加密算法简介

声明：

对称加密和非对称加密

对称加密是最快速、最简单得一种加密方式。加密（encryption）与解密（decryption）用的是同样的密钥（secret key）；反之，则是非对称加密。

AES算法加密算法背后的数学

AES是如今最为常用的堆成加密算法，它不仅安全，而且性能表现也很出色。稍后，我们也会用它来加密HTTP API返回的数据。不过在使用这种加密算法之前，我们先来了解下这种加密算法是如何实现的。

为了看懂AES的实现，我们得补充一些必要的数学知识。要说一下的是，对这些数学知识的理解视角，是为了看懂AES的实现代码，并非研究数学理论。因此，接下来的有些描述可能不那么科学和严谨，但对于理解代码却足够用了。

群

我们要介绍的第一个概念，叫做群（Group）。数学中群有下面几个性质：

- 首先，是**封闭性**。也就是说，对于可以在群里使用的操作（用*op*表示），如果*a*和*b*属于群*G*，那么*a op b*的结果也属于G；
- 其次，是群里的操作都支持**结合律**，也就是（a op b）**op** c = a **op**（b op c）；
- 第三，是群中一定存在一个元素e，使得群众的每一个元素a，都有（a **op** e） = （e **op** a）= a成立。我们管e叫做G的**单位元（Identity Element）**；
- 最后，群中的每一个元素a，一定可以在群中找到另外一个元素b，使得（a **op** b） = （b **op** a） = e。这时，我们管b叫做a的**逆元（Inverse Element）**；

我们来个具体的例子。假设，在一个所有证书...，-3，-2，-1，0，1，2，3，...构成的集合*Z*里，我们限制只能在这个集合里做加法，那么显然：

- 两个整数相加的结果肯定还是整数，它满足封闭性；
- 整数加法当然也支持结合律；
- 在这个集合中，整数0满足a + 0 = 0 + a = a；
- 类似的，在这个集合中，整数-a满足（a +（-a）=（-a）+ a = 0。

因此，这个**由所有整数构成的集合Z和整数的加法放在一起，就变成了一个群**，这个群的单位元时0，每一个整数a的逆元是-a。

有限群

接下来，我们让刚才的集合*Z*只包含有限个元素，让Z1等于-3，-2，-1，0，1，2，3。这时，Z1和架构构成的组合就不能叫做一个群了，因为显然，1+3，2+3都已经超过了Z1表达的范围。至于群剩余的属性，Z1仍旧是满足的。

那么如何让Z1也成为一个群呢？

其实也很简单，不就是要限定数值的范围么？把结果对3取模就行了。于是，我们只要求Z1可以进行的操作，是相加后对结果取模，组合上这种操作，Z1就是一个新的群了。通过这个过程，我们可以挖掘出三个新的认知：

- 首先，群操作**不一定有交换律**，对于Z来说，操作只有加法，因此a + b = b + a。但对于Z1来说，由于相加之后还要取模，这时操作的顺序就很重要了，它并不支持交换律。并且，在数学里，管这种支持交换律的群，叫做**可交换群**，也叫做**阿贝尔群**，这是用提出这个概念的挪威数学家尼尔斯·阿贝尔命名的；

- 其次，对于这种元素个数有限的群，也有一个专门的名字，叫做**有限群**，而元素的数量，叫做有限群的**阶**，也就是说，Z1是一个6阶有限群；
- 最后，**群的计算操作知识一种抽象描述，它并不局限于单一的初等代数中的四则运算**，请大家务要理解这个；

取模和取余

定义

取模运算（Module Operation）和**取余运算（Complementation）**两个概念有重叠的部分但又不完全一致。主要的区别在于对负数进行除法运算时操作不同。取模主要是用于计算机术语中。取余更多是数学概念。

区别

对于整数a，b来说，取模运算或者求余运算的方法都是：

1. 求 整数商：c = a/b；
2. 计算模或者余数：r = a - c * b。

求模运算和求余运算在**第一步不同**：取余运算在取c的值时，向0方向舍入（fix（）函数）；

而取模运算在计算c的值时，向负无穷大方向舍入（floor（）函数）。

例如计算：-7 Mod 4

那么：a = -7；b = 4；

第一步：求整数商c，如进行求模运算c = -2（向负无穷大方向舍入），求余c = -1（向0方向舍入）；

第二步：计算模和余数的方式相同，但因c的值不同，求模时r = 1，求余时人= -3。

归纳：当a和b符号一致时，求模运算和求余运算所得的c的值一致，因此结果一致。

当符号不一致时，结果不一样。求模运算结果的符号和b一致，求余运算结果的符号和a一致。

补充：

7 mod 4 = 3（商 = 1 或 2，1<2，取商=1）

-7 mod 4 = 1（商 = -1 或 -2，-2<-1，取商=-2）

7 mod -4 = -1（商 = -1或-2，-2<-1，取商=-2）

-7 mod -4 = -3（商 = 1或2，1<2，取商=1）

这里模是4，取模其实全称应该是取模数的余数，或**取模余**。

运算规则

模运算与基本四则运算有些相似，但是除法例外。其规则如下：

1. (a + b) % p = (a % p + b % p) % p （1）
2. (a - b) % p = (a % p - b % p) % p （2）
3. (a * b) % p = (a % p * b % p) % p （3）
4. a ^ b % p = ((a % p)^b) % p（4）

- 结合律：

  ((a+b) % p + c) % p = (a + (b+c) % p) % p （5）

  ((a*b) % p * c)% p = (a * (b*c) % p) % p （6）

- 交换律：

(a + b) % p = (b+a) % p （7）

(a * b) % p = (b * a) % p （8）

- 分配律：

(a+b) % p = (a % p + b % p) % p （9）

((a +b)% p * c) % p = ((a * c) % p + (b * c) % p) % p （10）

重要定理

- 若a≡b (% p)，则对于任意的c，都有(a + c) ≡ (b + c) (%p)； （11）
- 若a≡b (% p)，则对于任意的c，都有(a * c) ≡ (b * c) (%p)； （12）
- 若a≡b (% p)，c≡d (% p)，则 (a + c) ≡ (b + d) (%p)，(a - c) ≡ (b - d) (%p)，

(a * c) ≡ (b * d) (%p)； （13）

应用

- **判别奇偶数**

奇偶数判别时模运算最基本的应用，也非常简单。

已知一个整数n对2取模，如果余数为0，则表示n为偶数，否则n为奇数。

- 约数

约数，又称因数。整数a除以整数b（b≠0）除得的商正好是整数而没有余数，我们就说a能被b整除，或b能整除a。a称为b的倍数，b称为a的约数。

```
a = 9, b = 3
a / b = 3 整除无余数
则b = 3是a的约数
b = 1
a / b = 9
则b = 1也是a的约数
```

- 素数

素数又称`质数`，是指再大于1的自然数中，出了1和它本身意外不再有其他`因数`的自然数。

- **求最大公约数**

最大公因数，也称最大公约数、最大公因子，指两个或多个整数共有`约数`中最大的一个。a，b的最大公约数记为（a，b），同样的，a，b，c的最大公约数记为（a，b，c）。

求最大公约数最常见的办法是`欧几里得算法`（又称辗转相除法），其计算原理依赖于定理：gcd（a，b）= gcd（b，a mod b）

```
inline int GCD(int x,int y)
{
    int r=x%y;
    while(r) x=y,y=r,r=x%y;
    return y;
}
```

- **模幂运算**

利用模运算的运算规则，我们可以使某些计算得到简化。

例如，我们想知道3333^5555的末位是什么。很明显不可能直接把3333^5555的结果计算出来，那样太长了。但我们想要确定的是3333^5555（%10），所以问题就简化了。

根据运算规则（4）a^b % p = ((a % p)^b) % p，我们知道3333^5555（%10）= 3^5555（%10）。

根据运算规则（3）(a * b) % p = (a % p * b % p) % p，由于5555 = 4 * 1388 + 3，我们得到3^5555（%10）=（3^(4*1388) * 3^3）（%10）=（（3^(4*1388)）（%10）* 3^3（%10））（%10）

=（（3^(4*1388)）（%10）* 7）（%10）。

根据欧拉定理可以得到 3 ^ (4 * k) % 10 = 1, 所以（（3^(4*1388)）（%10）* 7）（%10）= (1 * 7) (% 10) = 7

计算完毕。

域

了解了群子厚，我们来介绍第二个概念，叫做域（Field）。组成一个域的元素有下面这些性质：

- 可以形成一个加法交换群，这个群的单位元为0；
- 除了0之外，可以形成一个乘法交换群，这个群的单位元为1；
- 当混合使用加法和乘法的时候，分配律总是成立的，即：（a + b）* c =（a * c）+（b * c）；

那什么样的集合又这样的性质呢？其中最家常的例子，就是实数集合。因为：

- 实数集合的加法支持交换律，并且任何数加0都是自己；
- 实数集合的称发也支持交换律，任何数乘以1都是自己；
- 实数集合上的乘法和加法当然也支持结合律；

伽罗瓦域（Galois field）

和刚才研究群类似，如果我们限定一下域中的元素又如何呢？由于域要求其中的元素同时可以形式加法交换群和乘法交换群，它一下子就不那么容易直观想象的出来了。不过，好在已经有数学家为我们研究出了一个定理：

> 如果m等于一个素数p的n次幂（n为正整数），即：m = p^n时，才存在一个m阶的域。

其中：`n`等于1时叫做素域（Prime Field），`n >= 2`时，叫做扩展域（Field Extension）。

并且，这种元素个数有限的域还有一个特别的名字，叫做伽罗瓦域（Galois Field），记做`GF(m)`这是用它的提出者，法国数学家伽罗瓦的名字来命名的。

接下来，我们来看个最简单的2阶伽罗瓦域`GF(2)`。它只包含两个元素：`{0, 1}`。但很明显，`1+1`已经超过了`GF(2)`能表达的范围，因此，这个域里面，它的加法和乘法同样是特别定制的，而定制的方法，就是对2取模：

```
(0 + 0) mod 2 = 0
(0 + 1) mod 2 = 1
(1 + 0) mod 2 = 1
(1 + 1) mod 2 = 0

(0 * 0) mod 2 = 0
(0 * 1) mod 2 = 0
(1 * 0) mod 2 = 0
(1 * 1) mod 2 = 1
```

也就是说，在`GF(2)`定义的这个世界里，加法操作的规则是：全0或全1相加等于0，否则等于1，它和计算机中二进制中的`XOR`是一样的。而乘法规则是：全1相乘等于1，否则等于0，它和二进制运算中的`AND`是一样的。

基于这样运算法则，不难推断，在`GF(2)`里，**1的加法和乘法逆元都是它自己。**这有什么用呢？这样，在`GF(2)`里，就可以支持减法和除法操作了。因为减1就是加上1的逆元，除以1就是乘以1的逆元。**再说的直白一些，在`GF(2)`里，减法就是加法，除法就是乘法。**在后面的内容中，我们就会看到，AES加密算法中，对每一个bit的处理，就用了`GF(2)`中的运算。

接下来，如果我们把bit放大到字节，就不难想到，为了可以用域表示一个字节的所有整数，我们得使用一个扩展域`GF(2^8)`。

用多项式表达的扩展域

对于`GF(2^8)`来说，我们并不能直接用`0-255`这256个数字来表示了，取而代之的方法，是使用多项式。`GF(2^8)`中的每一个元素，都用形如：

```
a7x^7 + a6x^6 + a5x^5 + ... + a1x + a0
```

这样的多项式表达。其中，`a0 - a7`是多项式的系数，它们只能从`GF(2)`中取值。这样，`a0 - a7`就代表了一个字节中的对应bit。并且，这些系数的运算法则，也使用刚才`GF(2)`中提到的加法和乘法法则。来看个例子：下面这两个，都是`GF(2^8)`中的元素：

```
A(x) = x^7 + x^5 + x^3 + 1
B(x) = x^2 + 1
```

多项式加法

按照之前的约定，它们分别表示二进制数的`10101001`和`00000101`。接下来，为了在`GF(2^8)`中计算`A(x) + B(x)`，我们只要按位依次执行`GF(2)`中的加法就好了，也就是说它等于：

```
  1 0 1 0 1 0 0 1
+ 0 0 0 0 0 1 0 1
------------------
  1 0 1 0 1 1 0 0
```

把这个结果对应到多项式，就是`x^7 + x^5 + x^3 + x^2`，而这个多项式对应的字节，是`0xab`。

多项式乘法

但`GF(2^8)`多项式的乘法，就麻烦一些了。为了计算`A(x)B(x)`，我们先按照普通代数的计算方法展开多项式：

```
A(x)B(x) = (x^7 + x^5 + x^3 + 1)(x^2 + 1)
  = x^9 + x^7 + x^5 + x^2 + x^7 + x^5 + x^3 + 1
  = x^9 + 2x^7 + 2x^5 + x^3 + x^2 + 1
```

然后，把这个多项式的系数对2取模：

```
    x^9 + 2x^7 + 2x^5 + x^3 + x^2 + 1
mod 2
-------------------------------------
    x^9                 x^3 + x^2 + 1
```

这样，就可以得到`x^9 + x^3 + x^2 + 1`。但问题是`x^9`已经超过了`GF(2^8)`可以表达的范围，它也对应不到计算机的一个字节中的bit。显然这个结果是不对的，怎么办呢？

其实，思路和刚才在处理`GF(2)`加法结果的时候是类似的。既然我们要把结果限定在`x^8`以内，那就把多项式对`x^8`取模不就行了。道理是这么个道理，不过这个取模的多项式，可不是随便选的。AES的设计者Vincent Rijmen 使用了一个经典的**不可约多项式**：`P(x) = x^8 + x^4 + x^3 + x + 1`。什么是不可约呢？就是说，`P(x)`不能表示为`GF(2^8)`域中两个多项式的乘积。这和自然数中的素数是类似的，因此，`P(x)`也叫**素多项式**。

由于`P(x)`的最高次幂是8，因此对`P(x)`取模，结果就一定会落在`GF(2^8)`里了，我们来算一下。先算`xP(x)`让`P(x)`升到9次幂：

```
xP(x) = x^9 + x^5 + x^4 + x^2 + x
```

其次，用`x^9 + x^3 + x^2 + 1 - x(Px)`：

```
  x^9 +             x^3 + x^2     + 1
- x^9 + x^5 + x^4 +       x^2 + x
---------------------------------------
        x^5 + x^4 + x^3       + x + 1
```

这样，得到的结果就落在`GF(2^8)`里了，它表示的二进制数是`00111011`，这就是`GF(2^8)`中`A(x)B(x)`的运算结果。下一节就会看到，AES正是使用这种方法对字节进行了处理。

求多项式的逆元

了解了多项式乘法之后，我们就能计算`GF(2^8)`中任意一个多项式`A(x)`的逆元了，AES加密算法需要这个值。要说明的是，选取的素多项式不同，`GF(2^8)`中元素的逆元也不同。这里，我们就用刚才说到的`P(x)`举例了。

对于给定的一个`GF(2^8)`中的多项式`A(x)`，如果`A(x)B(x)`的结果对`P(x)`取模等于1，那么`B(x)`就是`A(x)`的逆元了。那该怎么计算这个`B(x)`呢？

第一种方法，当然就是暴力穷举。用`GF(2^8)`中的每一个元素分别和`A(x)`计算；

第二种方法，来自于一个有趣的现象。`GF(2^8)`中的每一个非0多项式，都可以表示成`(x+1)`的`n`次幂，这里`0<=n<=254`，例如：

```
(x+1)^0 = 1
(x+1)^1 = x+1
(x+1)^2 = x^2+1
...
(x+1)^254 = x^7+x^6+x^5+x^4+x^3+x^2+x
```

注意这里我们执行的是`GF(2)`中的加法和乘法。并且，当`n`从255开始，就又会用和之前同样的顺序遍历`GF(2^8)`中的多项式。

```
(x+1)^255 = 1
(x+1)^256 = x+1
...
```

正是由于这种特性，`(x+1)`有了一个形象的名字，叫做**生成元**。有了生成元之后，我们可以把`GF(2^8)`中的每一个多项式和它对应的生成元的幂保存成一个映射表。接下来，对于给定的`A(x)`，如果`B(x)`是`A(x)`的逆元，则一定有下面的关系成立：

```
A(x)B(x) = (x+1)^a (x+1)^b = (x+1)^(a+b) = 1 (mod P(x))
```

于是，我们只要根据之前创建的映射表，查到`A(x)`对应的幂`a`，然后用`255-a`计算出`b`，再从表中查出`b`对应的多项式，就是`A(x)`的逆元了。

当然，除了这两种方法之外，还有很多数学上的手段来计算逆元，不过就像一开始说的，我们的重点并不是研究数学，也不会手动去计算逆元值。通过这些计算过程，大家理解和伽罗瓦域、单位元以及逆元这些概念的含义就行了。

What's next?

至此，为了实现AES加密算需要铺垫的数学知识，就说完了。下一节，我们就可以来看AES究竟是如何实现加密解密的了。

AES加密对数据的混淆

这一节，我们通过一个GitHub上的开源项目tiny-AES-c，来学习AES加密算法的内部实现。参与这个AES加密的成员主要有两个：一个是要加密的原始数据，它必须是固定的16字节长度；把它用C语言表示，就是这样：

```
typedef uint8_t state_t[4][4];
```

另一个是加密使用的密钥，它的长度可以是16/24/32字节，分别对应着平时我们说的AES-128/192/256加密算法。我们可以定义一些宏来表示不同AES加密算法使用的密钥长度：

```
#if defined(AES256) && (AES256 == 1)
    #define AES_KEYLEN 32
#elif defined(AES192) && (AES192 == 1)
    #define AES_KEYLEN 24
#else
    #define AES_KEYLEN 16   // Key length in bytes
#endif
```

SubBytes

了解了这些基本信息之后，我们就从对原始数据的预处理开始。AES加密算法的第一步，叫做`SubBytes`。它的目的是对原始数据的每个字节进行混淆。混淆的方法是先计算每一个字节在`GF(2^8)`的乘法逆元，再把这个逆元按照下面的方式做一次仿射变换（就是乘以一个矩阵再加上一个向量）：

```
|s0|    |1 0 0 0 1 1 1 1| |b0|   |1|
|s1|    |1 1 0 0 0 1 1 1| |b1|   |1|
|s2|    |1 1 1 0 0 0 1 1| |b2|   |0|
|s3| =  |1 1 1 1 0 0 0 1| |b3| + |0|
|s4|    |1 1 1 1 1 0 0 0| |b4|   |0|
|s5|    |0 1 1 1 1 1 0 0| |b5|   |1|
|s6|    |0 0 1 1 1 1 1 0| |b6|   |1|
|s7|    |0 0 0 1 1 1 1 1| |b7|   |0|
```

这里的`b0-b7`表示原始数据中每一个字节的bit，经过上面一番计算，就得到变换之后的值了。由于`GF(2^8)`中，0没有乘法逆元，我们约定**0的乘法逆元就是0**。把0带入到上面的公式，不难计算到0混淆之后的值是`0x63`（这里的乘法和加法要使用`GF(2)`中的计算规则）。

接下来，我们按照这样的方法，就可以计算出256个值，每个值都对应一个字节的值混淆之后的结果。在很多AES的实现里，为了效率，这些值都是硬编码出来的：

```
static const uint8_t sbox[256] = {
//0     1     2     3     4     5     6     7     8     9     ...
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, ...
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, ...
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, ...
  ...
};
```

这里，受篇幅限制，我们只列出了前9列和前3行，实际上，`sbox`是一个16x16的方针，`sbox`是Substitution-box的缩写，也就是“替换盒子”。

理解了`sbox`之后，就不难想到，在解密的时候，还应该有一个盒子用于还原混淆后的结果。这个盒子，叫做Reverse Substitution-box。它的计算方法和加密混淆时是一样的，只是仿射变换时使用的矩阵和向量不同罢了：

```
|b0|    |0 0 1 0 0 1 0 1| |s0|   |1|
|b1|    |1 0 0 1 0 0 1 0| |s1|   |0|
|b2|    |0 1 0 0 1 0 0 1| |s2|   |1|
|b3| =  |1 0 1 0 0 1 0 0| |s3| + |0|
|b4|    |0 1 0 1 0 0 1 0| |s4|   |0|
|b5|    |0 0 1 0 1 0 0 1| |s5|   |0|
|b6|    |1 0 0 1 0 1 0 0| |s6|   |0|
|b7|    |0 1 0 0 1 0 1 0| |s7|   |0|
```

当然，我们只要知道这种方法就好了，实际编码的时候，这个“逆替换盒子”，也是硬编码实现的：

```
static const uint8_t rsbox[256] = {
//0     1     2     3     4     5     6     7     8     9     ...
  0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, ...
  0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, ...
  0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, ...
  ...
};
```

于是，加密和解密的查表过程，就可以写成这样：

```
#define getSBoxValue(num) (sbox[(num)])
#define getSBoxInvert(num) (rsbox[(num)])
```

ShiftRows

看到这里，不难理解`SubBytes`是`bit`级别的混淆。接下来，AES还在`byte`级别进行了混淆，这一步叫做`ShiftRows`。它的操作用一张图表示就直观了：

图中的每一个方块，都表示一个字节。其中`d0-d15`表示原始数据序列。经过第一轮`SubBytes`变换之后，变成了`s0-s15`。所谓`ShiftRows`，就是把`s0-s15`放到箭头指定的位置。左边的线条看着可能有点乱，但如果把`s0-s15`**每四个字节一列**变成一个方阵，计算方法就清楚了。所谓的`ShiftRows`就是把这个方阵的每一行，向左循移动：第0行保持不动，第1行向左移动1个字节，第2行向左移动2个字节，第3行向左移动3个字节。

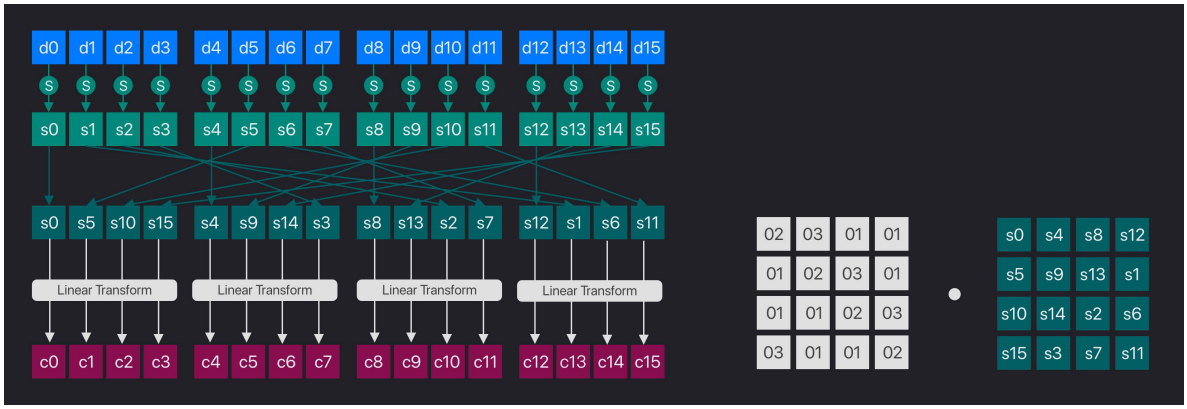
这个过程实现出来就是这样的，它的输入参数是`SubBytes`之后的16字节数据：

```
static void ShiftRows(state_t* state)
{
  uint8_t temp;

  // Rotate first row 1 columns to left
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp           = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp           = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}
```

而对于解密的过程，则是把方阵的每一行按照上面的规则向右平移就好了。

MixColumn

`ShiftRows`完成后，我们就得到了一个进一步被打乱次序的字节序列。接下来，AES要继续以每4个字节为单位进行混淆，这一步叫做`MixColumn`。



如图中所示，我们用下面这个方阵和`ShiftRows`之后的结果相乘：

```
|02 03 01 01|
|01 02 03 01|
|01 01 02 03|
|03 01 01 02|
```

得到的新方阵，就是`MixColumn`的结果。不过，**这里的矩阵运算，要使用`GF(2^8)`中的加法和乘法。**例如，假设`ShiftRows`之后的方阵是这样的（这里只是为了演示方便，实际不可能是这种结果的）：

```
|11 11 11 11|
|11 11 11 11|
|11 11 11 11|
|11 11 11 11|
```

00010001 对于十六进制数字`0x11`，它对应的多项式是`x^4+1`，和它进行计算的矩阵中的`01 / 02 / 03`对应的矩阵分别是`1 / x / x+1`，于是：

```
01 * 11 = 1 * (x^4 + 1) = x^4 + 1
02 * 11 = x * (x^4 + 1) = x^5 + x
03 * 11 = (x + 1)(x^4 + 1) = x^5 + 1
```

因此，相乘之后这个方阵的结果就是：

```
(01 + 01 + 02 + 03) * 11
= (1 + 1 + x + x + 1) * (x^4 + 1)
= x^4 + 1
```

由于这个结果的最高次幂没有超过`x^7`，我们就不用对`P(x)`取模了。当然，这个运算的结果并不重要，大家理解这个以4字节为单位的混淆过程就好了。接下来的问题就是，如何通过编程实现呢？

为了理解这个方法，我们先手动算几个值。现在，假设`state_t`中已经是`ShiftRows`操作完成之后的结果了。按照之前的说明：`c0 = 2s0 + 3s5 + s10 + s15`。

但别忘了，我们执行的是多项式运算，因此要把2和3替换成对应的多项式：

```
xs0 + (x+1)s5 + s10 + s15
```

把它展开，就是：

```
x(s0 + s5) + s5 + s10 + s15
```

这里的`s0 / s5 / s10 / s15`对应着`state_t[0][0] / [0][1] / [0][2] / [03]`，于是，这个式子还能写成这样：

```
c0 = x(state_t[0][0] + state_t[0][1])
     + state_t[0][1] + state_t[0][2] + state_t[0][3]
```

接下来，算`c1`，用同样的方式，不难得到这样的结果：

```
c1 = x(state_t[0][1] + state_t[0][2])
     + state_t[0][0] + state_t[0][2] + state_t[0][3]
```

最后，我们把`c0 - c3`的结果罗列在一起：

```
c0 = x(state_t[0][0] + state_t[0][1])
     + state_t[0][1] + state_t[0][2] + state_t[0][3]

c1 = x(state_t[0][1] + state_t[0][2])
     + state_t[0][0] + state_t[0][2] + state_t[0][3]

c2 = x(state_t[0][2] + state_t[0][3])
     + state_t[0][0] + state_t[0][1] + state_t[0][3]

c3 = x(state_t[0][3] + state_t[0][0])
     + state_t[0][0] + state_t[0][1] + state_t[0][2]
```

发现其中的规律了么？在每一个结果的第二行，其实都差了一个元素`state_t[0][i]`，只要补上这个值，第二行的计算就相同了，于是`c0`还可以写成这样（注意这里执行的是`GF(2)`加法，`1+1=0`）：

```
c0 = x(state_t[0][0] + state_t[0][1])
     + state_t[0][1] + state_t[0][2] + state_t[0][3] + state_t[0][0]
     + state_t[0][0]
```

接下来，来看第一行的部分。还是用`c0`举例，在乘以`x`之后，结果有两种情况：

- 第一种情况是最高次幂没有超过`x^7`，这时的结果就是相乘之后的结果，对应的计算方法，就是把`state_t[0][0] + state_t[0][1]`向左移动一个bit（这里执行的是`GF(2)`乘法）；
- 第二种情况是最高次幂变成了`x^8`，这时，我们就要把这个多项式对`P(x)`取模，这个操作是有固定方法的，就是把结果左移`1bit`之后和`0x1B`进行`xor`操作；

于是，这个`乘以x`的部分，可以实现成下面就这样：

```
uint8_t xtime(uint8_t x) {
  return (x<<1) ^ (((x>>7) & 1) * 0x1b);
}
```

至此，用于计算`MixColumn`的细节就都说完了，把它们组合成一个函数，就是这样的：

```
void MixColumns(state_t* state)
{
  uint8_t i;
  uint8_t Tmp, Tm, t;

  for (i = 0; i < 4; ++i)
```

```
  {
    t   = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;

    Tm  = (*state)[i][0] ^ (*state)[i][1] ;
    Tm = xtime(Tm);
    (*state)[i][0] ^= Tm ^ Tmp ;

    Tm  = (*state)[i][1] ^ (*state)[i][2] ;
    Tm = xtime(Tm);
    (*state)[i][1] ^= Tm ^ Tmp ;

    Tm  = (*state)[i][2] ^ (*state)[i][3] ;
    Tm = xtime(Tm);
    (*state)[i][2] ^= Tm ^ Tmp ;

    Tm  = (*state)[i][3] ^ t ;
    Tm = xtime(Tm);
    (*state)[i][3] ^= Tm ^ Tmp ;
  }
}
```

理解了上面的计算过程，这段像"天书"一样的代码就不难理解了。要再次强调的是，我们执行的是`GF(2)`中的加法和乘法，也就是说，加法是`xor`运算，乘法是`and`运算。

而解密时`MixColumns`执行的操作，和加密是一样的，只不过方阵相乘的时候，使用的常数方阵值不一样罢了

```
|0E 0B 0D 09|
|09 0E 0B 0D|
|0D 09 0E 0B|
|0B 0D 09 0E|
```

What's next?

当执行完了`MixColumns`的四轮操作之后，对原始数据的混淆工作，就结束了。接下来要做的事情，就是根据使用的密钥长度，对混淆过的数据进行不同轮数的加密。下一节，我们就来实现这个过程。

AES加密算法的密钥扩展算法

这一节我们来看AES是如何对混淆后的数据实施加密的。为了便于理解这个过程，我们先假设原始数据和密钥的长度，都是128位，也就是AES-128加密算法。

其实，最终的加密过程很简单，就是把密钥和混淆后的结果进行`xor`运算（也就是`GF(2)`上的加法运算）就好了。但在AES里，这个过程会执行多次，每一次叫做一轮加密，并且每一轮使用的密钥都是根据上一轮的密钥变换而来的。因此，只要我们搞清楚密钥的生成算法，就能理解AES全部的加密/解密流程了。

密钥扩展的轮数

那么，这个密钥的扩展是如何完成的呢？同样，我们用一些图结合代码来理解。首先是原始的128位密钥，一共16个字节：

其中：`k0-k15`表示原始key中的每一个字节，我们把它们4字节分成一组，计作`w0-w3`。每一轮密钥的扩展用一张图表示，就是这样的：

先计算`w4`，我们要把`w3`经过一个函数`g`处理之后，与`w0`异或；然后用下面的方式计算`w5-w7`：

```
w4 = g(w3) xor w0
w5 = w4 xor w1
w6 = w5 xor w2
w7 = w6 xor w3
```

把得到的`w4-7`再组合起来，就是第一轮加密使用的密钥了。把这个计算过程写成更一般的形式，就是这样的：

```
W(4i)   = g(W(4i-1)) xor W(4i-4)
W(4i+1) = W(4i)      xor W(4i-3)
W(4i+2) = W(4i+1)    xor W(4i-2)
W(4i+3) = W(4i+2)    xor W(4i-1)
```

其中，`i=1, 2, 3, ..., 10`，这是因为之前我们说过，128位的密钥，要执行10轮加密。

密钥自身的混淆

理解了这个过程之后，剩下的问题，就是刚才说过的函数`g`了，它做了什么呢？我们还是用一张图来表示：

其中，`b0-b3`表示输入的4个字节，也就是之前的图中我们看到的`w3`。然后，先把`w3`向左移动一个字节，再把得到的结果，根据之前的SBox进行`SubBytes`替换。替换之后，把得到的结果的第一个字节，和图中的`RC`数组进行`xor`操作，就可以得到第二个密钥中的`w5`。就这样，每一轮加密密钥中的`w(4i)`组，都采用同样的计算方法。

至于这个`RC`是怎么算出来的，我们就不展开说了，反正它不会变，当成常量处理就行。大家要想了解它的算法，可以参考这里并且，`RC[0]`在AES加密中是没意义的，第一轮加密使用的值是`RC[1]`。

扩展密钥算法的实现

了解了扩展密钥算的算法之后，我们来看它的实现过程：

```
#define Nb 4
#define Nk 4        // The number of 32 bit words in a key.
#define Nr 10       // The number of rounds in AES Cipher.

void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
  unsigned i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for (i = 0; i < Nk; ++i)
```

```
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }
}
```

这里：

- `Nb`表示原始数据以4字节为单位的长度；
- `Nk`表示以4字节为单位的密钥长度；
- `Nr`表示加密执行的轮数；
- 参数`Key`表示原始密钥；
- 参数`RoundKey`用于返回生成的新密钥；

在上面的代码里，执行的是初始化阶段，`RoundKey`的前16个字节（也就是`W0-W3`），是原始密钥。

然后，定义一个执行`Nb * Nr`次的循环，每循环一次，就生成密钥中的一个`W(i)`。`Nr`轮加密一共需要`Nb * Nr`组`W`。在这个循环的一开始，先读出`W(i-1)`组四个字节的值：

```
void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
  /// ...

  // All other round keys are found from the previous round keys.
  for (i = Nk; i < Nb * (Nr + 1); ++i) {
    // Initial round
    {
      k = (i - 1) * 4;
      tempa[0] = RoundKey[k + 0];
      tempa[1] = RoundKey[k + 1];
      tempa[2] = RoundKey[k + 2];
      tempa[3] = RoundKey[k + 3];
    }
  }
}
```

如果`i`的值是4的倍数，就要执行函数`g`变换，这个变换里，要先向左移动一个字节：

```
// All other round keys are found from the previous round keys.
for (i = Nk; i < Nb * (Nr + 1); ++i)
{
  // Initial round
  // ...

  if (i % Nk == 0)
  {
    // RotWord
    {
      const uint8_t u8tmp = tempa[0];
      tempa[0] = tempa[1];
      tempa[1] = tempa[2];
      tempa[2] = tempa[3];
      tempa[3] = u8tmp;
    }
  }
```

```
}
```

对移动后的结果做`SubBytes`替换：

```
// Initial round
// ...
if (i % Nk == 0)
{
  // RotWord
  // ...

  // SubWord
  {
    tempa[0] = getSBoxValue(tempa[0]);
    tempa[1] = getSBoxValue(tempa[1]);
    tempa[2] = getSBoxValue(tempa[2]);
    tempa[3] = getSBoxValue(tempa[3]);
  }
}
```

再把变换后的第一个字节，和`RC`常量数组进行`xor`计算：

```
if (i % Nk == 0)
{
  // RotWord

  // SubWord

  tempa[0] = tempa[0] ^ Rcon[i/Nk];
}
```

至此，`W(i-1)`这一组的`g`变换就完成了。接下来，根据上图的规则生成`W(i)`这一组的4字节数据。并且，如果`i`不是4的倍数，也直接执行这里的变换就好了：

```
for (i = Nk; i < Nb * (Nr + 1); ++i)
{
  /// ...

  if (i % Nk == 0)
  {
    // RotWord

    // SubWord

    // xor
  }

  j = i * 4; k = (i - Nk) * 4;
  RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
  RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
  RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
  RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
}
```

这样，密钥扩展的流程就完成了。此时`RoundKey`中包含的，就是扩展出来的密钥。真正的加密工作之前我们说过了，就是把数据和密钥进行`xor`，因此这一步的实现就很简单了：

```
void AddRoundKey(uint8_t round, state_t* state, const uint8_t* RoundKey) {
  uint8_t i,j;
  for (i = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j) {
      (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
    }
  }
}
```

其中：

- `round`表示轮数；
- `state`表示上一轮加密后的结果；
- `RoundKey`表示之前生成好的扩展密钥；

密钥长度带来的变化

在这一节最后，我们来说说AES-192和AES-256这两种加密算法。它们的数据加密流程和AES128是完全一样的。唯一的差别就是在扩展密钥时的方法和执行加密的轮数不同。

AES-192

对于AES-192来说，密钥的长度是24字节，要加密12轮，也就是`Nk = 6; Nr=12`。但用于加密的数据仍旧是16字节，即`Nb = 4`。

如上图所示，此时，密钥的扩展就变成了每6字节为单位，而要应用函数`g`的`W`组，也变成了6的倍数。并且，初始轮的密钥，以及第一轮密钥的前8个字节，都直接来自原始密钥。

AES-256

最后，我们来说说AES-256，它的密钥长度为32字节，要加密14轮，也就是`Nk = 8; Nr = 14`。要加密的数据同样是16字节，即`Nb = 4`。既然`Nk = 8`，不难想象，`W`组的`g`变换要8的倍数一组执行了。但除此之外，`W`组每4组，还要额外进行另外一次变换`h`，把这个过程用一张图表示，就是这样的：

相比函数`g`，`h`就简单多了，只要把作为输入的4字节做一次`SubBytes`替换就好了。因此，在刚才实现的`KeyExpansion`函数里，当使用`AES-256`算法的时候，生成最终的密钥之前，还要添加一段额外的代码：

```
for (i = Nk; i < Nb * (Nr + 1); ++i)
{
  /// ...

  if (i % Nk == 0)
  {
    // RotWord

    // SubWord

    // xor
  }
```

```
  #if defined(AES256) && (AES256 == 1)
    if (i % Nk == 4)
    {
      // h
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }
    }
#endif

  // Generate RoundKey
}
```

What's next?

至此，AES加密算法的计算细节，我们就都说完了。但看到这里，相信你心里一直都有一个疑问，既然AES只能针对定长数据进行加密，它是如何处理任意长度数据的呢？下一节，我们就来实现最终的加密算法，并了解AES这种分组密码的几种不同工作模式的实现。

AES加密解密的实现以及工作模式

这一节，我们完成两个工作。首先，实现AES加密的基本流程；其次，了解AES加密的几种常用工作模式。

实现AES加密

实际上，有了前面几节的内容准备，实现AES加密就非常简单了。直接来看代码：

```
void Cipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;
  AddRoundKey(0, state, RoundKey);

  for (round = 1; round < Nr; ++round)
  {
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }

  // The last round is given below.
  SubBytes(state);
  ShiftRows(state);
  AddRoundKey(Nr, state, RoundKey);
}
```

其中：

- `state`指向要加密的钥匙数据；
- `RoundKey`指向扩展好的密钥；

整体的执行流程，就是在前面几节介绍的过程。在这段代码里，我们能更清楚地看到加密过程中的“轮”究竟是如何计算的。例如，AES-128，我们之前说过，它要执行10轮加密。但实际的执行过程是这样的：

- 第0轮，把原始数据和原始密钥进行异或；
- 第1-9轮，执行`SubBytes -> ShiftRows -> MixColumns -> AddRoundKey`的完整流程；
- 第10轮，不执行`MixColumns`；

因此实际执行的，是11轮操作。而对于解密算法，就是把`Cipher`的执行倒过来：

```
void InvCipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;
  AddRoundKey(Nr, state, RoundKey);

  for (round = (Nr - 1); round > 0; --round)
  {
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(round, state, RoundKey);
    InvMixColumns(state);
  }

  InvShiftRows(state);
  InvSubBytes(state);
  AddRoundKey(0, state, RoundKey);
}
```

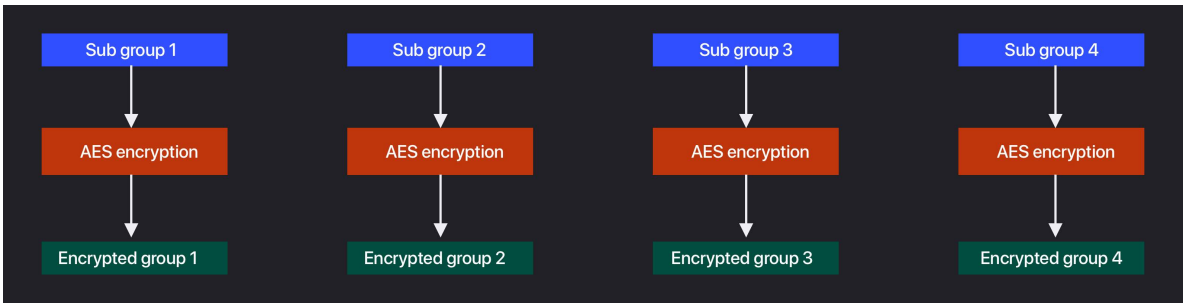
其中的`InvShiftRows / InvSubBytes / InvMixColumns`分别是对应算法的逆运算。大家可以到GitHub去看代码，理解它们并没有难度，我们就不重复了。至此，关于AES加密算法的执行流程，就说完了。不过，只能对定长数据加密始终都是一个缺陷。因此也就有了接下来关于AES加密工作模式的话题。简单来说，就是把不定长的数据，切分成若干定长的数据，再进行加密和解密。

分组密码的工作模式

当我们搜索AES加密应用的时候，经常能看到诸如`AES-256-ECB / AES-256-CBC`这样的名字。它们执行的都是AES加密算法，其中的`ECB / CBC`指的就是它们的工作模式。

ECB

我们先从最简单也最不安全的ECB模式说起。所谓ECB模式，就是直接把原始要加密的数据按照定长分组，然后每一个分组用同样的Key执行加密：



很好理解对不对？但这种模式最大的问题就在于一旦原始数据的存储格式暴露了，攻击者可以无需AES密钥就对加密数据进行攻击。来看个经典的例子，假设银行账号A向账户B转账的信息是通过下面的格式存储的：

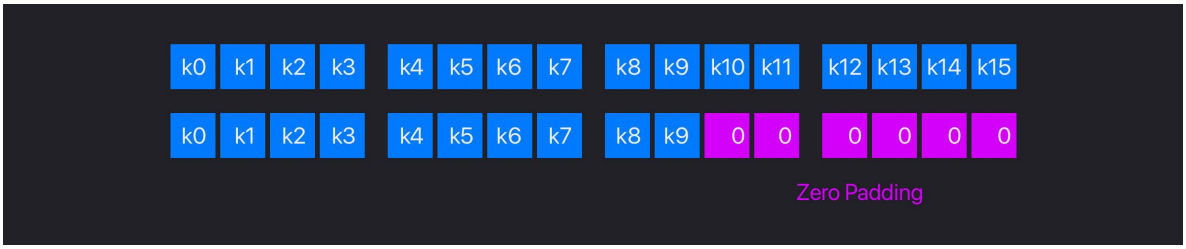
```
账户A: uJb17CO3LSj+SFPSTSRpUg==
账户B: 3HFW8zvoG9f4kB80B+6hrg==
转账金额: T48Km+rTBItaZim/kRvgww==
```

对于这组加密的数据，出于ECB模式的特性，任意改变这三行的顺序，都不会影响解密结果，因为它们的加密解密过程是完全独立的。于是，攻击者只要尝试把数据的前两行换位，这个转账的顺序就变成了账户B向账户A转账，从而达成了在不知道AES密钥的情况下攻击原始数据。因此，除非是为了向既有系统兼容，现如今ECB这种工作模式已经不被使用了。

数据的padding

不过看到这，你可能会想了，怎么那么巧原始数据能分到AES加密数据块的整数倍呢？实际上当然不可能这么巧了。这就涉及到数据的补全问题。我们常见的补全策略，有三种，分别是：zero padding，PKCS #5 padding和PKCS #7 padding。对于PKCS #5和PKCS #7的“身世”，现在我们不用追求太多，它们涉及到和证书相关的内容，等稍后用到RSA加密算法的时候再说。但现在我们可以从编程的角度来理解下这三种填充方法的实现，因为这涉及到AES加密函数的用法。

首先是zero padding，这种方法最简单，在欠缺的位置都填充0就行了。



例如，目标字节是16，当前字节数是10，那么就在结尾补上6个0。例如，图中每一个方块表示1字节，我们要求数据的分组是每16字节分组，对于第二行的情况，就会在末尾补充上6字节的0。这种方法的缺点就是，在解密之后，如果看到结尾包含一个0，我们无法区分究竟是之前填充进来的应该去掉，还是原始数据本身就是0。因此，zero padding通常只用来填充字符串，而不直接用于AES加密算法，因为表示字符串结尾的0本身就是无意义的。

其次，是PKCS #7 padding。它的工作方式是这样的：

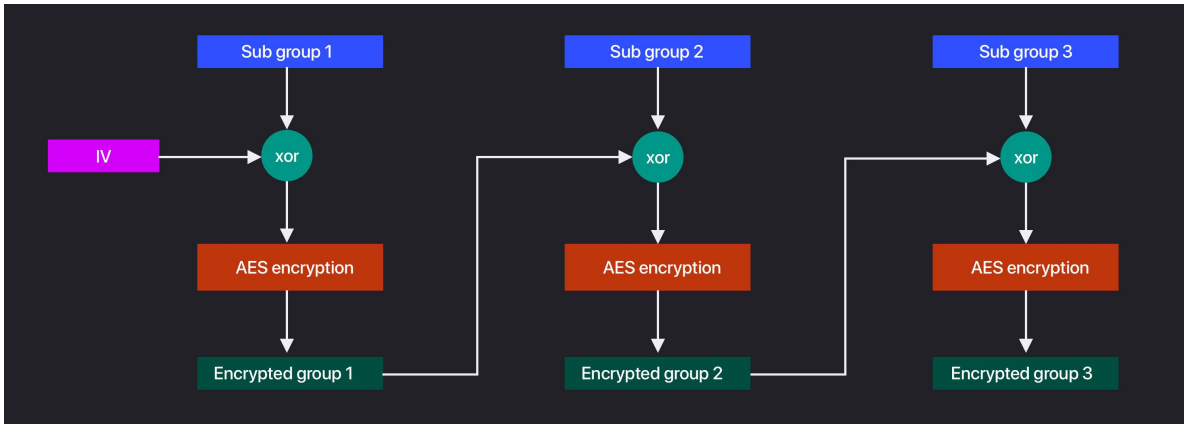
- 假设要求的分块大小是`blockSize`；
- 如果当前数据大小正好是`blockSize`，就在当前数据后面再填充上`blockSize`个值为`blockSize`的字节；
- 如果当前数据尺寸`currSize`小于`blockSize`，就把空缺的部分都填充上`blockSize - (currSize % blockSize)`；

通过这种方式，我们就知道，解密后得到的数据块，最后一个字节一定是填充的，并且从它的值我们还能知道填充进来的字节数，这样也就能还原回原始的值了。这也是AES加密使用的填充方式。

最后，是PKCS #5 padding。从实现上方式上说，它和PKCS #7是一样的。只不过，它只用于填充以8字节为单位的数据块。因此，PKCS #5并不能用于AES加密算法，只能用于DES加密算法。

CBC

了解了数据padding策略之后，我们继续回到AES的工作模式。既然ECB这种直接分组映射的方式不安全，该怎么办呢？一个思路就是让下一个分组的加密和上一个分组的加密相关，有了这种关联性，移动了加密结果之后，就无法成功解密了。把这个思路具体到CBC上，就是这样的：



也就是说，第`i+1`个分组要加密的数据，要先和第`i`个分组的加密结果异或。这样一来，我们就要为第一个分组的加密创建一个和分组长度相同的随机向量来参与异或（对于AES来说，就是个16字节的数组），这个随机向量，叫做`Initial Vector`，也叫做`IV`。

我们不妨来看看它的实现：

```
void AES_CBC_encrypt_buffer(
  struct AES_ctx *ctx, uint8_t* buf, uint32_t length)
{
  uintptr_t i;
  uint8_t *Iv = ctx->Iv;
  for (i = 0; i < length; i += AES_BLOCKLEN)
  {
    XorWithIv(buf, Iv);
    Cipher((state_t*)buf, ctx->RoundKey);
    Iv = buf;
    buf += AES_BLOCKLEN;
  }
  /* store Iv in ctx for next call */
  memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
}
```

这里，唯一要解释一下的就是最后一个`memcpy`的调用，它会保存上一次加密时使用的`Iv`，也就是说，只要在执行AES加密之初设置了`Iv`，接下来所有的加密使用`Iv`就都可以自动生成出来了，我们可以把它看成区块链最原始的形式。

除了ECB和CBC之外，AES还支持CFB / OFB / CTR等工作模式，不过理解了分组加密的想法之后，理解它们并不困难，大家感兴趣的话可以自己去研究，我们就不一一展开了。

What's next?

至此，对于AES加密的执行细节，我们就说的差不多了，从支持这种加密的数学知识，到加密过程的执行细节，再到对不定长数据的扩展方法。下一节，我们就用AES-256-CBC这种方式，来加密APN Provider返回的数据结果。

ELGamal Encryption Schema

ElGamal is a multiplicatively homomorphic encryption scheme.

ELGamal是一种乘法同态加密方案。属于非对称加密，需要公钥加密之后，用私钥解密。

具体的算法信息记录在：https://en.wikipedia.org/wiki/ElGamal_encryption

可交换加密方案

不得不顺便提一下，一些文章抄袭现象相当严重。戳下面👇👇👇👇👇

到底还要不要脸

文章指明道姓作者何人，可是，早于该文章一年，看到一篇一毛一样的英文论文。后被google代码实现。

- 结论

设置固定`（a， b）`以及素数`p`，定义$\mathbf{f}_a = m^a \pmod p$。使$m \in \mathbf{Z}_p$，$a \in \mathbf{Z}_{p-1}$；可知`gcd(a，p - 1) = 1`

则，$\mathbf{f}_a \cdot \mathbf{f}_b（m）= \mathbf{f}_b \mathbf{f}_a（m）$

这就是可交换加密方案的核心算法。

下问将介绍具体推导过程。

拓展欧几里得算法

先来看看一个重要的基本定理

裴蜀定理

对于整数a,b，他们关于x,y的线性不定方程`ax + by = d`，设`gcd(a, b) = g`，则可证明`g | d`，换句话说，就是**g是a,b的最小线性组合。**

设`ax + by = d`，`g = gcd(a, b)`，设`ax + by`的最小值为s，

∵ `g|a`，`g|b`

∴ `g|d`，`g|s`

设`q = [a/s]`。则`r = a mod s = a − q * s = a − q * (ax + by) = a(1 − qx) + b(−qy)`。

可见`r`也是`x,y`的线性组合，又`r`是`s`的余数，`r∈[0, s−1]`，又s是最小线性组合，∴`r=0`。

推出`s|a`，同理有`s|b`，则`s|g`，又已经有`g|s`，所以`g=s`。可知`g`是`ax + by`的最小线性组合。

推论：

`a`和`b`互质的充要条件是存在`x,y`使`ax + by = 1`，因为由上面的证明结论知道`d`（这里是1）一定是`gcd(a, b)`的倍数。

其实还可以推广到一堆数互质（这些数的`gcd`为1）的情况。

拓展欧几里得

我们知道，一般的 欧几里得算法是来求两个数的最大公因数的，但是拓展欧几里得可以走得更远，也就是说，它不仅仅求一个`gcd`，而是可以求出一些额外的东西。前面的定理说了`a,b`线性组合最小是`gcd(a,b)`，我们就可以求出`ax + by = gcd(a, b)`——**贝祖等式**对应的`x,y`出来。

怎么求呢？我们已经知道当欧几里得算法递归到终点时，`gcd(a,b)`中的`b`已经是零，也就是说，这个时候`a * 1 + b * 0 = a`，`x = 1, y = 0`，我们就可以在这时**反推**上去求解最终的`x,y`

现在是递归返回过程的某一步，我们已经求得了`b`和`a%b`的`gcd`，还有此时的`x1, y1`，就是说`b * x1 + a % b * y1 = gcd`，要怎么求`a, b`的`x, y`呢？

我们知道`a % b = a – [a/b]*b`，

`gcd = b * x1 + (a - [a/b] * b) * y1`

`= b * x1 + a * y1 - [a/b] * y1 * b`

`=a * y1 + (x1 – [a/b] * y1) * b`

`=a * x + b * y`

可以看出`x = y1`，`y = x1 – [a/b] * b`.递归返回时即可更新掉`x, y`的值返回即可。

代码：

```
int ex_gcd(int a,int b,int& x,int& y)
{
    if(b==0)
    {
        x=1;
        y=0;
        return a;
    }
    int ans = ex_gcd(b,a%b,x,y);
    int tmp = x;
    x = y;
    y = tmp-a/b*y;
    return ans;
}
```

拓展欧几里得算法演算示例

扩展欧几里得算法（Extended Euclidean algorithm）是欧几里得算法（又叫辗转相除法）的扩展。已知整数a、b，扩展欧几里得算法可以在求得a、b的最大公约数的同时，能找到整数x、y，使它们满足**贝祖等式** $ax + by = gcd(a, b)$

演算过程

求二元一次不定方程 $47x + 30y = 1$ 的整数解。

```
47=30*1+17 // y=1, x=1
30=17*1+13 //
17=13*1+4 // 同上
13=4*3+1 // 同上
```

然后把它们改写成"余数等于"的形式

```
17=47*1+30*(-1)
13=30*1+17*(-1)
4=17*1+13*(-1)
1=13+4*(-3)
```

然后把它们“倒回去”

```
1=13+4*(-3)
1=13+[17*1+13*(-1)]*(-3)
1=17*(-3)+13*4
1= 17*(-3)+[30*1+17*(-1)]*4
1=30*4+17*(-7)
1=30*4+[47*1+30*(-1)]*(-7)
1=47*(-7)+30*11
```

求得 x=-7，y=11。

python

```
def ext_euclid(a, b):
     if b == 0:
         return 1, 0, a
     else:
         x, y, q = ext_euclid(b, a % b) # q = gcd(a, b) = gcd(b, a%b)
         x, y = y, (x - (a // b) * y)
         return x, y, q
```

拓展欧几里得求逆元

什么是逆元?

`a * x ≡ 1 mod m`，这里`x`就是`a`的逆元。

逆元有什么用呢?

如果我们要求`a / b mod m`的值，而`a，b`很大，设`b`的逆元为`x`，

这个时候注意到：

`(a/b) * x * b = (a/b) mod m`

`= a * x mod m`

巧妙地把出发转换成了乘法。

为什么求逆元跟欧几里得算法联系起来了呢?

根据上面裴蜀定理，我们知道gcd是a,b两个数线性组合的最小值，其他组合值都是`gcd`的倍数，当`gcd`为1时，`a，b`互质，满足`ax + by = 1`，

移项得`ax = -by + 1`，即`ax ≡ 1 mod b`，此时的`x`就是`a`的逆元。

实际上线性不定方程组有无穷多解，这里只求正的最小的逆元。

代码

```
int cal(int a,int m)
{
    int x,y;
    int gcd = ex_gcd(a,m,x,y);
    //cout << "a " << a << " m " << m << " x " << x << " y " << y << endl;
    if(1%gcd!=0) return -1;
    x*=1/gcd;
    m = abs(m);
    int ans = x%m;
    if(ans<=0) ans += m;
    return ans;
}
```

这里`1 % gcd`是看`gcd`是不是1，前面说了，

`d`（这里是1）应该是`gcd`的倍数，而且不互质的两个数没有逆元。

`x* = 1 / gcd`实际上更一般的写为`x* = d / gcd`，

也就是求解一般不定方程`ax + by = d`的解，因为`d`是`gcd`的倍数，我们就把倍数乘上去解得`x' = x * (d/gcd)`。

`m`是负数的话，我们取`|m|`，如果求出来`x`是负数，就`x % |m|`，结果再加上`|m|`即可。（因为有无穷个解，通解为`x + m * t`

参考文章

zhj5chengfeng，扩展欧几里德算法详解，https://blog.csdn.net/zhjchengfeng5/article/details/7786595

leader_one，欧几里得算法/扩展欧几里得算法，https://blog.csdn.net/leader_one/article/details/75222771

Jollwish，扩展欧几里得算法，https://wenku.baidu.com/view/a75fdfd376a20029bd642de5.html