File Blocking Algorithm

1. 算法简析

分布式文件系统中，因文件丢失易导致文件无法恢复的问题。相较于传统全备份方式，本算法采用文件编码方式，以较低的备份冗余度、一定的计算消耗，来实现动态、高容错性的文件恢复。

2. 算法基础

2.1 编码恢复数学原理【网络编码方式】

- 编码过程：A 为编码矩阵（目前以 5*3 为例）；B 为数据块；C 为编码后的数据

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \\ A_{51} & A_{52} & A_{53} \end{bmatrix} \otimes \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \\ C_{41} & C_{42} & C_{43} \\ C_{51} & C_{52} & C_{53} \end{bmatrix}$$

- 译码过程：（假设只取回 C_{1i}、C_{3i} 、C_{5i} 三个编码数据块）。**只要 A 矩阵满秩，则一定可以得到唯一解，即正确恢复原数据** B 。

$$\begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{31} & A_{32} & A_{33} \\ A_{51} & A_{52} & A_{53} \end{bmatrix}^{-1} \otimes \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{31} & C_{32} & C_{33} \\ C_{51} & C_{52} & C_{53} \end{bmatrix}$$

3. 有限域

3.1 群、环、域

一组元素的集合，以及在集合上的四则运算，构成一个域。其中加法和乘法必须满足交换、结合和分配的规律。加法和乘法具有封闭性，即加法和乘法结果仍然是域中的元素。

域中必须有加法单位元和乘法单位元，且每一个元素都有对应的加法逆元和乘法逆元。但不要求域中的 0 有乘法逆元。

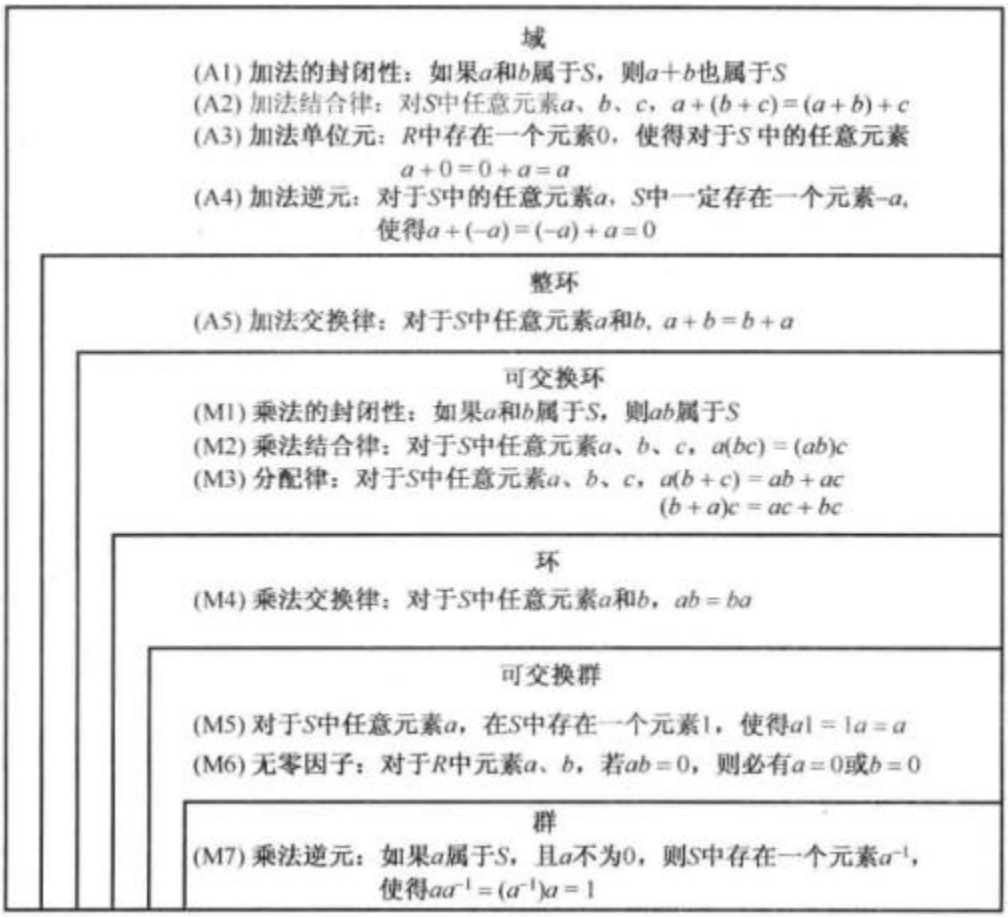


图 4.2 群、环和域的公理

3.2 有限域

有理数 Q、复数 C，以及其它的一些数域都满足这些条件，它们都是域。不过这些域都是无限域，因为信息科学领域里用到的域，都是有限域。

在数学上，有限域（finite field）是包含有限个数的域，有限域是进行加减乘除运算都有定于，并且满足特定规则的集合。有限域的乘法群是循环群，即若 F 是有限群，则存在 $\alpha \in F$，使得 $F^* = \{x \in F | x \neq 0\} = \langle \partial \rangle$ 。

因为在数据传输过程，如果选择实数域的矩阵运算，数据范围会很大，网络传输不方便，因此一般采用有限域的矩阵运算。

有限域 GF(p)，其中 p 为素数。GF(p)里面的加法和乘法与一般的加法和乘法差不多，区别是结果需要 mod p，以保证结果都是域中的元素。**GF(p)的加法和乘法单位元分别是 0 和 1**。

GF(p)加法是(a+b) mod p，乘法是(a*b)mod p。对于域中的乘法，当 p 为素数时，才能保证集合中的所有的元素都有乘法逆元(0 除外)。即对于域中的任一个元素 a，总能在域中找到另外一个元素 b，使得 a*b mod p 等于 1。

说明：假如 p 等于 10，其乘法单位元为 1。对于元素 2，找不到一个数 a，使得 2*a mod 10 等于 1，即 2 没有乘法逆元。这时，在域上就不能进行除 2 运算。

3.2.1 单位元、逆元

单位元

通常使用 e 来表示单位元。单位元和其他元素结合时，并不会改变那些元素。对于二元运算*，若 a*e=a，e 称为右单位元；若 e*a=a，e 称为左单位元，若 a*e=e*a=a，则 e 称为单位元。

逆元

对于二元运算*，若 a*b=e，则 a 称为 b 的左逆元素，b 称为 a 的右逆元素。若 a*b=b*a=e，则称 a 为 b 的逆元，b 为 a 的逆元。

3.3 有限域 GF(q^m)

一类最简单的有限域：Z_p，它的意思是整数集合 Z 模 p 得到的同余类，其中 p 为素数。简单点说，就是包含 p 个数的集合$\{0, 1, 2, \dots, p\text{-}1\}$，只不过这些数进行加法和乘法运算时需要模 p，保证算完了还在这个集合里面。

注意这里 p 一定要是素数，如果是合数 n，Z_n 就只是一类环，不是域了

其次，为了得到更多的有限域，需要一个定理做支撑：

◆ **定理7** 多项式模首一多项式$p(x)$环成为域的充分必要条件是$p(x)$为素多项式。

按照这个定理，**给定一个已知的有限域 GF(q)，找到这个有限域上的一个 m 次素多项式，我们就能构造出一个新的有限域，这个有限域包含的元素个数为 q^m，也就是得到了一个 GF(q^m)**。根据第一条，我们已经有了一类有限域：Zp，其中 p 为素数。根据第二条，为了得到更多的有限域，我们需要寻找有限域 Zp 上的素多项式。

按照定义，素多项式是首项系数为 1 的不可约多项式，那么只要找到不可约多项式，除以首项系数，就得到素多项式了。

3.4 多项式

3.4.1 素多项式/本原多项式

本原多项式 （primitive polynomial）是一种特殊的不可约多项式。当一个域上的本原多项式确定了，这个域上的运算也就确定了。通过将域中的元素化为多项式形式，可以将**域上的乘法运算转化为普通的多项式乘法再模本原多项式**。

部分 GF（2^w）域经常使用的本原多项式如下:

$$
\begin{aligned}
w = 4: &\quad x^4 + x + 1 \\
w = 8: &\quad x^8 + x^4 + x^3 + x^2 + 1 \\
w = 16: &\quad x^{16} + x^{12} + x^3 + x + 1 \\
w = 32: &\quad x^{32} + x^{22} + x^2 + x + 1 \\
w = 64: &\quad x^{64} + x^4 + x^3 + x + 1
\end{aligned}
$$

3.4.2 素多项式运算

指数小于 3 的多项式有 8 个，分别是 0， 1， x， x+1， x^2， x^2+1， x^2＋x，x^2+x+1。对于 GF(2^3)来说，其中一个素多项式为 x^3+x+1。**上面 8 个多项式进行四则运算后 mod (x^3+x+1)的结果都是 8 个之中的某一个，可以证明这是一个域，**所以每一个多项式都是有加法和乘法逆元的(0 除外)。注意，这些逆元都是和素多项式相关的，同一个多项式，取不同的素多项式，就有不同的逆元多项式。

对于 GF(2^8)，其中一个素多项式为 x^8 + x^4 + x^3 +x +1。对应地，小于 8 次的多项式有 256 个。

由素多项式得到的域，其加法单位元都是 0，乘法单位元是 1。

前面讲到了对素多项式取模，然后可以得到一个域。但这和最初的目的有什么关系吗？多项式和 0， 1， ⋯⋯，255 没有什么关系。确实是没有什么关系，但多项式的系数确可以组成 0， 1， 2，⋯⋯255 这些数。回到刚才的 GF(2^3),对应的 8 个多项式，其系数刚好就是 000,001, 010, 011, 100, 101, 110, 111。这不正是 0 到 7 这 8 个数的二进制形式吗？也就是说，它们有一一对应映射的关系。**多项式对应一个值**，我们可以称这个值为多项式值。

对于 GF(2^3)，取素多项式为 x^3 + x+1，那么多项式 x^2+x 的乘法逆元就是 x+1。系数对应的二进制分别为 110 和 011。此时，我们就认为对应的十进制数 6 和 3 互为逆元。即使 mod 8 不能构成一个域，但通过上面的对应映射，0 到 7 这 8 个数一样有对应逆元了(为了顺口，说成 0 到 7。实际 0 是没有乘法逆元的)。同样，对于 GF(2^8)也是一样的。所以 0 到 255，这 256 个数都可以通过这样的方式得到乘法逆元(同样，0 是没有乘法逆元的)。

3.4.3 加法/减法【亦或】

合并同类项时，系数们进行异或操作，不是平常的加法操作。比如 x^4 + x^4 等于 0*x^4。因为两个系数都为 1， 进行异或后等于 0。

无所谓的减法(减法就等于加法)，或者负系数。所以，x^4 – x^4 就等于 x^4 + x^4。-x^3 就是 x^3

3.4.4 乘法/除法

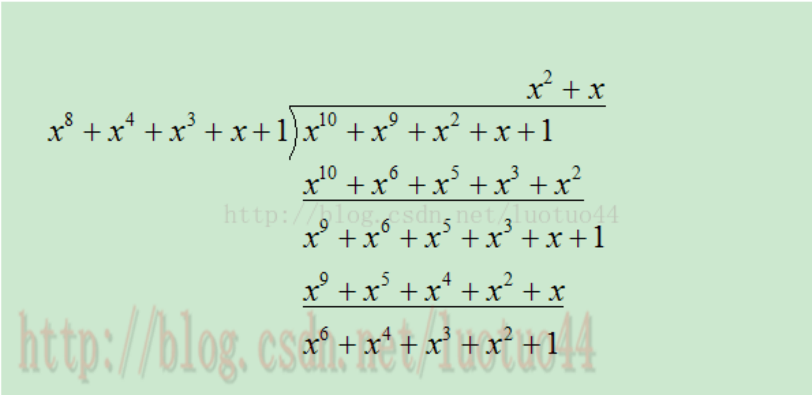
(F(x)*G(x))mod m(x) ： m(x)为本原多项式

看一些例子吧。对于f(x) = x^6 + x^4 + x^2 + x + 1。g(x) = x^7 + x + 1。

那么f(x) + g(x) = x^7 + x^6 + x^4+ x^2 + (1+1)x + (1+1)1 = x^7 + x^6 + x^4 + x^2。f(x) – g(x)等于f(x) + g(x)。

f(x) * g(x) =(x^13 + x^11 + x^9 + x^8 + x^7) + (x^7 + x^5 + x^3 + x^2 + x) + (x^6+ x^4 + x^2 + x + 1) = x^13 + x^11 + x^9 + x^8 + x^6 + x^5+ x^4+ x^3+1。

下图是除法，除法得到的余数，也就是mod操作的结果。



3.5 查表

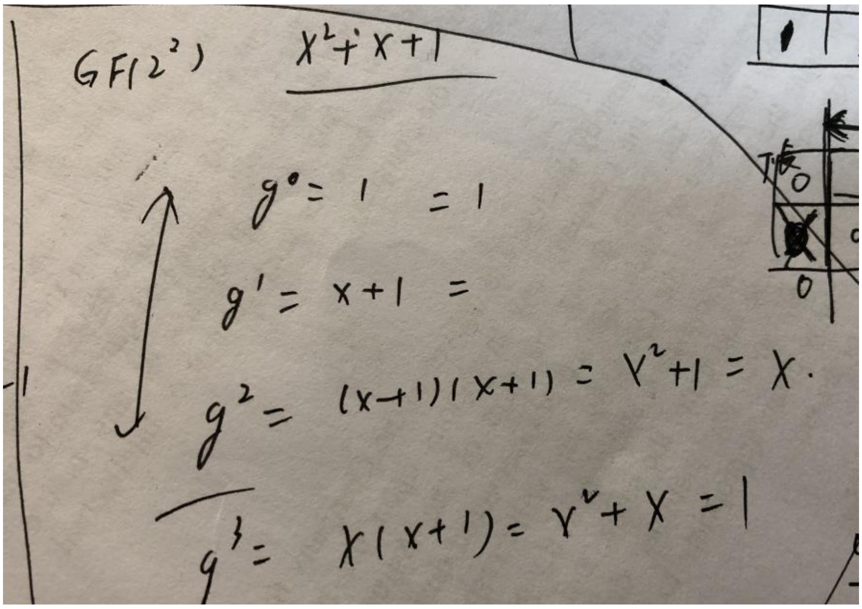
3.5.1 单位元

如果元素 g 满足下面的条件，我们就称 g 为生成元：对于集合中的任何的一个元素，都可以通过元素 g 的幂 g^k 得到。并定义 g^0 = e，假设 h 为 g 的逆元，那么还定义 g^(-k) = h^k。比如，整数集合，都可以由生成元 1 得到。2 = 1 + 1 = 1^2、3 = 1^3=1 + 1 + 1、……。负数可以通过幂取负数得到。

对于 g^k = a，有正过程和逆过程。知道 k 求 a 是正过程，知道了 a 反过来求 k 是逆过程。同样，假设有 g^n = a 和 g^m = b。现在需要求 a*b，那么就有 a*b = g^n* g^m = g^(n+m)。我们只需要：根据 a 和 b，分别求得 n 和 m。然后直接计算 g^(n+m) 即可。这里，构造两个表，正表和反表。正表是知道了指数，求值。反表是知道了值，求指数。接下来要做的就是构造这两个表。为了做除法运算，还要构造逆元表。

假设 g 是域 GF(2^w)上生成元，那么集合 {g0 ，g1 ， ……，g(2^w-1)} 包含了域 GF(2^w)上所有非零元素。**在域 GF(2^w)中 2 总是生成元。**

GF(2^w)是一个有限域，就是元素个数是有限的，但指数 k 是可以无穷的。所以必然存在循环。这个循环的周期是 2^w-1（g 不能生成多项式 0）。所以当 k 大于等于 2^w-1 时，g^k =g^(k%(2^w-1))。



3.5.2 正表反表构建

对于正表，生成元的指数，取 0 到 254 即可，对应地生成 255 个不同的多项式，多项式的取值范围为 1 到 255。

对于正表，只需依次计算 g^0、g^1、g^2，……，g^254 即可。对于 GF(2^8)，素多项式 m(x) = x^8 + x^4 + x^3 +x +1，对应的生成元 g(x) = x + 1。

```
int table[256];
int i;

table[0] = 1;//g^0
for(i = 1; i < 255; ++i)//生成元为x + 1
{
        //下面是m_table[i] = m_table[i-1] * (x + 1)的简写形式
        table[i] = (table[i-1] << 1 ) ^ table[i-1];

        //最高指数已经到了8，需要模上m(x)
        if( table[i] & 0x100 )
        {
                table[i] ^= 0x11B;//用到了前面说到的乘法技巧
        }
}
```

反表和正表是对应的，所以反表中元素的个数也是 255 个。正表中，生成元 g 的

指数 k 的取值范围为 0 到 254。多项式值 g^k 的取值范围为 1 到 255。所以在反表中，下标的取值范围为 1 到 255，元素值的取值范围为 0 到 254。

```
int arc_table[256];

for(i = 0; i < 255; ++i)
        arc_table[ table[i] ] = i;
```

对于逆元表，先看逆元的定义。若 a 和 b 互为逆元，则有 a*b = e。用生成元表示为：g^n* g^m = e = 1。又因为 e = g^0 = g^255(循环，回头了)。所以 g^k * g(255-k) = g^(k + 255 -k) = e。于是 g^k 和 g^(255-k)互为逆元。对于多项式值 val，求其逆元。可以先求 val 对应的 g 幂次是多少。即 g 的多少次方等于 val。可以通过反向表查询，设为 k。那么其逆元的幂次为 255-k。此时再通过正向表查询即可。

```
int inverse_table[256];

for(i = 1; i < 256; ++i)//0没有逆元，所以从1开始
{
        int k = arc_table[i];
        k = 255 - k;
        k %= 255;//m_table的取值范围为 [0, 254]
        inverse_table[i] = table[k];
}
```

3.5.3 查表计算

见附 C 语言实现

4. 译码

4.1 高斯消元

首先举一个例子：求解如下方程组：

$$\begin{cases} x-2y+3z=6 \text{..........①} \\ 4x-5y+6z=12 \text{.......②} \\ 7x-8y+10z=21 \text{.....③} \end{cases}$$

我们手算一下这个方程组，过程如下：

1. ② - 4*①；③ - 7*①，得到如下式子：

$$\begin{cases} x-2y+3z=6............① \\ 0x+3y-6z=-12.......② \\ 0x-6y-11z=-21.....③ \end{cases}$$

2. ②两边同除以3，得到：

$$\begin{cases} x-2y+3z=6............① \\ 0x+y-2z=-4..........② \\ 0x-6y-11z=-21.....③ \end{cases}$$

3. ① - ②*(-2)；③-②*6，得到：

$$\boxed{\begin{cases} x+0y-z=-2............① \\ 0x+y-2z=-4..........② \\ 0x+0y+z=3..............③ \end{cases}}$$

4. 从③式得到 $z=3$，再代入②式得到 $y=2$，再代入到①式得到 $x=1$

5. 算法实现

5.1 系数矩阵寻找

保留

5.2 编码有限域计算

```
//加法
unsigned char uf_ywj_add(unsigned char a, unsigned char b)
{
    return a ^ b;
}

//乘法：
unsigned char  uv_ywj_mul(unsigned char a, unsigned char b)
{
    if (a && b)
        return uv_ywj_Alogtable[(uv_ywj_Logtable[a] + uv_ywj_Logtable[b]) % 255];
    else return 0;
}
```

```c
//除法实现
unsigned char uf_ywj_divi(unsigned char a, unsigned char b)
{
    int j;
    if (a == 0)
        return (0);
    if ((j = uv_ywj_Logtable[a] - uv_ywj_Logtable[b]) < 0)
        j += 255;
    return (uv_ywj_Alogtable[j]);
}
//逆元
unsigned char uf_ywj_inv(unsigned char in)
{
    /* 0 is self inverting */
    if (in == 0)
        return 0;
    else
        return uv_ywj_Alogtable[(255 - uv_ywj_Logtable[in])];
}
```

5.3 高斯消元

【见附录 c-code】