目录

[第一章 网络安全技术的特点 2](#_Toc407361109)

[1.1 网络安全与现代社会安全的关系 2](#_Toc407361110)

[1.2 网络安全与信息安全的关系 3](#_Toc407361111)

[1.3 网络安全与网络新技术的关系 3](#_Toc407361112)

[1.4 网络安全与密码学的关系 4](#_Toc407361113)

[第二章 基于DES加密的TCP聊天程序 4](#_Toc407361114)

[2.1 目的与需求 4](#_Toc407361115)

[2.2 DES算法的基本内容 5](#_Toc407361116)

[2.2.1 初始置换IP 5](#_Toc407361117)

[2.2.2 逆初始置换 5](#_Toc407361118)

[2.2.3 16圈迭代 6](#_Toc407361119)

[2.2.4 子密钥生成 11](#_Toc407361120)

[2.3 TCP协议 11](#_Toc407361121)

[2.4 套接字 12](#_Toc407361122)

[2.5 TCP通信的相关函数 12](#_Toc407361123)

[2.5.1 socket函数 12](#_Toc407361124)

[2.5.2 bind函数 12](#_Toc407361125)

[2.5.3 listen函数 13](#_Toc407361126)

[2.5.4 accept函数 13](#_Toc407361127)

[2.5.5 connect函数 13](#_Toc407361128)

[2.5.6 write函数 13](#_Toc407361129)

[2.5.7 read函数 14](#_Toc407361130)

[2.5.8 send函数 14](#_Toc407361131)

[第三章 DES加密解密设计 14](#_Toc407361132)

[3.1 DES类的定义 14](#_Toc407361133)

[3.2 DES算法中用到的静态数组 15](#_Toc407361134)

[3.3 DES密钥生成 18](#_Toc407361135)

[3.4 DES加密运算 19](#_Toc407361136)

[3.5 封装DES加密函数 23](#_Toc407361137)

[第四章 基于TCP的聊天功能模块设计 24](#_Toc407361138)

[4.1 建立连接 24](#_Toc407361139)

[4.2 多进程全双工聊天程序分析 26](#_Toc407361140)

# 第一章 网络安全技术的特点

## 1.1 网络安全与现代社会安全的关系

生活在现实世界的人类创造了网络虚拟社会的繁荣，同时也造成了网络虚拟社会的问题。现实世界中真善美的东西，网络的虚拟社会都有。同样，现实社会中丑陋的东西，网络的虚拟社会一般也会有，只是表现形式不一样。如果透过复杂的技术术语和计算机屏幕，人们会发现：计算机网络的虚拟社会和现实社会之间，在很多方面都存在着“对应”关系。现实社会中人与人在交往中形成了复杂的社会与经济关系，在网络社会中，这些社会与经济关系以数字化的形式延续着。

网络安全是现实社会安全的反映。网络安全问题实际上是个社会问题，光靠技术来解决这些问题是不可能的。网络安全是一个系统的社会工程，它涉及技术、政策、道德与法律法规等多方面。

## 1.2 网络安全与信息安全的关系

应用是网络存在和发展的理由。所有的信息系统与现代服务业都是建立在计算机网络与Internet环境之中的。正是由于这个原因，可以说网络应用系统的安全都是建立在计算机网络安全的基础之上的。

用户的各种信息被保存在不同类型的应用系统之中，这些应用系统都是建立在不同的计算机系统之中的。计算机系统包括硬件、操作系统、数据库系统等，它们是保证各类信息系统正常运行的基础。而运行信息系统的大型服务器或服务器集群及用户的个人计算机都是以固定或移动的方式接入到计算机网络与Internet中的。任何一种网络功能的服务实现都需要通过网络在不同的计算机系统之间多次进行数据与协议信息交换。

病毒、木马、蠕虫、脚本攻击代码等恶意代码利用E-mail、FTP与Web系统进行传播，网络攻击、网络诱骗、信息窃取也都是在网络环境中进行的。网络安全是信息系统安全的基础，不能保证网络的安全性，信息系统的安全性就无从谈起。因此，网络安全研究是信息安全研究的重要组成部分，也是信息安全研究的基础。

## 1.3 网络安全与网络新技术的关系

按照正常人的思维方式，一位技术人员在研究和开发一种基于网络的新的应用技术与系统时，只会想到这种应用可以给人们的生活和工作带来什么样的好处和乐趣，一般不会想到黑客或居心不良的人会利用这种技术做什么坏事。而黑客恰恰是一类逆向思维和不按正常规律办事的人，他们不遵守正常人所遵循的道德规范，“Everything over IP, IP over everything.”说明了计算机网络技术的成功，但是它所带来的问题也是网络技术人员始料未及的。P2P是一种十分有价值的网络应用模式，但是P2P除了可以方面信息共享之外，同时也给恶意代码的传播提供了一种新的途径。手机病毒的出现与无线射频标识RFID芯片可能感染病毒的研究结果公布，表明移动设备将成为黑客和恶意软件编写者下一个主攻的目标。

网络技术不是在真空之中，计算机网络是要提供给全世界的用户使用的，网络技术人员在研究和开发一种新的基于网络的应用技术与系统时，必须面对这样一个复杂的局面，成功的网络应用技术与成功的应用系统的标志是功能性与安全性的统一。网络安全问题不应该简单地认为是从事网络安全技术工程师的事，也是每位信息技术领域的工程师与管理人员需要共同面对的问题。

## 1.4 网络安全与密码学的关系

密码学是信息安全研究的重要工具，密码学在网络安全中有很多重要的应用，但是网络安全涵盖的问题远远超出了密码学涉及的范围。人们对密码学与网络安全的关系的认识有一个过程，这个问题可以用美国著名的密码学专家Bruce Schneier在《Secrets and Lies:Digital Security in a Networked World》一书的前言中讲述的观点来说。Schneier说过：我描述了一个数学的乌托邦：密码算法能将你最深的秘密保持数千年。但是，他现在认为：“事实并非如此，密码学并不能做那么多的事。”密码学并非存在于真空之中。密码学是数学的一个分支，它涉及数学、公式与逻辑。数学是完美的，而现实社会却无法用数学精确地描述。数学是精确的和遵循逻辑规律的，而计算机和网络安全涉及的是人所不知道的事，人与人之间的关系以及人与机器之间的关系。

密码学是研究网络安全所必需的一个重要的工具和方法，但是网络安全研究涉及的问题要广泛得多。

# 第二章 基于DES加密的TCP聊天程序

## 2.1 目的与需求

DES(Data Encryption Standard)算法是一种典型的对称分组加密算法，也是应用密码学中最基本的加密算法之一，目前广泛应用于网络通信加密、数据存储加密、口令与访问控制系统之中。

设计基于DES加密的TCP聊天程序的需求主要在于：在Linux环境下利用socket编写一个TCP聊天程序，网络传输中的数据通过DES算法进行加密。

## 2.2 DES算法的基本内容

DES算法包括初始置换IP、逆初始置换、16圈迭代以及子密钥生成算法。

### 2.2.1 初始置换IP

将64bit的明文重新排列，而后分成左右两块，每块32bit，分别用和表示，IP置换表如表1所示。通过对该表进行观察可以发现其中相邻两列的元素位置号数相差8，前32个元素均为偶数号码，后32个均为奇数号码，这样的置换相当于将明文的各字节按列写出，各列经过偶采样置换后，再对其进行逆序排列，将阵中元素按行读出以便构成置换的输出。

表1 IP置换表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

### 2.2.2 逆初始置换

在16圈迭代之后，将左右两端合并为64bit，进行逆初始置换，得到输出的64bit密文，如表2所示。

表2 逆初始置换表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

输出的64bit为表中元素按行读出的结果。

IP和的输入与输出是已知的一一对应关系，它们的作用在于打乱原来输入的ASCII码顺序，并将原来明文的校验位，，……，变为IP输出的一个字节。

### 2.2.3 16圈迭代

16圈迭代是DES算法的核心部分。将经过IP置换后的数据分成32bit的左右两段，进行16圈迭代，每轮迭代只对右边的32bit进行一系列的加密变换，在一轮加密变换结束时，将左边的32bit与右边进行异或后得到的32bit，作为下一轮时右边的段，并将这轮迭代中的右边段未经任何加密变换时的初始值直接作为下一轮迭代时左边的段，这需要在每轮迭代开始时，先将右边段保存一个副本，以便在该轮迭代结束时，将该副本直接赋值给下一轮迭代的左边段。在每轮迭代时，右边的数据段要经过的加密运算包括选择扩展运算E、密钥加运算、选择压缩运算S，这些变换合称为f函数。

1. 选择扩展运算

选择扩展运算（也称为E盒）的目的是将输入的右边32bit扩展成为48bit输出，其变换表如表3所示。置换结果按行输出的结果即为密钥加运算48bit的输入。

表3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32 |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

1. 密钥加运算

密钥加运算，是将选择扩展运算输出的48bit作为输入，与48bit的子密钥进行异或运算，异或的结果作为选择压缩运算（S盒）的输入。

1. 选择压缩运算

选择压缩运算（S盒）是DES算法中唯一的非线性部分，它是一个查表运算，共有8张非线性的变换表，如表4至表11，每张表的输入为64bit，输出为64bit。在查表之前，将密钥加运算的输出作为48bit的输入，将其分为8组，每组6bit，分别进入8个S盒进行运算，得出32bit的输出结果作为置换运算的输入。

表4 选择压缩运算变换表 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0xe | 0x0 | 0x4 | 0xf | 0xd | 0x7 | 0x1 | 0x4 |
| 9-16 | 0x2 | 0xe | 0xf | 0x2 | 0xb | 0xd | 0xb | 0xe |
| 17-24 | 0x3 | 0xa | 0xa | 0x6 | 0x6 | 0xc | 0xc | 0xb |
| 25-32 | 0x5 | 0x9 | 0x9 | 0x5 | 0x0 | 0x3 | 0x7 | 0x8 |
| 33-40 | 0x4 | 0xf | 0x1 | 0xc | 0xe | 0x8 | 0x8 | 0x2 |
| 41-48 | 0xd | 0x4 | 0x6 | 0x9 | 0x2 | 0x1 | 0xb | 0x7 |
| 49-56 | 0xf | 0x5 | 0xc | 0xb | 0x9 | 0x3 | 0x7 | 0xe |
| 57-64 | 0x3 | 0xa | 0xa | 0x0 | 0x5 | 0x6 | 0x0 | 0xd |

表5 选择压缩运算变换表 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0xf | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x6 | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x9 | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0xc | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x0 | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0xa | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x5 | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x9 | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表6选择压缩运算变换表 3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表7选择压缩运算变换表 4

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表8选择压缩运算变换表 5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表9选择压缩运算变换表 6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表10选择压缩运算变换表 7

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

表11选择压缩运算变换表 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1-8 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 9-16 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 17-24 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 25-32 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 33-40 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 41-48 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 49-56 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |
| 57-64 | 0x | 0x | 0x | 0x | 0x | 0x | 0x | 0x |

S盒算法流程如下。假设输入的48bit为，需要将其转换为32bit值，先把输入值视为由8个6bit的二进制块组成，如下所示。

a = =

b = =

c = =

d = =

e = =

f = =

g = =

h = =

其中a、b、···、h都是6位，故其十进制范围为0~63，将转换后的十进制数值加一与对应表中的十六进制数值对应，查表得到8个4bit的结果，将其串在一起的32bit结果作为置换运算的输入。其中，a对应表4，b对应表5，以此类推。

1. 置换运算

置换运算P是一个32bit的换位运算，对选择压缩运算输出的32bit数据按表12进行换位。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 7 | 20 | 21 | 29 | 12 | 28 | 17 |
| 1 | 15 | 23 | 26 | 5 | 18 | 31 | 10 |
| 2 | 8 | 24 | 14 | 32 | 27 | 3 | 9 |
| 19 | 13 | 30 | 6 | 22 | 11 | 4 | 25 |

至此，最终获得的32bit数据，即为此轮迭代的输出。此输出与左边的32bit进行异或作为下一轮的右边段，进行加密运算前的原始的右边段作为下一轮的左边段。

### 2.2.4 子密钥生成

64bit初始密钥经过置换选择PC-1、循环左移运算LS、置换选择PC-2，产生16圈迭代所用到的子密钥。初始密钥的第8、16、24、32、40、48、56、64位是奇偶校验位，其余56位为有效位。

## 2.3 TCP协议

TCP协议是一种面向连接、面向字节流的可靠传输层协议。两台采用TCP协议通信的计算机首先要建立TCP连接。TCP协议以它自己的方式缓存数据，缓存过程对程序员和用户是透明的。TCP协议采用piggybacking ACK的方法，允许双方同时发送数据。TCP规定了报文段的最大报文段长度（MSS），默认的MSS值为536个字节。

## 2.4 套接字

套接字（socket）是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。它把复杂的TCP/IP协议族隐藏在socket接口的背后，通过调用简单的socket函数完成特定协议的数据传输任务。TCP/IP提供了3中类型的套接字。

1. 流式套接字（SOCK\_STREAM）

流式套接字是面向连接的，可靠的数据传输服务，可保证无差错、无重复且按发送顺序提交给接收方。流式套接字在传输层使用TCP协议。

1. 数据报套接字（SOCK\_DGRAM）

数据报套接字提供无连接服务，数据以独立的数据报形式被传送，并且在传输过程中没有差错控制和流量控制，数据可能丢失、重复或者乱序。数据报套接字在传输层使用UDP协议。

1. 原始套接字（SOCK\_RAW）

原始套接字允许对较低层协议（如网络层的IP、ICMP）直接进行访问。用于实现自己定制的协议或者对数据报作较低层的控制。

## 2.5 TCP通信的相关函数

Linux系统通过socket来进行网络编程。网络程序通过socket和其他几个函数的调用，会返回一个通信的文件描述符，程序员可以将这个描述符看成普通的文件描述符来操作，通过对描述符的读写操作可以实现网络中计算机之间的数据传输。这充分体现出Linux操作系统的设备无关性的优点。

### 2.5.1 socket函数

int socket(int domain, int type, int protocol);

该函数用于创建通信的套接字，并返回该套接字的文件描述符。

### 2.5.2 bind函数

int bind(int sockfd, const struct sockaddr\* my\_addr, socklen\_t addrlen);

该函数用于将套接字与指定端口的相连。

sockaddr结构体的定义如下：

struct sockaddr

{

unsigned short sa\_family;

char sa\_data[4];

};

不过由于系统的兼容性，一般不用这个头文件，而使用另外一个结构（struct sockaddr\_in）来代替。sockaddr\_in的定义如下：

struct sockaddr\_in

{

unsigned short sin\_family;

unsigned short int sin\_port;

struct in\_addr sin\_addr;

unsigned char sin\_zero[8];

};

### 2.5.3 listen函数

int listen(int sockfd, int backlog);

该函数用于实现服务器等待客户端请求的功能。

### 2.5.4 accept函数

int accept(int sockfd, struct sockaddr\* addr, socklen\_t\* addrlen);

该函数用于处于监听状态的服务器，在获得客户机连接请求后，会将其放置在等待队列中，当系统空闲时，服务器用该函数接受客户机的连接请求。

### 2.5.5 connect函数

int connect(int sockfd, const struct sockaddr\* serv\_addr, socklen\_t\* addrlen);

该函数用于客户端向服务器发出连接请求。

### 2.5.6 write函数

ssize\_t write(int fd, const void\* buf, size\_t nbytes);

该函数用于服务器和客户端建立连接后，将buf中nbytes字节的内容写入文件描述符。

### 2.5.7 read函数

ssize\_t read(int fd, void\* buf, size\_t nbytes);

该函数用于从文件描述符fd中读取内容。

### 2.5.8 send函数

ssize\_t send(int s, const void\* buf, size\_t len, int flags);

该函数的作用基本同write函数相同，用于将信息发送到指定的套接字文件描述符中，其功能比write函数更为全面。

2.5.9 recv函数

ssize\_t recv(int s, void\* buf, size\_t len, int flags);

该函数的作用基本同read函数相同，用于从指定的套接字中获取信息。

2.5.10 close函数

该函数用于关闭套接字，其调用形式为：close(sockfd)。

# 第三章 DES加密解密设计

## 3.1 DES类的定义

在DES部分的实现代码中，首先定义封装DES操作的类，CDesOperate，类的私有成员包括生成的16圈迭代密钥，初始密钥以及加密、解密流程中用到的四个函数。公有成员包括构造函数、析构函数以及根据上述4个函数封装的加密函数与解密函数，以方便调用。

typedef int INT32;

class CDesOperate

{

private:

ULONG32 m\_arrOutKey[16][2];/\*输出的key\*/

ULONG32 m\_arrBufKey[2];/\*形成起始密钥\*/

INT32 MakeData(ULONG32 \*left ,ULONG32 \*right ,ULONG32 number);

INT32 HandleData(ULONG32 \*left , ULONG8 choice);

INT32 MakeKey( ULONG32 \*keyleft,ULONG32 \*keyright ,ULONG32 number);

INT32 MakeFirstKey( ULONG32 \*keyP );

public:

CDesOperate();

~CDesOperate();

INT32 Encry(char\* pPlaintext,int nPlaintextLength,char \*pCipherBuffer,int &nCipherBufferLength, char \*pKey,int nKeyLength);

INT32 Decry(char\* pCipher,int nCipherBufferLength,char \*pPlaintextBuffer, int &nPlaintextBufferLength, char \*pKey,int nKeyLength);

};

其中HandleData用来执行一次完整的加密或解密操作，MakeData用来实现16轮加密或解密迭代中的每一轮除去初始置换和逆初始置换的中间操作，MakeFirstKey用来利用用户输入的初始密钥，来形成16个迭代用到的子密钥，MakeKey用来形成16个密钥中的每一个子密钥，Encry用于加密，Decry用于解密。

## 3.2 DES算法中用到的静态数组

初始置换IP：

static const ULONG8 pc\_first[64] = {/\*初始置换IP\*/

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 } ;

逆初始置换：

static const ULONG8 pc\_last[64] = { /\*逆初始置换IP-1\*/

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};

按位取值或赋值：

static const ULONG32 pc\_by\_bit[64] = { /\*按位取值或赋值\*/

0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,

0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,

0x200000L, 0x100000L, 0x80000L, 0x40000L, 0x20000L,0x10000L,

0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,

0x100L, 0x80L,0x40L,0x20L, 0x10L, 0x8L, 0x4L, 0x2L, 0x1L,

0x80000000L,0x40000000L,0x20000000L,0x10000000L, 0x8000000L,

0x4000000L, 0x2000000L, 0x1000000L, 0x800000L, 0x400000L,

0x200000L, 0x100000L, 0x80000L, 0x40000L, 0x20000L, 0x10000L,

0x8000L, 0x4000L, 0x2000L, 0x1000L, 0x800L, 0x400L, 0x200L,

0x100L, 0x80L, 0x40L,0x20L, 0x10L, 0x8L, 0x4L, 0x2L, 0x1L, };

置换运算P：

static const ULONG8 des\_P[32] = {/\*置换运算P\*/

16,7,20,21, 29,12,28,17, 1,15,23,26,

5,18,31,10, 2,8,24,14, 32,27,3,9,

19,13,30,6, 22,11,4,25};

选择扩展运算E盒：

static const ULONG8 des\_E[48] = {/\*数据扩展\*/

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 };

选择压缩运算S盒：

static const ULONG8 des\_S[8][64] = {/\*数据压缩\*/

{

0xe,0x0,0x4,0xf,0xd,0x7,0x1,0x4,0x2,0xe,0xf,0x2,0xb,

0xd,0x8,0x1,0x3,0xa,0xa,0x6,0x6,0xc,0xc,0xb,0x5,0x9,

0x9,0x5,0x0,0x3,0x7,0x8,0x4,0xf,0x1,0xc,0xe,0x8,0x8,

0x2,0xd,0x4,0x6,0x9,0x2,0x1,0xb,0x7,0xf,0x5,0xc,0xb,

0x9,0x3,0x7,0xe,0x3,0xa,0xa,0x0,0x5,0x6,0x0,0xd

},

{

0xf,0x3,0x1,0xd,0x8,0x4,0xe,0x7,0x6,0xf,0xb,0x2,0x3,

0x8,0x4,0xf,0x9,0xc,0x7,0x0,0x2,0x1,0xd,0xa,0xc,0x6,

0x0,0x9,0x5,0xb,0xa,0x5,0x0,0xd,0xe,0x8,0x7,0xa,0xb,

0x1,0xa,0x3,0x4,0xf,0xd,0x4,0x1,0x2,0x5,0xb,0x8,0x6,

0xc,0x7,0x6,0xc,0x9,0x0,0x3,0x5,0x2,0xe,0xf,0x9

},

{

0xa,0xd,0x0,0x7,0x9,0x0,0xe,0x9,0x6,0x3,0x3,0x4,0xf,

0x6,0x5,0xa,0x1,0x2,0xd,0x8,0xc,0x5,0x7,0xe,0xb,0xc,

0x4,0xb,0x2,0xf,0x8,0x1,0xd,0x1,0x6,0xa,0x4,0xd,0x9,

0x0,0x8,0x6,0xf,0x9,0x3,0x8,0x0,0x7,0xb,0x4,0x1,0xf,

0x2,0xe,0xc,0x3,0x5,0xb,0xa,0x5,0xe,0x2,0x7,0xc

},

{

0x7,0xd,0xd,0x8,0xe,0xb,0x3,0x5,0x0,0x6,0x6,0xf,0x9,

0x0,0xa,0x3,0x1,0x4,0x2,0x7,0x8,0x2,0x5,0xc,0xb,0x1,

0xc,0xa,0x4,0xe,0xf,0x9,0xa,0x3,0x6,0xf,0x9,0x0,0x0,

0x6,0xc,0xa,0xb,0xa,0x7,0xd,0xd,0x8,0xf,0x9,0x1,0x4,

0x3,0x5,0xe,0xb,0x5,0xc,0x2,0x7,0x8,0x2,0x4,0xe

},

{

0x2,0xe,0xc,0xb,0x4,0x2,0x1,0xc,0x7,0x4,0xa,0x7,0xb,

0xd,0x6,0x1,0x8,0x5,0x5,0x0,0x3,0xf,0xf,0xa,0xd,0x3,

0x0,0x9,0xe,0x8,0x9,0x6,0x4,0xb,0x2,0x8,0x1,0xc,0xb,

0x7,0xa,0x1,0xd,0xe,0x7,0x2,0x8,0xd,0xf,0x6,0x9,0xf,

0xc,0x0,0x5,0x9,0x6,0xa,0x3,0x4,0x0,0x5,0xe,0x3

},

{

0xc,0xa,0x1,0xf,0xa,0x4,0xf,0x2,0x9,0x7,0x2,0xc,0x6,

0x9,0x8,0x5,0x0,0x6,0xd,0x1,0x3,0xd,0x4,0xe,0xe,0x0,

0x7,0xb,0x5,0x3,0xb,0x8,0x9,0x4,0xe,0x3,0xf,0x2,0x5,

0xc,0x2,0x9,0x8,0x5,0xc,0xf,0x3,0xa,0x7,0xb,0x0,0xe,

0x4,0x1,0xa,0x7,0x1,0x6,0xd,0x0,0xb,0x8,0x6,0xd

},

{

0x4,0xd,0xb,0x0,0x2,0xb,0xe,0x7,0xf,0x4,0x0,0x9,0x8,

0x1,0xd,0xa,0x3,0xe,0xc,0x3,0x9,0x5,0x7,0xc,0x5,0x2,

0xa,0xf,0x6,0x8,0x1,0x6,0x1,0x6,0x4,0xb,0xb,0xd,0xd,

0x8,0xc,0x1,0x3,0x4,0x7,0xa,0xe,0x7,0xa,0x9,0xf,0x5,

0x6,0x0,0x8,0xf,0x0,0xe,0x5,0x2,0x9,0x3,0x2,0xc

},

{

0xd,0x1,0x2,0xf,0x8,0xd,0x4,0x8,0x6,0xa,0xf,0x3,0xb,

0x7,0x1,0x4,0xa,0xc,0x9,0x5,0x3,0x6,0xe,0xb,0x5,0x0,

0x0,0xe,0xc,0x9,0x7,0x2,0x7,0x2,0xb,0x1,0x4,0xe,0x1,

0x7,0x9,0x4,0xc,0xa,0xe,0x8,0x2,0xd,0x0,0xf,0x6,0xc,

0xa,0x9,0xd,0x0,0xf,0x3,0x3,0x5,0x5,0x6,0x8,0xb

} };

等分密钥，密钥循环左移及密钥选取：

static const ULONG8 keyleft[28] = {/\*等分密钥\*/

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};

static const ULONG8 keyright[28] = {/\*等分密钥\*/

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};

static const ULONG8 lefttable[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};/\*密钥移位\*/

static const ULONG8 keychoose[48] ={/\*密钥选取\*/

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,

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

## 3.3 DES密钥生成

DES密钥是一个64bit的分组，但是其中8bit用于奇偶校验，所以密钥的有效位只有56bit，由这56bit生成16轮子密钥。

首先将有效的56bit进行置换选择，将结果等分为28bit的两个部分，再根据所在的迭代轮数进行循环左移，左移后将两个部分合并为56bit的密钥，从中选取48bit作为此轮迭代的最终密钥，共生成16个48bit的密钥。每一个密钥，分为两个24bit的部分放在一个ULONG32的二维数组中保存。

每一轮密钥的生成，由MakeKey函数实现：

INT32 MakeKey( ULONG32 \*keyleft,ULONG32 \*keyright ,ULONG32 number)

{

ULONG32 tmpkey[2] ={0};

ULONG32 \*Ptmpkey = (ULONG32\*)tmpkey;

ULONG32 \*Poutkey = (ULONG32\*)&m\_arrOutKey[number];

INT32 j;

memset((ULONG8\*)tmpkey,0,sizeof(tmpkey));

/\*要最高的一位或两位\*/

\*Ptmpkey = \*keyleft&wz\_leftandtab[lefttable[number]] ;

Ptmpkey[1] = \*keyright&wz\_leftandtab[lefttable[number]] ;

if ( lefttable[number] == 1)

{

\*Ptmpkey >>= 27;

Ptmpkey[1] >>= 27;

}

else

{

\*Ptmpkey >>= 26;

Ptmpkey[1] >>= 26;

}

Ptmpkey[0] &= 0xfffffff0;

Ptmpkey[1] &= 0xfffffff0;

/\*得到高位的值\*/

\*keyleft <<= lefttable[number] ;

\*keyright <<= lefttable[number] ;

\*keyleft |= Ptmpkey[0] ;

\*keyright |= Ptmpkey[1] ;

Ptmpkey[0] = 0;

Ptmpkey[1] = 0;

/\*从56位中选出48位,3个16位\*/

for ( j = 0 ; j < 48 ; j++)

{

if ( j < 24 )

{

if ( \*keyleft&pc\_by\_bit[keychoose[j]-1])

{

Poutkey[0] |= pc\_by\_bit[j] ;

}

}

else /\*j>=24\*/

{

if ( \*keyright&pc\_by\_bit[(keychoose[j]-28)])

{

Poutkey[1] |= pc\_by\_bit[j-24] ;

}

}

}

return SUCCESS;

}

## 3.4 DES加密运算

DES的加密运算也分为16bit圈迭代。

首先将明文分为64bit的数据块，不够64bit的用0补齐。在每一轮中，对每一个64bit的数据块，首先进行初始换位，并将数据块分为32bit的两部分：

INT32 number = 0 ,j = 0;

ULONG32 \*right = &left[1] ;

ULONG32 tmp = 0;

ULONG32 tmpbuf[2] = { 0 };

/\*第一次调整pc\_first[64]\*/

for ( j = 0 ; j < 64 ; j++)

{

if (j < 32 )

{

if ( pc\_first[j] > 32)/\*属于right\*/

{

if ( \*right&pc\_by\_bit[pc\_first[j]-1] )

{

tmpbuf[0] |= pc\_by\_bit[j] ;

}

}

else

{

if ( \*left&pc\_by\_bit[pc\_first[j]-1] )

{

tmpbuf[0] |= pc\_by\_bit[j] ;

}

}

}

else

{

if ( pc\_first[j] > 32)/\*属于right\*/

{

if ( \*right&pc\_by\_bit[pc\_first[j]-1] )

{

tmpbuf[1] |= pc\_by\_bit[j] ;

}

}

else

{

if ( \*left&pc\_by\_bit[pc\_first[j]-1] )

{

tmpbuf[1] |= pc\_by\_bit[j] ;

}

}

}

}

\*left = tmpbuf[0];

\*right = tmpbuf[1];

经过初始置换并且分组之后，将进行DES加密算法的核心部分。

首先，保持左部不变，将右部由32bit扩展成为48bit，分别存在两个ULONG32类型的变量里，每个占24bit：

/\*从56位中选出48位,3个16位\*/

for ( j = 0 ; j < 48 ; j++)

{

if ( j < 24 )

{

if ( \*keyleft&pc\_by\_bit[keychoose[j]-1])

{

Poutkey[0] |= pc\_by\_bit[j] ;

}

}

else /\*j>=24\*/

{

if ( \*keyright&pc\_by\_bit[(keychoose[j]-28)])

{

Poutkey[1] |= pc\_by\_bit[j-24] ;

}

}

}

在将右部扩展为48bit之后，与该轮的密钥进行异或操作，由于48bit分在一个ULONG32数组中的两个元素中，因此要进行两次异或操作：

for ( j = 0 ; j < 2 ; j++)

{

exdes\_P[j] ^= m\_arrOutKey[number][j] ;

}

在异或操作完成之后，对新的48bit进行压缩操作，即S盒。

将其每取6bit，进行一次操作：

/\*由48－>32\*/

exdes\_P[1] >>= 8 ;

rexpbuf[7] = (ULONG8) (exdes\_P[1]&0x0000003fL) ;

exdes\_P[1] >>= 6 ;

rexpbuf[6] = (ULONG8) (exdes\_P[1]&0x0000003fL) ;

exdes\_P[1] >>= 6 ;

rexpbuf[5] = (ULONG8) (exdes\_P[1]&0x0000003fL) ;

exdes\_P[1] >>= 6 ;

rexpbuf[4] = (ULONG8) (exdes\_P[1]&0x0000003fL) ;

exdes\_P[0] >>= 8 ;

rexpbuf[3] = (ULONG8) (exdes\_P[0]&0x0000003fL) ;

exdes\_P[0] >>= 6 ;

rexpbuf[2] = (ULONG8) (exdes\_P[0]&0x0000003fL) ;

exdes\_P[0] >>= 6 ;

rexpbuf[1] = (ULONG8) (exdes\_P[0]&0x0000003fL) ;

exdes\_P[0] >>= 6 ;

rexpbuf[0] = (ULONG8) (exdes\_P[0]&0x0000003fL) ;

exdes\_P[0] = 0 ;

exdes\_P[1] = 0 ;

8个6bit的数据存在ULONG rexpbuf[8]中，然后进行数据压缩操作，每一个6bit经过运算之后输出4bit，因此最终输出的是压缩后的32bit数据：

/\*由48－> 32\*/

\*right = 0 ;

for ( j = 0 ; j < 7 ; j++)

{

\*right |= des\_S[j][rexpbuf[j]] ;

\*right <<= 4 ;

}

\*right |= des\_S[j][rexpbuf[j]] ;

对新的32bit数据，进行一次置换操作：

/\*又要换位了\*/

datatmp = 0;

for ( j = 0 ; j < 32 ; j++)

{

if ( \*right&pc\_by\_bit[des\_P[j]-1] )

{

datatmp |= pc\_by\_bit[j] ;

}

}

\*right = datatmp ;

再把左右部分进行异或作为右半部分，最原始的右边作为左半部分：

/\*一轮结束收尾操作\*/

\*right ^= \*left;

\*left = oldright;

最后进行逆初始置换，完成一轮完整的加密操作：

/\*最后一次调整pc\_last[64]\*/

for ( j = 0 ; j < 64 ; j++)

{

if (j < 32 )

{

if ( pc\_last[j] > 32)/\*属于right\*/

{

if ( \*right&pc\_by\_bit[pc\_last[j]-1] )

{

tmpbuf[0] |= pc\_by\_bit[j] ;

}

}

else

{

if ( \*left&pc\_by\_bit[pc\_last[j]-1] )

{

tmpbuf[0] |= pc\_by\_bit[j] ;

}

}

}

else

{

if ( pc\_last[j] > 32)/\*属于right\*/

{

if ( \*right&pc\_by\_bit[pc\_last[j]-1] )

{

tmpbuf[1] |= pc\_by\_bit[j] ;

}

}

else

{

if ( \*left&pc\_by\_bit[pc\_last[j]-1] )

{

tmpbuf[1] |= pc\_by\_bit[j] ;

}

}

}

}

\*left = tmpbuf[0] ;

\*right = tmpbuf[1];

## 3.5 封装DES加密函数

将上述运算整合在一起，可以封装成一个加密函数，以便于调用，其中pPlaintext为明文部分，nPlaintextLength为明文长度，pCipherBuffer为准备存放密文的缓冲区，nCipherBufferLength为密文长度，pKey为密钥，nKeyLength为密钥长度：

INT32 Encry(char\* pPlaintext,int nPlaintextLength,char \*pCipherBuffer,int &nCipherBufferLength, char \*pKey,int nKeyLength)

由于加密、解密均要以32bit为单位进行操作，故需要计算相关参数，以确定加密的循环次数以及密文缓冲区是否够用，确定后将需要加密的明文格式化到新分配的缓冲区内：

int nLenthofLong = ((nPlaintextLength+7)/8)\*2;

if(nCipherBufferLength<nLenthofLong\*4)

{//out put buffer is not enough

nCipherBufferLength=nLenthofLong\*4;

return 0;

}

memset(pCipherBuffer,0,nCipherBufferLength);

ULONG32 \*pOutPutSpace = (ULONG32 \*)pCipherBuffer;

ULONG32 \*pSource;

if(nPlaintextLength != sizeof(ULONG32)\*nLenthofLong)

{

pSource = new ULONG32[nLenthofLong];

memset(pSource,0,sizeof(ULONG32)\*nLenthofLong);

memcpy(pSource,pPlaintext,nPlaintextLength);

}

else

{

pSource = (ULONG32 \*)pPlaintext;

}

开始对明文进行加密，加密后将之前分配的缓冲区从内存中删除：

ULONG32 gp\_msg[2] = {0,0};

for (int i=0;i<(nLenthofLong/2);i++)

{

gp\_msg[0] = pSource[2\*i];

gp\_msg[1] =pSource[2\*i+1];

HandleData(gp\_msg,DESENCRY);

pOutPutSpace[2\*i] = gp\_msg[0];

pOutPutSpace[2\*i+1] = gp\_msg[1];

}

if(pPlaintext!=(char \*)pSource)

{

delete []pSource;

}

最后需要说明，上述函数为一次完整的加密流程，解密流程与加密流程基本一致，不同的的地方在于所生成的16个密钥的使用顺序，加密运算与解密运算的使用顺序正好想法。

# 第四章 基于TCP的聊天功能模块设计

## 4.1 建立连接

对于客户端，首先输入服务器IP地址，建立并初始化连接套接字和sockaddr\_in结构体，向服务器请求连接，进行实时聊天，关闭套接字：

char strIpAddr[16];

printf("Please input the server address:\r\n");

cin>>strIpAddr;

int nConnectSocket, nLength;

struct sockaddr\_in sDestAddr;

if ((nConnectSocket = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

perror("Socket");

exit(errno);

}

bzero(&sDestAddr, sizeof(sDestAddr));

sDestAddr.sin\_family = AF\_INET;

sDestAddr.sin\_port = htons(SERVERPORT);

sDestAddr.sin\_addr.s\_addr = inet\_addr(strIpAddr);

/\* 连接服务器 \*/

if (connect(nConnectSocket, (struct sockaddr \*) &sDestAddr, sizeof(sDestAddr)) != 0)

{

perror("Connect ");

exit(errno);

}

else

{

printf("Connect Success! \nBegin to chat...\n");

SecretChat(nConnectSocket,strIpAddr,"benbenmi");

}

close(nConnectSocket);

对于服务器端，建立并初始化本地sockaddr\_in结构体，与本地套接字绑定并开始监听，建立远程sockaddr\_in和套接字，在接受客户端连接请求后存储客户端的相关信息：

int nListenSocket, nAcceptSocket;

socklen\_t nLength = 0;

struct sockaddr\_in sLocalAddr, sRemoteAddr;

if ((nListenSocket = socket(PF\_INET, SOCK\_STREAM, 0)) == -1)

{

perror("socket");

exit(1);

}

bzero(&sLocalAddr, sizeof(sLocalAddr));

sLocalAddr.sin\_family = PF\_INET;

sLocalAddr.sin\_port = htons(SERVERPORT);

sLocalAddr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(nListenSocket, (struct sockaddr \*) &sLocalAddr, sizeof(struct sockaddr))== -1)

{

perror("bind");

exit(1);

}

if (listen(nListenSocket, 5) == -1)

{

perror("listen");

exit(1);

}

printf("Listening...\n");

nLength = sizeof(struct sockaddr);

if ((nAcceptSocket = accept(nListenSocket, (struct sockaddr \*) &sRemoteAddr,&nLength)) == -1)

{

perror("accept");

exit(errno);

}

else

{

close(nListenSocket);

printf("server: got connection from %s, port %d, socket %d\n",inet\_ntoa(sRemoteAddr.sin\_addr),ntohs(sRemoteAddr.sin\_port), nAcceptSocket);

SecretChat(nAcceptSocket,inet\_ntoa(sRemoteAddr.sin\_addr),"benbenmi");

close(nAcceptSocket);

}

## 4.2 多进程全双工聊天程序分析

Linux是一种多用户、多进程的操作系统。每个进程都有一个唯一的进程标识符，操作系统通过对机器资源进行时间共享，并发地运行许多进程。

在Linux中，程序员可以使用fork()函数创建新进程，它可以与父进程完全并发地运行。fork()函数不接受任何参数，并返回一个int值。当它被调用时，创建出的子进程除了拥有自己的进程标识符以外，其余特征，例如数据段、堆栈段、代码段等和其父进程完全相同。fork()函数向子进程返回0，向父进程返回子进程的进程标识符，该标识符是一个非零的int值。

当fork()函数被执行后，一个完全独立的子进程已经创建完毕并开始运行，在代码中可以利用该函数的返回值来区分父进程和子进程。另外，每个进程都有自己独立的堆栈段，所以两个进程的局部变量相互独立，在任意一个进程中都可以随便访问而不必考虑同步问题，但是如果进程使用了文件指针，则必须小心对待，因为两个进程的文件指针将会指向同一个低层文件，并行的读写操作可能会造成冲突。

另外，如果调用fork()函数的次数过于频繁造成系统中进程总数过多，系统可能由于耗尽所有可用的资源而导致创建新进程失败。

该聊天功能在函数SecretChat()中实现：

void SecretChat(int nSock,char \*pRemoteName, char \*pKey)

{

CDesOperate cDes;

if(strlen(pKey)!=8)

{

printf("Key length error");

return ;

}

pid\_t nPid;

nPid = fork();

if(nPid != 0)

{

while(1)

{

bzero(&strSocketBuffer, BUFFERSIZE);

int nLength = 0;

nLength = recv(nSock, strSocketBuffer,BUFFERSIZE,0);

if(nLength !=BUFFERSIZE)

{

break;

}

else

{

int nLen = BUFFERSIZE;

cDes.Decry(strSocketBuffer,BUFFERSIZE,strDecryBuffer,nLen,pKey,8);

strDecryBuffer[BUFFERSIZE-1]=0;

if(strDecryBuffer[0]!=0&&strDecryBuffer[0]!='\n')

{

printf("Receive message form <%s>: %s\n", pRemoteName,strDecryBuffer);

if(0==memcmp("quit",strDecryBuffer,4))

{

printf("Quit!\n");

break;

}

}

}

}

}

else

{

while(1)

{

bzero(&strStdinBuffer, BUFFERSIZE);

while(strStdinBuffer[0]==0)

{

if (fgets(strStdinBuffer, BUFFERSIZE, stdin) == NULL)

{

continue;

}

}

int nLen = BUFFERSIZE;

cDes.Encry(strStdinBuffer,BUFFERSIZE,strEncryBuffer,nLen,pKey,8);

if(send(nSock, strEncryBuffer, BUFFERSIZE,0)!=BUFFERSIZE)

{

perror("send");

}

else

{

if(0==memcmp("quit",strStdinBuffer,4))

{

printf("Quit!\n");

break;

}

}

}

}

}

# 第五章 运行演示

# 第六章 总结