密码学实验二 AES 加密算法

08111505 1120152056 李世林

1 算法原理

1.1 简介

AES 算法是一个对称分组密码算法。数据分组长度必须是 128 bits，使用的密钥长度为 128，192 或 256 bits。对于三种不同密钥长度的 AES 算法，分别称为“AES-128”、“AES-192”、“AES-256”。本实验使用 AES-128，

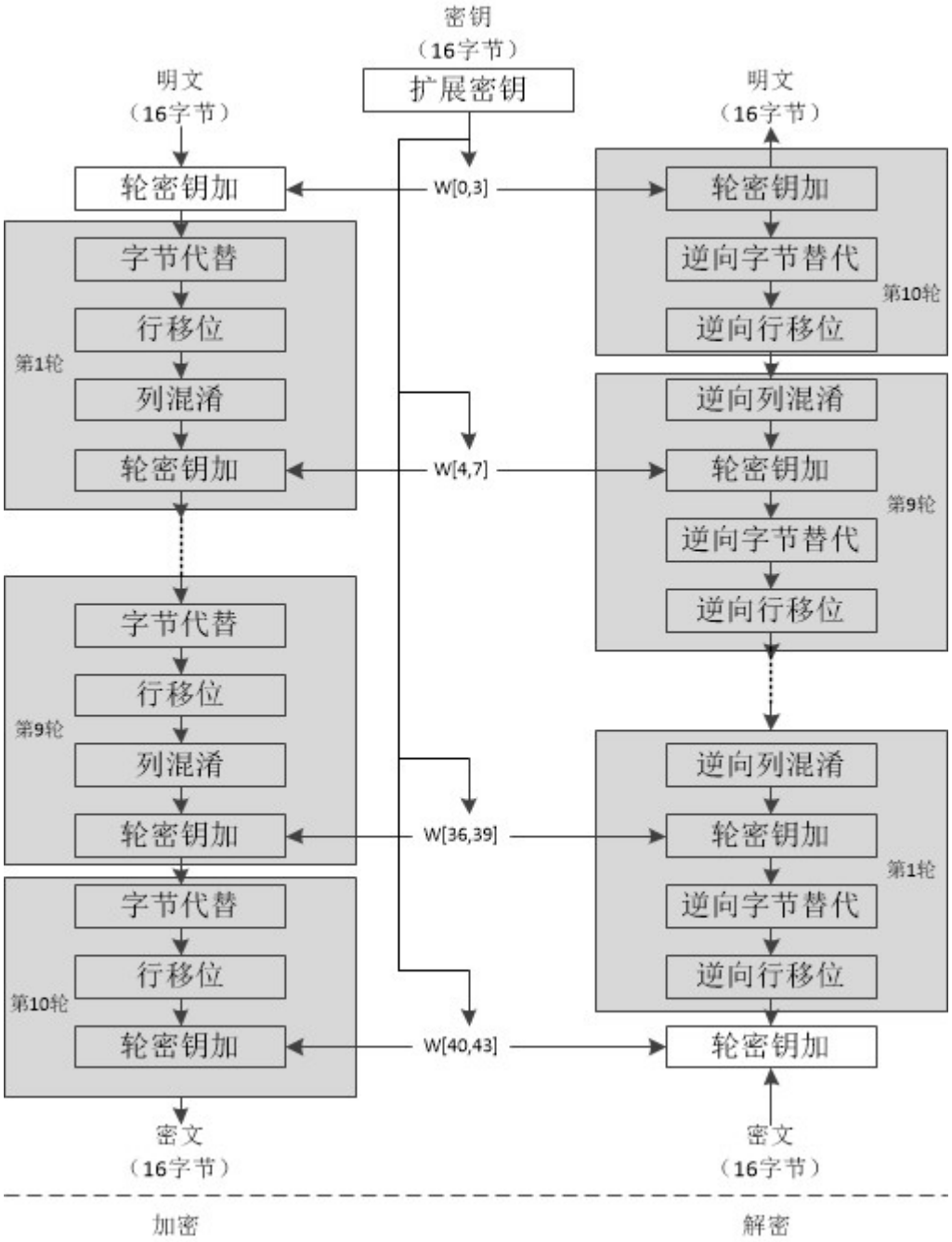
1.2AES 加密

AES 加密过程涉及到 4 种操作：字节替代（SubBytes）、行移位（ShiftRows）、列混淆（MixColumns）和轮密钥加（AddRoundKey）。

1.3AES 解密

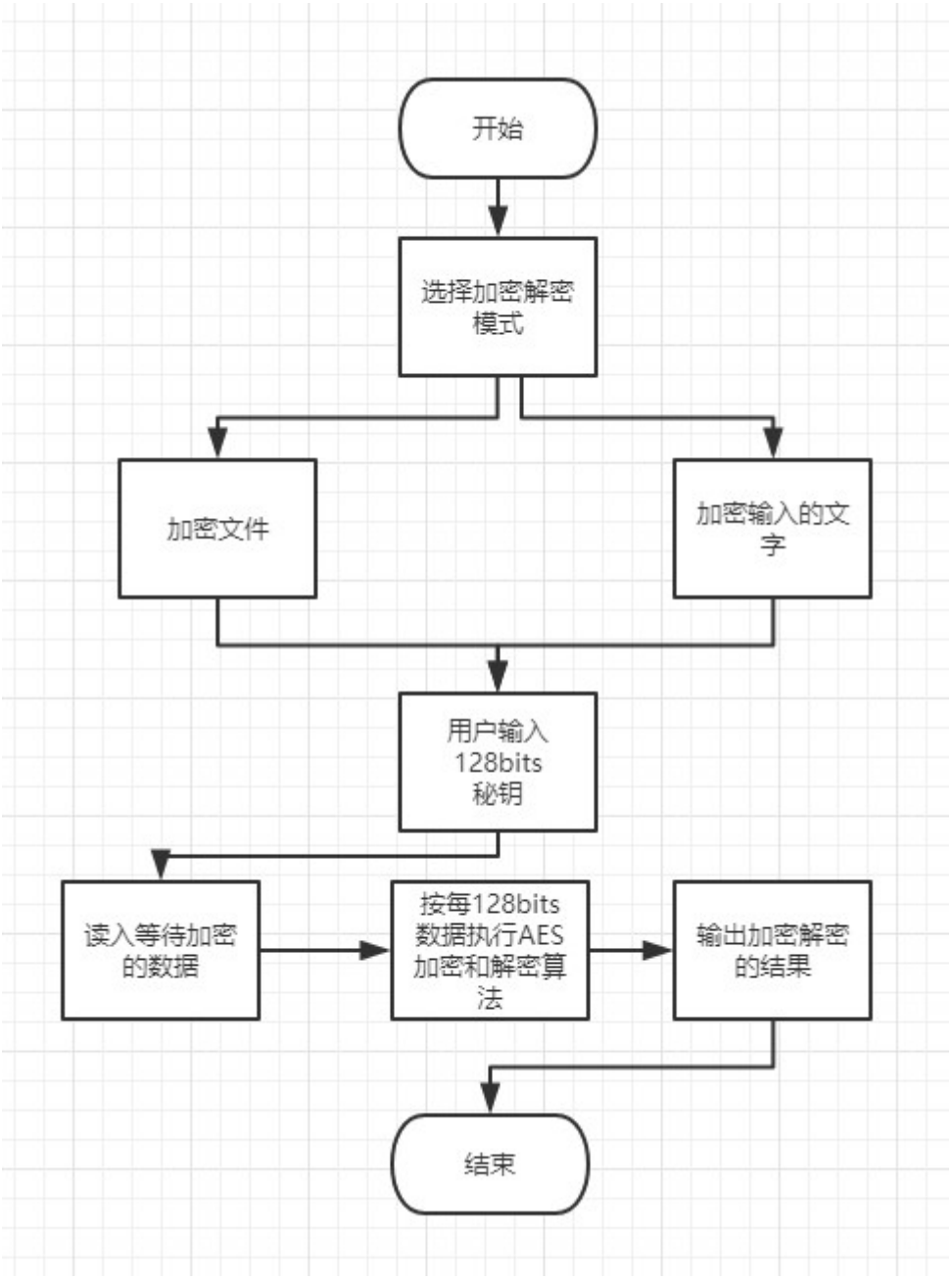
解密过程分别为对应加密过程的逆操作。由于每一步操作都是可逆的，按照相反的顺序进行解密即可恢复明文。加解密中每轮的密钥分别由初始密钥扩展得到。算法中 16 字节的明文、密文和轮密钥都以一个 4x4 的矩阵表示。

1.4 加解密流程图


密钥
（16字节）
扩展密钥
明文
（16字节）
轮密钥加
W[0,3]
字节代替
行移位
列混淆
轮密钥加
第1轮
W[4,7]
字节代替
行移位
列混淆
轮密钥加
第9轮
W[36,39]
字节代替
行移位
轮密钥加
第10轮
W[40,43]
密文
（16字节）
加密
明文
（16字节）
轮密钥加
逆向字节替代
逆向行移位
第10轮
逆向列混淆
轮密钥加
逆向字节替代
逆向行移位
第9轮
逆向列混淆
轮密钥加
逆向字节替代
逆向行移位
第1轮
轮密钥加
密文
（16字节）
解密

2C++代码实现 AES 加解密

2.1 程序流程图



2.2 AES 加解密详解

2.2.1 AES 秘钥扩展

AES 算法通过密钥扩展程序（Key Expansion）将用户输入的密钥 K 扩展生成 Nb(Nr+1)个字，存放在一个线性数组 w[Nb*(Nr+1)]中。

```
// 将 4 个 byte 转换为一个 word.
word Word(byte& k1, byte& k2, byte& k3, byte& k4)
{
   word result(0x00000000);
   word temp;
   temp = k1.to_ulong();  // K1
   temp <<= 24;
   result |= temp;
   temp = k2.to_ulong();  // K2
   temp <<= 16;
   result |= temp;
   temp = k3.to_ulong();  // K3
   temp <<= 8;
   result |= temp;
   temp = k4.to_ulong();  // K4
   result |= temp;
   return result;
```

```
}



//按字节 循环左移一位
//把[a0, a1, a2, a3]变成[a1, a2, a3, a0]

word RotWord(word& rw)
{
   word high = rw << 8;
   word low = rw >> 24;
   return high | low;
}

//对输入 word 中的每一个字节进行 S-盒变换
word SubWord(word& sw)
{
   word temp;
   for (int i = 0; i<32; i += 8)
   {
      int row = sw[i + 7] * 8 + sw[i + 6] * 4 + sw[i + 5]
* 2 + sw[i + 4];
      int col = sw[i + 3] * 8 + sw[i + 2] * 4 + sw[i + 1]
```

```
* 2 + sw[i];
      byte val = S_Box[row][col];
      for (int j = 0; j<8; ++j)
         temp[i + j] = val[j];
   }
   return temp;
}



// 密 钥 扩 展 函 数  -  对  128  位 密 钥 进 行 扩 展 得 到 w[4*(EncryptTimes+1)]


void KeyExpansion(byte key[4 * KeyWords], word w[4 * (EncryptTimes + 1)])
{
   word temp;
   int i = 0;
   // w[]的前 4 个就是输入的 key
   while (i < KeyWords)
   {
      w[i] = Word(key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i + 3]);
```

```
        ++i;
    }

    i = KeyWords;

    while (i < 4 * (EncryptTimes + 1))
    {
        temp = w[i - 1]; // 记录前一个 word
        if (i % KeyWords == 0)
            w[i] = w[i - KeyWords] ^ SubWord(RotWord(temp)) ^
Rcon[i / KeyWords - 1];
        else
            w[i] = w[i - KeyWords] ^ temp;
        ++i;
    }
}
```

2.2.2 S 盒变换-SubBytes()

S 盒是一个 16 行 16 列的表，表中每个元素都是一个字节。S 盒变换很简单：函数 SubBytes()接受一个 4x4 的字节矩阵作为输入，对其中的每个字节，前四位组成十六进制数 x 作为行号，后四位组成

的十六进制数 y 作为列号，查找表中对应的值替换原来位置上的字节.

```
byte S_Box[16][16] = {
   { 0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30,
0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76 },
   { 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD,
0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0 },
   { 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34,
0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15 },
   { 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07,
0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75 },
   { 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52,
0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84 },
   { 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A,
0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF },
   { 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45,
0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8 },
   { 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC,
0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2 },
   { 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4,
0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73 },
   { 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46,
```

```
0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB },
    { 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2,
0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79 },
    { 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C,
0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08 },
    { 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8,
0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A },
    { 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61,
0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E },
    { 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B,
0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF },
    { 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41,
0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16 }
};
```

2.2.3 行变换-ShiftRows()

行变换是将矩阵的每一行以字节为单位循环移位：第一行不变，第二行左移一位，第三行左移两位，第四行左移三位。

```
//2. 行变换 - 按字节循环移位
void ShiftRows(byte mtx[4 * 4])
{
    // 第二行循环左移一位
    byte temp = mtx[4];
```

```
    for (int i = 0; i<3; ++i)
        mtx[i + 4] = mtx[i + 5];
    mtx[7] = temp;
    // 第三行循环左移两位
    for (int i = 0; i<2; ++i)
    {
        temp = mtx[i + 8];
        mtx[i + 8] = mtx[i + 10];
        mtx[i + 10] = temp;
    }
    // 第四行循环左移三位
    temp = mtx[15];
    for (int i = 3; i>0; --i)
        mtx[i + 12] = mtx[i + 11];
    mtx[12] = temp;
}
```

2.2.4 列变换-MixColumns()

函数 MixColumns()同样接受一个 4x4 的字节矩阵作为输入，并对矩阵进行逐列变换

```
//  有限域上的乘法 GF(2^8)
byte GFMul(byte a, byte b) {
```

```
    byte p = 0;
    byte _bit_set;
    for (int counter = 0; counter < 8; counter++) {
        if ((b & byte(1)) != 0) {
            p ^= a;
        }
        _bit_set = (byte)(a & byte(0x80));
        a <<= 1;
        if (_bit_set != 0) {
            a ^= 0x1b;
            // x^8 + x^4 + x^3 + x + 1
        }
        b >>= 1;
    }
    return p;
}
// 列变换
void MixColumns(byte mtx[4 * 4])
{
    byte arr[4];
    for (int i = 0; i<4; ++i)
    {
```

```
        for (int j = 0; j<4; ++j)
            arr[j] = mtx[i + j * 4];

        mtx[i] = GFMul(0x02, arr[0]) ^ GFMul(0x03, arr[1]) ^
arr[2] ^ arr[3];
        mtx[i + 4] = arr[0] ^ GFMul(0x02, arr[1]) ^
GFMul(0x03, arr[2]) ^ arr[3];
        mtx[i + 8] = arr[0] ^ arr[1] ^ GFMul(0x02, arr[2]) ^
GFMul(0x03, arr[3]);
        mtx[i + 12] = GFMul(0x03, arr[0]) ^ arr[1] ^ arr[2]
^ GFMul(0x02, arr[3]);
    }
}
```

2.2.5 与扩展密钥的异或-AddRoundKey()

根据当前加密的轮数，用 w[]中的 4 个扩展密钥与矩阵的 4 个列进行按位异或。

```
//  轮密钥加变换 - 将每一列与扩展密钥进行异或
void AddRoundKey(byte mtx[4 * 4], word k[4])
{
    for (int i = 0; i<4; ++i)
    {
```

```
        word k1 = k[i] >> 24;
        word k2 = (k[i] << 8) >> 24;
        word k3 = (k[i] << 16) >> 24;
        word k4 = (k[i] << 24) >> 24;

        mtx[i] = mtx[i] ^ byte(k1.to_ulong());
        mtx[i + 4] = mtx[i + 4] ^ byte(k2.to_ulong());
        mtx[i + 8] = mtx[i + 8] ^ byte(k3.to_ulong());
        mtx[i + 12] = mtx[i + 12] ^ byte(k4.to_ulong());
    }
}
```

2.3AES 解密

2.3.1 S-盒逆变换

```
byte Inv_S_Box[16][16] = {
    { 0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF,
0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB },
    { 0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34,
0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB },
    { 0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE,
0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E },
    { 0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76,
```

```
0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25 },
    { 0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4,
0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92 },
    { 0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E,
0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84 },
    { 0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7,
0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06 },
    { 0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1,
0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B },
    { 0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97,
0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73 },
    { 0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2,
0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E },
    { 0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F,
0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B },
    { 0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A,
0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4 },
    { 0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1,
0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F },
    { 0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D,
0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF },
    { 0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8,
```

```
0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61 },
    { 0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1,
0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D }
};
```

2.3.2 逆行变换-InvShiftRows()

ShiftRows()是对矩阵的每一行进行循环左移，InvShiftRows()是对矩阵每一行进行循环右移。

```
// 逆行变换 - 以字节为单位循环右移
void InvShiftRows(byte mtx[4 * 4])
{
    // 第二行循环右移一位
    byte temp = mtx[7];
    for (int i = 3; i>0; --i)
        mtx[i + 4] = mtx[i + 3];
    mtx[4] = temp;
    // 第三行循环右移两位
    for (int i = 0; i<2; ++i)
    {
        temp = mtx[i + 8];
        mtx[i + 8] = mtx[i + 10];
```

```
        mtx[i + 10] = temp;
    }
    // 第四行循环右移三位
    temp = mtx[12];
    for (int i = 0; i<3; ++i)
        mtx[i + 12] = mtx[i + 13];
    mtx[15] = temp;
}
```

2.3.3 逆 S 盒变换-InvSubBytes()

```
void InvSubBytes(byte mtx[4 * 4])
{
    for (int i = 0; i<16; ++i)
    {
        int row = mtx[i][7] * 8 + mtx[i][6] * 4 + mtx[i][5]
* 2 + mtx[i][4];
        int col = mtx[i][3] * 8 + mtx[i][2] * 4 + mtx[i][1]
* 2 + mtx[i][0];
        mtx[i] = Inv_S_Box[row][col];
    }
}
```

2.3.4 逆列变换-InvMixColumns()

```
void InvMixColumns(byte mtx[4 * 4])
{
   byte arr[4];
   for (int i = 0; i<4; ++i)
   {
      for (int j = 0; j<4; ++j)
         arr[j] = mtx[i + j * 4];

      mtx[i] = GFMul(0x0e, arr[0]) ^ GFMul(0x0b, arr[1]) ^ GFMul(0x0d, arr[2]) ^ GFMul(0x09, arr[3]);
      mtx[i + 4] = GFMul(0x09, arr[0]) ^ GFMul(0x0e, arr[1]) ^ GFMul(0x0b, arr[2]) ^ GFMul(0x0d, arr[3]);
      mtx[i + 8] = GFMul(0x0d, arr[0]) ^ GFMul(0x09, arr[1]) ^ GFMul(0x0e, arr[2]) ^ GFMul(0x0b, arr[3]);
      mtx[i + 12] = GFMul(0x0b, arr[0]) ^ GFMul(0x0d, arr[1]) ^ GFMul(0x09, arr[2]) ^ GFMul(0x0e, arr[3]);
   }
}
```

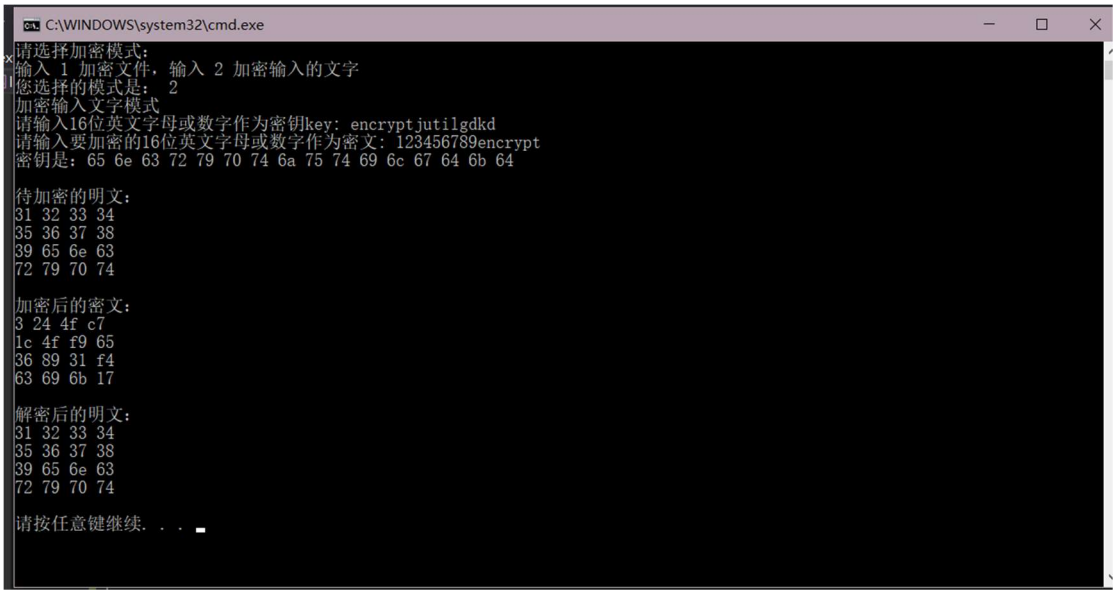
2.4 工具函数

```
//将一个 char 字符数组转化为二进制
```

```
//存到一个 byte 数组中


void charToByte(byte out[16], const char s[16])
{
   for (int i = 0; i<16; ++i)
      for (int j = 0; j<8; ++j)
         out[i][j] = ((s[i] >> j) & 1);
}


// 将连续的 128 位分成 16 组，存到一个 byte 数组中
void divideToByte(byte out[16], bitset<128>& data)
{
   bitset<128> temp;
   for (int i = 0; i<16; ++i)
   {
      temp = (data << 8 * i) >> 120;
      out[i] = temp.to_ulong();
   }
}


// 将 16 个 byte 合并成连续的 128 位
bitset<128> mergeByte(byte in[16])
```

```
{
    bitset<128> res;
    res.reset();  // 置0
    bitset<128> temp;
    for (int i = 0; i<16; ++i)
    {
        temp = in[i].to_ulong();
        temp <<= 8 * (15 - i);
        res |= temp;
    }
    return res;
}
```

2.5 运行结果



3 实验总结

AES 算法和 DES 算法都是对称加密算法，但是 AES 的加密过程比 DES 的加密过程更复杂，但是同时 AES 的加密效果也比 DES 的效果要好。因为 AES 算法的加密过程比较复杂，加密过程中针对 128 位的数据进行加密和解密，因此在操作的时候比较复杂，特别是加密中文的时候处理比较困难。