第3单元　对称密钥

3.1　知识讲解

3.1.1　DES算法的历史

DES (Data Encryption Standard)算法迄今为止已有近半个世纪的历史，虽然它已不再作为数据加密标准，但仍在世界范围内被广泛应用于网络通信加密、数据存储加密、口令和访问控制等诸多领域：

- 1973年，美国国家标准局公开征集国家密码标准
- 1974年，IBM公司提交加密算法草案
- 1976年，确定为联邦数据加密标准，用于非机密政府通信
- 1977年，确定为正式标准，广泛应用于美国政府、军队
- 1988年，服役期满转为民用，被美国商界和其它国家广泛采用
- 1998年，退出美国国家密码标准
- 2001年，AES (Advanced Encryption Standard)取代DES成为新的美国国家密码标准

3.1.2　DES算法的特点

DES算法的主要特点可以归纳为：

- 对称分组密码算法，明文和密文的分组长度都是64位
- 初始密钥长度为64位，其中包含8个奇偶校验位，有效密钥长度为56位
- 根据密钥生成算法，由初始密钥生成16个子密钥，每个子密钥的长度为48位
- 加密过程需要经历16轮迭代，每轮使用不同的子密钥
- 解密过程与加密过程的计算方法完全相同，只是子密钥的使用顺序相反
- DES算法的全部细节都是公开的，其安全性完全取决于密钥的保密程度

3.1.3　DES算法的内容

1. 初始置换

所谓置换，即将源数据中的各个二进制位按照某种规则重新排列得到目的数据的过程。

初始置换将64位源数据以8位(一个字节)为一行排成方阵，从左至右依次取第2、4、6、8四个偶数列和第1、3、5、7四个奇数列，构成一个新的方阵，沿顺时针方向旋转90度，再按行重新组合，即得到64位目的数据。如下表所示：

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

经换位后，还需将64位目的数据拆分为左右两个各32位的块(L_0和R_0)。

初始置换的输入和输出之间存在二进制位级别的一一对应关系，其目的在于打乱源数据的位排列顺序，例如：

```
BIT: 123                          ...                          64
     |||                                                        |
     vvv                                                        v
SRC: 10000000110000001110000011110000111100001111110011111110111111111
                                  | IP
                                  v
DST: 11111110111110001110000010000000111111111111110011110000011000000
           ^               ^             ^       ^
           |               |             |       |
BIT:       2              64             1       3
     \_____________________________/\_____________________________/
                   L0                            R0
```

加密过程中的初始置换记作：

$$L_0, R_0 = IP(M)$$

其中，M表示64位明文分组。

2. 逆初始置换

在换位前，先将左右两个各32位的块(L_{16}和R_{16})合并为一个64位源数据。

逆初始置换将64位源数据以8位(一个字节)为一行排成方阵，从上至下将前四行作为第2、4、6、8四个偶数行，后四行作为第1、3、5、7四个奇数行，构成一个新的方阵，沿逆时针方向旋转90度，再按行重新组合，即得到64位目的数据。如下表所示：

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

逆初始置换的输入和输出之间存在二进制位级别的一一对应关系，其目的在于打乱源数据的位排列顺序，例如：

```
                 L16                           R16
     ____________________________  ____________________________
    /                           \/                           \
BIT: 123                          ...                          64
     |||                                                        |
     vvv                                                        v
SRC: 10000000110000001110000011110000111100001111110011111110111111111
                                  | IP-1
                                  v
DST: 00000010000010100010101010101010101010111010111101111111111111111
           ^                                 ^       ^       ^
           |                                 |       |       |
BIT:      64                                 3       2       1
```

加密过程中的逆初始置换记作：

$$C = IP^{-1}(L_{16}, R_{16})$$

其中，C表示64位密文分组。

3. 16轮迭代

从初始置换的输出(L_0和R_0)到逆初始置换的输入(L_{16}和R_{16})共需经历16轮迭代。

前15轮迭代中的第i $(i = 1, 2, \cdots, 15)$轮完成如下操作：

- 将上轮输出的32位右块R_{i-1}，直接作为本轮输出的32位左块L_i
- 将上轮输出的32位右块R_{i-1}和本轮子密钥K_i作为输入，传递给加密变换函数F
- 将加密变换函数F的输出与上轮输出的32位左块L_{i-1}做异或，得到本轮输出的32位右块R_i

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \end{cases} \qquad (i = 1, 2, \cdots, 15)$$

最后一轮即第i $(i = 16)$轮迭代比较特殊：

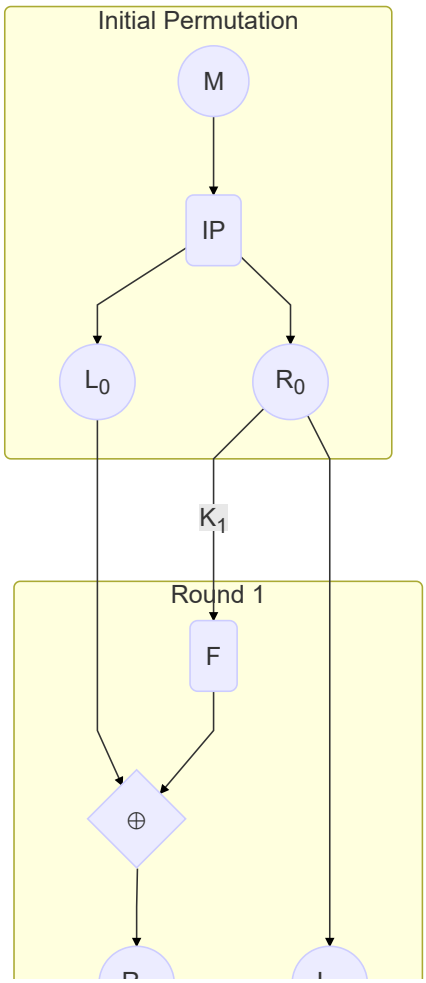
- 将上轮输出的32位右块R_{i-1}，直接作为本轮输出的32位右块R_i
- 将上轮输出的32位右块R_{i-1}和本轮子密钥K_i作为输入，传递给加密变换函数F
- 将加密变换函数F的输出与上轮输出的32位左块L_{i-1}做异或，得到本轮输出的32位左块L_i

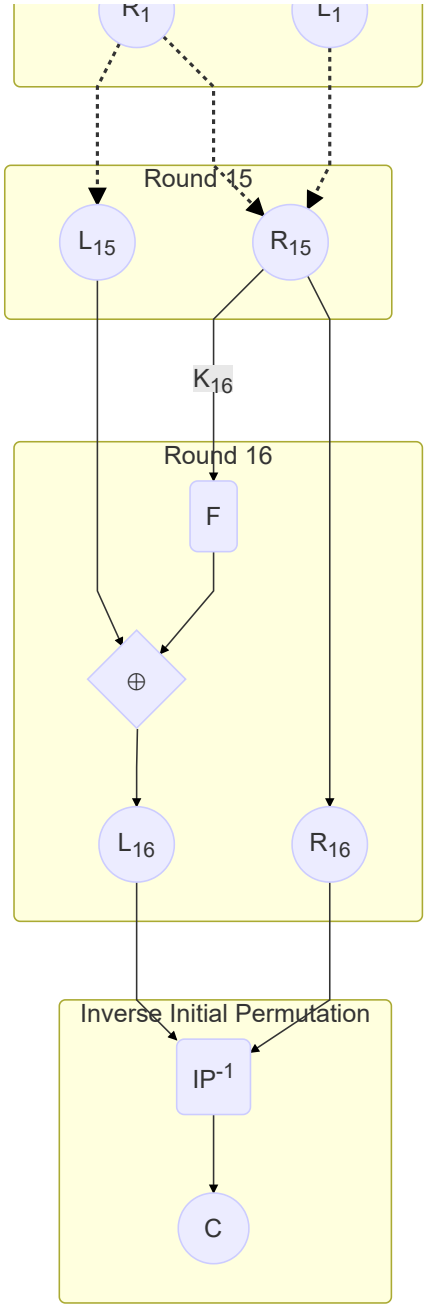
$$\begin{cases} L_i = L_{i-1} \oplus F(R_{i-1}, K_i) \\ R_i = R_{i-1} \end{cases} \qquad (i = 16)$$

即

$$\begin{cases} L_{16} = L_{15} \oplus F(R_{15}, K_{16}) \\ R_{16} = R_{15} \end{cases}$$

完整的加密过程如下图所示：





其中每一轮迭代都要将上轮输出的32位右块R_{i-1}和本轮子密钥K_i作为输入，传递给加密变换函数F。该函数包括如下运算：

- 选择扩展运算
- 密钥加运算
- 选择压缩运算
- 置换运算

1) 选择扩展运算

选择扩展运算，亦称E盒运算。将32位源数据按4位一行排列，在首列之前添加末列循环下移一位，在末列之后添加首列循环上移一位，再按行重新组合，即得到48位目的数据。如下表所示：

Source Bits				E	Destination Bits					
1	2	3	4		32	1	2	3	4	5
5	6	7	8		4	5	6	7	8	9
9	10	11	12		8	9	10	11	12	13
13	14	15	16		12	13	14	15	16	17

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

这里作为选择扩展运算源数据的是上轮迭代输出的32位右块R_{i-1}，目的数据为与每轮子密钥长度相等的48位扩展结果。记作：

$$E(R_{i-1}) \qquad (i=1,2,\cdots,16)$$

2) 密钥加运算

密钥加运算是将经选择扩展运算扩展至48位的上轮迭代输出右块$E(R_{i-1})$与本轮48位密钥K_i做异或运算。记作：

$$E(R_{i-1}) \oplus K_i \qquad (i=1,2,\cdots,16)$$

3) 选择压缩运算

选择压缩运算，亦称S盒运算，是DES算法中唯一的非线性部分。其作用是将48位源数据压缩至32位目的数据。在压缩过程中需要用到如下压缩变换表：

S	A	B	C	D	E	F	G	H
0	0xe	0xf	0xa	0x7	0x2	0xc	0x4	0xd
1	0x0	0x3	0xd	0xd	0xe	0xa	0xd	0x1
2	0x4	0x1	0x0	0xd	0xc	0x1	0xb	0x2
3	0xf	0xd	0x7	0x8	0xb	0xf	0x0	0xf
4	0xd	0x8	0x9	0xe	0x4	0xa	0x2	0x8
5	0x7	0x4	0x0	0xb	0x2	0x4	0xb	0xd
6	0x1	0xe	0xe	0x3	0x1	0xf	0xe	0x4
7	0x4	0x7	0x9	0x5	0xc	0x2	0x7	0x8
8	0x2	0x6	0x6	0x0	0x7	0x9	0xf	0x6
9	0xe	0xf	0x3	0x6	0x4	0x7	0x4	0xa
10	0xf	0xb	0x3	0x6	0xa	0x2	0x0	0xf
11	0x2	0x2	0x4	0xf	0x7	0xc	0x9	0x3
12	0xb	0x3	0xf	0x9	0xb	0x6	0x8	0xb
13	0xd	0x8	0x6	0x0	0xd	0x9	0x1	0x7
14	0xb	0x4	0x5	0xa	0x6	0x8	0xd	0x1
15	0xe	0xf	0xa	0x3	0x1	0x5	0xa	0x4
16	0x3	0x9	0x1	0x1	0x8	0x0	0x3	0xa
17	0xa	0xc	0x2	0x4	0x5	0x6	0xe	0xc
18	0xa	0x7	0xd	0x2	0x5	0xd	0xc	0x9
19	0x6	0x0	0x8	0x7	0x0	0x1	0x3	0x5
20	0x6	0x2	0xc	0x8	0x3	0x3	0x9	0x3
21	0xc	0x1	0x5	0x2	0xf	0xd	0x5	0x6

22	0xc	0xd	0x7	0x5	0xf	0x4	0x7	0xe
23	0xb	0xa	0xe	0xc	0xa	0xe	0xc	0xb
24	0x5	0xc	0xb	0xb	0xd	0xe	0x5	0x5
25	0x9	0x6	0xc	0x1	0x3	0x0	0x2	0x0
26	0x9	0x0	0x4	0xc	0x0	0x7	0xa	0x0
27	0x5	0x9	0xb	0xa	0x9	0xb	0xf	0xe
28	0x0	0x5	0x2	0x4	0xe	0x5	0x6	0xc
29	0x3	0xb	0xf	0xe	0x8	0x3	0x8	0x9
30	0x7	0xa	0x8	0xf	0x9	0xb	0x1	0x7
31	0x8	0x5	0x1	0x9	0x6	0x8	0x6	0x2
32	0x4	0x0	0xd	0xa	0x4	0x9	0x1	0x7
33	0xf	0xd	0x1	0x3	0xb	0x4	0x6	0x2
34	0x1	0xe	0x6	0x6	0x2	0xe	0x4	0xb
35	0xc	0x8	0xa	0xf	0x8	0x3	0xb	0x1
36	0xe	0x7	0x4	0x9	0x1	0xf	0xb	0x4
37	0x8	0xa	0xd	0x0	0xc	0x2	0xd	0xe
38	0x8	0xb	0x9	0x0	0xb	0x5	0xd	0x1
39	0x2	0x1	0x0	0x6	0x7	0xc	0x8	0x7
40	0xd	0xa	0x8	0xc	0xa	0x2	0xc	0x9
41	0x4	0x3	0x6	0xa	0x1	0x9	0x1	0x4
42	0x6	0x4	0xf	0xb	0xd	0x8	0x3	0xc
43	0x9	0xf	0x9	0xa	0xe	0x5	0x4	0xa
44	0x2	0xd	0x3	0x7	0x7	0xc	0x7	0xe
45	0x1	0x4	0x8	0xd	0x2	0xf	0xa	0x8
46	0xb	0x1	0x0	0xd	0x8	0x3	0xe	0x2
47	0x7	0x2	0x7	0x8	0xd	0xa	0x7	0xd
48	0xf	0x5	0xb	0xf	0xf	0x7	0xa	0x0
49	0x5	0xb	0x4	0x9	0x6	0xb	0x9	0xf
50	0xc	0x8	0x1	0x1	0x9	0x0	0xf	0x6
51	0xb	0x6	0xf	0x4	0xf	0xe	0x5	0xc
52	0x9	0xc	0x2	0x3	0xc	0x4	0x6	0xa
53	0x3	0x7	0xe	0x5	0x0	0x1	0x0	0x9
54	0x7	0x6	0xc	0xe	0x5	0xa	0x8	0xd
55	0xe	0xc	0x3	0xb	0x9	0x7	0xf	0x0
56	0x3	0x9	0x5	0x5	0x6	0x1	0x0	0xf

57	0xa	0x0	0xb	0xc	0xa	0x6	0xe	0x3
58	0xa	0x3	0xa	0x2	0x3	0xd	0x5	0x3
59	0x0	0x5	0x5	0x7	0x4	0x0	0x2	0x5
60	0x5	0x2	0xe	0x8	0x0	0xb	0x9	0x5
61	0x6	0xe	0x2	0x2	0x5	0x8	0x3	0x6
62	0x0	0xf	0x7	0x4	0xe	0x6	0x2	0x8
63	0xd	0x9	0xc	0xe	0x3	0xd	0xc	0xb

具体压缩过程如下：

- 将48位源数据$X_1X_2\cdots X_{48}$分成八组，每组6个二进制位：
  - A组：$X_1X_2X_3X_4X_5X_6$
  - B组：$X_7X_8X_9X_{10}X_{11}X_{12}$
  - C组：$X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}$
  - D组：$X_{19}X_{20}X_{21}X_{22}X_{23}X_{24}$
  - E组：$X_{25}X_{26}X_{27}X_{28}X_{29}X_{30}$
  - F组：$X_{31}X_{32}X_{33}X_{34}X_{35}X_{36}$
  - G组：$X_{37}X_{38}X_{39}X_{40}X_{41}X_{42}$
  - H组：$X_{43}X_{44}X_{45}X_{46}X_{47}X_{48}$
- $A\sim H$中的每个组对应压缩变换表中的一列，每组介于$0\sim 63$之间的6位二进制数对应该表中的一行。按组找列，按值找行，压缩变换表中的每个十六进制数，均可表示一个4位的二进制数：
  - 由A组的$X_1X_2X_3X_4X_5X_6$查表得到$Y_1Y_2Y_3Y_4$
  - 由B组的$X_7X_8X_9X_{10}X_{11}X_{12}$查表得到$Y_5Y_6Y_7Y_8$
  - 由C组的$X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}$查表得到$Y_9Y_{10}Y_{11}Y_{12}$
  - 由D组的$X_{19}X_{20}X_{21}X_{22}X_{23}X_{24}$查表得到$Y_{13}Y_{14}Y_{15}Y_{16}$
  - 由E组的$X_{25}X_{26}X_{27}X_{28}X_{29}X_{30}$查表得到$Y_{17}Y_{18}Y_{19}Y_{20}$
  - 由F组的$X_{31}X_{32}X_{33}X_{34}X_{35}X_{36}$查表得到$Y_{21}Y_{22}Y_{23}Y_{24}$
  - 由G组的$X_{37}X_{38}X_{39}X_{40}X_{41}X_{42}$查表得到$Y_{25}Y_{26}Y_{27}Y_{28}$
  - 由H组的$X_{43}X_{44}X_{45}X_{46}X_{47}X_{48}$查表得到$Y_{29}Y_{30}Y_{31}Y_{32}$
- 将八组4位二进制数按顺序拼接在一起，即得到压缩结果，32位目的数据$Y_1Y_2\cdots Y_{32}$

这里作为选择压缩运算源数据的是48位密钥加运算结果$E(R_{i-1})\oplus K_i$，目的数据为32位压缩结果。记作：

$$S(E(R_{i-1})\oplus K_i)\qquad (i=1,2,\cdots,16)$$

4) 置换运算

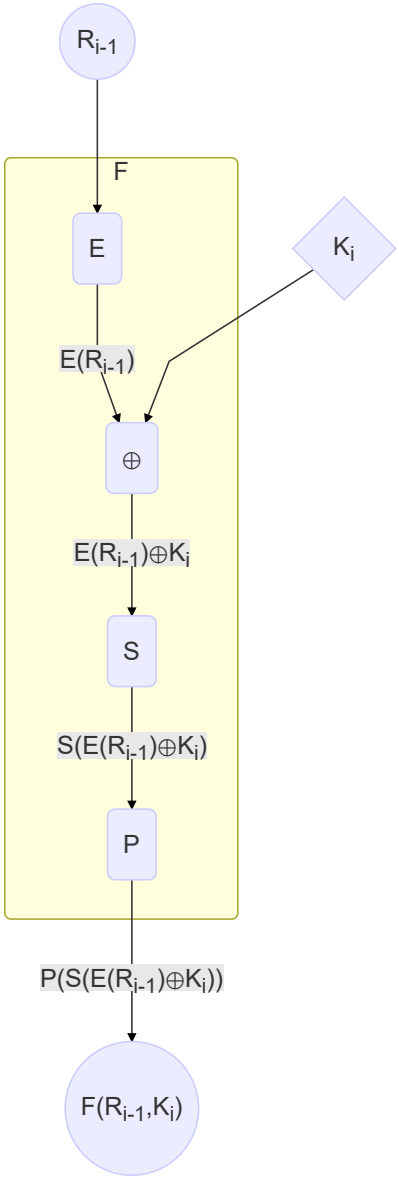
置换运算，亦称P盒运算，将32位源数据按下表所示规则进行换位，得到32位目的数据：

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

这里作为置换运算源数据的是选择压缩运算的32位压缩结果$S(E(R_{i-1})\oplus K_i)$，目的数据则作为加密变换函数F的输出。记作：

$$P(S(E(R_{i-1})\oplus K_i))\qquad (i=1,2,\cdots,16)$$

加密变换函数F的运算过程如下图所示：



加密变换函数F可表达为如下形式：

$$F(R_{i-1},K_i)=P(S(E(R_{i-1})\oplus K_i)) \qquad (i=1,2,\cdots,16)$$

其中，R_{i-1}为上轮迭代输出的32位右块，K_i为本轮迭代子密钥，E、S和P分别为选择扩展运算、选择压缩运算和置换运算。

4. 子密钥生成

64位初始密钥K_0中的第8、16、24、32、40、48、56、64位是奇偶校验位，其余56位为有效位。16轮迭代每轮都有自己的48位子密钥K_i。从初始密钥生成每轮子密钥，依次执行如下步骤：

- 置换选择$PC-1$
- 循环左移LS
- 置换选择$PC-2$

1) 置换选择$PC-1$

置换选择$PC-1$仅用于生成第一轮子密钥K_1，其作用在于从64位初始密钥K_0中选出56位有效位(其余8位为奇偶校验位)，同时将其分为两组，每组28位，分别放在C和D两个寄存器中。置换选择表如下所示：

K_0								PC-1	C_0,D_0							
1	2	3	4	5	6	7	8		57	49	41	33	25	17	9	1

9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

58	50	42	34	26	18	10	2
59	51	43	35	27	19	11	3
60	52	44	36	C			
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	D			

置换选择$PC-1$将64位初始密钥K_0以8位(一个字节)为一行排成方阵，排除最后一列奇偶校验位，将前七列从中间位置分开，得到左右两个相等的部分，各28位：

- 左半边沿顺时针方向旋转90度，再按行重新组合，进入C寄存器
- 右半边沿逆时针方向旋转90度，水平翻转后再按行重新组合，进入D寄存器

子密钥生成过程中的置换选择$PC-1$记作：

$$C_0, D_0 = PC-1(K_0)$$

2) 循环左移LS

为了生成包括第一轮在内的各轮子密钥$K_i\ (i=1,2,\cdots,16)$，需要先将C、D寄存器中各28位数据循环左移$N_i\ (i=1,2,\cdots,16)$位，N_i的取值如下表所示：

N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}	N_{16}
1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

子密钥生成过程中的循环左移LS记作：

$$\begin{cases} C_i = LS(C_{i-1}, N_i) \\ D_i = LS(D_{i-1}, N_i) \end{cases} \qquad (i=1,2,\cdots,16)$$

3) 置换选择$PC-2$

为了生成包括第一轮在内的各轮子密钥$K_i\ (i=1,2,\cdots,16)$，置换选择$PC-2$先将C、D寄存器中各28位数据合并成56位数据，再删除其中的第9、18、22、25、35、38、43、54位，将其余数据按下表所示规则进行换位，得到48位第i轮子密钥K_i：

C_i,D_i							
C	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
	15	16	17	18	19	20	21
	22	23	24	25	26	27	28
D	29	30	31	32	33	34	35
	36	37	38	39	40	41	42
	43	44	45	46	47	48	49
	50	51	52	53	54	55	56

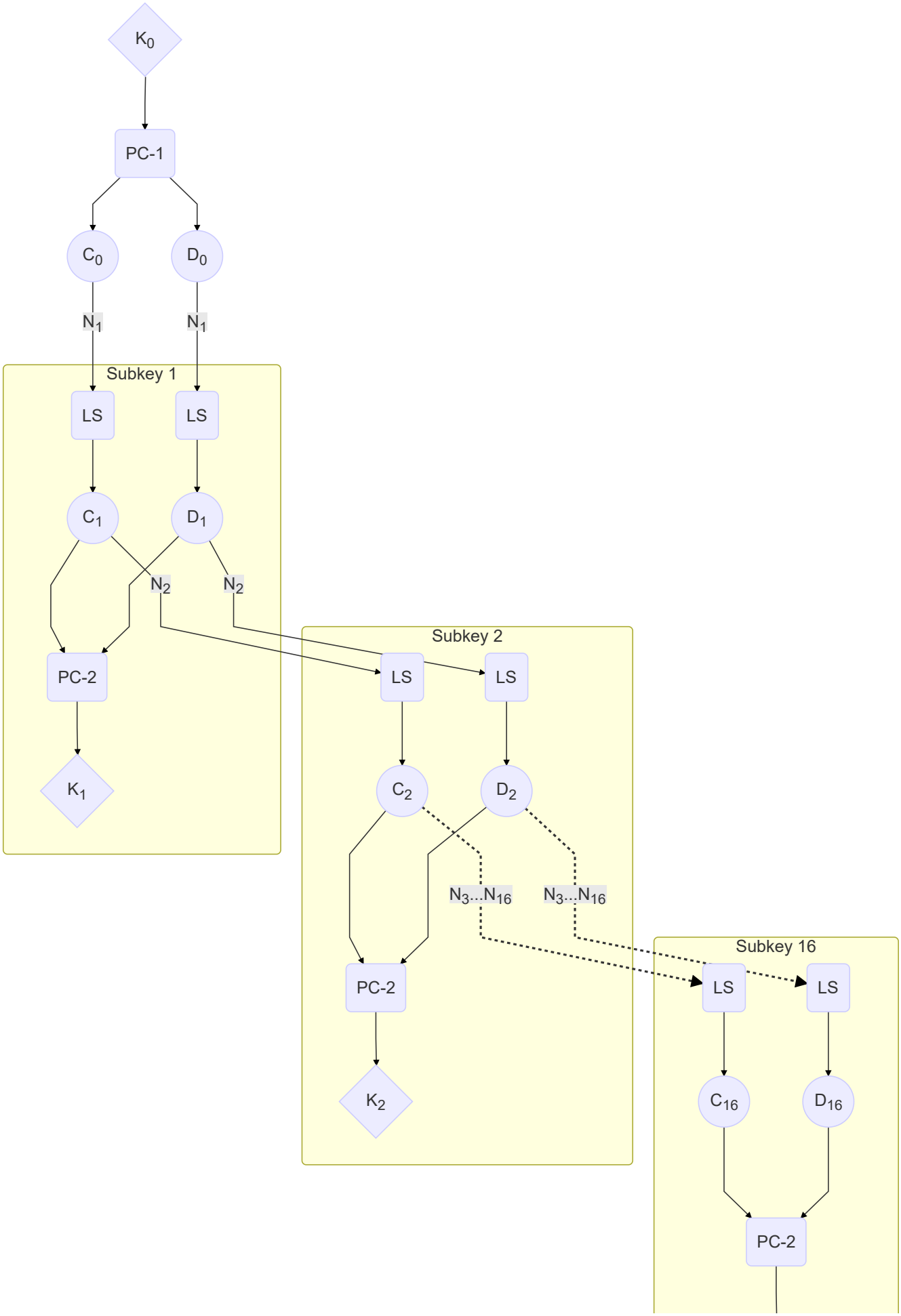
PC-2

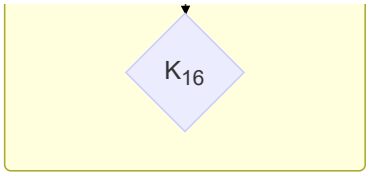
K_i					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

子密钥生成过程中的置换选择$PC-2$记作：

$$K_i = PC - 2(C_i, D_i) \qquad (i = 1, 2, \cdots, 16)$$

完整的子密钥生成过程如下图所示：

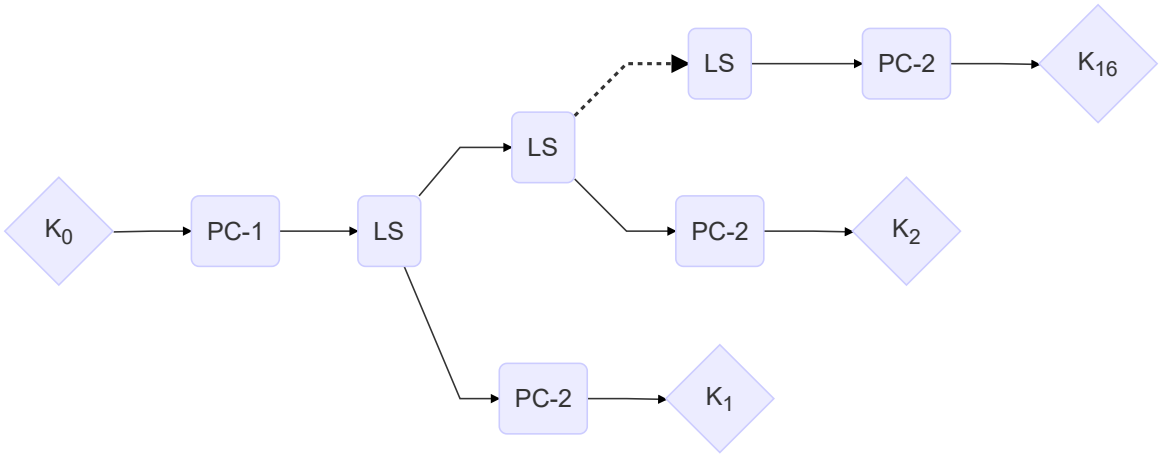




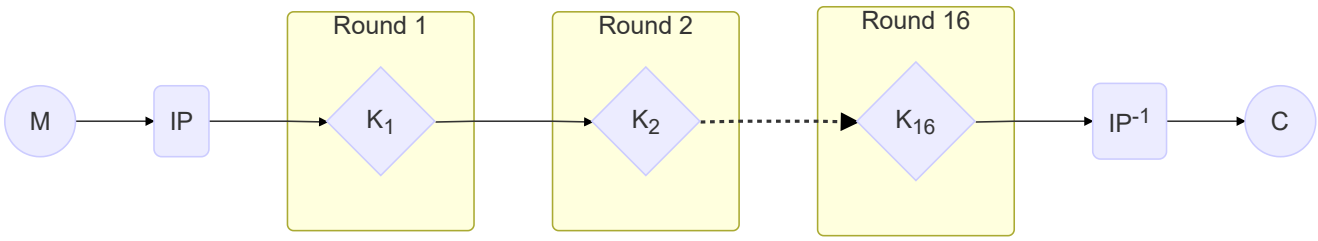
5. 对称密码

以上虽然是针对加密过程的描述，但对于解密过程亦同样适用。作为一种对称密码体系，DES算法的解密过程与加密过程完全相同，初始密钥亦完全相同，只是16轮子密钥的使用顺序相反。

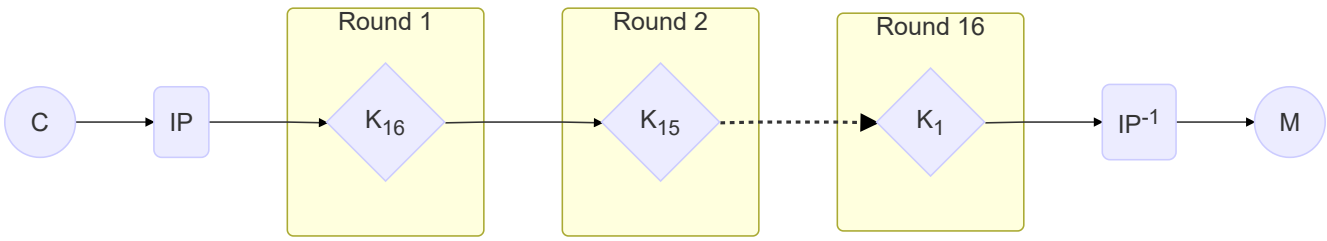
1) 子密钥生成过程



2) 加密过程



3) 解密过程



6. 算法描述

下面给出基于DES算法实现对称加解密的形式化描述，具体包括：

- 子密钥生成算法

- 加密算法
- 解密算法

1) 子密钥生成算法

$$C_0, D_0 = PC-1(K_0)$$

$$\begin{cases} C_i = LS(C_{i-1}, N_i) \\ D_i = LS(D_{i-1}, N_i) \end{cases} \qquad (i = 1, 2, \cdots, 16)$$

$$K_i = PC-2(C_i, D_i) \qquad (i = 1, 2, \cdots, 16)$$

2) 加密算法

$$L_0, R_0 = IP(M)$$

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \end{cases} \qquad (i = 1, 2, \cdots, 15)$$

$$\begin{cases} L_{16} = L_{15} \oplus F(R_{15}, K_{16}) \\ R_{16} = R_{15} \end{cases}$$

$$F(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i)) \qquad (i = 1, 2, \cdots, 16)$$

$$C = IP^{-1}(L_{16}, R_{16})$$

3) 解密算法

$$L_0, R_0 = IP(C)$$

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus F(R_{i-1}, K_{17-i}) \end{cases} \qquad (i = 1, 2, \cdots, 15)$$

$$\begin{cases} L_{16} = L_{15} \oplus F(R_{15}, K_1) \\ R_{16} = R_{15} \end{cases}$$

$$F(R_{i-1}, K_{17-i}) = P(S(E(R_{i-1}) \oplus K_{17-i})) \qquad (i = 1, 2, \cdots, 16)$$

$$M = IP^{-1}(L_{16}, R_{16})$$

3.1.4 TCP协议

1. TCP协议的特点

TCP协议，即传输控制协议(Transmission Control Protocol)，具有如下特点：

- 面向连接、面向字节流、可靠
- 通信之前先建立连接，以自己的方式缓存数据，缓存过程对用户透明
- 采用捎带确认(piggybacking，ACK)的方法，允许通信双发同时发送数据
- 最大报文长度(Maximum Segment Size, MSS)的默认值为536字节

2. TCP包头的结构

TCP包头由20字节的固定内容和最多40字节的可选内容组成，如下表所示：

字段	长度(位)	描述
源端口	16	本地TCP端口号
目的端口	16	远程TCP端口号

序号	32	跟踪发送包的序号	
确认序号	32	确认接收包的序号	
包头长度	4	以4字节为单位的包头长度	
保留	6	留作以后使用	
标志	6	URG	若为1，紧急指针字段有效
		ACK	若为1，确认序号字段有效
		PSH	若为1，告知接收端尽快将数据传送给应用层
		RST	若为1，复位连接
		SYN	若为1，ACK为0表示连接请求，ACK为1表示接受连接
		FIN	若为1，请求释放连接
窗口大小	16	要求对方必须维持的窗口字节数	
校验和	16	TCP包头和包体的校验和	
紧急指针	16	紧急数据的偏移值	
可选内容	≤40	MSS、窗口比例等	

3.1.5 套接字

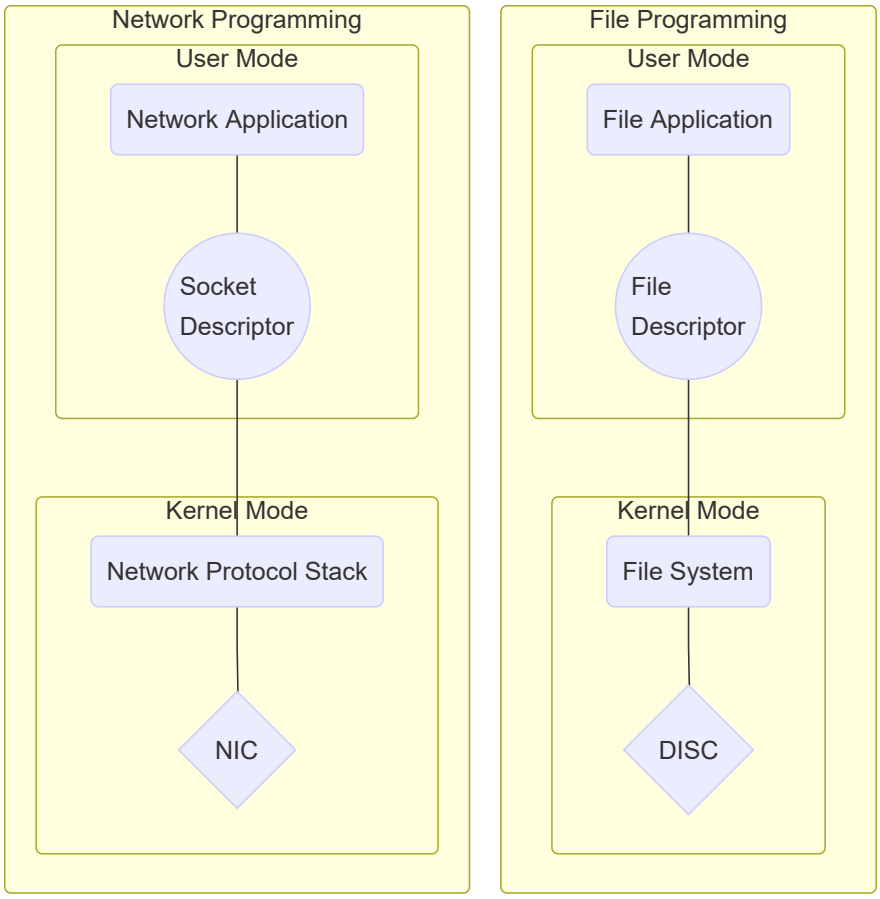
套接字(Socket)是应用层和TCP/IP协议族之间的软件抽象层，它以一组应用编程接口(API)的形式将复杂的网络通信协议隐藏在简单的函数调用背后，使应用程序具备基于特定通信协议访问网络资源的能力。

TCP/IP提供了三种类型的套接字：

- 流式套接字(SOCK_STREAM)
  - TCP协议
  - 有连接，无丢失，无差错，无重复，无乱序
- 数据报套接字(SOCK_DGRAM)
  - UDP协议
  - 无连接，有丢失，有差错，有重复，有乱序
- 原始套接字(SOCK_RAW)
  - IP或ICMP等较低层协议
  - 实现自定义网络协议
  - 对数据报文做较低层的控制

3.1.6 基本通信函数

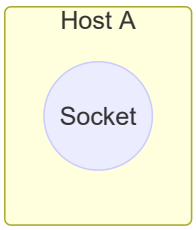
Linux系统通过套接字的概念来进行网络编程。应用程序通过调用socket和其它几个函数可以获得被称为套接字的文件描述符。程序的设计者可以将其当做普通文件描述符来操作，象读写磁盘文件一样收发网络中的数据，实现网络中计算机之间的通信。这充分体现了Linux系统设备无关性的优点。如下图所示：



1. socket函数

```
#include <sys/types.h>
#include <sys/socket.h>

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

该函数用于创建实现网络或本机通信的套接字，并返回该套接字的文件描述符，失败返回-1，并设置errno变量。如下图所示：



参数说明：

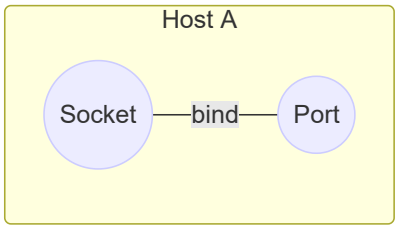
- domain：通信所使用的协议族
  - PF_UNIX：本机进程间的通信
  - PF_INET：网络主机间的通信
  - PF_PACKET：直接通过数据链路层或物理层实现网络通信
- type：套接字类型
  - SOCK_STREAM：流式套接字，传输层使用TCP协议
  - SOCK_DGRAM：数据报文式套接字，传输层使用UDP协议
  - SOCK_RAW：原始套接字，跨越传输层实现网络通信
- protocol：通信协议

- 通常情况下，对于给定协议族(domain参数)只有单一协议支持特定套接字类型(type参数)，此参数取0即可

2. bind函数

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

该函数用于将套接字与指定端口绑定。成功返回0，失败返回-1，并设置errno变量。如下图所示：



参数说明：

- sockfd：套接字文件描述符
- addr：描述绑定端口的地址结构
- addrlen：addr参数所指向结构的字节数

sockaddr是一个与协议无关的通用地址结构，相当于基类：

```
#include <sys/socket.h>

struct sockaddr {
    sa_family_t sa_family;   // 地址族
    char        sa_data[14]; // 地址数据
};
```

针对互联网通信，实际使用的是它的一个子类sockaddr_in：

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t    sin_family;  // 地址族，如AF_INET等
    in_port_t      sin_port;    // 端口号(16位网络字节序无符号整数)
    struct in_addr sin_addr;    // IP地址
    unsigned char  sin_zero[8]; // 不使用
};
```

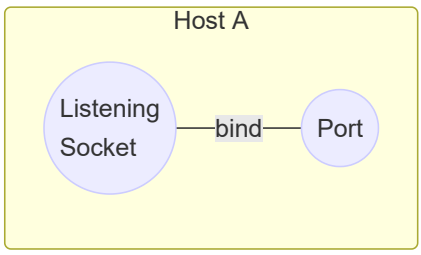
其中in_addr结构中只有一个字段：

```
#include <netinet/in.h>

typedef uint32_t in_addr_t;

struct in_addr {
    in_addr_t s_addr; // IPv4地址(32位网络字节序无符号整数)
};
```

3. listen函数

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

该函数用于将套接字设置为侦听状态。成功返回0，失败返回-1，并设置errno变量。如下图所示：



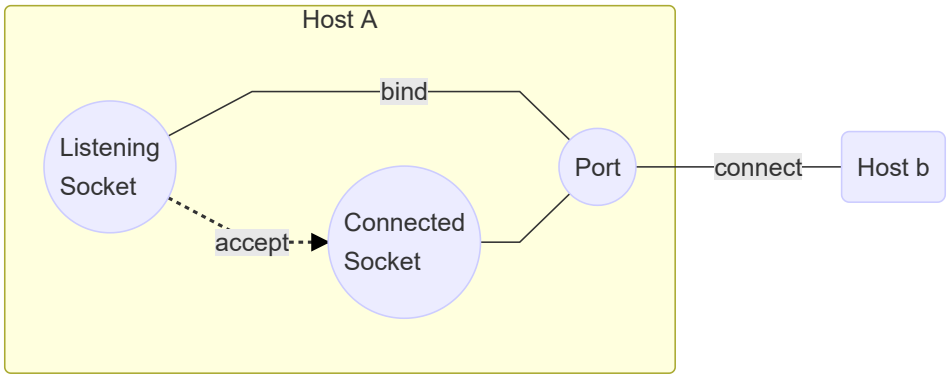
参数说明：

- sockdfd：套接字文件描述符
- backlog：未决连接队列最大长度

4. accept函数

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

该函数用于通过侦听套接字接受对方主机的连接请求。成功返回连接套接字，失败返回-1，并设置errno变量。如下图所示：



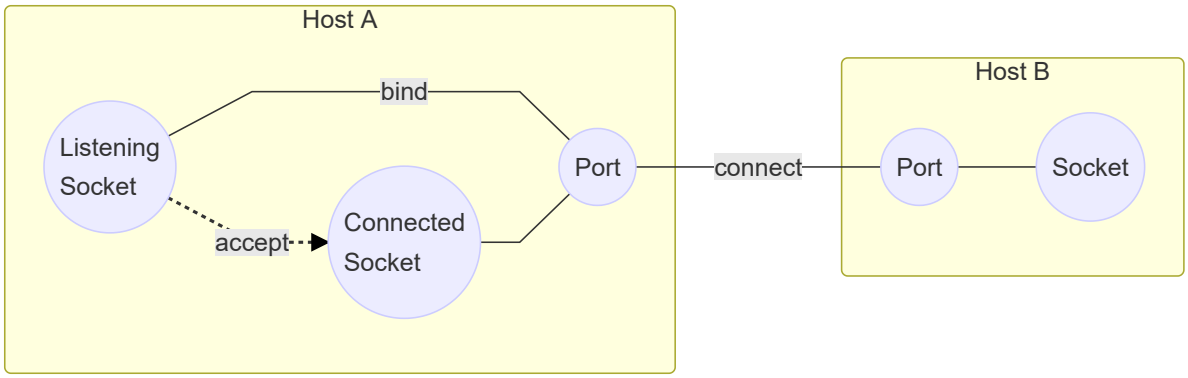
参数说明：

- sockfd：侦听套接字文件描述符
- addr：描述对方端口的地址结构
- addrlen：addr参数所指向结构的字节数

5. connect函数

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

该函数用于向对方主机发起连接请求。成功返回0，失败返回-1，并设置errno变量。如下图所示：



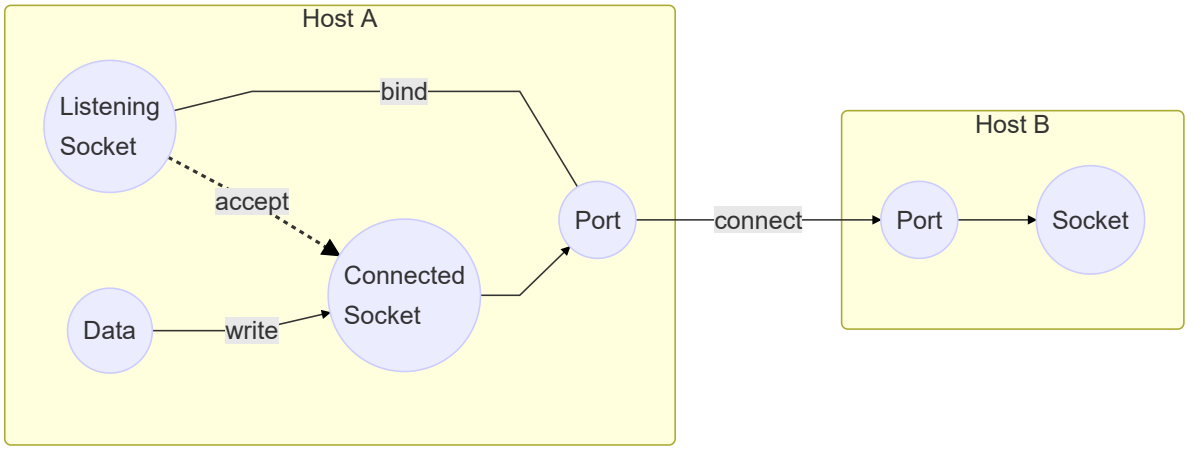
参数说明：

- sockfd：套接字文件描述符
- addr：描述对方端口的地址结构
- addrlen：addr参数所指向结构的字节数

6. write函数

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

该函数用于通过已连接套接字向对方主机发送数据。成功返回实际发送的字节数，失败返回-1，并设置errno变量。如下图所示：



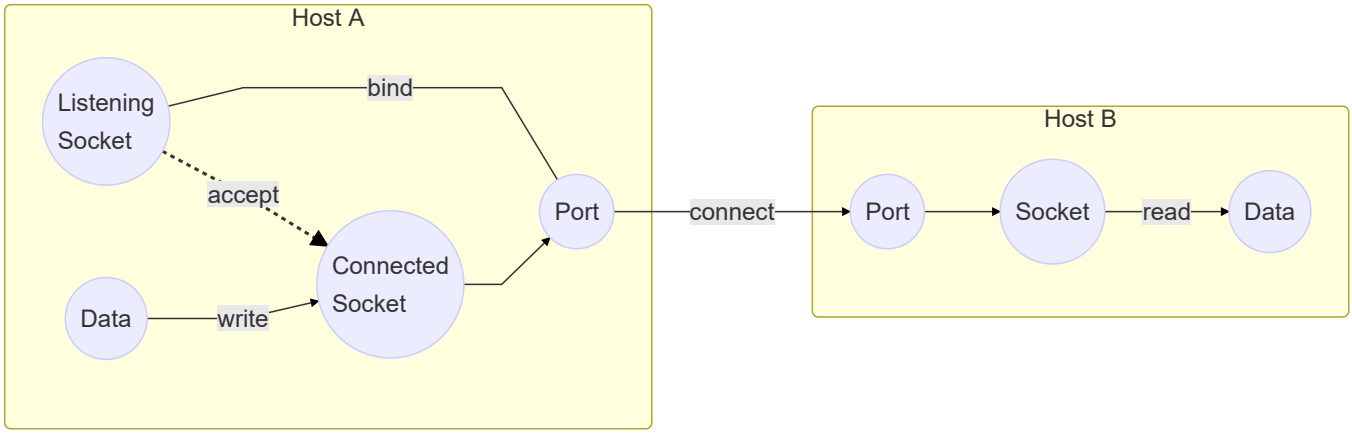
参数说明：

- fd：已建立连接的套接字文件描述符
- buf：发送数据缓冲区
- count：期望发送的字节数

7. read函数

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

该函数用于通过已连接套接字从对方主机接收数据。成功返回实际接收的字节数，失败返回-1，并设置errno变量。如下图所示：



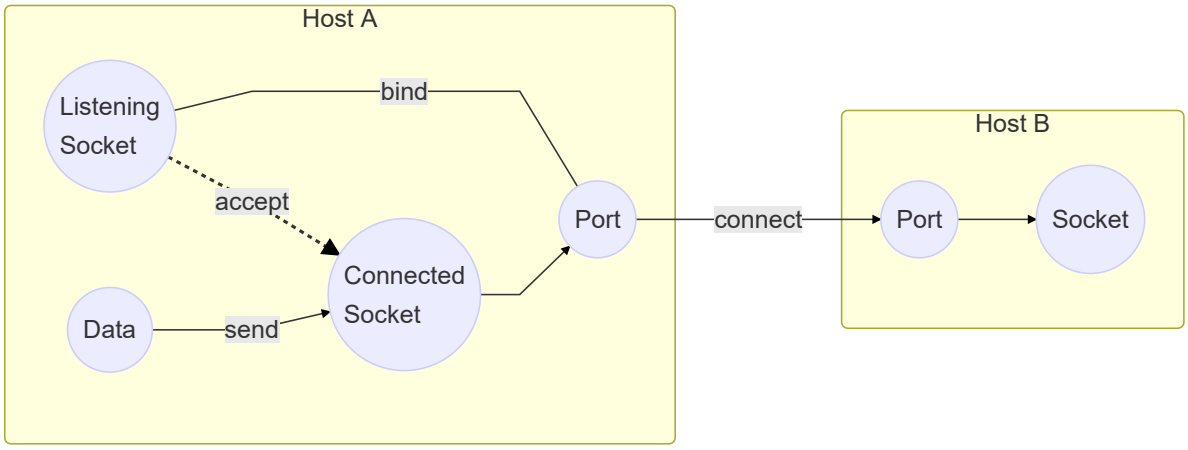
参数说明：

- fd：已建立连接的套接字文件描述符
- buf：接收数据缓冲区
- count：期望接收的字节数

8. send函数

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

该函数用于通过已连接套接字向对方主机发送数据。成功返回实际发送的字节数，失败返回-1，并设置errno变量。如下图所示：



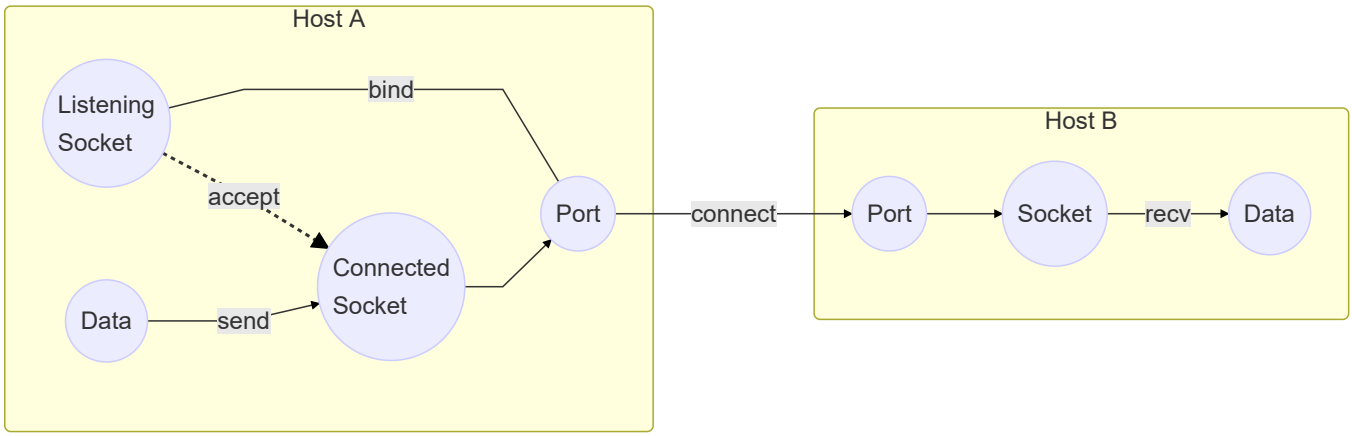
参数说明：

- sockfd：已建立连接的套接字文件描述符
- buf：发送数据缓冲区
- len：期望发送的字节数
- flags：发送标志，取0等价于write函数

9. recv函数

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

该函数用于通过已连接套接字从对方主机接收数据。成功返回实际接收的字节数，失败返回-1，并设置errno变量。如下图所示：



参数说明：

- sockfd：已建立连接的套接字文件描述符
- buf：接收数据缓冲区
- len：期望接收的字节数
- flags：接收标志，取0等价于read函数

10. close函数

```
#include <unistd.h>

int close(int fd);
```

该函数用于关闭套接字及与之相关的网络连接。成功返回0，失败返回-1，并设置errno变量。

参数说明：

- fd：套接字文件描述符

3.2 实训案例

3.2.1 基于DES加密的TCP聊天

在了解DES算法原理的基础上，编程实现对字符串的DES加密、解密操作。

在了解Linux操作系统中TCP套接字工作原理的基础上，编程实现简单的TCP通信。简化编程细节，不要求实现一对多通信。

将以上两部分结合到一起，编程实现对通信内容做DES加解密的TCP聊天程序：

- 通信双方事先约定密钥
- 发送方通过该密钥加密
- 接收方通过该密钥解密
- 网络上传输的都是密文

3.2.2 程序清单

1. 声明Des类

```cpp
// des.h
// 声明Des类

#pragma once

#include <stdint.h>

// 基于DES算法的加解密
class Des {
public:
    // 构造函数
    Des(void);

    // 生成初始密钥
    uint64_t generateKey(void) const;

    // 加密
    int encrypt(uint64_t key, const void* mbuf, size_t mlen,
        void* cbuf, size_t* clen) const;
    // 解密
    int decrypt(uint64_t key, const void* cbuf, size_t clen,
        void* mbuf, size_t* mlen) const;

private:
    // 校验初始密钥
    int verifyKey(uint64_t K0) const;

    // 置换选择PC-1
    void PC1(uint64_t K0, uint32_t* C, uint32_t* D) const;
    // 循环左移
    void LS(int iKey, uint32_t* C, uint32_t* D) const;
    // 置换选择PC-2
    uint64_t PC2(uint32_t C, uint32_t D) const;
    // 生成子密钥
    void generateRoundKeys(uint64_t K0, uint64_t* roundKeys) const;

    // 选择扩展运算
    uint64_t E(uint32_t R) const;
    // 密钥加运算
    uint64_t X(uint64_t R, uint64_t K) const;
    // 选择压缩运算
    uint32_t S(uint64_t R) const;
    // 置换运算
    uint32_t P(uint32_t R) const;
    // 加密变换
    uint32_t F(uint32_t R, uint64_t K) const;

    // 初始置换
    void IP(uint64_t M, uint32_t* L, uint32_t* R) const;
    // 16轮迭代
    void R16(const uint64_t* roundKeys, uint32_t* L, uint32_t* R) const;
    // 逆初始置换
    uint64_t IIP(uint32_t L, uint32_t R) const;
    // 加密分组
    uint64_t M2C(const uint64_t* roundKeys, uint64_t M) const;
    // 解密分组
    uint64_t C2M(const uint64_t* roundKeys, uint64_t C) const;

    static const uint8_t pc1[2][28]; // 置换选择表1
    static const uint8_t ls[16];     // 循环左移表
    static const uint8_t pc2[48];    // 置换选择表2

    static const uint8_t ip[64];   // 初始置换表
    static const uint8_t e[48];    // 扩展变换表
    static const uint8_t s[8][64]; // 压缩变换表
    static const uint8_t p[32];    // 置换变换表
    static const uint8_t iip[64];  // 逆初始置换表

    uint32_t bm32[32]; // 32位位掩码表
```

```
    uint64_t bm64[64]; // 64位位掩码表
};
```

2. 实现Des类

```cpp
// des.cpp
// 实现Des类

#include <time.h>
#include <stdlib.h>
#include <string.h>

#include "des.h"

// 置换选择表
const uint8_t Des::pc1[2][28] = {
    {
        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}};

// 循环左移表
const uint8_t Des::ls[16] = {
    1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1};

// 置换选择表
const uint8_t Des::pc2[48] = {
    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};

// 初始置换表
const uint8_t Des::ip[64] = {
    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};

// 扩展变换表
const uint8_t Des::e[48] = {
    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};

// 压缩变换表
const uint8_t Des::s[8][64] = {
    {
        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}};

// 置换变换表
const uint8_t Des::p[32] = {
    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};

// 逆初始置换表
const uint8_t Des::iip[64] = {
    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};

// 构造函数
Des::Des(void) {
    // 初始化32位位掩码表
    //
    // bm32 0 1 2 ... 31
    //    0 1 0 0 ... 0
    //    1 0 1 0 ... 0
    //    2 0 0 1 ... 0
    //    .
    //    .
    //    .
    //   31 0 0 0 ... 1
    //
    bm32[31] = 1;
    for (int i = 30; i >= 0; --i)
        bm32[i] = bm32[i+1] << 1;

    // 初始化64位位掩码表
    //
    // bm64 0 1 2 ... 63
    //    0 1 0 0 ... 0
    //    1 0 1 0 ... 0
    //    2 0 0 1 ... 0
    //    .
    //    .
    //    .
    //   63 0 0 0 ... 1
    //
    bm64[63] = 1;
    for (int i = 62; i >= 0; --i)
        bm64[i] = bm64[i+1] << 1;
}

// 生成初始密钥
uint64_t Des::generateKey(void) const {
    srand (time (NULL));
    uint64_t K0 = 0;
    int checksum = 0;

    for (int i = 0; i < 64; ++i)
        if ((i + 1) % 8) { // 非校验位
            if (rand () & 1) {
                K0 |= bm64[i];
                ++checksum;
            }
        }
        else { // 校验位
            if (! checksum & 1) // 奇校验
                K0 |= bm64[i];
            checksum = 0;
        }

    return K0;
}

// 加密
int Des::encrypt(uint64_t key, const void* mbuf, size_t mlen,
```

```
    void* cbuf, size_t* clen) const {
    // 检查密文缓冲区长度
    size_t nlen = (mlen / 8 + (mlen % 8 != 0)) * 8;
    if (*clen < nlen) {
        *clen = nlen;
        return -1;
    }

    *clen = nlen;

    // 校验初始密钥
    if (verifyKey(key) == -1)
        return -2;

    // 生成子密钥
    uint64_t roundKeys[16];
    generateRoundKeys(key, roundKeys);

    const uint64_t* M = (const uint64_t*)mbuf;
    uint64_t* C = (uint64_t*)cbuf;

    while (mlen) {
        if (mlen >= 8)
            // 加密分组
            *C = M2C(roundKeys, *M);
        else { // 最后一个分组不够8字节(64位)
            // 尾块补齐
            //
            // M
            // |
            // v
            // +---+---+---+---+---+
            // | X | X | X | X | X |
            // +---+---+---+---+---+
            // |<-----5 Bytes----->|
            //
            // tail:
            // +---+---+---+---+---+---+---+---+
            // | X | X | X | X | X | 0 | 0 | 3 |
            // +---+---+---+---+---+---+---+---+
            //                     |<-3 Bytes->|
            //
            uint8_t tail[8] = {0};
            memcpy(tail, M, mlen);
            tail[7] = 8 - mlen;
            // 加密分组
            *C = M2C(roundKeys, *(uint64_t*)tail);
            break;
        }

        ++C;
        ++M;
        mlen -= 8;
    }

    return 0;
}

// 解密
int Des::decrypt(uint64_t key, const void* cbuf, size_t clen,
    void* mbuf, size_t* mlen) const {
    // 检查明文缓冲区长度
    if (*mlen < clen) {
        *mlen = clen;
        return -1;
    }

    *mlen = clen;

    // 校验初始密钥
```

```
    if (verifyKey(key) == -1)
        return -2;

    // 生成子密钥
    uint64_t roundKeys[16];
    generateRoundKeys(key, roundKeys);

    const uint64_t* C = (const uint64_t*)cbuf;
    uint64_t* M = (uint64_t*)mbuf;

    while (clen) {
        // 解密分组
        *M = C2M(roundKeys, *C);

        if (clen == 8) { // 刚刚解密的是最后一个分组
            uint8_t tail[8];
            memcpy(tail, M, 8);
            if (tail[7] < 8) { // 可能存在尾块补齐
                size_t i;
                for (i = 8 - tail[7]; i < 7 && ! tail[i]; ++i);
                if (i == 7) // 存在尾块补齐
                    *mlen -= tail[7]; // 减去补齐部分的长度
            }
        }

        ++M;
        ++C;
        clen -= 8;
    }

    return 0;
}

// 校验初始密钥
int Des::verifyKey(uint64_t K0) const {
    int checksum = 0;

    for (int i = 0; i < 64; ++i) {
        if (K0 & bm64[i])
            ++checksum;

        if ((i + 1) % 8 == 0) { // 校验位
            if (! checksum & 1) // 奇校验
                return -1;

            checksum = 0;
        }
    }

    return 0;
}

// 置换选择PC-1
void Des::PC1(uint64_t K0, uint32_t* C, uint32_t* D) const {
    *C = *D = 0;

    for (int i = 0; i < 28; ++i) {
        if (K0 & bm64[pc1[0][i]-1])
            *C |= bm32[i];

        if (K0 & bm64[pc1[1][i]-1])
            *D |= bm32[i];
    }
}

// 循环左移
void Des::LS(int iKey, uint32_t* C, uint32_t* D) const {
    uint8_t N = ls[iKey];

    // 左端移出部分掩码：11000...0
```

```
    uint32_t lm = 0;
    for (int i = 0; i < N; ++i)
        lm |= bm32[i];

    *C = *C << N | (*C & lm) >> (28 - N);
    *D = *D << N | (*D & lm) >> (28 - N);
}

// 置换选择PC-2
uint64_t Des::PC2(uint32_t C, uint32_t D) const {
    uint64_t CD = C;
    CD = (CD << 28 | D) << 4;

    uint64_t K = 0;
    for (int i = 0; i < 48; ++i)
        if (CD & bm64[pc2[i]-1])
            K |= bm64[i];

    return K;
}

// 生成子密钥
void Des::generateRoundKeys(uint64_t K0, uint64_t* roundKeys) const {
    // 置换选择PC-1
    uint32_t C, D;
    PC1(K0, &C, &D);

    for (int iKey = 0; iKey < 16; ++iKey) {
        // 循环左移
        LS(iKey, &C, &D);
        // 置换选择PC-2
        roundKeys[iKey] = PC2(C, D);
    }
}

// 选择扩展运算
uint64_t Des::E(uint32_t R) const {
    uint64_t _R = 0;

    for (int i = 0; i < 48; ++i)
        if (R & bm32[e[i]-1])
            _R |= bm64[i];

    return _R;
}

// 密钥加运算
uint64_t Des::X(uint64_t R, uint64_t K) const {
    return R ^ K;
}

// 选择压缩运算
uint32_t Des::S(uint64_t R) const {
    uint64_t gm = 63; // 组掩码：111111
    uint32_t _R = 0;

    for (int i = 0; i < 8; ++i)
        _R |= s[i][(R & gm << (64 - (i + 1) * 6)) >> (64 - (i + 1) * 6)]
            << (32 - (i + 1) * 4);

    return _R;
}

// 置换运算
uint32_t Des::P(uint32_t R) const {
    uint32_t _R = 0;

    for (int i = 0; i < 32; ++i)
        if (R & bm32[p[i]-1])
            _R |= bm32[i];
```

```
    return _R;
}

// 加密变换
uint32_t Des::F(uint32_t R, uint64_t K) const {
    return P(S(X(E(R), K)));
}

// 初始置换
void Des::IP(uint64_t M, uint32_t* L, uint32_t* R) const {
    uint64_t _M = 0;
    for (int i = 0; i < 64; ++i)
        if (M & bm64[ip[i]-1])
            _M |= bm64[i];

    *L = _M >> 32;
    *R = _M;
}

// 16轮迭代
void Des::R16(const uint64_t* roundKeys, uint32_t* L, uint32_t* R) const {
    for (int i = 0; i < 16; ++i) {
        if (i < 15) { // 前15轮迭代
            uint32_t _L = *L;
            *L = *R;
            *R = _L ^ F(*R, roundKeys[i]);
        }
        else // 最后一轮迭代
            *L = *L ^ F(*R, roundKeys[i]);
    }
}

// 逆初始置换
uint64_t Des::IIP(uint32_t L, uint32_t R) const {
    uint64_t LR = L;
    LR = LR << 32 | R;

    uint64_t C = 0;
    for (int i = 0; i < 64; ++i)
        if (LR & bm64[iip[i]-1])
            C |= bm64[i];

    return C;
}

// 加密分组
uint64_t Des::M2C(const uint64_t* roundKeys, uint64_t M) const {
    uint32_t L, R;
    // 初始置换
    IP(M, &L, &R);
    // 16轮迭代
    R16(roundKeys, &L, &R);
    // 逆初始置换
    return IIP(L, R);
}

// 解密分组
uint64_t Des::C2M(const uint64_t* roundKeys, uint64_t C) const {
    // 16个子密钥逆序
    uint64_t inverseKeys[16];
    for (int i = 0; i < 16; ++i)
        inverseKeys[i] = roundKeys[15-i];

    uint32_t L, R;
    // 初始置换
    IP(C, &L, &R);
    // 16轮迭代
    R16(inverseKeys, &L, &R);
    // 逆初始置换
```

```
    return IIP(L, R);
}
```

3. 测试Des类

```
// des_test.cpp
// 测试Des类

#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <iomanip>
using namespace std;

#include "des.h"

int main(void) {
    // 生成初始密钥
    Des des;
    uint64_t key = des.generateKey();
    cout << hex << setfill('0') << setw(16) << key << endl;

    // 明文和密文缓冲区
    char mbuf[1024] = "Give back to Ceasar what is "
        "Ceasar's and to God what is God's";
    uint8_t cbuf[1024];
    size_t mlen = strlen(mbuf), clen;

    // 密文缓冲不足
    clen = 8;
    cout << dec << des.encrypt(key, mbuf, mlen, cbuf, &clen) << endl;
    cout << clen << endl;
    // 密文缓冲充足
    cout << des.encrypt(key, mbuf, mlen, cbuf, &clen) << endl;
    // 打印十六进制密文
    cout << hex;
    for (size_t i = 0; i < clen; ++i)
        cout << setw(2) << (unsigned int)cbuf[i];
    cout << endl;

    // 明文缓冲区不足
    mlen = 8;
    cout << dec << des.decrypt(key, cbuf, clen, mbuf, &mlen) << endl;
    cout << mlen << endl;
    // 明文缓冲区充足
    cout << des.decrypt(key, cbuf, clen, mbuf, &mlen) << endl;
    cout << mlen << endl;
    mbuf[mlen] = '\0';
    cout << mbuf << endl; // 打印明文字符串

    return EXIT_SUCCESS;
}
```

4. 测试Des类构建脚本

```
# des_test.mak
# 测试Des类构建脚本

PROJ   = des_test
OBJS   = des_test.o des.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.

$(PROJ): $(OBJS)
    $(LINK) $^ -o $@

.cpp.o:
    $(CXX) $(CFLAGS) $^

clean:
    $(RM) $(PROJ) $(OBJS)
```

5. 声明SecChat类

```
// secchat.h
// 声明SecChat类

#pragma once

#include <string>
using namespace std;

#include "des.h"

// 安全聊天
class SecChat {
public:
    // 构造函数
    SecChat(const char* port, const char* ip = "");

    // 在侦听套接字上等待并接受对方主机的连接请求
    int accept(void);
    // 向对方主机发起连接请求
    int connect(void);
    // 聊天
    int chat(void) const;

protected:
    // 从标准输入读取字符串发送给对方主机
    int send(void) const;
    // 从对方主机接收字符串打印到标准输出
    int recv(void) const;

    int       sockfd; // 套接字文件描述符
    const Des des;    // 基于DES算法的加解密
    uint64_t  key;    // 密钥

private:
    // SIGTERM(15)信号处理
    static void sigTerm (int sigNum);

    // 生成并发送密钥
    virtual int sendKey(void);
    // 接收并保存密钥
    virtual int recvKey(void);

    static const size_t MAX_MESSAGE; // 最大消息长度

    const string port; // 端口号
    const string ip;   // IP地址
};
```

6. 实现SecChat类

```cpp
// secchat.cpp
// 实现SecChat类

#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <strings.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;

#include "secchat.h"

const size_t SecChat::MAX_MESSAGE = 1024; // 最大消息长度

// 构造函数
SecChat::SecChat(const char* port,
    const char* ip /* = "" */) : port(port), ip(ip) {
    // 设置SIGTERM(15)信号处理
  signal (SIGTERM, sigTerm);
}

// 在侦听套接字上等待并接受对方主机的连接请求
int SecChat::accept(void) {
    // 创建套接字
    int sockls = socket(PF_INET, SOCK_STREAM, 0);
    if (sockls == -1) {
        perror("socket");
        return -1;
    }

    // 为套接字设置重用地址选项
    int on = 1;
    if (setsockopt(sockls, SOL_SOCKET, SO_REUSEADDR, &on,
        sizeof(on)) == -1) {
        perror("setsockopt");
        close(sockls);
        return -1;
    }

    // 绑定本地主机地址和端口
    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);
    bzero(&addr, addrlen);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(port.c_str()));
    addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sockls, (struct sockaddr*)&addr, addrlen) == -1) {
        perror ("bind");
        close(sockls);
        return -1;
    }

    // 设置套接字为侦听状态
    if (listen(sockls, 1024) == -1) {
        perror("listen");
        close(sockls);
        return -1;
    }

    // 等待并接受对方主机的连接请求
    if ((sockfd = ::accept(sockls, (struct sockaddr*)&addr,
        &addrlen)) == -1) {
        perror("accept");
        close(sockls);
```

```
        return -1;
    }

    // 向对方主机发送密钥
    if (sendKey() == -1){
        close(sockfd);
        close(sockls);
        return -1;
    }

    // 关闭侦听套接字
    close(sockls);
    return 0;
}

// 向对方主机发起连接请求
int SecChat::connect(void) {
    // 创建套接字
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return -1;
    }

    // 连接远程主机地址和端口
    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);
    bzero(&addr, addrlen);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(port.c_str()));
    addr.sin_addr.s_addr = inet_addr(ip.c_str());
    if (::connect(sockfd, (struct sockaddr*)&addr, addrlen) == -1) {
        perror("connect");
        close(sockfd);
        return -1;
    }

    // 从对方主机接收密钥
    if (recvKey() == -1) {
        close(sockfd);
        return -1;
    }

    return 0;
}

// 聊天
int SecChat::chat(void) const {
    // 创建子进程
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        close(sockfd);
        return -1;
    }

    if (pid) { // 父进程
        // 无限循环
        for (;;)
            // 从标准输入读取字符串发送给对方主机
            if (send() == -1) {
                close(sockfd);
                // 终止子进程
                kill(pid, SIGTERM);
                return -1;
            }
    }
    else { // 子进程
        // 无限循环
        for (;;)
            // 从对方主机接收字符串打印到标准输出
```

```
            if (recv() == -1) {
                close(sockfd);
                // 终止父进程
                kill(getppid(), SIGTERM);
                return -1;
            }
    }

    // 关闭套接字
    close(sockfd);
    return 0;
}

// 从标准输入读取字符串发送给对方主机
int SecChat::send(void) const {
    // 从标准输入读取字符串并去除结尾换行符
    char mbuf[MAX_MESSAGE+1];
    fgets (mbuf, sizeof(mbuf), stdin);
    ssize_t mlen = strlen(mbuf);
    if (mbuf[mlen-1] == '\n')
        mbuf[--mlen] = '\0';

    // 惊叹号表示退出
    if (! strcmp(mbuf, "!"))
        return -1;

    // 加密
    uint8_t cbuf[MAX_MESSAGE];
    size_t clen = sizeof(cbuf);
    if (des.encrypt(key, mbuf, mlen, cbuf, &clen) < 0) {
        cerr << "Unable to encrypt content" << endl;
        return -1;
    }

    // 向对方主机发送加密后的字符串
    if (::send(sockfd, cbuf, clen, 0) != (ssize_t)clen) {
        perror("send");
        return -1;
    }

    return 0;
}

// 从对方主机接收字符串打印到标准输出
int SecChat::recv(void) const {
    // 从对方主机接收加密后的字符串
    uint8_t cbuf[MAX_MESSAGE];
    ssize_t clen = ::recv(sockfd, cbuf, sizeof(cbuf), 0);

    if (clen == -1) {
        perror("recv");
        return -1;
    }

    if (! clen) {
        cerr << "recv: " << "Connection break" << endl;
        return -1;
    }

    // 解密
    char mbuf[MAX_MESSAGE+1];
    size_t mlen = sizeof(mbuf) - 1;
    if (des.decrypt(key, cbuf, clen, mbuf, &mlen) < 0) {
        cerr << "Unable to decrypt content" << endl;
        return -1;
    }

    // 将解密后的字符串打印到标准输出
    mbuf[mlen] = '\0';
    cout << "> " << mbuf << endl;
```

```
    return 0;
}

// SIGTERM(15)信号处理
void SecChat::sigTerm (int sigNum) {
    exit(EXIT_SUCCESS);
}

// 生成并发送密钥
int SecChat::sendKey(void) {
    // 生成密钥
    key = des.generateKey();

    // 发送密钥
    if (::send(sockfd, &key, sizeof(key), 0) != sizeof(key)) {
        perror("send");
        return -1;
    }

    return 0;
}

// 接收并保存密钥
int SecChat::recvKey(void) {
    // 接收密钥
    ssize_t len = ::recv(sockfd, &key, sizeof(key), 0);

    if (len == -1) {
        perror("recv");
        return -1;
    }

    if (! len) {
        cerr << "recv: " << "Connection break" << endl;
        return -1;
    }

    return 0;
}
```

7. 测试SecChat类

```cpp
// secchat_test.cpp
// 测试SecChat类

#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;

#include "secchat.h"

int main(int argc, char* argv[]) {
    if (argc < 3) // 检查命令行参数个数
        goto escape;

    if (! strcmp(argv[1], "-s")) { // 服务器
        // 安全聊天服务器
        SecChat server(argv[2]);

        // 在侦听套接字上等待并接受对方主机的连接请求
        if (server.accept() == -1)
            return EXIT_FAILURE;

        // 聊天
        if (server.chat() == -1)
            return EXIT_FAILURE;
    }
    else if (! strcmp(argv[1], "-c")) { // 客户机
        if (argc < 4) // 检查命令行参数个数
            goto escape;

        // 安全聊天客户机
        SecChat client(argv[3], argv[2]);

        // 向对方主机发起连接请求
        if (client.connect() == -1)
            return EXIT_FAILURE;

        // 聊天
        if (client.chat() == -1)
            return EXIT_FAILURE;
    }
    else
        goto escape;

    // 返回成功
    return EXIT_SUCCESS;

escape:
    // 打印命令行帮助信息并返回失败
    cerr << "Usage: " << argv[0] << " -s <port>" << endl;
    cerr << "Usage: " << argv[0] << " -c <ip> <port>" << endl;
    return EXIT_FAILURE;
}
```

8. 测试SecChat类构建脚本

```
# secchat_test.mak
# 测试SecChat类构建脚本

PROJ   = secchat_test
OBJS   = secchat_test.o secchat.o des.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.

$(PROJ): $(OBJS)
    $(LINK) $^ -o $@

.cpp.o:
    $(CXX) $(CFLAGS) $^

clean:
    $(RM) $(PROJ) $(OBJS)
```

3.3 扩展提高

3.3.1 DES算法的安全性

1. DES算法的安全性缺陷

DES算法的安全性缺陷主要集中在以下三个方面：

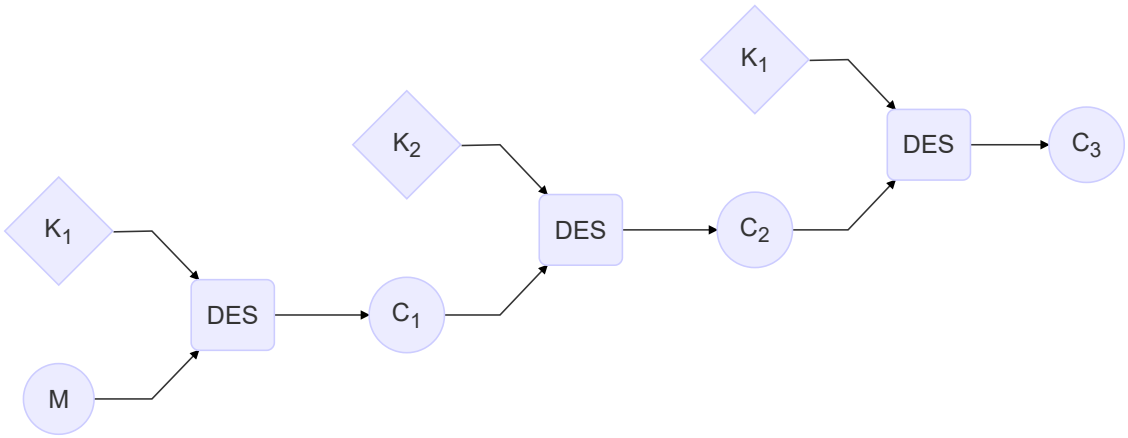
- 64位加密分组仅占8个字节，对于数据传输而言实在太小
- 64位初始密钥(实际56位，另8位用于奇偶校验)过于短小，16轮子密钥由递推产生，这都为针对密钥的暴力破解提供了可能
- 不能对抗差分和线性密码分析

2. 多重DES算法

多重DES算法，即用多个不同的密钥连续多次对一组明文进行加密。其中最常用的是三重DES算法：

- 先用64位(含8个奇偶校验位)密钥K_1对明文进行加密
- 再用64位(含8个奇偶校验位)密钥K_2对上一步加密的结果进行加密
- 最后用密钥K_1对上一步加密的结果再加一次密

如下图所示：



三重DES算法的密钥(K_1K_2)总长度为128位(含16个奇偶校验位)，在可以预见的未来被认为是合适的、安全的，截至目前还没有找到针对此方案的攻击方法。但是三重DES算法的耗时是普通DES算法的三倍，时间开销比较大。

3. 密钥管理

密钥管理的两大核心任务：

- 设置密钥更新周期
  - 密钥长度
  - 被保护信息的敏感程度
  - 系统环境的安全状况
  - 必要的安全冗余
- 归档废弃密钥备用

3.3.2 AES算法

由于DES及其改进算法的安全强度已经越来越难以满足新的安全需求，尤其是在对抗20世纪末出现的差分和线性密码分析方法时，更加显得不堪一击。美国政府从1997年开始公开征集新的数据加密标准(Advanced Encryption Standard, AES)以取代DES算法。经过三轮筛选，最后选中比利时密码学家Joan Daemen和Vincent Rijmen提出的密码算法Rijndael作为AES正式取代DES。

1. AES算法描述

AES算法的简单描述如下：

- 将16字节(128位)明文每4字节一列排成方阵
- 将16字节(128位)初始密钥每4字节一行排成方阵
- 将初始密钥扩展为11个子密钥方阵
- 迭代11轮，每轮使用一个子密钥
  - 如果不是第一轮
    - 字节置换
    - 行移位
    - 如果不是最后一轮
      - 列混淆
  - 密钥加运算
- 按行重新组合成16字节(128位)密文

也可以将字节置换、行移位和列混淆，合并为4张加密置换表，用查表的方式实现每一轮迭代。

2. AES算法与DES算法的比较

AES算法与DES算法的时间比较，如下表所示：

算法	加密(秒)	解密(秒)
DES	18	18
AES	6	10

AES算法与DES算法的效率比较，如下表所示：

算法	加密(M位/秒)	解密(M位/秒)
DES	1.676	1.676
AES	5.027	3.016

两种算法相比：

- 相比DES算法使用64位分组，AES算法使用128位分组，加密解密效率更高
- 相比DES算法使用64位密钥，AES算法使用128位密钥，暴力破解难度更大

3.3.3 高级通信函数

1. flags

send/recv函数提供了与write/read函数类似的功能，不同之处在于前者增加了第四个参数flags作为发送/接收标志：

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

flags可以是0或以下标志的组合：

- MSG_DONTROUTE：仅用于send函数，意在告诉IP协议，目的主机就在本地网络中，不必查找路由表，通常用在网络诊断或路由程序中
- MSG_OOB：收发带外(Out Of Band, OOB)数据。带外数据是TCP传输中独立且享有更高优先级的数据通道，它们可以独立于普通数据传送给用户
- MSG_PEEK：仅用于recv函数，从系统缓冲区中读取内容，但并不从系统缓冲区中清除该内容
- MSG_WAITALL：仅用于recv函数，等所有信息都到达时才返回。使用该标志的recv函数会一直阻塞，直到接收完期望接收的字节数或者发生错误。这时recv函数的返回值会有三种情况：
  - 接收到期望接收的字节数，返回值等于其第三个参数len
  - 等待过程中对方关闭了连接，即读到文件结尾，返回值小于len甚至可能是0
  - 发生错误，返回-1，同时设置errno变量

当flags为0时，send/recv函数与write/read函数完全等价。

2. shutdown

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

TCP连接是全双工的，可以同时读写一个流式套接字。对该套接字调用close函数，会将其读写通道同时关闭，而如果调用shutdown函数，则可通过其how参数指定不同的关闭方式：

- 0：只关闭读通道，该套接字不可读取，但仍可继续写入
- 1：只关闭写通道，该套接字不可写入，但仍可继续读取
- 2：读写通道都关，等价于close函数

达内集团◇C++教研部◇闵卫