MD5

MD5 算法主要步骤有三步：

第一步：将要处理的消息进行分组，每组 64 个字节，也就是 64 * 8 = 512 位。

第二步：对每一组消息进行函数计算，得到一个 128 位 16 进制表示的 32 位字符串。

第三步：输出字符串。

算法：

第一步：数据处理：

计算出将要处理的消息的初始长度 length，以 64 个字节为一组。分成 n 组。

这时判断是否已经将数据读取完，如果读取完，就将每一组数据都进行函数处理，如果没有读取完，需要填充数据。

第二步：数据填充

对 length 进行填充，使 length 模 64 的值为 56，比如读取 80 个字节，80 % 64 = 16 ，16 小于 56，就需要填充 56 – 16 = 40 个字节，填充的具体数据是：第一位为 1 ，后面的全是 0。

第三步：函数处理

函数处理一共分为 4 轮， 每一轮 16 步， 一共 64 步处理。

在进行函数处理之前要先进行常量和函数的定义。

首先是 4 个连接常量的定义：

```
//初始化链接常量
unsigned int A = 0x67452301,
             B = 0xefcdab89,
             C = 0x98badcfe,
             D = 0x10325476,
```

这里的 A B C D 是全局变量，把 4 个链接常量分别赋给 A B C D 。

然后定义四个非线性函数：

```
//四个非线性函数
#define F(x, y, z)    (((x) & (y)) | ((~(x)) & (z)))
#define G(x, y, z)    (((x) & (z)) | ((y) & (~(z))))
#define H(x, y, z)    ((x) ^ (y) ^ (z))
#define I(x, y, z)    ((y) ^ ((x) | (~(z))))
```

第一轮用 F 函数，第二轮用 G 函数，第三轮用 H 函数，第四轮用 I 函数。

定义移位函数：

```
//移位函数：x左循环移n位
#define LeftMove(x, n)  (((x) << (n)) | ((x) >> (32 - (n))))
```

在每一轮中的 16 步，需要用到的函数为：

```
//四大轮的每一小轮中的16步所需的函数

#define FF(a, b, c, d, arrTemp, s, nConstTable) a = b + (LeftMove((a + F(b,c,d) + arrTemp + nConstTable),s))
#define GG(a, b, c, d, arrTemp, s, nConstTable) a = b + (LeftMove((a + G(b,c,d) + arrTemp + nConstTable),s))
#define HH(a, b, c, d, arrTemp, s, nConstTable) a = b + (LeftMove((a + H(b,c,d) + arrTemp + nConstTable),s))
#define II(a, b, c, d, arrTemp, s, nConstTable) a = b + (LeftMove((a + I(b,c,d) + arrTemp + nConstTable),s))
```

在这四个函数中，a, b, c, d 是 unsigned int 类型的临时变量，arrTemp 是一个保存了要处理的数据的数组，s 代表位移函数中移动的位数，nConstTable 是一个常量表。

每次操作，对 a , b, c, d 中的三个数做一次非线性函数运算，得到的数值加上第四个数，再加上某一个要处理的数据 arrTemp[i], 最后加上一个常量。再将所得结果移动 s 位， 再将这个值加上 a , b, c, d 中的某一个数，最后把这个值赋给 a , b, c, d 中的某一个数。

左移位数 S 的初始化：

```
//初始化每步左循环移位的位数
int     g_MoveNumber[64] ={
            7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
            5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
            4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
            6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
            };
```

一共 64 个数据，代表了 64 步里每一步需要左移的位数。

nConstTable 的初始化：

在进行一共 64 步的函数处理时，每一次处理都有一个不同的常量。

这个常量的数值是 2^{32} * |sin (16 * Count * Step + 1)| 的整数部分。

其中 Count 表示是第几轮，Step 表示这一轮中的第几步。

```
//初始化常量表。
void initConstTable()
{
    int nCount = 0,
        nStep = 0;
    for (nCount = 0; nCount < 4; nCount++)
    {
        for (nStep = 0; nStep < 16; nStep++)
        {
            g_nConstTable[nCount * 16 + nStep] =
            (unsigned int)(pow(2, 32) * fabs(sin((double)(16 * nCount + nStep + 1))));
        }
    }
```

数据都准备好后，就可以进行四轮函数运算了。

先把链接变量 A, B, C, D 的值 赋给 临时变量。

```
nTemp_a = A;
nTemp_b = B;
nTemp_c = C;
nTemp_d = D;
```

进行第一轮运算：

```
//第一轮
FF (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 0], g_MoveNumber[0], g_nConstTable[0]); // 1
FF (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 1], g_MoveNumber[1], g_nConstTable[1]); // 2
FF (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 2], g_MoveNumber[2], g_nConstTable[2]); // 3
FF (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 3], g_MoveNumber[3], g_nConstTable[3]); // 4

FF (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 4], g_MoveNumber[4], g_nConstTable[4]); // 5
FF (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 5], g_MoveNumber[5], g_nConstTable[5]); // 6
FF (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 6], g_MoveNumber[6], g_nConstTable[6]); // 7
FF (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 7], g_MoveNumber[7], g_nConstTable[7]); // 8

FF (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 8], g_MoveNumber[8], g_nConstTable[8]); // 9
FF (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 9], g_MoveNumber[9], g_nConstTable[9]); // 10
FF (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[10], g_MoveNumber[10], g_nConstTable[10]); // 11
FF (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[11], g_MoveNumber[11], g_nConstTable[11]); // 12

FF (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[12], g_MoveNumber[12], g_nConstTable[12]); // 13
FF (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[13], g_MoveNumber[13], g_nConstTable[13]); // 14
FF (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[14], g_MoveNumber[14], g_nConstTable[14]); // 15
FF (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[15], g_MoveNumber[15], g_nConstTable[15]); // 16
```

进行第二轮运算：

```
//第二轮
GG (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 1], g_MoveNumber[16], g_nConstTable[16]); //17
GG (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 6], g_MoveNumber[17], g_nConstTable[17]); //18
GG (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[11], g_MoveNumber[18], g_nConstTable[18]); //19
GG (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 0], g_MoveNumber[19], g_nConstTable[19]); //20

GG (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 5], g_MoveNumber[20], g_nConstTable[20]); //21
GG (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[10], g_MoveNumber[21], g_nConstTable[21]); //22
GG (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[15], g_MoveNumber[22], g_nConstTable[22]); //23
GG (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 4], g_MoveNumber[23], g_nConstTable[23]); //24

GG (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 9], g_MoveNumber[24], g_nConstTable[24]); //25
GG (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[14], g_MoveNumber[25], g_nConstTable[25]); //26
GG (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 3], g_MoveNumber[26], g_nConstTable[26]); //27
GG (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 8], g_MoveNumber[27], g_nConstTable[27]); //28

GG (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[13], g_MoveNumber[28], g_nConstTable[28]); //29
GG (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 2], g_MoveNumber[29], g_nConstTable[29]); //30
GG (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 7], g_MoveNumber[30], g_nConstTable[30]); //31
GG (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[12], g_MoveNumber[31], g_nConstTable[31]); //32
```

进行第三轮运算：

```
//第三轮
HH (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 5], g_MoveNumber[32], g_nConstTable[32]); //33
HH (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 8], g_MoveNumber[33], g_nConstTable[33]); //34
HH (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[11], g_MoveNumber[34], g_nConstTable[34]); //35
HH (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[14], g_MoveNumber[35], g_nConstTable[35]); //36

HH (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 1], g_MoveNumber[36], g_nConstTable[36]); //37
HH (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 4], g_MoveNumber[37], g_nConstTable[37]); //38
HH (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 7], g_MoveNumber[38], g_nConstTable[38]); //39
HH (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[10], g_MoveNumber[39], g_nConstTable[39]); //40

HH (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[13], g_MoveNumber[40], g_nConstTable[40]); //41
HH (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 0], g_MoveNumber[41], g_nConstTable[41]); //42
HH (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 3], g_MoveNumber[42], g_nConstTable[42]); //43
HH (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 6], g_MoveNumber[43], g_nConstTable[43]); //44

HH (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 9], g_MoveNumber[44], g_nConstTable[44]); //45
HH (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[12], g_MoveNumber[45], g_nConstTable[45]); //46
HH (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[15], g_MoveNumber[46], g_nConstTable[46]); //47
HH (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 2], g_MoveNumber[47], g_nConstTable[47]); //48
```

进行第四轮运算：

```
//第四轮
II (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 0], g_MoveNumber[48], g_nConstTable[48]); //49
II (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 7], g_MoveNumber[49], g_nConstTable[49]); //50
II (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[14], g_MoveNumber[50], g_nConstTable[50]); //51
II (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 5], g_MoveNumber[51], g_nConstTable[51]); //52

II (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[12], g_MoveNumber[52], g_nConstTable[52]); //53
II (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[ 3], g_MoveNumber[53], g_nConstTable[53]); //54
II (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[10], g_MoveNumber[54], g_nConstTable[54]); //55
II (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 1], g_MoveNumber[55], g_nConstTable[55]); //56

II (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 8], g_MoveNumber[56], g_nConstTable[56]); //57
II (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[15], g_MoveNumber[57], g_nConstTable[57]); //58
II (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 6], g_MoveNumber[58], g_nConstTable[58]); //59
II (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[13], g_MoveNumber[59], g_nConstTable[59]); //60

II (nTemp_a, nTemp_b, nTemp_c, nTemp_d, arrTemp[ 4], g_MoveNumber[60], g_nConstTable[60]); //61
II (nTemp_d, nTemp_a, nTemp_b, nTemp_c, arrTemp[11], g_MoveNumber[61], g_nConstTable[61]); //62
II (nTemp_c, nTemp_d, nTemp_a, nTemp_b, arrTemp[ 2], g_MoveNumber[62], g_nConstTable[62]); //63
II (nTemp_b, nTemp_c, nTemp_d, nTemp_a, arrTemp[ 9], g_MoveNumber[63], g_nConstTable[63]); //64
```

四轮运算进行完毕后，将 A B C D 分别加上临时变量 nTemp_a，nTemp_b，nTemp_c，nTemp_d。

```
A += nTemp_a;
B += nTemp_b;
C += nTemp_c;
D += nTemp_d;
```

到这里，一组函数运算就完成了。

最开始将要处理的数据分成了 n 组，对每一组都进行一次函数处理。最后得到了 4 个 32 位的数据 A B C D。

数据输出：

由于内存中保存的数据的形式是 数据的低位保存在内存的高位，数据的高位保存在内存的低位，所以要将高低位互换后输出。

```
//小端存储数据的提取，高地位互换
#define HighLowExchange(x) (x << 24) | ((x << 8) & 0xff0000) | ((x >> 8) & 0xff00) | (x >> 24)
```

定义一个函数专门用来高低位互换。

顺序输出 A B C D，就是最后的 MD5 值：

```
printf("MD5: %08x%08x%08x%08x\n",HighLowExchange(A),HighLowExchange(B),
    HighLowExchange(C),HighLowExchange(D));
```

SHA-1

SHA-1 算法主要步骤有三步：

第一步：将要处理的消息进行分组，每组 64 个字节，也就是 64 * 8 = 512 位。如果长度不够要进行补位，之后还要在最后补上原数据长度。

第二步：对每一组消息进行函数计算，得到一个 5 个 32 位，也就是 160 位的字符串。

第三步：输出字符串。

算法：

第一步：数据处理

和 MD5 类似，以 64 位为一组分成 n 组，如果有剩余数据，要进行填充，和 MD5 有区别的地方在于最后还要填补上原数据的长度。

例如说现在要处理 “abc”，a = 97(01100001_2)，b = 98(01100010_2)，c = 99(01100011_2)

原始信息为： 01100001 01100010 01100011

首先补一个 1：01100001 01100010 01100011 1

再补 n 个 0，直到 mod 64 = 56：

01100001 01100010 01100011 100…..0000

写成 16 进制：

61626380 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000

计算原字符串长度为 3 个字节，也就是 24 位，转换成 16 进制为 18，补到最后：

61626380 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000018

第二步：函数计算

计算之前先定义 5 个链接常量，并赋值给变量 H0，H1，H2，H3，H4：

```
//5个链接常量
unsigned long H0 = 0x67452301,
              H1 = 0xefcdab89,
              H2 = 0x98badcfe,
              H3 = 0x10325476,
              H4 = 0xc3d2e1f0;
```

定义函数处理中要用到的循环移位运算：

```
//循环移位
#define LoopMove(x, n)  (((x) << (n)) | ((x) >> (32 - (n))))
```

定义函数处理中用到的 4 个函数：

```
#define E0(x, y, z)    (((x) & (y)) | ((~(x)) & (z)))
#define E1(x, y, z)    ((x) ^ (y) ^ (z))
#define E2(x, y, z)    ((x) & (y)) | ((x) & (z)) | ((y) & (z))
#define E3(x, y, z)    E1(x, y, z)
```

还需要一个 32 位的缓冲区数组 W[80]，对每一组数据进行临时存储。

首先初始化 W 数组，遍历 W 数组，i 从 0 到 79。将每组要处理的数据从左至右分割为 16 个 32 位的字符串，存储在 W[0] – W[15] 中。

对于 W[16] – W [79] ， W[i] = LoopMove((W[i - 3] ^ W[i - 8] ^ W[i - 14] ^ W[i - 16]) , 1)。 表示将 W[i - 3] ^ W[i - 8] ^ W[i - 14] ^ W[i - 16] 的值左移一位。到这里完成了对缓冲数组 W 的赋值。

准备三个临时变量：tempK：用来保存一个常量，

tempFunction：用来保存 E 函数的计算结果，

temp ：交换数值用到的临时变量

把 H0，H1，H2，H3，H4 的值赋给变量 A，B，C，D，E

```
A = H0;
B = H1;
C = H2;
D = H3;
E = H4;
```

然后开始 80 轮函数计算 ，i 从 0 到 80：

在前 0 - 19 轮中：tempK = 0x5A827999，tempFunction = E0 (B，C，D)

在 20 – 39 轮中：tempK = 0x6ED9EBA1，tempFunction = E1 (B，C，D)

在 40 – 59 轮中：tempK = 0x8F1BBCDC，tempFunction = E2 (B，C，D)

在 60 – 79 轮中：tempK = 0xCA62C1D6，tempFunction = E3 (B，C，D)

得到 tempK 和 tempFunction 的值后继续计算：

temp = LoopMove(A，5) + tempFunction + E + W[i] + tempK

E = D

D = C

C = LoopMove(B，30)

B = A

A = temp

到此完成了 80 轮的计算，把得到的 A，B，C，D，E 和 H0，H1，H2，H3，H4 相加再赋给 0，H1，H2，H3，H4。

```
H0 = H0 + A;
H1 = H1 + B;
H2 = H2 + C;
H3 = H3 + D;
H4 = H4 + E;
```

最后顺序输出 H0，H1，H2，H3，H4 就得到了 160 位的字符串。

RSA

首先介绍一下需要用到的数学知识

欧拉函数：

φ(n) 表示的是 在[1，n)中，与 n 互质的正整数的个数，

如果 n 是质数，φ(n) = n − 1，因为质数与小于它的每一个数，都构成互质关系，例如说 7，7 和 6,5,4,3,2,1 都是互质的，所以 φ(7) = 7 − 1 = 6。

如果 n 是两个互质正整数的积，那么φ(p * q) = φ(p) * φ(q)。证明需要用到中国剩余定理。

欧拉定理：

欧拉函数主要用在欧拉定理上。如果两个正整数 a 和 n 互质，则

$$a^{\varphi(n)} \equiv 1(\mathrm{mod}\ n)$$

也就是说 a 的φ(n)次方被 n 除的余数是 1。

欧拉函数中有一个特殊情况叫做费马小定理：

假设正整数 a 与质数 p 互质，因为φ(p) = p − 1,写成欧拉定理的形式为：

$$a^{p-1} \equiv 1(\mathrm{mod}\ p)$$

模反元素：

如果两个正整数 a 和 n 互质，那么一定有正整数 b， 使得 （a * b） % n = 1，例如：3 和 5 互质，那么 3 的模反元素就是 7，因为 （3 * 7）% 5 = 1。

现在假设甲，乙之间要通过 RSA 加密数据通信。

乙方需要生成两把密钥，一把公钥，公钥是公开的，任何人都可以获取；一把私钥，私钥则是乙方用来解密的，需要保密。

生成密钥的过程：

1.随机选定两个大质数 p，q

2.计算 n = p * q ，RSA 算法难以被破解的原因就是在选取了足够大的 p 和 q 的情况下难以因式分解 n。

3.计算 φ(n) = (p - 1)(q - 1)

4.选取正整数 e ，满足 1 < e < φ(n) 且 e 与 φ(n) 互质

5.计算 e 的模反元素 d ， d * e ≡ 1 (mod φ(n))，计算模反元素的算法为：

```
//计算 d, d * e ==1(mod Φ(n))
nD = 1;
while ((nD * nE) % nEuler != 1)
{
    nD++;
}
```

得到公钥 （e，n）私钥 （d，n）

加密过程：

假设甲向乙发送加密信息 m，他需要使用乙的公钥(e，n)来对 m 进行加密，m 必须是正整数。根据公式：

$$m^e \equiv c \pmod n$$

解出的 c 就是加密后的信息。

解密过程：

乙收到来自甲的密文数据 c ，利用乙自己的私钥 (d，n)，再根据公式：

$$c^d \equiv m \pmod n$$

可以解出明文数据 m，这里的计算需要用到幂取余的算法

```
//幂取余
unsigned int clalculateRemaider(unsigned int nSourceMessage,
                                unsigned int nKey, unsigned int nBaseNumber)
{
    unsigned int nDecodeMessage = 1;

    while (nKey--)
    {
        nDecodeMessage = nDecodeMessage * nSourceMessage  % nBaseNumber;
    }
    return nDecodeMessage;
}
```

Base64

Base64 是将所有二进制能表示的数据转换成 64 个可打印的字符的一种编码。

这 64 个字符的映射表为：

编号	字符	编号	字符	编号	字符	编号	字符
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

因为$2^6 = 64$ ，所以在 6 个 bit 为一组的情况下能够用这 64 个字符来表示所有二进制数据。

编码过程：

1. 每次取 3 个字节（24bit）的数据放在内存中，如果最后一次只能取到一个字节，最后得到的编码需要在最后加上 “==” ，如果只能取到两个字节，最后得到的编码要加上 “=” 。

2. 每次取 6 个 bit，不足 6bit 时在后面补 0 至 6 个 bit。其值就是映射表中的编号，根据编号得到对应的字符。

例如现在要对“Base”进行编码：

	B	a	s	e
Ascii	66	97	115	101
二进制	01000010	01100001	01110011	01100101

以三个字节也就是三个字符为一组首先取得“Bas”

	B	a	s
ASCII	66	97	115
二进制	01000010	01100001	01110011

将其二进制以 6 个 bit 为一组重新组合：

010000 ，100110，000101，110011

编码后的第一个字符值为原第一个字符的前 6 位：010000 ->16 ->Q

编码后的第二个字符值为原第一个字符的后 2 位和原第二个字符的前 4 位：100110 ->38 ->m

编码后的第三个字符值为原第二个字符的后 4 位和原第三个字符的前 2 位：000101 -> 5 -> F

编码后的第四个字符值为原第三个字符的后 6 位：110011->51 ->z

接着取下一组三字节，但是因为只剩一个字符 e 所以在对 e 编码完成后要在最后加上“==”。

	e
Ascii	101
二进制	01100101

将其二进制以 6 个 bit 为一组重新组合：

011001，01

编码后第一个字符： 011001 –> 25 ->Z

最后只剩下 01 不够 6bit，在后面补 0，所以编码后第二个字符：010000 –>16 ->Q

最后对 Base 的编码结果就是：QmFzZQ== 。

RC4

RC4 的加密过程简单来说就是通过算法将短密钥变成一串与明文相同长度的密钥流，与明文异或产生密文，与密文异或产生明文。

产生密钥流的步骤分为四步：

1. 初始化状态矢量 S：

S[256]是一个长度 256 的字符数组，i 从 0 到 255。线性的向 S[256]中赋值：S[i] = i 。

2. 初始化临时矢量 T：

T[256]也是一个长度为 256 的字符数组，用来保存密钥的长度 keyLength，如果 keyLength 的长度不够 256 字节，那么就轮转的把密钥赋给 T。假设密钥为 key，i 从 0 到 255，T[i] = key[i mod keyLength]。

3. 重新排列 S 数组，产生 S-Box：

需要一个临时变量 j，i 从 0 到 255 ，执行操作：首先计算 j，j = (j + S[i] + T[i]) % 256 ，然后交换 S[i] 和 S[j] 的值。以这样的操作遍历一遍 S 数组后，就产生了 S-Box。

4. 产生密钥流

密钥流的长度是根据明文的长度来决定的，首先需要三个临时变量 i，j，t 假设明文长度为：stringLength ，count 从 0 到 stringLength，对应位的密钥流为 keyStream[count]，执行以下操作：

1. i = (i + 1) % 256
2. j = (j + S[i]) % 256
3. 交换 S[i] 和 S[j] 的值
4. t = (S[i] + S[j]) % 256
5. keyStream [count] = S[t]

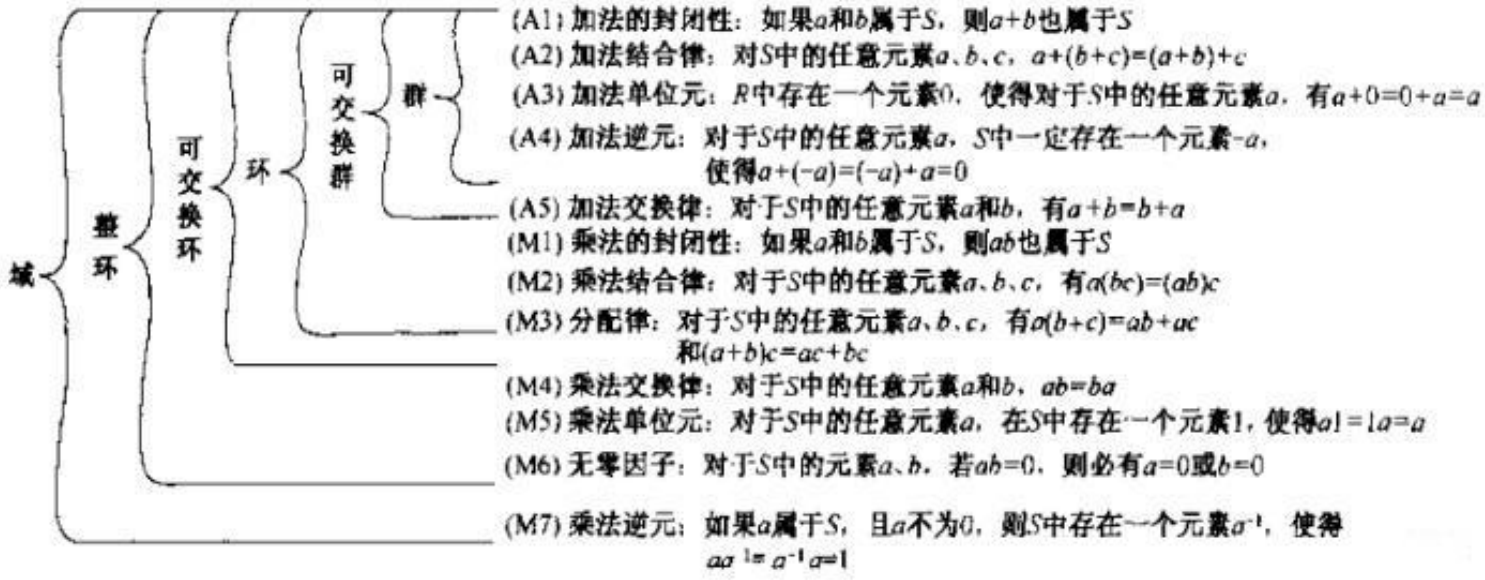
计算密钥流和明文的异或就产生密文。

计算密钥流和密文的异或就产生明文。

AES

需要用到的数学概念：

1. 域的概念：http://baike.baidu.com/link?url=lciY-h6aiKM1Gxwh9iZQo3ARIs2VBudnHxNLq6-pNIC4E_MGBEXH8yD_zQstSiPlNVMU8EAzfkCwV3TmpHLZAr7478GZ8ZKFpLEBxpglj37



2. 有限域：有限域的元素个数必须是一个质数的幂 p^n，元素个数为p^n的有限域记为 GF(p^n)。

3. 乘法逆元：对于有限域 GF(p^n),任意的存在 w∈GF(p^n),且 w ≠ 0,z ∈GF(p^n)，使得 w * z ≡ 1 mod p，则 z 为 w 在该有限域上的乘法逆元。

在 AES 算法里，在有限域 GF(2^8)上运算，为了构造 GF(2^8)，必须定义一个既约多项式 m(x) = $x^8 + x^4 + x^3 + x + 1$,下图为 AES 的乘法逆元求法：

乘法

简单的异或运算不能完成 GF(2^n)上的乘法。但是可以使用一种合理且容易实现的技巧。我们将在高级加密标准使用的有限域中讨论该技巧。这个有限域是 GF(2^8),其中模多项式为 $m(x) = x^8 + x^4 + x^3 + x + 1$。

这个技巧基于下面的等式:

$$x^8 \bmod m(x) = [m(x) - x^8] = (x^4 + x^3 + x + 1) \tag{4.7}$$

通过观察不难证明等式(4.7)是正确的。一般地,在 GF(2^n)上对于 n 次多项式 $p(x)$,有 $x^n \bmod p(x) = [p(x) - x^n]$。

现在考虑 GF(2^8)上的多项式 $f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$,将它乘以 x,可得:

$$x \times f(x) = (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_7x^8 + b_1x^2 + b_0x) \bmod m(x) \tag{4.8}$$

如果 $b_7 = 0$,那么结果就是一个次数小于 8 的多项式,不需要进一步计算。如果 $b_7 = 1$,那么可以通过等式(4.7)进行除 $m(x)$取余运算:

$$x \times f(x) = (b_6x^7 + b_7x^8 + b_4x^5 + b_3x^4 + b_7x^8 + b_1x^2 + b_0x) + (x^4 + x^3 + x + 1)$$

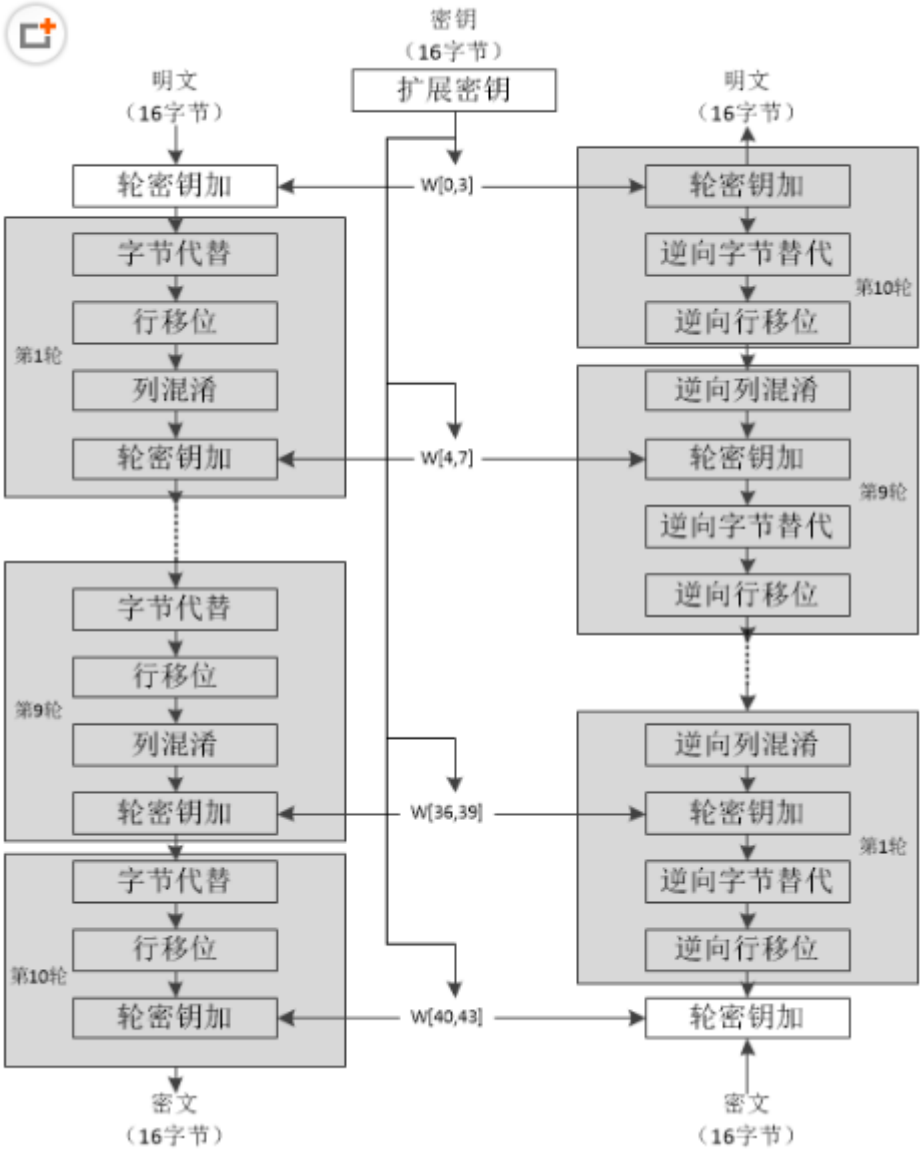
这表明乘以 x(如 00000010)的运算可以通过左移一位后按位异或 00011011 来实现,其表示为 $(x^4 + x^3 + x + 1)$。总结如下:

$$x \times f(x) = \begin{cases} (b_6b_5b_4b_3b_2b_1b_00) & \text{若 } b_7 = 0 \\ (b_6b_5b_4b_3b_2b_1b_00) \oplus (00011011) & \text{若 } b_7 = 1 \end{cases} \tag{4.9}$$

乘以一个高于一次的多项式可以通过重复使用等式(4.9)来实现。这样一来,GF(2^8)上的乘法可以用多个中间结果相加的方法实现。

划线处的推导过程可见:《密码编码学与网络安全——原理与实践》(第五版)p 96

AES 加密过程图：

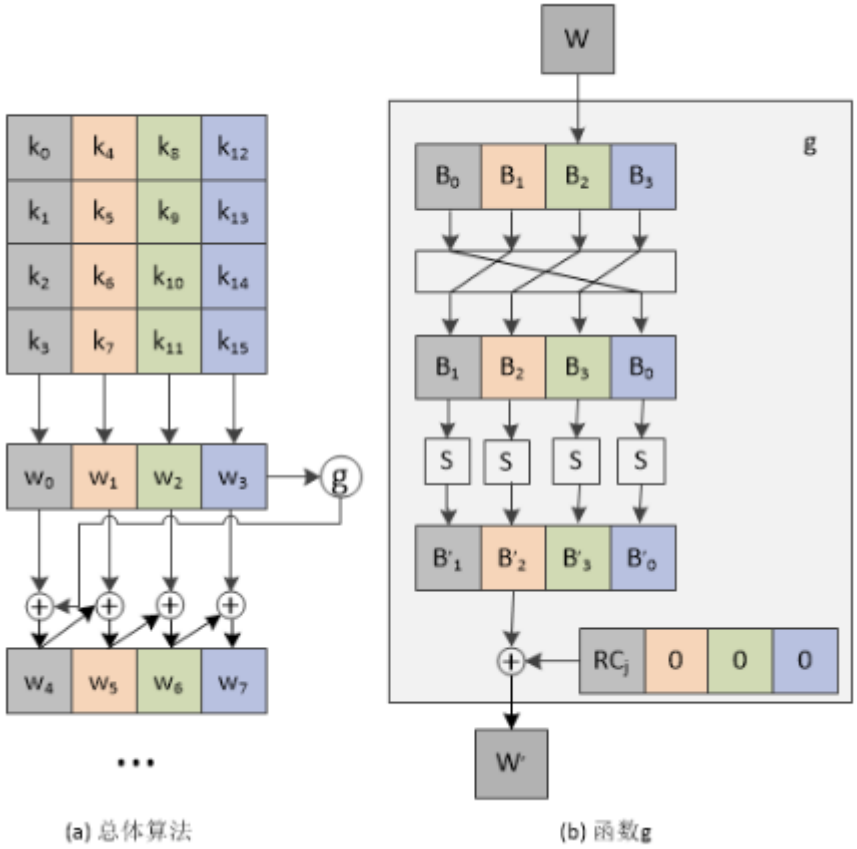


AES 加密用到 5 种主要操作，分别是：密钥拓展，字节代替，行移位，列混淆，轮密钥加。对应的解密 4 种操作分别是：逆字节代替，逆行移位，逆列混淆，轮密钥加。由于每一步都是可逆的所以只要按照加密的顺序反向操作就能恢复明文。

注意在 1 – 9 轮加密中用到四种操作，但是在第 10 轮中没有用到列混淆。对应的在解密时的第 10 轮也不用逆列混淆。

1. 密钥拓展

密钥拓展的原理图：



在拿到密钥之后，整个加密过程之前，需要对密钥进行处理，这个步骤叫密钥拓展。首先说明几个参数：

State[4][4]：状态矩阵中临时保存着要操作的数据，之后的操作大部分是针对状态矩阵进行的。也就是图中的 k 数组。

Nb：状态矩阵 State 所包含的列的个数，Nb = 4.

Nk ：密钥包含的 32bit 字的个数，Nk = 4 ， 6， 或 8

Nr：加密轮数

密钥拓展的操作，也就是图中的 g 函数：

1.）位置变换(rotWord)：对于 $[a_0, a_1, a_2, a_3]$ 循环左移为 $[a_1, a_2, a_3, a_0]$。

2.）S-Box 变换(subWord)，S-Box 的变换下面具体会说到。与这里是同样的操作。

3.）与轮常数数组 Rcon[] ={01,02,04,08,10,20,40,80,1B,36}进行异或。

4.）拓展密钥数组 w[]的前 Nk 个元素就是密钥 Key，以后的元素 w[i]等于它前一个元素 w[i - 1] 与第 Nk 个元素 w[i - Nk]的异或。但是，若 i 为 Nk 的倍数，那么 w[i] = w[i - Nk] ⊕ subWord(rotWord (w[i -1])) ⊕ Rcon[i / Nk - 1]

2. 字节代替：

字节代替的操作是保证 AES 算法安全的关键，需要用到一个非线性映射表 S-Box：

```
 //S盒
unsigned char S_Box[16][16] =
{
  // 0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, /*0*/
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, /*1*/
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, /*2*/
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75, /*3*/
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, /*4*/
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF, /*5*/
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8, /*6*/
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, /*7*/
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73, /*8*/
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB, /*9*/
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, /*A*/
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, /*B*/
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A, /*C*/
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, /*D*/
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF, /*E*/
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16  /*F*/
};
```

S-box 的构造方法为：

1）. 首先从 0 开始按照行号列号升序填入数字，例如：x 行 y 列就是：0xXY

2）. 求出每一个元素在 GF(2^8)上的逆，0 行 0 列元素映射为 0x63

3）. 仿射变换，对上一步中的每一个字节的每一位作以下变换：

$$y_i = x_{i+4} \bmod 8 + x_{i+5} \bmod 8 + x_{i+6} \bmod 8 + x_{i+7} \bmod 8 + c_i$$

c_i是字节 0x63 的第 i 位。用矩阵表示为：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

逆 S 盒：

```
//逆s盒
 unsigned char invS_Box[16][16] =
 {
//   0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB, /*0*/
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB, /*1*/
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E, /*2*/
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25, /*3*/
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92, /*4*/
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84, /*5*/
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06, /*6*/
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B, /*7*/
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73, /*8*/
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E, /*9*/
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B, /*A*/
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4, /*B*/
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F, /*C*/
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF, /*D*/
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61, /*E*/
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D  /*F*/
};
```

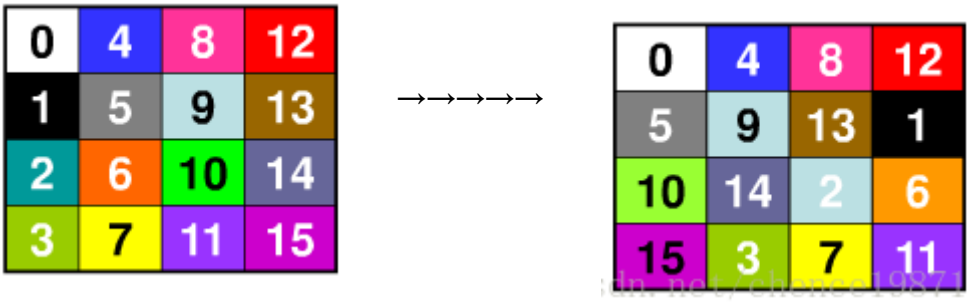
S-Box 替换具体操作为：对状态矩阵 Status[4][4]中的每一个值对照 S-Box 做映射，把要做映射的值的高 4 位作为行数，低 4 位作为烈说。

例如现在要对 0x55 进行 S-Box 替换，只需要找到 S_Box[5][5] = 0xFC 就完成了替换。

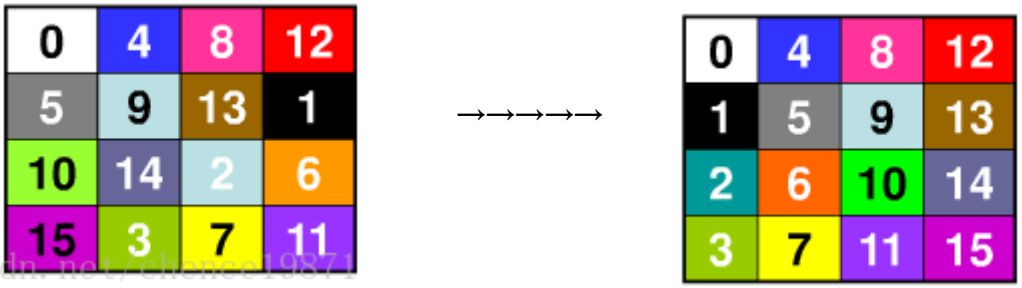
对应的逆 S 盒还原就是 查找 invS_Box[F][C] = 0x55。

3. 行移位

在进行行移位之前会接收状态矩阵 Status[4][4]，具体的操作就是对矩阵进行循环移位，实现为：第一行不变，第二行循环左移一字节，第三行循环左移两字节，第四行循环左移四字节。



对应的，逆行移位就是将行移位后的矩阵还原：



4. 列混淆

列混淆同样也是对状态矩阵 Status[4][4]的操作。用状态矩阵和常数矩阵进行 GF(2^8)上的乘法。需要用到的常数矩阵：

```
//列混淆的常数矩阵
unsigned char columMixTab[4][4]={
        0x02,0x03,0x01,0x01,
        0x01,0x02,0x03,0x01,
        0x01,0x01,0x02,0x03,
        0x03,0x01,0x01,0x02
};
```

两个矩阵相乘：

$$\begin{bmatrix}02 & 03 & 01 & 01\\ 01 & 02 & 03 & 01\\ 01 & 01 & 02 & 03\\ 03 & 01 & 01 & 02\end{bmatrix}\begin{bmatrix}s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3}\\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3}\\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3}\\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3}\end{bmatrix} = \begin{bmatrix}s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3}\\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3}\\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3}\\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3}\end{bmatrix}$$

$$s'_{0,j} = (02 \cdot s_{0,j}) \oplus (03 \cdot s_{1,j}) \oplus (s_{2,j}) \oplus (s_{3,j})$$

$$s'_{1,j} = (s_{0,j}) \oplus (02 \cdot s_{1,j}) \oplus (03 \cdot s_{2,j}) \oplus (s_{3,j})$$

$$s'_{2,j} = (s_{0,j}) \oplus (s_{1,j}) \oplus (02 \cdot s_{2,j}) \oplus (03 \cdot s_{3,j})$$

$$s'_{3,j} = (03 \cdot s_{0,j}) \oplus (s_{1,j}) \oplus s_{2,j} \oplus (02 \cdot s_{3,j})$$

“⊕”表示异或，" · "表示在 GF(2^8)上的乘法。

举个例子：

$$\begin{bmatrix}02 & 03 & 01 & 01\\ 01 & 02 & 03 & 01\\ 01 & 01 & 02 & 03\\ 03 & 01 & 01 & 02\end{bmatrix}\begin{bmatrix}C9\\ 6E\\ 46\\ A6\end{bmatrix} = \begin{bmatrix}DB\\ *\\ *\\ *\end{bmatrix}$$

计算：$s'_{0,0} = (02 \cdot C9) \oplus (03 \cdot 6E) \oplus (01 \cdot 46) \oplus (01 \cdot A6)$

其中：

$02 \cdot C9 = 02 \cdot 11001001 = 10010010 \oplus 000011011 = 10001001$

$02 \cdot C9$就是将 C9 的二进制左移移位，但是 C9 的最高位为 1，则还需要将移位后的结果和 0x1B（000011011）进行异或。

$03 \cdot 6E = (01 \oplus 02) \cdot 6E = 01101110 \oplus 11011100 = 10110010$

$01 \cdot 46 = 01000110$

$01 \cdot A6 = 10100110$

$$s'_{0,0} = 10001001 \oplus 10110010 \oplus 01000110 \oplus 10100110 = 11011011 = DB$$

对应的，逆列混淆也有一个常数矩阵：

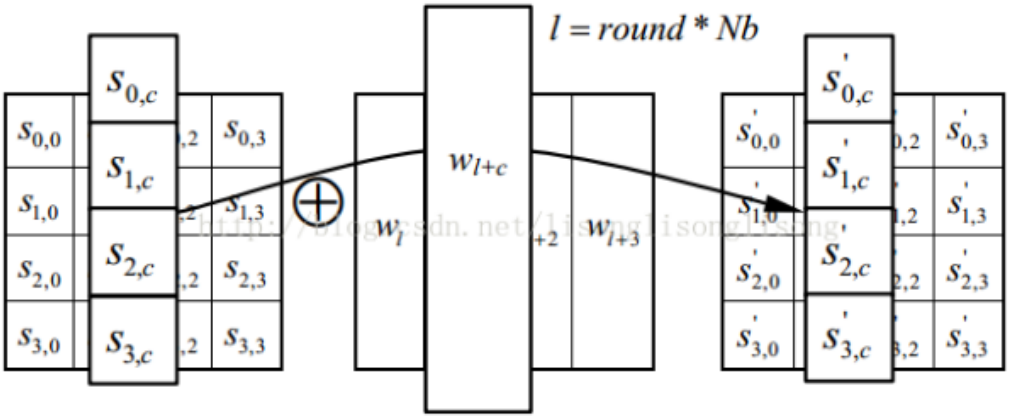
```
//逆列混淆的常数矩阵
unsigned char invColumMixTab[4][4]={
        0x0e,0x0b,0x0d,0x09,
        0x09,0x0e,0x0b,0x0d,
        0x0d,0x09,0x0e,0x0b,
        0x0b,0x0d,0x09,0x0e
};
```

同样也是尼列混淆常数矩阵和状态矩阵相乘：

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

5. 轮密钥加

拓展密钥参与这一步操作，根据加密的轮数，把状态数组 State 中的 4 个列和分别拓展密钥 w[]中的 4 个列进行按位异或，如图：



在解密时，也是用的同样的方法，能够还原是因为轮密钥加变换的逆就是它本身。

最后附上 AES 加密的动画演示：http://coolshell.cn/wp-content/uploads/2010/10/rijndael_ingles2004.swf