实验步骤及实验结果.....6
3.1 DES 算法的实现.....6
3.1.1 16 个子密钥的生成.....8
3.1.2 DES 加密与解密.....10
3.2 TCP 通信的实现.....15
3.3 实验结果展示.....24
3.3.1 运行程序的方法.....24
3.3.2 多线程聊天测试.....26
四、实验遇到的问题及其解决方法.....35
五、实验结论.....37

一、实验目的

1. 理解 DES 加密和解密的原理
2. 理解 TCP 协议的工作原理
3. 掌握使用 socket 进行网络编程的方法

二、实验内容

本次实验的总体要求是编写一个 TCP 聊天程序，通信内容需要经过 DES 加密和解密。不限制平台和编程语言。

2.1 DES 算法简述

DES 就是“数据加密标准”的意思，它是建立在 Feistel 密码系统上的密码系统。DES 算法是上世纪 70 年代到 90 年代被广泛采用的数据加密算法，曾经作为美国的数据加密标准。后因密钥长度太短和性能较差而被其他更先进的加密算法淘汰。

DES 加密是使用 64 bit 的明文（不足 64 bit 则用 0 补齐）和 64 bit 的初始密钥，生成同样也是 64 bit 的密文。在 64 bit 的初始密钥当中，第 $8i(i=1,2,...,8)$ 是奇偶校验位，只用于检错不用于计算，因此初始密钥只有 56 bit 是有效的。56 bit 有效密钥经过计算会产生 16 个 48 bit 的子密钥。在加密的过程中需要进行 16 次迭代，每次迭代都按顺序使用一个 48 bit 子密钥。解密和加密的过程其实是一样的，只是使用子密钥的次序是相反的。

令 M 、 C 、 K 分别为 64 bit 的明文、64 bit 的密文、64 bit 的初始密钥。
DES 产生 16 个子密钥的方式如下：

1. 令 $K=k_1k_2...k_{64}$，先对 K 进行如下图初始置换运算 I_k 重新排列生成 56 bit 的二进制字符串 $I_k(K)$ ，每个 bit 按行排列。

$$I_K(K)=\begin{matrix} k_{57} & k_{49} & k_{41} & k_{33} & k_{25} & k_{17} & k_9 & k_1 & k_{58} & k_{50} & k_{42} & k_{34} & k_{26} & k_{18} \\ k_{10} & k_2 & k_{59} & k_{51} & k_{43} & k_{35} & k_{27} & k_{19} & k_{11} & k_3 & k_{60} & k_{52} & k_{44} & k_{36} \\ k_{63} & k_{55} & k_{47} & k_{39} & k_{31} & k_{23} & k_{15} & k_7 & k_{62} & k_{54} & k_{46} & k_{38} & k_{30} & k_{22} \\ k_{14} & k_6 & k_{61} & k_{53} & k_{45} & k_{37} & k_{29} & k_{21} & k_{13} & k_5 & k_{28} & k_{20} & k_{12} & k_4 \end{matrix}$$

2. 将 $I_k(K)$ 分成左右两半部分 U_0 和 V_0，其中 $|U_0|=|V_0|=28$ 。
3. 之后子密钥 K_i 按照如下方式生成：

$$U_i = LS_{z(i)}(U_{i-1}),$$
$$V_i = LS_{z(i)}(V_{i-1}),$$
$$K_i = P_K(U_iV_i),$$
$$i = 1,2,...,16.$$

其中 $LS_{z(i)}(Y)$ 表示将 Y 循环左移 $z(i)$ 次， $z(i)$ 的定义如下：

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$z(i)$	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

P_K 是置换运算，它将 56 bit 二进制字符串压缩成 48 bit 的字符串，令 $X = x_1x_2...x_{56}$，$P_K(X)$ 的规则如下，每个 bit 按行排列：

$$P_K(X) = \begin{matrix} x_{14} & x_{17} & x_{11} & x_{24} & x_1 & x_5 & x_3 & x_{28} & x_{15} & x_6 & x_{21} & x_{10} \\ x_{23} & x_{19} & x_{12} & x_4 & x_{26} & x_8 & x_{16} & x_7 & x_{27} & x_{20} & x_{13} & x_2 \\ x_{41} & x_{52} & x_{31} & x_{37} & x_{47} & x_{55} & x_{30} & x_{40} & x_{51} & x_{45} & x_{33} & x_{48} \\ x_{44} & x_{49} & x_{39} & x_{56} & x_{34} & x_{53} & x_{46} & x_{42} & x_{50} & x_{36} & x_{29} & x_{32} \end{matrix}$$

DES 加密算法的步骤如下：

1. 先对明文 $M = m_1m_2...m_{64}$ 进行 IP 初始置换，规则如下（按行排列，下图的数字的位置代表 $m_i(i \in [1,64])$ 进行 IP 初始置换后所在的位置）：

$$IP(M) = \begin{matrix} 58 & 50 & 42 & 34 & 26 & 18 & 10 & 2 & 60 & 52 & 44 & 36 & 28 & 20 & 12 & 4 \\ 62 & 54 & 46 & 38 & 30 & 22 & 14 & 6 & 64 & 56 & 48 & 40 & 32 & 24 & 16 & 8 \\ 57 & 49 & 41 & 33 & 25 & 17 & 9 & 1 & 59 & 51 & 43 & 35 & 27 & 19 & 11 & 3 \\ 61 & 53 & 45 & 37 & 29 & 21 & 13 & 5 & 63 & 55 & 47 & 39 & 31 & 23 & 15 & 7 \end{matrix}$$

2. 将 $IP(M)$ 分成左右两半部分 L_0 和 R_0，其中 $|L_0|=|R_0|=32$ 。
3. 对 $i = 1,2,...,16$，按顺序作如下运算：

$$L_i = R_{i-1},$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i).$$

其中 $\oplus$ 是异或运算， $F(R_{i-1}, K_i) = P(S(EP(R_{i-1}) \oplus K_i))$ 。

扩展置换运算 $EP(U)$ 将长度为 32 bit 的二进制字符串 $U = u_1u_2...u_{32}$ 扩展成 48 bit 字符串，规则如下图（按行排列）：

$$EP(U) = \begin{matrix} u_{32} & u_1 & u_2 & u_3 & u_4 & u_5 \\ u_4 & u_5 & u_6 & u_7 & u_8 & u_9 \\ u_8 & u_9 & u_{10} & u_{11} & u_{12} & u_{13} \\ u_{12} & u_{13} & u_{14} & u_{15} & u_{16} & u_{17} \\ u_{16} & u_{17} & u_{18} & u_{19} & u_{20} & u_{21} \\ u_{20} & u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ u_{24} & u_{25} & u_{26} & u_{27} & u_{28} & u_{29} \\ u_{28} & u_{29} & u_{30} & u_{31} & u_{32} & u_1 \end{matrix}$$

S 盒替换为选择压缩运算。将 48 bit 的二进制字符串压缩成 32 bit 的二进制字符串，替换规则如下图（查下表得到数字要被转换成相应的 4 bit 二进制字符串）：

表 2.2 S 盒的定义

S_1：	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2：	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3：	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4：	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5：	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6：	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7：	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8：	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

令二进制字符串 $Y = y_1 y_2 ... y_{48}$，S 盒替换的公式为：

$S(Y) = S_1(Y[1,6])S_2(Y[7,12])...S_8(Y[43,48])$

其中 $Y[6r-5,6r](r=1,2,...,8)$ 都是 Y 的 6 bit 子字符串。以 $Y[1,6]$ 为例，查表的过程为：先计算出 $Y[1,6]$ 的数值 v 作为在表 S_1 中的位置，查表 S_1 的第 v 个位置的数字，将其转换为相应的 4 bit 字符串 $S_1(Y[1,6])$。

令二进制字符串 $V = v_1 v_2 ... v_{32}$，进行置换运算 P，将 $S(Y)$ 重新排列，规则如下图（按行排列）：

$$P(V) = \begin{matrix} v_{16} & v_7 & v_{20} & v_{21} & v_{29} & v_{12} & v_{28} & v_{17} & v_1 & v_{15} & v_{23} & v_{26} & v_5 & v_{18} & v_{31} & v_{10} \\ v_2 & v_8 & v_{24} & v_{14} & v_{32} & v_{27} & v_3 & v_9 & v_{19} & v_{13} & v_{30} & v_6 & v_{22} & v_{11} & v_4 & v_{25} \end{matrix}$$

4. 最后，令密文 $C = IP^{-1}(R_{16}L_{16})$（注意在这里 L_{16} 和 R_{16} 拼接成 64 bit 的二进制字符串时左右两部分要交换，即拼接成 $R_{16}L_{16}$）。$IP^{-1}(C)$ 是 $IP(M)$ 的逆置换，规则如下图（按行排列）：

$$IP^{-1}(C) = \begin{matrix} 40 & 8 & 48 & 16 & 56 & 24 & 64 & 32 & 39 & 7 & 47 & 15 & 55 & 23 & 63 & 31 \\ 38 & 6 & 46 & 14 & 54 & 22 & 62 & 30 & 37 & 5 & 45 & 13 & 53 & 21 & 61 & 29 \\ 36 & 4 & 44 & 12 & 52 & 20 & 60 & 28 & 35 & 3 & 43 & 11 & 51 & 19 & 59 & 27 \\ 34 & 2 & 42 & 10 & 50 & 18 & 58 & 26 & 33 & 1 & 41 & 9 & 49 & 17 & 57 & 25 \end{matrix}$$

DES 解密的步骤和加密步骤唯一的不同就是使用子密钥的次序相反，加密的时候使用子密钥的次序是 $K_1, K_2, ..., K_{16}$，解密时使用子密钥的次序是 $K_{16}, K_{15}, ..., K_1$。

DES 解密的步骤如下：

1. 先对密文 $C = c_1 c_2 ... c_{64}$ 进行 IP 初始置换得到 $IP(C)$ 。
2. 将 $IP(C)$ 分成左右两半部分 L_0' 和 R_0'，其中 $|L_0'| = |R_0'| = 32$
3. 对 $i = 1,2,...,16$，按顺序作如下运算：

   $L_i' = R_{i-1}',$

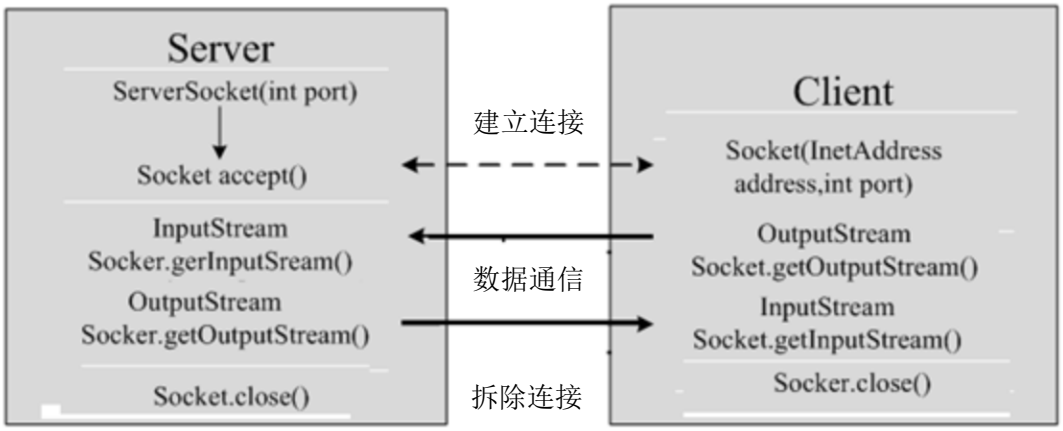
   $R_i' = L_{i-1}' \oplus F(R_{i-1}', K_{17-i}).$
4. 最后，令明文 $M = IP^{-1}(R_{16}'L_{16}')$（注意在这里 L_{16}' 和 R_{16}' 拼接成 64 bit 的二进制字符串时左右两部分要交换，即拼接成 $R_{16}'L_{16}'$）。

2.2 使用 socket 进行 TCP 通信过程概述（以 Java 为例）

这次实验我使用的编程语言是 Java，所以就以 Java 为例概括使用 socket 进行 TCP

通信的过程。

使用 Java 进行 TCP 通信的编程模型如下：



服务器程序的工作过程包含以下基本的步骤：

1. 使用 `ServerSocket(int port)` 创建一个服务器端套接字 `ServerSocket`，并绑定到指定端口上。用于监听客户端的请求。
2. 调用 `accept()`方法监听连接请求：如果客户端请求连接，则接受连接，创建与该客户端的通信套接字对象 `Socket`。否则该方法将一直处于等待。
3. 调用该 `Socket` 对象的 `getOutputStream()` 和 `getInputStream ()`获取输出流和输入流，开始网络数据的发送和接收。（如果数据发送完了或者接收完了需要调用 `shutdownOutput()`和 `shutdownInput()`关闭输出流和输入流）
4. 关闭 `Socket` 对象：某客户端访问结束，关闭与之通信的套接字，服务器与该客户端的连接断开。
5. 关闭 `ServerSocket`：如果不再接收任何客户端的连接的话，调用 `close()` 进行关闭。

客户端程序的工作过程包含以下基本的步骤：

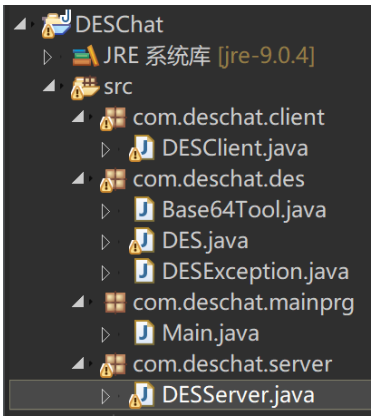
1. 创建 Socket：根据指定服务端的 IP 地址或端口号构造 Socket 类对象，创建的同时会自动向服务器方发起连接。若服务器端响应，则建立客户端到服务器的通信线路。若连接失败，会出现异常。
2. 使用 `getInputStream()`方法获得输入流，使用 `getOutputStream()`方法获得输出流，进行数据传输。通过输入流读取服务器发送的信息，通过输出流将信息发送给服务器。（如果数据发送完了或者接收完了需要调用 `shutdownOutput()`和 `shutdownInput()`关闭输出流和输入流）

4. 关闭 Socket：断开客户端到服务器的连接。

三、实验步骤及实验结果

本次实验的步骤为：编写代码→测试实验结果。
本次实验的编程部分主要分为两个步骤：DES 算法的实现、TCP 通信的实现。为了不让输出的结果过于凌乱，我采用了【Swing + 命令行 + 文件流】的方式实现。Swing 用来实现可视化界面，可视化界面包括客户端和服务器的通信日志和消息的发送；命令行用于显示 DES 加密和解密的输入和输出；Java 的错误流输出到错误日志文件流中，可由错误日志文件查看异常堆栈信息。

这里主要展示与原理相关的代码，报告中没有展示的代码可以查阅项目文件的源代码。与可视化界面相关的代码也会尽量省略。项目文件的源代码组织如下：



代码中的一些注释我使用的说法是实验文档上的说法，前面简述 DES 算法的时候使用的是课本上的说法，因此会出现和前面所说的一些说法不统一。

3.1 DES 算法的实现

在包 com.deschat.des 中，类 DES 用于 DES 算法的实现。实验文档中的实现主要是使用位操作，但是考虑到位操作实现比较繁琐，且对位操作不熟悉的情况下进行复杂的位操作比较容易出错还不太容易调试，因此我的实现是基于 Java 的 String 和 StringBuilder 类型，转换为对有效字符只有'0'和'1'的字符串进行操作。

DES 类的构造函数如下，在这里进行子密钥生成 genKey()：

```
1   /**
```

```
    * 构造方法
    * @param k 初始密钥（限8个字符）
    * @throws DESException 当密钥的字符个数不为8时，抛出此异常
    */
   public DES(String k)  throws DESException {
       try {
           //统一使用utf-8编解码是为了避免可能出现的乱码情况
           if(k.getBytes("utf-8").length!=8) {
               throw new DESException("The length of cipher key is not 8 !");
           }
       } catch (UnsupportedEncodingException e) {
           // TODO 自动生成的 catch 块
           e.printStackTrace();
       }
       srcKey=k;
       genKey();//在这里生成16个子密钥
   }
```

获取加密或解密结果的函数 getResult()定义如下，这是 public 方法，外部调用加密和解密函数必须先通过这个函数。生成子密钥的函数 genKey()放在了构造函数（参数为初始密钥），这样每个初始密钥对应一个 DES 类的对象，获取结果的时候只需要通过这个初始密钥所对应的 DES 类对象调用 getResult()并选择模式，不需要重新计算子密钥，以提高效率：

```
   /**
    * 获取加密或解密的结果
    * @param text 源文本，加密的源文本为本来的字符串，解密的源文本为加密生成的二进制字符串
    * @param m 模式选择，true为加密，false为解密
    * @return 加密或解密的结果
    */
   public String getResult(String text,boolean m) {
       mode=m;
       if(mode==true) {
           plaintext=text;
           encry();
           return ciphertext;
       }
       else {
           ciphertext=text;
           decry();
           return plaintext;
       }
   }
```

3.1.1 16 个子密钥的生成

类 DES 的 genKey()函数用于生成 16 个子密钥。genKey()函数如下：

```
/**
 * 生成16次迭代所使用的子密钥
 */
private void genKey() {
    String[] temp=ope_pc_1();
    for(int i=0;i<16;i++) {
        temp[0]=ope_shift(temp[0],lefttable[i]);
        temp[1]=ope_shift(temp[1],lefttable[i]);
        keys[i]=ope_pc_2(temp[0]+temp[1]);//左右两部分拼接，进行密钥置换选择PC-2运算
    }
}
```

其中使用的辅助函数 ope_pc_1()用于对初始密钥进行密钥初始置换运算 I_k（PC-1）：

```
/**
 * 使用初始64位密钥进行置换选择PC-1运算
 * @return 56位有效位输出，result[0]是左半部分，result[1]是右半部分，输出结果之后将进行循环左移生成16次迭代左密钥的过程
 */
private String[] ope_pc_1() {
    String[] result=new String[2];
    String src="0"+genBinaryKey(srcKey,8);
    StringBuilder s=new StringBuilder();
    //生成初始密钥对应的二进制字符串：初始密钥的每个字符8 bit，第8个bit为奇校验码
    //即使每8 bit出现的1的个数为奇数
    for(int i=1;i<=56;i++) {
        s.append(src.charAt(keyleftright[i]));
    }
    //将56位密钥拆分成左右两部分
    result[0]=s.toString().substring(0,28);
    result[1]=s.toString().substring(28);
    return result;
}
```

密钥初始置换运算 I_k（PC-1）对应的数据定义如下（最前面的-1 为占位）：

```
/** 等分密钥PC-1 */
private static final byte[] keyleftright =  {
```

```
        -1,57,49,41,33,25,17,9,1,58,50,42,34,26,18,
        10,2,59,51,43,35,27,19,11,3,60,52,44,36,
        63,55,47,39,31,23,15,7,62,54,46,38,30,22,
        14,6,61,53,45,37,29,21,13,5,28,20,12,4
};
```

辅助函数 ope_shift()用于密钥循环左移：

```
/**
 * 密钥循环左移运算
 * @param src 要循环左移的二进制字符串
 * @param index 循环左移的位数
 * @return 循环左移的结果
 */
private String ope_shift(String src,int index) {
    return src.substring(index)+src.substring(0,index);
}
```

密钥循环左移对应的数据定义如下：

```
/** 密钥循环左移 */
private static final byte[] lefttable = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
```

辅助函数 ope_pc_2()用于密钥置换运算 P_K（PC-2）：

```
/**
 * 密钥置换选择PC-2运算，生成48位子密钥
 * @param src 循环左移运算之后两部分拼接的结果
 * @return 48位子密钥
 */
private String ope_pc_2(String src) {
    src="0"+src;
    StringBuilder s=new StringBuilder();
    for(int i=1;i<=48;i++) {
        s.append(src.charAt(keychoose[i]));
    }
    return s.toString();
}
```

密钥置换运算 P_K（PC-2）对应的数据定义如下（最前面的-1 为占位）：

```
/** 密钥选取PC-2 */
private static final byte[] keychoose = {
        -1,14,17,11,24,1,5,3,28,15,6,21,10,
        23,19,12,4,26,8,16,7,27,20,13,2,
        41,52,31,37,47,55,30,40,51,45,33,48,
```

```
6           44,49,39,56,34,53,46,42,50,36,29,32
7   };
```

3.1.2 DES 加密与解密

DES 加密函数 encry()定义如下：

```
1   /**
2    * 加密生成二进制字符串，每 8 个字节进行一次加密
3    */
4   private void encry() {
5       String temp="";
6       String binary=genBinaryMsg(plaintext);//将明文转为二进制字符串，使用
    utf-8 编码
7       for(int i=0;i<binary.length();i+=64) {
8           String s=firstIP(check64(binary.substring(i)));
9           temp+=lastIP(itra16(s.substring(0,32),s.substring(32)));//初始 IP
    置换后的结果拆成左右两部分
10      }
11      ciphertext=temp;//加密产生结果输出为二进制字符串
12  }
```

DES 解密函数 decry()定义如下：

```
1   /**
2    * 解密二进制字符串生成明文，每 8 个字节进行一次解密
3    */
4   private void decry() {
5       String temp="";
6       String binary=ciphertext;
7       //先检查输入的密文是不是二进制字符串
8       for(int i=0;i<binary.length();i++) {
9           if(binary.charAt(i)!='0'&&binary.charAt(i)!='1')
10              throw new DESException("The ciphertext must be binary string !");
11      }
12      for(int i=0;i<binary.length();i+=64) {
13          String s=firstIP(check64(binary.substring(i)));
14          temp+=lastIP(itra16(s.substring(0,32),s.substring(32)));//初始 IP
    置换后的结果拆成左右两部分
15      }
16      plaintext=binary2String(temp); //将解密生成的二进制字符串转为普通字符串，
    按照 utf-8 编码转换
17  }
```

加密和解密的辅助函数 `firstIP()`定义如下，用于初始置换 *IP*：

```
/**
 * 初始置换 IP
 * @param src 待转换的对应源文本的二进制字符串
 * @return 转换结果，作为 16 次迭代的初始值
 */
private String firstIP(String src) {
    src="0"+src;
    StringBuilder s=new StringBuilder();
    for(int i=1;i<=64;i++) {
        s.append(src.charAt(pc_first[i]));
    }
    return s.toString();
}
```

初始置换 *IP* 对应的数据定义如下（最前面的-1 为占位）：

```
/** 初始置换 IP */
private static final byte[] pc_first =
{ -1,58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
        62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
        57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
        61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7
};
```

加密和解密的辅助函数 `lastIP()`定义如下，用于逆初始置换 IP^{-1}。

```
/**
 * 逆初始置换 IP^{-1}
 * @param src 待转换的二进制字符串
 * @return 转换结果，之后转为普通字符串作为加密或解密的结果
 */
private String lastIP(String src) {
    src="0"+src;
    StringBuilder s=new StringBuilder();
    for(int i=1;i<=64;i++) {
        s.append(src.charAt(pc_last[i]));
    }
    return s.toString();
}
```

逆初始置换 IP^{-1} 对应的数据定义如下（最前面的-1 为占位）：

```
/** 逆初始置换 IP^{-1} */
```

```
2   private static final byte[] pc_last = { -1,40,8,48,16,56,24,64,32,
    39,7,47,15,55,23,63,31,
3           38,6,46,14,54,22,62,30, 37,5,45,13,53,21,61,29,
4           36,4,44,12,52,20,60,28, 35,3,43,11,51,19,59,27,
5           34,2,42,10,50,18,58,26, 33,1,41,9,49,17,57,25
6   };
```

进行 16 次迭代的函数 itra16()定义如下，根据不同模式（加密还是解密）选择使用子密钥的顺序：

```
1   /**
2    * 16 次迭代
3    * @param left 初始置换 IP 产生的左半部分
4    * @param right 初始置换 IP 产生的右半部分
5    * @return 16 次迭代之后左右部分交换再拼接的结果，作为逆初始置换 IP^{-1}的输入
6    */
7   private String itra16(String left,String right) {
8       if(mode==true) {
9           for(int i=0;i<16;i++) {
10              String copyLeft=left;
11              left=right;
12              right=strxor(copyLeft,f(right,keys[i]));
13          }
14      }
15      else {
16          for(int i=15;i>=0;i--) {
17              String copyLeft=left;
18              left=right;
19              right=strxor(copyLeft,f(right,keys[i]));
20          }
21      }
22      return right+left; //注意 16 次迭代完了要交换左右两部分再拼接
23  }
```

辅助函数 strxor()定义如下，用于异或运算：

```
1   /**
2    * 二进制字符串异或
3    * @param s1 操作数 1
4    * @param s2 操作数 2
5    * @return s1 和 s2 异或的结果
6    */
7   private static String strxor(String s1,String s2) {
8       int len= s1.length()>s2.length()?s1.length():s2.length();
9       StringBuilder s=new StringBuilder();
```

```
    for(int i=0;i<len;i++) {
        if(i>=s1.length()||i>=s2.length()) {
            s.append("0");
        }
        else {
            if(s1.charAt(i)==s2.charAt(i)) {
                s.append("0");
            }
            else {
                s.append("1");
            }
        }
    }
    return s.toString();
}
```

16 次迭代的 f()函数定义如下：

```
/**
 * 16 次迭代的f 函数，包括E 运算、和子密钥的异或运算、S 运算、P 运算
 * @param right 本次迭代的右半部分原值
 * @param key 本次迭代的子密钥
 * @return 转换结果，之后和本次迭代的左半部分进行异或作为下一次迭代的右半部分
 */
private String f(String right,String key) {
    String addResult=strxor(ope_E(right),key);
    return ope_P(ope_S(addResult));
}
```

f()函数的辅助函数 ope_E()定义如下，用于扩展置换运算 EP：

```
/**
 * 选择扩展运算E
 * @param right 迭代中间结果的右半部分
 * @return 转换结果，之后和子密钥进行异或运算
 */
private String ope_E(String right) {
    String r="0"+right;
    StringBuilder s=new StringBuilder();
    for(int i=1;i<=48;i++) {
        s.append(r.charAt(des_E[i]));
    }
    return s.toString();
}
```

扩展置换运算 EP 对应的数据定义如下（最前面的-1 为占位）：

```
/** 选择扩展运算 E 盒 */
private static final byte[] des_E =
{ -1,32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,
        12,13,14,15,16,17,16,17,18,19,20,21,
        20,21,22,23,24,25,24,25,26,27,28,29,
        28,29,30,31,32,1
};
```

f()函数的辅助函数 ope_S()定义如下，用于选择压缩运算 S：

```
/**
 * 选择压缩运算 S，将 48 位的二进制字符串压缩成 32 位
 * @param right 迭代中间结果的右半部分和子密钥异或的结果
 * @return 32 位的转换结果，作为 P 运算的输入
 */
private String ope_S(String right) {
    String r="0"+right;
    StringBuilder s=new StringBuilder();
    int j=1;
    for(int i=1;i<=48;i+=6) {
        String temp=r.substring(i,i+6);
        s.append(toBinary(des_S[j][Integer.parseInt(temp,2)],4));
        j++;
    }
    return s.toString();
}
```

选择压缩运算 S 对应的数据定义由于太长而省略，可以在项目文件的源代码中查看。

f()函数的辅助函数 ope_P()定义如下，用于置换运算 P：

```
/**
 * 置换运算 P
 * @param right 进行 S 运算的结果
 * @return 转换结果，作为 16 次迭代中下一次迭代的初始值
 */
private String ope_P(String right) {
    String r="0"+right;
    StringBuilder s=new StringBuilder();
    for(int i=1;i<=32;i++) {
        s.append(r.charAt(des_P[i]));
    }
```

```
12      return s.toString();
13  }
```

置换运算 *P* 对应的数据定义如下（最前面的-1 为占位）：

```
1   /** 置换运算P */
2   private static final byte[] des_P = { -1,16,7,20,21, 29,12,28,17,
    1,15,23,26,
3           5,18,31,10, 2,8,24,14, 32,27,3,9,
4           9,13,30,6, 22,11,4,25
5   };
```

3.2 TCP 通信的实现

由于我使用了可视化界面，因此这里的与 TCP 通信相关的功能（除了发送消息以外）都使用了额外的线程，而不是集中在主界面线程，这样可以避免 TCP 的一些阻塞函数把主界面线程阻塞了。

客户端的相关实现在包 `com.deschat.client` 中，其中类 `DESClient` 是客户端的主界面，`private` 内部类 `ClientThread` 是客户端与服务器通信的线程类（是下面所说的**客户端线程**）；服务器的相关实现在 `com.deschat.server` 中，其中类 `DESServer` 是服务器的主界面，`private` 内部类 `ListenThread` 是服务器监听的线程类，`private` 内部类 `ServerThread` 是服务器与客户端通信的线程（一个客户端对应一个 `ServerThread` 实例，是下面所说的服务器线程）。建立客户端或服务器线程的代码在 `com.deschat.mainprg` 中的类 `Main`。

实现的大体效果是客户端能和服务器进行多对一通信（不同客户端的端口号不同且随机），但是每个客户端的通信密钥和服务器是一对一而且是随机生成的。在聊天之前服务器首先向客户端发送密钥，客户端首先接收服务器发来的密钥。这样可以保证不同客户端与服务器之间的通信的安全性。

客户端线程建立的代码如下，在类 `com.deschat.mainprg.Main` 中。需要先检查输入的服务器 IP 地址是否合法，通过判断是否和正则表达式匹配。只有输入的 IP 地址合法才建立套接字和客户端线程，并尝试与服务器连接。

```
1       if(option==0) { //建立客户端线程
2           Main.setLogFile("client");
```

```
3           String serverIP = JOptionPane.showInputDialog(null, "Please input
        the server IP address:\n", "Client Login",
        JOptionPane.QUESTION_MESSAGE);
4           String regex = "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
5                   +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
6                   +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
7                   +"(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$";
8           while(serverIP!=null&&!(serverIP.matches(regex))) { //检查输入的服
        务器IP地址格式是否合法
9               JOptionPane.showMessageDialog(null, "Invaild IP address, please
        input again!","Invalid",JOptionPane.ERROR_MESSAGE);
10              serverIP=JOptionPane.showInputDialog(null, "Please input the
        server IP address:\n", "Client Login", JOptionPane.QUESTION_MESSAGE);
11          }
12          System.out.println(serverIP);
13          if(serverIP!=null) {
14              try {
15                  //建立套接字并和服务器连接
16                  Socket s=new Socket(serverIP,2000);//随机分配空闲端口号
17                  s.setSoTimeout(0);//设置读取超时为0
18                  DESClient clientDlg=new DESClient(s,serverIP);//连接成功后
        建立客户端线程并打开对话框
19                  EventQueue.invokeLater(new Runnable() {
20                      @Override
21                      public void run() {
22                          clientDlg.setVisible(true);
23                      }
24                  });
25              } catch (UnknownHostException e) {
26                  // TODO 自动生成的 catch 块
27                  e.printStackTrace();
28                  System.out.println("UnknownHostException:
        "+e.getMessage());
29                  JOptionPane.showMessageDialog(null, "UnknownHostException:
        "+e.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
30              } catch (IOException e) {
31                  // TODO 自动生成的 catch 块
32                  e.printStackTrace();
33                  System.out.println("IOException: "+e.getMessage());
34                  JOptionPane.showMessageDialog(null, "IOException:
        "+e.getMessage(),"Error",JOptionPane.ERROR_MESSAGE);
35              }
36          }
37        }
```

服务器线程的建立较简单，也在类 com.deschat.mainprg.Main 中。直接打开对话框并建立监听线程即可：

```
1       else if(option==1) { //建立服务器线程
2           Main.setLogFile("server");
3           DESServer serverDlg=new DESServer();//打开对话框并建立监听线程
4           EventQueue.invokeLater(new Runnable() {
5               @Override
6                 public void run() {
7                   serverDlg.setVisible(true);
8                 }
9             });
10        }
```

服务器监听线程的代码如下，在类 com.deschat.server.DESServer.ListenThread 中。为了能够实现客户端与服务器的多对一通信和服务器总在线的效果，这里采取的是一直循环监听的方式，只要有客户端的请求就接受连接：

```
1       @Override
2       public void run() {
3           Socket socket=null;
4           try {
5               InetAddress address = InetAddress.getByName("127.0.0.1");
6               s=new ServerSocket(2000,50,address);//建立监听套接字，最大连接数
        50
7               System.out.println("Server Started");
8               updateListenState();
9               while(true) { //不断监听并接受客户端的连接请求
10                  socket=s.accept();//接受客户端的连接请求，没有连接请求则一直
        在这里阻塞
11                  socket.setSoTimeout(0);//设置读取超时为0
12                  count++;
13                  index++;
14                  InetAddress clientAddress=socket.getInetAddress();//获取客
        户端的连接
15                  addClient(socket,clientAddress);
16                  //新建并启动一个为新客户端服务的线程
17                  ServerThread thread=new
        ServerThread("client_"+index,clientAddress,socket);
18                  clients.put("client_"+index, thread);
19                  thread.start();//启动线程
20              }
```

```
        }
        catch (IOException e) {
            // TODO 自动生成的 catch 块
            e.printStackTrace();
            System.out.println("IOException: "+e.getMessage());
            //关闭与客户端通信的套接字 socket
            //socket.close();
        }
        finally {
            try {
                s.close();//关闭监听套接字
            } catch (IOException e) {
                // TODO 自动生成的 catch 块
                e.printStackTrace();
                System.out.println("IOException: "+e.getMessage());
            }
            finally {
                if(exit) {
                    System.exit(0);
                }
            }
        }
    }
```

客户端进程接收消息的代码如下，在类 com.deschat.client.DESClient.ClientThread 中。实现的主要逻辑是：接收服务器发来的 64 位通信密钥→循环接收消息并解密消息→显示解密后的消息原文→断线重连（隔一秒钟重新连接服务器一次，重新连接成功了将会重新获得新的密钥）→释放资源，关闭套接字。

```
    @Override
    public void run() {
        InputStream in=null;
        OutputStream out=null;
        InputStreamReader ir = null;
          BufferedReader br = null;
        while(true) {
            try {
                in = socket.getInputStream();//获取 TCP 连接的输入字节流
                out = socket.getOutputStream();
                //聊天前首先接收服务器发来的密钥
                if(!getKey) {
                    StringBuilder receiveKey=new StringBuilder();
                    int charcount=0;
```

```
15                  //逐字节解析密钥
16                  for (int c = in.read(); c!=-1; c = in.read()) {
17                      receiveKey.append((char)c);
18                      charcount++;
19                      if(charcount==8) { //服务器发来的前 8 个字节是密钥
20                          break;
21                      }
22                  }
23                  key=receiveKey.toString();
24                  getKey=true;
25                  updateInfo();
26              }
27              DES des=new DES(key);
28              String receiveMsg=null;
29              ir = new InputStreamReader(in);//将 TCP 字节流转为字符流
30              br = new BufferedReader(ir);//带缓冲区的文本读取
31              //接收消息并解密，不同消息的边界为换行符
32              while ((receiveMsg = br.readLine()) != null) {
33                  if(receiveMsg.equals("END")) break;
34                  System.out.println("Decry binary source: "+receiveMsg);
35                  String res=des.getResult(receiveMsg, false);//DES 解密
36                  System.out.println("Decry result: "+res);
37                  rcvLog(res);//将解密后的消息输出到对话框
38              }
39              //关闭连接的输入流和输出流
40              socket.shutdownInput();
41              socket.shutdownOutput();
42          }
43          catch(IOException e) {
44              e.printStackTrace();
45              System.out.println("IOException: "+e.getMessage());
46          }
47          finally {
48              try {
49                  //关闭流和套接字
50                  if(br!=null) {
51                      br.close();
52                  }
53                  if(ir!=null) {
54                      ir.close();
55                  }
56                  if(in!=null) {
57                      in.close();
58                  }
```

```
59                      if(out!=null) {
60                          out.close();
61                      }
62                      if(socket!=null) {
63                          if(!socket.isClosed()) {
64                              socket.close();
65                          }
66                      }
67                  }
68                  catch(IOException e1) {
69                      e1.printStackTrace();
70                      System.out.println("IOException: "+e1.getMessage());
71                  }
72                  finally {
73                      if(exit) {
74                          System.exit(0);
75                      }
76                      else {
77                          //断线重连，隔一秒尝试连接一次
78                          //重新连接成功以后会重新获得密钥
79                          reconnecting();
80                          try {
81                              Thread.sleep(1000);
82                          } catch (InterruptedException e) {
83                              // TODO 自动生成的 catch 块
84                              e.printStackTrace();
85                              System.out.println("InterruptedException:
      "+e.getMessage());
86                          }
87                          try {
88                              //重新建立套接字并和服务器连接，保持客户端原端口不
变
89                              socket=new Socket(serverIP,2000,address,port);
90                              socket.setSoTimeout(0);//设置读取超时为0
91                              getKey=false;
92                          } catch (IOException e) {
93                              // TODO 自动生成的 catch 块
94                              e.printStackTrace();
95                              System.out.println("IOException:
      "+e.getMessage());
96                          }
97                      }
98                  }
99              }
```

```
100         }
101     }
```

服务器接收消息的代码如下，在类 com.deschat.server.DESServer.ServerThread 中。实现的主要逻辑是：向客户端发送的 64 位通信密钥→循环接收消息并解密消息→显示解密后的消息原文→释放资源，关闭套接字。

实现逻辑和客户端接收消息的主要的不同是没有断线重连的部分，因为我认为服务器应该是总在线且是被动接受连接的，客户端断线视为客户端主动结束通信，服务器断线是被动的意外宕机。

```
1       @Override
2       public void run() {
3           InputStream in = null;
4           OutputStream out = null;
5           InputStreamReader ir = null;
6             BufferedReader br = null;
7           try {
8               in = socket.getInputStream();//获取 TCP 连接的输入字节流
9               out = socket.getOutputStream();
10              //聊天前首先向客户端发送密钥
11              if(!makeKey) {
12                  out.write(key.getBytes());
13                  out.flush();//清空缓冲区数据
14                  makeKey=true;
15                  gotKey();
16              }
17              DES des=new DES(key);
18              String receiveMsg = null;
19              ir = new InputStreamReader(in);//将 TCP 字节流转为字符流
20                br = new BufferedReader(ir);//带缓冲区的文本读取
21                //接收消息并解密，不同消息的边界为换行符
22                while ((receiveMsg = br.readLine()) != null) {
23                  if(receiveMsg.equals("END")) break;
24                  System.out.println("Decry binary source: "+receiveMsg);
25                  String res=des.getResult(receiveMsg, false);//DES 解密
26                  System.out.println("Decry result: "+res);
27                  rcvLog(res);//将解密后的消息输出到对话框
28                }
29                //关闭连接的输入流和输出流
30                socket.shutdownInput();
31                socket.shutdownOutput();
32          }
```

```
33          //关闭套接字和释放资源的部分省略
```

服务器随机生成密钥的代码如下，在类 com.deschat.server.DESServer.ServerThread 中。这样生成的密钥虽然有可能会和其他客户端的密钥重复，但是重复的概率极小，而且服务器被设置成最多只能同时连接 50 个客户端（否则会抛出异常）。

```
1       /**
2        * 生成随机密钥
3        */
4       private void genRandomKey() {
5           StringBuilder s=new StringBuilder();
6           String alpha="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7           Random r = new Random();
8           for(int i=0;i<8;i++) {
9               s.append(alpha.charAt(r.nextInt(26)));
10          }
11          key=s.toString();
12      }
```

发送消息的代码如下，由于这里服务器和客户端的逻辑差不多，因此只展示服务器的代码，在类 com.deschat.server.DESServer 中的 ActionListener 的匿名内部类中。实现的逻辑是：选择要发送的目标客户端（客户端相关代码没有这一步）→从输入对话框获取要发送的消息→加密消息并以字节的形式发送消息。我在这里把发送消息的功能放在了主界面线程，获取对方的套接字并通过“发送”按钮的监听器实现。这样做主要是因为要发送的消息要点击“发送”按钮才能发送，而且发送的相关函数不用考虑阻塞主界面线程的问题。

```
1       sendMsg.addActionListener(new ActionListener() {
2           public void actionPerformed(ActionEvent arg0) {
3               if(editMsg.getText().length()==0) {
4                   JOptionPane.showMessageDialog(null, "消息内容不能为空!
        ","Warning",JOptionPane.WARNING_MESSAGE);
5                   return;
6               }
7               String
        select=clientListComboBox.getSelectedItem().toString();//选择要发送的
        客户端
8               ServerThread thread=clients.get(select);//获取与该客户端通信的
        线程和套接字
9               if(thread==null) {
10                  JOptionPane.showMessageDialog(null, "此客户端不存在!
        ","Error",JOptionPane.WARNING_MESSAGE);
```

```
                return;
            }
            Socket s=thread.getSocket();
            try {
                OutputStream out;
                String response,temp;
                DES des=new DES(thread.getDESKey());
                out = s.getOutputStream();//获取 TCP 连接的输出字节流
                response=editMsg.getText().trim();//从输入框获取要发送的原
文
                System.out.println("Encry source: "+response);
                temp=des.getResult(response, true);//DES 加密
                System.out.println("Encry binary result: "+temp);
                temp+="\n";//消息边界为换行符（不需要加密），不加边界会一直阻
塞
                out.write(temp.getBytes());//以字节形式发送加密后的消息
                out.flush();//清空缓冲区数据
                workLog.append("server -> "+select+": \n"+response+"\n");
                editMsg.setText("");//发送后清空输入框
            }
            catch(IOException e) {
                e.printStackTrace();
                System.out.println("IOException: "+e.getMessage());
            }
        }
    });
```

客户端退出通信线程的代码如下，在类 com.deschat.client.DESClient 中的 WindowAdapter 的匿名内部类中。主要通过向服务器发送终止符”END”实现（注意这个消息是不需要加密和解密的）。客户端可以点击窗口的×退出客户端的通信线程。这里没有服务器退出线程的相关代码，还是一样，认为客户端发起连接和关闭连接的行为是主动的，服务器的相关行为是被动的。

```
    @Override
    public void windowClosing(WindowEvent e) {
        exit=true;
        Socket s=client.getSocket();
        //客户端终止与服务器通信的方式是点击窗口的×并向服务器发送 END 结束符，
释放资源，退出线程
        try {
            OutputStream out;
            out = s.getOutputStream();
            String temp="END";
            temp+="\n";
```

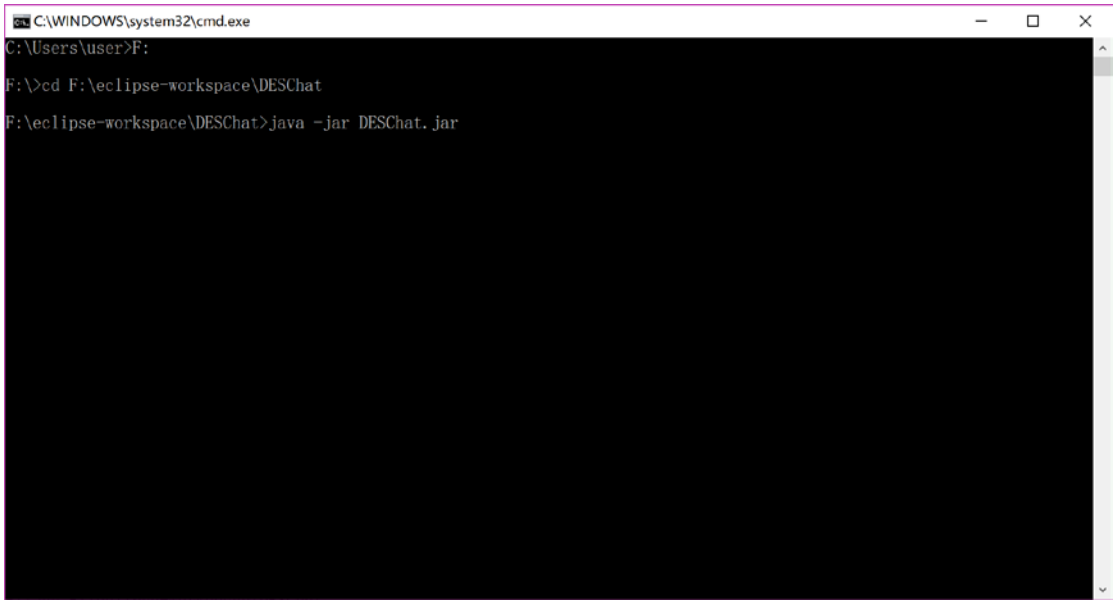
```
11              out.write(temp.getBytes());
12              out.flush();//清空缓冲区数据
13          }
14          catch(IOException e1) {
15              e1.printStackTrace();
16              System.out.println("IOException: "+e1.getMessage());
17          }
18      }
```

3.3 实验结果展示

3.3.1 运行程序的方法

以下的图片仅为演示建立线程的方法，不作为实验结果展示。

打开终端（Windows 或 Linux 终端均可，这里使用 Windows 的 cmd 演示），定位到 `DESChat.jar` 所在的文件夹，输入命令 `java -jar DESChat.jar` 启动程序：



程序启动后，看到以下对话框，点“Client”建立客户端线程，点“Server”则建立服务器线程。一定要先建立服务器线程再建立客户端线程，因为建立客户端线程前会尝试连接服务器线程，连接成功了才能建立客户端线程：

点击“Server”看到以下对话框则服务器线程建立成功（如果不成功则先关闭占用端口号 2000 的线程再尝试，点击窗口的×结束此线程及退出程序）：

点击“Client”之后要先输入服务器的 IP 地址（127.0.0.1），一定要输入正确（如果 IP 地址格式正确但不是服务器的 IP 地址则无法连接，也无法建立客户端线程）：

看到以下的对话框则成功建立客户端线程（点击窗口的×结束此线程及退出程

序）：

3.3.2 多线程聊天测试

以下测试按照上述的方法和下面的演示流程建立和退出线程，篇幅所限，仅展示该线程的最终输出结果。

程序演示的流程如下：

建立服务器线程

建立客户端线程 1

客户端 1 向服务器发送：Hello~I'm client~

服务器向客户端 1 发送：Hello~I'm server~

建立客户端线程 2

建立客户端线程 3

客户端 2 向服务器发送：Hello server

客户端线程 1 退出

服务器向客户端 2 发送：Hello client

客户端线程 2 退出

建立客户端线程 4

客户端 3 向服务器发送：Hi, this is client

服务器向客户端 3 发送：Ok, this is server

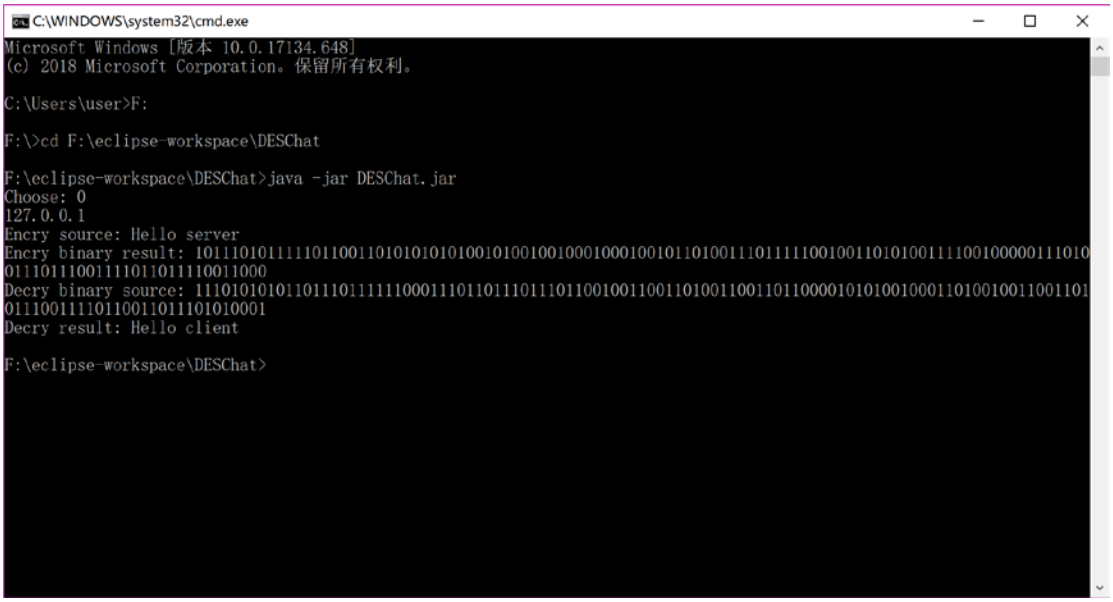
客户端线程 3 退出
客户端 4 向服务器发送：你好，server，哈哈！
服务器向客户端 4 发送：client，你好，呵呵！！
服务器线程退出
重新启动服务器线程
客户端 4 向服务器发送：server，哈哈，你又来啦
服务器向客户端 4 发送：是的，我又来了，client
客户端线程 4 退出
服务器线程退出

客户端 1 最终的可视化界面如下：

客户端 1 最终的终端输出如下：

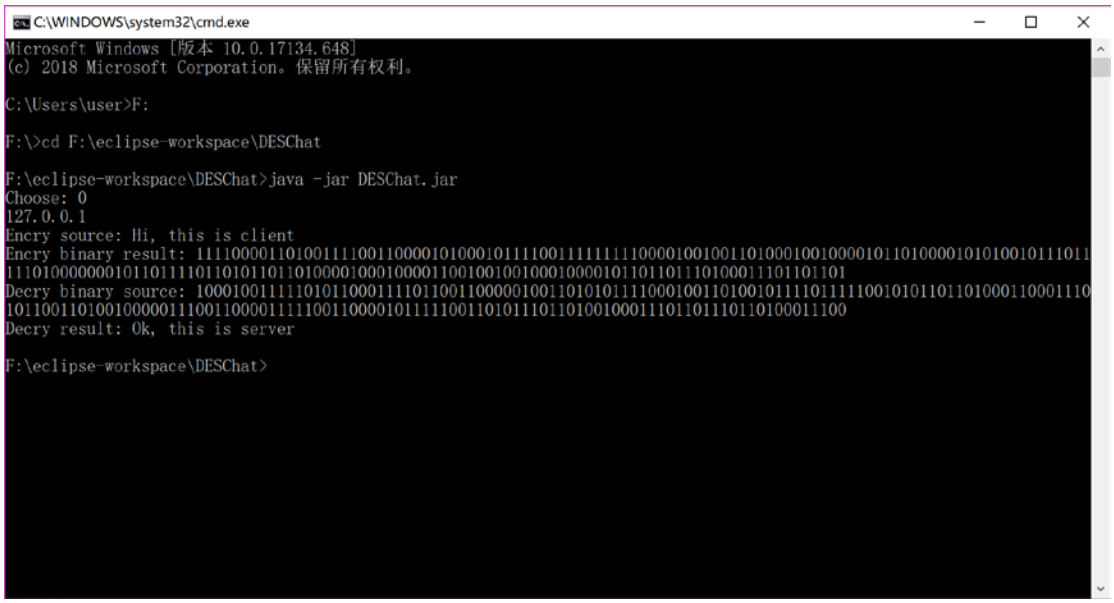
客户端 2 最终的可视化界面如下：

客户端 2 最终的终端输出如下：



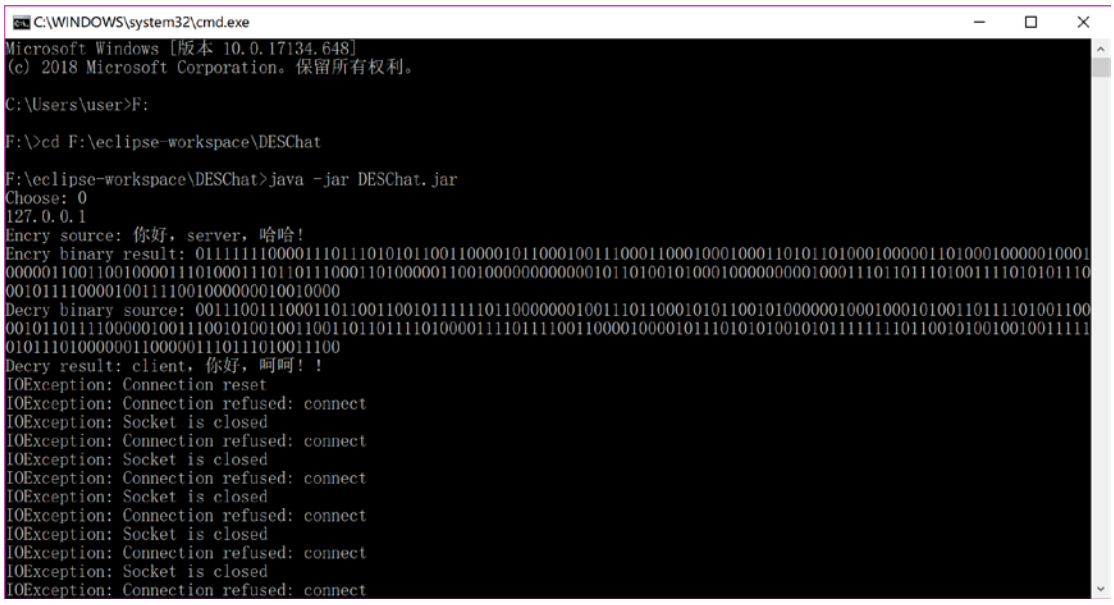
客户端 3 最终的可视化界面如下：

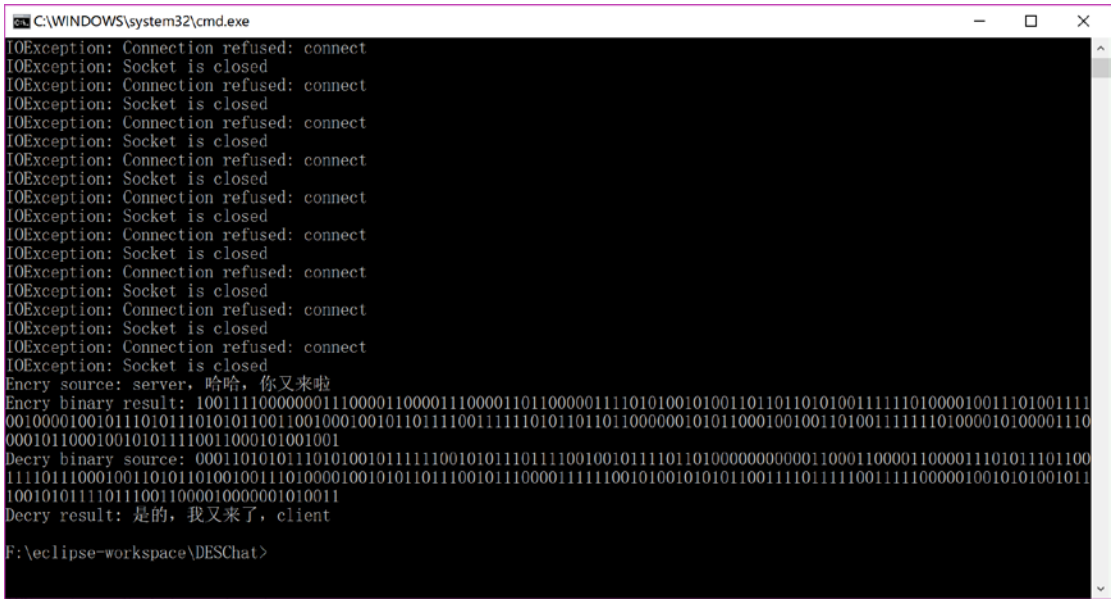
客户端 3 最终的终端输出如下：



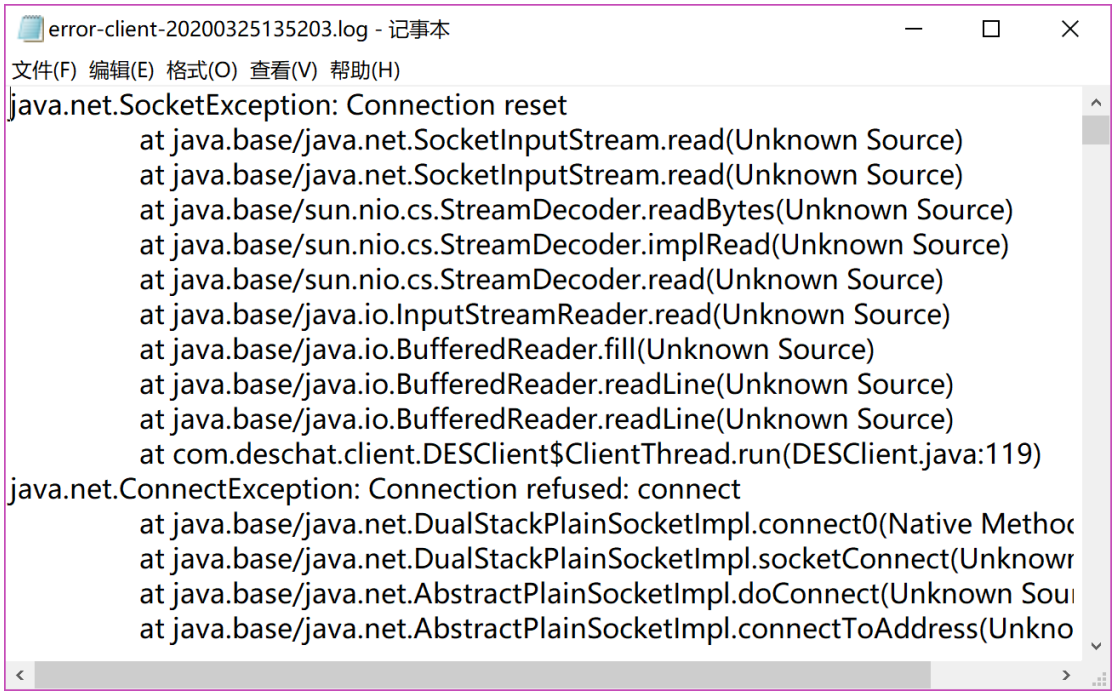
客户端 4 最终的可视化界面如下：

客户端 4 最终的终端输出如下：



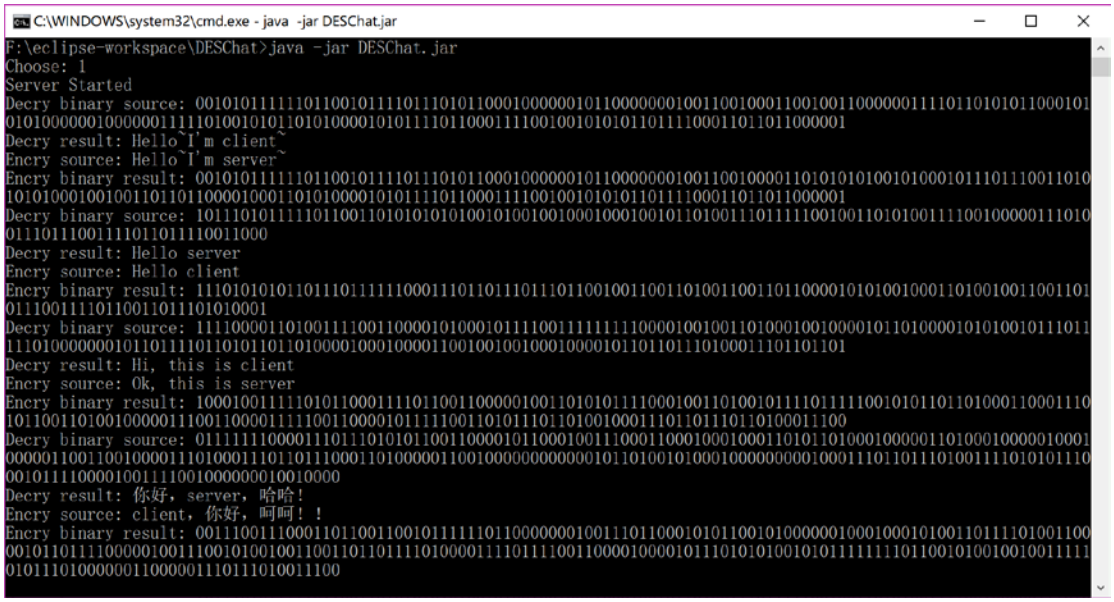


另外，打开客户端 4 对应的错误日志可以看到服务器断线期间的错误堆栈输出（不管是服务器线程还是客户端线程创建的时候都会有日志输出，这些日志可以删掉）：

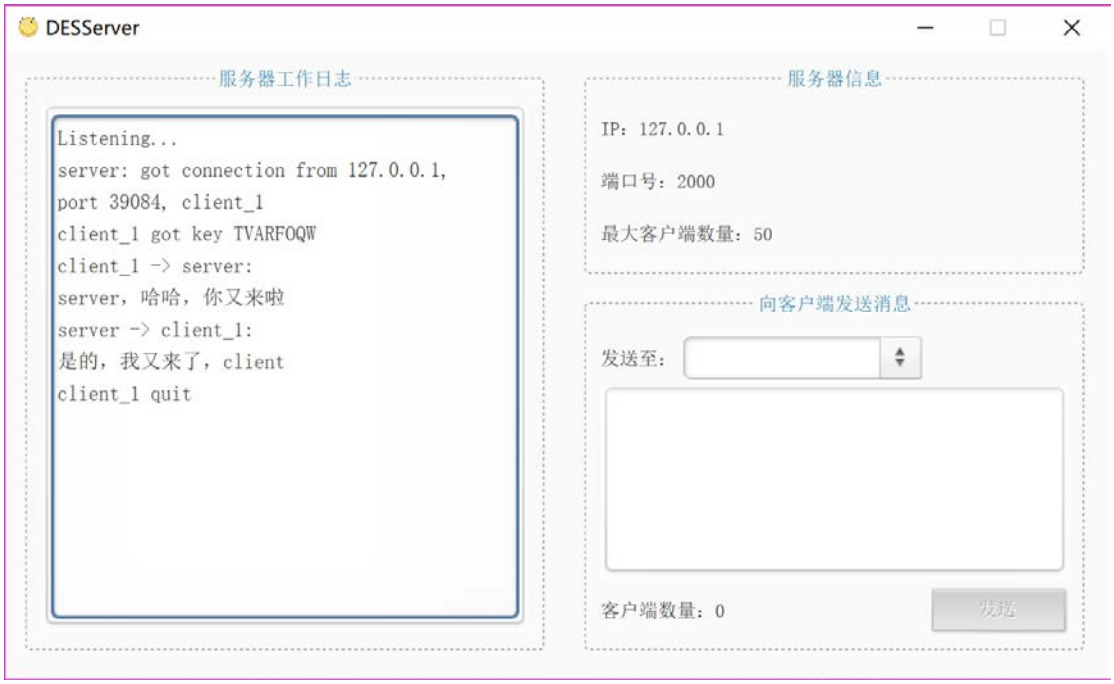


第一次开启服务器的最终可视化界面如下：

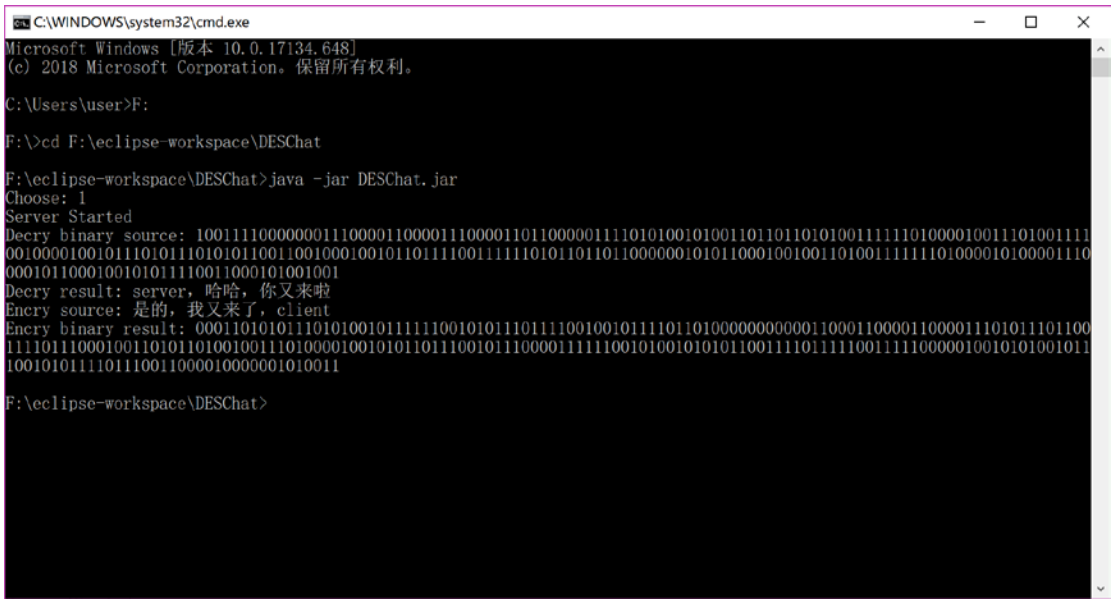
第一次开启服务器的最终的终端输出如下：



第二次开启服务器的最终可视化界面如下（由于是第二次开启服务器，客户端 4 在这里的编号又重新开始计算了，所以是 `client_1`）：



第二次开启服务器的最终的终端输出如下：



四、实验遇到的问题及其解决方法

至于遇到的问题，最大的问题居然不是 DES 算法实现的问题，而是对 Java 的字符编码和对 Java 当中的 `char`、`String` 类型的特性不熟悉的问题而导致的不必要出现的问题。

其中一个大问题是因为加密解密得到的字符串编码不一致而导致客户端和服务器无法顺利地使用 TCP 进行通信（向对方发送出去密文，对方解密后出现乱码）。排查了好久才发现这是加密和解密采用的字符集编码不一致而导致的问题，不是算法实现的问题。在写相关代码的时候我忘记了很重要的一点就是 Java 的 `char` 类型是 2 个字节的（不是和 C/C++一样是 1 个字节！），而且 utf-8 是变长编码，可以是 1 个字节或者多个字节（例如 7 位的 ASCII 码在 utf-8 中只占 1 个字节，汉字在 utf-8 中占 2 个字节甚至 3 个或以上的字节）。

起初我是将明文的字符都按照 1 个字节去解释，然后将加密得到的密文也编码成字符串（这是造成了对方解码得到的信息是乱码的根源），在使用了 `getBytes()` 函数（无参数默认就使用 ANSI，在我这里也就是 GBK 编码）将字符流转换成字节流之后，这个字节流所对应的编码就变了（原先是 8 bit 编码，现在是 GBK），有些无意义字符的编码从 1 个字节变成了 2 个字节，导致了对方实际收到的密文内容的二进制串变了。然后我就把 DES 加密得到的密文输出和解密的密文输入都改成了 `byte` 类型的数组形式，又发现由于程序错误识别了消息分隔符（`'\n'`）

导致了解密失败（因为传送的 byte 数组的中间的某个字节的值正好等于'\n'的 ASCII 码，就把这里当成了消息分隔符）。最后我就只能采取最折中的方式就是把 DES 加密得到的密文输出和解密的密文输入改成了二进制的字符串（即发送的内容只有字符'0'和'1'还有消息分隔符'\n'，消息分隔符会在接收消息的时候自动过滤掉），这样就没有这些连发送英文都会乱码的问题了，也不会错误判断消息分隔符了。

起初我是将明文的字符都按照 1 个字节去解释，这样还会带来的后果就是只要加密原文是中文就会乱码（这种方式放在 C/C++可能不会乱码，因为 C/C++的 char 类型就只占 1 个字节）。这个问题我起初是将客户端和服务器发出的消息先用 Base64 编码（见源代码 Base64Tool.java）再用 DES 加密发送出去，收到的消息先进行 DES 解密再用 Base64 解码得到原文。但是这样总有偷梁换柱的感觉。最后的解决方案就是在加密之前先把原文转为 utf-8 编码的字节数组，再把这个字节数组转为二进制字符串；解密之后转换为明文的时候先把二进制字符串转为字节数组，再把这个字节数组按照 utf-8 的编码转为相应的字符串。这样对明文进行加密和解密得到明文所采用的编码就统一了，从而就不会遇到中文就乱码了。当然，初始密钥的字符还是按照 ASCII 编码解释的，如果不是则抛出异常。

最后还有一个细节问题就是，由于 Java 的字符串非但不像 C/C++一样以'\0'作终止符，而且 Java 的字符串没有终止符，且有效字符可以包含数量不限的'\0'。这样带来的问题就是由于 DES 算法会对不满 64 bit 的原文进行补 0 操作，然后解密之后就会造成明文的最后多出很多个'\0'！如果使用 C/C++也不会有这种问题。这样可能还会造成一些误差。最终的解决方案是使用正则表达式过滤字符串结尾的'\0'。

以上的这些问题对应最终的解决方案（代码）就是：

```
1   /**
2    * 生成二进制文本字符串
3    * @param src 要转换的字符串
4    * @return 转换结果
5    */
6   private static String genBinaryMsg(String src) {
7       byte[] b=null;
8       try {
9           b=src.getBytes("utf-8");
10      } catch (UnsupportedEncodingException e) {
11          // TODO 自动生成的 catch 块
```

```
            e.printStackTrace();
        }
        StringBuilder s=new StringBuilder();
        for(int i=0;i<b.length;i++) {
            s.append(toBinary(b[i],8));
        }
        return s.toString();
    }
```

```
    /**
     * 二进制字符串转普通字符串
     * @param src 要转换的字符串
     * @return 转换结果
     */
    private String binary2String(String src) {
        List<Byte> list=new ArrayList<Byte>();
        for(int i=0;i<src.length();i+=8) {
            list.add((byte)(Integer.parseInt(src.substring(i,i+8),2)));
        }
        byte[] b=Byte2byte(list.toArray(new Byte[list.size()]));
        String s=null;
        try {
            s = new String(b,"utf-8");
        } catch (UnsupportedEncodingException e) {
            // TODO 自动生成的 catch 块
            e.printStackTrace();
        }
        if(mode==true) {
            return s;
        }
        else { //如果是解密，则必须去掉尾部用于凑整64位的\0，Java是把\0算作字符串长度的
            return s.replaceAll("\0+$", "");
        }
```

五、实验结论

1. 发送的消息原文使用这里实现的 DES 加密，收到的消息也使用这里实现的 DES 解密，解密得到的文本和原文一致，说明这里实现的 DES 是正确的。
2. 通过这次编程，更加深入地理解了 DES 算法的原理，之前看 DES 算法的描述各种看不懂。

3. 熟悉了使用 Java 进行网络编程的方法。上个学期的计算机网络课程中，我一般都使用 C++的 CAsyncSocket 类实现，它是异步非阻塞的 socket。使用同步阻塞 socket 编程的思想以前我并没有深入了解过。
4. 熟悉了使用多线程的编程方法。
5. 阴差阳错地了解了一些 Java 的 `char` 和 `String` 类型以及编码的一些知识。