DES 加密算法详细分析

作者　Tomcoding

2019 年 5 月 1 日

前言

很久以前用汇编语言实现过 DES 算法，时间久了细节都记不清楚了，后来想复习一下，到网上搜了一下，发现大部分都是转来转去的帖子和博文，原始的作者也都不知道是谁了。找了几个介绍比较详细的文章仔细研究了一下，原理和大部分细节都搞清楚了，于是想用 C 语言实现这个算法。还是在网上搜了一下，发现这些源代码都是拿数组来代替位操作，把 C 语言中很好的位操作给摒弃掉了，也没有耐心继续找下去，决定自己写一个，体现算法中的细节描述。

这个程序的目的是对照着文档中对 DES 算法的介绍，把每个步骤都用函数实现了，使得想学习 DES 算法的朋友可以很容易的理解每一步操作，也能体会编程中的一些细节。

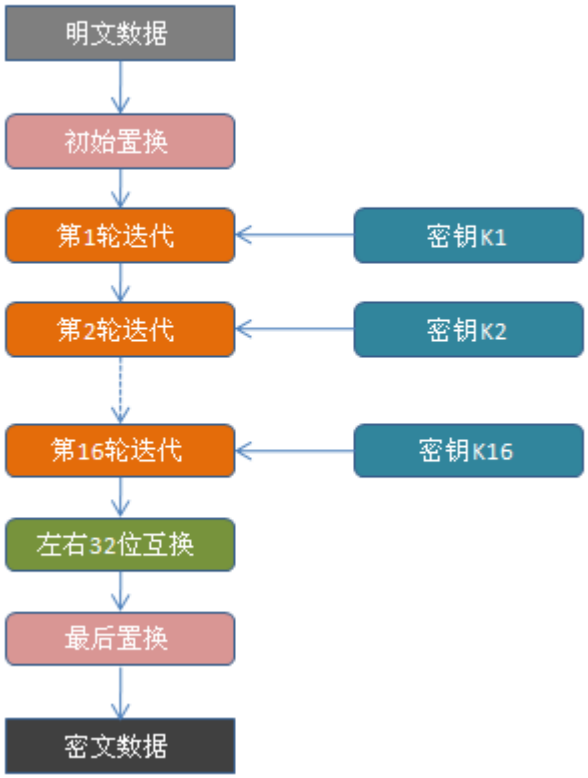
算法介绍

DES 算法是对称加密算法，以 64 位分组对数据进行加密，DES 算法的加密和解密使用了相同的算法。算法中都是按位操作的，数据和密钥都是 64 位，8 个字节。DES 算法已经使用了几十年的时间，但是对于理解密码学中的基本概念和算法的设计还是有重要意义的，理解了 DES 算法，再去学习其他的加密算法就会容易一些。

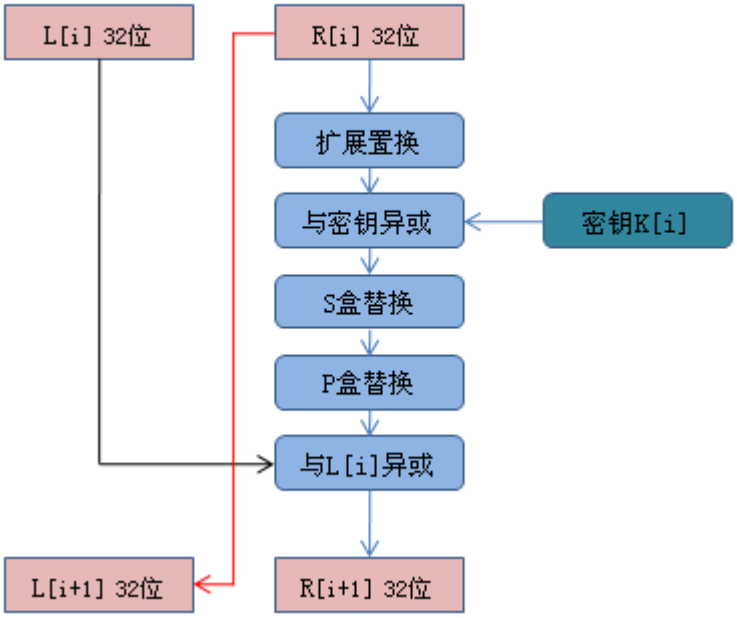
加密算法

DES 算法的输入是一组 8 字节（64 位）的明文数据，这 8 个字节先进行一个初始置换（Initial Permutation），把数据原来的位顺序打乱，然后把数据分成左右两组，前 4 个字节（32 位）是左边的组，后 4 个字节（32 位）是右边的组。把右边的组进行一系列运算，然后与左边的组做异或操作，新生成的 4 字节作为下一次计算的右边组，原来右边的组直接作为下一次计算的左边组，这样的计算轮次一直循环进行 16 次，最后把左右两组互换，还组成一个 64 位的字节序列，再进行最后一次数据置换，这样生成的数据就是密文数据。画个图来说明一下。

加密示例图



第i次迭代示例图



从上面的图中可以看出每次迭代都是对右边的组进行的运算，先进行一个扩展置换，把 32 位数据扩展成 48 位，然后与相应轮的 48 位密钥异或操作，再通过 S 盒替换选择，把 48 位数据再变回 32 位数据，再通过 P 盒替换进一步混淆，最后与左边组的 32 位数据进行异或

操作，作为新一轮的右边组数据，原来运算前的右边组数据直接作为左边组进行新一轮循环。

下面的章节中我们逐个分析一下每个步骤中都进行了什么样的操作。

初始置换

初始置换（Initial Permutation）是通过一个置换表把 64 位的位置重新排列了一次，生成了新的位顺序。置换表的定义如下。

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

表格中的数字是每一位在置换前的位置，取值范围是 1-64，不能把它当做数据替换。从上表中看到，原来数据的第 58 位成为置换后数据的第 1 位，第 50 位成为置换后的第 2 位，一直到第 7 位成为置换后的第 64 位。在数据初始置换后就会分成两组，前面的 4 个字节一组是 L_0（Left Half），后面的 4 个字节一组是 R_0（Right Half），作为初始分组进行下面的迭代运算。

扩展置换

扩展置换（Expansion）是要把右边组的 32 位数据扩展成 48 位，因为接下来一步要与密钥进行异或运算，这里的密钥是由原始密钥生成的子密钥，是 48 位长度，所以右边组的

32 位数据也要变成 48 位才行。扩展置换也是通过一个表来替换完成的。

32	01	02	03	04	05
04	05	06	07	08	09
08	09	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	01

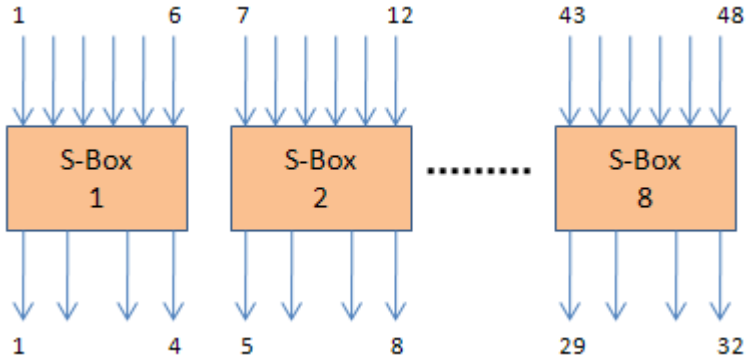
从上面可以很清楚的看到，中间的是原来的 32 位数据，两边红色的是扩展出来的数据，扩展的原则就是原来最右边的扩展下一位的数据，原来最左边的扩展上一位的数据，比如原来的 08 在第二行最右边，扩展下一位就是 09。

与密钥异或

这里的密钥是生成的子密钥（生成的方法在下面讨论），长度是 48 位，我们上面的 R_i 已经扩展出了 48 位，正好一一对应进行异或操作就可以了。

S 盒替换

在上面的扩展置换后，32 位数据变成了 48 位，但是为了后面的运算，还要再变回 32 位，S 盒（S-Box）替换就是实现这个功能的，S 盒有 6 位输入 4 位输出，48 位的输入就需要有 8 个 S 盒完成替换，替换后就形成了 32 位的数据。



8 个 S 盒的定义如下。

S-box 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-box 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-box 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-box 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-box 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-box 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-box 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-box 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

每个 S 盒都是一个 4 行 16 列的表，输入是一个 6 位值，高位和低位组成一个 0-3 的数作为行坐标，中间的四位组成 0-15 的数作为列坐标，在对应的 S 盒中定位到一个数据，这个数值是 4 位的，作为输出。注意在这里 S 盒中的数是真正的数值，不是位置。举个例子，我们以第 8 个 S 盒进行替换，输入的 6 位二进制值是 011001，高位和低位的组合是 01，那么行值是 1，中间 4 位是 1100=0x0C=12，那么列值是 12，查表 S-box 8 得到的值是 0。

P 盒替换

P 盒替换也是输入位的位置替换，是一个 32 位替换 32 位的操作，与初始置换类似。P

盒的定义如下。

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

轮迭代

轮迭代就是把每一轮的右面 32 位分组进行上面的运算，然后与左边的 32 位分组进行异或，作为新一轮的右面分组，左边的分组拷贝运算前右面的分组数据作为新一轮左面分组。一直循环 16 轮，完成全部迭代。

左右互换

在进行完 16 轮迭代后，我们得到 L_{16} 和 R_{16} 两个分组，把这两个分组互换位置，连接成一个右面在前左面在后的顺序，形成新的 64 位数据。

最后置换

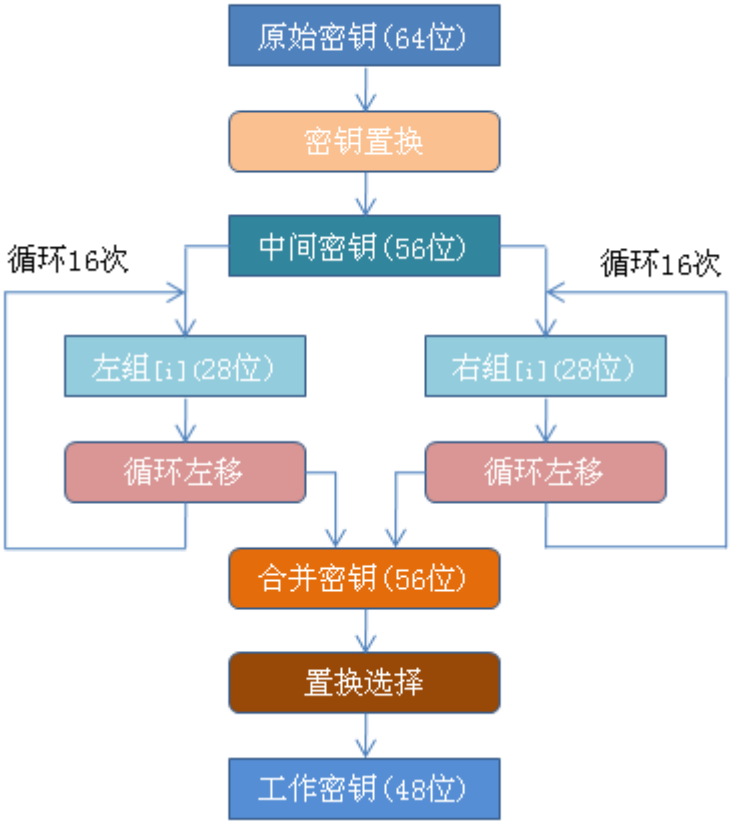
在上面的左右组互换后的 64 位数据还要经过一个最终置换（Final Permutation）的过程，才能产生密文数据，这个最终置换也可以说是初始置换的逆置换，与初始置换一样，只是置换表的差别而已。最终置换表定义如下。

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

密钥生成

原始的密钥有 64 位，从上面的加密算法中看到，要从原始密钥中产生出 16 组 48 位的工作密钥。64 位密钥先进行一次置换，产生 56 位的中间 Key，然后分为左右两组，每组 28 位，这两组 Key 按照一个表每轮进行循环移位（Rotation），合并后再通过一个置换表选出 48 位的工作密钥，移位后的每组数据进入下一个循环，一共进行 16 轮，产生出 16 组密钥。

密钥生成示例图



密钥置换

密钥置换与数据的初始置换一样也是位置转变，只不过密钥置换后产生的数据是 56 位而已，密钥置换表如下。

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

这是一个 56 个元素的表，在输入时每个字节的第 8 位不参与（这里说的第 8 位是从左到右的最后一位），因此表中没有 8 的整数倍的位数。

密钥循环左移

密钥分组的循环左移，根据轮数不同，左移的位数也不同，有的移一位，有的移两位，根据下面的移位表来确定。

轮数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
位数	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

合并密钥

这一步很简单，就是把两个 28 位的分组，按照左边在前，右边在后的顺序合并成 56 位的中间值。

置换选择

这一步是把上面生成的 56 位中间密钥通过一个置换表选择出 48 位的工作子密钥。置换表的定义如下。

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

解密算法

解密算法是加密的逆运算，DES 解密使用与加密相同的算法，把密文输入，使用相同的密钥，最后输出的就是明文数据。由于是逆运算，所以密钥使用的顺序与加密是相反的，即

第一轮循环使用 key16，第二轮使用 key15，依次到第十六轮使用 key1。

程序设计

从上面的文档分析中看到 DES 算法用到的函数并不多，我们把几个主要的函数设计一下，其他很简单的运算函数，编码时直接写代码就可以了。

置换函数

从上面的算法描述中我们看到了大量的置换操作，这些操作的原理是一样的，都是通过一个表把原始数据的位转换到一个新的位置。作为一个例子，我们还是来分析一下初始置换，为了方便，我们只看第一行。

58	50	42	34	26	18	10	2

置换后新数据的第 1 位是原来的 58 位，第 2 位是原来的 50 位，第 3 位是原来的 42 位，第 4 位是原来的 34 位，等等。如果原来的 58 位是 1，那么新的第 1 位也为 1，否则第 1 位就是 0。这个操作看起来非常简单，用程序实现时我们首先会想到，建立一个循环，遍历表中的所有元素，检查原始数据在表中的位置对应的位是 0 还是 1，是 1 则在新的位置设置成 1，是 0 则在新的位置设置成 0。这个想法一点问题都没有，可是怎样检查位和设置位呢？这里的 64 位可是分散在 8 个字节中的，按照从左到右的顺序排列。还是看一个例子，我们通过 4 个字符‘ABCD’，看看每位在字节中的规律。

字符	A	B	C	D
十六进制	0x41	0x42	0x43	0x44
二进制	0100 0001	0100 0010	0100 0011	0100 0100

位下标	0	1	2	3	4	5	6	7
在字节中的下标	7	6	5	4	3	2	1	0
第0字节	0	1	0	0	0	0	0	1
位下标	8	9	10	11	12	13	14	15
在字节中的下标	7	6	5	4	3	2	1	0
第1字节	0	1	0	0	0	0	1	0
位下标	16	17	18	19	20	21	22	23
在字节中的下标	7	6	5	4	3	2	1	0
第2字节	0	1	0	0	0	0	1	1
位下标	24	25	26	27	28	29	30	31
在字节中的下标	7	6	5	4	3	2	1	0
第3字节	0	1	0	0	0	1	0	0

我们来看看第 26 位，19 位，和 15 位在字节中对应的位置。

位数	字节编号	字节中的位
26	3	5
19	2	4
15	1	0

位数	字节编号	字节中的位	除以8的余数	位和余数关系
26	3=26/8	5	26%8=2	5+2=7
19	2=19/8	4	19%8=3	4+3=7
15	1=15/8	0	15%8=7	0+7=7

从上面看到第 n 位在哪个字节中很好确定，正好等于 n/8，第 n 位在字节中的位置与除以 8 后的余数也是有关联的，它们加在一起正好是 7，就是一个字节中的最高位，在字节中位是从右到左排列的。比如第 26 位，在字节中的位是 5，也就是 0x01 左移 5 位就是第 26 位在字节 3 中的位置，26 除以 8 的余数是 2，7 减去 2 正好就是 5，用公式表示就是 0x01<<(7 - 26%8)。如果是第 n 位，那么所在的字节是 n/8，在字节中的位数是 0x01<<(7 - n%8)。通过上面的分析，我们定义检查位的宏如下。

```
#define GET_BIT(p, n) p[n/8] & (0x01<<(7-n%8))
```

我们再想想，其实如果在一个字节中从高位往低位看，比如还是第 26 位，那么它的位置就是第 2 位（从 0 开始计算），如果用 0x80 右移 2 位，正好是它在字节中的位置，所以我们可以把上面的宏改写一下，顺便把设置位和清除位的宏也定义如下。

```
#define GET_BIT(p, n) p[n/8] &    (0x80>>n%8)
```

```
#define SET_BIT(p, n) p[n/8] |=  (0x80>>n%8)
#define CLR_BIT(p, n) p[n/8] &= ~(0x80>>n%8)
```

写的严谨一些，把变量都加上括号，就是下面的样子。

```
#define GET_BIT(p, n)  (p)[(n)/8] &   (0x80>>((n)%8))
#define SET_BIT(p, n)  (p)[(n)/8] |=  (0x80>>((n)%8))
#define CLR_BIT(p, n)  (p)[(n)/8] &= ~(0x80>>((n)%8))
```

有了上面定义的位操作的宏，我们就可以写置换函数了，先看看初始置换函数。

```
static uint8_t data_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
};
static void data_initial_permutation(uint8_t *in, uint8_t *out)
{
  int   i;

  for (i=0; i<64; i++)
  {
    if (GET_BIT(in, data_ip[i]-1))
      SET_BIT(out, i);
    else
      CLR_BIT(out, i);
  }
}
```

在程序的第一个版本中，为了看起来与文档中的描述符合，我们为每一个置换都写了一个函数，其实这些函数都是一样的，只是引用的置换表和置换的位数不一样而已，我们完全可以只写一个函数来代替，把置换表和置换位数当做参数传给函数即可。在程序的第二版中我们把置换函数简化成了一个，如下所示。

```
static void permutation(uint8_t *in, uint8_t *out, uint8_t *pc, int bits)
```

```
{
  int   i;

  for (i=0; i<bits; i++)
  {
    if (GET_BIT(in, pc[i]-1))
      SET_BIT(out, i);
    else
      CLR_BIT(out, i);
  }
}
```

密钥左移函数

密钥左移函数在描述中看起来很简单，但在实现时还是有些麻烦，因为 28 位是 3 个半字节，循环移位时半个字节也要通过一个字节的移位来实现，好在只有 4 个字节的操作，我们把 4 个字节都手工移一遍，也不用考虑什么循环了。

```
static void key_bits_rotation(uint8_t *half, int r)
{
  uint8_t       hb;    /* 最高位标记 */

  /* half[0], half[1], half[2], half[3] */
  hb = half[0] & 0x80;

  /* 移位的原则就是，下一个字节的高位是 1，那么移位后低位补 1 */
  half[0] = (half[1] & 0x80) ? (half[0]<<1 | 1) : (half[0]<<1);
  half[1] = (half[2] & 0x80) ? (half[1]<<1 | 1) : (half[1]<<1);
  half[2] = (half[3] & 0x80) ? (half[2]<<1 | 1) : (half[2]<<1);
  half[3] = (hb) ? (half[3]<<1 | 0x10) : (half[3]<<1);

  /* 由于移位表中只有 1 和 2 两种情况，所以为 2 时，再移一次 */
  if (key_rotation[r] == 2)
  {
    hb = half[0] & 0x80;

    half[0] = (half[1] & 0x80) ? (half[0]<<1 | 1) : (half[0]<<1);
    half[1] = (half[2] & 0x80) ? (half[1]<<1 | 1) : (half[1]<<1);
    half[2] = (half[3] & 0x80) ? (half[2]<<1 | 1) : (half[2]<<1);
    half[3] = (hb) ? (half[3]<<1 | 0x10) : (half[3]<<1);
  }
}
```

密钥生成函数

密钥生成函数流程比较清晰，根据文档中的图，把函数组合起来就可以了。输出时要定义一个 6*16 字节大小的数组存放生成的子密钥。

```
void key_generation(uint8_t *key, uint8_t *subkey)
{
  int           i;
  uint8_t       left[4], right[4];      /* left & right half data */
  uint8_t       pk[7];                  /* permuted key */

  key_permutation(key, pk);

  memcpy(left, pk, 4);
  left[3] &= 0xF0;

  /* 0 1 2 3  4 5 6 */
  right[0] = pk[3]<<4 | pk[4]>>4;
  right[1] = pk[4]<<4 | pk[5]>>4;
  right[2] = pk[5]<<4 | pk[6]>>4;
  right[3] = pk[6]<<4;


  for (i=0; i<16; i++)
  {
    key_bits_rotation(left,  i);
    key_bits_rotation(right, i);

    memcpy(pk, left, 4);
    pk[3] |= right[0]>>4;
    pk[4]  = right[0]<<4 | right[1]>>4;
    pk[5]  = right[1]<<4 | right[2]>>4;
    pk[6]  = right[2]<<4 | right[3]>>4;

    key_selection(pk, &subkey[i*6]);
  }
}
```

加密函数

加密函数的流程也很简单，也有文档中的流程图做参照。

```
void des_encrypt(uint8_t *data, uint8_t *subkey)
{
  int           i, j;
  uint8_t       middle[8], temp[4];
  uint8_t       left[4], right[4];

  /* 初始置换 */
  data_initial_permutation(data, middle);

  /* 中间数据分为左右两个初始分组 */
  memcpy(left, middle, 4);
  memcpy(right, middle+4, 4);

  for (i=0; i<16; i++)
  {
    /* 右分组扩展，从 32 位到 48 位*/
    data_bits_expansion(right, middle);

    /* 与这一轮的子密钥进行异或操作 */
    for (j=0; j<6; j++)
    {
      middle[j] ^= subkey[i*6+j];
    }

    /* 通过 S 盒置换，从 48 位到 32 位*/
    data_substitute(middle, temp);

    /* P 盒置换，从 32 位到 32 位 */
    data_permutation(temp, middle);

    /* 与左分组进行异或操作 */
    for (j=0; j<4; j++)
    {
      temp[j] = middle[j] ^ left[j];
    }

    /* 左分组直接从原来的右分组拷贝过来，右分组从上面的运算获得 */
    memcpy(left, right, 4);
    memcpy(right, temp, 4);
  }

  /* 左右分组互换 */
  memcpy(middle, right, 4);
  memcpy(middle+4, left, 4);
```

```
  /* 进行最后置换 */
  data_final_permutation(middle, data);
}
```

解密函数

解密函数与加密函数一样，只是与子密钥异或的顺序倒过来，从 K[15]到 K[0]。

```
void des_decrypt(uint8_t *data, uint8_t *subkey)
{
  int           i, j;
  uint8_t       middle[8], temp[4];
  uint8_t       left[4], right[4];


  data_initial_permutation(data, middle);

  memcpy(left, middle, 4);
  memcpy(right, middle+4, 4);

  for (i=15; i>=0; i--)
  {
    /* 32 bits to 48 bits */
    data_bits_expansion(right, middle);

    /* exclusive or with subkey */
    for (j=0; j<6; j++)
    {
      middle[j] ^= subkey[i*6+j];
    }

    /* 48 bits to 32 bits */
    data_substitute(middle, temp);

    /* 32 bits to 32 bits */
    data_permutation(temp, middle);

    for (j=0; j<4; j++)
    {
      temp[j] = middle[j] ^ left[j];
    }
```

```
    memcpy(left, right, 4);
    memcpy(right, temp, 4);
  }

  memcpy(middle, right, 4);
  memcpy(middle+4, left, 4);

  data_final_permutation(middle, data);
}
```