## **1. 背景**

之前我们有介绍FEC的原理，以及如何通过RTP来发送，见下面下面两篇文章。

[言剑：FEC深入浅出1 - 基础原理62 赞同 · 15 评论文章](https://zhuanlan.zhihu.com/p/603421239" \t "https://zhuanlan.zhihu.com/p/_blank)

[言剑：FEC深入浅出2 - RTP封装12 赞同 · 4 评论文章](https://zhuanlan.zhihu.com/p/603677215" \t "https://zhuanlan.zhihu.com/p/_blank)

今天我们要结合代码来讲解下WebRTC的FEC（FlexFEC和ULPFEC）实现，主要介绍下基本流程，细节的我希望放到后续章节慢慢介绍。希望这篇文章能够对大家有所启发。

## **2. WebRTC中的FEC**

在WebRTC中，FEC的实现见modules/rtp\_rtcp/source/forward\_error\_correction.cc，下面我们就从ForwardErrorCorrection这个入口介绍下FEC的编码和解码。这里的重点在于，FEC编解码实现以及RTP头部处理。

### **2.1 FEC编码流程**

编码处理简化后如下，简单点说：

* 根据protection\_factor和输入的包，确定FEC包个数
* 根据<[媒体包](https://zhida.zhihu.com/search?content_id=222740587&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)个数，FEC包个数>确定FEC保护mask
* 对于丢包情况调整FEC mask
* 根据mask，以xor方式生成FEC
* 处理FEC头部，填充seq base、ssrc、mask等

*// FEC编码***int** ForwardErrorCorrection**::**EncodeFec(**const** PacketList**&** media\_packets,

**uint8\_t** protection\_factor,

**int** num\_important\_packets,

**bool** use\_unequal\_protection,

FecMaskType fec\_mask\_type,

std**::**list**<**Packet**\*>\*** fec\_packets) {

**const** size\_t num\_media\_packets **=** media\_packets.size();

*// 最多支持多少个媒体包？* *// FEC算法本身没有限制媒体包个数上限，这个最大包个数限制是FlexFEC的限制* *// 因为RTP的长度有限、延迟有限制等，因此支持的最大FEC长度也有限制。* **const** size\_t max\_media\_packets **=** fec\_header\_writer\_**->**MaxMediaPackets();

**if** (num\_media\_packets **>** max\_media\_packets) {

**return** **-**1;

}

*// 包长度校验，代码略* ...

*// FEC包个数 = 媒体包个数 \* protection\_factor* *// protection\_factor这里是从[0, 1]映射到[0, 255]* **int** num\_fec\_packets **=** NumFecPackets(num\_media\_packets, protection\_factor);

**if** (num\_fec\_packets **==** 0) {

**return** 0;

}

*// FEC包内存申请，清空* **for** (**int** i **=** 0; i **<** num\_fec\_packets; **++**i) {

generated\_fec\_packets\_[i].data.EnsureCapacity(IP\_PACKET\_SIZE);

memset(generated\_fec\_packets\_[i].data.MutableData(), 0, IP\_PACKET\_SIZE);

*// Use this as a marker for untouched packets.* generated\_fec\_packets\_[i].data.SetSize(0);

fec\_packets**->**push\_back(**&**generated\_fec\_packets\_[i]);

}

*// 根据 <媒体包个数,FEC包个数> 确定编码的mask（FEC生成矩阵）* *// 关于mask，仔细阅读下前两篇文章，如何生成mask的代码，我们在后面介绍* internal**::**PacketMaskTable mask\_table(fec\_mask\_type, num\_media\_packets);

packet\_mask\_size\_ **=** internal**::**PacketMaskSize(num\_media\_packets);

memset(packet\_masks\_, 0, num\_fec\_packets **\*** packet\_mask\_size\_);

internal**::**GeneratePacketMasks(num\_media\_packets, num\_fec\_packets,

num\_important\_packets, use\_unequal\_protection,

**&**mask\_table, packet\_masks\_);

*// 一般我们认为输入的包序号是连续的，万一不连续的？下面逻辑是对这种情况做调整。* *// 那我们的mask也需要调整，对于不存在的sequence number，mask中直接插入0即可* **int** num\_mask\_bits **=** InsertZerosInPacketMasks(media\_packets, num\_fec\_packets);

**if** (num\_mask\_bits **<** 0) {

fec\_packets**->**clear();

**return** **-**1;

}

packet\_mask\_size\_ **=** internal**::**PacketMaskSize(num\_mask\_bits);

*// 根据上面的mask生成FEC，并存到`generated\_fec\_packets\_`.* GenerateFecPayloads(media\_packets, num\_fec\_packets);

*// 获取媒体包的SSRC，保护多条不同的流逝需要* *// FEC group的起始sequence base，seq base在RTP打包时需要* **const** **uint32\_t** media\_ssrc **=** ParseSsrc(media\_packets.front()**->**data.data());

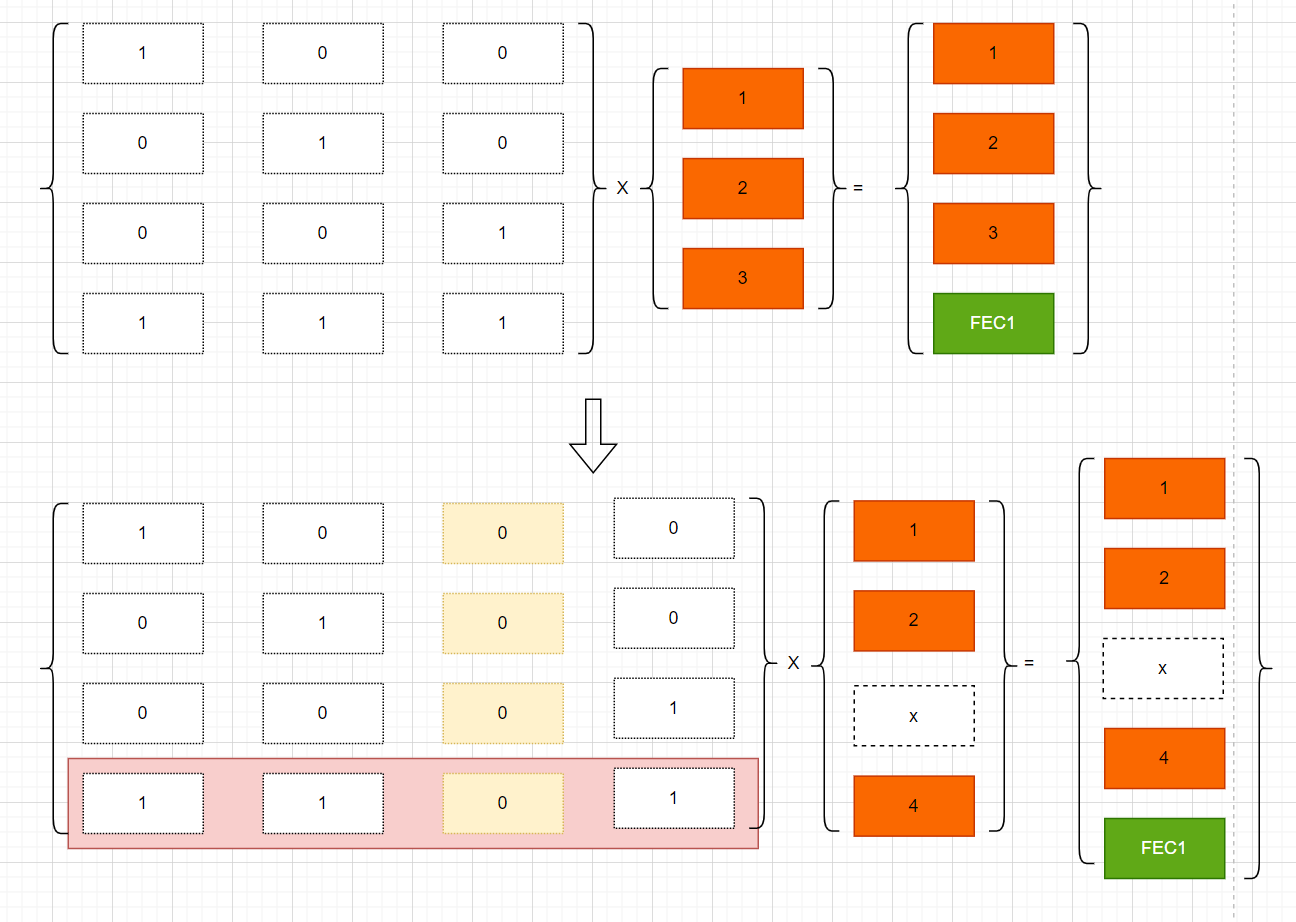
**const** **uint16\_t** seq\_num\_base **=**

ParseSequenceNumber(media\_packets.front()**->**data.data());

*// 对生成的所有FEC，处理FEC头部，比如填充seqbase，ssrc，mask等等* FinalizeFecHeaders(num\_fec\_packets, media\_ssrc, seq\_num\_base);

**return** 0;}

关于sequence不连续，在中间插入0的解释，可以用下面的图简单解释。sequence number 3没有参与FEC（3是因为丢失或者是其他原因），我们是通过sequence base和mask来通知接收端哪些包参与了FEC，seq 3没有参与运算要能够通过mask通知到接收端。  
我们可以假设仍然保留它，在矩阵中插入一列0，就相当于sequence number 3的包没有参与运算。扩展的代码见InsertZerosInPacketMasks，篇幅限制就不解读这块代码了。



FEC MASK插入0

紧接着上面，我们来介绍下，如何编码payload（GenerateFecPayloads）：

**void** ForwardErrorCorrection**::**GenerateFecPayloads(

**const** PacketList**&** media\_packets,

size\_t num\_fec\_packets) {

*// 回忆下，我们第一篇文章中介绍，对于M个媒体包生成F个FEC包* *// 在生成矩阵里，除了上面的单位矩阵，还有F行的矩阵，即F行mask* *// 这里遍历所有行mask，生成FEC* **for** (size\_t i **=** 0; i **<** num\_fec\_packets; **++**i) {

Packet**\*** **const** fec\_packet **=** **&**generated\_fec\_packets\_[i];

size\_t pkt\_mask\_idx **=** i **\*** packet\_mask\_size\_;

*// 我们知道，FlexFEC的FEC头部是变长的，根据mask长度来的* **const** size\_t min\_packet\_mask\_size **=** fec\_header\_writer\_**->**MinPacketMaskSize(

**&**packet\_masks\_[pkt\_mask\_idx], packet\_mask\_size\_);

**const** size\_t fec\_header\_size **=**

fec\_header\_writer\_**->**FecHeaderSize(min\_packet\_mask\_size);

*// 这个index用于在mask中索引生成[矩阵系数](https://zhida.zhihu.com/search?content_id=222740587&content_type=Article&match_order=1&q=%E7%9F%A9%E9%98%B5%E7%B3%BB%E6%95%B0&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)，因为seq之间可能有gap，因此需要这种方法* size\_t media\_pkt\_idx **=** 0;

**auto** media\_packets\_it **=** media\_packets.cbegin();

**uint16\_t** prev\_seq\_num **=** ParseSequenceNumber((**\***media\_packets\_it)**->**data.data());

**while** (media\_packets\_it **!=** media\_packets.end()) {

Packet**\*** **const** media\_packet **=** media\_packets\_it**->**get();

*// mask中对应的bit位如果为1（生成矩阵系数），则表示参与XOR FEC运算* **if** (packet\_masks\_[pkt\_mask\_idx] **&** (1 **<<** (7 **-** media\_pkt\_idx))) {

size\_t media\_payload\_length **=** media\_packet**->**data.size() **-** kRtpHeaderSize;

*// FEC包长度 这里会自动调整，最终会扩展到最长的媒体包* size\_t fec\_packet\_length **=** fec\_header\_size **+** media\_payload\_length;

**if** (fec\_packet\_length **>** fec\_packet**->**data.size()) {

fec\_packet**->**data.SetSize(fec\_packet\_length);

}

*// 对头部做XOR FEC处理，生成FEC头部* XorHeaders(**\***media\_packet, fec\_packet);

*// 对payload做XOR FEC处理* XorPayloads(**\***media\_packet, media\_payload\_length, fec\_header\_size,

fec\_packet);

}

media\_packets\_it**++**;

**if** (media\_packets\_it **!=** media\_packets.end()) {

**uint16\_t** seq\_num **=** ParseSequenceNumber((**\***media\_packets\_it)**->**data.data());

media\_pkt\_idx **+=** **static\_cast<uint16\_t>**(seq\_num **-** prev\_seq\_num);

prev\_seq\_num **=** seq\_num;

}

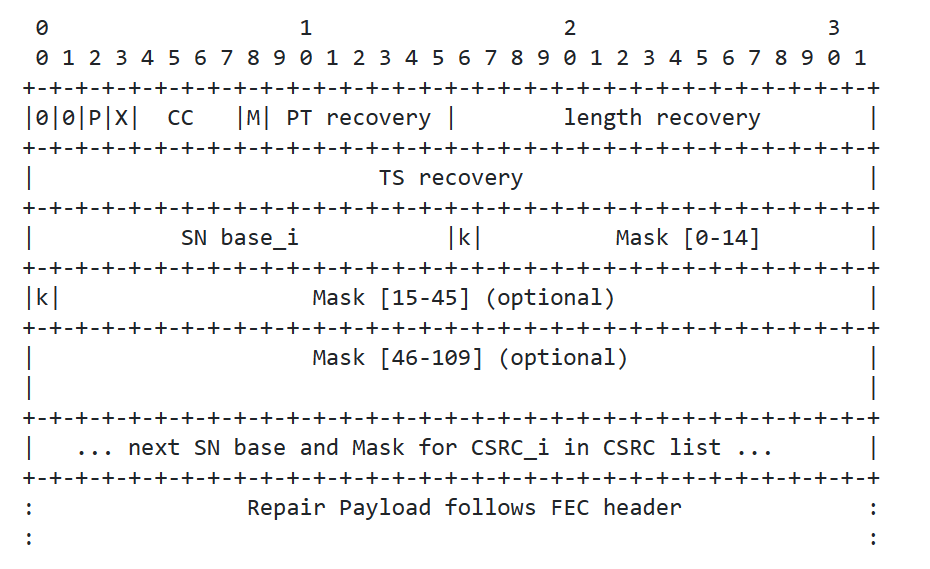
pkt\_mask\_idx **+=** media\_pkt\_idx **/** 8;

media\_pkt\_idx **%=** 8;

}

}}

XorHeaders是用来生成FEC头部，结合上一篇RTP封装介绍：



FEC Header for F=0

以下的代码就不用多介绍了：

**void** ForwardErrorCorrection**::**XorHeaders(**const** Packet**&** src, Packet**\*** dst) {

**uint8\_t\*** dst\_data **=** dst**->**data.MutableData();

**const** **uint8\_t\*** src\_data **=** src.data.cdata();

*// XOR the first 2 bytes of the header: V, P, X, CC, M, PT fields.* dst\_data[0] **^=** src\_data[0];

dst\_data[1] **^=** src\_data[1];

*// XOR the length recovery field.* **uint8\_t** src\_payload\_length\_network\_order[2];

ByteWriter**<uint16\_t>::**WriteBigEndian(src\_payload\_length\_network\_order,

src.data.size() **-** kRtpHeaderSize);

dst\_data[2] **^=** src\_payload\_length\_network\_order[0];

dst\_data[3] **^=** src\_payload\_length\_network\_order[1];

*// XOR the 5th to 8th bytes of the header: the timestamp field.* dst\_data[4] **^=** src\_data[4];

dst\_data[5] **^=** src\_data[5];

dst\_data[6] **^=** src\_data[6];

dst\_data[7] **^=** src\_data[7];

*// Skip the 9th to 12th bytes of the header.*}

XorPayloads就是无差别地对所有数据做XOR处理，不同的RTP数据有长有段，对于短的数据会不参与计算（这里不需要补0）:

**void** ForwardErrorCorrection**::**XorPayloads(**const** Packet**&** src,

size\_t payload\_length,

size\_t dst\_offset,

Packet**\*** dst) {

**if** (dst\_offset **+** payload\_length **>** dst**->**data.size()) {

dst**->**data.SetSize(dst\_offset **+** payload\_length);

}

**uint8\_t\*** dst\_data **=** dst**->**data.MutableData();

**const** **uint8\_t\*** src\_data **=** src.data.cdata();

**for** (size\_t i **=** 0; i **<** payload\_length; **++**i) {

dst\_data[dst\_offset **+** i] **^=** src\_data[kRtpHeaderSize **+** i];

}}

FEC的打包成ULP FEC还是FlexFEC，使用UlpfecHeaderWriter和FlexfecHeaderWriter[序列化](https://zhida.zhihu.com/search?content_id=222740587&content_type=Article&match_order=1&q=%E5%BA%8F%E5%88%97%E5%8C%96&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)。就只是遵循标准的实现。

### **2.2 FEC解码**

**void** ForwardErrorCorrection**::**DecodeFec(**const** ReceivedPacket**&** received\_packet,

RecoveredPacketList**\*** recovered\_packets) {

**const** size\_t max\_media\_packets **=** fec\_header\_reader\_**->**MaxMediaPackets();

*// 接收到的包已经到FEC保护最大长度，如果seq gap超过了这个长度* *// 此时需要reset FEC解码* **if** (recovered\_packets**->**size() **==** max\_media\_packets) {

**const** RecoveredPacket**\*** back\_recovered\_packet **=** recovered\_packets**->**back().get();

**if** (received\_packet.ssrc **==** back\_recovered\_packet**->**ssrc) {

**const** **unsigned** **int** seq\_num\_diff **=**

MinDiff(received\_packet.seq\_num, back\_recovered\_packet**->**seq\_num);

**if** (seq\_num\_diff **>** max\_media\_packets) {

*// A big gap in sequence numbers. The old recovered packets* *// are now useless, so it's safe to do a reset.* ResetState(recovered\_packets);

}

}

}

*// 插入数据，media包/FEC包*  InsertPacket(received\_packet, recovered\_packets);

*// 尝试恢复包* AttemptRecovery(recovered\_packets);}

解码恢复丢失的包，每次恢复一个，恢复一个即删除掉该FEC，再从头开始，直至所有包都恢复，和我们学过的[高斯消元法](https://zhida.zhihu.com/search?content_id=222740587&content_type=Article&match_order=1&q=%E9%AB%98%E6%96%AF%E6%B6%88%E5%85%83%E6%B3%95&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)比较类似：

**void** ForwardErrorCorrection**::**AttemptRecovery(

RecoveredPacketList**\*** recovered\_packets) {

**auto** fec\_packet\_it **=** received\_fec\_packets\_.begin();

**while** (fec\_packet\_it **!=** received\_fec\_packets\_.end()) {

*// FEC包带有mask，可以知道保护了哪些媒体包* *// 根据seq base和mask，以及收到的媒体包，确定还需要几个媒体包才能恢复* **int** packets\_missing **=** NumCoveredPacketsMissing(**\*\***fec\_packet\_it);

*// 一次只能恢复一个媒体包，丢包超过2个不能恢复，就先判断其他的FEC* **if** (packets\_missing **==** 1) {

std**::**unique\_ptr**<**RecoveredPacket**>** recovered\_packet(**new** RecoveredPacket());

recovered\_packet**->**pkt **=** **nullptr**;

*// 恢复XOR是逆运算，可以参考上面编码* *// 需要注意填充RTP的一些固定字段，如version* **if** (**!**RecoverPacket(**\*\***fec\_packet\_it, recovered\_packet.get())) {

fec\_packet\_it **=** received\_fec\_packets\_.erase(fec\_packet\_it);

**continue**;

}

**auto\*** recovered\_packet\_ptr **=** recovered\_packet.get();

recovered\_packets**->**push\_back(std**::**move(recovered\_packet));

recovered\_packets**->**sort(SortablePacket**::**LessThan());

*// 恢复了媒体包，在所有用到此媒体包的FEC中置标志位* UpdateCoveringFecPackets(**\***recovered\_packet\_ptr);

DiscardOldRecoveredPackets(recovered\_packets);

fec\_packet\_it **=** received\_fec\_packets\_.erase(fec\_packet\_it);

*// 一旦FEC恢复了包后，缺少2个包不能恢复，现在可能可以恢复* *// 因此从头开始再判断一次* fec\_packet\_it **=** received\_fec\_packets\_.begin();

} **else** **if** (packets\_missing **==** 0 **||**

IsOldFecPacket(**\*\***fec\_packet\_it, recovered\_packets)) {

*// 当前mask下，所有的媒体包都收到或者已经恢复，则删除掉这个FEC（mask）* *// 处理下一个FEC（mask）* fec\_packet\_it **=** received\_fec\_packets\_.erase(fec\_packet\_it);

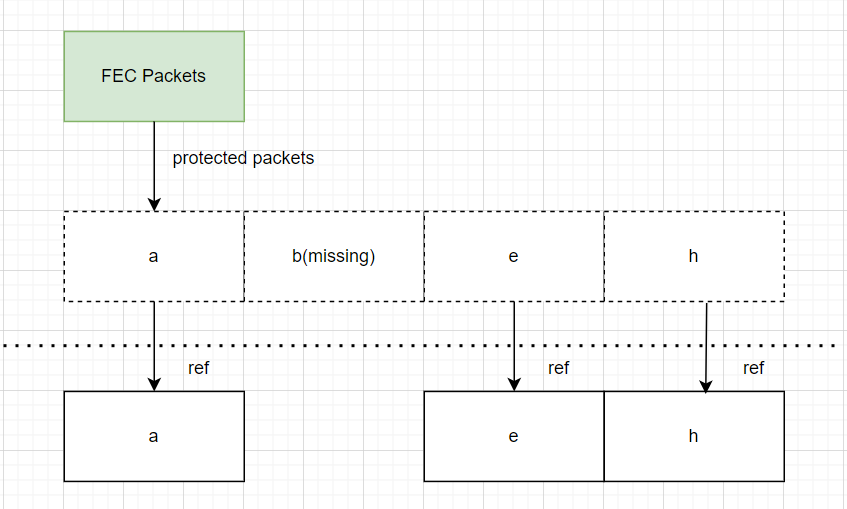
} **else** {

*// 丢包超过一个，暂时还不能恢复，先恢复其他的* fec\_packet\_it**++**;

}

}}

收到的FEC处理结构如下，我们会根据FEC的mask，生成一个其保护的包列表，一旦收到一个包，就会填充它。NumCoveredPacketsMissing判断有几个包丢失，也是根据这个列表来判断。



FEC存储结构

### **2.3 FEC的RTP头部处理**

头部处理比较简单，这里不多介绍，大家可以阅读UlpfecHeaderWriter和FlexfecHeaderWriter这两部分代码，这部分就是完全按照标准来填充对应字段，建议结合标准来看。

## **3. 总结**

以上便是WebRTC的FEC流程，相对比较简单。阅读这些对于后续讲解RS FEC的[编码原理](https://zhida.zhihu.com/search?content_id=222740587&content_type=Article&match_order=1&q=%E7%BC%96%E7%A0%81%E5%8E%9F%E7%90%86&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)有很大帮助，下一篇我们就好好介绍下RS FEC编码的处理流程。