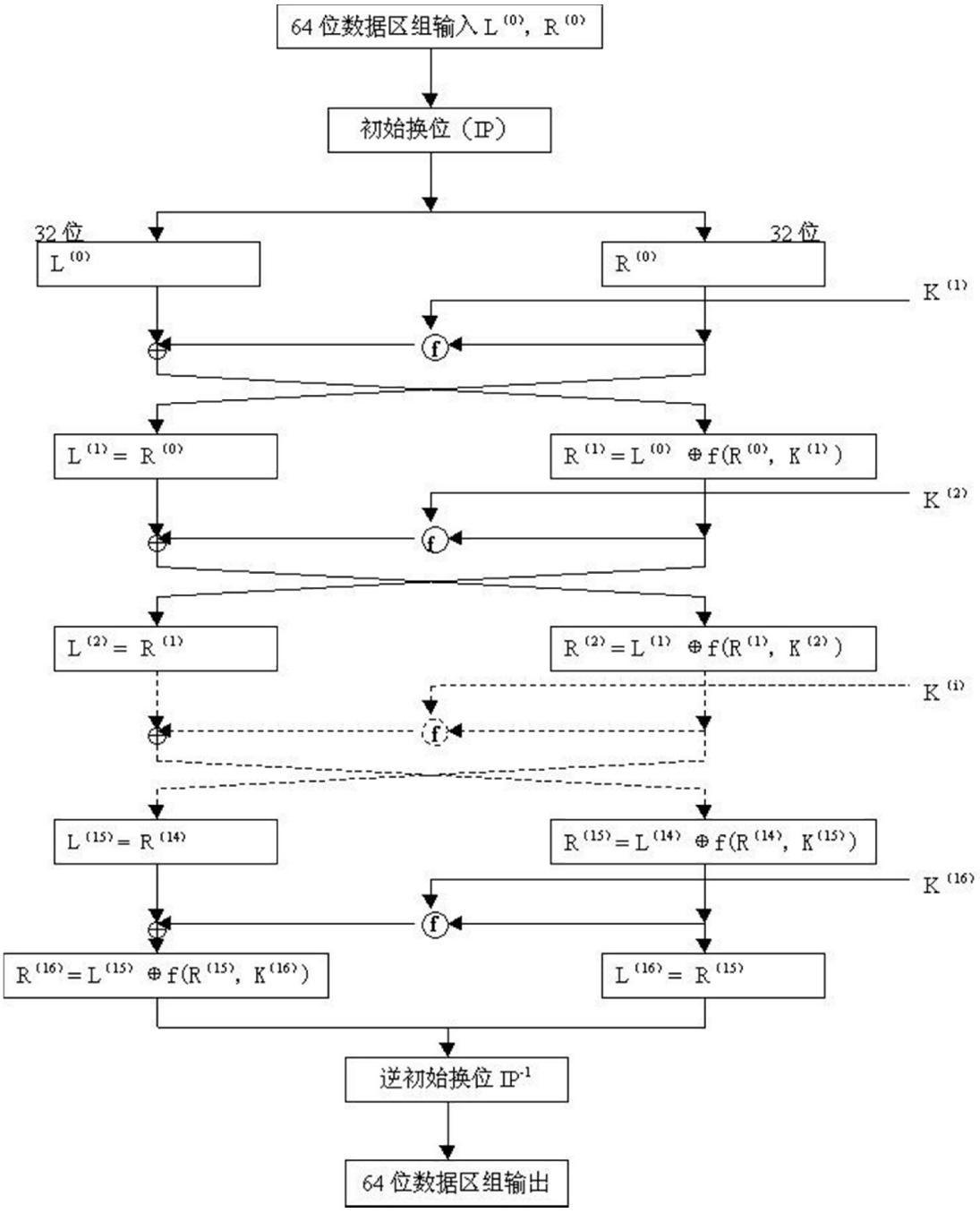
DES 算法分析 – By Lightless

DES 算法流程图

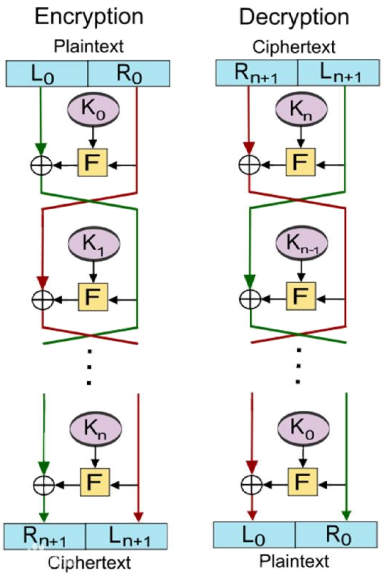


DES 是一种对称的分组密码，每个加密块为 64bit，即 8 个 ASCII 字符。密钥也为 64bit，但是加密过程中实际使用的只有 56bit，因为包含了 8 个校验位。分别是第 8 位，16 位……第

64 位。

这个分组密码使用的是 Feistel 的迭代结构，共进行迭代 16 次。每次迭代使用的密钥都是循环产生，叫做子密钥。

Feistel 密码结构主要如下图



主要思想是将明文平均分成两块，记为 L0 和 R0。在每一轮的迭代中都有如下的计算，

$L_{i+1} = R_i$

$R_{i+1} = L_i$ xor $F(R_i, K_i)$

所得到的结果为(R_{i+1}, L_{i+1})

这种结构简单，但是密码的扩散性较慢，至少需要两轮迭代才可以改变输入的每一个比特。

加密流程

在进行 DES 计算之前要解决的一个问题就是如何把字符串转换成二进制的比特位。先来考虑把单个字符转换为二进制，可以采用模拟除二取余的方式来进行，但是考虑到如果进行大文件的加解密，这种方法无疑会减慢计算速度。所以采用移位的方式来进行。

```
int ByteToBit(char ch, char bit[8])
{
    int cnt;
    ZeroMemory(bit, 8);
    for (cnt = 0; cnt < 8; ++cnt)
    {
        *(bit+cnt) = (ch>>cnt)&1;
    }
    return 0;
}
```

采用移位的方式可以提高运算速度。接下来考虑如何把 8bit 的二进制字串转换成字符，同

样采用移位和位运算的方式来实现。

```
int BitToByte(char bit[8], char *ch)
{
    int cnt;
    for(cnt = 0;cnt < 8; cnt++)
    {
        *ch |= *(bit + cnt)<<cnt;
    }
    return 0;
}
```

具体的原理随便找个字符算一下就知道了。还是比较简单的。下面就是要把长度为 8 的字符串转换成 64bit 的二进制。**在这里要说一点，静态的东西虽然看起来有点挫，但是效率一定比 malloc 那些看起来很牛逼的东西要高。**

把长度为 8 的字符串转换为 64bit 的二进制只需要循环调用 ByteToBit 这个函数，每次处理一个字符就可以了。

```
int Char8ToBit64(char ch[8], char bit[64])
{
    int cnt;
    ZeroMemory(bit, 64);
    for (cnt = 0; cnt < 8; cnt++)
    {
        ByteToBit(*(ch+cnt), bit+(cnt<<3)); //2^3 = 8
    }
    return 0;
}
```

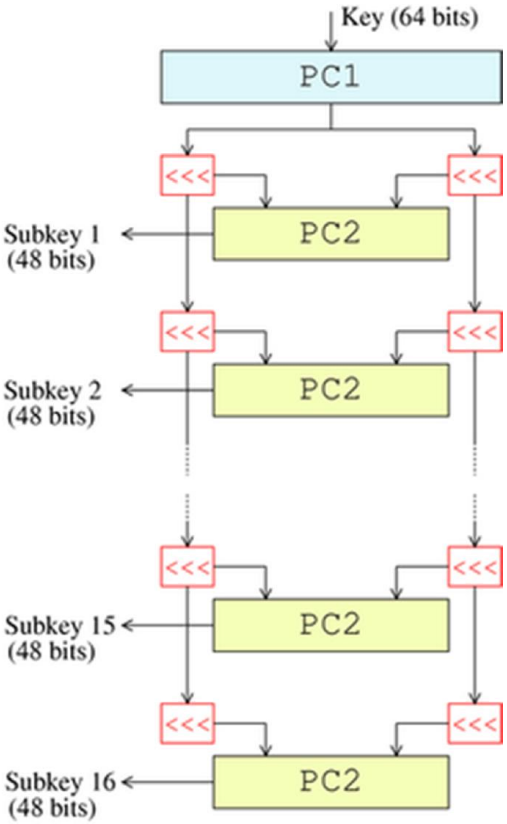
这里使用了一个 bit+(cnt<<3)，左移 3 位就是 2^3=8，在 bit 的基础上加 8，就是这个 bit 数组的下一行（如果将这个 64 长度的数组看作一个 8*8 的矩阵的话）。转成了二进制还应该把 64bit 转为一个长度为 8 的字符串。同理循环调用 BitToByte 就 OK 了。

```
int Bit64ToChar8(char bit[64], char ch[8])
{
    int cnt;
    memset(ch, 0, 8);
    for (cnt = 0; cnt < 8; cnt++)
    {
        BitToByte(bit+(cnt << 3), ch+cnt);
    }
    return 0;
}
```

现在已经解决了字符串和二进制比特的转换问题，可以开始加密了。

密钥调度

刚刚已经说过 DES 的密钥为 64bit，实际使用其中的 56bit。下图为密钥调度的流程图。



图中的 PC1 和 PC2 分别指选择置换 1 和选择置换 2 函数。大概是这个样子的：其中 PC-1 是从 64bit 的输入密钥中选出 56bit（通过选择置换表 1），剩下的 8 位通常作为校验位来处理。选出来的 56bit 将分为两组，每组 28bit，称为 C0 和 D0。将按照如下的方式进行迭代循环，共迭代 16 次，得到 16 个子密钥（轮密钥？）。

$C_i = C_{i-1} <<< r_i$

$D_i = D_{i-1} <<< r_i$

$K_i = PC\text{-}2(C_iD_i)$

其中<<<表示循环左移，PC-2 表示从 56bit 中选择 48bit，r_i 是循环移位的常量，一般为 1 或 2。

```
//置换选择1
int PC_1[56] = {
    56,48,40,32,24,16,8,
    0,57,49,41,33,25,17,
    9,1,58,50,42,34,26,
    18,10,2,59,51,43,35,
    62,54,46,38,30,22,14,
    6,61,53,45,37,29,21,
    13,5,60,52,44,36,28,
    20,12,4,27,19,11,3
};

//置换选择2
int PC_2[48] = {
    13,16,10,23,0,4,2,27,
    14,5,20,9,22,18,11,3,
```

```
    25,7,15,6,26,19,12,1,
    40,51,30,36,46,54,29,39,
    50,44,32,46,43,48,38,55,
    33,52,45,41,49,35,28,31
};

//对左移次数的规定
int MOVE_TIMES[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
```

下面从代码的层次来实现。先来写两个置换函数PC1和PC2。

```
int PC1_Transform(char key[64], char tempbts[56])
{
    int cnt;
    for (cnt = 0; cnt < 56; ++cnt)
    {
        tempbts[cnt] = key[PC_1[cnt]];
    }
    return 0;
}
```

只是简单的进行替换，很快就能写出PC2。

```
int PC2_Transform(char key[56], char tempbts[48])
{
    int cnt;
    for(cnt = 0; cnt < 48; cnt++)
    {
        tempbts[cnt] = key[PC_2[cnt]];
    }
    return 0;
}
```

下面要写一个循环左移的函数，这里的左移不能使用<<来进行，必须要自己来实现了。

```
int ROL(char data[56], int time)
{
    char temp[56];

    //保存将要循环移动到右边的位
    memcpy(temp,data,time);
    memcpy(temp+time,data+28,time);

    //前28位移动
    memcpy(data,data+time,28-time);
    memcpy(data+28-time,temp,time);

    //后28位移动
    memcpy(data+28,data+28+time,28-time);
    memcpy(data+56-time,temp+time,time);
```

```
    return 0;
}
```

除此之外有一种比较简单的写法

```
    ror eax,cl ==> eax=(eax>>cl)+(eax<<(32-cl));
    rol eax,cl ==> eax=(eax<<cl)+(eax>>(32-cl));
```

这是遇到ror汇编指令时，将其转换成C的方式，可以自己尝试实验一下。这里不采用了。现在可以写产生子密钥的函数了。

```
int MakeSubKeys(char key[64], char subKeys[16][48])
{
    char temp[56];
    int cnt;
    //先进行PC1置换
    PC1_Transform(key, temp);
    //16轮迭代，产生16个子密钥
    for (cnt = 0; cnt < 16; ++cnt)
    {
        ROL(temp, MOVE_TIMES[cnt]);
        PC2_Transform(temp, subKeys[cnt]);
    }
    return 0;
}
```

整体的思想跟流程图基本上是一样的。这里产生的16个子密钥Ki将会在后面的Feistel中用到。

IP 置换与 IP 逆置换

在 DES 的开始和结束部分，需要进行 IP 置换和 IP 逆置换两次操作。DES 提供了两张表分别是 IP 置换表和 IP 逆置换表。

```
int IP_Table[64] = {
    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,
    56,48,40,32,24,16, 8,0,
    58,50,42,34,26,18,10,2,
    60,52,44,36,28,20,12,4,
    62,54,46,38,30,22,14,6
};
int IP_1_Table[64] = {
    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,
    32,0,40, 8,48,16,56,24
};
```

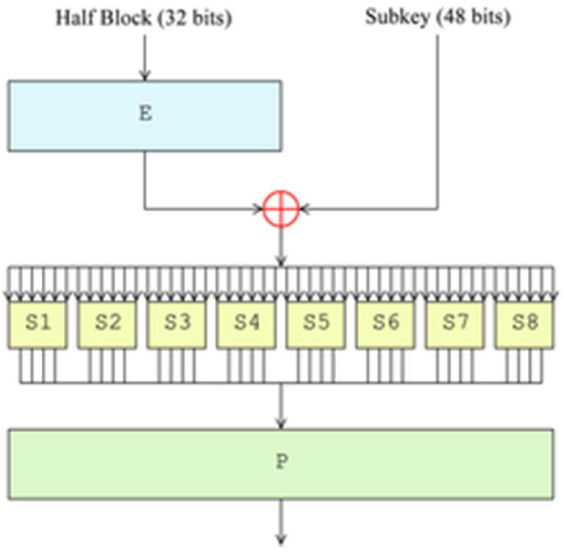
这两张表只是单纯的进行替换。据说这两次替换几乎没有密码学的重要性，只是为了简化输入输出数据库的过程而被显示的包括在标准中。（http://zh.wikipedia.org/wiki/DES）。
这两个置换从代码上实现也比较简单。

```
int IP_Transform(char data[64])
{
    int cnt;
    char temp[64];
    memset(temp, 0, 64);
    for (cnt = 0; cnt < 64; ++cnt)
    {
        temp[cnt] = data[IP_Table[cnt]];
    }
    memcpy(data,temp,64);
    return 0;
}

int IP_1_Transform(char data[64])
{
    int cnt;
    char temp[64];
    for (cnt = 0; cnt < 64; ++cnt)
    {
        temp[cnt] = data[IP_1_Table[cnt]];
    }
    memcpy(data, temp, 64);
    return 0;
}
```

Feistel 函数

这个函数可谓是 DES 中的核心。



这个函数每次对半个数据块 32bit 进行处理。主要有四个操作：扩展置换（E 置换），逐位异或，S 盒替换，P 置换。

扩展置换后会得到和子密钥长度一样的数据，以供后续的逐位异或运算，最重要的是会产生雪崩效应（http://blog.sina.com.cn/s/blog_8c23e74c010110vx.html），输入的 1 位将影响两个替换，使得输出对输入的依赖性将传播的更快。使用代码也比较容易实现。

```
int E_Transform(char data[48])
{
    int cnt;
    char temp[48];
    for(cnt = 0; cnt < 48; cnt++)
    {
        temp[cnt] = data[E_Table[cnt]];
    }
    memcpy(data,temp,48);
    return 0;
}
```

其中会用到一个扩展置换表

```
//扩展置换表E
int E_Table[48] = {
    31, 0, 1, 2, 3, 4,
    3, 4, 5, 6, 7, 8,
    7, 8, 9,10,11,12,
    11,12,13,14,15,16,
    15,16,17,18,19,20,
    19,20,21,22,23,24,
    23,24,25,26,27,28,
    27,28,29,30,31,0
};
```

接下来进行的操作是异或操作，将刚刚得到的48bit输出和相应的子密钥进行按位异或。

```
int fnXOR(char R[48], char L[48] ,int count)
```

```
{
    int cnt;
    for(cnt = 0; cnt < count; cnt++)
    {
        R[cnt] ^= L[cnt];
    }
    return 0;
}
```

接下来进行的是S盒，这个是DES密码安全的核心，S盒替代是一个非线性的过程，大大增加了安全性，使得密码不容易被线性分析所破译。用到的S盒如下。

```
//S盒
int S[8][4][16] =
//S1
{{{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}},
//S2
{{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}},
//S3
{{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}},
//S4
{{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}},
//S5
{{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}},
//S6
{{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}},
//S7
```

```
{{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}},
//S8
{{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}}};
```

8 个 S 盒的每一个都使用以查找表方式提供的非线性的变换将它的 6 个输入位变成 4 个输出位。下面使用代码实现：

```
int SBOX(char data[48])
{
    int cnt;
    int line,row,output;
    int cur1,cur2;
    for(cnt = 0; cnt < 8; cnt++)
    {
        cur1 = cnt*6;
        cur2 = cnt<<2;

        //计算在S盒中的行与列
        line = (data[cur1]<<1) + data[cur1+5];
        row = (data[cur1+1]<<3) + (data[cur1+2]<<2)
            + (data[cur1+3]<<1) + data[cur1+4];
        output = S[cnt][line][row];

        //化为2进制
        data[cur2] = (output&0X08)>>3;
        data[cur2+1] = (output&0X04)>>2;
        data[cur2+2] = (output&0X02)>>1;
        data[cur2+3] = output&0x01;
    }
    return 0;
}
```

再继续下去就是 P 置换了。这是为了将每个 S 盒的 4 位输出在下一次的扩张后，使用 4 个不同的 S 盒进行处理。

```
int P_Table[32] = {
    15, 6,19,20,28,11,27,16,
    0,14,22,25, 4,17,30, 9,
    1, 7,23,13,31,26, 2, 8,
    18,12,29, 5,21,10, 3,24
};
```

```
int P_Transform(char data[32])
{
    int cnt;
    char temp[32];
    for(cnt = 0; cnt < 32; cnt++)
    {
        temp[cnt] = data[P_Table[cnt]];
    }
    memcpy(data,temp,32);
    return 0;
}
```

P 置换也是比较简单的。现在这些需要的函数都有了，可以来写加密字符串的函数了。

```
int EncryptBlock(char plainBlock[8], char subKeys[16][48], char cipherBlock[8])
{
    char plainBits[64];
    char copyRight[48];
    int cnt;

    //把字符串转换为二进制64bits
    Char8ToBit64(plainBlock,plainBits);
    //初始置换（IP置换）
    IPTransform(plainBits);

    //16轮迭代
    for(cnt = 0; cnt < 16; cnt++)
    {
        memcpy(copyRight,plainBits+32,32);

        //将右半部分进行扩展置换，32位扩展到48位
        E_Transform(copyRight);

        //将右半部分与子密钥进行异或操作
        fnXOR(copyRight,subKeys[cnt],48);

        //异或结果进入S盒，输出32位结果
        SBOX(copyRight);

        //P置换
        P_Transform(copyRight);

        //将明文左半部分与右半部分进行异或
        fnXOR(plainBits,copyRight,32);

        if(cnt != 15){
```

```
            //最终完成左右部的交换
            Swap(plainBits,plainBits+32);
        }
    }
    //逆初始置换（IP^1置换）
    IP_1_Transform(plainBits);
    Bit64ToChar8(plainBits,cipherBlock);
    return 0;
}
```

其中的 SWAP 函数是交换左右部分，用代码实现如下。

```
int Swap(char left[32], char right[32])
{
    char temp[32];
    memcpy(temp,left,32);
    memcpy(left,right,32);
    memcpy(right,temp,32);
    return 0;
}
```

有了加密的方法，很容易就写出来解密的部分。

```
int DecryptBlock(char cipherBlock[8], char subKeys[16][48],char plainBlock[8])
{
    char cipherBits[64];
    char copyRight[48];
    int cnt;

    Char8ToBit64(cipherBlock,cipherBits);
    //初始置换（IP置换）
    IPTransform(cipherBits);

    //16轮迭代
    for(cnt = 15; cnt >= 0; cnt--)
    {
        memcpy(copyRight,cipherBits+32,32);
        //将右半部分进行扩展置换，从32位扩展到48位
        E_Transform(copyRight);
        //将右半部分与子密钥进行异或操作
        fnXOR(copyRight,subKeys[cnt],48);
        //异或结果进入S盒，输出32位结果
        SBOX(copyRight);
        //P置换
        P_Transform(copyRight);
        //将明文左半部分与右半部分进行异或
        fnXOR(cipherBits,copyRight,32);
```

```
        if(cnt != 0){
            //最终完成左右部的交换
            Swap(cipherBits,cipherBits+32);
        }
    }
    //逆初始置换（IP^1置换）
    IP_1_Transform(cipherBits);
    Bit64ToChar8(cipherBits,plainBlock);
    return 0;
}
```

这样整个 DES 的算法就算是基本完成了。