基于二进制矩阵的纠删码简单介绍

1、背景

当前主流的分布式存储系统都是通过数据多副本的方式保证数据高可靠，当有节点宕机或者存储设备损坏，可以副本中获取数据保证数据高可靠。

比如我有一部电影，多副本的方式以什么样的姿势存入到分布式存储系统中，如图1：

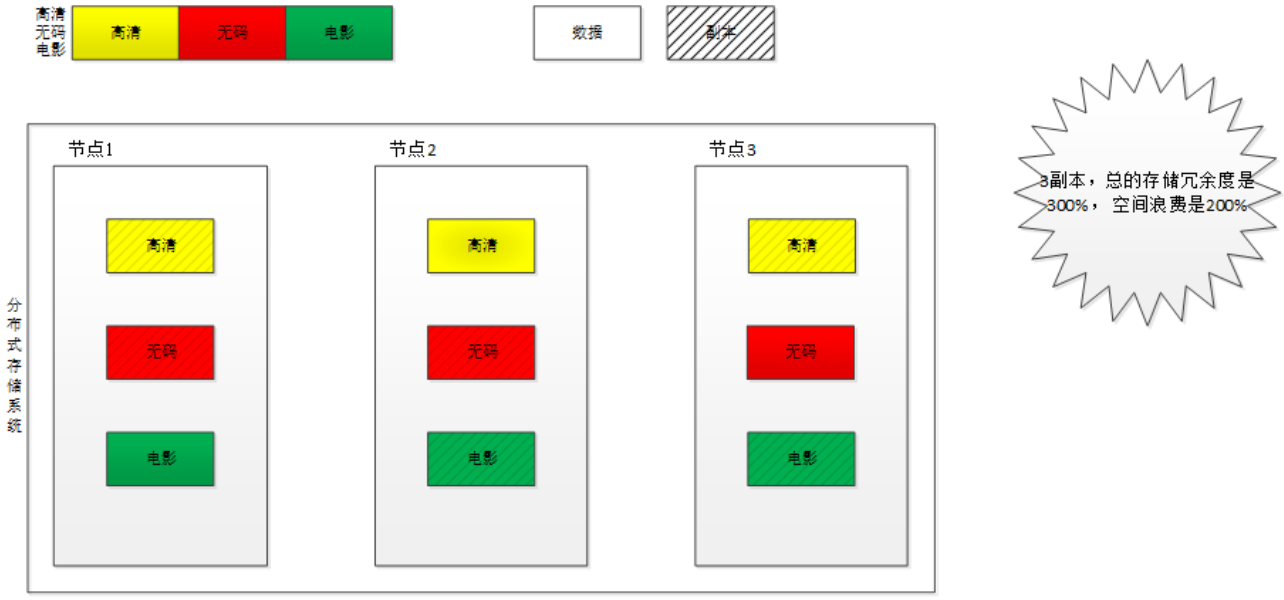


图1

多副本的优点是容错性提高了，但是空间利用率太低；像平安包头项目采用两副本的机制，导致占用太多存储空间，存储服务器的数量增加了一倍，增加了设备购买成本。

为了提供空间利用率，很多存储系统选择的纠删码的存储方案。纠删码的方式以什么样的姿势存入到分布式存储系统中，如图2：

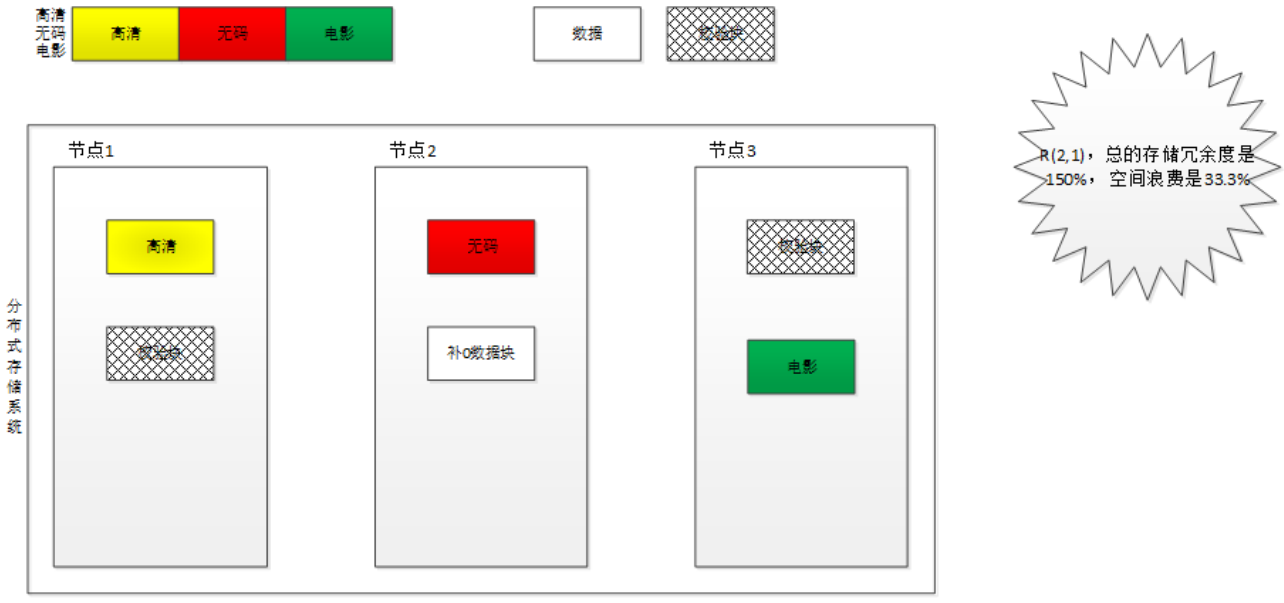


图2

2、编码的一般形式

比如实现3+1的冗余策略，如式1，丢失任何一个块数据d_i（i=1，2,3）都可以通过y1和其他数据块进

行运算恢复

$$d_1+d_2+d_3=y_1$$

实现3+2的冗余策略，如式2：

$$d_1+d_2+d_3=y_1$$

$$d_1+2d_2+4d_3=y_2$$

同样可以扩展到实现k+m的冗余，如图3：

$$y_1=d_1+d_2\cdot 1+d_3\cdot 1^2+..+d_k\cdot 1^{k-1}$$

$$y_2=d_1+d_2\cdot 2+d_3\cdot 2^2+..+d_k\cdot 2^{k-1}$$

$$y_3=d_1+d_2\cdot 3+d_3\cdot 3^2+..+d_k\cdot 3^{k-1}$$

$$\cdots$$

图3

将图3的方程组写成矩阵的形式[Y]=[E][D]，如图4：

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1^2 & \cdots & 1^{k-1} \\ 1 & 2 & 2^2 & \cdots & 2^{k-1} \\ 1 & 3 & 3^2 & \cdots & 3^{k-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & m & m^2 & \cdots & m^{k-1} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdots \\ d_k \end{bmatrix}$$

方程组系数构成的矩阵就是著名的范德蒙德矩阵，范德蒙德矩阵的特征就是任意子矩阵都是可逆的，矩阵保证了任意m行m列组成的子矩阵都是线性无关的，构成的方程肯定有确定解，有同等性质的还有柯西矩阵。

例如 d2,d3 丢失了，下面用 u2,u3 表示（只丢失了2块数据，不需要所有的m个校验块参与，只需要2个校验块来恢复数据），

$$\begin{bmatrix} 1 & 2 & 4 & \cdots & 2^{k-1} \\ 1 & 3 & 9 & \cdots & 3^{k-1} \end{bmatrix} \times \begin{bmatrix} d_1 \\ u_2 \\ u_3 \\ \cdots \\ d_k \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \end{bmatrix}$$

矩阵表示的方程组里有2个未知数 u2,u3，解方程即可得到 u2,u3 这2块丢失的数据。

3、有限域运算

通过范德蒙德矩阵生成的编码矩阵和解码矩阵可以得到，矩阵中的元素可能会非常大，导致数据的乘积和加法运算超过计算机的表示范围。

将实数域的的运算同构到伽罗瓦域，将四则运算结果限定在伽罗瓦域中，可以将实数域的乘积和加法的结果限定到特定的范围，乘法运算查表，加法运算异或，比如GF(2^3)的伽罗瓦域运算：

（1）GF(2^3)本原多项式：

$$q(x)=x^3+x+1$$

（2）GF(2^3)元素表：

生成元素	多项式	二进制	十进制
0	0	000	0
x^0	1	001	1
x^1	x	010	2
x^2	x^2	100	4
x^3	x+1	011	3
x^4	x^2+ x	110	6
x^5	x^2+ x +1	111	7
x^6	x^2+1	101	5
x^7	1	001	1
x^8	x	010	2
x^9	x^2	100	4
x^{10}	x+1	011	3
x^{11}	x^2+ x	110	6
x^{12}	x^2+ x +1	111	7

（3）运算样例：

4 + 5 = x^2 + x^2 + 1 = 1

4 * 5 = x^2 * (x^2 + 1) = x^4 + x^2 = x^2 + x^2 + x = x = 2

4、二进制矩阵

通过引入伽罗瓦域可以将乘积和加法的运算结果限定到一个有限域中，但是编码和解码的运算还是需要进行伽罗瓦域的乘积运算，虽然可以通过查表解决乘积运算和加法运算的复杂度，但是有没有一种方法可以将乘积运算全部转换为计算机可以直接操作的指令运算，二进制矩阵的引入可以解决以上问题。

二进制矩阵的运算引入和实数域的的运算同构到伽罗瓦域类似，将伽罗瓦域的乘积运算同构到二进制矩阵运算上，将伽罗瓦域的乘积运算和加法运算变为二进制的Bit位的与运算和异或运算，比如GF(2^3)的伽罗瓦域的运算可以用下图表示：

$$\begin{pmatrix}0&0&0\\0&0&0\\0&0&0\end{pmatrix}\quad\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\quad\begin{pmatrix}0&0&1\\1&0&1\\0&1&0\end{pmatrix}\quad\begin{pmatrix}1&0&1\\1&1&1\\0&1&1\end{pmatrix}$$

0　　1　　2　　3

$$\begin{pmatrix}0&1&0\\0&1&1\\1&0&1\end{pmatrix}\quad\begin{pmatrix}1&1&0\\0&0&1\\1&0&0\end{pmatrix}\quad\begin{pmatrix}0&1&1\\1&1&0\\1&1&1\end{pmatrix}\quad\begin{pmatrix}1&1&1\\1&0&0\\1&1&0\end{pmatrix}$$

4　　5　　6　　7

$GF(2^3)$: 4 * 5 = 2

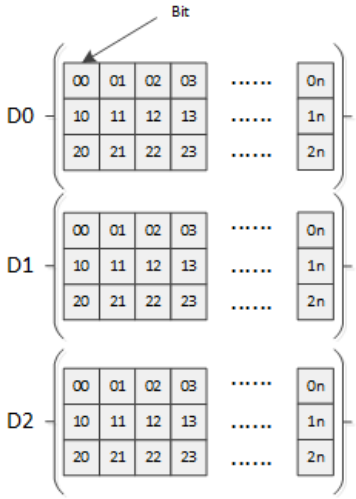
Bit_Matrix : $\begin{pmatrix}0&1&0\\0&1&1\\1&0&1\end{pmatrix} * \begin{pmatrix}1&1&0\\0&0&1\\1&0&0\end{pmatrix} = \begin{pmatrix}0&0&1\\1&0&1\\0&1&0\end{pmatrix}$

根据同构原理，同样RS（3，2）的运算可以同构到二进制的矩阵运算，运算如下：

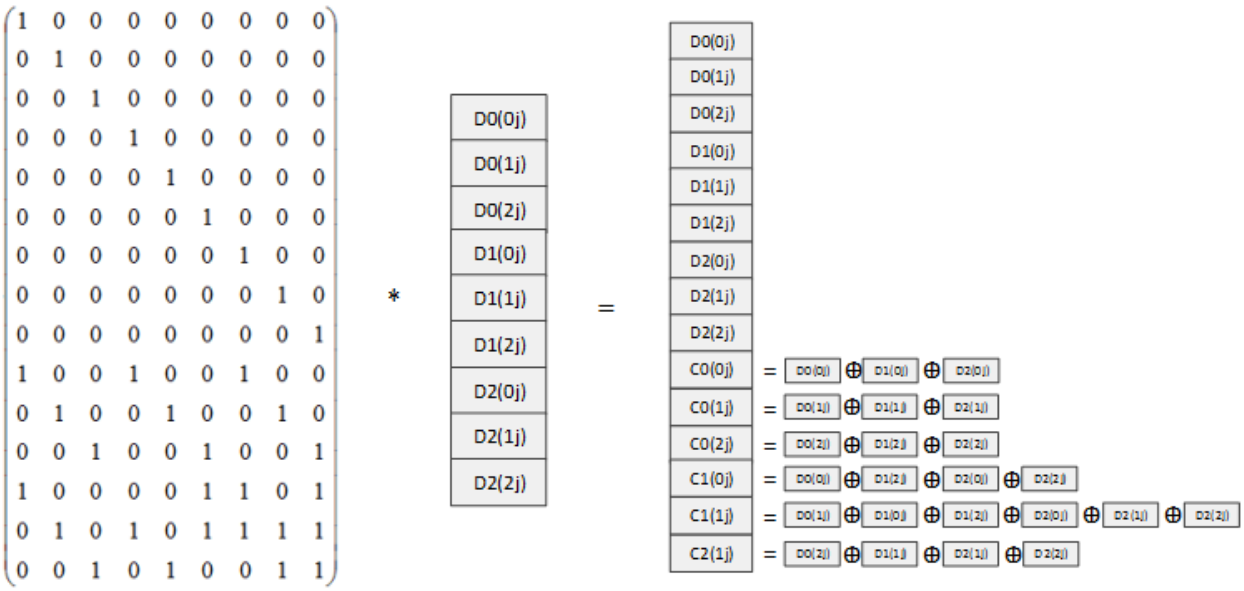
（1）生成矩阵：

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\\1&1&1\\1&2&3\end{pmatrix} \xrightarrow{\text{二进制变换}} \begin{pmatrix}1&0&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0&1\\1&0&0&1&0&0&1&0&0\\0&1&0&0&1&0&0&1&0\\0&0&1&0&0&1&0&0&1\\1&0&0&0&0&1&1&0&1\\0&1&0&1&0&1&1&1&1\\0&0&1&0&1&0&0&1&1\end{pmatrix}$$

（2）数据块结构：



（3）矩阵运算：



（4）扩展：

以上运算单位都是Bit运算，同样可以扩展到Byte的运算，最终可以扩展到数据块之间的异或运算，比如

C0[0] = D0[0]　　D1[0]　　D2[0]

C0[1] = D0[1]　　D1[1]　　D2[1]

C0[2] = D2[2]　　D2[2]　　D2[2]

C1[0] = D0[0]　　D1[2]　　D2[0]　　D2[2]

C1[1] = D0[0]　　D1[0]　　D1[2]　　D2[0]　　D2[1]　　D2[2]

C1[2] = D0[2]　　D1[1]　　D2[1]　　D2[2]

（5）约束条件：

R(n,m,w)，n是数据块个数据，m是校验块个数，w是二进制矩阵的bit个数，

$$n+m \le 2^w$$

4、R(4，2)的编码和解码矩阵

（1）选择柯西矩阵：

[1　1　1　1]

[1　2　9　4]

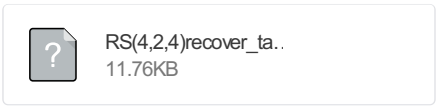
（2）二进制编码矩阵：

```
1 0 0 0   1 0 0 0
0 1 0 0   0 1 0 0
0 0 1 0   0 0 1 0
0 0 0 1   0 0 0 1

1 0 0 0   0 0 0 1
0 1 0 0   1 0 0 1
0 0 1 0   0 1 0 0
0 0 0 1   0 0 1 0
                  :
1 0 0 0   1 1 0 0
0 1 0 0   0 0 1 0
0 0 1 0   0 0 0 1
0 0 0 1   1 0 0 0

1 0 0 0   0 0 1 0
0 1 0 0   0 0 1 1
0 0 1 0   1 0 0 1
0 0 0 1   0 1 0 0
```

（3）求解数据失效解码矩阵：

```
D0 D1失效 以这个为准

1 0 0 0   1 0 0 0      1 0 0 0   0 0 0 0
0 1 0 0   0 1 0 0      0 1 0 0   0 0 0 0
0 0 1 0   0 0 1 0      0 0 1 0   0 0 0 0
0 0 0 1   0 0 0 1      0 0 0 1   0 0 0 0

1 0 0 0   0 0 0 1      0 0 0 0   1 0 0 0
0 1 0 0   1 0 0 1      0 0 0 0   0 1 0 0
0 0 1 0   0 1 0 0      0 0 0 0   0 0 1 0
0 0 0 1   0 0 1 0      0 0 0 0   0 0 0 1
                  ===>>>
1 0 0 0   1 1 0 0      0 1 0 0   1 1 0 0
0 1 0 0   0 0 1 0      0 1 1 0   0 0 1 0
0 0 1 0   0 0 0 1      0 0 1 1   0 0 0 1
0 0 0 1   1 0 0 0      1 0 0 1   1 0 0 0

1 0 0 0   0 0 1 0      0 0 0 1   1 0 0 1
0 1 0 0   0 0 1 1      1 0 0 1   1 1 0 1
0 0 1 0   1 0 0 1      0 1 0 0   0 1 1 0
0 0 0 1   0 1 0 0      0 0 1 0   0 0 1 1
----------------      ----------------
^         ^           ^         ^
|         |           |         |
归        |           |         归
零                              零

1 0 0 0   0 0 0 0      1 1 1 1   0 1 1 1
0 1 0 0   0 0 0 0      1 0 0 0   1 1 0 0
0 0 1 0   0 0 0 0      1 1 0 0   1 1 1 0
0 0 0 1   0 0 0 0      1 1 1 0   1 1 1 1

0 0 0 0   1 0 0 0      0 1 1 1   0 1 1 1
0 0 0 0   0 1 0 0      1 1 0 0   1 1 0 0
0 0 0 0   0 0 1 0      1 1 1 0   1 1 1 0
0 0 0 0   0 0 0 1      1 1 1 1   1 1 1 1
```

其他数据块失效的解码矩阵类似



4、对比测试：

（1） 硬件环境：

```
[root@localhost ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Stepping:              1
CPU MHz:               1201.593
BogoMIPS:              4195.20
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
[root@localhost ~]# free -m
              total        used        free      shared  buff/cache   available
Mem:          64026       18291        4266        1299       41469       43926
Swap:          6107           3        6104
[root@localhost ~]#
```

（2）Jerasure库测试结果：

```
[root@localhost Examples]# ./encoder '/tmp/ECTEST/ec_oringial_data_16M.jpg' 4 2 'cauchy_orig'  8 2048 262144; rm -f Coding/ec_oringial_data_16M_k[2]*; ./decoder "ec_oringial_data_16M.jpg"
encode total:100 encode_size:64K, encode_count:4, total_cost_time:0.0268871021.
Encoding (MB/sec): 569.4333910328
En_Total (MB/sec): 5.8832831888
decode total:100 decode_size:64K, decode_count:64, total_cost_time:0.0196411610.
Decoding (MB/sec): 755.6623727592
De_Total (MB/sec): 8.0263708600

[root@localhost Examples]# ./encoder '/tmp/ECTEST/ec_oringial_data_16M.jpg' 4 2 'cauchy_orig'  8 2048 4194304; rm -f Coding/ec_oringial_data_16M_k[2]*; ./decoder "ec_oringial_data_16M.jpg"
encode total:100 encode_size:1024K, encode_count:4, total_cost_time:0.0278804374.
Encoding (MB/sec): 564.2719582948
En_Total (MB/sec): 5.6781261105
decode total:100 decode_size:1024K, decode_count:4, total_cost_time:0.0151355696.
Decoding (MB/sec): 1036.7505638807
De_Total (MB/sec): 10.3862242689

[root@localhost Examples]# ./encoder '/tmp/ECTEST/ec_oringial_data_16M.jpg' 4 2 'cauchy_orig'  8 2048 16777216; rm -f Coding/ec_oringial_data_16M_k[2]*; ./decoder "ec_oringial_data_16M.jpg"
encode total:100 encode_size:4096K, encode_count:4, total_cost_time:0.0302966905.
Encoding (MB/sec): 596.1046376322
En_Total (MB/sec): 5.2140989883
decode total:100 decode_size:4096K, decode_count:1, total_cost_time:0.0160905790.
Decoding (MB/sec): 1085.6404432581
De_Total (MB/sec): 9.7382475530
```

（3）二进制矩阵测试结果：

```
[root@localhost my_ec_test]# make clean;make;./my_encode "/tmp/ECTEST/ec_oringial_data_16M.jpg"; ./my_decode "ec_oringial_data_16M.jpg"
rm -f *.o my_encode my_decode
gcc    -g  -o my_encode my_encode.c -lpthread
gcc    -g  -o my_decode decode_my.c -lpthread
encode loop_count:100 encode_size:4M, ec_unit:1024K, encode_count:4, cost_time:0.5731439590, arvage_time:0.0057314396.
encode loop_count:100 encode_size:4M, ec_unit:512K, encode_count:4, cost_time:0.5766329765, arvage_time:0.0057663298.
encode loop_count:100 encode_size:4M, ec_unit:256K, encode_count:4, cost_time:0.4964790344, arvage_time:0.0049647903.
encode loop_count:100 encode_size:4M, ec_unit:128K, encode_count:4, cost_time:0.5166449547, arvage_time:0.0051664495.
encode loop_count:100 encode_size:4M, ec_unit:64K, encode_count:4, cost_time:0.6633400917, arvage_time:0.0066334009.
encode loop_count:100 encode_size:4M, ec_unit:32K, encode_count:4, cost_time:0.8180670738, arvage_time:0.0081806707.
decode loop_count:100 recover_size:4M,  ec_unit:1024K, recover_count:1, cost_time:0.2620270252, arvage_time:0.0026202703.
decode loop_count:100 recover_size:4M,  ec_unit:512K, recover_count:1, cost_time:0.2594830990, arvage_time:0.0025948310.
decode loop_count:100 recover_size:4M,  ec_unit:256K, recover_count:1, cost_time:0.2282691002, arvage_time:0.0022826910.
decode loop_count:100 recover_size:4M,  ec_unit:128K, recover_count:1, cost_time:0.2254660130, arvage_time:0.0022546601.
decode loop_count:100 recover_size:4M,  ec_unit:64K, recover_count:1, cost_time:0.2710690498, arvage_time:0.0027106905.
decode loop_count:100 recover_size:4M,  ec_unit:32K, recover_count:1, cost_time:0.3864719868, arvage_time:0.0038647199.
```

（4）表格：

开源库测试	RS(4,2),条带大小4M		失效一个条带
	一次处理数据大小	编码时间	解码时间
	64K	0.02688s	0.01964s
	1M	0.02788s	0.01513s
	4M	0.03029s	0.01609s
二进制矩阵	RS(4,2),条带大小4M		失效一个条带
	一次处理数据大小	编码时间	解码时间
	32K	0.00818s	0.00386s
	64K	0.00663s	0.00271s
	128K	0.00516s	0.00225s
	256K	0.00496s	0.00228s
	512K	0.00576s	0.00259s
	1M	0.00573s	0.00262s

（5）测试代码：

Jerasure-1.2_change.rar
145.89KB

my_ec_test.rar
1.12MB

5、讨论

（1）编码单元说明：

由于二进制矩阵是一个4×4的bit矩阵，所以需要一个编码单元能够被4整除，这个编码单元对应传

统RAID上的条带。

（2）如何求解任意n+m的解码矩阵

从上述将二进制矩阵对角化的方案可以看到，手工运算量很大，需要通过程序计算将bit矩阵进行对角化。这里提供一种对角化的思路：

（a）伽罗瓦域的运算能够同构到二进制矩阵运算域中，那么同样二进制矩阵的对角化也可以同构到伽罗瓦域中，

（b）将需要进行对角化的二进制子矩阵对应的伽罗瓦域的值，通过矩阵初等变换变为1，伽罗瓦域的1对应的也是二进制矩阵中单位矩阵，同样消原也可以在伽罗瓦域中进行。

（c）构造好柯西矩阵，在伽罗瓦域中进行高斯消元完成初等变化 ，就可以求解解码矩阵。

（d）测试代码在上述附件"my_ec_test.rar"的matrix_complute.c中

（e）github：https://github.com/xiaoguangwen/bit_matrix_erasure