

AES 加密算法详细分析

作者 Tomcoding

2019 年 4 月 29 日

简介

AES 是 The **A**dvanced **E**ncryption **S**tandard 的缩写，是美国国家标准与技术研究所用于加密电子数据的规范。AES 是新一代的加密算法，基于对称密钥，分组加密解密，密钥可以采用 128 位，192 位和 256 位长度，数据分组长度固定为 128 位。

基本原则

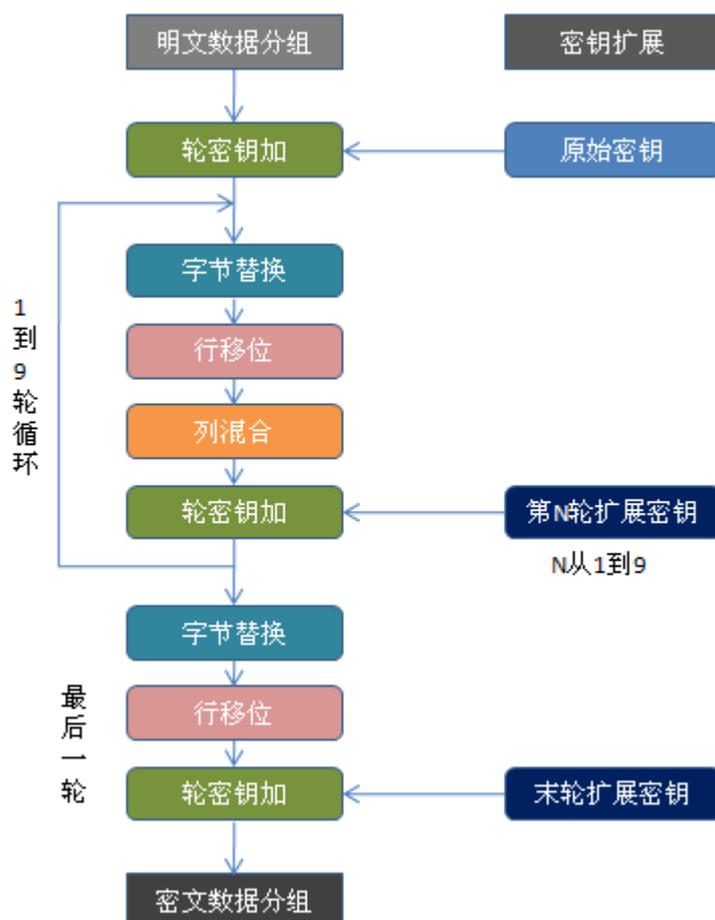
对称数据加密基本原则是混淆和扩散。最先想到是混淆，用来掩盖明文与密文之间的关系，最容易实现的方法就是替换，就是把一个符号用另一个符号代替，相当于有一个密码本，用密码本中的字符替换掉明文中的字符。扩散的原理是将明文中的单个数据尽可能多的影响密文中的多个数据，就是把明文中的一个小改变扩散到密文中去，造成密文较大的差别，比如明文中改变一位，产生的密文看起来与改变前产生的密文有很大不同，这个原理实现起来最容易的方法就是换位。在后面的详细分析中我们会看到这两种方法。

状态矩阵

用于加密或解密的数据分组都是 128 位，也就是 16 个字节（ $128/8=16$ ），这 16 字节的数据被放在一个 4x4 的矩阵中，叫做状态矩阵。所有的加密或解密中的各个步骤都是对这个状态矩阵的操作。状态矩阵的数据是按列存放的，例如数据以 16 进制表示为 0x4142434445464748494A4B4C4D4E4F50，那么在状态矩阵中存放的格式如下图：

41	45	49	4D
42	46	4A	4E
43	47	4B	4F
44	48	4C	50

加密算法描述



加密算法的流程很清晰，我们以 128 位密钥为基础来描述算法，其他 192 位和 256 位与之相似，我们后面再讨论。

首先把 128 位（16 字节）的数据分组放到 4x4 的状态矩阵（state matrix）中，然后对原始的 16 字节密钥进行扩展，128 位的加密要进行 10 轮循环，所以要扩展出 10 组密钥，加上原始密钥，一共需要 11 组密钥。每一个循环中执行四步操作，分别是字节替换，行移位，列混合（最后一轮中没有列混合），和轮密钥加，这四步操作我们在后面分析。每一个循环的轮密钥加用到扩展出的一组密钥，一一对应。进行完所有这些操作后，状态矩阵中的数据就是密文数据，再把矩阵中的数据放到一维数组中得到 16 字节的密文字节串。

字节替换

我们先看一下字节替换（Substitute Bytes），这就是对称数据加密中的混淆原则。字

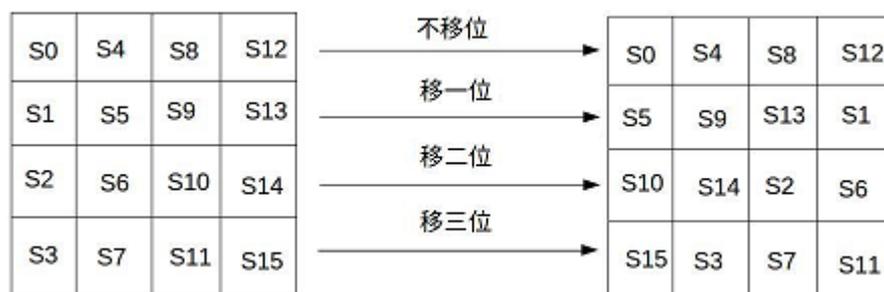
节替换的核心是一张表叫做 S-Box，一个字节查表后得到替换的另一个字节，这样就完成了替换操作，非常简单。S-Box 的定义如下，是一个 16x16 字节的二维表。

列 行																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

替换的方法是把一个字节的低四位作为行坐标，高四位作为列坐标，找到表中对应的数值作为替换字节。比如字节 0x42，对应的行坐标是 4，对应的列坐标是 2，那么查表得到的替换字节就是 0x2C。

这儿的字节替换在密钥扩展时还要用到。

行移位



行移位 (Shift Rows) 是对称加密算法中扩散原则的体现。在状态矩阵中，从零开始计算行数，那么第零行循环移位零次，就是不移位，第一行向左循环移位一个字节，第二行向左循环移位两个字节，第三行向左循环移位三个字节，如上图中所示。移位后得到了新的矩阵。

列混合

列混合 (Mix Columns) 也是扩散原则的体现。列混合是一个矩阵相乘的运算，状态矩阵是一个乘数，还有一个固定数值的常量矩阵，这两个矩阵相乘得到一个新的状态矩阵，这就是列混合。

下图中 S 是原始的数据，S' 是混合后的数据，最左边的矩阵是常量矩阵。

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} = \begin{bmatrix} s'_{00} & s'_{01} & s'_{02} & s'_{03} \\ s'_{10} & s'_{11} & s'_{12} & s'_{13} \\ s'_{20} & s'_{21} & s'_{22} & s'_{23} \\ s'_{30} & s'_{31} & s'_{32} & s'_{33} \end{bmatrix}$$

矩阵的乘法是常量矩阵的行与原始状态矩阵的列相乘，作为输出状态矩阵的列。输出状态矩阵中的第 j 列 ($0 \leq j \leq 3$) 的列混合可以用下图的公式表示。

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

在上面的公式中，加法和乘法不同于我们平时熟悉的加法和乘法，它是伽罗瓦域内的加法和乘法，我们不需要明白域论的复杂知识，只要知道怎样计算就可以。这里的加法等价于两个字节异或。乘法相对复杂一些，其实乘法可以把乘数分解成乘以 2 的次数，然后根据余数决定是否加上被乘数，举例说明一下，如果是 $8*5$ ，那么可以分解成 $8*(4+1)=(8*2*2)+8$ 。看一些复杂的组合。

$$a * 0x09 = a * (8 + 1) = (a * 2 * 2 * 2) + a$$

$$a * 0x0b = a * (8 + 2 + 1) = (a * 2 * 2 * 2) + (a * 2) + a$$

$$a * 0x0d = a * (8 + 4 + 1) = (a * 2 * 2 * 2) + (a * 2 * 2) + a$$

$$a * 0x0e = a * (8 + 4 + 2) = (a * 2 * 2 * 2) + (a * 2 * 2) + (a * 2)$$

实际上乘数是被分解成了 2 的幂相加，上面的加法应该是异或操作。

从上面看到，我们只需要计算出一个数乘以 2 的域乘积就可以组合成任意乘数的乘法结果。在计算机中我们知道一个数乘以 2，等于左移一位，低位补 0，但是有一个问题，这里

的被乘数都是一个字节，如果最高位是 1，就是被乘数大于等于 0x80，那么就溢出了，所以在域乘法中，如果最高位是 1，那么被乘数左移一位后要与 0x1B=00011011 进行异或操作。

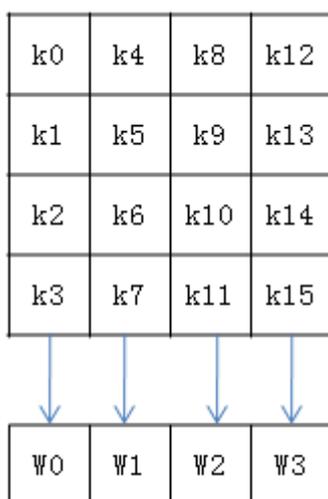
用 C 语言表示为：

```
n = (n & 0x80) ? ((n<<1) ^ 0x1B) : (n<<1);
```

其中 n 是被乘数。

密钥扩展

密钥扩展是 AES 算法中比较难理解的部分，我们尽量使它看起来简单些。我们以 128 位密钥为例，也就是 16 个字节，把它放到一个 4*4 的矩阵中，数据也是竖着排列的。由于在上面的加密算法中我们看到有 11 次轮密钥加，所以需要 11 组密钥，除去原始的密钥外，需要扩展出 10 组新的密钥。



原始的密钥是 k0 到 k15，共 16 个字节，放到矩阵中如上图所示，矩阵中每一列组成一个 4 字节的字（word），一组密钥由 4 个字组成。下面扩展第一组密钥，也是由 4 个字组成，关键是第一个字的生成，其他 3 个字的生成方法很简单。

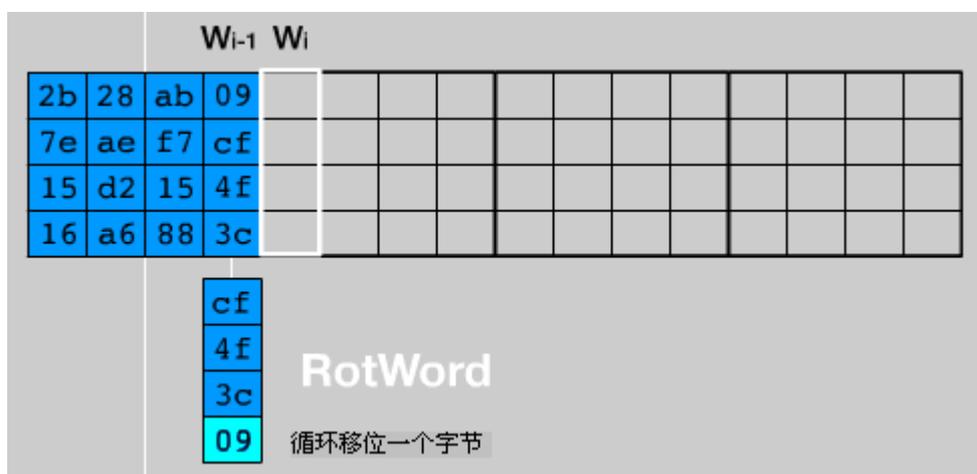
第一个字（W[4]）的生成有三个步骤：

1. 将 W[3] 向上循环移位一个字节，这里 $3=4-1$ （4 是密钥字的个数）
2. 生成的新字通过 S-box 进行字节替换，生成新的字
3. 上一步生成的新字与一个轮常量进行异或，然后再与 W[0] 异或，这里 $0=4-4$ 轮常量如下表所示，每个字中只有第一个字节有数字，其他的三个都是零。

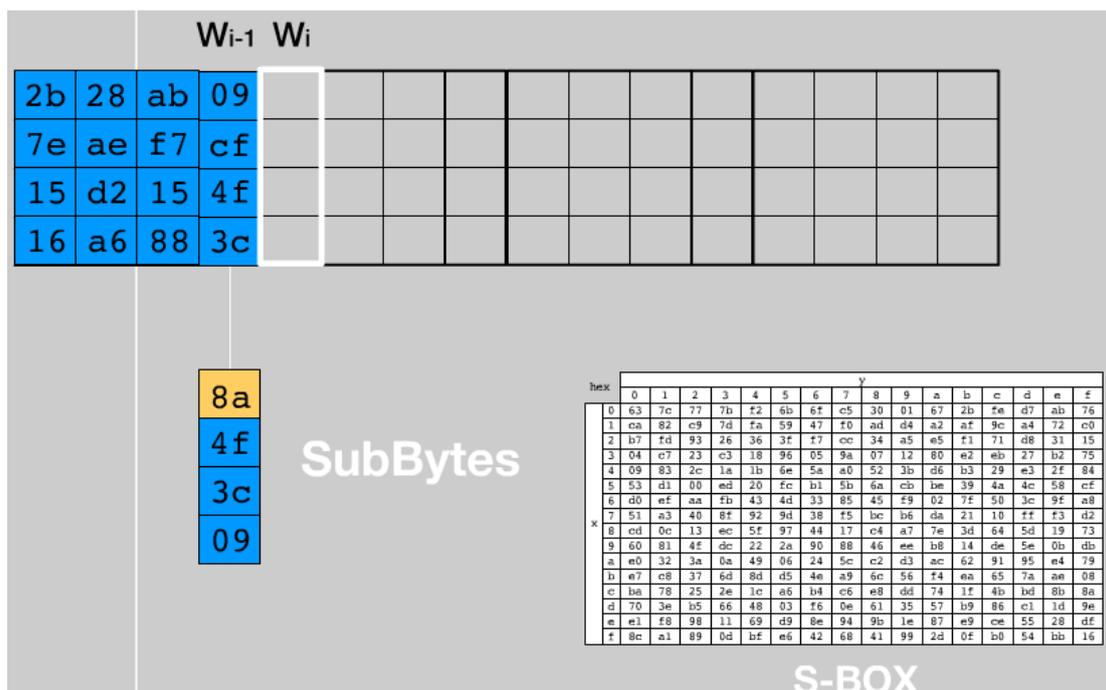
轮数	1	2	3	4	5	6	7	8	9	10
轮常量 RC[i]	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1B	0x36
	0x00									
	0x00									
	0x00									

网上有一个动画演示的 AES 算法，我们把其中的例子拿出来看一看，加深理解。

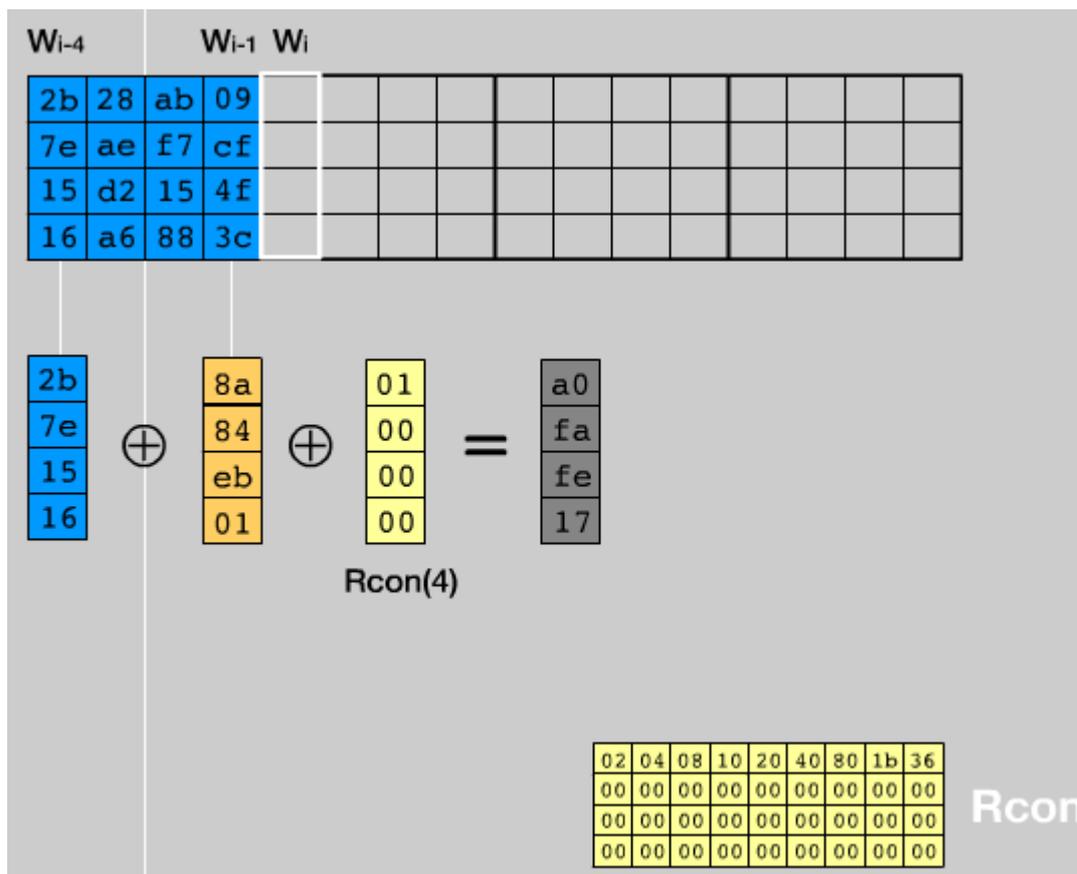
1. 字节移位。



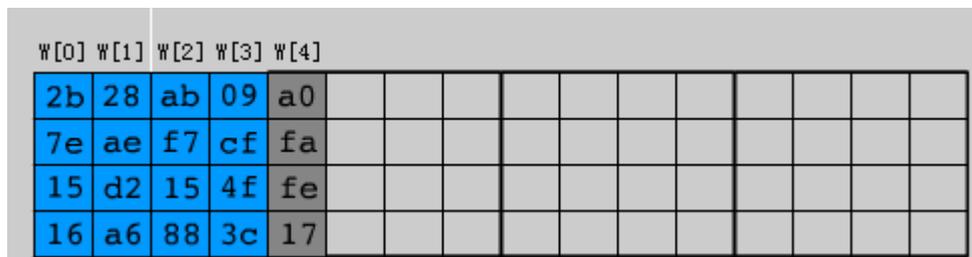
2. 字节替换。



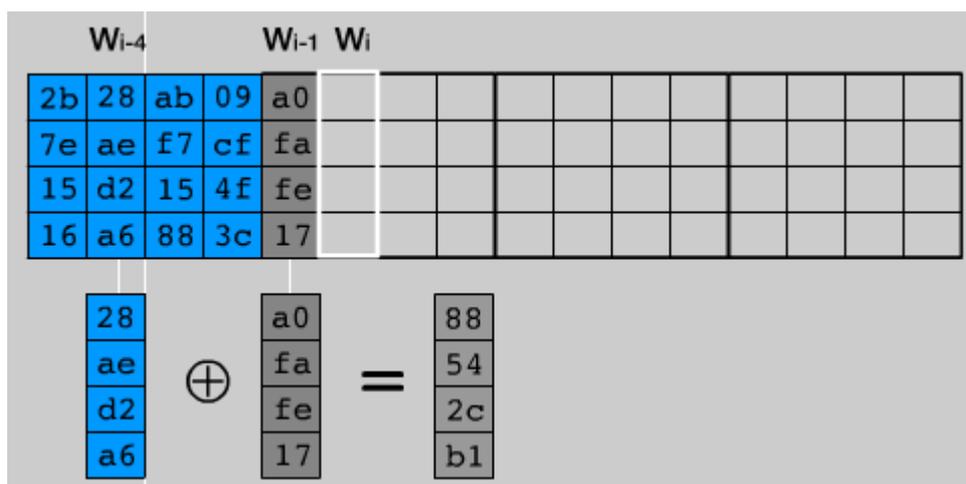
3. 轮常量异或。



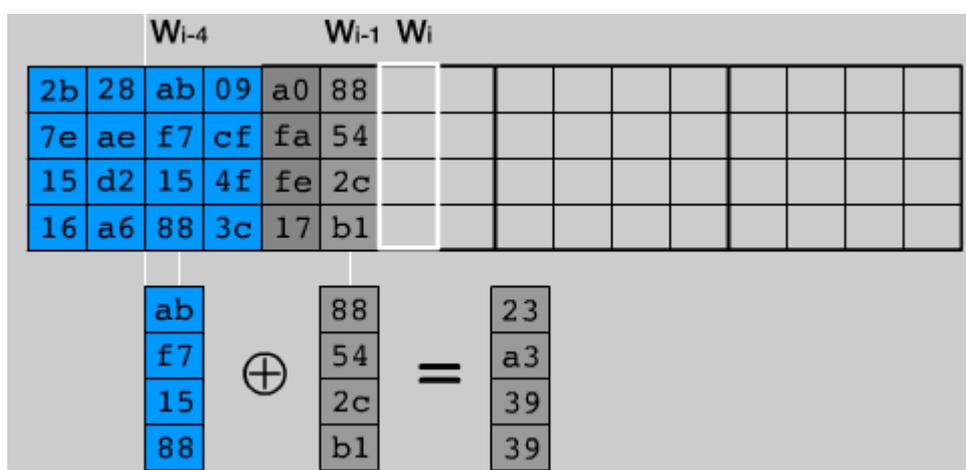
4. 得到第一组扩展密钥的第一个字。



扩展密钥第二个字生成很简单，就是刚生成的第一个字与原始密钥第二个字的异或。



扩展密钥的第三个字等于上面生成的第二个字与原始密钥 W[2] 的异或，剩下的两个字依次类推。



总结一下扩展密钥的生成规则。生成的密钥字下标用 i 来表示，那么 i 能被 4 整除的字就是第 $i/4$ 组密钥的第一个字，比如上面 $i=4$ ，那么 $W[4]$ 就是第 1 组 ($4/4=1$) 的第一个字 ($4\%4=0$)。所以可以用下面的公式来表示。

如果 $i\%4$ 等于 0:

$$M1 = \text{ShiftRows}(W[i-1])$$

$$M2 = \text{SubBytes}(M1)$$

$$M3 = M2 \wedge \text{RC}[i/4]$$

$$W[i] = W[i-4] \wedge M3$$

其中 $M1$, $M2$, $M3$ 是中间结果， RC 是轮常数， $W[i]$ 是扩展密钥的输出字，连起来就是:

$$W[i] = \text{SubBytes}(\text{ShiftRows}(W[i-1])) \wedge \text{RC}[i/4] \wedge W[i-4]$$

如果 $i\%4$ 不等于 0:

$$W[i]=W[i-1]^W[i-4]$$

轮密钥加

轮密钥加的算法很简单，状态矩阵是 4*4 的结构，每一轮的扩展密钥也是 4*4 的矩阵，对应的每个字节进行异或操作，产生的新字节作为状态矩阵的结果即可。

看下面的例子。

04	e0	48	28
66	cb	f8	06
81	19	d3	26
e5	9a	7a	4c

a0	88	23	2a
fa	54	a3	6c
fe	2c	39	76
17	b1	39	05

Round key

矩阵列异或。

e0	48	28
cb	f8	06
19	d3	26
9a	7a	4c

04
66
81
e5

 \oplus

a0
fa
fe
17

 $=$

a4
9c
7f
f2

88	23	2a
54	a3	6c
2c	39	76
b1	39	05

Round key

产生新的列。

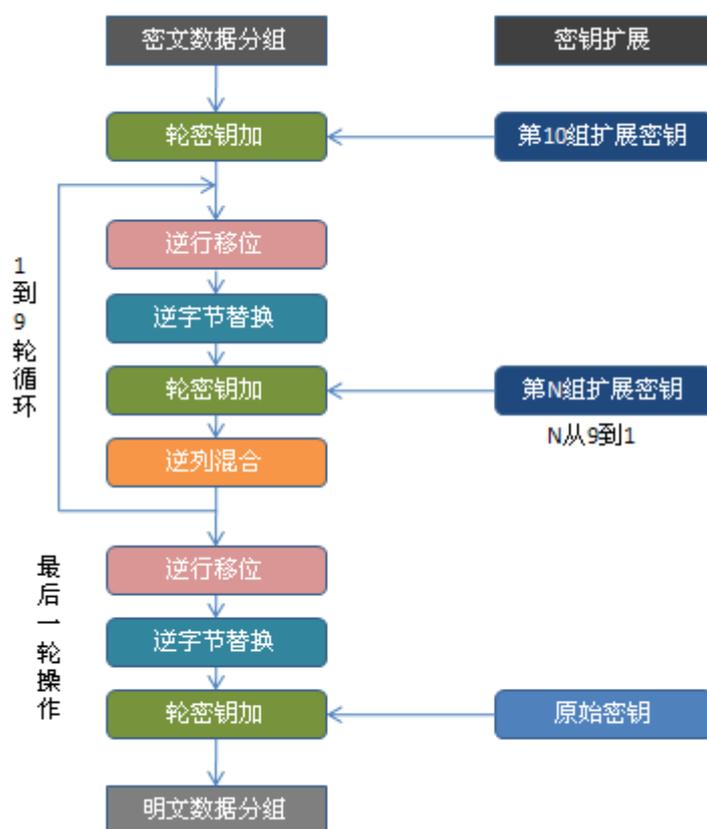
a4	e0	48	28
9c	cb	f8	06
7f	19	d3	26
f2	9a	7a	4c

88	23	2a
54	a3	6c
2c	39	76
b1	39	05

Round key

解密算法描述

解密算法是加密算法的逆运算,就是把加密过程倒过来执行一遍。还是设定 128 位密钥,我们也画个图来演示一下。



解密过程的密钥扩展与加密过程中的操作是一样的,产生出与加密相同的 11 组扩展密钥。中间执行 9 轮相同的循环,不过与加密过程不同,执行的四种操作分别是逆行移位,逆字节替换,轮密钥加和逆列混合,只有轮密钥加与加密过程是同一个操作,其他的都是加密过程的逆操作。

逆字节替换

逆字节替换 (Inverse Substitute Bytes) 与字节替换是一样的算法,只不过替换的表数据改变了,叫做 Inverse S-Box, 定义如下。

列 行	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

逆行移位

逆行移位就是与行移位进行相反的操作，在状态矩阵中，还是从 0 行开始，第 0 行不移位，第 1 行循环右移一个字节，第 2 行循环右移两个字节，第 3 行循环右移三个字节。注意这里是右移，与行移位的左移正好相反。

逆列混合

逆列混合与列混合的算法一样，只是常量矩阵变了，定义如下。

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

192 位密钥

192 位密钥正好是 24 个字节，与 128 位密钥的算法是一样的，只不过在加解密的过程中要进行 12 轮的循环，所以密钥扩展时也要生成 12 组扩展密钥。生成扩展密钥时轮常数要

往后扩展，第一个字节如下。

0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36,
0x6C, 0xD8, 0xAB, 0x4D

256 位密钥

256 位密钥算法大致与 128 位相同，加解密的轮数增加到了 14 轮，密钥扩展也增加到 14 组，轮常数在上一节 192 位密钥中已经给出了。与 128 位密钥扩展的区别在于除了在密钥字是 8 的整数倍，也就是每组密钥的开始字要特殊处理外，每一组内在 4 的整数倍的字也要特殊处理，这个处理相对简单，只是用 `s_box` 替换了一下 `W[i-1]` 的字而已。

程序设计

在 AES 算法中，数据在状态矩阵中是按列排放的，在 C 语言处理时，我们的数据是在一个一维数组中存放，这样就要用一维数组来表示二维的矩阵，在程序编写时需要把矩阵元素的位置映射到一维数组中的下标，也就是找到数组中的位置。

字节替换函数

字节替换比较简单，一共 16 字节的数据，每个字节的高 4 位对应 S-box 中的行，低 4 位对应列，在 S-box 中得到一个值，用这个值替换原来的字节即可。

```
static uint8_t 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 */
};

```

```

static void substitute_bytes(uint8_t *state)
{
    int i;          /* state 矩阵中元素计数 */
    int row, col;   /* s_box 中行和列变量 */

    /* state 矩阵中的 16 个字节，逐个替换 */
    for (i=0; i<16; i++)
    {
        row = state[i]>>4; /* 行=高 4 位 */
        col = state[i]&0x0F; /* 列=低 4 位 */

        state[i] = s_box[row][col]; /* s_box 中的值替换原始值 */
    }
}

```

如果我们把 s_box 定义成一维数组 s_box[256]，那么就要计算要替换的字节在 s_box 中的位置，s_box 每一行的长度是 16，所以 state[i]=s_box[row*16+col];

```

static uint8_t s_box[256] =
{
    /* 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 */
};
```

```
static void substitute_bytes(uint8_t *state)
{
    int i;          /* state 矩阵中元素计数 */
    int row, col;   /* s_box 中行和列变量 */

    /* state 矩阵中的 16 个字节，逐个替换 */
    for (i=0; i<16; i++)
    {
        row = state[i]>>4;    /* 行=高 4 位 */
        col = state[i]&0x0F;  /* 列=低 4 位 */

        state[i] = s_box[row * 16 + col]; /* s_box 中的值替换原始值 */
    }
}
```

从上面的函数中我们看到， $row = state[i] \gg 4$ ，在 $s_box[row * 16 + col]$ 中， row 又乘以 16，其实右移 4 位等于除以 16，余数就是 $state[i] \& 0x0F$ ，所以一个字节的数值在经过上述的变化后其实还是原来的值，比如数字 $n = n \gg 4$ (行) + $n \& 0x0F$ (列)， $row * 16$ 相当于左移了 4 位，其实还是还原成了高 4 位。举个例子，一个字节 0x63，高 4 位是 6，低 4 位是 3，那么 $row * 16 + col = 99 = 0x63$ ，跟原来的值一样。所以最终我们的字节替换函数就是下面的样子。

```
static void substitute_bytes(uint8_t *state)
{
    int i;          /* state 矩阵中元素计数 */

    for (i=0; i<16; i++)
    {
        state[i] = s_box[state[i]];
    }
}
```

行移位函数

行移位函数看起来很简单，状态矩阵的第几行就循环左移几个字节，但是如果用一维数组来表示矩阵，数组的下标看起来就不那么清晰了，我们先理顺这里的关系。我们把数组的下标排成一个矩阵的形式。

`state[16]`

0 4 8 c

1 5 9 d

2 6 a e

3 7 b f

如果用 i 和 j 来为行和列进行计数，都从 0 开始，我们看一下每一行要移位的下标要怎样计算，第零行不需要移位，第一行是 1, 5, 9, $d = 0*4+0$, $1*4+1$, $2*4+1$, $3*4+1$ ，按照这个方法也排成一个矩阵形式。

$0*4+0$ $1*4+0$ $2*4+0$ $3*4+0$

$0*4+1$ $1*4+1$ $2*4+1$ $3*4+1$

$0*4+2$ $1*4+2$ $2*4+2$ $3*4+2$

$0*4+3$ $1*4+3$ $2*4+3$ $3*4+3$

看一下每一个元素的下标似乎可以表示成 $i*4+j$ ，这里又有一个问题，由于是 4×4 的矩阵，所以 i 和 j 哪个是行哪个是列我们其实并没搞清楚，但这不影响编写程序。看一个 3×4 的矩阵。

0 3 6 9

1 4 7 a

2 5 8 b

```
0*3+0  1*3+0  2*3+0  3*3+0
0*3+1  1*3+1  2*3+1  3*3+1
0*3+2  1*3+2  2*3+2  3*3+2
```

这下就能看清楚了，这里下标= $j*3+i$ ， i 表示行， j 表示列，3是行的总个数。

编写一个移位函数。

```
static void shift_rows(uint8_t *state)
{
    int    i, row; /* i 是移位计数器，row 是移位的行，从 0 开始计算 */

    for (row=1; row<4; row++) /* 第零行不移位，行计数从 1 开始 */
    {
        for (i=0; i<row; i++)
        {
            shift_one_byte(state);
        }
    }
}

static void shift_one_byte(uint8_t *state, int row)
{
    uint8_t  temp_byte;

    temp_byte    = state[0*4+row];
    state[0*4+row] = state[1*4+row];
    state[1*4+row] = state[2*4+row];
    state[2*4+row] = state[3*4+row];
    state[3*4+row] = temp_byte;
}
```

上面的公式很简单，为了执行效率，我们把里面的常数算好，把两个函数合并，函数如下。

```
static void shift_rows(uint8_t *state)
{
    int    i, row; /* i 是移位计数器，row 是移位的行，从 0 开始计算 */
    uint8_t  temp_byte;

    for (row=1; row<4; row++) /* 第零行不移位，行计数从 1 开始 */
    {
```

```
for (i=0; i<row; i++)
{
    temp_byte    = state[row];
    state[row]   = state[4+row];
    state[4+row] = state[8+row];
    state[8+row] = state[12+row];
    state[12+row] = temp_byte;
}
}
```

列混合函数

列混合的操作是状态矩阵的列与常量矩阵的行相乘。

02	03	01	01	S[0, 0]	S[0, 1]	S[0, 2]	S[0, 3]
01	02	03	01	S[1, 0]	S[1, 1]	S[1, 2]	S[1, 3]
01	01	02	03	S[2, 0]	S[2, 1]	S[2, 2]	S[2, 3]
03	01	01	02	S[3, 0]	S[3, 1]	S[3, 2]	S[3, 3]

S[0, 0]等于状态矩阵的第零列与常量矩阵的第零行相乘

S[1, 0]等于状态矩阵的第零列与常量矩阵的第一行相乘

S[2, 0]等于状态矩阵的第零列与常量矩阵的第二行相乘

S[3, 0]等于状态矩阵的第零列与常量矩阵的第三行相乘

S[0, 1]等于状态矩阵的第一列与常量矩阵的第零行相乘

S[1, 1]等于状态矩阵的第一列与常量矩阵的第一行相乘

S[2, 1]等于状态矩阵的第一列与常量矩阵的第二行相乘

S[3, 1]等于状态矩阵的第一列与常量矩阵的第三行相乘

S[0, 2]等于状态矩阵的第二列与常量矩阵的第零行相乘

S[1, 2]等于状态矩阵的第二列与常量矩阵的第一行相乘

S[2, 2]等于状态矩阵的第二列与常量矩阵的第二行相乘

S[3, 2]等于状态矩阵的第二列与常量矩阵的第三行相乘

$S[0, 3]$ 等于状态矩阵的第三列与常量矩阵的第三行相乘

$S[1, 3]$ 等于状态矩阵的第三列与常量矩阵的第二行相乘

$S[2, 3]$ 等于状态矩阵的第三列与常量矩阵的第一行相乘

$S[3, 3]$ 等于状态矩阵的第三列与常量矩阵的第三行相乘

这是我们前面总结的公式，其中 $0 \leq j \leq 3$

$$S'[0, j] = (2 * S[0, j]) \wedge (3 * S[1, j]) \wedge S[2, j] \wedge S[3, j]$$

$$S'[1, j] = S[0, j] \wedge (2 * S[1, j]) \wedge (3 * S[2, j]) \wedge S[3, j]$$

$$S'[2, j] = S[0, j] \wedge S[1, j] \wedge (2 * S[2, j]) \wedge (3 * S[3, j])$$

$$S'[3, j] = (3 * S[0, j]) \wedge S[1, j] \wedge S[2, j] \wedge (2 * S[3, j])$$

从上面看到，如果 i 表示行， j 表示列，每个 $S[i, j]$ 在一维数组中的下标值等于 $4*j+i$ ，所以 $S'[i, j]=S[i, j]=state[4*j+i]$ 。

下面是列混合函数的实现。

```
static void mix_columns(uint8_t *state)
{
    int i;          /* i 是列计数器 */
    uint8_t u[4];  /* state 中的元素会变化，需要中间存储 */

    for (i=0; i<4; i++)
    {
        u[0] = gf_mult(state[4*i+0], 2) ^ gf_mult(state[4*i+1], 3)
            ^ state[4*i+2] ^ state[4*i+3];
        u[1] = state[4*i+0] ^ gf_mult(state[4*i+1], 2)
            ^ gf_mult(state[4*i+2], 3) ^ state[4*i+3];
        u[2] = state[4*i+0] ^ state[4*i+1]
            ^ gf_mult(state[4*i+2], 2) ^ gf_mult(state[4*i+3], 3);
        u[3] = gf_mult(state[4*i+0], 3) ^ state[4*i+1]
            ^ state[4*i+2] ^ gf_mult(state[4*i+3], 2);

        state[4*i+0] = u[0];
        state[4*i+1] = u[1];
        state[4*i+2] = u[2];
        state[4*i+3] = u[3];
    }
}
```

上面的函数 `gf_mult()` 是伽罗瓦域乘法的实现函数，在描述时可能比较简单，把乘数按照 2 的幂进行分解后再组合就可以了，但实现起来还是很有难度，我们通过例子来看一下。如果乘数是 0x0D，分解后如下。

$$a * 0x0D = a * (8 + 0 + 2 + 1) = (a * 2 * 2 * 2) + (a * 2) + a$$

0x0D 的二进制是 0000 1011，我们看到第四位的每一位正好对应着 8, 0, 2, 1, 8 是 2 的 3 次幂，2 是 2 的 1 次幂。我们可以设置一个数组来存放乘数字节中的每一位。定义一维数组 `uint8_t p[8]`，如果乘数是 b，那么 b 的第 0 位存放在 `p[0]`，第 1 位存放在 `p[1]`……，第 7 位存放在 `p[7]`。还是以 0x0D 作为乘数，那么 `p[0]` 到 `p[7]` 的值如下。

```
p[0] = 1
p[1] = 1
p[2] = 0
p[3] = 1
p[4] = 0
p[5] = 0
p[6] = 0
p[7] = 0
```

从上面我们看到数组的下标正好就是 2 的幂指数，也就是分解后乘以 2 的次数。那么我们的程序就可以写成下面这样。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int          i, j;    /* i, j=循环计数器, i 是数组下标, 也是乘以 2 的次数 */
    uint8_t      sign;   /* sign=被乘数高位 */
    uint8_t      a[8];

    if (u2 & 0x01)
        a[0] = 0x01;
    else
        a[0] = 0x00;

    if (u2 & 0x02)
        a[1] = 0x01;
    else
        a[1] = 0x00;

    if (u2 & 0x04)
        a[2] = 0x01;
    else
        a[2] = 0x00;
```

```
if (u2 & 0x08)
    a[3] = 0x01;
else
    a[3] = 0x00;

if (u2 & 0x10)
    a[4] = 0x01;
else
    a[4] = 0x00;

if (u2 & 0x20)
    a[5] = 0x01;
else
    a[5] = 0x00;

if (u2 & 0x40)
    a[6] = 0x01;
else
    a[6] = 0x00;

if (u2 & 0x80)
    a[7] = 0x01;
else
    a[7] = 0x00;

/* 把被乘数乘以 2 后的数值存放在数组相应的位置
 * 比如  $u1 * 0x0D = (u1 * 2 * 2 * 2) + (u1 * 2) + u1$ 
 *  $(u1 * 2 * 2 * 2)$  存放在 a[3] 中
 *  $(u1 * 2)$  存放在 a[1] 中
 *  $(u1)$  存放在 a[0] 中
 */
for (i=0; i<8; i++)
{
    if (a[i])
    {
        for (a[i]=u1, j=0; j<i; j++)
        {
            sign = a[i] & 0x80;
            a[i] <<= 1;
            if (sign)
                a[i] ^= 0x1B;
        }
    }
}
```

```
/* 最后做加法操作 */
for (u1=0, i=0; i<8; i++)
    u1 ^= a[i];

return (u1);
}
```

上面的程序看起来很臃肿，简化一下。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int          i, j;
    uint8_t      a[8];

    a[0] = (u2 & 0x01) ? 0x01 : 0x00;
    a[1] = (u2 & 0x02) ? 0x01 : 0x00;
    a[2] = (u2 & 0x04) ? 0x01 : 0x00;
    a[3] = (u2 & 0x08) ? 0x01 : 0x00;
    a[4] = (u2 & 0x10) ? 0x01 : 0x00;
    a[5] = (u2 & 0x20) ? 0x01 : 0x00;
    a[6] = (u2 & 0x40) ? 0x01 : 0x00;
    a[7] = (u2 & 0x80) ? 0x01 : 0x00;

    for (i=0; i<8; i++)
    {
        if (a[i])
        {
            for (a[i]=u1, j=0; j<i; j++)
            {
                a[i] = (a[i] & 0x80) ? ((a[i]<<1) ^ 0x1B) : (a[i]<<1);
            }
        }
    }

    for (u1=0, i=0; i<8; i++)
        u1 ^= a[i];

    return (u1);
}
```

看起来简洁了许多，但还是感觉人干了计算机该干的活。其实仔细分析一下，乘数每往右移位一次，如果低位为 1，就设置数组相应的元素，与我们上面设置数组的方式是一样的。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int          i, j;
```

```
uint8_t      a[8];

for (i=0; i<8; i++)
{
    a[i] = (u2 & 0x01) ? 0x01 : 0x00;
    u2 >>= 1;
}

for (i=0; i<8; i++)
{
    if (a[i])
    {
        for (a[i]=u1, j=0; j<i; j++)
        {
            a[i] = (a[i] & 0x80) ? ((a[i]<<1) ^ 0x1B) : (a[i]<<1);
        }
    }
}

for (u1=0, i=0; i<8; i++)
    u1 ^= a[i];

return (u1);
}
```

可以把上面函数中开始的两个循环合并到一个循环中。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int      i, j;
    uint8_t  a[8];

    for (i=0; i<8; i++)
    {
        a[i] = (u2 & 0x01) ? 0x01 : 0x00;
        u2 >>= 1;

        if (a[i])
        {
            for (a[i]=u1, j=0; j<i; j++)
            {
                a[i] = (a[i] & 0x80) ? ((a[i]<<1) ^ 0x1B) : (a[i]<<1);
            }
        }
    }
}
```

```
for (u1=0, i=0; i<8; i++)
    u1 ^= a[i];

return (u1);
}
```

再把循环合并一次，这次 u1 在上个循环中被使用了，设置一个新的临时变量 b 保存结果。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int          i, j;
    uint8_t      a[8], b;

    for (b=0, i=0; i<8; i++)
    {
        a[i] = (u2 & 0x01) ? 0x01 : 0x00;
        u2 >>= 1;

        if (a[i])
        {
            for (a[i]=u1, j=0; j<i; j++)
            {
                a[i] = (a[i] & 0x80) ? ((a[i]<<1) ^ 0x1B) : (a[i]<<1);
            }

            b ^= a[i];
        }
    }

    return (b);
}
```

从上面看到其实数组 a[i] 也是用来存放中间结果的，可以把它去掉，把 a[i] 乘以 2 的操作从判断语句 if (a[i]) 中移出来，每次循环都乘以 2，当 u2 移位后最后一位为 1 时，乘积就是这一位对应的数值，累加起来就是最后结果。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int          i;
    uint8_t      b;

    for (b=0, i=0; i<8; i++)
    {
```

```
    if (u2 & 0x01)
    {
        b ^= u1;
    }

    u1 = (u1 & 0x80) ? ((u1<<1) ^ 0x1B) : (u1<<1);

    u2 >>= 1;
}

return (b);
}
```

其实不用都循环 8 次，如果 u2 高位为 0 时，循环就没有意义了，再修改一下作为最终结果。

```
static uint8_t gf_mult(uint8_t u1, uint8_t u2)
{
    int i; /* i=循环计数器 */
    uint8_t b; /* 存放中间结果的变量 */

    for (b=0, i=0; u2&&(i<8); i++)
    {
        if (u2 & 0x01)
        {
            b ^= u1;
        }

        u1 = (u1 & 0x80) ? ((u1<<1) ^ 0x1B) : (u1<<1);

        u2 >>= 1;
    }

    return (b);
}
```

密钥扩展函数

密钥扩展后最少需要 44 个字，176 个字节，定义一个一维数组存放原始密钥和扩展密钥 `ekey[256]`。前 16 字节存放原始密钥。在 AES 中密钥也表示成 4x4 的矩阵，按列来处理，我们要换算成一维数组的下标来计算。

11 个密钥，每个密钥 4 个字，每个字 4 个字节

在一维数组 `ekey[256]` 中，各个密钥，字的下标地址

每个密钥的开始下标是 $4*(i/4)*4$ (i 大于等于 4，小于等于 44， i 是密钥的字计数)

密钥每个字的下标是 $4*i$

密钥每个字第 1 个字节的下标是 $4*i + 0$

密钥每个字第 2 个字节的下标是 $4*i + 1$

密钥每个字第 3 个字节的下标是 $4*i + 2$

密钥每个字第 4 个字节的下标是 $4*i + 3$

```
static uint8_t rcon[11] = {0x00, 0x01, 0x02, 0x04, 0x08,
    0x10, 0x20, 0x40, 0x80, 0x1B, 0x36};
```

```
static void key_expansion(uint8_t *key, uint8_t *ekey)
{
    int      i, j; /* 密钥字计数器 */
    uint8_t  m[4]; /* 中间变量 */
    uint8_t  u;    /* 临时变量 */

    /* 把原始密钥 key 放到扩展密钥数组的开始位置 */
    for (i=0; i<16; i++)
        ekey[i] = key[i];

    for (i=4; i<44; i++)
    {
        if (i%4 == 0)
        {
            /* 扩展密钥的第一个字 */
            /* 把要计算的密钥字放入中间变量 */
            for (j=0; j<4; j++)
                m[j] = ekey[(i-1)*4+j];

            /* 循环左移一个字节 */
            u = m[0];
            m[0] = m[1];
            m[1] = m[2];
            m[2] = m[3];
            m[3] = u;

            /* 字节替换 */
            for (j=0; j<4; j++)
```

```

        m[j] = s_box[m[j]];

    /* 轮常量异或 */
    m[0] ^= rcon[i/4];

    /* 与 W[i-4]异或 */
    for (j=0; j<4; j++)
        ekey[i*4+j] = m[j] ^ ekey[(i-4)*4+j];
    }
    else
    {
        /* 扩展密钥的其他三个字
        * 每个字节在一维数组中的下标分别是 i*4+[0, 1, 2, 3]
        */
        for (j=0; j<4; j++)
        {
            ekey[i*4+j] = ekey[(i-1)*4+j] ^ ekey[(i-4)*4+j];
        } /* 密钥字的 4 个字节结束循环 */
    }
} /* 44 个密钥字循环结束 */
}

```

轮密钥加函数

我们在前面已经把状态矩阵和扩展密钥都表示成了一维数组，所以轮密钥加变得非常简单，只有计算出轮密钥在一维数组中的首地址，与状态矩阵中的元素连续异或 16 个字节，就形成了新的状态矩阵。

```

/*
 * state - 状态矩阵
 * ekey - 扩展密钥
 * r - 轮数
 */
static void add_round_key(uint8_t *state, uint8_t *ekey, int r)
{
    int i, j; /* i=轮密钥在扩展密钥数组中的下标, j=异或的字节计数 */

    for (i=r*16, j=0; j<16; j++)
    {
        state[j] ^= ekey[i+j];
    }
}

```

加密算法函数

加密函数的过程我们在前面描述过了，就是把上面写的函数组合起来，循环成需要的轮数。

```
static void aes_encrypt(uint8_t *state, uint8_t *ekey)
{
    int i;      /* 轮循环计数器 */

    add_round_key(state, ekey, 0);

    for (i=1; i<11; i++)
    {
        substitute_bytes(state);
        shift_rows(state);
        if (i < 10)
            mix_columns(state);
        add_round_key(state, ekey, i);
    }
}
```

解密算法函数

把解密算法描述中的函数组合起来，解密算法函数如下。

```
static void aes_decrypt(uint8_t *state, uint8_t *ekey)
{
    int i;      /* loop counter */

    add_round_key(state, ekey, 10);

    for (i=9; i>0; i--)
    {
        inv_shift_rows(state);
        inv_substitute_bytes(state);
        add_round_key(state, ekey, i);
        inv_mix_columns(state);
    }

    inv_shift_rows(state);
}
```

```
    inv_substitute_bytes(state);
    add_round_key(state, ekey, 0);
}
```

192 位密钥的程序设计

与 128 位密钥算法一样，只是扩展密钥轮数和加解密的循环轮数都增加到了 12，增加一个全局变量 nr 表示循环的轮数，增加一个全局变量 nk 表示原始密钥字的个数，把上面的程序中与轮数相关的常数替换成 nr，与密钥字数相关的常量替换成 nk，程序就能既适应 128 位密钥又能适应 192 位密钥了。

256 位密钥的程序设计

256 位密钥的程序只需要在密钥扩展函数中把每组密钥内是 4 的整数倍的字进行字节替换就可以了。

为了方便记忆，增加一个初始化函数，用定义的常数去初始化 nr 和 nk，常数如下：

```
KEY_BITS_128
```

```
KEY_BITS_192
```

```
KEY_BITS_256
```

这样就不用去记忆密钥长度对应的轮数和密钥的字数了。

在这里把上面修改过 nr 和 nk，以及 256 位密钥扩展特殊处理的函数都列出来。

密钥扩展函数：

```
void key_expansion(uint8_t *key, uint8_t *ekey)
{
    int          i, j, n;          /* 计数器 */
    uint8_t      m[4];            /* 临时变量数组 */
    uint8_t      u;               /* t 临时变量 */

    /* 把原始密钥放到扩展密钥数组开始的位置 */
    for (j=4*nk, i=0; i<j; i++)
        ekey[i] = key[i];

    /* 生成扩展密钥的循环 */
    for (n=4*(nr+1), i=nk; i<n; i++)
```

```
{
  if (i%nk == 0)
  {
    /* 生成一组扩展密钥的第一个字，4 个字节，放到中间数组中 */
    for (j=0; j<4; j++)
      m[j] = ekey[(i-1)*4+j];

    /* 字节移位 */
    u = m[0];
    m[0] = m[1];
    m[1] = m[2];
    m[2] = m[3];
    m[3] = u;

    /* 字节替换 */
    for (j=0; j<4; j++)
      m[j] = s_box[m[j]];

    /* 异或轮常量 */
    m[0] ^= rcon[i/nk];

    /* 异或第(i-4)个字 W[i-4] */
    for (j=0; j<4; j++)
      ekey[i*4+j] = m[j] ^ ekey[(i-4)*4+j];
  }
  else if ((nk == 8) && (i%nk == 4))
  {
    /* 256 位密钥，4 的整数倍的字进行字节替换 W[i-1] */
    for (j=0; j<4; j++)
    {
      ekey[(i-1)*4+j] = s_box[ekey[(i-1)*4+j]];
      ekey[i*4+j] = ekey[(i-1)*4+j] ^ ekey[(i-4)*4+j];
    }
  }
  else
  {
    /* 每组中其他的密钥字处理 */
    for (j=0; j<4; j++)
    {
      ekey[i*4+j] = ekey[(i-1)*4+j] ^ ekey[(i-4)*4+j];
    }
  }
} /* 所有扩展密钥生成结束 */
}
```

加密函数:

```
void aes_encrypt(uint8_t *state, uint8_t *ekey)
{
    int i;      /* 轮循环计数器 */

    add_round_key(state, ekey, 0);

    for (i=1; i<nr; i++)
    {
        substitute_bytes(state);
        shift_rows(state);
        mix_columns(state);
        add_round_key(state, ekey, i);
    }

    substitute_bytes(state);
    shift_rows(state);
    add_round_key(state, ekey, nr);
}
```

解密函数:

```
void aes_decrypt(uint8_t *state, uint8_t *ekey)
{
    int i;      /* 轮循环计数器 */

    add_round_key(state, ekey, nr);

    for (i=nr-1; i>0; i--)
    {
        inv_shift_rows(state);
        inv_substitute_bytes(state);
        add_round_key(state, ekey, i);
        inv_mix_columns(state);
    }

    inv_shift_rows(state);
    inv_substitute_bytes(state);
    add_round_key(state, ekey, 0);
}
```