[!NOTE] 为什么要写这篇文章？ 已经有很多人写了FEC相关文章，但是总感觉还是留有遗憾，看完还不知道到底该怎么去写代码实现。因此我希望图文并茂地、深入浅出地介绍原理。为了控制每篇文章长度，我将分成多个章节去介绍，目前打算分为基础原理、RTP封装、代码实现这三个章节。如果觉得里面[数学原理](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%95%B0%E5%AD%A6%E5%8E%9F%E7%90%86&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)不清楚，可能还会再开篇章介绍。

## **1. 背景介绍**

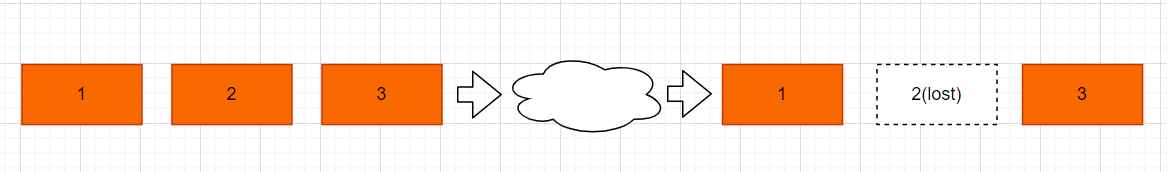
在一个不可靠信道中传输数据，发生丢帧和误码是比较常见的，比如[蜂窝网](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E8%9C%82%E7%AA%9D%E7%BD%91&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)、WIFI、卫星通信等[无线信道](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%97%A0%E7%BA%BF%E4%BF%A1%E9%81%93&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)，为了对抗这种信道的不可靠我们会使用[信道编码](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%BF%A1%E9%81%93%E7%BC%96%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)（Channel Coding，或者叫差错控制编码），它们在[数据链路层](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%95%B0%E6%8D%AE%E9%93%BE%E8%B7%AF%E5%B1%82&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)就会实现[纠错编码](https://zhuanlan.zhihu.com/p/en.wikipedia.org/wiki/Error_correction_code)（Error Correction Code，ECC），可以在数据传输出现错误的时候发现错误并纠正错误。当然，这个纠错技术不仅仅可以用于数据链路层，还可以用于物理层以及更高的层次，如传输层、应用层。

[纠错码](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%BA%A0%E9%94%99%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)（Error Correction Code，ECC）通常也被称作前向纠错（Forward Error Correction, FEC）。常见的纠错编码有LDPC、Turbo、Polar、RS Code、[卷积码](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8D%B7%E7%A7%AF%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)等，这些编码各有自己的特点和适用场景。

本文将聚焦于音视频通信中所使用的FEC技术，由浅入深地介绍FEC原理以及使用。

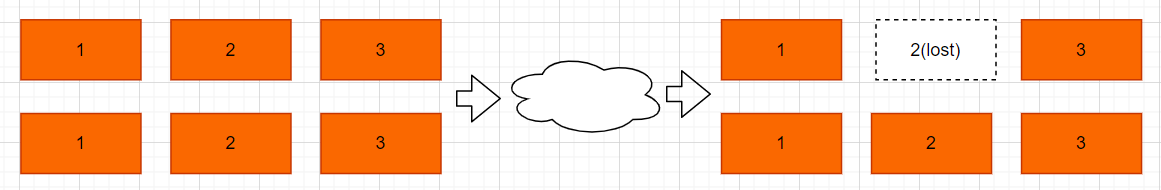
## **2. 问题建模**

RTC通信中，音视频的数据一般都是通过IP网络传输，音视频数据通常是以一个个包（packet）的形式传输，因此在网络层以上只会出现丢包，如下图所示：



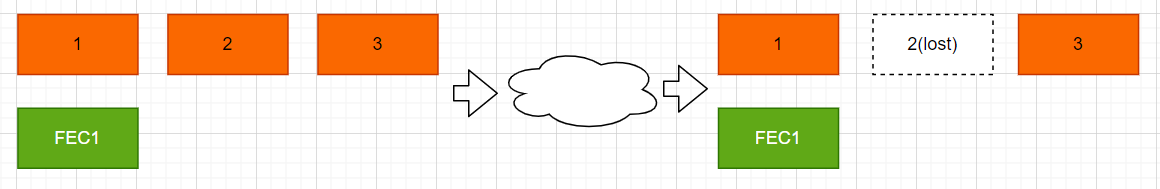
丢包模型

从[信息论](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%BF%A1%E6%81%AF%E8%AE%BA&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)角度来看，如果需要对抗这样的丢包，就必须要增加[冗余](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%86%97%E4%BD%99&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)。这里我们想到一个最简单的方法：重复码，也就是对所有的包都冗余N份，这样丢了一个也不用担心，只要收到其中一个即可。如下图所示我们采用了1倍冗余方式，包2丢失也了没有关系，我们还能收到另外一份：

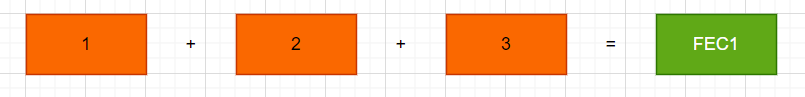


重复码

以上的冗余方式是能达到我们的目的，但是**代价也是比较大**，为了对抗丢包我们引入了1倍冗余。因此，我们就得探索更有效的信道编码方式。如果你了解信息论，那你肯定知道，我要讲的就是[线性分组码](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%BA%BF%E6%80%A7%E5%88%86%E7%BB%84%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)了。我们还是先从简单的方法介绍起，每3个包生成一个FEC：



是如何做到的呢？我们先假设，以上每个包只包含一个字节（分别为B1、B2、B3、FEC1），我们定义我们的FEC生成方式为B1+B2+B3=FEC1:



如果包2丢失，那么可以通过以下方(B2 = FEC1 - B1 - B3)式恢复：

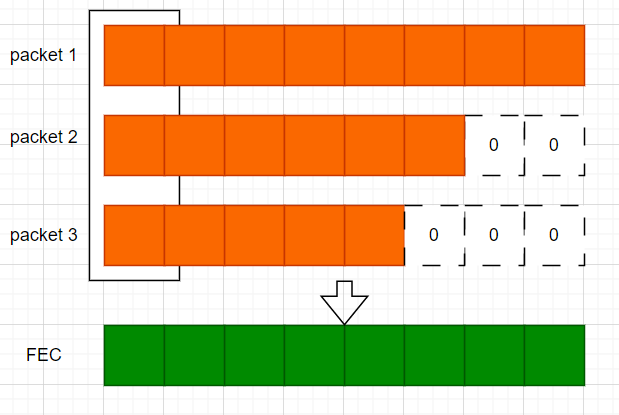


如果仔细思考就会发现，以上的方法会出现很多的问题：

* 以上的模型是1byte的，实际上每个包是多个字节的，而且长度不相同
* 只能3[原始包](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8E%9F%E5%A7%8B%E5%8C%85&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)+1个FEC包中，最多只能丢失一个，否则将无法恢复
* 生成FEC包的方式会存在溢出，如果我们的生成FEC方式复杂点，如k1xB1+k2xB2+k3xB3=FEC1，可能会出现丢包无法恢复

先解决第一个问题，上面的例子是假设一个报文只有一个字节，实际上IP报文可能有1000字节左右，而且不同报文见长度不同，那么怎么处理的？

* 对所有报文做补0，对齐到最大的报文长度（一般来说都会有信息记录当前数据的实际长度，所以不用担心长度丢失）
* 取所有报文相同偏移的码字，逐个码字做FEC，码字可以是上面距离的1byte，也可以是1bit，16bits unit、32bits unit等等，这取决于设计。
* 将生成的FEC数据逐个拼接起来便得到一个FEC报文

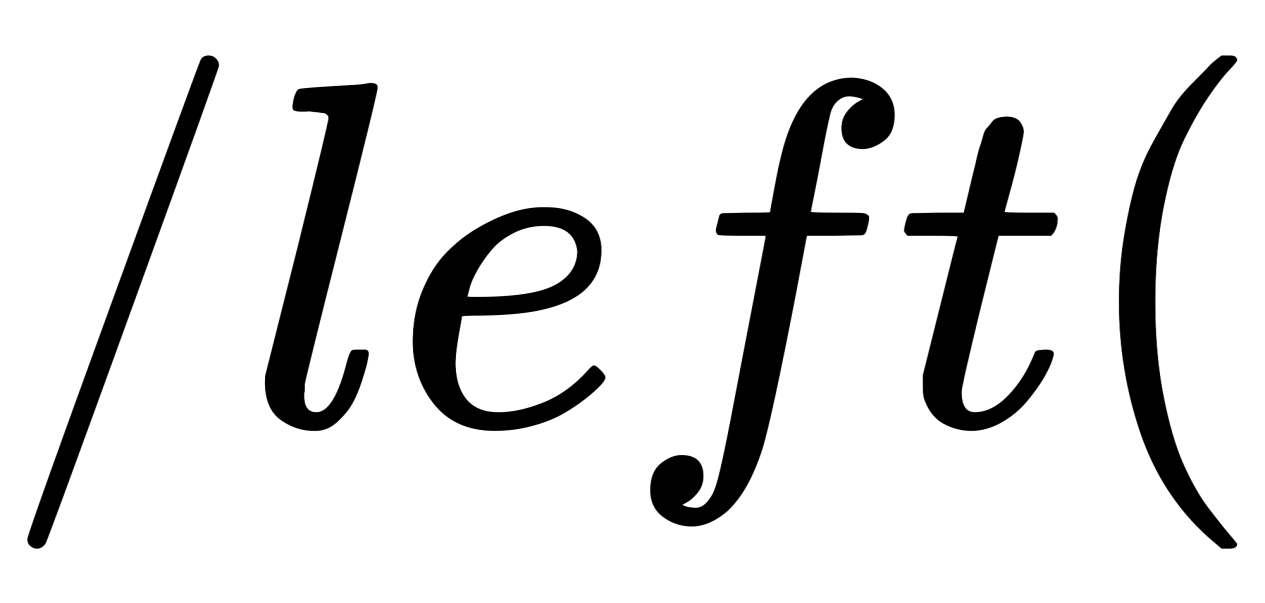


## **3. 线性分组码**

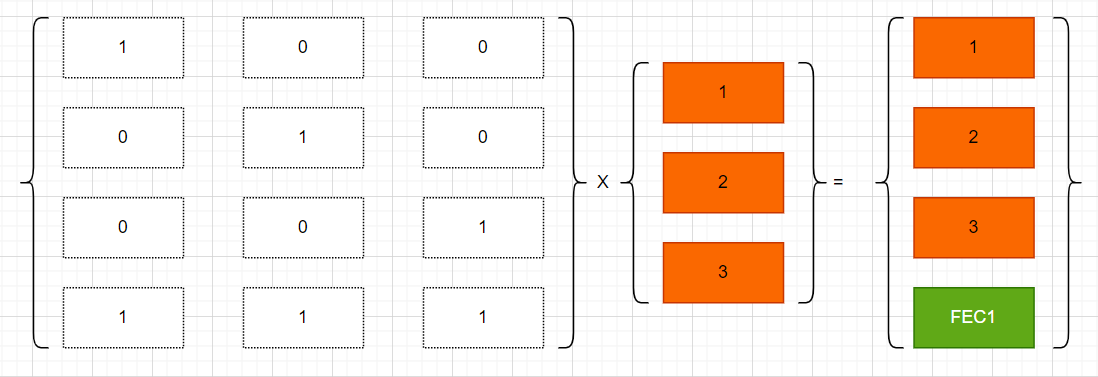
通过上面的介绍，我们实际上已经再使用线性分组码了，我们先要了解下概念，线性分组码可以拆成线性 + 分组码。

分组码，顾名思义，就是把输入的数据分成 k个一组， 把这k个数据映射为n个数据，一般使用 (n, k) 这个记号来描述。所谓的线性是指k个数据到n个数据的映射是一个[线性映射](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%BA%BF%E6%80%A7%E6%98%A0%E5%B0%84&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)。

我们重新表示下上面的例子，用公式表示（我们假设你是了解代数相关知识的）：

lcr%7D%0AB1%5C%5C%0AB2%5C%5C%0AB3%5C%5C%0AFEC1%0A%5Cend%7Barray%7D%0A%5Cright%29" alt="\left( \begin{array}{lcr} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{array} \right) \* \left( \begin{array}{lcr} B1\\ B2\\ B3 \end{array} \right) = \left( \begin{array}{lcr} B1\\ B2\\ B3\\ FEC1 \end{array} \right)" eeimg="1"/>

下图看起来可能会更直观：



以上的生成FEC方式较为简单，只能生成一个FEC，我们可以更复杂点：

(100010001g1,1g1,2g1,3g2,1g2,2g2,3)∗(B1B2B3)=(B1B2B3FEC1FEC2)

通过以上方法，我们不就生成了两个FEC了吗，你想要更多的都可以，只要合理设计[多项式系数](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%A4%9A%E9%A1%B9%E5%BC%8F%E7%B3%BB%E6%95%B0&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)即可。通过以上的方式，我们可以生成两个FEC，因此可以解决丢两个包的问题了。

不是一般性，我们可以想到，我们可以想到k个数据生成n个数据表达式（n-k个[冗余数据](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%86%97%E4%BD%99%E6%95%B0%E6%8D%AE&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)）：

(10...0001...00...............00...1000......1g1,1g1,2...g1,k−1g1,kg2,1g2,2...g2,k−1g2,k...............gn−k−1,1gn−k−1,2...gn−k−1,k−1gn−k−1gn−k,1gn−k2,2...gn−k,k−1gn−k)∗(B1B2...Bk)=(B1B2...BkFEC1FEC2...FECn−k)

简写如下，其中**I**表示[单位矩阵](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8D%95%E4%BD%8D%E7%9F%A9%E9%98%B5&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)，G是FEC的[生成矩阵](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%94%9F%E6%88%90%E7%9F%A9%E9%98%B5&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)（通常将I+G整体成为线性分组码的生成矩阵），D是原始数据，F是生成的FEC：

(IG)∗D=(DF)

以上便是FEC最基本的思想了。当然，溢出的问题还没有解决，其次又有一个新的问题，我们的FEC生成表达式如何设计。

## **4. FEC运算的域**

这里介绍两种方法，一个是XOR（[二进制域](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%BA%8C%E8%BF%9B%E5%88%B6%E5%9F%9F&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)上的XOR运算），一个是RS Code（[伽罗华域](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%BC%BD%E7%BD%97%E5%8D%8E%E5%9F%9F&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)上的运算），这两个方法都能解决运算溢出的问题。

### **4.1 XOR**

WebRTC里面默认的FEC方式便是XOR，其优势是运算速度非常快，缺点在于抗丢包的能力不是特别好。

#### **异或运算**

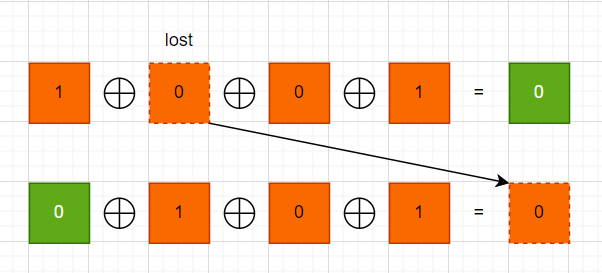
[!NOTE] 在XOR FEC里，乘法保持不变，加法、减法使用异或代替。乘法的系数，这里也叫做[掩码](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%8E%A9%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)（mask）。

XOR（exclusive or，异或）是一个按位运算方法，计算机中通常用符号^表示，数学符号为⊕。两个bit不相同则异或结果为1，相同则异或结果为0，即0⊕0=0，1⊕0=1，0⊕1=1，1⊕1=0。

对于任意K个8bits的数据，异或结果仍然是8bits，运算不存在溢出问题，而且运算速度还非常快，丢失任何一个数据都可以恢复：

B1⊕B2⊕...⊕Bk=FBi=F⊕B1⊕B2⊕...⊕Bk

对于4个bit做一组FEC，丢失任何一个bit都可以通过异或方式恢复：



XOR运算

一般来说，对于XOR方式[增广矩阵](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%A2%9E%E5%B9%BF%E7%9F%A9%E9%98%B5&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)下面部分的系数只能是1（选择与不选择），如果只生成一个FEC那么不能抗连续丢包，因此一般都会生成多个FEC。下面这个示例会生成两个FEC。生成矩阵中的1表示选择该元素，0表示没有选择该元素，元素之间[加法运算](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8A%A0%E6%B3%95%E8%BF%90%E7%AE%97&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)是XOR运算。

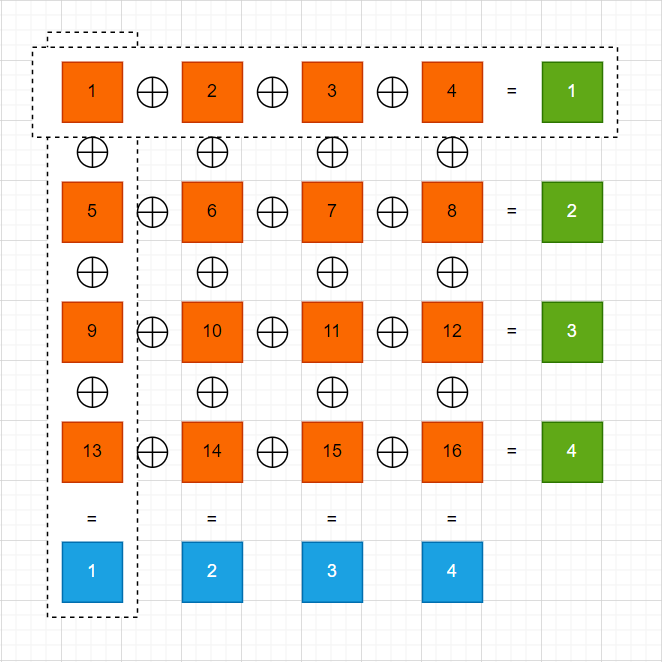
(100010001101111)∗(B1B2B3)=(B1B2B3FEC1FEC2)

#### **FlexFEC**

XOR方式的增广矩阵的所有系数只能是1，生成FEC的形式有限，其抗连续丢包能力比较差。不像RS code接收到任意k个数据都能恢复，XOR的分组内只要丢两个包就无法恢复。因此，WebRTC里面采用交织（interleave）方式生成FEC。

所谓的交织如下图所示：

* 绿色是非交织生成的，连续丢包无法抗，如丢了包1、2，仅收到3、4和绿色FEC1是无法恢复。
* 蓝色是交织生成的，如果连续丢了1、2，可以通过蓝色FEC恢复



交织的好处在于可以抗连续丢包，缺点也很明显，**会引入较大的恢复延迟**。比如需要通过交织的FEC恢复，丢了包1，需要等到收到包13才能恢复，需要等待的时间过长。

WebRTC支持：

* 非交织方式（绿色，1D非交织）
* 交织方式（蓝色，1D交织）
* 非交织+交织组合（绿色+蓝色，2D）
* 以及更灵活的flex方式

上面的交织+非交织的2D方式，即16个原始数据生成8个FEC，同样也可以表示成矩阵方式（此处省略了空格）：

(10000000000000000100000000000000...0000000000000010000000000000000111110000000000000000111100000000000000001111000000000000000011111000100010001000010001000100010000100010001000100001000100010001)

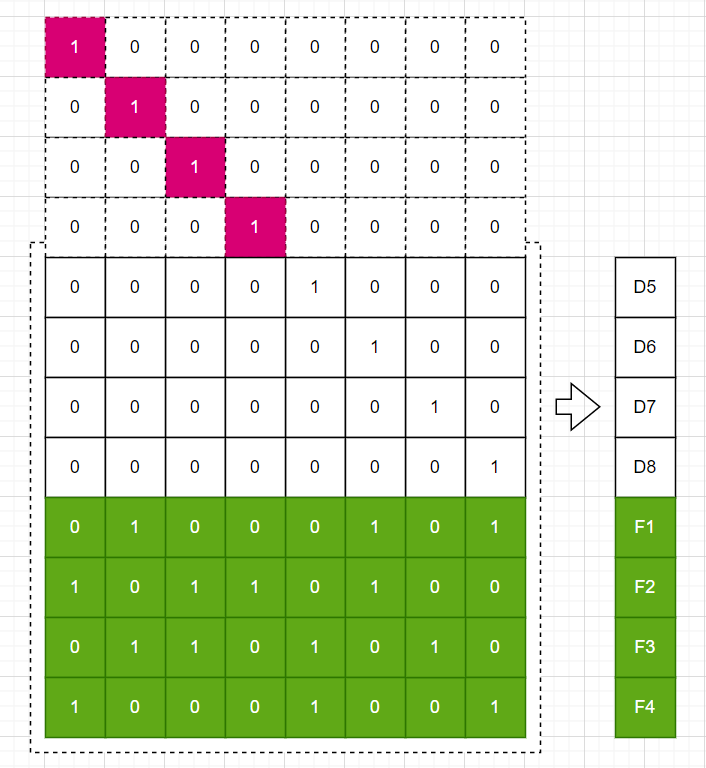
#### **随机丢包和突发（连续）丢包**

在WebRTC源码中，通常会将增广矩阵的最下面8行表示成[十六进制](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8D%81%E5%85%AD%E8%BF%9B%E5%88%B6&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)方式，大家阅读代码的时候需要注意。对于flex FEC，我们可以看下WebRTC的随机丢包和突发丢包的生成矩阵分别是什么。

随机丢包的生成矩阵：

*// 8个[媒体包](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%AA%92%E4%BD%93%E5%8C%85&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank) 生成8个媒体包+4个FEC包***#define kMaskRandom8\_4 \ 0x45, 0x00, \ 0xb4, 0x00, \ 0x6a, 0x00, \ 0x89, 0x00**

以上的mask改写成二进制的矩阵形式如下(矩阵绿色部分)：



从数学角度来看，丢包就相当于从单位矩阵中删除几行。如上图连续删除了4行，还剩8行，是一个8x8的矩阵。

M∗D=[D5 D6 D7 D8 F1 F2 F3 F4]T

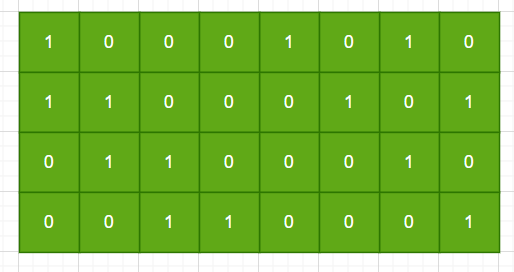
是否能够从结果恢复出原来的数据D，需要M可逆，则得到：

D=M−1∗[D5 D6 D7 D8 F1 F2 F3 F4]T

按上面方式删除前4行刚好可逆（只需要关注左下角4x4即可），通过多次迭代总是能够恢复所有数据。大家可以尝试下删除数据D2~D5，会发现矩阵不可逆，无论如何无法完全恢复所有数据。

突发（连续）丢包的生成矩阵（省略了上面的单位矩阵）：

*// 8个媒体包 生成8个媒体包+4个FEC包***#define kMaskBursty8\_4 \ 0x8a, 0x00, \ 0xc5, 0x00, \ 0x62, 0x00, \ 0x31, 0x00**



可以看到，这个bursty mask，每一个包至少用到两次。任意选择连续丢4个包都可以恢复，大家可以自己试下。按照以上方式组织，会发现删除任意行，矩阵均可逆。

#### **解码**

因为XOR方式，每次只能恢复一个数据，因此只需要逐个公式迭代，直到所有的数据都恢复。

### **4.2 RS Code**

RS Code，即Reed-solomon Code，理得-[所罗门](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%89%80%E7%BD%97%E9%97%A8&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)编码，简称理所码。它是一个定义在[有限域](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%9C%89%E9%99%90%E5%9F%9F&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)（伽罗华域，Galois Field）上的线性分组码，属于BCH码变种，它的纠错能力非常强，不仅能抗突发丢包，也能抗随机丢包。

k个原始数据生成n个最终码字的RS code表示为 RS(n,k)。为了能达到任意去掉n-k行，矩阵都可逆，我们一般选择范德蒙（Vandermonde）矩阵或柯西（Cauchy）矩阵，这两个矩阵恰好能满足要求。

[范德蒙矩阵](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E8%8C%83%E5%BE%B7%E8%92%99%E7%9F%A9%E9%98%B5&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)形式生成矩阵，（这里第一行指数的底我从0开始的）：

(10...0011...111a11...a1k−2a1k−11a21...a2k−2a2k−1...1an−11...an−1k−2an−1k−1)

柯西矩阵形式生成矩阵：

(1x1+y11x1+y2...1x1+yk−11x1+yk1x2+y11x2+y2...1x2+yk−11x2+yk...1xn+y11xn+y2...1xn+yk−11xn+yk)

任意删除n-k行，得到kxk的矩阵可逆，这个高等代数知识，此处不做证明，后续文章会详细介绍下RS code的原理。

RS Code的加密和上面的XOR基本上一样，也就只是[矩阵乘法](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%9F%A9%E9%98%B5%E4%B9%98%E6%B3%95&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)运算，主要三层循环：行x列x数据长度。但是其原理理解的难点主要有两个，一个是[可逆矩阵](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%8F%AF%E9%80%86%E7%9F%A9%E9%98%B5&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)的设计与求解，还有就是在有限域上的运算。这里将以范德蒙矩阵为例，教你理解原理和实现。

#### **基于范德蒙矩阵**

我们先看远古版本的范德蒙矩阵的设计，代码可以参考这个开源库： [WojciechMigda/fec: forward error correction based on Vandermonde matrices (github.com)](https://link.zhihu.com/?target=https://github.com/WojciechMigda/fec" \t "https://zhuanlan.zhihu.com/p/_blank)

为了简化，我们这里还是在整数域上介绍生成冗余的原理。我们重写下nxk的范德蒙矩阵：

G=(10...0011...111a11...a1k−2a1k−11a21...a2k−2a2k−1...1an−11...an−1k−2an−1k−1)

拿这个生成矩阵去生成数据，我们可以发现，生成的矩阵并不包含原始的数据。学过[编码理论](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%BC%96%E7%A0%81%E7%90%86%E8%AE%BA&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)的 可能知道，这个不是[系统码](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Systematic_code" \t "https://zhuanlan.zhihu.com/p/_blank)，所谓的系统码是指生成的冗余中包含了原始数据。所有的非系统码都可以转化为系统码。所有的系统码的形式一定是下面这种，即上面的kxk方阵一定是单位矩阵。因此，第一步先需要将生成矩阵的上半部分转为单位矩阵。

步骤如下：

G=(G1G2)

其中G1是kxk的方阵，我们可以对G1求逆（kxk范德蒙矩阵一定存在逆矩阵，求逆方法后续介绍），整个生成矩阵G乘上G1的逆可以保证上半部分是单位矩阵I：

G′=G∗G1−1=(G1∗G1−1G2∗G1−1)=(IkG2∗G1−1)

用这个系统矩阵生成的数据，一定会包含原始数据。WebRTC正需要系统码的这种好处，以兼容不支持FEC的客户端，这写客户端只接收原始数据也可以解码。

#### **基于柯西矩阵**

柯西矩阵比范德蒙矩阵要简单点，编码时不需要求逆了，其生成矩阵如下：

(10...0001...00...............00...1000...011x1+y11x1+y2...1x1+yk−11x1+yk1x2+y11x2+y2...1x2+yk−11x2+yk...1xn+y11xn+y2...1xn+yk−11xn+yk)

#### **有限域运算**

如果在整数域上计算求逆等运算，必然会引入很多问题，如引入小数、导致溢出等等。因此，我们需要一个新的“有限域”，无论如何运算，结果仍然在这个域的集合中。

[!NOTE] 域 简单地理解，域就是集合以及定义在这个集合上的运算，这里包含加法和乘法运算，它存在以下性质：

1. 加法和乘法满足[交换律](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%BA%A4%E6%8D%A2%E5%BE%8B&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)，即a+b=b+a，a\*b=b\*a。
2. 加法和乘法满足[结合律](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E7%BB%93%E5%90%88%E5%BE%8B&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)，即a+(b+c) = (a+b)+c, a\*(b\*c)=(a\*b)\*c。
3. 存在单位元和零元，类似于整数域的0和1，这里我们分别记作0，i，有a+0=a，a\*1=a。
4. 加法和乘法存在逆元（除了0以外），a的加法逆元记作-a，a的[乘法逆元](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%B9%98%E6%B3%95%E9%80%86%E5%85%83&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)记作a−1，有a+(−a)=0,a∗a−1=1。
5. 满足结合律：a\*(b+c) = a\*b + a\*c。

因为加法和乘法存在逆元，因此减法和除法可以转换为加法和乘法。另外，域里面的元素不仅仅可以是数，还可以是任何东西，只要能满足以上的运算。因此，元素也可以是多项式！

有限域（Finite Fields），顾名思义就是集合元素有限的域，也被称作伽罗华域（Galois Fields），有限域里面元素经过运算后结果仍在这个有限域中。其中有限域中元素的个数q被称为阶（order），有限域作GF(q)，有限域的阶一定是素数的幂，即GF(pn)，其中p是素数，即有：

其中GF(pn)=a0+a1∗x+a2∗x2+...+an−1∗xn−1,其中ai∈Zp,Zp={0,1,...,p−1}

在计算机里面我们一般用GF(2n)，二进制更适合计算机，如GF(28)：

其中GF(28)=a0+a1∗x+a2∗x2+...+a7∗x7,其中ai∈Z2,Z2={0,1}

表示成二进制就是0ba7a6a5a4a3a2a1a0，GF(28)的阶为256，即包含256个多项式，这样多项式就可以以二进制方式来表示了，更适合计算机存储。

多项式的加法和乘法和普通的多项式运算类似，但是其系数最终结果总是要模p的，在GF(28)里面就是要模2。比如：

(x+1)∗(x+1)=x2+2∗x+1=x2+1

多项式的除法也是类似。

[不可约多项式](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E4%B8%8D%E5%8F%AF%E7%BA%A6%E5%A4%9A%E9%A1%B9%E5%BC%8F&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)：如果多项式p(x)=a0+a1∗x+a2∗x2+...+an−1∗xn不能被任何一个多项式整除，那么称这个多项式不可约。 [本原多项式](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%9C%AC%E5%8E%9F%E5%A4%9A%E9%A1%B9%E5%BC%8F&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)：如果不可约多项式能够整除xn的n最小取值为n=qm−1，那么称p(x)为本原多项式。

为了能生成有限域里面的元素（多项式），我们引入本原多项式。本原多项式一定是一个不可约多项式，可以类比整数域里面的素数。我们需要使用本原多项式来构造伽罗华域。本原多项式不一定是唯一的。这里以GF(24)为例，其本原多项式可以取x4+x3+1和x4+x+1。大家可以根据本原多项式的性质证明。

如下是一个本原多项式列表：

*/\* \* Primitive polynomials - see Lin & Costello, Appendix A, \* and Lee & Messerschmitt, p. 453. \*/***static** **char** **\***allPp[] **=** {

*/\* GF\_BITS polynomial \*/*

NULL, */\* 0 no code \*/*

NULL, */\* 1 no code \*/*

"111", */\* 2 1+x+x^2 \*/*

"1101", */\* 3 1+x+x^3 \*/*

"11001", */\* 4 1+x+x^4 \*/*

"101001", */\* 5 1+x^2+x^5 \*/*

"1100001", */\* 6 1+x+x^6 \*/*

"10010001", */\* 7 1 + x^3 + x^7 \*/*

"101110001", */\* 8 1+x^2+x^3+x^4+x^8 \*/*

"1000100001", */\* 9 1+x^4+x^9 \*/*

"10010000001", */\* 10 1+x^3+x^10 \*/*

"101000000001", */\* 11 1+x^2+x^11 \*/*

"1100101000001", */\* 12 1+x+x^4+x^6+x^12 \*/*

"11011000000001", */\* 13 1+x+x^3+x^4+x^13 \*/*

"110000100010001", */\* 14 1+x+x^6+x^10+x^14 \*/*

"1100000000000001", */\* 15 1+x+x^15 \*/*

"11010000000010001" */\* 16 1+x+x^3+x^12+x^16 \*/*};

如何生成本原多项式呢？这里不给出证明，直接给出方法：

对于GF(2n)，使用x0,X1,...,xn−1分别模其本原多项式p(x)，就能按顺序得到除了0之外的所有元素。

以下以GF(24)为例，其中p(x)=x4+x+1：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | x^i | x^i mod p(x) | 二进制 | 十进制 |
| - | 0 | 0 | 0000 | 0 |
| 0 | x^0 | 1 | 0001 | 1 |
| 1 | x^1 | x | 0010 | 2 |
| 2 | x^2 | x^2 | 0100 | 4 |
| 3 | x^3 | x^3 | 1000 | 8 |
| 4 | x^4 | x+1 | 0011 | 3 |
| 5 | x^5 | x^2+x | 0110 | 6 |
| 6 | x^6 | x^3+x^2 | 1100 | 12 |
| 7 | x^7 | x^3+x+1 | 1011 | 11 |
| 8 | x^8 | x^2+1 | 0101 | 5 |
| 9 | x^9 | x^3+x | 1010 | 10 |
| 10 | x^10 | x^2+x+1 | 0111 | 7 |
| 11 | x^11 | x^3+x^2+x | 1110 | 14 |
| 12 | x^12 | x^3+x^2+x+1 | 1111 | 15 |
| 13 | x^13 | x^3+x^2+1 | 1101 | 13 |
| 14 | x^14 | x^3+1 | 1001 | 9 |

可以证明，x15 mod p(x)=1，且后续再计算结果都会在集合中。通过以上的计算，我们得到一个序号和多项式的对应关系：

* gfexp(i)表示index到多项式的转换，如gfexp(9) = 10，注意gfexp的index取值范围为2n−1
* gflog(i)表示多项式到index的转换，如gflog(12)=6

这里的运算和指数、对数比较类似，因此用exp和log来表示。因为0是一个比较特殊的，需要注意。这里的指数运算和[对数运算](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E5%AF%B9%E6%95%B0%E8%BF%90%E7%AE%97&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)基本可以直接套用普通的数学运算，除了要取模，如GF(2^4)里要模15（注意不是16）。

i∗j=>xi∗xj=x(i+j) mod (2n−1−1)

伽罗华域axb：

a∗b=gfexp[gflog(a∗b)]=gfexp[gflog(a)+gflog(b)]=gfexp[ia+ib]

如3∗4=gfexp[i3+i4]=gfexp[4+2]=gfexp[6]=12

以上的运算方法注意需要牢记，因为在代码实现上会有很多这样的转换。

#### **开源实现推荐**

下面推荐几个RS码的开源实现和优化，包含go、python、c的实现，大家可以参考。

[klauspost/reedsolomon: Reed-Solomon Erasure Coding in Go (github.com)](https://link.zhihu.com/?target=https://github.com/klauspost/reedsolomon" \t "https://zhuanlan.zhihu.com/p/_blank)

[templexxx/reedsolomon: Reed-Solomon Erasure Code engine in Go, could more than 15GB/s per core (github.com)](https://link.zhihu.com/?target=https://github.com/templexxx/reedsolomon" \t "https://zhuanlan.zhihu.com/p/_blank)

[tomerfiliba/reedsolomon: A pure-python Reed Solomon encoder/decoder (github.com)](https://link.zhihu.com/?target=https://github.com/tomerfiliba/reedsolomon" \t "https://zhuanlan.zhihu.com/p/_blank)

[jannson/reedsolomon-c: C compatible version for https://github.com/klauspost/reedsolomon.](https://link.zhihu.com/?target=https://github.com/jannson/reedsolomon-c" \t "https://zhuanlan.zhihu.com/p/_blank)

## **5. 总结**

花了很久，终于写完了FEC的基础原理，但是感觉并没有完全写完，因为RS Code部分还有很多细节没有说清楚。本来想在RS code部分可以画更多图便于大家理解，但是最终还是没有画，这部分实在是需要一些[数学基础](https://zhida.zhihu.com/search?content_id=222225785&content_type=Article&match_order=1&q=%E6%95%B0%E5%AD%A6%E5%9F%BA%E7%A1%80&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)才能够理解，所以后续会继续完善下后面部分，希望能够以最直白的方式让大家理解。