

密码学综合设计实验

实验 3：AES 加密解密算法实现

学号：031803108

姓名：苏煜程

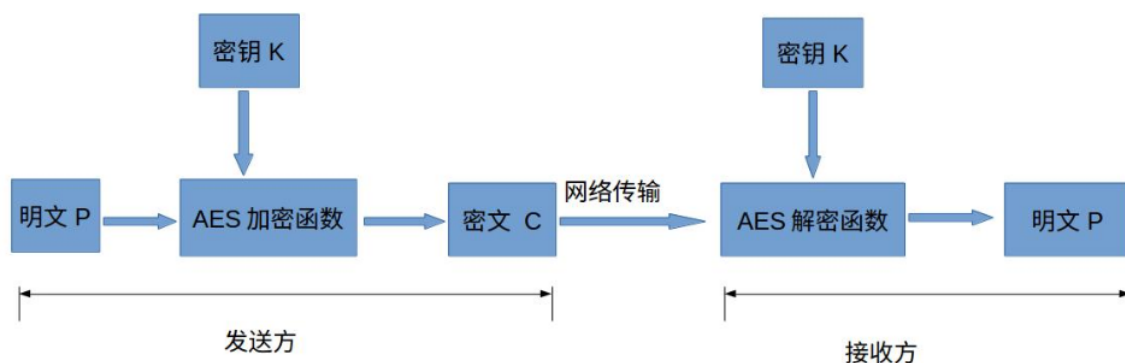
2019 年 10 月 10 日

实验要求

- 实现GF(28)加法与乘法运算并验算正确性
 - 算法运算采用多项式表示法
 - 加法运算实际上就是按位异或
 - 乘法运算实现要利用XTime运算来设计算法，取模11B
 - 要用课本的例子验算是否正确实现
- 实现系数在GF(28)上的多项式的乘法
 - 运算是4字节向量乘4字节向量（向量的分量表示多项式的系数）
 - 输出是4字节向量
 - 运算过程，字节对字节的加法就是异或，乘法需调用第1步实现的乘法
- AES算法实现
 - 输入：Nb, Nk, 明文分组，种子密钥，
输出：密文分组
明文分组以字符串形式输入，密文分组以16进制形式输出，密钥的输入形式自行设计
 - 实现字节代换ByteSub及逆代换InvByteSub
 - 实现行移位ShiftRow及逆行移位InvShiftRow
 - 实现列混淆MixColumn及逆列混淆InvMixcColumn，注意，（逆）列混淆实现需要调用第2步实现的多项式乘法
 - 用密钥编排算法获取轮密钥
 - 实现AES加密与解密，注意加密密钥与解密密钥次序与差别
- 附加内容
 - 用代码自动生成字节替换所用的S盒与逆S盒

实验原理

高级加密标准(AES, Advanced Encryption Standard)为最常见的对称加密算法(微信小程序加密传输就是用这个加密算法的)。对称加密算法也就是加密和解密用相同的密钥，具体的加密流程如下图：



http://blog.csdn.net/qq_28205153

下面简单介绍下各个部分的作用与意义：

- 明文P
没有经过加密的数据。
- 密钥K

用来加密明文的密码，在对称加密算法中，加密与解密的密钥是相同的。密钥为接收方与发送方协商产生，但不可以直接在网络上传输，否则会导致密钥泄漏，通常是通过非对称加密算法加密密钥，然后再通过网络传输给对方，或者直接面对面商量密钥。密钥是绝对不可以泄漏的，否则会被攻击者还原密文，窃取机密数据。

- AES加密函数

设AES加密函数为E，则 $C = E(K, P)$ ，其中P为明文，K为密钥，C为密文。也就是说，把明文P和密钥K作为加密函数的参数输入，则加密函数E会输出密文C。

- 密文C

经加密函数处理后的数据

- AES解密函数

设AES解密函数为D，则 $P = D(K, C)$ ，其中C为密文，K为密钥，P为明文。也就是说，把密文C和密钥K作为解密函数的参数输入，则解密函数会输出明文P。

在这里简单介绍下对称加密算法与非对称加密算法的区别。

- 对称加密算法

加密和解密用到的密钥是相同的，这种加密方式加密速度非常快，适合经常发送数据的场合。缺点是密钥的传输比较麻烦。

- 非对称加密算法

加密和解密用的密钥是不同的，这种加密方式是用数学上的难解问题构造的，通常加密解密的速度比较慢，适合偶尔发送数据的场合。优点是密钥传输方便。常见的非对称加密算法为RSA、ECC和ElGamal。

实际中，一般是通过RSA加密AES的密钥，传输到接收方，接收方解密得到AES密钥，然后发送方和接收方用AES密钥来通信。

本文下面AES原理的介绍参考自《现代密码学教程》，AES的实现在介绍完原理后开始。

AES的基本结构

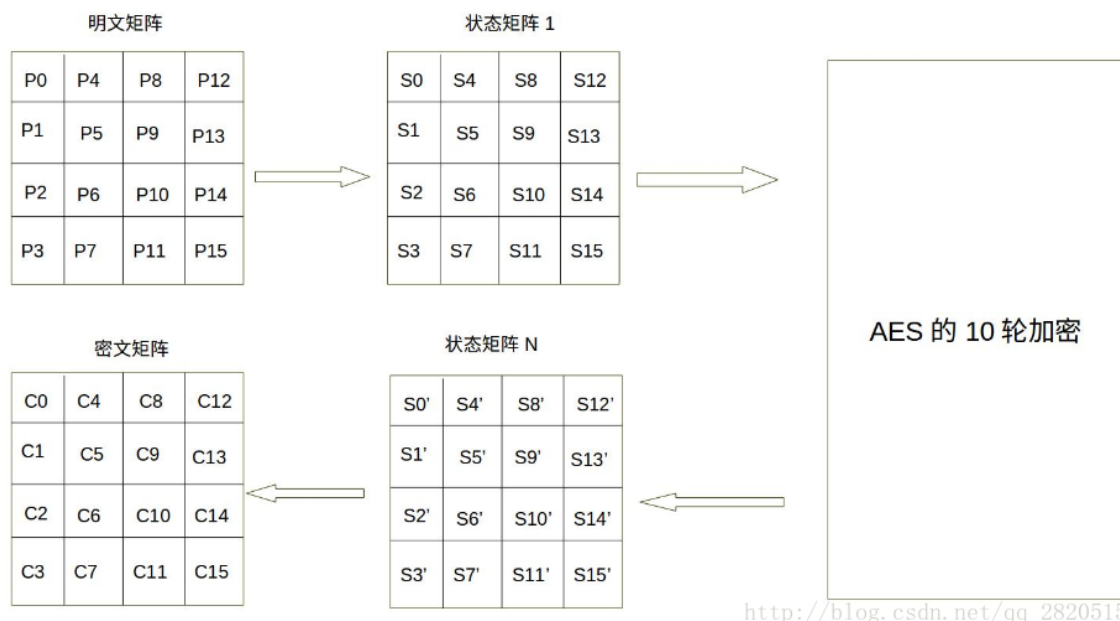
AES为分组密码，分组密码也就是把明文分成一组一组的，每组长度相等，每次加密一组数据，直到加密完整个明文。在AES标准规范中，分组长度只能是128位，也就是说，每个分组为16个字节（每个字节8位）。密钥的长度可以使用128位、192位或256位。密钥的长度不同，推荐加密轮数也不同，如下表所示：

AES	密钥长度（32位比特字）	分组长度(32位比特字)	加密轮数
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

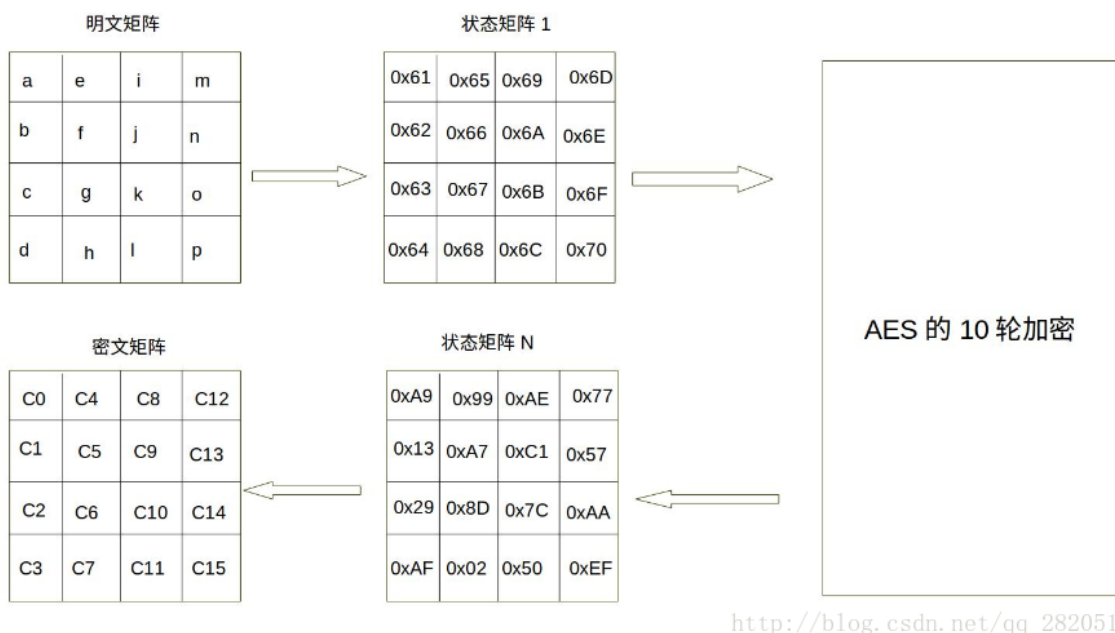
轮数在下面介绍，这里实现的是AES-128，也就是密钥的长度为128位，加密轮数为10轮。

上面说到，AES的加密公式为 $C = E(K, P)$ ，在加密函数E中，会执行一个轮函数，并且执行10次这个轮函数，这个轮函数的前9次执行的操作是一样的，只有第10次有所不同。也就是说，一个明文分组会被加密10轮。AES的核心就是实现一轮中的所有操作。

AES的处理单位是字节，128位的输入明文分组P和输入密钥K都被分成16个字节，分别记为P = P0 P1 ... P15 和 K = K0 K1 ... K15。如，明文分组为P = abcdefghijklmnop,其中的字符a对应P0，p对应P15。一般地，明文分组用字节为单位的正方形矩阵描述，称为状态矩阵。在算法的每一轮中，状态矩阵的内容不断发生变化，最后的结果作为密文输出。该矩阵中字节的排列顺序为从上到下、从左至右依次排列，如下图所示：

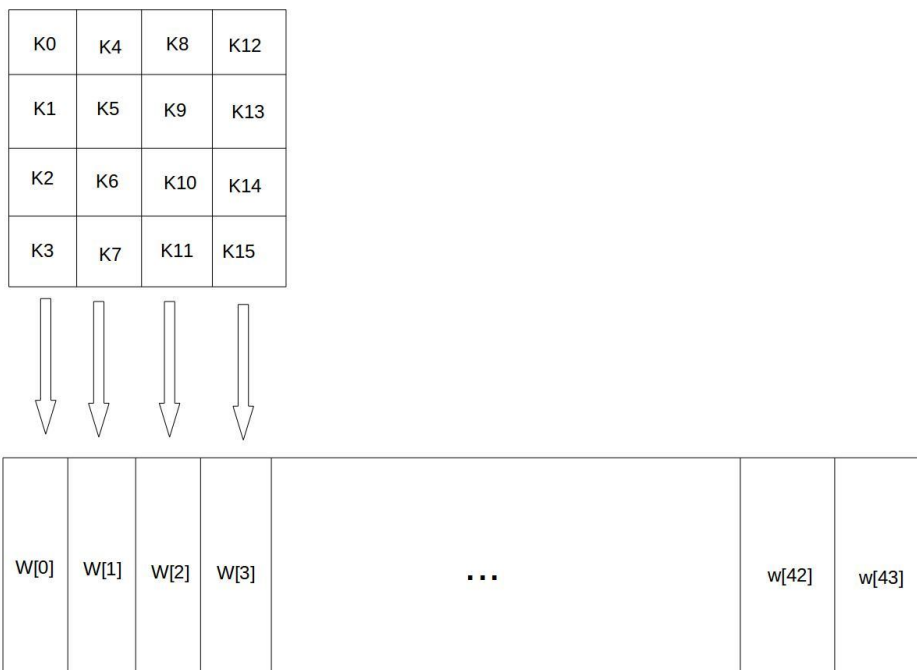


现在假设明文分组P为“abcdefghijklmnop”，则对应上面生成的状态矩阵图如下：



上图中，0x61为字符a的十六进制表示。可以看到，明文经过AES加密后，已经面目全非。

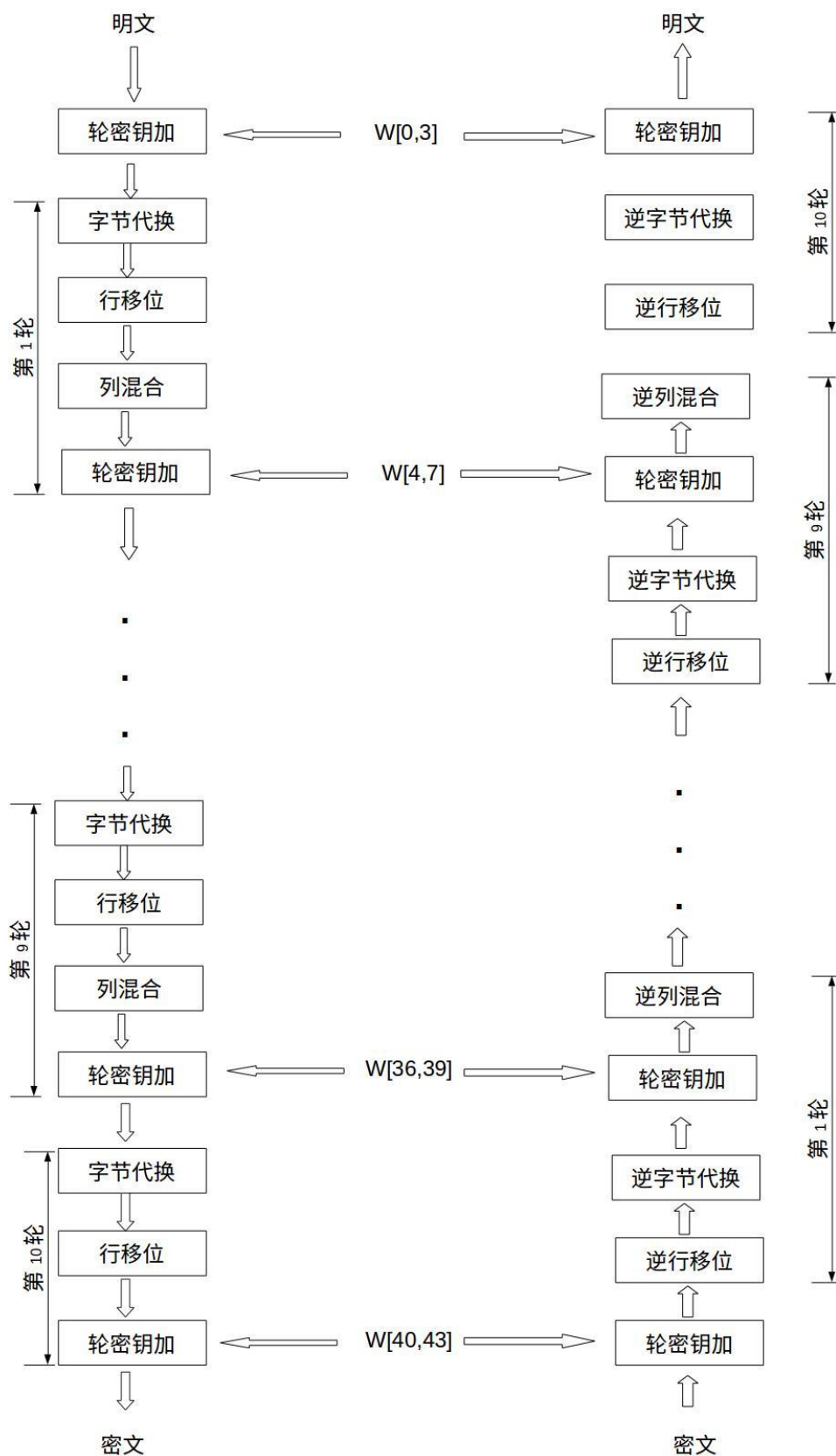
类似地，128位密钥也是用字节为单位的矩阵表示，矩阵的每一列被称为1个32位比特字。通过密钥编排函数该密钥矩阵被扩展成一个44个字组成的序列W[0],W[1], ..., W[43],该序列的前4个元素W[0],W[1],W[2],W[3]是原始密钥，用于加密运算中的初始密钥加（下面介绍）；后面40个字分为10组，每组4个字（128比特）分别用于10轮加密运算中的轮密钥加，如下图所示：



http://blog.csdn.net/qq_28205153

上图中，设K = “abcdefghijklmnop”，则K0 = a, K15 = p, W[0] = K0 K1 K2 K3 = “abcd”。

AES的整体结构如下图所示，其中的W[0,3]是指W[0]、W[1]、W[2]和W[3]串联组成的128位密钥。加密的第1轮到第9轮的轮函数一样，包括4个操作：字节代换、行位移、列混合和轮密钥加。最后一轮迭代不执行列混合。另外，在第一轮迭代之前，先将明文和原始密钥进行一次异或加密操作。



上图也展示了AES解密过程，解密过程仍为10轮，每一轮的操作是加密操作的逆操作。由于AES的4个轮操作都是可逆的，因此，解密操作的一轮就是顺序执行逆行移位、逆字节代换、轮密钥加和逆列混合。同加密操作类似，最后一轮不执行逆列混合，在第1轮解密之前，要执行1次密钥加操作。

下面分别介绍AES中一轮的4个操作阶段，这4分操作阶段使输入位得到充分的混淆。

一、字节代换

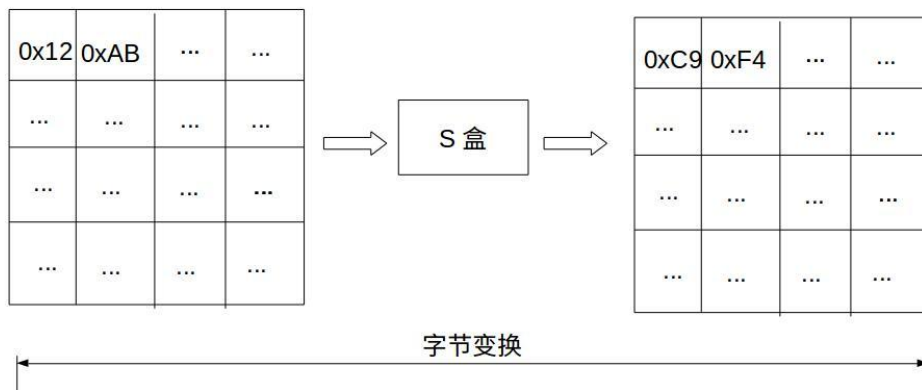
1.字节代换操作

AES的字节代换其实就是一个简单的查表操作。AES定义了一个S盒和一个逆S盒。

AES的S盒：

行/ 列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x63	0x7c	0x77	0x7b	0xf2	0x6b	0x6f	0xc5	0x30	0x01	0x67	0x2b	0xfe	0xd7	0xab	0x76
1	0xca	0x82	0xc9	0x7d	0xfa	0x59	0x47	0xf0	0xad	0xd4	0xa2	0xaf	0x9c	0xa4	0x72	0xc0
2	0xb7	0xfd	0x93	0x26	0x36	0x3f	0xf7	0xcc	0x34	0xa5	0xe5	0xf1	0x71	0xd8	0x31	0x15
3	0x04	0xc7	0x23	0xc3	0x18	0x96	0x05	0x9a	0x07	0x12	0x80	0xe2	0xeb	0x27	0xb2	0x75
4	0x09	0x83	0x2c	0x1a	0x1b	0x6e	0x5a	0xa0	0x52	0x3b	0xd6	0xb3	0x29	0xe3	0x2f	0x84
5	0x53	0xd1	0x00	0xed	0x20	0xfc	0xb1	0x5b	0x6a	0xcb	0xbe	0x39	0x4a	0x4c	0x58	0xcf
6	0xd0	0xef	0xaa	0xfb	0x43	0x4d	0x33	0x85	0x45	0xf9	0x02	0x7f	0x50	0x3c	0x9f	0xa8
7	0x51	0xa3	0x40	0x8f	0x92	0x9d	0x38	0xf5	0xbc	0xb6	0xda	0x21	0x10	0xff	0xf3	0xd2
8	0xcd	0x0c	0x13	0xec	0x5f	0x97	0x44	0x17	0xc4	0xa7	0x7e	0x3d	0x64	0x5d	0x19	0x73
9	0x60	0x81	0x4f	0xdc	0x22	0x2a	0x90	0x88	0x46	0xee	0xb8	0x14	0xde	0x5e	0x0b	0xdb
A	0xe0	0x32	0x3a	0x0a	0x49	0x06	0x24	0x5c	0xc2	0xd3	0xac	0x62	0x91	0x95	0xe4	0x79
B	0xe7	0xc8	0x37	0x6d	0x8d	0xd5	0x4e	0xa9	0x6c	0x56	0xf4	0xea	0x65	0x7a	0xae	0x08
C	0xba	0x78	0x25	0x2e	0x1c	0xa6	0xb4	0xc6	0xe8	0xdd	0x74	0x1f	0x4b	0xbd	0x8b	0x8a
D	0x70	0x3e	0xb5	0x66	0x48	0x03	0xf6	0x0e	0x61	0x35	0x57	0xb9	0x86	0xc1	0x1d	0x9e
E	0xe1	0xf8	0x98	0x11	0x69	0xd9	0x8e	0x94	0x9b	0x1e	0x87	0xe9	0xce	0x55	0x28	0xdf
F	0x8c	0xa1	0x89	0x0d	0xbf	0xe6	0x42	0x68	0x41	0x99	0x2d	0x0f	0xb0	0x54	0xbb	0x16

状态矩阵中的元素按照下面的方式映射为一个新的字节：把该字节的高4位作为行值，低4位作为列值，取出S盒或者逆S盒中对应的行的元素作为输出。例如，加密时，输出的字节S1为0x12,则查S盒的第0x01行和0x02列，得到值0xc9,然后替换S1原有的0x12为0xc9。状态矩阵经字节代换后的图如下：



http://blog.csdn.net/qq_28205153

2.字节代换逆操作

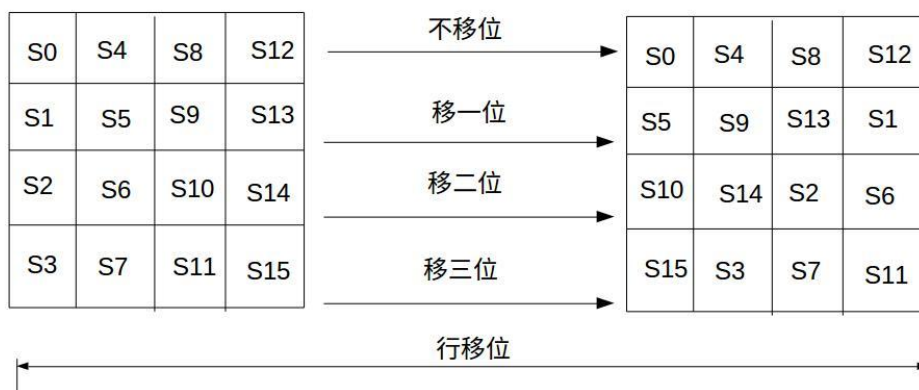
逆字节代换也就是查逆S盒来变换，逆S盒如下：

行/ 列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x52	0x09	0x6a	0xd5	0x30	0x36	0xa5	0x38	0xbf	0x40	0xa3	0x9e	0x81	0xf3	0xd7	0xfb
1	0x7c	0xe3	0x39	0x82	0x9b	0x2f	0xff	0x87	0x34	0x8e	0x43	0x44	0xc4	0xde	0xe9	0xcb
2	0x54	0x7b	0x94	0x32	0xa6	0xc2	0x23	0x3d	0xee	0x4c	0x95	0x0b	0x42	0xfa	0xc3	0x4e
3	0x08	0x2e	0xa1	0x66	0x28	0xd9	0x24	0xb2	0x76	0x5b	0xa2	0x49	0x6d	0x8b	0xd1	0x25
4	0x72	0xf8	0xf6	0x64	0x86	0x68	0x98	0x16	0xd4	0xa4	0x5c	0xcc	0x5d	0x65	0xb6	0x92
5	0x6c	0x70	0x48	0x50	0xfd	0xed	0xb9	0xda	0x5e	0x15	0x46	0x57	0xa7	0x8d	0x9d	0x84
6	0x90	0xd8	0xab	0x00	0x8c	0xbc	0xd3	0x0a	0xf7	0xe4	0x58	0x05	0xb8	0xb3	0x45	0x06
7	0xd0	0x2c	0x1e	0x8f	0xca	0x3f	0x0f	0x02	0xc1	0xaf	0xbd	0x03	0x01	0x13	0x8a	0x6b
8	0x3a	0x91	0x11	0x41	0x4f	0x67	0xdc	0xea	0x97	0xf2	0xcf	0xce	0xf0	0xb4	0xe6	0x73
9	0x96	0xac	0x74	0x22	0xe7	0xad	0x35	0x85	0xe2	0xf9	0x37	0xe8	0x1c	0x75	0xdf	0x6e
A	0x47	0xf1	0x1a	0x71	0x1d	0x29	0xc5	0x89	0x6f	0xb7	0x62	0x0e	0xaa	0x18	0xbe	0x1b
B	0xfc	0x56	0x3e	0x4b	0xc6	0xd2	0x79	0x20	0x9a	0xdb	0xc0	0xfe	0x78	0xcd	0x5a	0xf4
C	0x1f	0xdd	0xa8	0x33	0x88	0x07	0xc7	0x31	0xb1	0x12	0x10	0x59	0x27	0x80	0xec	0x5f
D	0x60	0x51	0x7f	0xa9	0x19	0xb5	0x4a	0x0d	0x2d	0xe5	0x7a	0x9f	0x93	0xc9	0x9c	0xef
E	0xa0	0xe0	0x3b	0x4d	0xae	0x2a	0xf5	0xb0	0xc8	0xeb	0xbb	0x3c	0x83	0x53	0x99	0x61
F	0x17	0x2b	0x04	0x7e	0xba	0x77	0xd6	0x26	0xe1	0x69	0x14	0x63	0x55	0x21	0x0c	0x7d

二、行移位

1.行移位操作

行移位是一个简单的左循环移位操作。当密钥长度为128比特时，状态矩阵的第0行左移0字节，第1行左移1字节，第2行左移2字节，第3行左移3字节，如下图所示：



http://blog.csdn.net/qq_28205153

2.行移位的逆变换

行移位的逆变换是将状态矩阵中的每一行执行相反的移位操作，例如AES-128中，状态矩阵的第0行右移0字节，第1行右移1字节，第2行右移2字节，第3行右移3字节。

三、列混合

1.列混合操作

列混合变换是通过矩阵相乘来实现的，经行移位后的状态矩阵与固定的矩阵相乘，得到混淆后的状态矩阵，如下图所示的公式所示：

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

http://blog.csdn.net/qq_28205153

状态矩阵中的第j列(0 ≤ j ≤ 3)的列混合可以表示为下图所示：

$$\begin{aligned} s'_{0,j} &= (2 * s_{0,j}) \oplus (3 * s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 * s_{1,j}) \oplus (3 * s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 * s_{2,j}) \oplus (3 * s_{3,j}) \\ s'_{3,j} &= (3 * s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 * s_{3,j}) \end{aligned}$$

http://blog.csdn.net/qq_28205153

其中，矩阵元素的乘法和加法都是定义在基于GF(2⁸)上的二元运算,并不是通常意义上的乘法和加法。这里涉及到一些信息安全上的数学知识，不过不懂这些知识也行。其实这种二元运算的加法等价于两个字节的异或，乘法则复杂一点。对于一个8位的二进制数来说，使用域上的乘法乘以(00000010)等价于左移1位(低位补0)后，再根据情况同(00011011)进行异或运算，设S1 = (a7 a6 a5 a4 a3 a2 a1 a0)，刚0x02 * S1如下图所示：

$$(00000010) * (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) = \begin{cases} (a_6 a_5 a_4 a_3 a_2 a_1 a_0 0), & a_7 = 0 \\ ((a_6 a_5 a_4 a_3 a_2 a_1 a_0 0) \oplus (00011011)), & a_7 = 1 \end{cases}$$

http://blog.csdn.net/qq_28205153

也就是说，如果a7为1，则进行异或运算，否则不进行。

类似地，乘以(00000100)可以拆分成两次乘以(00000010)的运算：

$$(00000100) * (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) = (00000010) * (00000010) * (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0)$$

http://blog.csdn.net/qq_28205153

乘以(0000 0011)可以拆分成先分别乘以(0000 0001)和(0000 0010)，再将两个乘积异或：

$$\begin{aligned} (00000011) * (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) &= [(00000010) \oplus (00000001)] * (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) \\ &= [(00000010) \oplus (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0)] \oplus (a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) \end{aligned}$$

http://blog.csdn.net/qq_28205153

因此，我们只需要实现乘以2的函数，其他数值的乘法都可以通过组合来实现。

下面举个具体的例子,输入的状态矩阵如下：

C9	E5	FD	2B
7A	F2	78	6E
63	9C	26	67
B0	A7	82	E5

下面，进行列混合运算：

以第一列的运算为例：

$$\begin{aligned} s'_{0,0} &= (2 * 0xC9) \oplus (3 * 0x7A) \oplus 0x63 \oplus 0xB0 = 0xD4 \\ s'_{1,0} &= 0xC9 \oplus (2 * 0x7A) \oplus (3 * 0x63) \oplus 0xB0 = 0x28 \\ s'_{2,0} &= 0xC9 \oplus 0x7A \oplus (2 * 0x63) \oplus (3 * 0xB0) = 0xBE \\ s'_{3,0} &= (3 * 0xC9) \oplus 0x7A \oplus 0x63 \oplus (2 * 0xB0) = 0x22 \end{aligned}$$

http://blog.csdn.net/qq_28205153

其它列的计算就不列举了，列混合后生成的新状态矩阵如下：

D4	E7	CD	66
28	02	E5	BB
BE	C6	D6	BF
22	0F	DF	A5

2.列混合逆运算

逆向列混合变换可由下图的矩阵乘法定义：

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

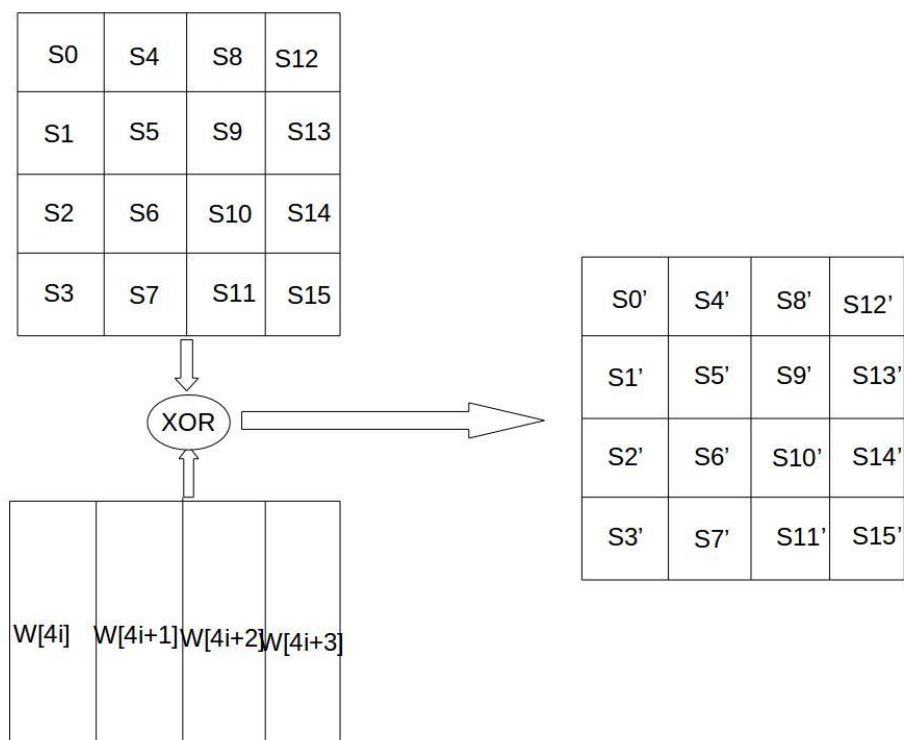
↩

http://blog.csdn.net/qq_28205153

可以验证，逆变换矩阵同正变换矩阵的乘积恰好为单位矩阵。

四、轮密钥加

轮密钥加是将128位轮密钥 K_i 同状态矩阵中的数据进行逐位异或操作，如下图所示。其中，密钥 K_i 中每个字 $W[4i], W[4i+1], W[4i+2], W[4i+3]$ 为32位比特字，包含4个字节，他们的生成算法下面在下面介绍。轮密钥加过程可以看成是字逐位异或的结果，也可以看成字节级别或者位级别的操作。也就是说，可以看成 $S_0 \ S_1 \ S_2 \ S_3$ 组成的32位字与 $W[4i]$ 的异或运算。

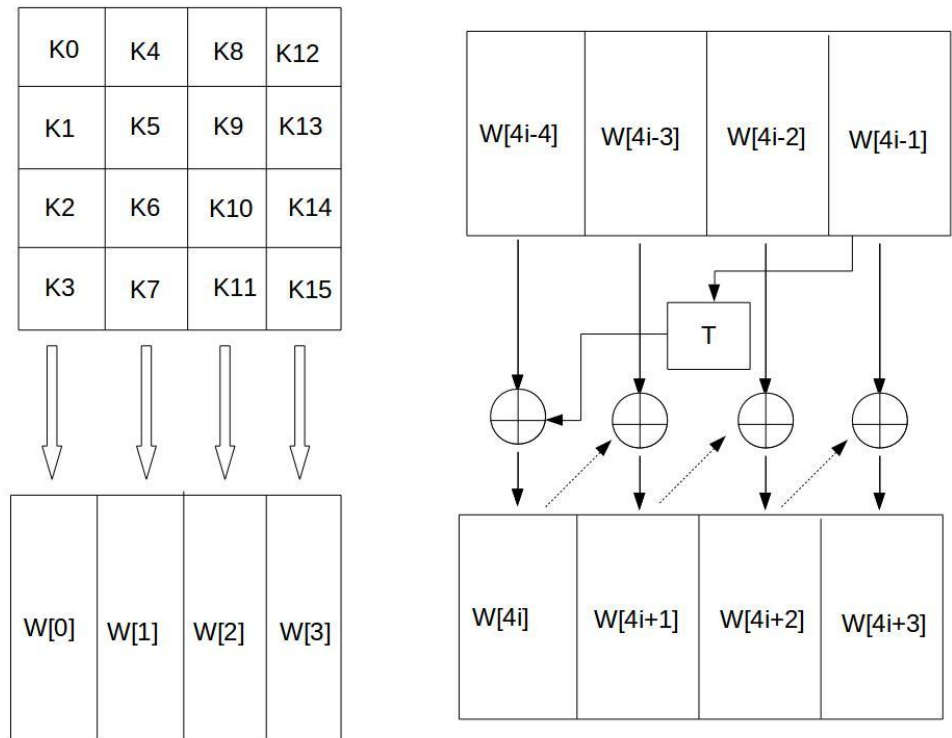


http://blog.csdn.net/qq_28205153

轮密钥加的逆运算同正向的轮密钥加运算完全一致，这是因为异或的逆操作是其自身。轮密钥加非常简单，但却能够影响S数组中的每一位。

五、密钥扩展

AES首先将初始密钥输入到一个44的状态矩阵中，如下图所示。



http://blog.csdn.net/qq_28205153

这个44矩阵的每一列的4个字节组成一个字，矩阵4列的4个字依次命名为W[0]、W[1]、W[2]和W[3]，它们构成一个以字为单位的数组W。例如，设密钥K为“abcdefghijlmnop”，则K0 = ‘a’，K1 = ‘b’，K2 = ‘c’，K3 = ‘d’，W[0] = “abcd”。

接着，对W数组扩充40个新列，构成总共44列的扩展密钥数组。新列以如下的递归方式产生：

1.如果i不是4的倍数，那么第i列由如下等式确定：

$$W[i]=W[i-4]\oplus W[i-1]$$

2.如果i是4的倍数，那么第i列由如下等式确定：

$$W[i]=W[i-4]\oplus T(W[i-1])$$

其中，T是一个有点复杂的函数。

函数T由3部分组成：字循环、字节代换和轮常量异或，这3部分的作用分别如下。

a.字循环：将1个字中的4个字节循环左移1个字节。即将输入字[b0, b1, b2, b3]变换成[b1,b2,b3,b0]。

b.字节代换：对字循环的结果使用S盒进行字节代换。

c.轮常量异或：将前两步的结果同轮常量Rcon[j]进行异或，其中j表示轮数。

轮常量Rcon[j]是一个字，其值见下表。

j	1	2	3	4	5
Rcon[j]	01 00 00 00	02 00 00 00	04 00 00 00	08 00 00 00	10 00 00 00
j	6	7	8	9	10
Rcon[j]	20 00 00 00	40 00 00 00	80 00 00 00	1B 00 00 00	36 00 00 00

下面举个例子：

设初始的128位密钥为：

3C A1 0B 21 57 F0 19 16 90 2E 13 80 AC C1 07 BD

那么4个初始值为：

$W[0] = 3C\ A1\ 0B\ 21$

$W[1] = 57\ F0\ 19\ 16$

$W[2] = 90\ 2E\ 13\ 80$

$W[3] = AC\ C1\ 07\ BD$

下面求扩展的第1轮的子密钥($W[4], W[5], W[6], W[7]$)。

由于4是4的倍数，所以：

$W[4] = W[0] \oplus T(W[3])$

$T(W[3])$ 的计算步骤如下：

\1. 循环地将 $W[3]$ 的元素移位：AC C1 07 BD变成C1 07 BD AC；

\2. 将 C1 07 BD AC 作为S盒的输入，输出为78 C5 7A 91；

\3. 将78 C5 7A 91与第一轮轮常量 $Rcon[1]$ 进行异或运算，将得到79 C5 7A 91，因此， $T(W[3])=79\ C5\ 7A\ 91$ ，故

$W[4] = 3C\ A1\ 0B\ 21 \oplus 79\ C5\ 7A\ 91 = 45\ 64\ 71\ B0$

其余的3个子密钥段的计算如下：

$W[5] = W[1] \oplus W[4] = 57\ F0\ 19\ 16 \oplus 45\ 64\ 71\ B0 = 12\ 94\ 68\ A6$

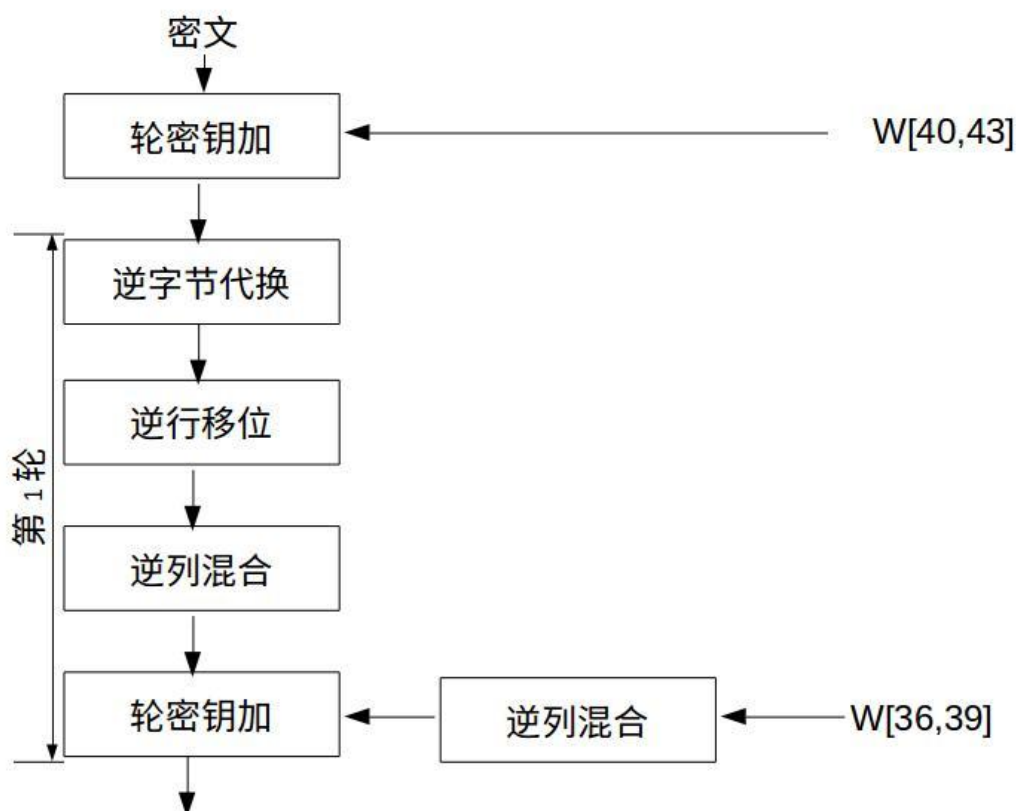
$W[6] = W[2] \oplus W[5] = 90\ 2E\ 13\ 80 \oplus 12\ 94\ 68\ A6 = 82\ BA\ 7B\ 26$

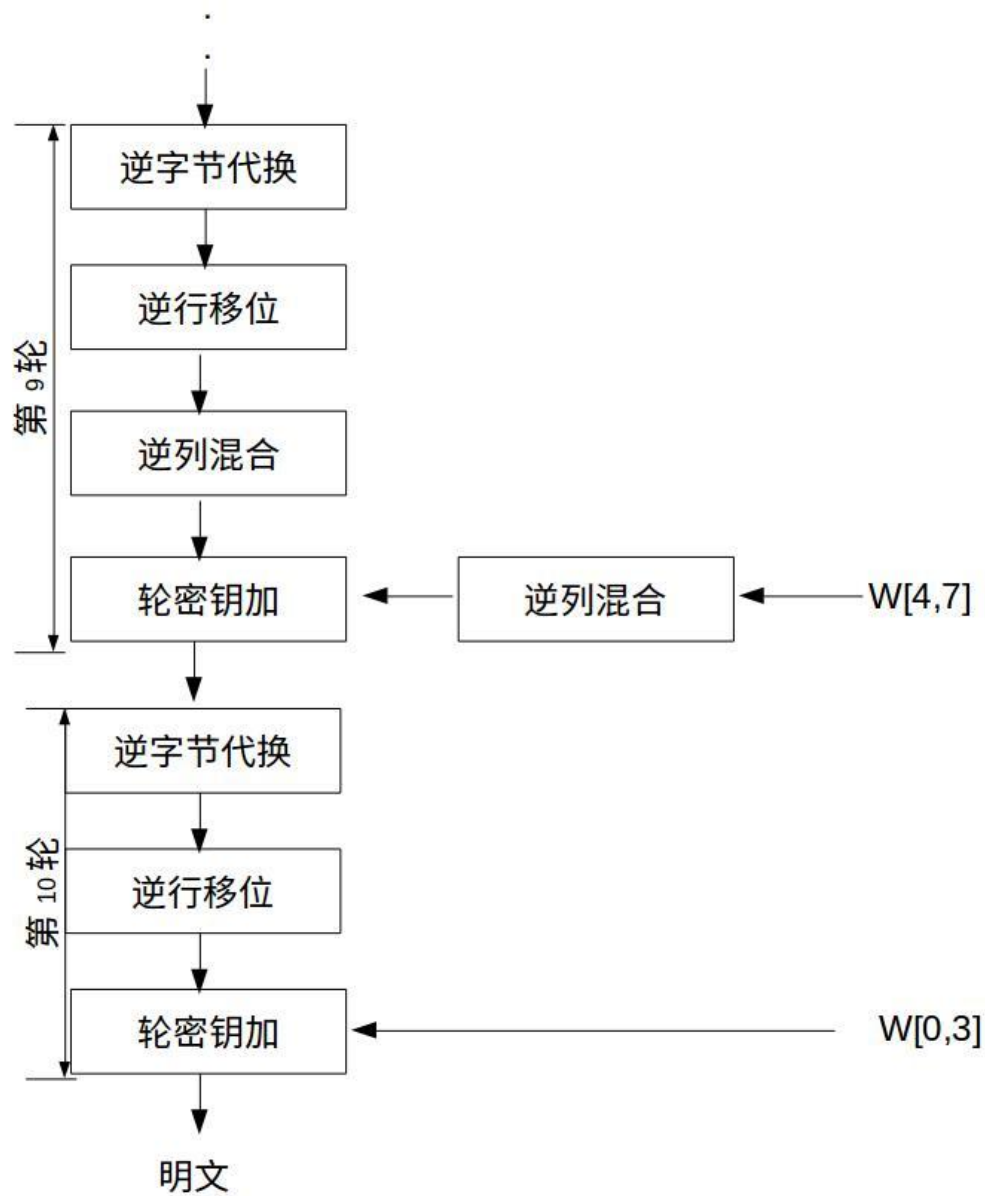
$W[7] = W[3] \oplus W[6] = AC\ C1\ 07\ BD \oplus 82\ BA\ 7B\ 26 = 2E\ 7B\ 7C\ 9B$

所以，第一轮的密钥为 45 64 71 B0 12 94 68 A6 82 BA 7B 26 2E 7B 7C 9B。

六、AES解密

在文章开始的图中，有AES解密的流程图，可以对应那个流程图来进行解密。下面介绍的是另一种等价的解密模式，流程图如下图所示。这种等价的解密模式使得解密过程各个变换的使用顺序同加密过程的顺序一致，只是用逆变换取代原来的变换。





http://blog.csdn.net/qq_28205153

软件系统设计

XTime运算

```
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)
```

字节替代

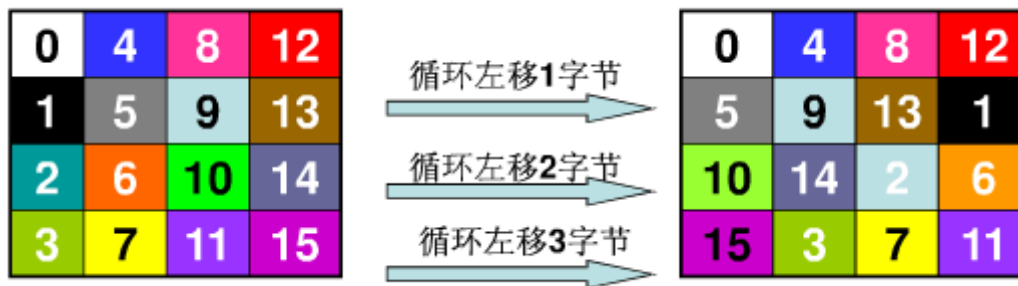
```
def __sub_bytes(self, s):
    for i in range(4):
        for j in range(4):
            s[i][j] = Sbox[s[i][j]]
```

AES的字节代换其实就是一个简单的查表操作。AES定义了一个S盒和一个逆S盒,状态矩阵中的元素按照下面的方式映射为一个新的字节：把该字节的高4位作为行值，低4位作为列值，取出S盒或者逆S盒中对应的行的元素作为输出

逆字节替换

```
def __inv_sub_bytes(self, s):
    for i in range(4):
        for j in range(4):
            s[i][j] = InvSbox[s[i][j]]
```

行移位变换

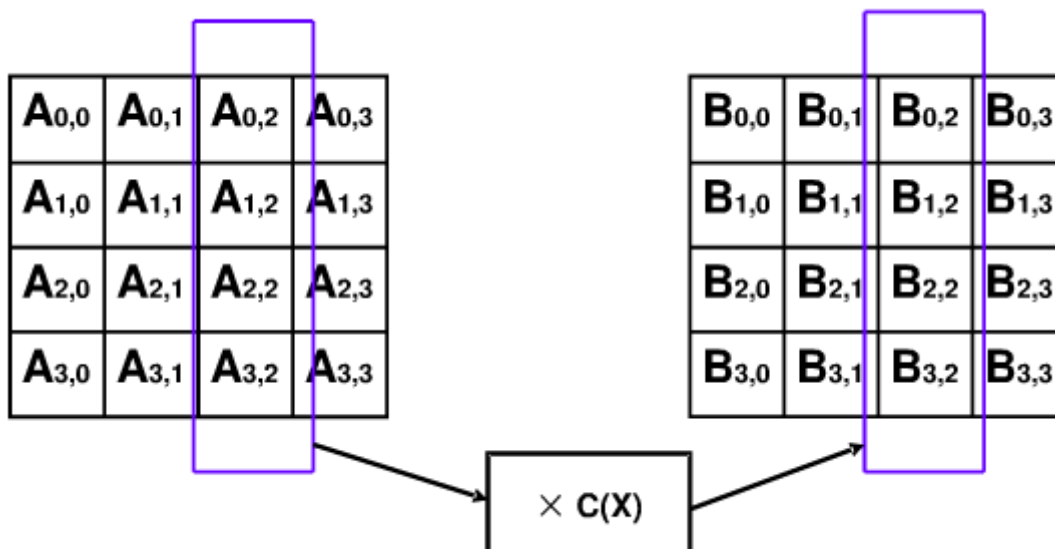


```
def __shift_rows(self, s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]
```

逆行移位变换

```
def __inv_shift_rows(self, s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]
```

单列列混淆变换



```
def __mix_single_column(self, a):
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)
```

列混淆变换

```
def __mix_columns(self, s):
    for i in range(4):
        self.__mix_single_column(s[i])
```

逆混淆列变换

```
def __inv_mix_columns(self, s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v
    self.__mix_columns(s)
```

密钥处理

轮密钥加变换

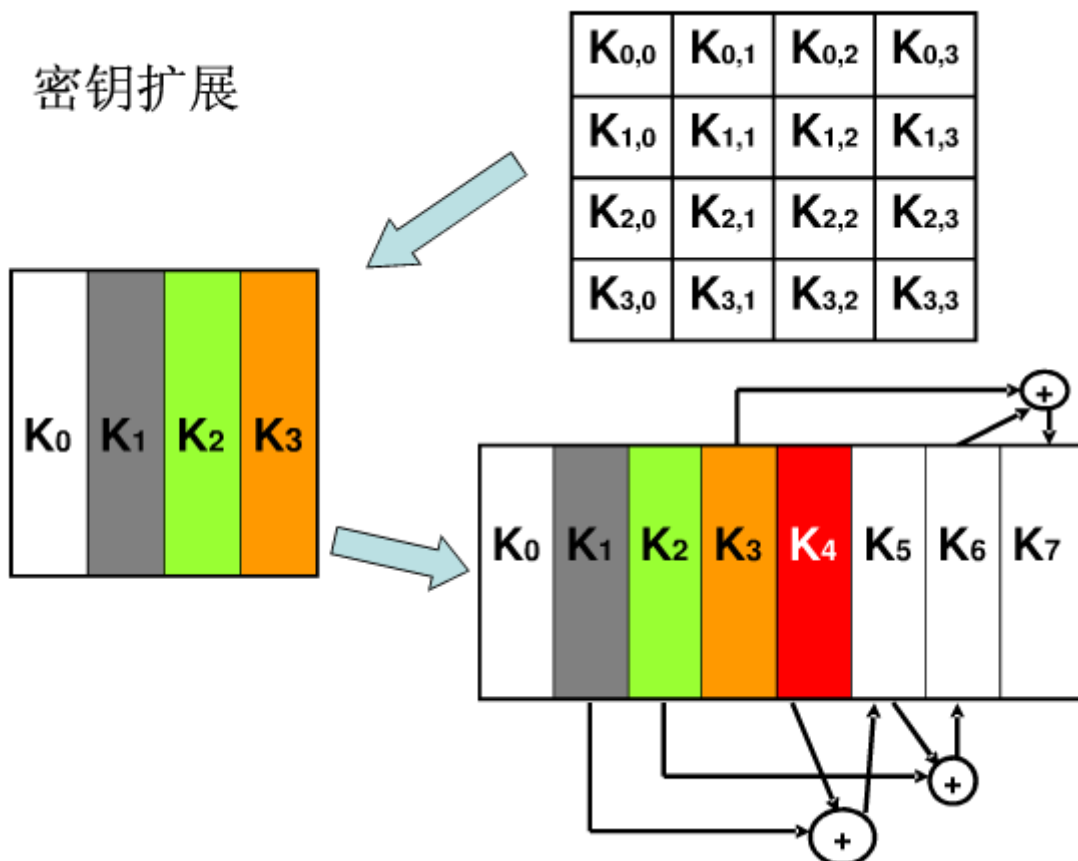
```
def __add_round_key(self, s, k):
    for i in range(4):
        for j in range(4):
            s[i][j] ^= k[i][j]
```

密钥扩展

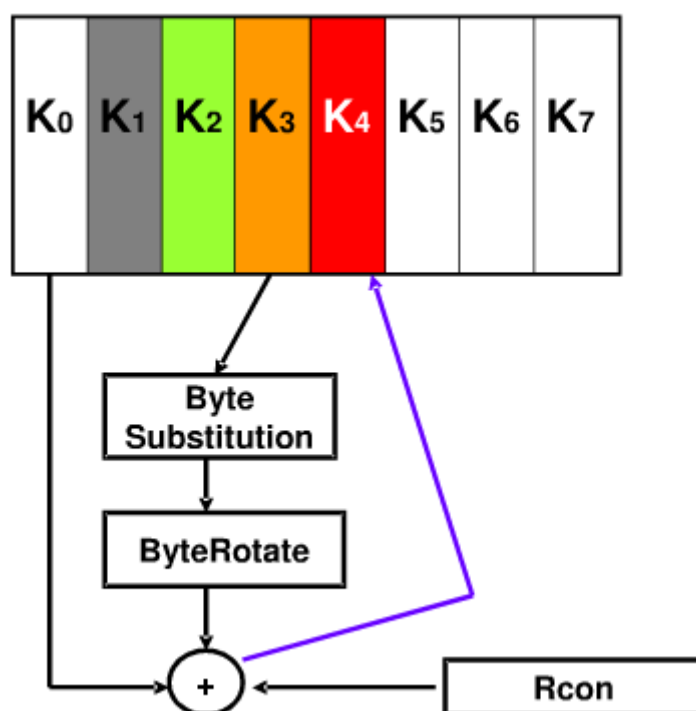
将输入的密钥扩展为11组128位密钥组，其中第0组为输入密钥本身

其后第n组第i列为 第n-1组第i列 与 第n组第i-1列之和（模2加法， $1 \leq i \leq 3$ ）

密钥扩展



对于每一组 第一列即 $i=0$ ，有特殊的处理



将前一列即第 $n-1$ 组第3列的4个字节循环左移1个字节，

并对每个字节进行字节替代变换SubBytes

将第一行（即第一个字节）与轮常量 $rc[n]$ 相加

最后再与前一组该列相加

```
def __init__(self, master_key):
```

```

self.change_key(master_key)
def change_key(self, master_key):
    self.round_keys = text2matrix(master_key)
    for i in range(4, 4 * 11):
        self.round_keys.append([])
        if i % 4 == 0:
            byte = self.round_keys[i - 4][0] \
                ^ Sbox[self.round_keys[i - 1][1]] \
                ^ Rcon[int(i/4)]
            self.round_keys[i].append(byte)
            for j in range(1, 4):
                byte = self.round_keys[i - 4][j] \
                    ^ Sbox[self.round_keys[i - 1][(j + 1) % 4]]
                self.round_keys[i].append(byte)
        else:
            for j in range(4):
                byte = self.round_keys[i - 4][j] \
                    ^ self.round_keys[i - 1][j]
                self.round_keys[i].append(byte)

```

加密函数

```

def encrypt(self, plaintext):
    self.plain_state = text2matrix(plaintext)
    self.__add_round_key(self.plain_state, self.round_keys[:4])
    for i in range(1, 10):
        self.__round_encrypt(self.plain_state, self.round_keys[4 * i : 4 * (i + 1)])
        self.__sub_bytes(self.plain_state)
        self.__shift_rows(self.plain_state)
        self.__add_round_key(self.plain_state, self.round_keys[40:])
    return matrix2text(self.plain_state)

```

解密函数

```

def decrypt(self, ciphertext):
    self.cipher_state = text2matrix(ciphertext)
    self.__add_round_key(self.cipher_state, self.round_keys[40:])
    self.__inv_shift_rows(self.cipher_state)
    self.__inv_sub_bytes(self.cipher_state)
    for i in range(9, 0, -1):
        self.__round_decrypt(self.cipher_state, self.round_keys[4 * i : 4 * (i + 1)])
        self.__add_round_key(self.cipher_state, self.round_keys[:4])
    return matrix2text(self.cipher_state)

```

AES的S盒

参考了GitHub和看雪论坛的代码实现

```

#include <windows.h>
#include <iostream>
#include "MatrixGF2.h"
#include "PolynomialGF2.h"
MatrixGF2<BYTE> g_mtxPositiveBox(16, 16, 0);
MatrixGF2<BYTE> g_mtxReverseBox(16, 16, 0);

```

```

MatrixGF2<BYTE> g_mtxBytePositiveTransformMatrix(8, 8, 0);
MatrixGF2<BYTE> g_mtxByteReverseTransformMatrix(8, 8, 0);
BYTE g_bytPositiveFixed = 0;
BYTE g_bytReverseFixed = 0;
PolynomialGF2 g_polyModle;
#define MAKEBYTE(_high, _low) (((_high) << 4) | (_low))
PolynomialGF2 PolynomialEuclidEx(const PolynomialGF2 &kref_polyA,
                                const PolynomialGF2 &kref_polyB,
                                PolynomialGF2 &ref_polyX,
                                PolynomialGF2 &ref_polyY)
{
    if (kref_polyB.EqualZero())
    {
        throw std::runtime_error("zero!");
    }
    int iHighestBitIndex = kref_polyA.GetHighestBitIndex();
    PolynomialGF2 polyQuotient(iHighestBitIndex + 1, 0);
    PolynomialGF2 polyRemainder(iHighestBitIndex + 1, 0);
    PolynomialGF2 polyLastLastX(iHighestBitIndex + 1, 0);
    polyLastLastX = 1;
    PolynomialGF2 polyLastLastY(iHighestBitIndex + 1, 0);
    PolynomialGF2 polyLastLastRemainder = kref_polyA;
    PolynomialGF2 polyLastX(iHighestBitIndex + 1, 0);
    PolynomialGF2 polyLastY(iHighestBitIndex + 1, 0);
    polyLastY = 1;
    PolynomialGF2 polyLastRemainder = kref_polyB;
    ref_polyX.Clear();
    ref_polyY.Clear();
    ref_polyX.Insert(0, iHighestBitIndex + 1, 0);
    ref_polyY.Insert(0, iHighestBitIndex + 1, 0);
    ref_polyY = 1;
    do
    {
        polyLastLastRemainder.Division(polyLastRemainder,
                                       polyQuotient,
                                       polyRemainder);
        if (polyRemainder.EqualZero())
        {
            break;
        }
        ref_polyX = polyLastLastX - polyLastX * polyQuotient;
        ref_polyY = polyLastLastY - polyLastY * polyQuotient;
        polyLastLastRemainder = polyLastRemainder;
        polyLastRemainder = polyRemainder;
        polyLastLastX = polyLastX;
        polyLastLastY = polyLastY;
        polyLastX = ref_polyX;
        polyLastY = ref_polyY;
    } while (TRUE);
    return polyLastRemainder;
}
void InitByteTransformMatrix(PolynomialGF2 &ref_polyInit,
                             MatrixGF2<BYTE> &ref_mtxTarget)
{
    size_t uiColumnNumber = ref_mtxTarget.GetColumnNumber();
    if (ref_polyInit.GetSize() != ref_mtxTarget.GetColumnNumber())
    {
        ref_polyInit.PaddingZero(uiColumnNumber);
    }
}

```

```

}
size_t uiRowNumber = uiColumnNumber;
for (size_t cntY = 0; cntY < uiRowNumber; cntY++)
{
    size_t ioffset = cntY;
    for (size_t cntX = 0; cntX < uiColumnNumber; cntX++)
    {
        size_t iActualPos = cntX + ioffset;
        if (iActualPos >= uiColumnNumber)
        {
            iActualPos -= uiColumnNumber;
        }
        ref_mtxTarget[cntY][iActualPos] = ref_polyInit[cntX];
    }
}
}

void InitPositiveBox()
{
    MatrixGF2<BYTE> mtxFixed =
    PolynomialGF2(g_bytPositiveFixed).GetDequeFormat();

    for (size_t cntY = 0; cntY < 16; cntY++)
    {
        for (size_t cntX = 0; cntX < 16; cntX++)
        {
            BYTE bytSource = (BYTE)MAKEBYTE(cntY, cntX);
            if (0 == bytSource)
            {
                g_mtxPositiveBox[cntY][cntX] = g_bytPositiveFixed;
                continue;
            }
            PolynomialGF2 polyInverseElement;
            PolynomialGF2 polyX;
            PolynomialGF2 polySource = bytSource;
            if (!polySource.EqualZero())
            {
                PolynomialGF2 polyResult =
                    PolynomialEuclidEx(g_polyModle,
                                        polySource,
                                        polyX,
                                        polyInverseElement);
            }
            else
            {
                polyInverseElement = { 0 };
            }
            polyInverseElement.ClearInvalidZero();
            MatrixGF2<BYTE> mtxValue = polyInverseElement.GetDequeFormat();

            mtxValue.PaddingRow(
                g_mtxBytePositiveTransformMatrix.GetColumnNumber()
            );
            mtxFixed.PaddingRow(mtxValue.GetRowNumber());

            mtxValue = g_mtxBytePositiveTransformMatrix * mtxValue + mtxFixed;
            g_mtxPositiveBox[cntY][cntX] =
                (BYTE)PolynomialGF2(mtxValue.Transform2Vector()).ToNumber();
        }
    }
}

```

```

    }
}

void InitReverseBox()
{
    MatrixGF2<BYTE> mtxFixed =
PolynomialGF2(g_bytReverseFixed).GetDequeFormat();
    for (size_t cntY = 0; cntY < 16; cntY++)
    {
        for (size_t cntX = 0; cntX < 16; cntX++)
        {
            BYTE bytSource = (BYTE)MAKEBYTE(cntY, cntX);
            MatrixGF2<BYTE> mtxValue = bytSource;
            int iValueColumnNumber =
                g_mtxByteReverseTransformMatrix.GetColumnNumber();
            if (0 == bytSource)
            {
                mtxValue.InsertRow(0, iValueColumnNumber, 1, 0);
            }
            mtxValue.PaddingRow(iValueColumnNumber);
            mtxFixed.PaddingRow(mtxValue.GetRowNumber());
            mtxValue = g_mtxByteReverseTransformMatrix * mtxValue + mtxFixed;
            PolynomialGF2 polyInverseElement;
            PolynomialGF2 polyX;
            PolynomialGF2 polyValue = mtxValue.Transform2Vector();
            if (!polyValue.EqualZero())
            {
                PolynomialGF2 polyResult =
                    PolynomialEuclidEx(g_polyModle,
                                        polyValue,
                                        polyX,
                                        polyInverseElement);
            }
            else
            {
                polyInverseElement = { 0 };
            }

            g_mtxReverseBox[cntY][cntX] = (BYTE)polyInverseElement.ToNumber();
        }
    }
}

void InitStandardBox()
{
    g_polyModle = { 1, 1, 0, 1, 1, 0, 0, 0, 1 };
    PolynomialGF2 polyPositiveSeed = { 1, 0, 0, 0, 1, 1, 1, 1 };
    InitByteTransformMatrix(polyPositiveSeed, g_mtxBytePositiveTransformMatrix);
    PolynomialGF2 polyReverseSeed = { 0, 0, 1, 0, 0, 1, 0, 1 };
    InitByteTransformMatrix(polyReverseSeed, g_mtxByteReverseTransformMatrix);
    g_bytPositiveFixed = 0x63;
    g_bytReverseFixed = 0x05;
    InitPositiveBox();
    InitReverseBox();
    printf("Positive box: \n%s\n",
        g_mtxPositiveBox.GetWrittenFormat().c_str());
    printf("Reverse box: \n%s\n",
        g_mtxReverseBox.GetWrittenFormat().c_str());
}

int main()

```

```

{
    InitStandardBox();
    return 0;
}

```

重要的实现细节

GF2域上的计算

```

PolynomialGF2 &PolynomialGF2::operator=(const PolynomialGF2 &kref_polyRight)
{
    m_deqPolynomial = kref_polyRight.m_deqPolynomial;
    return *this;
}
PolynomialGF2 &PolynomialGF2::operator=(PolynomialGF2 &&rvalue_polyRight)
{
    m_deqPolynomial = std::move(rvalue_polyRight.m_deqPolynomial);
    return *this;
}
PolynomialGF2 PolynomialGF2::operator+(const PolynomialGF2 &ref_polyRight)
{
    PolynomialGF2 polyResult = *this;
    int iRightHighestBitIndex = ref_polyRight.GetHighestBitIndex();
    int iLeftHighestBitIndex = polyResult.GetHighestBitIndex();
    if (iRightHighestBitIndex > iLeftHighestBitIndex)
    {
        int iGap = iRightHighestBitIndex - iLeftHighestBitIndex;
        polyResult.Insert(iLeftHighestBitIndex + 1, iGap, 0);
    }

    for (size_t cntI = 0; cntI < (size_t)(iRightHighestBitIndex + 1); cntI++)
    {
        polyResult[cntI] ^= ref_polyRight[cntI];
    }
    return polyResult;
}
PolynomialGF2 PolynomialGF2::operator-(const PolynomialGF2 &ref_polyRight)
{
    PolynomialGF2 polyResult = *this;
    int iRightHighestBitIndex = ref_polyRight.GetHighestBitIndex();
    int iLeftHighestBitIndex = polyResult.GetHighestBitIndex();
    if (iRightHighestBitIndex > iLeftHighestBitIndex)
    {
        int iGap = iRightHighestBitIndex - iLeftHighestBitIndex;
        polyResult.Insert(iLeftHighestBitIndex + 1, iGap, 0);
    }

    for (size_t cntI = 0; cntI < ref_polyRight.GetSize(); cntI++)
    {
        polyResult[cntI] ^= ref_polyRight[cntI];
    }
    return polyResult;
}
PolynomialGF2 PolynomialGF2::operator-(const PolynomialGF2 &ref_polyRight) const
{
    PolynomialGF2 polyResult = *this;

```

```

    int iRightHighestBitIndex = ref_polyRight.GetHighestBitIndex();
    int iLeftHighestBitIndex = polyResult.GetHighestBitIndex();
    if (iRightHighestBitIndex > iLeftHighestBitIndex)
    {
        int iGap = iRightHighestBitIndex - iLeftHighestBitIndex;
        polyResult.Insert(iLeftHighestBitIndex + 1, iGap, 0);
    }

    for (size_t cntI = 0; cntI < ref_polyRight.GetSize(); cntI++)
    {
        polyResult[cntI] ^= ref_polyRight[cntI];
    }
    return polyResult;
}

PolynomialGF2 PolynomialGF2::operator*(const PolynomialGF2 &ref_polyRight)
{
    PolynomialGF2 polyResult;
    for (size_t cntI = 0; cntI < ref_polyRight.GetSize(); cntI++)
    {
        PolynomialGF2 polyTmp = *this;
        if (1 == ref_polyRight[cntI])
        {
            polyTmp.Insert(0, cntI, 0);
            polyResult = polyResult + polyTmp;
        }
    }
    return polyResult;
}

PolynomialGF2 PolynomialGF2::operator/(const PolynomialGF2 &kref_right)
{
    return *this;
} //! Polynomial::operator/() END

PolynomialGF2 PolynomialGF2::operator%(const PolynomialGF2 &kref_polyRight)
{
    PolynomialGF2 polyResult;
    int iLeftHighestBitIndex = GetHighestBitIndex();
    int iRightHighestBitIndex = kref_polyRight.GetHighestBitIndex();

    if (iLeftHighestBitIndex == iRightHighestBitIndex)
    {
        polyResult = kref_polyRight - *this;
    }
    else if (iLeftHighestBitIndex < iRightHighestBitIndex)
    {
        polyResult = *this;
    }
    else if (iLeftHighestBitIndex > iRightHighestBitIndex)
    {
        PolynomialGF2 polyQuotient;
        Division(kref_polyRight, polyQuotient, polyResult);
    }

    return polyResult;
}

PolynomialGF2 &PolynomialGF2::operator<=(const size_t &kref_uiNumber)
{
    m_deqPolynomial.insert(m_deqPolynomial.cbegin(), kref_uiNumber, 0);
    if (m_bEnableLeftMoveLimit &&

```

```

        GetHightestBitIndex() >= (int)m_uiLeftMoveLimitBit)
    {
        m_deqPolynomial.erase(std::next(m_deqPolynomial.cbegin(),
                                         m_uiLeftMoveLimitBit),
                              m_deqPolynomial.cend());
    }
    return *this;
}
PolynomialGF2 &PolynomialGF2::operator^=(const PolynomialGF2 &kref_polyRight)
{
    size_t uiheighBit = 0;
    size_t uiLeftSize = m_deqPolynomial.size();
    size_t uiRightSize = kref_polyRight.GetSize();

    size_t uiOperationSize = uiLeftSize;
    if (uiLeftSize > uiRightSize)
    {
        uiOperationSize = uiRightSize;
    }

    for (size_t cntI = 0; cntI < uiOperationSize; cntI++)
    {
        m_deqPolynomial[cntI] ^= kref_polyRight[cntI];
    }
    return *this;
}

```

实现效果

```

master_key = 0x2b7e151628aed2a6abf7158809cf4f3c
plaintext = 0x3243f6a8885a308d313198a2e0370734
ciphertext = 0x3925841d02dc09fdbc118597196a0b32
encrypt: 0x3925841d02dc09fdbc118597196a0b32
decrypt: 0x3243f6a8885a308d313198a2e0370734

```

总结

基本实现了AES的基本加密算法的实现