**摘要**

Erasure Coding（纠删码）被广泛用于存储和网络传输中。每天刷的二维码中就有 Erasure Coding，以至于二维码不全或被部分破坏后还可以扫。在这篇文章中，我希望能把其中的一些数学解释清楚，并且简单而明了地给出一个计算机中的实现。为此，我先会介绍一些线性代数，然后会说到 Galios Field，最后就是 Go 语言中的实现以及 SSE 指令的运用。

# 1 一些线性代数

举个例子，我有两个数 1，2，通过 1+2 = 3，如果 2 丢失，我们还可以通过代数 3-1 重新算出 2，用线性代数表达：

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ \cancel{0} & \cancel{1} \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ \cancel{2} \end{bmatrix} = \begin{bmatrix} 1 \\ \cancel{2} \\ 3 \end{bmatrix} \; row^2 - row^0 \; => \begin{bmatrix} 1 & 0 \\ \cancel{0} & \cancel{1} \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ \cancel{2} \end{bmatrix} = \begin{bmatrix} 1 \\ \cancel{2} \\ 2 \end{bmatrix}$$

这种只有一份冗余方法也叫做 RAID 5，是 Erasure Coding 中的一种特例，计算机中最容易的异或 (xor) 运算就可以做了。如果我们想要更高的冗余度（比如说 2 份）：

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} \cancel{1} & \cancel{0} \\ \cancel{0} & \cancel{1} \\ 1 & 1 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} \cancel{1} \\ \cancel{2} \end{bmatrix} = \begin{bmatrix} \cancel{1} \\ \cancel{2} \\ 3 \\ 5 \end{bmatrix} row^3 - row2 => \begin{bmatrix} \cancel{1} & \cancel{0} \\ \cancel{0} & \cancel{1} \\ 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} \cancel{1} \\ \cancel{2} \end{bmatrix} = \begin{bmatrix} \cancel{1} \\ \cancel{2} \\ 3 \\ 2 \end{bmatrix} row^2 - row^3 => \begin{bmatrix} \cancel{1} & \cancel{0} \\ \cancel{0} & \cancel{1} \\ 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} \cancel{1} \\ \cancel{2} \end{bmatrix} = \begin{bmatrix} \cancel{1} \\ \cancel{2} \\ 1 \\ 2 \end{bmatrix}$$

上边的例子中用的是 Gaussian Decomposition（高斯分解法），先是向前做减法使矩阵下三角变 0，然后向前做减法使上三角变 0，当矩阵变成单位矩阵时，解码也就完成了。相关的论文解释解码的时候通常用反矩阵去做。

# 2 Galios Field

上边的线性代数一点也不复杂，可是到了计算机中，数据是会溢出的（overflow）。于是不得不从实数域转换到 GF 域，在 GF 域中，我们只关心数与数之间的关系，之于数字的实际意义则被忽略，具体可以阅读 https://research.swtch.com/field。整数不是 GF 域，因为不存在整数 x*2==1。整数对质数 p 取模是 GF 域，如 $2*3 == 1(mod5)$，我们可以写成 $2*3 == 1, 1/2 == 3$。Z/2, Z/5, Z/7 都可以被视为 GF 域，其中 Z/2 会很有意思。

计算机中一个字节是 8bit，可以表示 0 ~ 255 共 256 个数，149 可以表示为 10010101，用多项式表示为 $x^7+x^4+x^2+1$。其中每一个 bit 看作是 Z/2 GF 域的话，多项式的加法就变成亦或（xor）运算：$1001,0101+0001,1001 = 0000,1100$，乘法 $(a = 10010101, b = 00011001)$ 变成：

$$
\begin{array}{r}
10010101 \\
\underline{\times\ 00011001} \\
10010101 \\
10010101\phantom{000} \\
\underline{10010101\phantom{0000}} \\
110101101101
\end{array}
\tag{1}
$$

用一个 Go 函数来做这个乘法运算:

```
func gfMult(a, b uint8) uint8 {
        r := uint8(0)
        for b != 0 {
                if b & 0x01 == 0x01 {
                        r ^= a
                }
                a = a << 1
                b >>= 1
        }
        return r
}
```

这个函数稍作修改就可以用作 Erasure coding 的乘法函数了，注意当 a « 1 后可能会 Overflow，这时候我们需要减去（xor）一个质数多项式 $POLY == x^8 + x^4 + x^3 + x^2 + 1$：

```
const POLY uint8 = 0x1d

func gfMult(a, b uint8) uint8 {
        r := uint8(0)

        for b != 0 {
                if b & 0x01 == 0x01 {
                        r ^= a
                }

                if a & 0x80 == 0x80 {
                        a = (a << 1) ^ POLY
                } else {
                        a = a << 1
                }

                b >>= 1
        }
        return r
}
```

# 3 乘法，取反表

在字节之间的乘法，可以用于一个 255 X 255 的来表示 $a*b == mt[a][b]$，除法则转换成乘法 $a/b == a*1/b == mt[a][1/b]$，其中 1/b 可以用另一个 255 的表来表示，这样我们不需要每次都运行上边那个函数。

```
GF_ZERO uint8 = 0x00
GF_ONE  uint8 = 0x01
POLY    uint8 = 0x1d
```

```
type MultTable [256][256]uint8
type InvTable  [256]uint8

func genMultTable() *MultTable {
	mt := &MultTable{}
	for a := 0; a < 256; a += 1 {
		for b := 0; b < 256; b += 1 {
			mt[a][b] = gfMult(uint8(a), uint8(b))
		}
	}
	return mt
}

func genInvTable() *InvTable {
	invt := &InvTable{}
	for a := 1; a < 256; a += 1 {
		for b := 1; b < 256; b += 1 {
			if gfMult(uint8(a), uint8(b)) == GF_ONE {
				invt[a] = uint8(b)
				break
			}
		}
	}
	return invt
}
```

下边的两个函数用来验证这两个表格的正确性

```
func (mt *MultTable) Verify()  {
	ec.Assert(mt[0][0] == GF_ZERO)

	for a := 1; a < 256; a += 1 {
		// a * 0 = 0
		ec.Assert(mt[a][GF_ZERO] == GF_ZERO)
		// 0 * a = 0
```

```
            ec.Assert(mt[GF_ZERO][a] == GF_ZERO)
            // a * 1 = a
            ec.Assert(mt[a][GF_ONE] == uint8(a))
            // 1 * a = a
            ec.Assert(mt[GF_ONE][a] == uint8(a))

            for b := 0; b < 256; b += 1 {
                // a * b = b * a
                ec.Assert(mt[a][b] == mt[b][a])

                for c := 0; c < 256; c += 1 {
                    // (a * b) * c == a * (b * c)
                    ec.Assert(mt[mt[a][b]][c] == mt[a][mt[b][c]]

                    // (a + b) * c == a * c + b * c
                    ec.Assert(mt[a ^ b][c] == mt[a][c] ^ mt[b][c
                }
            }
        }
}

func (invt *InvTable) Verify(mt *MultTable) {
        // 1/0 invalid
        ec.Assert(invt[0] == 0)

        for a := 1; a < 256; a += 1 {
            // a * 1/a == 1
            ec.Assert(mt[a][invt[a]] == GF_ONE)
        }
}
```

# 4 Reed Solomon vs Cauchy

Reed Solomon 矩阵 $\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots \\ 1 & 1 & 1 & \dots & 1 \\ 2^0 & 2^1 & 2^2 & \dots & 2^{k-1} \\ 3^0 & 3^1 & 3^2 & \dots & 3^{k-1} \\ \dots \end{bmatrix}$

Cauchy 矩阵则更方便 $\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots \\ 1 & 1 & 1 & \dots & 1 \\ 1/(i+j) & 1/(i+j) & 1/(i+j) & \dots & 1/(i+j) \\ 1/(i+j) & 1/(i+j) & 1/(i+j) & \dots & 1/(i+j) \\ \dots \end{bmatrix}$

Cauchy 矩阵用 Go 函数生成

```
func cauchy(i, j, k int) uint8 {
        if i < k {
                if i == j {
                        return GF_ONE
                }
                return GF_ZERO
        } else if i == k {
                return GF_ONE
        }
        // 1 / (i + j)
        return INVT[i ^ j]
}
```

# 5 编码解码函数

具体的编码和解码函数代码我会稍后放到 GitHub，目前 Go 实现还没有用到 SSE 指令,所以比较慢。在 k=5, m=3 的单线程运行大概是 180MB/s,用到 SSE 指令后可以跑到 2 ~ 3GB/s，可以说在 X86 的年代，这也是 Erasure coding 的极限了。

在存储中，Erasure Coding 的运行速度已经不是瓶颈，瓶颈在网络，即便这样，我还是看到很多系统（如 HDFS，还有某某 Block Storage）只是用了简单的复制。当我们透过数学去理解它，一切变得一目了然。

仅以这篇文章与程序员群体共勉！徐华良 (hualiang.xu@gmail.com)