目录

[一． 音视频基础 3](#_Toc24898)

[1. 基础概念介绍 3](#_Toc28556)

[1.1 GOP（Group of picture） 3](#_Toc17416)

[1.2 CABAC / CAVLC 3](#_Toc31801)

[1.3 Deblocking / FORCE\_IDR 3](#_Toc22440)

[1.4 frame / tff / bff 4](#_Toc2691)

[1.5 码流 / 码率 4](#_Toc28393)

[1.6 采样率 4](#_Toc9603)

[1.7 比特率 5](#_Toc1603)

[1.8 常见编码模式 5](#_Toc1674)

[1.9 帧速率 6](#_Toc19861)

[1.10 分辨率 6](#_Toc21122)

[1.11 高清视频 6](#_Toc24704)

[1.12 QP 6](#_Toc8037)

[1.13 PSNR 7](#_Toc10892)

[1.14 profile level 7](#_Toc27205)

[1.15 Reference 8](#_Toc7555)

[1.16 主码流/副码流 8](#_Toc10938)

[1.17 I P B帧 8](#_Toc21421)

[1.18 MPEG-4协议（ISO/IEC 14496标准） 9](#_Toc22204)

[1.19 PCR 11](#_Toc17610)

[1.20 数据冗余类型 11](#_Toc547)

[1.21 视频压缩编码的基本技术 11](#_Toc11261)

[1.22 h261 h263标准 MPEG等其他标准 12](#_Toc24431)

[2. 音视频入门 13](#_Toc28173)

[2.1 音视频通信流程 13](#_Toc21378)

[2.2视音频格式简介 14](#_Toc2724)

[2.3 DTS&PTS 15](#_Toc7279)

[2.4 mpeg4视频中，I帧、p帧、B帧的判定 15](#_Toc14441)

[3. 图片入门 16](#_Toc19349)

[二．音视频格式与传输协议 16](#_Toc2872)

[1. MPEG2 PS&TS 17](#_Toc5517)

[1.1简介 17](#_Toc15550)

[1.2 PS流格式 19](#_Toc20170)

[1.3 TS流格式 42](#_Toc13118)

[1.4 PTS DTS PCR同步机制 52](#_Toc25035)

[2. H.264 60](#_Toc32598)

[2.1 简介 60](#_Toc13396)

[2.2 NALU H264网络传输的结构 76](#_Toc4334)

[2.3 H.264码流结构图 91](#_Toc23312)

[2.4 H.264 slice header格式 98](#_Toc9415)

[2.5 H264 Slice Body宏块结构 99](#_Toc14206)

[2.6 sps pps slice header数据解析 101](#_Toc14135)

[3. RTP 103](#_Toc14880)

[3.1 RTP介绍 103](#_Toc28186)

[3.2 RTP数据格式 104](#_Toc4927)

[3.3 RTP会话 108](#_Toc15107)

[3.4 RTP荷载H264码流 108](#_Toc11978)

[3.5 RTP荷载PS流 111](#_Toc16428)

[3.6 RTP的时间戳 同步 111](#_Toc31759)

[3.7 RTP打包方式 114](#_Toc5564)

[4. RTSP 115](#_Toc18532)

[4.1 RTSP概述 115](#_Toc14933)

[4.2 RTSP状态 116](#_Toc27642)

[4.3 资源预订协议RSVP协议 116](#_Toc4367)

[7. AAC音频格式 116](#_Toc17012)

[5.1 音频初步介绍 116](#_Toc8264)

[5.2 AAC 音频格式 117](#_Toc29939)

[5.3 AAC音频文件格式 119](#_Toc8176)

[5.4 AAC 与 MP3比较 124](#_Toc2480)

[8. AVI封装格式 125](#_Toc6970)

[6.1 AVI概述 125](#_Toc3434)

[6.2 AVI Layout(AVI文件层次划分) 128](#_Toc18654)

[9. MP4封装格式 134](#_Toc3369)

[7.1 MP4概述 134](#_Toc5703)

[7.2 MP4结构 134](#_Toc2849)

[7.3 MP4术语解释及知识补充 150](#_Toc1384)

[7.4 Mp4info工具各个参数介绍： 150](#_Toc12227)

[7.5MP4中如何seek 151](#_Toc14865)

[三．编解码 151](#_Toc16443)

[1.FFMPEG 151](#_Toc15088)

[1.1API学习整理 151](#_Toc28136)

[1.2 ffmpeg问题 152](#_Toc19370)

# 音视频基础

## 基础概念介绍

### 1.1 GOP（Group of picture）

两个IDR帧之间的距离。增加关键帧个数可改善质量，但是同时增加带宽和网络负载。GOP=30，30FPS，即每个I帧之间相差1s。需要说明的是，通过提高GOP值来提高图像质量是有限度的，在遇到场景切换的情况时，H.264编码器会自动强制插入一个I帧，此时实际的GOP值被缩短了。另一方面，在一个GOP中，P、B帧是由I帧预测得到的，当I帧的图像质量比较差时，直到下一个GOP开始才有可能得以恢复，所以GOP值也不宜设置过大。过长的GOP还会影响Seek操作的响应速度，Seek操作需要先解码得到本GOP内的I帧及之前的N个预测帧才可以，GOP值越长，需要解码的预测帧就越多，seek响应的时间也越长。

### 1.2 CABAC / CAVLC 帧场编码

H.264/AVC标准中两种熵编码方法：CABAC叫自适应二进制算数编码，CAVLC叫前后自适应可变长度编码。其实就是帧编码和场编码。

CABAC：是一种无损编码方式，画质好，X264就会舍弃一些较小的DCT系数，码率降低，可以将码率再降低10-15%（特别是在高码率情况下），会降低编码和解码的速速。

CAVLC：占用更少的CPU资源，但会影响压缩性能。

帧：当采样视频信号时，如果是通过逐行扫描，那么得到的信号就是一帧图像，通常帧频为25帧每秒（PAL制）、30帧每秒（NTSC制）；

场：当采样视频信号时，如果是通过隔行扫描（奇、偶数行），那么一帧图像就被分成了两场，通常场频为50Hz（PAL制）、60Hz（NTSC制）；

帧频、场频的由来：最早由于抗干扰和滤波技术的限制，电视图像的场频通常与电网频率（交流电）相一致，于是根据各地交流电频率不同就有了欧洲和中国等PAL制的50Hz和北美等NTSC制的60Hz，但是现在并没有这样的限制了，帧频可以和场频一样，或者场频可以更高。

帧编码、场编码方式：逐行视频帧内邻近行空间相关性较强，因此当活动量非常小或者静止的图像比较适宜采用帧编码方式；而场内相邻行之间的时间相关性较强，对运动量较大的运动图像则适宜采用场编码方式。

### 1.3 Deblocking / FORCE\_IDR

Deblocking：开启会减少块效应

FORCE\_IDR：是否让每个I帧变成IDR帧，如果是IDR帧，支持随机访问。

### 1.4 frame / tff / bff

frame 将两场合并作为一帧进行编码

tff Enable interlaced mode (开启隔行编码并设置上半场在前)

bff Enable interlaced mode。

PAFF 和MBAFF：当对隔行扫描图像进行编码时，每帧包括两个场，由于两个场之间存在较大的扫描间隔，这样，对运动图像来说，帧中相邻两行之间的空间相关性相对于逐行扫描时就会减小，因此这时对两个场分别进行编码会更节省码流。

对帧来说，存在三种可选的编码方式：将两场合并作为一帧进行编码(frame 方式)或将两场分别编码(field 方式)或将两场合并起来作为一帧，但不同的是将帧中垂直相邻的两个宏块合并为宏块对进行编码；前两种称为PAFF 编码，对运动区域进行编码时field 方式有效，对非运区域编码时，由于相邻两行有较大的相关性，因而frame 方式会更有效。当图像同时存在运动区域和非运动区域时，在MB 层次上，对运动区域采取field 方式，对非运动区域采取frame 方式会更加有效，这种方式就称为MBAFF，预测的单位是宏块对。

### 1.5 码流 / 码率

码流(Data Rate)是指视频文件在单位时间内使用的数据流量，也叫码率或码流率，通俗就是取样率，画面质量控制中最重要的部分，一般是kb/s或者Mb/s。一般来说同样分辨率下，视频文件的码流越大，压缩比就越小，画面质量就越高，说明单位时间内取样率越大，精度就越高，处理出来的文件就越接近原始文件，图像质量越好，要求播放设备的解码能力也越高。

　　当然，码流越大，文件体积也越大，其计算公式是文件体积=时间X码率/8。例如，网络上常见的一部90分钟1Mbps码流的720P RMVB文件，其体积就=5400秒×1Mb/8=675MB。

　　通常来说，一个视频文件包括了画面及声音，而我们所说的一个视频文件码流率大小，一般是指视频文件中音频及视频信息码流率的总和。

以以国内最流行，大家最熟悉的RMVB视频文件为例，RMVB中的VB，指的是VBR，中文含义是可变比特率，表示RMVB采用的是动态编码的方式，把较高的采样率用于复杂的动态画面(歌舞、飞车、战争、动作等)，而把较低的采样率用于静态画面，达到画质与体积可兼得的效果。

### 1.6 采样率

采样率（也称为采样速度或者采样频率）定义了每秒从连续信号中提取并组成离散信号的采样个数，它用赫兹（Hz）来表示。采样率是指将模拟信号转换成数字信号时的采样频率，也就是单位时间内采样多少点。一个采样点数据有多少个比特。比特率是指每秒传送的比特(bit)数。单位为 bps(Bit Per Second)，比特率越高，传送的数据越大，音质越好.比特率 =采样率 x 采用位数 x声道数.

采样率类似于动态影像的帧数，比如电影的采样率是24赫兹，PAL制式的采样率是25赫兹，NTSC制式的采样率是30赫兹。当我们把采样到的一个个静止画面再以采样率同样的速度回放时，看到的就是连续的画面。同样的道理，把以44.1kHZ采样率记录的CD以同样的速率播放时，就能听到连续的声音。显然，这个采样率越高，听到的声音和看到的图像就越连贯。当然，人的听觉和视觉器官能分辨的采样率是有限的，基本上高于44.1kHZ采样的声音，绝大部分人已经觉察不到其中的分别了。

而声音的位数就相当于画面的颜色数，表示每个取样的数据量，当然数据量越大，回放的声音越准确，不过受人的器官的机能限制，16位的声音和24位的画面基本已经是普通人类的极限了，更高位数就只能靠仪器才能分辨出来了。比如电话就是3kHZ取样的7位声音，而CD是44.1kHZ取样的16位声音，所以CD就比电话更清楚。

当你理解了以上这两个概念，比特率就很容易理解了。以电话为例，每秒3000次取样，每个取样是7比特，那么电话的比特率是21000。 而CD是每秒 44100次取样，两个声道，每个取样是13位PCM编码，所以CD的比特率是44100\*2\*13=1146600，也就是说CD每秒的数据量大约是 144KB，而一张CD的容量是74分等于4440秒，就是639360KB＝640MB。

码率和取样率最根本的差别就是码率是针对源文件来讲的。

### 1.7 比特率

比特率是指每秒传送的比特(bit)数。单位为bps(Bit Per Second)，比特率越高，传送的数据越大。在视频领域,比特率常翻译为码率 !!!

比特率表示经过编码（压缩）后的音、视频数据每秒钟需要用多少个比特来表示，而比特就是二进制里面最小的单位，要么是0，要么是1。比特率就是比特率越高，音、视频的质量就越好，但编码后的文件就越大；如果比特率越少则情况刚好相反。

比特率是指数字声音、视频由模拟格式转化成数字格式的采样率，采样率越高，还原后的音质、画质就越好。

### 1.8 常见编码模式

VBR（Variable Bitrate）动态比特率 也就是没有固定的比特率，压缩软件在压缩时根据音频数据即时确定使用什么比特率，这是以质量为前提兼顾文件大小的方式，推荐编码模式；

ABR（Average Bitrate）平均比特率 是VBR的一种插值参数。LAME针对CBR不佳的文件体积比和VBR生成文件大小不定的特点独创了这种编码模式。ABR在指定的文件大小内，以每50帧（30帧约1秒）为一段，低频和不敏感频率使用相对低的流量，高频和大动态表现时使用高流量，可以做为VBR和CBR的一种折衷选择。

CBR（Constant Bitrate），常数比特率 指文件从头到尾都是一种位速率。相对于VBR和ABR来讲，它压缩出来的文件体积很大，而且音质相对于VBR和ABR不会有明显的提高。

### 1.9 帧速率

帧速率也称为FPS(Frames PerSecond)的缩写——帧/秒。是指每秒钟刷新的图片的帧数，也可以理解为图形处理器每秒钟能够刷新几次。越高的帧速率可以得到更流畅、更逼真的动画。每秒钟帧数(FPS)越多，所显示就会越流畅。

### 1.10 分辨率

帧大小每一帧就是一副图像。640\*480分辨率的视频，建议视频的码速率设置在700以上，音频采样率44100就行了。

一个音频编码率为128Kbps，视频编码率为800Kbps的文件，其总编码率为928Kbps，意思是经过编码后的数据每秒钟需要用928K比特来表示。

计算输出文件大小公式：（音频编码率（KBit为单位）/8 +视频编码率（KBit为单位）/8）×影片总长度（秒为单位）=文件大小（MB为单位）。不同的编码需要不同的系统资源，大概可以认为是H.264>VC-1>MPEG2。相对于MPEG2，VC-1的压缩比更高，但相对于H.264而言，编码解码的计算则要稍小一些，一般来说，VC-1多为 “.wmv”后缀，但这都不是绝对的，具体的编码格式还是要通过软件来查询。

总的来说，从压缩比上来看，H.264的压缩比率更高一些，也就是同样的视频，通过H.264编码算法压出来的视频容量要比VC-1的更小，但是VC-1 格式的视频在解码计算方面则更小一些，一般通过高性能的CPU就可以很流畅的观看高清视频。相信这也是目前NVIDIA Geforce 8系列显卡不能完全解码VC-1视频的主要原因。

### 1.11 高清视频

目前的高清视频编码格式主要有H.264、VC-1、MPEG-2、MPEG-4、DivX、XviD、WMA-HD以及X264。

事实上，现在网络上流传的高清视频主要以两类文件的方式存在：一类是经过MPEG-2标准压缩，以tp和ts为后缀的视频流文件;一类是经过WMV-HD(Windows Media Video HighDefinition)标准压缩过的wmv文件，还有少数文件后缀为avi或mpg，其性质与wmv是一样的。真正效果好的高清视频更多地以H.264与VC-1这两种主流的编码格式流传。一般来说，H.264格式以“.avi”、“.mkv”以及“.ts”封装比较常见。

### 1.12 QP

介于0~31之间，值越小，量化越精细，图像质量就越高，而产生的码流也越长。

### 1.13 PSNR

允许计算峰值信噪比(PSNR,Peak signal-to-noise ratio),编码结束后在屏幕上显示PSNR计算结果。开启与否与输出的视频质量无关，关闭后会带来微小的速度提升。

### 1.14 profile level

 MPEG-2按压缩比大小的不同分成五个档次（profile），每一个档次又按图像清晰度的不同分成四种图像格式，或称为级别（level）。五个档次四种级别共有20种组合，但实际应用中有些组合不太可能出现，较常用的是11种组合（下表中标识“√”的项）。这11种组合分别应用在不同的场合，如MP@ML。

分别是BP、EP、MP、HP：  
　　1、BP-Baseline Profile：基本画质。支持I/P 帧，只支持无交错（Progressive）和CAVLC；  
　　2、EP-Extended profile：进阶画质。支持I/P/B/SP/SI 帧，只支持无交错（Progressive）和CAVLC；  
　　3、MP-Main profile：主流画质。提供I/P/B 帧，支持无交错（Progressive）和交错（Interlaced），也支持CAVLC 和CABAC 的支持；  
　　4、HP-High profile：高级画质。在main Profile 的基础上增加了8x8内部预测、自定义量化、无损视频编码和更多的YUV 格式；

H.264规定了三种档次，每个档次支持一组特定的编码功能，并支持一类特定的应用。

1）基本档次：利用I片和P片支持帧内和帧间编码，支持利用基于上下文的自适应的变长编码进行的熵编码（CAVLC）。主要用于可视电话、会议电视、无线通信等实时视频通信；

2）主要档次：支持隔行视频，采用B片的帧间编码和采用加权预测的帧内编码；支持利用基于上下文的自适应的算术编码（CABAC）。主要用于数字广播电视与数字视频存储；

3）扩展档次：支持码流之间有效的切换（SP和SI片）、改进误码性能（数据分割），但不支持隔行视频和CABAC。主要用于网络的视频流，如视频点播。

### 1.15 Reference

Reference：指两个P帧之间的距离。

### 1.16 主码流/副码流

主码流/副码流：主码流位率高，图像质量高，便于本地存储；副码流位率低，图像质量低，便于网络传输。

### 1.17 I P B帧

I frame ：帧内编码帧 又称intra picture，I 帧通常是每个 GOP（MPEG 所使用的一种视频压缩技术）的第一个帧，经过适度地压缩，做为随机访问的参考点，可以当成图象。I帧可以看成是一个图像经过压缩后的产物。

P frame: 前向预测编码帧 又称predictive-frame，通过充分将低于图像序列中前面已编码帧的时间冗余信息来压缩传输数据量的编码图像，也叫预测帧；

B frame: 双向预测内插编码帧 又称bi-directional interpolated prediction frame，既考虑与源图像序列前面已编码帧，也顾及源图像序列后面已编码帧之间的时间冗余信息来压缩传输数据量的编码图像，也叫双向预测帧；

PTS：Presentation Time Stamp。PTS主要用于度量解码后的视频帧什么时候被显示出来

DTS：Decode Time Stamp。DTS主要是标识读入内存中的bit流在什么时候开始送入解码器中进行解码。

在没有B帧存在的情况下DTS的顺序和PTS的顺序应该是一样的。

I frame:自身可以通过视频解压算法解压成一张单独的完整的图片。

P frame：需要参考其前面的一个I frame 或者P frame来生成一张完整的图片。

B frame:则要参考其前一个I或者P帧及其后面的一个P帧来生成一张完整的图片。

如上图：I frame 的解码不依赖于任何的其它的帧.而p frame的解码则依赖于其前面的I frame或者P frame。B frame的解码则依赖于其前的最近的一个I frame或者P frame 及其后的最近的一个P frame。

对于I帧则PTS=DTS，P帧的PTS > DTS，B帧PTS < DTS。当然这里的大于和小于是相对BP帧而言。时间戳小则意味着先解码或者先显示，值越大意味着后处理。

### 1.18 MPEG-4协议（ISO/IEC 14496标准）

MPEG-4是个协议簇，包含了多个协议。



MPEG-4中只是定义了一部分AAC类型，AAC主要包括MPEG-2 AAC和MPEG-4 AAC, 详细如下图:



可以看出MPEG-4中只是定义了一部分类型的AAC.

MP4V在MPEG-4 的第2部分MPEG-4 Video中定义。

AVC在MPEG-4 的第10部分H264 or MPEG-4 AVC定义。

另: MP4A在MPEG-4的第三部分MPEG-4 Audio中定义。与MP4V相比，AVC具备更高的压缩效率。MP4V是原始视频数据。

MP4V-ES是RTP数据, 可通过RTP协议进行传输。

ISO/IEC 14496是MPEG专家组制定的MPEG-4标准，分为多个部分（仍在更新）。

参考：https://en.wikipedia.org/wiki/Category:ISO/IEC\_14496

第一部分（ISO/IEC 14496-1）：系统：描述视频和音频数据流的控制、同步以及混合方式（即混流Multiplexing，简写为MUX）。

第二部分（ISO/IEC 14496-2）：视频：定义一个对各种视觉信息（包括自然视频、静止纹理、计算机合成图形等等）的编解码器。（例如XviD编码就属于MPEG-4 Part 2）

第三部分（ISO/IEC 14496-3）：音频：定义一个对各种音频信号进行编码的编解码器的集合。包括高级音频编码（Advanced Audio Coding，缩写为AAC）的若干变形和其他一些音频/语音编码工具。

第四部分（ISO/IEC 14496-4）：一致性：定义对本标准其他的部分进行一致性测试的程序。

第五部分（ISO/IEC 14496-5）：参考软件：提供用于演示功能和说明本标准其他部分功能的软件。

第六部分（ISO/IEC 14496-6）：多媒体传输集成框架（DMIF for Delivery Multimedia Integration Framework）

第七部分（ISO/IEC 14496-7）：优化的参考软件：提供对实现进行优化的例子（这里的实现指的是第五部分）。

第八部分（ISO/IEC 14496-8）：在IP网络上传输：定义在IP网络上传输MPEG-4内容的方式。

第九部分（ISO/IEC 14496-9）：参考硬件：提供用于演示怎样在硬件上实现本标准其他部分功能的硬件设计方案。

第十部分（ISO/IEC 14496-10）：高级视频编码或称高级视频编码（Advanced Video Coding，缩写为AVC）：定义一个视频编解码器（codec）。AVC和XviD都属于MPEG-4编码，但由于AVC属于MPEG-4 Part 10，在技术特性上比属于MPEG-4 Part2的XviD要先进。另外，它和ITU-T H.264标准是一致的，故又称为H.264。

第十二部分（ISO/IEC 14496-12）：基于ISO的媒体文件格式：定义一个存储媒体内容的文件格式。

第十三部分（ISO/IEC 14496-13）：知识产权管理和保护（IPMP for Intellectual Property Management and Protection）拓展。

第十四部分（ISO/IEC 14496-14）：MPEG-4文件格式：定义基于第十二部分的用于存储MPEG-4内容的视频文件格式。

第十五部分（ISO/IEC 14496-15）：AVC文件格式：定义基于第十二部分的用于存储第十部分的视频内容的文件格式。

第十六部分（ISO/IEC 14496-16）：动画框架扩展（AFX : Animation Framework eXtension）。

第十七部分（ISO/IEC 14496-17）：同步文本字幕格式。

第十八部分（ISO/IEC 14496-18）：字体压缩和流式传输（针对开放字体格式Open Font Format）。

第十九部分（ISO/IEC 14496-19）：合成材质流（Synthesized Texture Stream）。

第二十部分（ISO/IEC 14496-20）：简单场景表示（LASeR for Lightweight Scene Representation。

第二十一部分（ISO/IEC 14496-21）：用于描绘（Rendering）的MPEG-J拓展。

第二十二部分（ISO/IEC 14496-22）：开放字体格式（Open Font Format）。

第二十三部分（ISO/IEC 14496-23）：符号化音乐表示（Symbolic Music Representation）。

第二十四部分（ISO/IEC 14496-24）：音频与系统交互作用（Audio and systems interaction）。

第二十五部分（ISO/IEC 14496-25）：3D图形压缩模型（3D Graphics Compression Model）。

第二十六部分（ISO/IEC 14496-26）：音频一致性检查：定义测试音频数据与ISO/IEC 14496-3是否一致的方法（Audio conformance）。

第二十七部分（ISO/IEC 14496-27）：3D图形一致性检查：定义测试3D图形数据与ISO/IEC 14496-11:2005, ISO/IEC 14496-16:2006, ISO/IEC 14496-21:2006,和ISO/IEC 14496-25:2009是否一致的方法（3D Graphics conformance）。

MP4是在“ISO/IEC 14496-14”标准文件中定义的一种多媒体容器格式，它是MPEG4 （ISO/IEC 14496）标准的一部分。是“ISO/IEC 14496-12(MPEG-4 Part 12 ISO base media file format)”标准中所定义的媒体格式的一种实现。

### 1.19 PCR

PCR 仅仅在实时播出流中才有意义（即前端编码器实时编码或视频服务器实时播出，后端机顶盒实时接收的情况），对DVD机这类从文件中取TS流的设备PCR毫无意义。

前端编码器工作原理：编码器中有一个27M的时钟，编码过程中不断读取硬件时钟放入TS中PCR字段，和PES的PTS，DTS字段。

后端机顶盒工作原理：机顶盒中也有一个27M的时钟，它根据PES中的PTS，DTS字段中指示的时间进行解码和显示。那么TS中的PCR还有什么用呢？首先如果前端编码器的时钟与后端机顶盒中时钟"绝对"同步，那么TS中的PCR就没有任何意义了。但是如果"绝对"变对"相对"以后会出先什么情况？它们之间的"微小"误差经过长时间的累积（1-2小时就足以）机顶盒中解码器会因为自己时钟"快"了而造成buffer中没有数据--停帧，会因为时钟"慢"了而造成buffer中数据溢出--丢帧。所对它需要TS中的PCR字段来不断修正自己时钟与编码器时钟间的"微小"误差。

TS流在经过复用器，加扰机这些中间设备时，PCR"必须"进行"非均匀"延迟修正（原因如上）。所以对于没有PCR修正功能的复用器和加扰机，建设大家一定不要买。

为什么说TS流中的PCR对DVD机毫无意义呢？DVD机是从光盘中读取TS流，读盘速度与TS流传送速率毫无关联，PCR还有指示时间的意义吗？但因为它不是"实时流"传送，所以不会出现buffer溢出的问题，也就不会产生停丢帧现象。

### 1.20 数据冗余类型

主要类型有：

时间冗余：视频相邻的两帧之间内容相似，存在运动关系

空间冗余：视频的某一帧内部的相邻像素存在相似性

编码冗余：视频中不同数据出现的概率不同

视觉冗余：观众的视觉系统对视频中不同的部分敏感度不同

### 1.21 视频压缩编码的基本技术

为了专门处理视频信息中的多种冗余，视频压缩编码采用了多种技术来提高视频的压缩比率。其中常见的有预测编码、变换编码和熵编码等。

1. 预测编码

预测编码可以用于处理视频中的时间和空间域的冗余。视频处理中的预测编码主要分为两大类：帧内预测和帧间预测。

帧内预测：预测值与实际值位于同一帧内，用于消除图像的空间冗余；帧内预测的特点是压缩率相对较低，然而可以独立解码，不依赖其他帧的数据；通常视频中的关键帧都采用帧内预测。

帧间预测：帧间预测的实际值位于当前帧，预测值位于参考帧，用于消除图像的时间冗余；帧间预测的压缩率高于帧内预测，然而不能独立解码，必须在获取参考帧数据之后才能重建当前帧。

通常在视频码流中，I帧全部使用帧内编码，P帧/B帧中的数据可能使用帧内或者帧间编码。

2. 变换编码

目前主流的视频编码算法均属于有损编码，通过对视频造成有限而可以容忍的损失，获取相对更高的编码效率。而造成信息损失的部分即在于变换量化这一部分。在进行量化之前，首先需要将图像信息从空间域通过变换编码变换至频域，并计算其变换系数供后续的编码。

在视频编码算法中通常使用正交变换进行变换编码，常用的正交变换方法有：离散余弦变换(DCT)、离散正弦变换(DST)、K-L变换等。

3. 熵编码

视频编码中的熵编码方法主要用于消除视频信息中的统计冗余。由于信源中每一个符号出现的概率并不一致，这就导致使用同样长度的码字表示所有的符号会造成浪费。通过熵编码，针对不同的语法元素分配不同长度的码元，可以有效消除视频信息中由于符号概率导致的冗余。

在视频编码算法中常用的熵编码方法有变长编码和算术编码等，具体来说主要有上下文自适应的变长编码(CAVLC)和上下文自适应的二进制算术编码(CABAC)。

### 1.22 h261 h263标准 MPEG等其他标准

H.261码流复合器输出的码流，总共可以分为四层，从上到下分别为帧层、块组层、宏块层和块层。每一层按照不同的封装格式包含了头信息和下一层的结构。

帧层：由帧首和块组数据构成；帧首包括帧起始码（ PSC）、帧计数码（ TR）、帧类型（ PTYPE）等；

块组层：由块组首和宏块数据组成；块组首包括组起始码（ GBSC）、块组编号码（ GN）、块组量化步长等；

宏块层(Macroblock,MB)：由宏块首和块数据构成；宏块首包括宏块地址码、宏块类型、宏块量化步长、运动矢量数据、编码模式等；

块层：包括每个8×8块的DCT系数按之字形扫描后的熵编码码流，以块结束符结尾

H.263是相对于H.261的改进标准，同样以低码率视频通信为目标，但是具有更好的压缩效率。与H.261相比，H.263支持更多种分辨率的图像格式：

Sub-QCIF: 128×96

QCIF: 176×144

CIF:352×288

4CIF:704×576

16CIF:1480×1152

MPEG-1是国际标准化组织ISO下属的移动图像专家组负责制定的早期视频压缩标准，主要用于视频信息的存储、广播电视和网络传输应用。其中在VCD中保存的视频信息便使用MPEG-1标准进行压缩，其码率越为1.2~1.5Mb/s。MPEG-1标准所支持的图像类型与H.263类似，支持I、B、P帧类型。在码流的结构上，MPEG-1采用了与H.261类似的分层码流结构。MPEG-1相比于H.261增加了条带层，每一个宏块所有的编码操作只能在一个条带内实现，条带层可以有效防止编码错误在一帧内扩散。

MPEG-2标准是ITU-T和ISO合作制定的编码标准，其视频部分也称作H.262标准，其标准编号为ISO-13818。ISO-13818是一系列标准的集合，包括了系统、视频、音频、一致性、参考软件等10个部分。另一方面，MPEG-2根据不同的编码工具定义了5个Profile：简单SP、主要MP、SNR可分级SNP、空间可分级SSP和高级HP。根据视频分辨率的不同定义了4个Level：低级LL、主级ML、高-1440级High1440和高级HL。MPEG-2的码流分为三层：

基本流：ES，由视频编码的视频基本流和音频编码的音频基本流构成

打包基本流：PES，为音视频ES分别打包

传输流、节目流：TS/PS，若干个PES进行复用后输出，分别用于传输和存储

在MPEG-2的ES流层中，其码流结构采用了类似MPEG-1码流结构的分层封装的方法：

图像序列层：包括若干GOP，序列头包含起始码和序列参数等；

图像组（GOP）层：包括若干图像，GOP头包括起始码、GOP标识等；

图像层：包括若干个Slice，图像头中包括起始码和图像参数等；

片（Slice）层：最小的同步单位，包括若干宏块，Slice头中包括起始码、片地址、量化步长等信息；

宏块（Macroblock）层：由4个亮度块和2个色度块组成，宏块头包括地址、类型、MV等信息

MPEG-4是ISO与1999年颁布的视频编码标准。同前任的MPEG-1和MPEG-2相比，MPEG-4更注重多媒体系统的交互性、灵活性和可扩展性。MPEG-4的标准编号为ISO-14496。MPEG-4中最为显著的特点是采用了基于对象的编码。在MPEG-4中，一个视频对象主要定义为画面中分割出来的不同物体，每个物体由三类信息描述：运动信息、轮廓信息和纹理信息。MPEG-4通过编码这三类信息来实现对视频对象的编码。

## 音视频入门

### 音视频通信流程



### 2.2视音频格式简介

视频、图像、音频的格式。包括MP4，FLV这样的封装格式，也包括H264，MPEG2这样的编码格式，还包括420 422 444这样的取样格式，甚至RGB，HSV，YUV这样的彩色空间也可以算作一种格式，将来在应用时还可能会遇到RTMP，HLS这样的流媒体传输协议或者其他的协议，都可以算作格式。

### 2.3 DTS&PTS

### 2.4 mpeg4视频中，I帧、p帧、B帧的判定

mpeg4的每一帧开头是固定的：00 00 01 b6，那么我们如何判断当前帧属于什么帧呢？在接下来的2bit，将会告诉我们答案。注意：是2bit，不是byte，下面是各类型帧与2bit的对应关系：

　　00: I Frame

　　01: P Frame

10: B Frame

为了更好地说明，我们举几个例子，以下是16进制显示的视频编码：

　　00 00 01 b6 10 34 78 97 09 87 06 57 87 …… I帧

　　00 00 01 b6 50 78 34 20 cc 66 b3 89 …… P帧

00 00 01 b6 96 88 99 06 54 34 78 90 98 …… B帧

下面我们来分析一下为什么他们分别是I、P、B帧

　　0x10 = 0001 0000

　　0x50 = 0101 0000

　　0x96 = 1001 0100

大家看红色的2bit，再对照开头说的帧与2bit的对应关系，是不是符合了呢？

下面给出一段c++代码供大家参考：

switch(buf[i] & (BYTE)0xc0)

{

case 0x00:

//I Frame

break;

case 0x40:

//P Frame

break;

case 0x80:

//B Frame

break;

default:

break;

}

## 图片入门

#### PNG

便携式网络图形（Portable Network Graphics）是一种无损压缩的位图图形格式。其设计目的是试图替代GIF和TIFF文件格式，同时增加一些GIF文件格式所不具备的特性。PNG使用从LZ77派生的无损数据压缩算法，一般应用于JAVA程序、网页或S60程序中，原因是它压缩比高，生成文件体积小。

PNG格式的图片具有如下特点：

体积小：网络通讯中因受带宽制约，保证图片清晰、逼真。

无损压缩：PNG文件采用LZ77算法获得高的压缩比，不损失数据。标记重复出现的数据，对图像的颜色没有影响，也不可能产生颜色的损失。

索引彩色模式：PNG-8格式与GIF图像类似，同样采用8位调色板将RGB彩色图像转换为索引彩色图像。图像中保存的不再是各个像素的彩色信息，而是从图像中挑选出来的具有代表性的颜色编号，每一编号对应一种颜色，图像的数据量也因此减少，这对彩色图像的传播非常有利。

更优化的网络传输显示： PNG图像在浏览器上采用流式浏览，即使经过交错处理的图像会在完全下载之前提供浏览者一个基本的图像内容，然后再逐渐清晰起来。它允许连续读出和写入图像数据，这个特性很适合于在通信过程中显示和生成图像。

支持透明效果：PNG可以为原图像定义256个透明层次，使得彩色图像的边缘能与任何背景平滑地融合，从而彻底地消除锯齿边缘。这种功能是GIF和JPEG没有的。

PNG同时还支持真彩和灰度级图像的Alpha通道透明度。

最高支持24位真彩色图像以及8位灰度图像。

支持Alpha通道的透明/半透明特性。

支持图像亮度的Gamma校准信息。

支持存储附加文本信息，以保留图像名称、作者、版权、创作时间、注释等信息。

渐近显示和流式读写，适合在网络传输中快速显示预览效果后再展示全貌。

使用CRC防止文件出错。

最新的PNG标准允许在一个文件内存储多幅图像。

# 二．音视频格式与传输协议

MPEG2系统任务包括：1. 规定以包传输数据的协议；2. 规定收发两端数据流同步的协议；3. 提供多个数据流的复用和解复用协议；3. 提供数据流加密的协议。以包形式存储和传送数据流是MPEG2系统之要点。

## 1. MPEG2 PS&TS

### 1.1简介

#### 1.1.1 PS TS ES介绍

1）ES ES--Elementary  Streams

原始流是直接从编码器出来的数据流，可以是编码过的视频数据流（H.264,MJPEG等），音频数据流（AAC），或其他编码数据流的统称。ES流经过PES打包器之后，被转换成PES包。每个ES都由若干个存取单元（AU）组成，每个视频AU或音频AU都是由头部和编码数据两部分组成，1个AU相当于编码的1幅视频图像或1个音频帧，也可以说，每个AU实际上是编码数据流的显示单元，即相当于解码的1幅视频图像或1个音频帧的取样。

1. PES PES--Packetized  Elementary Streams

ES形成的分组称为PES分组，是用来传递ES的一种数据结构。PES流是ES流经过PES打包器处理后形成的数据流，在这个过程中完成了将ES流分组、打包、加入包头信息等操作（对ES流的第一次打包）。PES流的基本单位是PES包。PES包由包头和payload组成。 3）

1. PS PS--Program Stream 节目流

PS流由PS包组成，而一个PS包又由若干个PES包组成（到这里，ES经过了两层的封装）。PS包的包头中包含了同步信息与时钟恢复信息。一个PS包最多可包含具有同一时钟基准的16个视频PES包和32个音频PES包。

1. TS TS--Transport Stream 传输流

由定长的TS包组成（188字节），而TS包是对PES包的一个重新封装（到这里，ES也经过了两层的封装）。PES包的包头信息依然存在于TS包中。TS流与PS流的区别在于TS流的包结构是固定长度的,而PS流的包结构是可变长度的。PS包由于长度是变化的,一旦丢失某一PS包的同步信息,接收机就会进入失步状态,从而导致严重的信息丢失事件。而TS码流由于采用了固定长度的包结构,当传输误码破坏了某一TS包的同步信息时,接收机可在固定的位置检测它后面包中的同步信息,从而恢复同步,避免了信息丢失。因此在信道环境较为恶劣、传输误码较高时一般采用TS码流,而在信环境较好、传输误码较低时一般采用PS码流。

PS&TS：两种视频或影片封装格式，常用于高清片。扩展名分别为VOB/EVO和TS等；其文件编码一般用MPEG2/VC-1/H.264。TS流与PS流的区别在于TS流的包结构是固定长度的，而PS流的包结构是可变长度的。

MPEG2(PS)就是常见dvd上那种vob格式的（PS是Program Stream程序流缩写），TS的全称则是Transport Stream（传输流），TS格式在网络上现在主要在一些高清视频，至pal和ntsc是两种不同的制式,二者在分辨率和帧数上有区别。MPEG2-PS主要应用于存储的具有固定时长的节目，如DVD电影，而MPEG-TS则主要应用于实时传送的节目，比如实时广播的电视节目。这两种格式的主要区别是什么呢？简单地打个比喻说，你将DVD上的VOB文件的前面一截cut掉（或者干脆就是数据损坏），那么就会导致整个文件无法解码了，而电视节目是你任何时候打开电视机都能解码（收看）的，所以，MPEG2-TS格式的特点就是要求从视频流的任一片段开始都是可以独立解码的。

PS包与TS包在结构上的这种差异，导致了它们对传输误码具有不同的抵抗能力，因而应用的环境也有所不同。TS码流由于采用了固定长度的包结构，当传输误码破坏了某一TS包的同步信息时，接收机可在固定的位置检测它后面包中的同步信息，从而恢复同步，避免了信息丢失。而PS包由于长度是变化的，一旦某一PS包的同步信息丢失，接收机无法确定下一包的同步位置，就会造成失步，导致严重的信息丢失。因此，在信道环境较为恶劣，传输误码较高时，一般采用TS码流；而在信道环境较好，传输误码较低时，一般采用PS码流。

ES是直接从编码器出来的数据流，可以是编码过的视频数据流，音频数据流，或其他编码数据流的统称。ES流经过PES打包器之后，被转换成PES包。PES包由包头和payload组成。PTS/DTS是打在PES包里面的，这两个parameters是解决视音频同步显示，防止解码器输入缓存上溢或下溢的关键。PTS表示显示单元出现在系统目标解码器(STD: system target

decoder)的时间，DTS表示将存取单元全部字节从STD的ES解码缓存器移走的时刻。每个I、P、B帧的包头都有一个PTS和DTS，但PTS与DTS对B帧都是一样的，无须标出B帧的DTS。对I帧和P帧，显示前一定要存储于视频解码器的重新排序缓存器中，经过延迟（重新排序）后再显示，一定要分别标明PTS和DTS。

ES首先需打包成PES流包，然后PES根据需要打包成PS或TS包进行存储或传输。其每路ES只包含一路信源的编码数据流，所以每路PES也只包含相对应信源的数据流。

对PS流而言，每个PES包头含有PTS和DTS，流识别码，用于区别不同性质ES。然后通过PS复用器将PES包复用成PS包。实际上是将PES包分解为更细小的PS包。在解码的时候，解复用器将PS分解成一个个PES包，拆包器然后将PES包拆成视频和音频的ES，最后输入至各自解码器进行解码。一个问题是：各个ES在解码时，如何保证视音频的同步呢？除了PTS和DTS的配合工作外，还有一个重要的参数是SCR(system clockreference)。在编码的时候，PTS，DTS和SCR都是由STC(system timeclock)生成的，在解码时，STC会再生，并通过锁相环路（PLL－phase lockloop），用本地SCR相位与输入的瞬时SCR相位锁相比较，以确定解码过程是否同步，若不同步，则用这个瞬时SCR调整27MHz的本地时钟频率。最后，PTS，DTS和SCR一起配合，解决视音频同步播放的问题。

TS流也是由一个或多个PES组合而来的，他们可以具有相同的时间基准，也可以不同。其基本的复用思想是，对具有相同时间基准的多个PES现进行节目复用，然后再对相互有独立时间基准的各个PS进行传输复用，最终产生出TS。

#### 1.1.2 参考链接：

1.PS流的格式和解析总结

https://blog.csdn.net/rankun1/article/details/69950873

2.MPEG-2 PS流

<https://blog.csdn.net/duanbeibei/article/details/1698183>

1. MPEG-TS 格式解析

https://blog.csdn.net/Kayson12345/article/details/81266587

1. 浅析mpeg-ts封装结构

<https://blog.csdn.net/michaeluo/article/details/75263462>

1. MPEG2-TS音视频同步原理(PCR dts pts)

<https://blog.csdn.net/charleslei/article/details/52299388>

### 1.2 PS流格式

将具有共同时间基准的一个或多个PES组合（复合）而成的单一的数据流称为节目流（Program Stream）。ES是直接从编码器出来的数据流，可以是编码过的视频数据流，音频数据流，或其他编码数据流的统称。ES流经过PES打包器之后，被转换成PES包。

针对H264 做PS 封装：每个IDR NALU 前一般都会包含SPS、PPS 等NALU，因此将SPS、PPS、IDR 的NALU 封装为一个PS 包，包括ps 头，然后加上PS system header，PS system map，PES header+h264 raw data。所以一个IDR NALU PS 包由外到内顺序是：

PSheader | PS system header | PS system Map | PES header | h264 raw data。

对于其它非关键帧的PS 包，就简单多了，直接加上PS头和PES 头就可以了。顺序为：

PS header | PES header | h264raw data。

以上是对只有视频video 的情况，如果要把音频Audio也打包进PS 封装，也可以。当有音频数据时，将数据加上PES header 放到视频PES 后就可以了。顺序如下：PS 包=PS头|PES(video)|PES(audio)，再用RTP 封装发送就可以了。

PES包长度域有两个字节，共16比特，因此PES包的最大长度是65535字节。PS不能在rtsp中运输只能用h264搭载rtsp。

**1.IDR关键帧封装：**

每个IDR NALU 前一般都会包含SPS、PPS等NALU，因此将SPS、PPS、IDR 的NALU 封装为一个PS 包，包括ps 头，然后加上PSsystem header，PS system map，PESheader+h264 raw data。

所以一个IDR NALU PS 包由外到内顺序是：

**PSheader|PS system header | PS system Map | PES header | h264 raw data。**

**2.非关键帧封装：**

PS包从外带内的顺序是：

**PSheader|PES header | h264 raw data。**

**3.含有音频的PS封装：**

PS头|PES(video)|PES(audio)

**PSheader|PES header | h264 raw data|PES header|AAC raw data**

#### 1.2.1 PS包结构

PS包结构由包头、系统头、系统映射、PES包四部分构成。具体参考以上。

包头由PS包起始码、系统时钟基准（SCR-System Clock Reference）的基本部分、SCR的扩展部分和PS复用速率4部分组成。

结构如下：

#### IMG_2561.2.2 PS包头PSheader

位串，用来标志一个包的开始PACK identifier -- 0xBA，marker\_bit=1



SCR和SCR\_ext：是系统时钟参考，一个以27MHz驱动的计数器，用作同步流的参考。时钟除以300以上（以匹配PTS / DTS等90KHz时钟），商为SCR（33位），余数余数为SCR\_ext（9位）

Program\_Mux\_Rate：这是一个22位整数，指定程序流目标解码器在包含它的包中接收节目流的速率。 program\_mux\_rate的值以50字节/秒为单位进行测量。值0被禁止。该字段所表示的值用于在2.5.2中定义P-STD输入端的字节到达时间。该字段值在本标准中的节目多路复合流的不同包中取值可能不同。

pack\_stuffing\_length：一个3位整数，指定此字段后面的填充字节数。

stuffing\_byte：填充字节，这是一个固定的8位值，等于'1111 1111'，可以由编码器插入，例如，以满足通道的要求。它被解码器丢弃。

**实例分析：**

**原始待分析数据：00 00 01 BA 44 00 04 00 0401 00 03 FF F8**

**1. Pack\_start\_code ps包开始标志位:**

十六进制：00 00 01 BA

标识一个PS包的开始;

2. **System\_clock\_reference系统时钟参考字段和扩展字段（六字节）**：系统时钟参考(SCR)分两部分编码的42位字段。第一部分system\_clock\_reference\_base是一个长度为33位的字段；第二部分system\_clock\_reference\_extenstion是一个长度为9位的字段。SCR字段指出了基本流中包含ESCR\_base最后一位的字节到达节目目标解码器输入端的期望时间。

十六进制：44 00 04 00 04 01

二进制：01-00 0-1-00 0000 00000000 0-1-00 0000 0000 0000 0-1-00 0000 000-1

即可以看到参考时钟还是参考时钟的扩展都是0;

3. **program\_mux\_rate字段节目符合速率字段（三字节）**：一个22位整数，规定P-STD在包含该字段的包期间接收节目流的速率。其值以50字节/秒为单位。不允许取0值。该字段值在本标准中的节目多路复合流的不同包中取值可能不同。

十六机制：00 03 FF

二进制：0000 00000000 0011 1111 1111

十进制：255 那么说明速率大小为255\*50=12750Byte/s

4. **pack\_stuffing\_length填充字段长度**：

十六机制：F8

二进制：1111 1000

说明是0则表示无填充字节。

**总结：**

1. ps头无论是什么样的包都是要有的，00 00 01 BA是关键，其它字段基本可以忽略不计；

2. 无扩展字段是直接读十四个字节跳过解析即可，但是当pack\_stuffing\_length不为0时，需要解析出来，加上该长度跳过。该字段位于第十四字节的后面三bit位；

3. 也可以根据该值进行打包和封装，基本对应bit位都是用默认值；

#### 1.2.3 PS系统头 PSsystem header：

位串，BB指出系统标题的开始

|  |  |  |
| --- | --- | --- |
| 语  法 | 位数 | 助记符 |
| system\_header() { |  |  |
| system\_header\_start\_code | 32 | bslbf |
| header\_length | 16 | uimsbf |
| marker\_bit | 1 | bslbf |
| rate\_bound | 22 | uimsbf |
| marker\_bit | 1 | bslbf |
| audio\_bound | 6 | uimsbf |
| fixed\_flag | 1 | bslbf |
| CSPS\_flag | 1 | bslbf |
| system\_audio\_lock\_flag | 1 | bslbf |
| system\_video\_lock\_flag | 1 | bslbf |
| marker\_bit | 1 | bslbf |
| vedio\_bound | 5 | uimsbf |
| packet\_rate\_restriction\_flag | 1 | bslbf |
| reserved\_bits | 7 | bslbf |
| while (nextbits()=='1') { |  |  |
| stream\_id | 8 | uimsbf |
| '11' | 2 | bslbf |
| P-STD\_buffer\_bound\_scale | 1 | bslbf |
| P-STD\_buffer\_size\_bound | 13 | uimsbf |
| } |  |  |
| } |  |  |

header\_length：标题长度字段，16位字段。指出该字段后的系统标题的字节长度。在本规范将来的扩充中可能扩展该字段。

rate\_bound ：速率界限字段，22位字段，取值不小于编码在节目流的任何包中的program\_mux\_rate字段的最大值。该字段可被解码器用于估计是否有能力对整个流解码。

audio\_bound：音频界限字段，6位字段，取值是在从0到32的闭区间中的整数，且不小于节目流中解码过程同时活动的GB/T XXXX.3和GB/T AAAA.3音频流的最大数目。在本小节中，若STD缓冲区非空或展现单元正在P-STD模型中展现，则GB/T XXXX.3和GB/T AAAA.3音频流的解码过程是活动的。

fixed\_flag：固定标志字段，1位标志位。置'1'时表示比特率恒定的操作；置'0'时，表示操作的比特率可变。在恒定比特率的操作期间，复合的GB/T XXXX.1流中的system\_clock\_reference字段值应遵从下面的线性公式:

SCR\_base(i)＝((c1×i＋c2) DIV 300) % 233 (2-22)

SCR\_ext(i)＝((c1×i＋c2) DIV 300) % 300 (2-23)

其中：

c1 对所有i均有效的实型常数；

c2 对所有i均有效的实型常数；

i 在GB/T XXXX.1复合流中包含任何system\_clock\_reference字段的最后一位的字节索引。

CSPS\_flag：CSPS标志字段，1位字段。置'1'时，节目流符合2.7.9中定义的限制

system\_audio\_lock\_flag：系统音频锁定标志字段，1位字段。表示在系统目标解码器的音频采样率和system\_clock\_frequency之间存在规定的比率。system\_clock\_frequency在2.5.2.1中定义而音频采样率由GB/T XXXX.3规定。如果对节目流中所有音频基本流的所有展现单元，system\_clock\_frequency和实际音频采样率的比例SCASR是恒定的，且对音频流中所指出的标准采样率和下表中数值相等，则该字段只能为'1'。

SCASR＝(system\_clock\_frequency) / audio\_sample\_rate\_in\_the\_P-STD (2-24)

记号X/Y表示实数除法。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 标准音频采  样频率(kHz) | 16 | 32 | 22.05 | 44.1 | 24 | 48 |
| SCASR | 27 000 000  --------  16 000 | 27 000 000  -------  32 000 | 27 000 000  ------  22 050 | 27 000 000  ------  44 100 | 27 000 000  ------  24 000 | 27 000 000  --------  48 000 |

system\_video\_lock\_flag：系统视频锁定标志字段，1位字段。表示在系统目标解码器的视频帧速率和system\_clock\_frequency之间存在规定的比率。system\_clock\_frequency在2.5.2.1中定义而视频帧速率由GB/T XXXX.2规定。如果对GB/T XXXX.1中所有视频基本流的所有展现单元，system\_clock\_frequency和实际视频帧速率的比例SCFR是恒定的，且对视频流中所指出的标准帧速率和下表中数值相等，则该字段只能为'1'。

SCFR＝system\_clock\_frequency / frame\_rate\_in\_the\_P-STD (2-25)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 标准帧速率(Hz) | 23.976 | 24 | 25 | 29.97 | 30 | 50 | 59.94 | 60 |
| SCFR | 1 126 125 | 1 125 000 | 1 080 000 | 900 900 | 900 000 | 540 000 | 450 450 | 450 000 |

比率SCFR的值是精确的。对于23.976，29.97或59.94帧/秒的标准速率，实际的帧速率与标准速率略有不同。

video\_bound：视频界限字段。5位字段，取值是在从0到16的闭区间中的整数且不小于节目流中解码过程同时活动的GB/T XXXX.2和GB/T AAAA.2流的最大数目。在本小节中，若P-STD缓冲区非空或展现单元正在P-STD模型中展现，则GB/T XXXX.2和GB/T AAAA.2视频流的解码过程是活动的。

packet\_rate\_restriction\_flag：分组速率限制标志字段，1位标志位。若CSPS标识为'1'，则该字段表示2.7.9中规定的哪个限制适用于分组速率。若CSPS标识为'0'，则该字段的含义未定义。

reserved\_bits：保留位字段，7位字段。被保留供ISO/IEC将来使用。它的值应为'111 1111'，除非ISO/IEC对它作出其它规定。

stream\_id：流标识字段，8位字段。指示其后的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound字段所涉及的流的编码和基本流号码。

若取值'1011 1000'，则其后的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound字段指节目流中所有的音频流。

若取值'1011 1001'，则其后的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound字段指节目流中所有的视频流。

若stream\_id取其它值，则应该是大于或等于'1011 1100'的一字节值且应根据表2-18解释为流的编码和基本流号码。

节目流中的每个基本流应在每个系统标题中通过这种机制精确地规定一次它的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound。

P-STD\_buffer\_bound\_scale：P-STD缓冲区界限比例字段。1位字段。表示用于解释后续P-STD\_buffer\_size\_bound字段的比例系数。若前面的stream\_id表示一个音频流，则该字段值为'0'。若表示一个视频流，则该字段值为'1'。对于所有其它的流类型，该字段值可以为'0'也可以为'1'。

P-STD\_buffer\_size\_bound：P-STD缓冲区大小界限字段。13位无符号整数，取值不小于节目流中流n的所有分组的P-STD缓冲区大小BSn的最大值。若P-STD\_buffer\_bound\_scale的值为'0'，则该字段以128字节为单位来度量缓冲区大小的边界。若P-STD\_buffer\_bound\_scale的值为'1'，则该字段以1024字节为单位来度量缓冲区大小的边界。因此：

if (P-STD\_buffer\_bound\_scale = = 0)

BSn≤P-STD\_buffer\_size\_bound×128

else

BSn≤P-STD\_buffer\_size\_bound×1024

**实例分析：**

**原始待分析数据：00 00 01 BB 00 09 81 86 A105 E1 7F E0 E8 00**

**1. System\_header\_start\_code系统头（四字节）：**指出系统标题的开始

十六机制：00 00 01 BB

2.   **header\_length标题长度字段（二字节）**：16位字段。指出该字段后的系统标题的字节长度。在本规范将来的扩充中可能扩展该字段。

十六进制：00 09 说明后面还有9字节，大部分系统头则表示有十五字节，其实解析到这里我们就可以直接跳过4+2+9字节继续解析后面的包，因为系统头里面的所有字节含义我们都不关注；

3.   **Rate\_bound 速率界限字段（三字节）：**22位字段，取值不小于编码在节目流的任何包中的program\_mux\_rate字段的最大值。该字段可被解码器用于估计是否有能力对整个流解码。

十六机制：81 86 A1

二进制：1000 0001 1000 0110 1010 0001

**4.   Audio\_bound 音频界限字段（1字节前6bit位）：**6位字段，取值是在从0到32的闭区间中的整数;

十六机制：05

二进制：0000 0101

十进制：1

**5.   Fixed\_flag固定标记字段：**1位标志位。置'1'时表示比特率恒定的操作；置'0'时，表示操作的比特率可变。

十六机制：05

二进制：000 0101

**6.   CSPS\_flag CSPS标志字段:**置'1'时，节目流符合2.7.9中定义的限制。一般就是默认为1；

十六机制：05

二进制：000 0101

**7.   System\_aduio\_lock\_flag系统音频锁定标记字段：**1位字段。表示在系统目标解码器的音频采样率和system\_clock\_frequency之间存在规定的比率。一般默认1；

十六机制：E1

二进制：1110 0001

十进制：1

**8.   System\_video\_lock\_flag系统视频锁定标志字段：**1位字段。表示在系统目标解码器的视频帧速率和system\_clock\_frequency之间存在规定的比率。一般默认1；

十六机制：E1

二进制：1110 0001

十进制：1

**9.   Video\_bound视频界限字段：**5位字段，取值是在从0到16的闭区间中的整数且不小于节目流中解码过程同时活动的GB/T XXXX.2和GB/T AAAA.2流的最大数目。一般也是默认为1；

十六机制：E1

二进制：1110 0001

十进制：1

**10.  Packet\_rate\_restriction\_flag:分组速率限制标志字段：**1位标志位。若CSPS标识为'1'，则该字段表示2.7.9中规定的哪个限制适用于分组速率。若CSPS标识为'0'，则该字段的含义未定义。

十六机制：7F

二进制：0111 1111

十进制：0

**11.   Reserced\_bit保留位字段:**7位字段。被保留供ISO/IEC将来使用。它的值应为'111 1111'，除非ISO/IEC对它作出其它规定。

十六机制：7F

二进制：0111 1111

十进制：0

**12.  Stream\_id流标识字段：**8位字段。指示其后的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound字段所涉及的流的编码和基本流号码。节目流中的每个基本流应在每个系统标题中通过这种机制精确地规定一次它的P-STD\_buffer\_bound\_scale和P-STD\_buffer\_size\_bound。其实就是看到E0就是视频流;

十六机制：E0

二进制：1110 1111

十进制：224

**13.  P-STD\_buffer\_bound\_scale P-STD缓存区界限比例字段：**1位字段。表示用于解释后续P-STD\_buffer\_size\_bound字段的比例系数。若前面的stream\_id表示一个音频流，则该字段值为'0'。若表示一个视频流，则该字段值为'1'。对于所有其它的流类型，该字段值可以为'0'也可以为'1'。

十六机制：**E8**

二进制：1110 1000

十进制：1

**14.  P-STD\_buffer\_size\_bound  P-STD缓冲区大小界限字段：**

十六机制：**E8 00**

二进制：1110 1000 0000 0000

十进制：17592186044416

**我们也用专门的分析工具进行了分析，得到的结论和二进制对起来的各个字段取值是一致的。**

**总结：**

**所以对于系统头部的解析，我们一般只要先首先判断是否存在系统头（根据系统头的起始码0x000001BB），然后我们读取系统头的头部长度,即PS SYSTEM HEADER LENGTH部分，然后根据头部的长度，跳过PS系统头。进入下一个部分，即PS 节目流映射头。**

#### 1.2.4 PS 映射头 PS system Map

PS流的节目映射流部分(节目流映射)定义

|  |  |  |
| --- | --- | --- |
| 语  法 | 位数 | 助记符 |
| program\_stream\_map() { |  |  |
| packet\_start\_code\_prefix | 24 | bslbf |
| map\_stream\_id | 8 | uimsbf |
| program\_stream\_map\_length | 16 | uimsbf |
| current\_next\_indicator | 1 | bslbf |
| reserved | 2 | bslbf |
| program\_stream\_map\_version | 5 | uimsbf |
| reserved | 7 | bslbf |
| marker\_bit | 1 | bslbf |
| program\_stream\_info\_length | 16 | uimsbf |
| for (i=0;i<N;i++){ |  |  |
| descriptor() |  |  |
| } |  |  |
| elementary\_stream\_map\_length | 16 | uimsbf |
| for (i=0;i<N1;i++){ |  |  |
| stream\_type | 8 | uimsbf |
| elementary\_stream\_id | 8 | uimsbf |
| elementary\_stream\_info\_length | 16 | uimsbf |
| for (i=0;i<N2;i++) { |  |  |
| descriptor() |  |  |
| } |  |  |
| } |  |  |
| CRC\_32 | 32 | rpchof |
| } |  |  |

packet\_start\_code\_prefix：分组起始码前缀字段。24位码。它和跟随其后的map\_stream\_id共同组成一个分组起始码以标志分组的开始。该字段是值为'0000 0000 0000 0000 0000 0001' (0x000001)的位串。

map\_stream\_id：映射流标识字段。8位字段，值为0xBC。

program\_stream\_map\_length：节目流映射长度字段。16位字段。指示紧跟在该字段后的program\_stream\_map中的字节数。该字段的最大值为1018(0x3FA)。

current\_next\_indicator：当前下一个指示符字段。1位字段。置'1'时表示传送的节目流映射当前是可用的。置'0'时表示传送的节目流映射还不可用，但它将是下一个生效的表。

program\_stream\_map\_version：节目流映射版本字段。5位字段，表示整个节目流映射的版本号。一旦节目流映射的定义发生变化，该字段将递增1，并对32取模。在current\_next\_indicator为'1'时，该字段应该是当前适用的节目流映射的版本号；在current\_next\_indicator为'0'时，该字段应该是下一个适用的节目流映射的版本号。

program\_stream\_info\_length：节目流信息长度字段。16位字段，指出紧跟在该字段后的描述符的总长度。

marker\_bit：标记位字段。1位字段，取值为'1'。

elementary\_stream\_map\_length：基本流映射长度字段。16位字段，指出在该节目流映射中的所有基本流信息的字节长度。它只包括stream\_type、elementary\_stream\_id和elementary\_stream\_info\_length字段。(这里注意一下，这里的基本流映射长度，他只包括他后面的指定的那几个定义字段的总和，即从从这个长度，我们可以知道后面他根了几种类型的流定义，因为一种流的这个定义字段：stream\_type(1BYTE)、elementary\_stream\_id(1byte)和elementary\_stream\_info\_length(2byte)字段总和为4个字节，所以用elementary\_stream\_map\_length/4可以得到后面定义了几个流类型信息。)

stream\_type：流类型字段 。8位字段，根据表2-29规定了流的类型。该字段只能标志包含在PES分组中的基本流且取值不能为0x05。

(这里我们暂时根据国标GB28181中的定义可以知道

1、MPEG-4 视频流： 0x10；2、H.264 视频流： 0x1B；3、SVAC 视频流： 0x80；4、G.711 音频流： 0x90；5、G.722.1 音频流： 0x92；6、G.723.1 音频流： 0x93；7、G.729 音频流： 0x99；8、SVAC音频流： 0x9B。

因为节目映射流字段只有在关键帧打包的时候，才会存在，所以如果要判断PS打包的流编码类型，就根据这个字段来判断。)

elementary\_stream\_id：基本流标识字段。8位字段，指出该基本流所在PES分组的PES分组标题中stream\_id字段的值。（这个字段的定义，其中0x(C0~DF)指音频，0x(E0~EF)为视频）

elmentary\_stream\_info\_length：基本流信息长度字段。16位字段，指出紧跟在该字段后的描述符的字节长度。(即这个类型的流描述长度。这个后面的字段后面的指定长度不在elementary\_stream\_map\_length指定的范围类。)

CRC\_32：CRC 32字段。32位字段，它包含CRC值以在处理完整个节目流映射后在附录A中定义的解码器寄存器产生0输出值。

对于这个字段的解析，我们需要取值0x000001BC的位串，指出节目流映射的开始，暂时不需要处理，读取Header Length直接跳过即可，如果需要解析流编码类型，必须详细解析这个字段。

**实例分析：**

**待分析数据：00 00 01 BC 00 0E E0 FF 0000 00 04 1B E0 00 00 43 80 9B 58**

**1. Packet\_start\_code\_prefix 分组起始前缀字段：**24位码。它和跟随其后的map\_stream\_id共同组成一个分组起始码以标志分组的开始。该字段是值为'00000000 0000 0000 0000 0001' (0x000001)的位串。

十六机制：00 00 01

**2. Map\_stream\_id 映射流标识字段：**8位字段，值为0xBC

十六机制：BC

**3. Program\_stream\_map\_length 节目流映射长度字段：**16位字段。指示紧跟在该字段后的program\_stream\_map中的字节数。该字段的最大值为1018(0x3FA)

十六机制：00 0E

十进制：15 表示后面还有14字节，除了自己。

**4. Current\_next\_indicator 当前下一个指示符字段：**1位字段。置'1'时表示传送的节目流映射当前是可用的。置'0'时表示传送的节目流映射还不可用，但它将是下一个生效的表。一般默认值为1；

十六机制：E0

二进制：1110 1111

十进制：1

**5. Program\_stream\_map\_version 节目流映射版本字段：**5位字段，表示整个节目流映射的版本号。一旦节目流映射的定义发生变化，该字段将递增1，并对32取模。在current\_next\_indicator为'1'时，该字段应该是当前适用的节目流映射的版本号；在current\_next\_indicator为'0'时，该字段应该是下一个适用的节目流映射的版本号。

十六机制：E0

二进制：1110 1111

十进制：16

**6. Marker\_bit标记位字段：**1位字段，取值为'1'。

十六机制：FF

二进制：1111 1111

十进制：1

**7.** **Program\_stream\_info\_length节目流信息长度字段：**16位字段，指出紧跟在该字段后的描述符的总长度。

十六机制：00 00

二进制：0000 0000 0000 0000

十进制：0

**8. Elementary\_stream\_map\_length基本映射流长度字段：**16位字段，指出在该节目流映射中的所有基本流信息的字节长度。它只包括stream\_type、elementary\_stream\_id和elementary\_stream\_info\_length字段。(这里注意一下，这里的基本流映射长度，他只包括他后面的指定的那几个定义字段的总和，即从从这个长度，我们可以知道后面他根了几种类型的流定义，因为一种流的这个定义字段：stream\_type(1BYTE)、elementary\_stream\_id(1byte)和elementary\_stream\_info\_length(2byte)字段总和为4个字节，所以用elementary\_stream\_map\_length/4可以得到后面定义了几个流类型信息。)

十六机制：00 04

二进制：0000 0000 0000 0100

十进制：4

**这个值除以4则表示9、10、11这三个字段的循环次数。这里为4，说明后面只有一组，只需要循环一次。**

**9. Stream\_type流类型字段：**8位字段，根据表2-29规定了流的类型。该字段只能标志包含在PES分组中的基本流且取值不能为0x05。这里我们看到的是H264编码的视频和0x90的G711的音频。

(这里我们暂时根据国标GB28181中的定义可以知道

1、MPEG-4 视频流： 0x10；

2、H.264 视频流： 0x1B；

3、SVAC 视频流： 0x80；

4、G.711 音频流： 0x90；

5、G.722.1 音频流： 0x92；

6、G.723.1 音频流： 0x93；

7、G.729 音频流： 0x99；

8、SVAC音频流： 0x9B

十六机制：1B

二进制：0001 1011

**10. 基本流标识字段  elementary\_stream\_id：**8位字段，指出该基本流所在PES分组的PES分组标题中stream\_id字段的值。（这个字段的定义，其中0x(C0~DF)指音频，0x(E0~EF)为视频）

十六机制：E0

二进制：1110 1111

十进制：224

**11. Elementary\_stream\_info\_length基本流信息长度字段：**16位字段，指出紧跟在该字段后的描述符的字节长度。(即这个类型的流描述长度。这个后面的字段后面的指定长度不在elementary\_stream\_map\_length指定的范围类。)

十六机制：00 00

二进制：0000 0000 0000 0000

**12. CRC32校验字段：**

32位字段，它包含CRC值以在处理完整个节目流映射后在附录A中定义的解码器寄存器产生0输出值。

十六机制：**43 80 9B 58**

**总结**

对于这个字段的解析，我们需要取值0x000001BC的位串，指出节目流映射的开始，暂时不需要处理，读取HeaderLength直接跳过即可，如果需要解析流编码类型，必须详细解析这个字段。

实际上我们需要在解析节目映射流头字段时，需要解析基本映射流长度字段,这里面包含这是否含有音视频以及音视频对应的编码格式。

#### 1.2.5 PES包 PES\_packet

PES包=PES header+code raw data。



这个是ps对pes包得包装。Ps自带标志位ba bb后面紧跟pes 包，pes包可以是eo bc等stream id标识。





|  |  |  |
| --- | --- | --- |
| 语  法 | 位数 | 助记符 |
| PES\_packet(){ |  |  |
| packet\_start\_code\_prefix | 24 | bslbf |
| stream\_id | 8 | uimsbf |
| PES\_packet\_length | 16 | uimsbf |
| if(stream\_id != program\_stream\_map |  |  |
| && stream\_id !=padding\_stream |  |  |
| && stream\_id !=private\_stream\_2 |  |  |
| && stream\_id !=ECM |  |  |
| && stream\_id !=EMM |  |  |
| && stream\_id !=program\_stream\_directory |  |  |
| && stream\_id !=DSMCC\_stream |  |  |
| && stream\_id !=ITU-T Rec.H.222.1 type E stream){ |  |  |
| '10' | 2 | bslbf |
| PES\_scrambling\_control | 2 | bslbf |
| PES\_priority | 1 | bslbf |
| data\_alignment\_indicator | 1 | bslbf |
| copyright | 1 | bslbf |
| original\_or\_copy | 1 | bslbf |
| PTS\_DTS\_flags | 2 | bslbf |
| ESCR\_flag | 1 | bslbf |
| ES\_rate\_flag | 1 | bslbf |
| DSM\_trick\_mode\_flag | 1 | bslbf |
| additional\_copy\_info\_flag | 1 | bslbf |
| PES\_CRC\_flag | 1 | bslbf |
| PES\_extension\_flag | 1 | bslbf |
| PES\_header\_data\_length | 8 | uimsbf |
| if(PTS\_DTS\_flags =='10'){ |  |  |
| '0010' | 4 | bslbf |
| PTS[32..30] | 3 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[29..15] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[14..0] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| } |  |  |
| if(PTS\_DTS\_flags =='11'){ |  |  |
| '0011' | 4 | bslbf |
| PTS[32..30] | 3 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[29..15] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[14..0] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| '0001' | 4 | bslbf |
| PTS[32..30] | 3 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[29..15] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| PTS[14..0] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| } |  |  |
| if(ESCR\_flag =='1'){ |  |  |
| reserved | 2 | bslbf |
| ESCR\_base[32..30] | 3 | bslbf |
| marker\_bit | 1 | bslbf |
| ESCR\_base[29..15] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| ESCR\_base[14..0] | 15 | bslbf |
| marker\_bit | 1 | bslbf |
| ESCR\_extension | 9 | uimsbf |
| marker\_bit | 1 | bslbf |
| } |  |  |
| if(ES\_rate\_flag =='1'){ |  |  |
| marker\_bit | 1 | bslbf |
| ES\_rate | 22 | uimsbf |
| marker\_bit | 1 | bslbf |
| } |  |  |
| if (DSM\_trick\_mode\_flag =='1'){ |  |  |
| trick\_mode\_control | 3 | uimsbf |
| if ( trick\_mode\_control = =fast\_forward ) { |  |  |
| field\_id | 2 | bslbf |
| intra\_slice\_refresh | 1 | bslbf |
| frequency\_truncation | 2 | bslbf |
| } |  |  |
| else if ( trick\_mode\_control = = slow\_motion ) { |  |  |
| rep\_cntrl | 5 | uimsbf |
| } |  |  |
| else if ( trick\_mode \_control = = freeze\_frame ) { |  |  |
| field\_id | 2 | uimsbf |
| reserved | 3 | bslbf |
| } |  |  |
| else if ( trick\_mode \_control = = fast\_reverse ) { |  |  |
| field\_id | 2 | bslbf |
| intra\_slice\_refresh | 1 | bslbf |
| frequency\_truncation | 2 | bslbf |
| else if ( trick\_mode\_control = = slow\_reverse ) { |  |  |
| rep\_cntrl | 5 | uimsbf |
| } |  |  |
| else |  |  |
| reserved | 5 | bslbf |
| } |  |  |
| if ( additional\_copy\_info\_flag = ='1'){ |  |  |
| marker\_bit | 1 | bslbf |
| additional\_copy\_info | 7 | bslbf |
| } |  |  |
| if (PES\_CRC\_flag==‘1’){ |  |  |
| previous\_PES\_packet\_CRC | 16 | bslbf |
| } |  |  |
| if ( PES\_extension\_flag =='1') { |  |  |
| PES\_private\_data\_flag | 1 | bslbf |
| pack\_header\_field\_flag | 1 | bslbf |
| program\_packet\_sequence\_counter\_flag | 1 | bslbf |
| P-STD\_buffer\_flag | 1 | bslbf |
| reserved | 3 | bslbf |
| PES\_extension\_flag\_2 | 1 | bslbf |
| if(PES\_private\_data\_flag =='1'){ |  |  |
| PES\_private\_data | 128 | bslbf |
| } |  |  |
| if (pack\_header\_field\_flag == '1'){ |  |  |
| pack\_field\_length | 8 | uimsbf |
| pack\_header() |  |  |
| } |  |  |
| if (program\_packer\_sequence\_counter\_flag == '1'){ |  |  |
| marker\_bit | 1 | bslbf |
| program\_packet\_sequence\_counter | 7 | uimsbf |
| marker-bit | 1 | bslbf |
| MPEG1\_MPEG2\_indentifier | 1 | bslbf |
| original\_stuff\_length | 6 | uimsbf |
| } |  |  |
| if (P-STD\_buffer\_flag = = '1'({ |  |  |
| '01' | 2 | bslbf |
| P-STD\_buffer\_scale | 1 | bslbf |
| P-STD\_buffer\_size | 13 | uimsbf |
| } |  |  |
| if (PES\_extension\_flag\_2 == '1'{ |  |  |
| marker\_bit | 1 | bslbf |
| PES\_extension\_field\_length | 7 | uimsbf |
| for(i=0;i<PES\_extension\_field\_length;i++){ |  |  |
| reserved | 8 | bslbf |
| } |  |  |
| } |  |  |
| } |  |  |
| for (i=0;i<N1;i++){ |  |  |
| stuffing\_byte | 8 | bslbf |
| } |  |  |
| for (i=0;i<N2;i++){ |  |  |
| PES\_packet\_data\_byte | 8 | bslbf |
| } |  |  |
| } |  |  |
| else if (stream\_id = = program\_stream\_map |  |  |
| || stream\_id = = private\_stream\_2 |  |  |
| || stream\_id = = ECM |  |  |
| || stream\_id = = EMM |  |  |
| || stream\_id = = program\_stream\_directory |  |  |
| || stream\_id = = DSMCC\_stream |  |  |
| || stream\_id = = ITU-T Rec. H.222.1 type E stream ){ |  |  |
| for (i=0;i<PES\_packet\_length;i++){ |  |  |
| PES\_packet\_data\_byte | 8 | bslbf |
| } |  |  |
| } |  |  |
| else if (steam\_id = = padding\_stream){ |  |  |
| for (i=0;i<PES\_packet\_length;i++){ |  |  |
| padding\_byte | 8 | bslbf |
| } |  |  |
| } |  |  |
| } |  |  |

以上都是根据steam\_id进行判断处理

表2-18 Stream\_id赋值

|  |  |  |
| --- | --- | --- |
| stream\_id | 注 | 流 编 码 |
| 1011 1100 | 1 | program\_stream\_map（0xBC） |
| 1011 1101 | 2 | private\_stream\_1(0xBD) |
| 1011 1110 |  | padding\_stream(0xBE) |
| 1011 1111 | 3 | private\_stream-2(0xBF) |
| 110x xxxx |  | GB/T XXXX.3或GB/T AAAA.3音频流编号xxxx(0xC0~0xDF) |
| 1110 xxxx |  | GB/T XXXX.2或GB/T AAAA.2视频流编号xxxx(0xE0~0xEF) |
| 1111 0000 | 3 | ECM\_stream(0xF0) |
| 1111 0001 | 3 | EMM\_stream(0xF1) |
| 1111 0010 | 5 | GB/T XXXX.1附录B或GB/T XXXX.6\_DSMCC\_stream(0xF2) |
| 1111 0011 | 2 | ISO/IEC\_13522\_stream(0xF3) |
| 1111 0100 | 6 | ITU-T Rec. H.222.1类型A |
| 1111 0101 | 6 | ITU-T Rec. H.222.1类型B |
| 1111 0110 | 6 | ITU-T Rec. H.222.1类型C |
| 1111 0111 | 6 | ITU-T Rec. H.222.1类型D |
| 1111 1000 | 6 | ITU-T Rec. H.222.1类型E |
| 1111 1001 | 7 | ancillary\_stream(0xF9) |
| 1111 1010…1111 1110 |  | 保留数据流 |
| 1111 1111 | 4 | program\_stream\_directory(0xFF) |

符号x表示值'0'或'1'均被允许且可产生相同的流类型。流号码由x的取值决定。

注

1 类型为program\_stream\_map的PES分组有唯一的语法，在2.5.4.1中作了规定。

2 类型为private\_stream\_1和ISO/IEC\_13352\_stream的PES分组与GB/T XXXX.2及GB/T XXXX.3音频流服从相同的PES分组语法。

3 类型为private\_stream\_2，ECM\_stream和EMM\_stream的PES分组与private\_stream\_1相似，除了在PES\_packet\_length字段后未规定语法。

4 类型为program\_stream\_directory的PES分组有唯一的语法，在2.5.5中作了规定。

5 类型为DSM\_CC\_stream的PES分组有唯一的语法，在GB/T XXXX.6中作了规定。

6 stream\_id与表2-29中的stream\_type 0x09相关联。

7 stream\_id仅用于PES分组。PES分组在传输流中携带了来源于节目流或GB/T AAAA.1系统流的数据(参见2.4.3.7)。

0x0000 01B9结束，即MPEG\_Program\_end\_code

PES分组中各字段的语义定义：

packet\_start\_code\_prefix：分组起始码前缀字段。24位代码，它和后面的stream\_id构成了标识分组开始的分组起始码。它是一个值为'0000 0000 0000 0000 0000 0001' (0x000001)的位串。

stream\_id：流标识字段。在节目流中，它规定了基本流的号码和类型。定义参见表2-18。在传输流中，它可以被设定为正确描述表2-18中定义的基本流类型的任何有效值。在传输流中，基本流类型在2.4.4的节目特定信息中作了规定。

PES\_packet\_length：PES分组长度字段。16位字段，指出了PES分组中跟在该字段后的字节数目。值为0表示PES分组长度要么没有规定要么没有限制。这种情况只允许出现在有效负载包含来源于传输流分组中某个视频基本流的字节的PES分组中。

PES\_scrambling\_control：PES加扰控制字段。2位字段，表示PES分组有效负载的加扰方式。当加扰发生在PES层，PES分组标题，如果有可选字段的话也包括在内，不应被加扰(参见2-19)。

PES\_priority：PES优先级字段。1位字段，指示PES分组中有效负载的优先级。'1'表示PES分组中有效负载的优先级高于该字段为'0'的PES分组有效负载。多路复合器能使用该字段来区分安排基本流中数据的优先级。传输机制不应改动该字段。

data\_alignment\_indicator：数据对齐指示符字段。1位标志。置'1'时表示PES分组标题后紧跟着在2.6.10中的data\_alignment\_indicator所指出的视频起始码或音频同步字，如果有data\_alignment\_indicator描述符的话。若其值为'1'且无该描述符，则需要在表2-47和2-48中alignment\_type '01'所表示的对齐。当值为'0'时，没有定义是否有任何此种的对齐。

Copyright：版权字段。1位字段。置'1'时表示相关PES分组有效负载的材料受到版权保护。当值为'0'时，没有定义该材料是否受到版权保护。2.6.24中描述的版权描述符与包含PES分组的基本流相关。若描述符作用于包含PES分组的材料，则版权标志被置为'1'。

original\_or\_copy：原始或拷贝字段。1位字段。置'1'时表示相关PES分组有效负载的内容是原始的；值为'0'表示相关PES分组有效负载的内容是一份拷贝。

PTS\_DTS\_flags：PTS DTS标志字段。2位字段。当值为'10'时，PTS字段应出现在PES分组标题中；当值为'11'时，PTS字段和DTS字段都应出现在PES分组标题中；当值为'00'时，PTS字段和DTS字段都不出现在PES分组标题中。值'01'是不允许的。

ESCR\_flag：ESCR标志字段。1位标志。置'1'时表示ESCR基础和扩展字段出现在PES分组标题中；值为'0'表示没有ESCR字段。

ES\_rate\_flag：ES速率标志字段。1位标志。置'1'时表示ES\_rate字段出现在PES分组标题中；值为'0'表示没有ES\_rate字段。

DSM\_trick\_mode\_flag：DSM特技方式标志字段。1位标志。置'1'时表示有8位特技方式字段；值为'0'表示没有该字段。

additional\_copy\_info\_flag：附加版权信息标志字段。1位标志。置'1'时表示有附加拷贝信息字段；值为'0'表示没有该字段。

PES\_CRC\_flag：PES CRC标志字段。1位标志。置'1'时表示CRC字段出现在PES分组标题中；值为'0'表示没有该字段。

PES\_extension\_flag ：PES扩展标志字段。1位标志。置'1'时表示PES分组标题中有扩展字段；值为'0'表示没有该字段。

PES\_header\_data\_length：PES标题数据长度字段。8位字段。指出包含在PES分组标题中的可选字段和任何填充字节所占用的总字节数。该字段之前的字节指出了有无可选字段。

marker\_bit：标记位字段。值为'1'的1位字段。

PTS：展现时间戳字段。展现时间与解码时间的关系如下：PTS是一个编码在三个分离字段中的33位数字。它指出了基本流n的第k个展现单元在系统目标解码器中的展现时间tpn(k)。PTS的值以系统时钟频率的1/300(即90 kHz)为单位。展现时间由PTS根据式2-11计算而来。对编码展现时间戳频率的约束参见2.7.4。

PTS(k)＝((system\_clock\_frequency×tpn(k)) DIV 300) % 233 (2-11)

其中，tpn(k)是展现单元Pn(k)的展现时间。对音频而言，若PES分组标题中有PTS，则它是指PES分组中开始的第一个存取单元。若PES分组中有音频存取单元的首字节，则有一个音频存取单元开始于该PES分组中。对视频而言，若PES分组标题中有PTS，则它是指包含PES分组中开始的第一个画面起始码的存取单元。若PES分组中有画面起始码的首字节，则有一个画面起始码开始于该PES分组中。对音频展现单元(PU)，low\_delay序列中的视频PU以及B画面，展现时间tpn(k)应等于tdn(k)。对于非low\_delay中的I画面和P画面，在存取单元(AU) k和k'之间无解码不连续时，展现时间tpn(k)应等于下一个传输的I画面或P画面的解码时间tdn(k) (参见2.7.5)。若有解码不连续或流终止，则tpn(k)和tdn(k)之间的差别应与初始流一直延续，没有不连续也没有终止时完全相同。

注1： low\_delay序列是low\_delay标志被设置的视频序列(参见GB/T XXXX.2中的6.2.2.3)。

若音频中有滤波，则系统模型假定滤波不会导致延迟。因此，编码时PTS所涉及的采样与解码时PTS所涉及的采样是相同的。对于可伸缩编码，参见2.7.6。

DTS：解码时间戳字段。DTS是一个编码在三个分离字段中的33位数字。它指出了基本流n的第j个展现单元在系统目标解码器中的解码时间tdn(j)。DTS的值以系统时钟频率的1/300 (即90 kHz)为单位。解码时间由DTS根据式2-12计算而来：

DTS(j)＝((system\_clock\_frequency×tdn(j)) DIV 300) % 233 (2-12)

其中，tdn(j)是存取单元An(j)的解码时间。

对视频而言，若PES分组标题中有DTS，则它是指包含PES分组中开始的第一个画面起始码的存取单元。若PES分组中有画面起始码的首字节，则该画面起始码开始于该PES分组中。

对于可伸缩编码，参见2.7.6。

ESCR\_base，ESCR\_extension：ESCR字段。42位字段，分两部分编码。第一部分是一个长度为33位的字段，其值ESCR\_base(i)由式2-14给出；第二部分是一个长度为9位的字段，其值ESCR\_ext(i)由式2-15给出。ESCR字段指出了基本流中包含ESCR\_base最后一个比特的字节到达PES-STD输出端的期望时间(参见2.5.2.4)。

特别地

ESCR(i)＝ESCR\_base(i)×300＋ESCR\_ext(i) (2-13)

其中：

ESCR\_base(i)＝((system\_clock\_frequency×t(i)) DIV 300) % 233 (2-14)

ESCR \_ext(i)＝((system\_clock\_frequency×t(i)) DIV 1) % 300 (2-15)

ESCR和ES\_rate字段(参见下面紧接的语义)包含与PES流序列相关的时间信息。这些字段应满足2.7.3中定义的约束。

ES\_rate：基本流速率字段 。22位无符号整数。对于PES流而言，它指出了系统目标解码器接收PES分组的速率。该字段在它所属的PES分组以及同一个PES流的后续PES分组中一直有效，直到遇到一个新的ES\_rate字段。该字段的值以50字节/秒为单位，且不能为0。该字段用于定义PES流的字节到达P-STD输入端的时间(参见2.5.2.4中的定义)。在各个PES分组中，编码在该字段中的值可能不同。

trick\_mode\_control：特技方式控制字段。3位字段。它表示作用于相关视频流的特技方式。对其它类型的基本流，该字段及其后5位的含义没有定义。trick\_mode状态的定义参见2.4.2.3的特技模式部分。当trick\_mode状态为假时，对GB/T XXXX.2视频而言，解码过程输出渐进序列中每幅画面的次数N由repeat\_first\_field和top\_field\_first字段来规定。对GB/T AAAA.2视频而言，由序列标题决定。对于隔行序列，当trick\_mode状态为假时，对GB/T XXXX.2视频而言，次数N由repeat\_first\_field和progressive\_frame字段来规定。

当trick\_mode状态为真时，画面的播放次数依赖于值N。

表2-20 特技方式控制值

|  |  |
| --- | --- |
| 值 | 描  述 |
| '000' | 快进 |
| '001' | 慢动作 |
| '010' | 冻结帧 |
| '011' | 快倒 |
| '100' | 慢倒 |
| '101'－'111' | 保留 |

当该字段值发生变化或特技模式操作停止时，可能会出现下列情况的任意组合：

l 时基不连续；

l 解码不连续；

l 连续性计数器不连续；

在特技模式的情况下，解码和展现的非标准速度可能会导致视频基本流数据中定义的某些字段值不正确。同样，片段结构的语义约束也可能无效。这些例外所涉及的视频语法元素为：

l bit\_rate;

l vbv\_delay;

l repeat\_first\_field;

l v\_axis\_positive;

l field\_sequence;

l subcarrier;

l burst\_amplitude;

l subcarrier\_phase;

在特技模式中，解码器不应该依赖于编码在这些字段中的值。

标准并不要求解码器能解码trick\_mode\_control字段。但是，能解码该字段的解码器应能满足以下标准要求。

fast forword：快进。trick\_mode\_control字段中的值'000'。当该值出现时，它表示一个快进视频流并定义了PES分组标题中后续5位的含义。intra\_slice\_refresh位可以被设定为'1'以指出可能有丢失的宏块。解码器可以用前一个解码画面中相同位置的宏块来代替。表2-21中定义的field\_id字段，表示应该显示哪个或哪些字段。frequency\_truncation字段指出了可能包括的一个系数受限集合。该字段值的含义如表2-22所示。

slow motion：慢动作。trick\_mode\_control字段中的值'001'。当该值出现时，它表示一个慢动作视频流，并定义了PES分组标题中后续5个比特的含义。对渐进序列而言，该画面应被显示N×rep\_cntrl时间，其中N定义如上。

对GB/T AAAA.2视频和GB/T XXXX.2视频渐进序列而言，该画面应被显示N×rep\_cntrl时间，其中N定义如上。

对GB/T XXXX.2隔行序列而言，该画面应被显示N×rep\_cntrl时间。若该画面是一个帧画面，则待显示的第一个字段在top\_field\_first为1时应该是顶字段，在top\_field\_first为0时，应该是底字段(参见GB/T XXXX.2)。该字段被显示N×rep\_cntrl / 2时间。该画面的其它字段被显示N－N×rep\_cntrl / 2时间。

freeze frame：冻结帧。trick\_mode\_control字段中的值'010'。当该值出现时，它表示冻结帧视频流，并定义了PES分组标题中后续5位的含义。表2-21中定义的field\_id字段，表示应该显示哪个(些)字段。field\_id字段指出了包含该字段的PES分组中开始的第一个视频存取单元，除非该PES分组包含0个有效负载字节。在后一种情况下，field\_id字段指出了最近的前一个视频存取单元。

fast reverse：快倒。trick\_mode\_control字段中的值'011'。当该值出现时，它表示一个快倒视频流并定义了PES分组标题中后续5位的含义。intra\_slice\_refresh位可以被设定为'1'以指出可能有丢失的宏块。解码器可以用前一个解码画面中相同位置的宏块来代替。表2-21中定义的field\_id字段，表示应该显示哪个或哪些字段。frequency\_truncation字段指出了可能包括的一个系数受限集合。该字段值的含义如表2-22“系数选择值”所示。

slow reverse：慢倒。trick\_mode\_control字段中的值'100'。当该值出现时，它表示一个慢倒视频流并定义了PES分组标题中后续5位的含义。对GB/T AAAA.2视频和GB/T XXXX.2视频渐进序列而言，该画面应被显示N×rep\_cntrl时间，其中N定义如上。

对GB/T XXXX.2隔行序列而言，该画面应被显示N×rep\_cntrl时间。若该画面是一个帧画面，则待显示的第一个字段在top\_field\_first为1时应该是底字段，在top\_field\_first为'0'时，应该是顶字段(参见GB/T XXXX.2)。该字段被显示N×rep\_cntrl/2时间。该画面的其它字段被显示N－N×rep\_cntrl / 2时间。

field\_id：字段标识字段。2位字段，表示应该显示哪个(些)字段。根据表2-21对其进行编码。

表2-21 field\_id字段控制值

|  |  |
| --- | --- |
| 值 | 描  述 |
| '00' | 仅自顶向下播放 |
| '01' | 仅自底向上播放 |
| '10' | 播放所有帧 |
| '11' | 保留 |

intra\_slice\_refresh：片内参考字段。1位标志。置'1'时表示PES分组的视频数据编码片中可能有丢失的宏块；置'0'时，表示上述情况可能不出现。更多的信息可参见GB/T XXXX.2。解码器可以用前一个解码画面中同一个位置的宏块来代替丢失的宏块。

frequency\_truncation：频率截断字段。2位字段。指出在对PES分组中数据进行编码时可能用到受限系数集合。其值定义于表2-22。

表2-22 系数选择值

|  |  |
| --- | --- |
| 值 | 描述 |
| '00' | 仅DC系数非0 |
| '01' | 仅前三个系数非0 |
| '10' | 仅前六个系数非0 |
| '11' | 所有系数均可能非0 |

rep\_cntrl：显示次数控制字段。5位字段，指出隔行画面中每一字段的显示次数或渐进画面显示次数。对隔行画面而言，顶字段或底字段是否应首先显示是视频序列标题中trick\_mode\_control字段和top\_field\_first字段的功能。该字段值不能为'0'。

additional\_copy\_info：附加版权信息字段。7位字段，包含与版权信息有关的专用数据。

previous\_PES\_packet\_CRC：前PES分组CRC字段。16位字段。在对前一个PES分组(不包括该PES分组的标题)进行处理后，该字段包含一个在解码器的16个寄存器中生成0输出的CRC值。该CRC值与附录A中所定义的相类似，但具有以下多项式：

x16＋x12＋x5＋1

注2： 该CRC值是为了用于网络维护，例如将有间隙性错误的源隔离开来，而不是为了供基本流解码器使用。它仅用于计算数据字节，因为在传输过程中PES分组标题数据可能被修改。

PES\_private\_data\_flag：PES专用数据标志字段。1位标志。置'1'时表示PES分组标题中包含专用数据；置'0'时表示PES分组标题中无专用数据。

pack\_header\_field\_flag：包标题字段标志字段。1位标志。置'1'时表示PES分组标题中有GB/T AAAA.2包标题或节目流包标题。若该字段在包含于节目流中的PES分组中，其值应为'0'。在传输流中，当值为'0'时表示PES标题中无包标题。

program\_packet\_sequence\_counter\_flag：节目分组序列计数标志字段。1位标志。值为'1'时表示PES分组有program\_packet\_sequence\_counter，MPEG1\_MPEG2\_identifier和original\_stuff\_length字段。值为'0'时表示PES分组标题中无这些字段。

P-STD\_buffer\_flag：P-STD缓冲区标志字段。1位标志。置'1'时表示PES分组标题中有P-STD\_buffer\_scale和P-STD\_buffer\_size字段。值为'0'时表示PES标题中无这些字段。

PES\_extension\_flag\_2：PES扩展标志字。1位标志，置'1'时表示有PES\_extension\_field\_length及相关字段。

PES\_private\_data：PES专用数据字段。16位字段。包含专用数据。这些数据与其前后的字段组合在一起时，不能与packet\_start\_code\_prefix (0x000001)冲突。

pack\_field\_length：包字段长度字段。8位字段。表示pack\_header\_field()以字节为单位时的长度。

program\_packet\_sequence\_counter：节目分组序列计数字段。7位字段。它是一个可选的计数器，随着来自于节目流或GB/T AAAA.1流的每一个后续的PES分组或传输流中具有单个节目定义的PES分组而递增，以提供与连续性计数器(参见2.4.3.2)相似的功能。它能用于检索节目流或原始GB/T AAAA.1流中的初始PES分组序列。该计数器在达到最大值后回卷为0。PES分组不能出现重复。因此，复合节目中任何两个连续的PES分组不应具有相同的program\_packet\_sequence\_counter值。

MPEG1\_MPEG2\_identifier：MPEG1 MPEG2标识符字段。1位标志。置'1'时表示PES分组携带的信息来自于GB/T AAAA.1流；置'0'时表示PES分组携带的信息来自于节目流。

original\_stuff\_length：初始填充长度字段。6位字段。指定用于初始GB/T XXXX.1分组标题或初始GB/T AAAA.1分组标题中的填充字节数。

P-STD\_ buffer\_scale：P-STD缓冲区比例字段。1位字段。仅当该PES分组包含于节目流中时才有意义。它指出了用来解释后续P-STD\_buffer\_size字段的比例因子。若前面的stream\_id表示一个音频流，该字段值应为'0'；若前面的stream\_id表示一个视频流，该字段值应为'1'。对于所有的其它流类型，其值可以为'0'或'1'。

P-STD\_buffer\_size：P-STD缓冲区大小字段 。13位无符号整数。仅当该PES分组包含于节目流中时才有意义。它定义了P-STD输入缓冲区的大小BSn。若P-STD\_ buffer\_scale的值为'0'，那么P-STD\_buffer\_size以128字节为单位来度量缓冲区的大小。若P-STD\_buffer\_scale的值为'1'，那么P-STD\_buffer\_size以1024字节为单位来度量缓冲区的大小。因此：

if ( P-STD\_buffer\_scale == 0)

BSn＝P-STD\_buffer\_size×128

else

BSn＝P-STD\_buffer\_size×1024

当该字段被GB/T XXXX.1系统目标解码器收到后，其编码值立即生效。

PES\_extension\_field\_length：PES扩展字段长度字段。7位字段。指出了跟在该字段之后在PES扩展字段中直到且包括任何保留字节的数据的字节长度。

stuffing\_byte ：填充字节字段。8位字段，其值恒定为'1111 1111'。可以由编码器插入以满足通道的需求等。解码器丢弃该字段。一个PES分组标题中只能出现32个填充字节。

PES\_packet\_data\_byte ：PES分组数据字节字段。该字段应该是来自于由分组的stream\_id或PID所指定的基本流的连续数据字节。当基本流数据符合GB/T XXXX.2或GB/T XXXX.3时，该字段应该是与本标准的字节相对齐的字节。基本流的字节序应得到保持。该字段的字节数N由PES\_packet\_length字段规定。N应等于PES\_packet\_length减去在PES\_packet\_length字段的最后一个字节与第一个 PES\_packet\_data\_byte间的字节数。

padding\_byte：填料字节字段 ，8位字段，其值恒定为'1111 1111'。该字段被解码器丢弃。

**实例分析：**

**分析数据：00 00 01 E0 B8 D7 80 C0 0A 31 00 01 00 01 11 00 01 00 01**

**1. Packet\_start\_code\_prefix 分组起始前缀字段：**24位代码，它和后面的stream\_id构成了标识分组开始的分组起始码。它是一个值为'0000 0000 0000 0000 0000 0001'(0x000001)的位串。

十六机制：**00 00 01**

**2. Stream\_id流标识字段:**在节目流中，它规定了基本流的号码和类型。

|  |  |  |
| --- | --- | --- |
| stream\_id | 注 | 流编码 |
| 1011 1100 | 1 | program\_stream\_map（0xBC） |
| 1011 1101 | 2 | private\_stream\_1(0xBD) |
| 1011 1110 |  | padding\_stream(0xBE) |
| 1011 1111 | 3 | private\_stream-2(0xBF) |
| 110x xxxx |  | GB/T XXXX.3或GB/T AAAA.3音频流编号xxxx(0xC0~0xDF) |
| 1110 xxxx |  | GB/T XXXX.2或GB/T AAAA.2视频流编号xxxx(0xE0~0xEF) |
| 1111 0000 | 3 | ECM\_stream(0xF0) |
| 1111 0001 | 3 | EMM\_stream(0xF1) |
| 1111 0010 | 5 | GB/T XXXX.1附录B或GB/T XXXX.6\_DSMCC\_stream(0xF2) |
| 1111 0011 | 2 | ISO/IEC\_13522\_stream(0xF3) |
| 1111 0100 | 6 | ITU-T Rec. H.222.1类型A |
| 1111 0101 | 6 | ITU-T Rec. H.222.1类型B |
| 1111 0110 | 6 | ITU-T Rec. H.222.1类型C |
| 1111 0111 | 6 | ITU-T Rec. H.222.1类型D |
| 1111 1000 | 6 | ITU-T Rec. H.222.1类型E |
| 1111 1001 | 7 | ancillary\_stream(0xF9) |
| 1111 1010…1111  1110 |  | 保留数据流 |
| 1111 1111 | 4 | program\_stream\_directory(0xFF) |
| 符号x表示值'0'或'1'均被允许且可产生相同的流类型。流号码由x的取值决定。  注  1  类型为program\_stream\_map的PES分组有唯一的语法，在2.5.4.1中作了规定。  2  类型为private\_stream\_1和ISO/IEC\_13352\_stream的PES分组与GB/T XXXX.2及GB/T XXXX.3音频流服从相同的PES分组语法。  3  类型为private\_stream\_2，ECM\_stream和EMM\_stream的PES分组与private\_stream\_1相似，除了在PES\_packet\_length字段后未规定语法。  4  类型为program\_stream\_directory的PES分组有唯一的语法，在2.5.5中作了规定。  5  类型为DSM\_CC\_stream的PES分组有唯一的语法，在GB/T XXXX.6中作了规定。  6  stream\_id与表2-29中的stream\_type  0x09相关联。  7  stream\_id仅用于PES分组。PES分组在传输流中携带了来源于节目流或GB/T AAAA.1系统流的数据(参见2.4.3.7)。 | | |

能看到的就是BC，节目映射流标记。BD一些国标厂家会填充一些私有流。E0代表视频流,C0代表音频流，其它的基本不需要关注。

十六机制：**E0**

二进制：1110 0000

**00 00 01 E0 B8 D7 80 C0 0A 31 00 01 00 01 11 00 01 00 01**

3.**PES\_packet\_lengthPES分组数据长度字段：**16位字段，指出了PES分组中跟在该字段后的字节数目。值为0表示PES分组长度要么没有规定要么没有限制。这种情况只允许出现在有效负载包含来源于传输流分组中某个视频基本流的字节的PES分组中。

十六机制：**B8 D7**

二进制：1110 1111

十进制：47319

说明该PES的长度是47319，意思从这个字段之后再数这么多字节将是下一个PS包。

**00 00 01 E0 B8 D7 80 C0 0A 31 00 01 00 01 11 00 01 00 01**

4.**PES\_scrambling\_controlPES加扰控制字段：**2位字段，表示PES分组有效负载的加扰方式。当加扰发生在PES层，PES分组标题，如果有可选字段的话也包括在内，不应被加扰(参见2-19)。

十六机制：**80**

二进制：1000 0000

这两个bit位一般是默认为10的，如果是其它stream\_id则无所谓填什么。

5.**PES\_priority优先级字段：**1位字段，指示PES分组中有效负载的优先级。'1'表示PES分组中有效负载的优先级高于该字段为'0'的PES分组有效负载。多路复合器能使用该字段来区分安排基本流中数据的优先级。传输机制不应改动该字段。

十六机制：**80**

二进制：1000 0000

十进制：0

6.**Data\_alignment\_indicator数据对齐指示符字段：**1位标志。置'1'时表示PES分组标题后紧跟着在2.6.10中的data\_alignment\_indicator所指出的视频起始码或音频同步字，如果有data\_alignment\_indicator描述符的话。若其值为'1'且无该描述符，则需要在表2-47和2-48中alignment\_type '01'所表示的对齐。当值为'0'时，没有定义是否有任何此种的对齐。

十六机制：**80**

二进制：1000 0000

十进制：0

7.**Copyright版权字段：**1位字段。置'1'时表示相关PES分组有效负载的材料受到版权保护。当值为'0'时，没有定义该材料是否受到版权保护。2.6.24中描述的版权描述符与包含PES分组的基本流相关。若描述符作用于包含PES分组的材料，则版权标志被置为'1'。

十六机制：**80**

二进制：1000 0000

十进制：0

8.**Original\_or\_copy原始或者拷贝字段：**1位字段。置'1'时表示相关PES分组有效负载的内容是原始的；值为'0'表示相关PES分组有效负载的内容是一份拷贝。

十六机制：**80**

二进制：1000 0000

十进制：0

**00 00 01 E0 B8 D7 80 C0 0A 31 00 01 00 01 11 00 01 00 01**

**9. PTS\_DTS\_flags PTS DTS标志字段：**2位字段。当值为'10'时，PTS字段应出现在PES分组标题中；当值为'11'时，PTS字段和DTS字段都应出现在PES分组标题中；当值为'00'时，PTS字段和DTS字段都不出现在PES分组标题中。值'01'是不允许的。

十六机制：**C0**

二进制：1100 0000

**10. ESCR\_flag ESCR标志字段：**1位标志。置'1'时表示ESCR基础和扩展字段出现在PES分组标题中；值为'0'表示没有ESCR字段。

十六机制：**C0**

二进制：1100 0000

接下来这6个bit位都是0，说明接下来的标记都是0说明标记的字段在PES头里面都不存在。也是就11-15对应字段不用分析了。

**11. ES\_rate\_flag ES速率标志字段：**1位标志。置'1'时表示ES\_rate字段出现在PES分组标题中；值为'0'表示没有ES\_rate字段。

十六机制：**C0**

二进制：1100 0000

12.**DSM\_trick\_mode\_flagDSM特技方式标志字段：**1位标志。置'1'时表示有8位特技方式字段；值为'0'表示没有该字段。

十六机制：**C0**

二进制：1100 0000

13.**Additional\_copy\_info\_flag附加版权信息标志字段：**1位标志。置'1'时表示有附加拷贝信息字段；值为'0'表示没有该字段。

十六机制：**C0**

二进制：1100 0000

**14. PES\_CRC\_flag PES CRC标志字段：**1位标志。置'1'时表示CRC字段出现在PES分组标题中；值为'0'表示没有该字段。

十六机制：**C0**

二进制：1100 0000

**15. PES\_extension\_flag PES扩展字段：**1位标志。置'1'时表示PES分组标题中有扩展字段；值为'0'表示没有该字段。

十六机制：**C0**

二进制：1100 0000

16.**PES\_header\_data\_lengthPES标题数据长度字段：**8位字段。指出包含在PES分组标题中的可选字段和任何填充字节所占用的总字节数。该字段之前的字节指出了有无可选字段。

十六机制：**0A**

十进制：10

17.**Marker\_bit标记位字段：**值为'1'的1位字段。

18.**PTS 展示时间戳字段：**展现时间与解码时间的关系如下：PTS是一个编码在三个分离字段中的33位数字。它指出了基本流n的第k个展现单元在系统目标解码器中的展现时间tpn(k)。PTS的值以系统时钟频率的1/300(即90 kHz)为单位。展现时间由PTS根据式2-11计算而来。对编码展现时间戳频率的约束参见2.7.4。

PTS(k)＝((system\_clock\_frequency×tpn(k)) DIV 300) % 233           (2-11)

其中，tpn(k)是展现单元Pn(k)的展现时间。

对音频而言，若PES分组标题中有PTS，则它是指PES分组中开始的第一个存取单元。若PES分组中有音频存取单元的首字节，则有一个音频存取单元开始于该PES分组中。

对视频而言，若PES分组标题中有PTS，则它是指包含PES分组中开始的第一个画面起始码的存取单元。若PES分组中有画面起始码的首字节，则有一个画面起始码开始于该PES分组中。

对音频展现单元(PU)，low\_delay序列中的视频PU以及B画面，展现时间tpn(k)应等于tdn(k)。

对于非low\_delay中的I画面和P画面，在存取单元(AU) k和k'之间无解码不连续时，展现时间tpn(k)应等于下一个传输的I画面或P画面的解码时间tdn(k) (参见2.7.5)。若有解码不连续或流终止，则tpn(k)和tdn(k)之间的差别应与初始流一直延续，没有不连续也没有终止时完全相同。

注1： low\_delay序列是low\_delay标志被设置的视频序列(参见GB/T XXXX.2中的6.2.2.3)。

若音频中有滤波，则系统模型假定滤波不会导致延迟。因此，编码时PTS所涉及的采样与解码时PTS所涉及的采样是相同的。对于可伸缩编码，参见2.7.6。

十六机制：**31 00 01 00 01**

二进制：0011 0001 0000 0000 0000 0001 0000 0000 0000 0001

十进制：0

说明此处的PTS是0.

前面的0011取决于前面的**PTS\_DTS\_flags PTS DTS标记，前面是11则这里是0011，前面是10则这里是0010.**

19.**DTS 解码时间戳字段：**DTS是一个编码在三个分离字段中的33位数字。它指出了基本流n的第j个展现单元在系统目标解码器中的解码时间tdn(j)。DTS的值以系统时钟频率的1/300 (即90 kHz)为单位。解码时间由DTS根据式2-12计算而来：

DTS(j)＝((system\_clock\_frequency×tdn(j))DIV 300) % 233            (2-12)

其中，tdn(j)是存取单元An(j)的解码时间。

对视频而言，若PES分组标题中有DTS，则它是指包含PES分组中开始的第一个画面起始码的存取单元。若PES分组中有画面起始码的首字节，则该画面起始码开始于该PES分组中。

对于可伸缩编码，参见2.7.6。

十六机制：11**00 01 00 01**

二进制：0001 0001 0000 0000 0000 0001 0000 0000 0000 0001

十进制：0

说明此处的DTS是0.

**20. ESCR\_base ESCR\_extension ESCR字段：**

42位字段，分两部分编码。第一部分是一个长度为33位的字段，其值ESCR\_base(i)由式2-14给出；第二部分是一个长度为9位的字段，其值ESCR\_ext(i)由式2-15给出。ESCR字段指出了基本流中包含ESCR\_base最后一个比特的字节到达PES-STD输出端的期望时间(参见2.5.2.4)。

特别地

ESCR(i)＝ESCR\_base(i)×300＋ESCR\_ext(i)                                (2-13)

其中：

ESCR\_base(i)＝((system\_clock\_frequency×t(i))DIV 300) % 233            (2-14)

ESCR \_ext(i)＝((system\_clock\_frequency×t(i))DIV 1) %300               (2-15)

ESCR和ES\_rate字段(参见下面紧接的语义)包含与PES流序列相关的时间信息。这些字段应满足2.7.3中定义的约束

**21. ES\_rate 基本流速率字段：**22位无符号整数。对于PES流而言，它指出了系统目标解码器接收PES分组的速率。该字段在它所属的PES分组以及同一个PES流的后续PES分组中一直有效，直到遇到一个新的ES\_rate字段。该字段的值以50字节/秒为单位，且不能为0。该字段用于定义PES流的字节到达P-STD输入端的时间(参见2.5.2.4中的定义)。在各个PES分组中，编码在该字段中的值可能不同

**22. Trick\_mode\_control 特技方式控制字段：**3位字段。它表示作用于相关视频流的特技方式。对其它类型的基本流，该字段及其后5位的含义没有定义。一般不会在国标中出现该字段，前面标记为0.

**23. Fast\_forword 快进：**

**24. Freeze\_frame 冻结帧：**

**25. Fast\_reverse 快倒：**

**26. Slow\_reverse 慢倒：**

**27. Filed\_id 字段标识字段：**

2位字段，表示应该显示哪个(些)字段。根据表2-21对其进行编码。

表2-21  field\_id字段控制值

|  |  |
| --- | --- |
| 值 | 描  述 |
| '00' | 仅自顶向下播放 |
| '01' | 仅自底向上播放 |
| '10' | 播放所有帧 |
| '11' | 保留 |

**28. Intra\_slice\_refresh 片内参考字段：**

1位标志。置'1'时表示PES分组的视频数据编码片中可能有丢失的宏块；置'0'时，表示上述情况可能不出现。更多的信息可参见GB/T XXXX.2。解码器可以用前一个解码画面中同一个位置的宏块来代替丢失的宏块。

**29. Frequency\_truncation 频率截断字段：**

2位字段。指出在对PES分组中数据进行编码时可能用到受限系数集合。其值定义于表2-22。

表2-22  系数选择值

|  |  |
| --- | --- |
| 值 | 描述 |
| '00' | 仅DC系数非0 |
| '01' | 仅前三个系数非0 |
| '10' | 仅前六个系数非0 |
| '11' | 所有系数均可能非0 |

**30. Rep\_contrl 显示次数控制字段：**5位字段，指出隔行画面中每一字段的显示次数或渐进画面显示次数。对隔行画面而言，顶字段或底字段是否应首先显示是视频序列标题中trick\_mode\_control字段和top\_field\_first字段的功能。该字段值不能为'0'。

**31. Additional\_copy\_info 附加版权信息字段：**7位字段，包含与版权信息有关的专用数据。

**32. Previous\_PES\_Packet\_CRC 前PES分组CRC字段：**

16位字段。在对前一个PES分组(不包括该PES分组的标题)进行处理后，该字段包含一个在解码器的16个寄存器中生成0输出的CRC值。该CRC值与附录A中所定义的相类似，但具有以下多项式：

x16＋x12＋x5＋1

注2：  该CRC值是为了用于网络维护，例如将有间隙性错误的源隔离开来，而不是为了供基本流解码器使用。它仅用于计算数据字节，因为在传输过程中PES分组标题数据可能被修改。

**33. PES\_private\_data\_flag PES专用数据标志字段：**1位标志。置'1'时表示PES分组标题中包含专用数据；置'0'时表示PES分组标题中无专用数据。

**34. Pack\_header\_filed\_flag 包标题域标志字段：**1位标志。置'1'时表示PES分组标题中有GB/T AAAA.2包标题或节目流包标题。若该字段在包含于节目流中的PES分组中，其值应为'0'。在传输流中，当值为'0'时表示PES标题中无包标题。

**35. Program\_packet\_sequence\_counter\_flag 节目分组序列计数标志字段：**1位标志。值为'1'时表示PES分组有program\_packet\_sequence\_counter，MPEG1\_MPEG2\_identifier和original\_stuff\_length字段。值为'0'时表示PES分组标题中无这些字段。

**36. P-STD\_buffer\_flag P\_STD缓冲区标志字段：**1位标志。置'1'时表示PES分组标题中有P-STD\_buffer\_scale和P-STD\_buffer\_size字段。值为'0'时表示PES标题中无这些字段。

**37. PES\_extension\_flag\_2 PES扩展2标志字段：**1位标志，置'1'时表示有PES\_extension\_field\_length及相关字段。

**38. PES\_private\_flag PES专用数据字段：**16位字段。包含专用数据。这些数据与其前后的字段组合在一起时，不能与packet\_start\_code\_prefix (0x000001)冲突。

**39. Pack\_filed\_length 包字段长度字段：**8位字段。表示pack\_header\_field()以字节为单位时的长度。

**40. Program\_packet\_sequence\_counter 节目分组序列计数字段：**8位字段。表示pack\_header\_field()以字节为单位时的长度。

**41. MPEG1\_MPEG2\_identifier MPEG1 MPEG2标识符字段：**1位标志。置'1'时表示PES分组携带的信息来自于GB/T AAAA.1流；置'0'时表示PES分组携带的信息来自于节目流。

**42. Original\_stuff\_scale 初始填充长度字段：**6位字段。指定用于初始GB/T XXXX.1分组标题或初始GB/T AAAA.1分组标题中的填充字节数。

**43. P-STD\_buffer\_scale p-STD缓冲区比例字段：**1位字段。仅当该PES分组包含于节目流中时才有意义。它指出了用来解释后续P-STD\_buffer\_size字段的比例因子。若前面的stream\_id表示一个音频流，该字段值应为'0'；若前面的stream\_id表示一个视频流，该字段值应为'1'。对于所有的其它流类型，其值可以为'0'或'1'

**44. P-STD\_buffer\_size 缓冲区大小字段:**13位无符号整数。仅当该PES分组包含于节目流中时才有意义。它定义了P-STD输入缓冲区的大小BSn。若P-STD\_ buffer\_scale的值为'0'，那么P-STD\_buffer\_size以128字节为单位来度量缓冲区的大小。若P-STD\_buffer\_scale的值为'1'，那么P-STD\_buffer\_size以1024字节为单位来度量缓冲区的大小。

**45. PES\_extension\_filed\_length PES扩展长度字段：**7位字段。指出了跟在该字段之后在PES扩展字段中直到且包括任何保留字节的数据的字节长度。

**46. Stuffing\_byte 填充字节字段：**8位字段，其值恒定为'1111 1111'。可以由编码器插入以满足通道的需求等。解码器丢弃该字段。一个PES分组标题中只能出现32个填充字节

**47. PES\_packet\_data\_byte PES分组数据字节字段：**该字段应该是来自于由分组的stream\_id或PID所指定的基本流的连续数据字节。

**48. Padding\_byte 填充字节字段：**

**总结**

PES显然是里面所有头最复杂的，但是很多字段到底有没有定义，需要看第八个C0字节的六大标记，如果没有此标记解析起来相当简单。特别是PES扩展字段标记和特技方式字段。再后面是长度0A，后面的字节则就是PES重要信息PTS和DTS。当然有没有DTS还是要看前面的标记。一种时间戳占五个字节。

1. 先解析头的前缀四字节，根据stream\_id判断是音频还是视频；

2. 紧接着长度，则为下一个ps包的起始位置；

3. 判断PTS\_DTS\_flags PTS DTS标记；

4. 解析PTS和DTS值即可；

#### 1.2.6 PES中DTS&PTS

可以看到PTS/DTS（流识别码，用于区别不同性质ES）是打在PES包里面的，这两个参数是解决视音频同步显示，防止解码器输入缓存上溢或下溢的关键。PTS表示显示单元出现在系统目标解码器(STD: system target decoder)的时间，DTS表示将存取单元全部字节从STD的ES解码缓存器移走的时刻。每个I、P、B帧的包头都有一个PTS和DTS，但PTS与DTS对B帧都是一样的，无须标出B帧的DTS。对I帧和P帧，显示前一定要存储于视频解码器的重新排序缓存器中，经过延迟（重新排序）后再显示，一定要分别标明PTS和DTS。

关于音视频的同步：

除了PTS和DTS的配合工作外，还有一个重要的参数是SCR(system clock reference)。在编码的时候，PTS，DTS和SCR都是由STC(system time clock)生成的，在解码时，STC会再生，并通过锁相环路（PLL－phase lock loop），用本地SCR相位与输入的瞬时SCR相位锁相比较，以确定解码过程是否同步，若不同步，则用这个瞬时SCR调整27MHz的本地时钟频率。最后，PTS，DTS和SCR一起配合，解决视音频同步播放的问题。

MPEG-PS和MPTG-TS 媒体封装实例解析和说明

<https://mp.weixin.qq.com/s/_qHNTiP9CC6hx5sWvsf_Ww>

### 1.3 TS流格式

#### 1.3.1 简介

因为Apple公司提出的HLS(http live streaming)格式的流行，mpeg-ts封装的文件在互联网上已经随处可见。这套体系的强大之处就在于它的简单，只要你有工具可以把一个或多个视频文件切割成一堆的小ts文件，并且生成一个内容非常简单的m3u8文件，然后只要有一个标准的web server（apache, nginx）就能对外提供点播视频流媒体服务了。假如可以对直播数据流进行切片存储成小文件，那就又具备了直播流媒体服务能力。

介绍一个非常好用的解析库bitstream，并提供据此库实现的ts解析工具与大家共同学习探讨。在MPEG-2系统中编码端实现单一码流的详细过程如下：

（1）. A/D转换后，通过MPEG-2压缩编码得到ES基本流。该流数据量很大，并且只是I P B帧的取样信息。

（2）.通过PES打包器，打包并在每个帧中插入PTS/DTS标识，编程PES。原来是流的格式，现在成了数据包的分割形式。

（3）.PES根据需要打包成PS或者TS包进行存储

MPEG-TS一种标准数据容器格式，传输与存储音视频、节目与系统信息协议数据，应用于数字广播系统，譬如DVB,ATSC与IPTV。MPEG2/DVB是一种多媒体传输、复用技术，在数字电视广播中可提供数百个节目频道。注意，DVB全称为Digital Video Broadcasting，包括不同的系统，如卫星数字电视广播系统，有线数字电视广播系统，地面开路数字电视广播系统，交互式数字电视广播系统以及数字电视加扰系统。DVB系统标准是一种全球数字电视技术的标准。MPEG-2标准是定义比特流的语法与句法的一个ISO/IEC标准，即13818-1标准。DVB系统的核心技术是采用MPEG-2技术进行视频、音频的编码，使用统一的MPEG-2传输流（TS流）。

MPEG-2标准中，有两种不同的码流输出到信道，一种是节目码流（PS: Program Stream），适用于没有传输误差的场景；一种是传送流（TS：Transport Stream)，适用于有信道噪声的传输场景。节目流设计用于合理可靠的媒体，如光盘（如DVD），而传输流设计用于不太可靠的传输，即地面或卫星广播。此外，传输流可以携带多个节目。

MPEG-2 system（编号13818-1）是MPEG-2标准的其中一部分，该部分描述了多个视频，音频和数据多种基本流（ES）合成传输流（TS）和节目流（PS）的方式。一路TS比特流通常由连续的固定字节的TS包组成，所包含的内容有：

（1）.一路或多路视频流（多个PES包组成，每个PES包的PID是一致的，一个PES包可能由若干个TS包组成）

（2）.一路或多路音频流（通常为杜比的音频格式）

（3）.一路或多路字幕

（4）.PSI表格信息（Program Specific Information，包括PAT与PMT表，即节目关联表与节目映射表）

（5）.PES: Packetized Elementary Stream，一路基本码流（如MEPG2视频流）会在编码器端被打包成PES流，由多个PES包组成，打包的过程中主要加入了PTS/DTS信息。

TS过一段时间要有一个pat，否则可能被认为是错误的流。

PAT描述有多少路节目，每路节目的PMT表的PID是多少，PMT则描述了本节目有多少流，每一路流的类型与PID是多少，举个例子，你找个一个TS包，它的PID是0，说明它的负载内容是PAT信息，解析PAT信息，你发现节目1的PMT表的PID是0x10，接着，你在比特流中寻找一个PID为0x10的TS包，它的负载内容是节目1的PMT表信息，解析该PMT信息，你可以发现第一路流是MPEG2音频流，PID号0x21，第二路流是MPEG2视频流，PID号是0x22，第三路流是DVB字幕流，PID号是0x23，解析完毕，凡是比特流中PID号为0x22的TS包，所负载的内容为MPEG2视频流，把这些包一个一个找出来，把其中的有效码流一部分一部分的拼接起来，然后送给解码器去解码。PAT介绍有多少路节目和节目对应PMT,PMT介绍有多少流和对应流。

注意，就一般的视频流而言，只要拼接成一个完整的PES包，就可以送出去给解码器，然后再继续拼接下一个PES包。

TS PS是在pes流包得基础上插入了所需要得数据，如上图所示。TS是将pes所有内容进行分组切割成188字节，每个字节都分成ts包头和ts内容。

PSI：节目特定信息，该表格信息用来描述传送流的组成结构。PSI信息由四种类型的表组成，包括节目关联表（PAT），节目映射表（PMT），条件接收表（CAT），网络信息表（NIT）。PAT与PMT两张表帮助我们找到该传送流中的所有节目与流。



所以我们发送的TS数据包格式顺序为：

**PAT PMT PMT PMT DATA DATA ......PAT PMT PMT PMT DATA DATA......**

 MPEG-2规定一个PES包必须由整数个TS包传输，如果承载一个PES包的最后一个TS包没有被装满，则用填充字节进行填充。

第一个TS包：ts header + PES header + 部分ES

第二个TS包：ts header+部分ES

..........

最后一个包：ts header + 填充字节 + 部分ES

传输流包的长度必须是188字节，sync\_byte必须是0x47,任何有用的数据都不会是0xFF。PAT定义了当前TS流中所有的节目，其中PID恒为0x00，这个是关键。唯一分配给了PAT，也是分析PMT，音视频包的开始，它是PSI信息的根节点，要查找节目必须从PAT开始。

#### 1.3.2 TS结构

**TS是将pes所有内容进行分组切割成188字节，每个字节都分成ts包头和ts内容。**

**TS头一共32位 。TS头4字节，TS内容184字节。**

TS流的形成过程：

1) 将原始音视频数据压缩之后，压缩结果组成一个基本码流（ES）。

2) 对ES（基本码流）进行打包形成PES。

3) 在PES包中加入时间戳信息(PTS/DTS)。

4) 将PES包内容分配到一系列固定长度的传输包（TS Packet）中。

5) 在传输包中加入定时信息(PCR)。

6) 在传输包中加入节目专用信息(PSI) 。

7) 连续输出传输包形成具有恒定比特率的MPEG-TS流。

TS流的解析过程，可以说是生成的逆过程：

1) 从复用的MPEG-TS流中解析出TS包；

2) 从TS包中获取PAT及对应的PMT（PSI中的表格）；

3) 从而获取特定节目的音视频PID；

4) 通过PID筛选出特定音视频相关的TS包，并解析出PES；

5) 从PES中读取到PTS/DTS，并从PES中解析出基本码流ES；

6) 将ES交给解码器，获得压缩前的原始音视频数据

主要的字段解析如下：

**Sync byte:** 同步字节，值为0x47；可以理解为开始标志位。

**Transport error indicator:** 传输错误指示位，置1时，表示传送包中至少有一个不可纠正的错误位。

**Payload unit start indicator:** 负载单元起始指标位，表示TS包的有效净荷以PES/PSI包的第一个字节开始，举个例子，一个PES包可能由多个TS包构成，第一个TS包的负载单元起始指标位才会被置位。

**Transport priority:** 传输优先级，表明该包比同个PID的但未置位的TS包有更高的优先级。

**PID:** 该TS包的ID号，如果净荷是PAT包，则PID固定为0x00。

**Transport scrambling control:** 传输加扰控制位

**Adaption field control:** 自适应调整域控制位，置位则表明该TS包存在自适应调整字段。

**Continuity counter:** 连续计数器，随着具有相同PID的TS包的增加而增加，达到最大时恢复为0，如果两个连续相同PID的TS包具有相同的计数，则表明这两个包是一样的，只取一个解析即可。 可以理解为相同pid不同得ts包id

**Payload:** 负载内容，可能为PAT/PMT/PES。data\_byte为1B长度的数据，为负载字节。

#### IMG_256IMG_2561.3.3 Adaptation field解析

Adaptation field是自适应数据区，用以指示接下来的数据是否为逻辑上独立的一段内容，并且存放了PCR信息。是否存在由TS头中**Adaption field control**控制。

#### IMG_2561.3.4 PAT包解析

根据从头部取得的PID接下来要做就是确定这个PID所属的类型。而类型可分为TS固有类型与音视频流类型等。TS流的特点是没有严格意义上的起始或结束部分，所以也不能像mp4或者avi格式那样从meta域取得整体信息，这样我们势必要有一个办法知道当前的TS流的结构是怎样的，包含哪些音视频流等。所有的一切都要从一个叫做PAT的包开始。

TS固有类型的第一个要知道的就是PAT包，它的全称是Program Association Table（节目关联表）。PID的值等于0就判断是否PAT包。PAT包采用section格式封装，section结构简单说来可分为头部与负载区，头部区由若通用干字段组成。





**table\_id:** 标识一个TS PSI 分段的内容是节目关联分段，条件访问分段还是节目映射分段。对于PAT，置为0x00。

**section\_syntax\_indicator:** 对于PAT，置为0x01。

**section\_length**: 分段长度字段，其值为从section\_length（包括在内）到CRC\_32字段的字节数，其值不超过1021。

**transport\_stream\_id**: 区别与其他复用流的标识。

**version\_number:** PAT的版本号，如果PAT有变，则版本号加1。

**current\_next\_indicator**:置0时，表明该传送的表分段不能使用，下一个表分段才有效。

**section\_number**: 表明该TS包属于该PAT的第几个分段，分段号从0开始。

**last\_section\_number**: 表明最后一个分段号，同时表明该PAT的最大分段数目。一般，一个PAT表由一个TS包传送。

**program\_number**: 节目的编号。

**network\_PID:** NIT表的PID值。

**program\_map\_PID**: PMT表的PID值。 指定节目pmt对应pid值

**CRC\_32:** CRC校验。

#### IMG_256**1.3.5 PMT包解析**

PMT包括了一个节目相关的所有流信息。流这个概念对应的是一路音频或一路视频。我们知道，一个节目播放时有图像也有声音，但在编码层面它们往往是分开并且独立的，因为它们在最终呈现时是由不同的硬件设备来渲染的。音频与视频的同步则要借助于PCR，它是同步参考时钟，决定着哪一帧画面与哪一段音频相对应。





**table\_id**: 标识一个TS PSI 分段的内容是节目关联分段，条件访问分段还是节目映射分段。对于PMT，置为0x02。

**section\_syntax\_indicator**: 对于PMT，置为0x01。

**section\_length**: 分段长度字段，其值为从section\_length（包括在内）到CRC\_32字段的字节数，其值不超过1021。

**program\_number:** 表明一共有多少个节目。

**version\_number**: PMT的版本号，如果字段中有关信息有变，则版本号以32为模加1。版本号是对一个节目的定义。

**current\_next\_indicator**:置0时，表明该传送的表分段不能使用，下一个表分段才有效。

section\_number: 总为0x00。

**last\_section\_number**: 总为0x00。

**PCR\_PID:** 指示含有该节目的PCR字段的TS包的PID。

**program\_info\_length**: 表明跟随其后的对节目信息描述的字节数，也就是第一个N loop descriptors的字节数。

**stream\_type:** 表明PES流的类型。譬如，0x01表明是MPEG-1视频，0X03表明是MPEG-1音频。

**elementary\_PID**: 表明该负载有该PES流的TS包的PID值。

**ES\_info\_length**: 表明跟随其后的描述相关节目元素的字节数，也就是第二个N loop descriptors的字节数。

**CRC\_32:** 在CEDARX代码中仅对DVB的场景下作校验。



Streamtype值的列表：

0x00 ITU-T | ISO/IEC Reserved

0x01 ISO/IEC 11172 Video

0x02 ITU-T Rec. H.262 | ISO/IEC 13818-2 Video or ISO/IEC 11172-2 constrainedparameter video stream

0x03 ISO/IEC 11172 Audio

0x04 ISO/IEC 13818-3 Audio

0x05 ITU-T Rec. H.222.0 | ISO/IEC 13818-1 private\_sections

0x06 ITU-T Rec. H.222.0 | ISO/IEC 13818-1 PES packets containing private data

0x07 ISO/IEC 13522 MHEG

0x08 ITU-T Rec. H.222.0 | ISO/IEC 13818-1 Annex A DSM-CC

0x09 ITU-T Rec. H.222.1

0x0A ISO/IEC 13818-6 type A

0x0B ISO/IEC 13818-6 type B

0x0C ISO/IEC 13818-6 type C

0x0D ISO/IEC 13818-6 type D

0x0E ITU-T Rec. H.222.0 | ISO/IEC 13818-1 auxiliary

0x0F ISO/IEC 13818-7 Audio with ADTS transport syntax

0x10 ISO/IEC 14496-2 Visual

0x11 ISO/IEC 14496-3 Audio with the LATM transport syntax as defined in ISO/IEC14496-3 / AMD 1

0x12 ISO/IEC 14496-1 SL-packetized stream or FlexMux stream carried in PES packets

0x13 ISO/IEC 14496-1 SL-packetized stream or FlexMux stream carried inISO/IEC14496\_sections.

0x14 ISO/IEC 13818-6 Synchronized Download Protocol

0x15-0x7F ITU-T Rec. H.222.0 | ISO/IEC 13818-1 Reserved

0x80-0xFF User Private.

#### 1.3.6 SDT包解析

SDT是流信息描述表，它的作用是传输节目的内容名称。例如我们在制作节目单时就可以这个表中取出cctv 1, cctv2这样的内容名称。



音视频封装：MPTG2-TS 媒体封装实例解析和说明

<https://mp.weixin.qq.com/s?__biz=MzI0NTMxMjA1MQ==&mid=2247483684&idx=1&sn=dce8821fbf385a808af96a9bd2f85c45&chksm=e9513e78de26b76e8bb7d310e61a823bb81cd7ca34528780253e6830c80d56963befcdf5324f&token=1491357593&lang=zh_CN#rd>

基本上从TS流分析H264码流这块，简单的方法就是先分析PID为0x00的PAT，然后分析PMT里面对音频和视频的PID定义。

然后以定义PID分别过滤音频，然后再过滤视频，这样就完成了从TS文件到音视频数据的过滤。

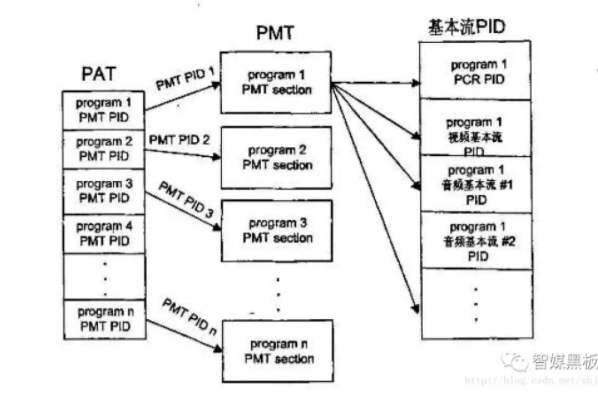
1. 第一步找PAT,注重分析PMT的表的PID；

2. 第二步找PMT，分析里面是否含有音视频，音视频编码格式，音视频的PID；

3. 第三步根据音视频的PID过滤音频TS包和视频TS包

4. 拿掉TS头，PES头，后即可得到音频数据和视频数据的裸码流；

TS文件解复用示意图如下：



### 1.4 PTS DTS PCR同步机制

#### 1.4.1 PTS DTS PCR介绍

当新节目的PCR到达解码器时，需要更新时间基点，STC就被置位。通常第一个从解复用器中 解出的PCR被直接装入到STC计数器，其后PLL闭环操作。每当一个新节目的PCR到达解码器时，此值被认为是锁相环的参考频率，用来与STC的当前值 比较，产生的差值e经过脉宽调制后被输入低通滤波器并经放大，输出控制信号f，用来控制振荡器（VCO）的瞬时频率，VCO输出的频率是在27MHz左右 振荡的信号，作为解码器的系统时钟。27MHz时钟经过波形整理后输入到计数器中，产生当前的STC值，其33bits的90kHz部分用于和 PTS/DTS比较，产生解码和显示的同步信号。其实pcr用于生成27mz的系统时钟与PTS/DTS进行比较进行同步。

MPEG-2系统用于视音频同步以及系统时钟恢复的时间标签分别在ES，PES和TS这3个层次中。在ES层，与同步有关的主要是视频缓冲验证VBV（Video Buffer Verifier），用以防止解码器的缓冲器出现上溢或者下溢；在PES层，主要是在PES头信息里出现的显示时间标签PTS（Presentation Time Stamp）和解码时间标签DTS（Decoding Time Stamp）；在TS层中，TS头信息包含了节目时钟参考PCR（Program Clock Reference），用于恢复出与编码端一致的系统时序时钟STC（System Time Clock）。在节目流PS包头中加入SCR,它的作用与PCR域相似。标准规定在原始音频和视频流中，PTS的间隔不能超过0.7s，而出现在TS包头的PCR间隔不能超过0.1s。

编码器中有一个系统时钟STC（其频率是27MHz）（是编码器中自带的），此时钟用来产生指示音视频的正确显示和解码的时间标签，同时可用来指示在采样过程中系统时钟本身的瞬时值。指示音视频显示时间的时间标签称为显示时间标签（PTS），指示音视频的解码时间标签称为解码时间标签（DTS），指示系统时钟本身的瞬时值的时间标签称为节目参考时钟标签（PCR）。解码器通过VBV\_delay（视频流延时值，在解码时利用视频流缓冲区把视频流缓存到相应的vbv\_delay时间后，再启动解码器解码、显示、实现音视频的同步。vbv\_delay存在于视频ES的头部，长度为16bit）的数值来确定解码的开始，用解码时间标签（DTS）和显示时间标签（PTS）来确定解码和显示的次序，用PCR来获得系统时钟的同步。解码器中的系统时钟及其同步就是依靠这些时间标签来进行恢复和修正的。

PCR的插入必须在PCR字段的最后离开复用器的那一时刻，同时把27MHz系统时钟的采样瞬时值作为PCR字段插入到相应的PCR域。是放在TS包头的自适应区中传送。27 MHz时钟经波形整理后分两路，一路是由27MHz脉冲直接触发计数器生成扩展域PCR\_ext，长度为9bits。另一路经一个300分频器后的90 kHz脉冲送入一个33位计数器生成90KHZ基值，列入PCR\_base（基值域），长度33bits，用于和PTS/DTS比较，产生解码和显示所需要的同步信号。这两部分被置入PCR域，共同组成了42位的PCR。

PCR-Base(i)=90kHz×t(i), mod(233)

PCR-Ext(i)=27MHz×t(i), mod(300)

PCR(i)=PCR-Base(i)×300＋PCR-Ext(i)

PTS域为33bits，它也是一个33b的计数值，也是对系统时钟的300分频的时钟的计数值。被编码成为3个独立的字段，表示此分组中第一个访问单元在系统目标解码器中的预定显示时间。

DTS域也为33bits，编码成为3个独立的字段，表示此分组中第一个访问单元在系统目标解码器中的预定解码时间。DTS就视频来说，因为视频编码的时候用到了双向预测，一个图像单元被解出，并非马上就被显示，可能在存储器中留一段时间，作为其余图像单元的解码参考，在被参考完毕后，才被显示。针对视频的显示，Mpeg还提出了一个视频PTS。针对音频和视频的同步显示，又提出了一个音频PTS。由于声音没有用到双向预测，它的解码次序就是它的显示次序，故对它只提出PTS的概念。

解码器在解码时，首先利用PCR重建和编码器同步的27 MHz系统时钟，恢复27 MHz系统时钟后，再利用PES流中的DTS，PTS进行音频与视频间的同步，同时利用VBV\_Delay设置解码器的缓冲时长，后启动初始解码。下图是利用PCR恢复STC的原理图。

PCR-Base的作用是在解码器切换节目时，提供对解码器PCR计数器的初始值，以让该PCR值与PTS、DTS最大可能地达到相同的时间起点。PCR-ext的作用是通过解码器端的锁相环电路修正解码器的系统时钟，使其达到和编码器一致的27MHz。

在PCR域的33 bits中的90 kHz部分（即PCR-Base域）用于与PTS和DTS作比较，当二者相同时，相应的单元被显示或者解码。但按照MPEG-2标准的规定，PTS/DTS位于PES包的包头中，而解复用器所解出的PES包头的字节不被送入到任何基本流解码器的输入缓冲区，仅用于控制各解码器工作。每当解复用器解到一个PTS/DTS时，PTS/DTS就作为其后送入各个基本流解码器的输入缓冲器中的数据的显示/解码时间，但此时送入信道缓冲器中的基本流并不一定立即被解码，因此基本流解码器必须记录当前这个PTS/DTS及所对应的码流位置，以利于后面的解码和显示同步。

针对以上分析，我们采用如下方法来达到解码与显示的同步：在每一个基本流解码器中建立一个33 bits的PTS/DTS先入先出存储器（FIFO），用以存放PTS/DTS值，同时在解复用器中加入一个检测电路，每当码流开始新的一帧图像时，则产生一个信号，用来通知基本流解码器，并将该帧图像的PTS/DTS写入到FIFO中。每当解码器解到一帧图像时，就从FIFO中读出相应的PTS/DTS，此PTS/DTS与STC进行比较，当两者相等时，当前图像就开始显示/解码。

Mpeg-2规定的系统时钟频率为27MHz，传输流中的PCR，PTS/DTS等均为对该共同系统时钟的采样值。解码端捕获PCR，恢复出本地的STC，作为音视频同步控制的基准，并依据PTS（DTS）时间标签来安排解码和显示时间表，使音视频分别同步于STC，以实现音视频之间的同步。标准规定在原始音频和视频流中，PTS的间隔不能超过0.7s，而出现在TS包头的PCR间隔不能超过0.1s。

在MPEG-2标准中，并非每一个PES包或每一帧图像均有PTS和DTS，一般PS流中每两个PTS之间的间隔是0.7 s，而TS流中每两个PTS之间的间隔是0.1 s，所以在没有PTS和DTS的一帧图像出现时，我们可以在前一PTS/DTS的基础上加一增量得到对应该帧图像的PTS/DTS，并将计算出来的新的PTS/DTS插入到存放PTS/DTS的FIFO中去，

即：PTSn=PTSn－1+ΔPTS（n为第n帧图像）。

解码器同步算法总结如下：

（1）解码器从输入码流的包头中解出时间信息PCR送入到系统时间时钟恢复电路；系统时间时钟恢复电路在接收到每一个新的PCR时，进行本地系统时间时钟恢复和锁相。

（2）解复用器后，从PES包头中解出显示时间标签PTS和解码时间标签DTS，并送入到基本流解码器中。

（3）基本流解码器在接收到新的PTS/DTS后，存入对应的FIFO(先进先处存储器)中进行管理；对于没有PTS/DTS的显示单元，需要对其时间标签进行插值，并送入到FIFO中管理。

（4）每一显示单元开始解码前，用其对应的DTS与STC进行比较，当STC与DTS相等时开始解码；

（5）每一显示单元开始显示前，用其对应的PTS与STC进行比较，当STC与PTS相等时开始显示。

27 MHz系统时钟经过300分频后，得到本地的33 bits PCR\_Base，该时钟与寄存器中当前图像的PTS/DTS进行比较，系统软件根据比较结果做出相应的处理：

（1）若当前的PTS/DTS比PCR计数器的值小于半帧以上，即PTS\_Base≤-ΔPTS/2，此时说明系统解码过慢，解码器处于失步状态，应根据该帧的结构做出相应的同步调整；

（2）若当前的PTS/DTS比PCR计数器的值在半帧时间以内，我们认为此时系统解码正常，立即显示/解码当前帧；

（3）若当前的PTS/DTS大于PCR计数器的值，则此时解码器稍快，在这种情况下，只需等到PCR与PTS/DTS相等时，就可显示/解码。

#### 1.4.2 PCR测量

㈠PCR 精度(PCR\_AC)：接收PCR中所含27MHz时钟的不准确度，但不包含任何传输定时损伤。测量时传输码流中PCR字节位置作为起点，计算出PCR 到达时间。

㈡PCR漂移率(PCR\_DR)：PCR漂移率测量指的是PCR中的低频误差并计入了由PCR发生和再生引起的误差以及由传输损伤所引起的到达时间误差。即PCR内所含27MHz时钟的低频变化速率，测量时以某一稳定外部基准作为参考，因此包含了任何传输定时变化。下面的例子可用来说明PCR 漂移率的重要性：我们来看一下由PLL控制的本地时钟再生器。再生器输出端的信号频率应当跟踪输入信号频率，随着输入信号频率的变化，PLL也随之作出响应，本地再生器将跟踪输入信号，随着输入信号变化速率的增加，PLL的跟踪能力会有所降低，最后将导致本地发生信号失锁而丢失节目。PCR 漂移率的测量给出了一种测定方法，即如何更好地使去复用器/解码器或再复用器的节目时钟锁定于输入TS中所包含的PCR(注意这里指的是漂移率的测量而不是绝对频率的测量，接收机端的PLL电路可以锁定于系统时

钟，该时钟有着固定的频率误差－如果是快速变化或漂移率大，则会出现问题)。

㈢PCR总抖动(PCR\_OJ)：PCR内所含27MHz时钟的高频变化，测量时以某一稳定外部基准作为参考，因此包含了任何传输定时变化。它是PCR测量中最重要的一项，是PCR中高频误差的总体测量并且还包括来自PCR发生和PCR再生的误差以及由传输损伤引起的到达时间误差。例如该项传输损伤可能会引入到接收机中的RF解调器中。接收机内再生的TS时钟以及去复用器/解码器所使用的TS时钟可能包含有基本高频变化，它会对PCR的恢复带来直接影响。在这种情形下，解码器在复原PCR中所发现的PCR到达时间误差可能是±500ns容限的许多倍，给PCR精度带来不利。任何解码器均靠接收PCR以准确再生其节目时钟并提供稳定的视频输出。与此相同，任何再复用器均靠接收PCR以在其输出端精确地重新作出PCR标记。PCR的到达时间是严格的；因此，测量包含有到达时间误差和精度误差的总抖动（PCR\_OJ）就是非常必要的。在理想运行系统中，传输损伤应当是可以设计的。然而在目前典型的分配网络中，包含有再复用器、ATM 层等各级链路，它们均会引入误差。PCR\_AC是在复用器输出端直接测量的数值，不过它在网络中的运行监视常会使人迷惑不解，因为按照定义，它应当消除一切传输损伤。

㈣PCR频率偏置(PCR\_FO)：即PCR中所含27MHz 时钟的频率偏置，测量时以某一稳定的外部基准作为参考

备注：

1. SCR

SCR是存在于PS中的，即PS的pack里面的一个field。他用来指定这个PS的pack期望的到达decoder的时间。

2. ESCR

ESCR是位于PES里面的，即PES的header里面可能会有，当PES的packet要表示一个PES stream的时候，PES packet里面的ESCR就表示这个PES packet的所期望的到达decoder的时间。

3．PCR

PCR是TS里面的，即TS packet的header里面可能会有，他用来指定所期望的该ts packet到达decoder的时间，他的作用于SCR类似。

4．DTS, PTS

对于一个ES来说，比如视频，他又许多I,P,B帧，而P, B帧都是以I，P帧作为参考。由于B帧是前向后向参考，因此要对B帧作decode的话，就必须先decode该B帧后面的P,或者I帧，于是，decode的时间与帧的真正的present的时间就不一致了，按照DTS一次对各个帧进行decode，然后再按照PTS对各个帧进行展现。有时候PES包头里面也会有DTS，PTS，对于PTS来说，他代表了这个PES包得payload里面的第一个完整地audio access unit或者video access unit的PTS时间（并不是每个audio/video access unit都带有PTS/DTS，因此，你可以在PES里面指定一个，作为开始）。PES包头的DTS也是这个原理，只不过注意的是：对于video来说他的DTS和PTS是可以不一样的，因为B帧的存在使其顺序可以倒置。而对于audio来说，audio没有双向的预测，他的DTS和PTS可以看成是一个顺序的，因此可一直采用一个，即可只采用PTS。

#### 1.4.3 ES PES TS

MPEG2系统用于视音频同步以及系统时钟恢复的时间标签分别在ES，PES和TS这3个层次中。

在TS 层, TS头信息包含了节目时钟参考PCR(Program Clock Reference),用于恢复出与编码端一致的系统时序时钟STC(System Time Clock)。

在PES层, 在PES头信息里包含有表示时间戳PTS(Presentation Time Stamp)和解码时间戳DTS(Decoding Time Stamp);

在ES 层, 与同步有关的主要是视频缓冲验证VBV(Video Buffer Verifier),用以防止解码器的缓冲器出现上溢或者下溢;

标准规定在原始音频和视频流中,PTS的间隔不能超过0.7s，出现在TS包头的PCR间隔不能超过0.1s。



从ES到PES的示意图，插入pts dts如图如上。

在插入PTS/DTS标志时，

对于B帧, 由于在B帧PTS和DTS是相等的，所以无须在B帧插入DTS(参见图1).

对于I帧和P帧, 由于经过复用后, 数据包的顺序会发生变化,显示前一定要存储于视频解码器的排序缓存器中，经过从新排序后再显示，所以一定要同时插入PTS和DTS作为从新排序的依据.

编码器

系统时钟STC:

编码器中有一个系统时钟(其频率是27MHz),

此时钟用来产生指示音视频的正确显示和解码的时间戳,

同时可用来指示在采样过程中系统时钟本身的瞬时值。

PCR(Program Clock Reference):

指示系统时钟本身的瞬时值的时间标签称为节目参考时钟标签(PCR)。

PCR的插入必须在PCR字段的最后离开复用器的那一时刻,

同时把27MHz系统时钟的采样瞬时值作为PCR字段插入到相应的PCR域。

它是放在TS包头的自适应区中传送.

27MHz的系统时钟STC经波形整理后分成两路:

PCR\_ext (9bits ), 由27MHz脉冲直接触发计数器生成扩展域.

PCR\_base(33bits), 经300分频器分频成90kHz脉冲送入一个33位计数器生成90kHz基值,用于和PTS/DTS比较，产生解码和显示所需要的同步信号.这两部分被置入PCR域，共同组成42位的PCR.

输入到T-STD解码器的第i个字节的PCR值:

PCR(i) = PCR\_base(i)\*300 + PCR\_ext(i) i: 包含program\_clock\_reference\_base域的最后一个比特的字节号.

PCR\_base(i) = ((system\_clock\_frequency \* t(i)) / 300) % 2^33

PCR\_ext(i) = ((system\_clock\_frequency \* t(i)) / 1 ) % 300 t(i): 字节i的编码时间.

例如:

时间"03:02:29.012"的PCR计算如下:

03:02:29.012 = ((3 \* 60) + 2) \* 60 + 29.012 = 10949.012s

PCR\_base = ((27 000 000 \* 10949.012) / 300) % 2^33 = 98 541 080

PCR\_ext = ((27 000 000 \* 10949.012) / 1 ) % 300 = 0

PCR = 98 541 080 \* 300 + 0 = 295 623 324 000

PCR-base的作用:

a. 与PTS和DTS作比较, 当二者相同时, 相应的单元被显示或者解码.

b. 在解码器切换节目时,提供对解码器PCR计数器的初始值,

以让该PCR值与PTS、DTS最大可能地达到相同的时间起点.

PCR-ext的作用:

通过解码器端的锁相环路修正解码器的系统时钟, 使其达到和编码器一致的27MHz.

PTS(Presentation Time Stamp):  
  指示音视频显示时间的时间戳称为显示时间戳(PTS);  
  
  PTS域为33bits, 是对系统时钟的300分频的时钟的计数值.  
  它被编码成为3个独立的字段:  
     PTS[32..30][29..15][14..0]  
  表示此分组中第一个访问单元在系统目标解码器中的预定显示时间.  
  
PTS值为:  
  PTS(k) = ((system\_clock\_frequency \* TPn(k)) / 300) % 2^33  
TPn(k): 表示单元Pn(k)的表示时间.  
  
DTS(Decoding Time Stamp):  
  指示音视频的解码时间戳称为解码时间戳(DTS),  
    
  DTS域为33bits,编码成为3个独立的字段:  
     DTS[32..30][29..15][14..0]  
  表示此分组中第一个访问单元在系统目标解码器中的预定解码时间.  
  
DTS值为:  
  DTS(j) = ((system\_clock\_frequency \* TDn(j)) / 300) % 2^33  
TDn(j): 第n个ES流的第j个存取单元An(j)的解码时间.  
     
  DTS就视频来说,因为视频编码的时候用到了双向预测,  
  一个图像单元被解出,并非马上就被显示,可能在存储器中留一段时间,作为其余图像单元的解码参考,  
  在被参考完毕后,才被显示.  
  
音频PTS:  
  针对音频和视频的同步显示，MPEG提出了一个音频PTS.  
  由于声音没有用到双向预测，它的解码次序就是它的显示次序，故它只有PTS.

VBV\_delay:  
  视频流延时值,  
  在解码时利用视频流缓冲区把视频流缓存到相应的vbv\_delay时间后,  
  再启动解码器解码、显示、实现音视频的同步.  
  VBV\_delay存在于视频ES的头部，长度为16bit.  
  
解码器  
首先,解析PCR, 重建和编码器同步的27MHz系统时钟, 恢复27MHz系统时钟后;再通过VBV\_delay(视频流延时值)的数值来确定解码的开始;之后,利用PES流中解码时间戳(DTS)和显示时间戳(PTS)来确定解码和显示的次序.用PCR来对系统时钟进行修正.  
解码器同步算法总结如下:  
(1).解码器从输入码流的包头中解出时间信息PCR送入到系统时间时钟恢复电路;系统时间时钟恢复电路在接收到每一个新的PCR时，进行本地系统时间时钟恢复和锁相。  
(2).解复用器后，从PES包头中解出显示时间标签PTS和解码时间标签DTS，并送入到基本流解码器中。  
(3).基本流解码器在接收到新的PTS/DTS后，存入对应的FIFO(先进先处存储器)中进行管理；对于没有PTS/DTS的显示单元，需要对其时间标签进行插值，并送入到FIFO中管理。  
(4).每一显示单元开始解码前，用其对应的DTS与STC进行比较，当STC与DTS相等时开始解码；  
(5).每一显示单元开始显示前，用其对应的PTS与STC进行比较，当STC与PTS相等时开始显示。

27 MHz系统时钟经过300分频后，得到本地的33 bits PCR\_Base, 该时钟与寄存器中当前图像的PTS/DTS进行比较,  
系统软件根据比较结果做出相应的处理:  
(1).  若当前的PTS/DTS比PCR计数器的值小于半帧以上，即PTS\_Base≤-ΔPTS/2,  
      此时说明系统解码过慢，解码器处于失步状态，应根据该帧的结构做出相应的同步调整；  
(2).  若当前的PTS/DTS比PCR计数器的值在半帧时间以内，  
      我们认为此时系统解码正常，立即显示/解码当前帧；  
(3).  若当前的PTS/DTS大于PCR计数器的值，则此时解码器稍快，  
      在这种情况下，只需等到PCR与PTS/DTS相等时，就可显示/解码。

#### 1.4.4 PTS DTS STC PCR

PTS：Presentation Time Stamp，显示时间，pes中添加

DTS：Decoding Time Stamp，编解码时间，pes中添加

PCR：Program Clock Reference，节目时钟参考，ps中添加

STC：System Time Clock，系统时钟，27Mhz

Vbv：上下溢出时间，es中添加

SCR：存在于PS中的，即PS的pack里面的一个field。他用来指定这个PS的pack期望的到达decoder的时间。

ESCR：是位于PES里面的，即PES的header里面可能会有，当PES的packet要表示一个PES stream的时候，PES packet里面的ESCR就表示这个PES packet的所期望的到达decoder的时间。

PCR：是TS里面的，即TS packet的header里面可能会有，他用来指定所期望的该ts packet到达decoder的时间，他的作用于SCR类似。

#### 1.4.5 参考链接

1.Mpeg-2的同步及时间恢复--STC，PCR，DTS，PTS

<https://www.bbsmax.com/A/x9J2omDW56/>

2.PTS PCR DTS 详解

<https://www.cnblogs.com/jiangzhaowei/p/4317755.html>

PES是ps 和ts的基础

## H.264

### 2.1 简介

#### 2.1.1 H264和MPEG-4标准

视频编解码技术有两套标准，国际电联（ITU-T）的标准H.261、H.263、H.263+等；还有ISO 的MPEG标准Mpeg1、Mpeg2、Mpeg4等等。H.264/AVC是两大组织集合H.263+和Mpeg4的优点联合推出的最新标准，最具价值的部分无疑是更高的数据压缩比。在同等的图像质量条件下，H.264的数据压缩比能比H.263高2倍，比MPEG-4高1.5倍。H.264/AVC 是ITU-T 视频编码专家组（VCEG）和ISO/IEC 动态图像专家组（MPEG ）联合组成的联合视频组（JVT）共同努力制订的新一代视频编码标准，它最大的优势是具有很高的数据压缩比率。

H.264/AVC标准是由ITU-T和ISO/IEC联合开发的，定位于覆盖整个视频应用领域，包括：低码率的无线应用、标准清晰度和高清晰度的电视广播应用、Internet上的视频流应用，传输高清晰度的DVD视频以及应用于数码相机的高质量视频应用等等。

ITU-T给这个标准命名为H.264（以前叫做H.26L），而ISO/IEC称它为MPEG-4 高级视频编码（Advanced Video Coding，AVC）,并且它将成为MPEG-4标准的第10部分。既然AVC是当前MPEG-4标准的拓展，那么它必然将受益于MPEG-4开发良好的基础结构（比如系统分层和音频等）。

H.264最具价值的部分无疑是更高的数据压缩比。在同等的图像质量条件下，H.264的数据压缩比能比当前DVD系统中使用的MPEG-2高2-3倍，比MPEG-4高1.5-2倍。在网络传输过程中所需要的带宽更少，也更加经济。在MPEG-4需要6Mbps的传输速率匹配时，H.264只需要3Mbps-4Mbps的传输速率。

H.264获得优越性能的代价是计算复杂度的大幅增加，例如分层设计、多帧参论、多模式运动估计、改进的帧内预测等，这些都显著提高了预测精度，从而获得比其他标准好得多的压缩性能。

H.264 与MPEG-4的比较：在极低码率（32-128Kbps）的情况下，H.264与MPEG-4相比具有性能倍增效应，即：相同码率的H.26L媒体流和MPEG-4媒体流相比，H.26L拥有大约3个分贝的增益（画质水平倍增）。 32Kbps的H.26L媒体流，其信躁比与128K的MPEG-4媒体流相近。即在同样的画面质量下，H.264的码率仅仅为MPEG-4的四分之一。

#### 2.1.2 H.264组成

H.264 是一次概念的革新，它打破常规，完全没有 I 帧、P帧、B 帧的概念，也没有 IDR帧的概念。对于 H.264中出现的一些概念从大到小排序依次是：序列、图像、片组、片、NALU、宏块、亚宏块、块、像素。这里有几点值得说明：

（1）、在 H.264协议中图像是个集合概念，顶场、底场、帧都可以称为图像（本文图像概念时都是集合概念）。因此我们可以知道，对于H.264 协议来说，我们平常所熟悉的那些称呼，例如：I 帧、P 帧、B帧等等，实际上都是我们把图像这个概念具体化和细小化了。我们在 H.264里提到的“帧”通常就是指不分场的图像；

（2）、如果不采用FMO（灵活宏块排序） 机制，则一幅图像只有一个片组；

（3）、如果不使用多个片，则一个片组只有一个片；

（4）、如果不采用DP（数据分割）机制，则一个片就是一个NALU，一个 NALU 也就是一个片。

否则，一个片由 三个 NALU 组成（即标准“表7-1”中 nal\_unit\_type 值为2、3、4 的三个 NALU 属于 一个片）；

一幅图像根据组成它的片类型来分，可以分为标准“表7-5”中的 8种类型。我们平常应用中所最常见到的其实是这些类型的特例。例如：我们平常所谓的“I帧”和“IDR 帧”，其实是 primary\_pic\_type 值为 0的图像，我们平常所谓的“P帧”其实是 primary\_pic\_type 值为 1的图像的特例，我们平常所谓的“B帧”其实是 primary\_pic\_type 值为 2的图像的特例。



一幅图像根据概念来分可以分为两种：IDR 图像和非 IDR图像。一幅图像是否是 IDR 图像是由组成该图像的 NALU决定的，如果组成该图像的 NALU 为标准“表7-1”中 nal\_unit\_type 值为 5 的NALU，则该图像为 IDR 图像，否则为非 IDR图像。这里也有几点值得说明：

（1）、nal\_unit\_type 值为 5 的 NALU 只会出现在 IDR 图像中，而 IDR图像中的所有 NALU 都是nal\_unit\_type 值为 5 的NALU；

（2）、我们以组成一幅图像的片的类型来区分该图像是否是IDR 图像是错误的。

例如：一幅图像中的所有片都是 I 片并不代表这幅图像就是 IDR 图像。因为 I片也可以从属于 nal\_unit\_type 值为 1 的 NALU 也即非IDR图像有可能全部包含I片。只不过我们平常最常见到的形式是：所有片都是 I片的图像就是 IDR 图像。其实这个时候 IDR 图像的概念也被我们具体化和细小化了。但IDR图像必定全部包含I片或SI片，不过只有用NALU的类型才能判断是不是IDR图像

一幅图像由 1～N个片组组成，而每一个片组又由一个或若干个片组成一个片由一个NALU或三个NALU（假如有数据分割）组成。图像解码过程中总是按照片进行解码，然后按照片组将解码宏块重组成图像。从这种意义上讲，片实际是最大的解码单元。

1 frame的数据可以分为多个slice。每个slice中的数据，在帧内预测只用到自己slice的数据，与其他slice数据没有依赖关系。 NAL 是用来将编码的数据进行大包的。 比如，每一个slice 数据可以放在NAL 包中。I frame 是自己独立编码，不依赖于其他frame 数据。P frame 依赖 I frame 数据。B frame 依赖 I frame, P frame 或其他 B frame 数据。

Slice是片的意思，264中把图像分成一帧（frame）或两场（field），而帧又可以分成一个或几个片（Slilce）；片由宏块（MB）组成。宏块是编码处理的基本单元。

**场和帧** ：视频的一场或一帧可用来产生一个编码图像。在电视中，为减少大面积闪烁现象，把一帧分成两个隔行的场。

**片**： 每个图象中，若干宏块被排列成片的形式。片分为I片、B片、P片和其他一些片。

I片只包含I宏块，P片可包含P和I宏块，而B片可包含B和I宏块。片的主要作用是用作宏块（Macroblock）的载体（ps：下面会介绍到宏块的概念）。片之所以被创造出来，主要目的是为限制误码的扩散和传输。 如何限制误码的扩散和传输？ 每个片（slice）都应该是互相独立被传输的，某片的预测（片（slice）内预测和片（slice）间预测）不能以其它片中的宏块（Macroblock）为参考图像。

I宏块利用从当前片中已解码的像素作为参考进行帧内预测。

P宏块利用前面已编码图象作为参考图象进行帧内预测。

B宏块则利用双向的参考图象（前一帧和后一帧）进行帧内预测。

片的目的是为了限制误码的扩散和传输，使编码片相互间是独立的。

某片的预测不能以其它片中的宏块为参考图像，这样某一片中的预测误差才不会传播到其它片中去。

对于切片（slice）来讲，分为以下几种类型：

I片：只包 I宏块，I 宏块利用从当前片中已解码的像素作为参考进行帧内预测(不能取其它片中的已解码像素作为参考进行帧内预测)。

P片：可包 P和I宏块，P 宏块利用前面已编码图象作为参考图象进行帧内预测，一个帧内编码的宏块可进一步作宏块的分割:即 16×16、16×8、8×16 或 8×8 亮度像素块(以及附带的彩色像素);如果选了 8×8 的子宏块，则可再分成各种子宏块的分割，其尺寸为 8×8、8×4、4×8 或 4×4 亮度像素块(以及附带的彩色像素)。

B片：可包 B和I宏块，B 宏块则利用双向的参考图象(当前和 来的已编码图象帧)进行帧内预测。

SP片(切换P)：用于不同编码流之间的切换，包含 P 和/或 I 宏块

SI片：扩展档次中必须具有的切换，它包 了一种特殊类型的编码宏块，叫做 SI 宏块，SI 也是扩展档次中的必备功能。

**宏块** ：一个编码图像通常划分成若干宏块组成，一个宏块由一个16×16亮度像素和附加的一个8×8 Cb和一个8×8 Cr彩色像素块组成。

数据之间的关系：H264结构中，一个视频图像编码后的数据叫做一帧，一帧由一个片（slice）或多个片组成，一个片由一个或多个宏块（MB）组成，一个宏块由16x16的yuv数据组成。宏块作为H264编码的基本单位。

H264编码过程中的三种不同的数据形式：

SODB：数据比特串 —-＞最原始的编码数据，即VCL数据；

RBSP：原始字节序列载荷 —-＞在SODB的后面填加了结尾比特（RBSP trailing bits　一个bit“1”）若干比特“0”,以便字节对齐；

EBSP：扩展字节序列载荷 —- > 在RBSP基础上填加了仿校验字节（0X03）它的原因是：在NALU加到Annexb上时，需要添加每组NALU之前的开始码StartCodePrefix,如果该NALU对应的slice为一帧的开始则用4位字节表示，ox00000001,否则用3位字节表示ox000001（是一帧的一部分）。另外，为了使NALU主体中不包括与开始码相冲突的，在编码时，每遇到两个字节连续为0，就插入一个字节的0x03。解码时将0x03去掉。也称为脱壳操作。

H.264的主要目标是：1．高的视频压缩比；2．良好的网络亲和性；

为了完成这些目标H264的解决方案是：

1. VCL video coding layer 视频编码层；
2. NAL network abstraction layer 网络提取层；

其实vcl就是编码成sodb成rbsp送给nal层，nal就是进行nalu包装。

H.264/AVC核心技术概览

这个新的标准是由下面几个处理步骤组成的：1）帧间和帧内预测；2）变换（和反变换）；3）量化（和反量化）；4）环路滤波；5）熵编码。

单张的图片流组成了视频，它能分成16X16像素的“宏块”，这种分块方法简化了在视频压缩算法中每个步骤的处理过程。举例来说，从标准清晰度标准视频流解决方案（720X480）中截取的一幅图片被分成1350（45X30）个宏块，然后在宏块的层次进行进一步的处理。

#### 2.1.3 H.264打包

IP网络的RTP打包封装分组打包的规则：

(1)额外开销要少，使MTU尺寸在100～64k字节范围都可以；

(2)不用对分组内的数据解码就可以判别该分组的重要性；

(3)载荷规范应当保证不用解码就可识别由于其他的比特丢失而造成的分组不可解码；

(4)支持将NALU分割成多个RTP分组；

(5)支持将多个NALU汇集在一个RTP分组中。

RTP的头标可以是NALU的头标，并可以实现以上的打包规则。

一个RTP分组里放入一个NALU，将NALU(包括同时作为载荷头标的NALU头)放入RTP的载荷中，设置RTP头标值。为了避免IP层对大分组的再 一次分割，片分组的大小一般都要小于MTU尺寸。由于包传送的路径不同，解码端要重新对片分组排序，RTP包含的次序信息可以用来解决这一问题。

NALU分割

对于预先已经编码的内容，NALU可能大于MTU尺寸的限制。虽然IP层的分割可以使数据块小于64千字节，但无法在应用层实现保护，从而降低了非等重保 护方案的效果。由于UDP数据包小于64千字节，而且一个片的长度对某些应用场合来说太小，所以应用层打包是RTP打包方案的一部分。

新的讨论方案(IETF)应当符合以下特征：

(1)NALU的分块以按RTP次序号升序传输；

(2)能够标记第一个和最后一个NALU分块；

(3)可以检测丢失的分块。

NALU合并

一些NALU如SEI、参数集等非常小，将它们合并在一起有利于减少头标开销。已有两种集合分组：

(1)单一时间集合分组(STAP)，按时间戳进行组合；

(2)多时间集合分组(MTAP)，不同时间戳也可以组合。

#### 2.1.4 H264压缩

H264的基本原理其实非常简单，下我们就简单的描述一下H264压缩数据的过程。通过摄像头采集到的视频帧（按每秒 30 帧算），被送到 H264 编码器的缓冲区中。编码器先要为每一幅图片划分宏块。

以下面这张图为例:



划分宏块

H264默认是使用 16X16 大小的区域作为一个宏块，也可以划分成 8X8 大小。



划分好宏块后，计算宏块的象素值。



以此类推，计算一幅图像中每个宏块的像素值，所有宏块都处理完后如下面的样子。



划分子块

H264对比较平坦的图像使用 16X16 大小的宏块。但为了更高的压缩率，还可以在 16X16 的宏块上更划分出更小的子块。子块的大小可以是 8X16､ 16X8､ 8X8､ 4X8､ 8X4､ 4X4非常的灵活。



上幅图中，红框内的 16X16 宏块中大部分是蓝色背景，而三只鹰的部分图像被划在了该宏块内，为了更好的处理三只鹰的部分图像，H264就在 16X16 的宏块内又划分出了多个子块。

这样再经过帧内压缩，可以得到更高效的数据。下图是分别使用mpeg-2和H264对上面宏块进行压缩后的结果。其中左半部分为MPEG-2子块划分后压缩的结果，右半部分为H264的子块划压缩后的结果，可以看出H264的划分方法更具优势。



宏块划分好后，就可以对H264编码器缓存中的所有图片进行分组了。

帧分组

对于视频数据主要有两类数据冗余，一类是时间上的数据冗余，另一类是空间上的数据冗余。其中时间上的数据冗余是最大的。

对于这些关联特别密切的帧，其实我们只需要保存一帧的数据，其它帧都可以通过这一帧再按某种规则预测出来，所以说视频数据在时间上的冗余是最多的。

为了达到相关帧通过预测的方法来压缩数据，就需要将视频帧进行分组。那么如何判定某些帧关系密切，可以划为一组呢？我们来看一下例子，下面是捕获的一组运动的台球的视频帧，台球从右上角滚到了左下角。

H264编码器会按顺序，每次取出两幅相邻的帧进行宏块比较，计算两帧的相似度。如下图：

通过宏块扫描与宏块搜索可以发现这两个帧的关联度是非常高的。进而发现这一组帧的关联度都是非常高的。因此，上面这几帧就可以划分为一组。其算法是：在相邻几幅图像画面中，一般有差别的像素只有10%以内的点,亮度差值变化不超过2%，而色度差值的变化只有1%以内，我们认为这样的图可以分到一组。

在这样一组帧中，经过编码后，我们只保留第一帖的完整数据，其它帧都通过参考上一帧计算出来。我们称第一帧为IDR／I帧，其它帧我们称为P／B帧，这样编码后的数据帧组我们称为GOP。

运动估计与补偿

在H264编码器中将帧分组后，就要计算帧组内物体的运动矢量了。还以上面运动的台球视频帧为例，我们来看一下它是如何计算运动矢量的。

H264编码器首先按顺序从缓冲区头部取出两帧视频数据，然后进行宏块扫描。当发现其中一幅图片中有物体时，就在另一幅图的邻近位置（搜索窗口中）进行搜索。如果此时在另一幅图中找到该物体，那么就可以计算出物体的运动矢量了。下面这幅图就是搜索后的台球移动的位置。通过上图中台球位置相差，就可以计算出台图运行的方向和距离。H264依次把每一帧中球移动的距离和方向都记录下来就成了下面的样子。我们把运动矢量与补偿称为帧间压缩技术，它解决的是视频帧在时间上的数据冗余。除了帧间压缩，帧内也要进行数据压缩，帧内数据压缩解决的是空间上的数据冗余。下面我们就来介绍一下帧内压缩技术。

帧内预测

人眼对图象都有一个识别度，对低频的亮度很敏感，对高频的亮度不太敏感。所以基于一些研究，可以将一幅图像中人眼不敏感的数据去除掉。这样就提出了帧内预测技术。

H264的帧内压缩与JPEG很相似。一幅图像被划分好宏块后，对每个宏块可以进行 9 种模式的预测。找出与原图最接近的一种预测模式。



下面这幅图是对整幅图中的每个宏块进行预测的过程。



然后，将原始图像与帧内预测后的图像相减得残差值。再将我们之前得到的预测模式信息一起保存起来，这样我们就可以在解码时恢复原图了。经过帧内与帧间的压缩后，虽然数据有大幅减少，但还有优化的空间。

对残差数据做DCT

可以将残差数据做整数离散余弦变换，去掉数据的相关性，进一步压缩数据。如下图所示，左侧为原数据的宏块，右侧为计算出的残差数据的宏块。将残差数据宏块数字化后如下图所示：



将残差数据宏块进行 DCT 转换。去掉相关联的数据后，我们可以看出数据被进一步压缩了。做完 DCT 后，还不够，还要进行 CABAC 进行无损压缩。

CABAC

上面的帧内压缩是属于有损压缩技术。也就是说图像被压缩后，无法完全复原。而CABAC属于无损压缩技术。

无损压缩技术大家最熟悉的可能就是哈夫曼编码了，给高频的词一个短码，给低频词一个长码从而达到数据压缩的目的。MPEG-2中使用的VLC就是这种算法，我们以 A-Z 作为例子，A属于高频数据，Z属于低频数据。看看它是如何做的。

CABAC也是给高频数据短码，给低频数据长码。同时还会根据上下文相关性进行压缩，这种方式又比VLC高效很多。其效果如下：



现在将 A-Z 换成视频帧，它就成了下面的样子。



从上面这张图中明显可以看出采用 CACBA 的无损压缩方案要比 VLC 高效的多。

H264编码的基本原理，包括：

宏块的划分

图像分组

帧内压缩技术原理

帧间压缩技术原理。

DCT

CABAC压缩原理。

#### 2.1.5 H.264序列的说明

在H264中图像以序列为单位进行组织，一个序列是一段图像编码后的数据流，以 I 帧开始，到下一个 I 帧结束。一个序列的第一个图像叫做 IDR 图像（立即刷新图像），IDR 图像都是 I 帧图像。H.264 引入 IDR 图像是为了解码的重同步，当解码器解码到 IDR 图像时，立即将参考帧队列清空，将已解码的数据全部输出或抛弃，重新查找参数集，开始一个新的序列。这样，如果前一个序列出现重大错误，在这里可以获得重新同步的机会。IDR 图像之后的图像永远不会使用 IDR 之前的图像的数据来解码。

I 帧：帧内编码帧 ，I 帧表示关键帧，你可以理解为这一帧画面的完整保留；解码时只需要本帧数据就可以完成（因为包含完整画面）。

P 帧：前向预测编码帧。P 帧表示的是这一帧跟之前的一个关键帧（或P 帧）的差别，解码时需要用之前缓存的画面叠加上本帧定义的差别，生成最终画面。（也就是差别帧，P 帧没有完整画面数据，只有与前一帧的画面差别的数据）。P帧的预测与重构：P 帧是以 I 帧为参考帧,在 I 帧中找出 P 帧“某点”的预测值和运动矢量，取预测差值和运动矢量一起传送。在接收端根据运动矢量从 I 帧中找出 P 帧“某点”的预测值并与差值相加以得到 P 帧“某点”样值,从而可得到完整的 P 帧。P 帧属于前向预测的帧间编码。它只参考前面最靠近它的 I 帧或 P 帧;P 帧可以是其后面 P 帧的参考帧,也可以是其前后的 B 帧的参考帧;由于 P 帧是参考帧,它可能造成解码错误的扩散;由于是差值传送，P 帧的压缩比较高。

B 帧：双向预测内插编码帧。B 帧是双向差别帧，也就是 B 帧记录的是本帧与前后帧的差别（具体比较复杂，有 4 种情况，但我这样说简单些），换言之，要解码 B 帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面的与本帧数据的叠加取得最终的画面。B 帧压缩率高，但是解码时 CPU 会比较累。B 帧的预测与重构：B 帧以前面的 I 或 P 帧和后面的 P 帧为参考帧,“找出”B 帧“某点”的预测值和两个运动矢量,并取预测差值和运动矢量传送。接收端根据运动矢量在两个参考帧中“找出(算出)”预测值并与差值求和,得到 B帧“某点”样值,从而可得到完整的 B 帧。

举例说明：



在如上图中，GOP (Group of Pictures) 长度为 13，S0~S7 表示 8 个视点，T0~T12 为 GOP 的 13 个时刻。每个 GOP 包含帧数为视点数 GOP 长度的乘积。在该图中一个 GOP 中，包含 94 个 B 帧。B 帧占一个 GOP 总帧数的 90.38%。GOP 越长，B 帧所占比例更高，编码的率失真性能越高。下图测试序列 Race1 在不同 GOP 下的率失真性能对比。

#### 2.1.6 H.264 压缩方法

（1）分组：把几帧图像分为一组(GOP，也就是一个序列)，为防止运动变化,帧数不宜取多。

（2）定义帧：将每组内各帧图像定义为三种类型,即 I 帧、B 帧和 P 帧;

（3）预测帧：以I帧做为基础帧，以 I 帧预测 P 帧,再由 I 帧和 P 帧预测 B 帧;

（4）数据传输：最后将 I 帧数据与预测的差值信息进行存储和传输。

#### 2.1.7 H.264 I 帧与 IDR 帧的区别

IDR（Instantaneous Decoding Refresh）--即时解码刷新。

    I 和 IDR 帧都是使用帧内预测的。它们都是同一个东西而已，在编码和解码中为了方便，要首个 I 帧和其他 I 帧区别开，所以才把第一个首个 I 帧叫 IDR，这样就方便控制编码和解码流程。IDR 帧的作用是立刻刷新，使错误不致传播,从IDR帧开始，重新算一个新的序列开始编码。而 I 帧不具有随机访问的能力，这个功能是由 IDR 承担。IDR 会导致DPB（参考帧列表——这是关键所在）清空，而 I 不会。IDR 图像一定是 I 图像，但I图像不一定是 IDR 图像。一个序列中可以有很多的I图像，I 图像之后的图像可以引用 I 图像之间的图像做运动参考。一个序列中可以有很多的 I 图像，I 图像之后的图象可以引用I图像之间的图像做运动参考。

对于 IDR 帧来说，在 IDR 帧之后的所有帧都不能引用任何 IDR 帧之前的帧的内容，与此相反，对于普通的 I 帧来说，位于其之后的 B- 和 P- 帧可以引用位于普通 I- 帧之前的 I- 帧。从随机存取的视频流中，播放器永远可以从一个 IDR 帧播放，因为在它之后没有任何帧引用之前的帧。但是，不能在一个没有 IDR 帧的视频中从任意点开始播放，因为后面的帧总是会引用前面的帧。

举个例子，在一段视频中，存在以下帧：I P B P B P B B P I P B…

如果这段视频应用了多重参照帧，那么蓝色的 P 帧在参照他前面的 I 帧（红色）的同时，还可能会参照 I 帧之前的 P （绿色），由于 I 帧前后的场景可能会有很大的反差甚至根本不同，所以此时 P 帧参考I帧之前的帧不但会没有意义，反而会造成很多问题。所以一种新型的帧被引入，那就是 IDR 帧。如果这段视频应用了多重参考帧的同时采用了 IDR 帧，那么帧的顺序就会变成这样：I P B P B P B B P IDR P B…由于 IDR 帧禁止后面的帧向自己前面的帧参照，所以这回那个蓝色的 P 帧就不会参照绿色的 P 帧了。使用 ffmpeg 获取 I 帧 ：ffmpeg -i wkll.mp4 -ss 00:00:00 -t 1 -r 1 -q:v 2 -f image2 pic-%03d.jpeg

#### 2.1.8 h264基础知识补充

1.YCbCr

在 4:2:0 样点中，两个色度阵列的高度和宽度均为亮度阵列的一半。

在 4:2:2 样点中，两个色度阵列的高度等于亮度阵列的高度，宽度为亮度阵列的一半。

在 4:2:4 样点中，两个色度阵列的高度和宽度与亮度阵列的相等。

两个整列。一个顶场，一个底场。一半意思就是原有两个点=现在一个点。上面三个两个是宽度一半。一帧由两个场组成。

在H.264进行编码的过程中，每一帧的H图像被分为一个或多个条带(slice)进行编码。每一个条带包含多个宏块(MB,Macroblock)。宏块是H.264标准中基本的编码单元，其基本结构包含一个包含16×16个亮度像素块和两个8×8色度像素块，以及其他一些宏块头信息。在对一个宏块进行编码时，每一个宏块会分割成多种不同大小的子块进行预测。帧内预测采用的块大小可能为16×16或者4×4，帧间预测/运动补偿采用的块可能有7种不同的形状：16×16、16×8、8×16、8×8、8×4、4×8和4×4。

为了适应不同的应用场景，H.264也定义了三种不同的档次：

基准档次(Baseline Profile)：主要用于视频会议、可视电话等低延时实时通信领域；支持I条带和P条带，熵编码支持CAVLC算法。

主要档次(Main Profile)：主要用于数字电视广播、数字视频数据存储等；支持视频场编码、B条带双向预测和加权预测，熵编码支持CAVLC和CABAC算法。

扩展档次(Extended Profile)：主要用于网络视频直播与点播等；支持基准档次的所有特性，并支持SI和SP条带，支持数据分割以改进误码性能，支持B条带和加权预测，但不支持CABAC和场编码。

在H.264中使用的编码技术主要有以下类型：

帧内预测

H.264中采用了基于像素块的帧内预测技术。主要可分为以下不同类型：

16×16大小的亮度块：4种预测模式

4×4大小的亮度块：9种预测模式

色度块：4种预测模式，同16×16的亮度块

帧间预测

H.264中的帧间预测方法采用了基于块的运动估计和补偿方法，其主要特点有：

多个候选参考帧；

B帧作为参考帧；

任意参考帧排序；

多种运动补偿像素块形状，包括16×16、16×8、8×16、8×8、8×4、4×8和4×4像素；

1/4像素（亮度）的亚像素插值；

对交错视频的基于帧或场的运动估计；

交错视频编码

针对隔行扫描的视频，H.264专门定义了用于处理此类交错视频的算法。

PicAFF：Picture Adaptive Frame Field——图像层的帧场自适应；

MBAFF：MacroBlock Adaptive Frame Field——宏块层的帧场自适应；

变换和量化编码

H.264的变换编码创新性地采用了类DCT的整数变换，有效降低了运算复杂度。对于基础版H.264，变换矩阵为4×4；在FRExt扩展中，还支持8×8的变换矩阵。H.264的量化算法仍然采用标量量化方法。

无损熵编码算法

H.264标准针对不同的语法元素指定了不同的熵编码算法，主要有：

UVLC(Universal Variable Length Coding)：主要采用指数哥伦布编码；

CAVLC(Context Adaptive Variable Length Coding)：上下文自适应的变长编码；

CABAC(Context Adaptive Binary Arithmetic Coding)：上下文自适应的二进制算数编码；

其他技术

除了上述的核心算法之外，H.264还定义了包括去块环路滤波器、SI/SP帧、码率控制等多种技术。

H264的码流的打包方式有两种,一种为annex-b byte stream format的格式，这个是绝大部分编码器的默认输出格式，就是每个帧的开头的3~4个字节是H264的start\_code,0x00000001或者0x000001。另一种是原始的NAL打包格式，就是开始的若干字节（1，2，4字节）是NAL的长度，而不是start\_code,此时必须借助某个全局的数据来获得编码器的profile,level,PPS,SPS等信息才可以解码。

I片：只包 I宏块，I 宏块利用从当前片中已解码的像素作为参考进行帧内预测(不能取其它片中的已解码像素作为参考进行帧内预测)。

P片：可包 P和I宏块，P 宏块利用前面已编码图象作为参考图象进行帧内预测，一个帧内编码的宏块可进一步作宏块的分割:即 16×16、16×8、8×16 或 8×8 亮度像素块(以及附带的彩色像素);如果选了 8×8 的子宏块，则可再分成各种子宏块的分割，其尺寸为 8×8、8×4、4×8 或 4×4 亮度像素块(以及附带的彩色像素)。

B片：可包 B和I宏块，B 宏块则利用双向的参考图象(当前和 来的已编码图象帧)进行帧内预测。

SP片(切换P)：用于不同编码流之间的切换，包含 P 和/或 I 宏块

SI片：扩展档次中必须具有的切换，它包 了一种特殊类型的编码宏块，叫做 SI 宏块，SI 也是扩展档次中的必备功能。

#### 2.1.9 熵编码

“熵”这一概念原本来自于化学和热力学，用于度量能量退化的指标，即熵越高，物体或系统的做功能力越低。后来香农将这一概念引入到信息论中，用于表示消息的平均信息量。信源的熵通常可以表示信源所发出信息的不确定性，即越是随机的、前后不相关的信息，其熵越高。

熵编码的简单实现——哈夫曼编码，哈夫曼树是一种特殊的二叉树，其终端节点的个数与待编码的码元的个数等同，而且每个终端节点上都带有各自的权值。每个终端节点的路径长度乘以该节点的权值的总和称为整个二叉树的加权路径长度。在满足条件的各种二叉树中，该路径长度最短的二叉树即为哈夫曼树。

哈夫曼编码(Huffman Coding)，是可变字长编码(VLC)的一种。完全依据字符出现概率来构造异字头的平均长度最短的码字。



编码过程最后两个相加算和，然后重新排序。大概重新排n-1次，可看上图。然后最后两个标 0 1值，下面0上面1。计算编码值，找到x4得值，然后找一行最近得得值，那一行没有得值重新一列开始找，找到0.1到0.25到0.6 编码反过来即100。即使相同得也不一样。

编码过程

（1）对图像中出现的不同像素值进行概率统计，得到n个不同概率的信息符号。

（2）按符号出现的概率由大到小、由上到下排列。

（3）对两个最低概率符号分别以0、1赋值。

（4）两最低概率相加后作为一个新符号的概率重新值入符号序列中。

（5）对概率按从大到小重新排列。

（6）重复（1）-（5），直到只剩下两个概率符号的序列。

（7）分别以二进制0、1赋值后，以此为根结点，沿赋值的顺序的逆序依次写出该路径上的二进制代码，得到哈夫曼编码。

注意事项：

①哈夫曼编码方法构造出来的编码不是唯一的。但对于同一信源而言，其平均码长是相同的，其编码效率不同

②哈夫曼编码对不同的信源其编码效率是不同的。只有当信源概率分布很不均匀是，哈夫曼编码才会受到显著效果。

指数哥伦布编码。同上篇介绍的哈夫曼编码一样，指数哥伦布编码同样属于变长编码(VLC)的一种。指数哥伦布编码同哈夫曼编码最显著的一点不同在于，哈弗曼编码构建完成后必须在传递的信息中加入码字和码元值的对应关系，也就是编码的码表，而指数哥伦布编码则不需要。

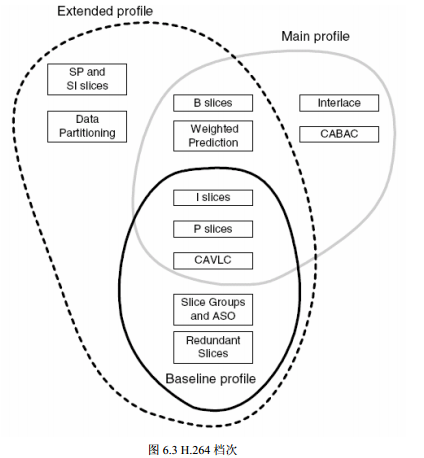
#### 2.1.10 档次和级

H.264 规定了三种档次，每个档次支持一组特定的编码功能，并支持一类特定的应用。

1）基本档次：利用 I 片和 P 片支持帧内和帧间编码，支持利用基于上下文的自适应的变长编码进行的熵编码（CAVLC）。主要用于可视电话、会议电视、无线通信等实时视频通信；

2）主要档次：支持隔行视频，采用 B 片的帧间编码和采用加权预测的帧内编码；支持利用基于上下文的自适应的算术编码（CABAC）。主要用于数字广播电视与数字视频存储；

3）扩展档次：支持码流之间有效的切换（SP 和 SI 片）、改进误码性能（数据分割），但不支持隔行视频和 CABAC。



### 2.2 NALU H264网络传输的结构

H264在网络传输的是NALU，NALU的结构是：NAL头+RBSP

NALU头用来标识后面的RBSP是什么类型的数据，他是否会被其他帧参考以及网络传输是否有错误。

每个NAL前有一个起始码 0x00 00 01（或者0x00 00 00 01），解码器检测每个起始码，作为一个NAL的起始标识，当检测到下一个起始码时，当前NAL结束。同时H.264规定，当检测到0x000000时，也可以表征当前NAL的结束。那么NAL中数据出现0x000001或0x000000时怎么办？H.264引入了防止竞争机制，如果编码器检测到NAL数据存在0x000001或0x000000时，编码器会在最后个字节前插入一个新的字节0x03，这样：

0x000003－>0x00000303。解码器检测到0x000003时，把03抛弃，恢复原始数据（脱壳操作）。解码器在解码时，首先逐个字节读取NAL的数据，统计NAL的长度，然后再开始解码。

#### 2.2.1 NALU头结构

NALU头结构长度：1byte

forbidden\_bit(1bit) + nal\_reference\_bit(2bit) + nal\_unit\_type(5bit)

1.forbidden\_bit：禁止位，初始为0，当网络发现NAL单元有比特错误时可设置该比特为1，以便接收方纠错或丢掉该单元。

2.nal\_reference\_bit：nal重要性指示，标志该NAL单元的重要性，值越大，越重要，解码器在解码处理不过来的时候，可以丢掉重要性为0的NALU。

3.nal\_unit\_type：不同类型的NALU的重要性指示如下表所示。

|  |  |  |
| --- | --- | --- |
| nal\_unit\_type | NAL类型 | nal\_reference\_bit |
| 0 | 未使用 | 0 |
| 1 | 非IDR的片 | 此片属于参考帧，则不等于0，  不属于参考帧，则等与0 |
| 2 | 片数据A分区 | 同上 |
| 3 | 片数据B分区 | 同上 |
| 4 | 片数据C分区 | 同上 |
| 5 | IDR图像的片 | 5 |
| 6 | 补充增强信息单元（SEI） | 0 |
| 7 | 序列参数集sps | 非0 |
| 8 | 图像参数集pps | 非0 |
| 9 | 分界符 | 0 |
| 10 | 序列结束 | 0 |
| 11 | 码流结束 | 0 |
| 12 | 填充 | 0 |
| 13..23 | 保留 | 0 |
| 24..31 | 不保留（RTP打包时会用到） | 0 |

RTP 打包时的扩展类型

|  |  |  |
| --- | --- | --- |
| 24 | STAP-A | Single-time aggregation packet |
| 25 | STAP-B | Single-time aggregation packet |
| 26 | MTAP16 | Multi-time aggregation packet |
| 27 | MTAP24 | Multi-time aggregation packet |
| 28 | FU-A | Fragmentation unit |
| 29 | FU-B | Fragmentation unit |
| 30-31 | undefined |  |

#### 2.2.2 RBSP数据

RBSP数据是下表中的一种，rbsp是根据nalu头类型决定。

|  |  |  |
| --- | --- | --- |
| RBSP类型 | 所写 | 描述 |
| 参数集 | PS | 序列的全局信息，如图像尺寸，视频格式等 |
| 增强信息 | SEI | 视频序列解码的增强信息 |
| 图像界定符 | PD | 视频图像的边界 |
| 编码片 | SLICE | 编码片的头信息和数据 |
| 数据分割 |  | DP片层的数据，用于错误恢复解码 |
| 序列结束符 |  | 表明一个序列的结束，下一个图像为IDR图像 |
| 流结束符 |  | 表明该流中已没有图像 |
| 填充数据 |  | 亚元数据，用于填充字节 |

要完成视频的解码，不仅需要传输VCL层编码出来的视频帧数据，还需要传输序列参数集、图像参数集等数据。

参数集:包括序列参数集 SPS 和图像参数集 PPS

**SPS：**包含的是针对一连续编码视频序列的参数，如标识符 seq\_parameter\_set\_id、帧数及 POC 的约束、参考帧数目、解码图像尺寸和帧场编码模式选择标识等等。

**PPS：**对应的是序列中某一幅图像或者某几幅图像，其参数如标识符pic\_parameter\_set\_id、可选的 seq\_parameter\_set\_id、熵编码模式选择标识、片组数目、初始量化参数和去方块滤波系数调整标识等等。

**SEi:**自定义信息，

数据分割：组成片的编码数据存放在 3 个独立的 DP（数据分割，A、B、C）中，各自包含一个编码片的子集。

分割Ａ包含片头和片中每个宏块头数据。

分割Ｂ包含帧内和 SI 片宏块的编码残差数据。

分割 C包含帧间宏块的编码残差数据。

每个分割可放在独立的 NAL 单元并独立传输。

**AUD**：一般文档没有对AUD进行描叙，其实这是一个帧开始的标志，字节顺序为：00 00 00 01 09 f0从结构上看，有start code, 所以确实是一个NALU，类型09在H264定义里就是AUD（分割器）。大部分播放器可以在没有AUD的情况下正常播放。紧随AUD，一般是SPS/PPS/SEI/IDR的组合或者简单就是一个SLICE，也就是一个帧的开始。像Flash这样的播放器，每次需要一个完整的帧数据，那么把2个AUD之间的数据按照格式打包给播放器就可以了。

h264有两种封装：一种是annexb模式，传统模式，有startcode，SPS和PPS是在ES中；一种是mp4模式，一般mp4 mkv会有，没有startcode，SPS和PPS以及其它信息被封装在container中，每一个frame前面是这个frame的长度。

很多解码器只支持annexb这种模式，因此需要将mp4做转换：

在ffmpeg中用h264\_mp4toannexb\_filter可以做转换

Ffmpeg中实现：

注册filter

avcbsfc = av\_bitstream\_filter\_init(“h264\_mp4toannexb”);

转换bitstream

av\_bitstream\_filter\_filter(AVBitStreamFilterContext \*bsfc,

AVCodecContext \*avctx, const char \*args,

uint8\_t \*\*poutbuf, int \*poutbuf\_size,

const uint8\_t \*buf, int buf\_size, int keyframe)

对于MP4文件，NAL单元之前没有同步码，却有若干字节的长度码，来表示NAL单元的长度，这个长度码所占用的字节数由MP4文件头给出；此外，从MP4读出来的视频帧不包含PPS和SPS，这些信息位于MP4的文件头中，解析器必须在打开文件的时候就获取它们。从MP4文件读出的一个H.264帧往往是下面的形式（假设长度码为2字节）：

00 19 06 [... 25 字节...] 24 aa 65 [... 9386 字节...]

SEI信息 IDR Slice

H264 over RTP基本上分三种类型：

(1)Single NAL unit packet 也就是实际的NAL类型，可以理解为一个包就是一帧H264数据，这个在实际中是比较多的。

(2)Aggregation packet 一包数据中含有多个H264帧。

STAP-A 包内的帧含有相同的NALU-Time，没有DON

STAP-B 包内的帧含有相同的NALU-Time，有DON

MTAP16 包内的帧含有不同的NALU-Time，timestamp offset = 16

MTAP24 包内的帧含有不同的NALU-Time，timestamp offset = 24

封装在Aggregation packet中的 NAL单元大小为65535字节

1. Fragmentation unit 一帧数据被分为多个RTP包，这也是很常见的，特别是对于关键帧。现存两个版本FU-A，FU-B。

h264包在传输的时候，如果包太大，会被分成多个片。NALU头会被如下的2个自己代替。

The FU indicator octet has the following format:

      +---------------+

      |0|1|2|3|4|5|6|7|

      +-+-+-+-+-+-+-+-+

      |F|NRI|  Type   |

      +---------------+

   别被名字吓到这个格式就是上面提到的RTP h264负载类型，Type为FU-A

下面这个应该就是rtp中h264格式The FU header has the following format:

      +---------------+

      |0|1|2|3|4|5|6|7|

      +-+-+-+-+-+-+-+-+

      |S|E|R|  Type   |

      +---------------+

   S bit为1表示分片的NAL开始，当它为1时，E不能为1

   E bit为1表示结束，当它为1，S不能为1

   R bit保留位

   Type就是NALU头中的Type,取1-23的那个值

#### 2.2.3 SPS语法元素及其含义

SPS即Sequence Paramater Set，又称作序列参数集。类型7表示该NAL Unit内保存的数据为Sequence ParamaterSet。丢失或出现错误，那么解码过程很可能会失败。SPS及后续将要讲述的图像参数集PPS在某些平台的视频处理框架（比如iOS的VideoToolBox等）还通常作为解码器实例的初始化信息使用。SPS中保存了一组编码视频序列(Coded

videosequence)的全局参数。所谓的编码视频序列即原始视频的一帧一帧的像素数据经过编码之后的结构组成的序列。而每一帧的编码后数据所依赖的参数保存于图像参数集中。一般情况SPS和PPS的NALUnit通常位于整个码流的起始位置。但在某些特殊情况下，在码流中间也可能出现这两种结构，主要原因可能为：

解码器需要在码流中间开始解码；

编码器在编码的过程中改变了码流的参数（如图像分辨率等）；

在做视频播放器时，为了让后续的解码过程可以使用SPS中包含的参数，必须对其中的数据进行解析。其中H.264标准协议中规定的SPS格式位于文档的7.3.2.1.1部分，如下图所示：



其中的每一个语法元素及其含义如下：

**(1) profile\_idc**：

标识当前H.264码流的profile。我们知道，H.264中定义了三种常用的档次profile：

基准档次：baseline profile;

主要档次：main profile;

扩展档次：extended profile;

在H.264的SPS中，第一个字节表示profile\_idc，根据profile\_idc的值可以确定码流符合哪一种档次。判断规律为：

profile\_idc = 66 → baseline profile;

profile\_idc = 77 → main profile;

profile\_idc = 88 → extended profile;

在新版的标准中，还包括了High、High 10、High 4:2:2、High 4:4:4、High 10 Intra、High

4:2:2 Intra、High 4:4:4 Intra、CAVLC 4:4:4 Intra等，每一种都由不同的profile\_idc表示。

另外，constraint\_set0\_flag ~ constraint\_set5\_flag是在编码的档次方面对码流增加的其他一些额外限制性条件。

在我们实验码流中，profile\_idc = 0x42 = 66，因此码流的档次为baseline profile。

**(2) level\_idc**

标识当前码流的Level。编码的Level定义了某种条件下的最大视频分辨率、最大视频帧率等参数，码流所遵从的level由level\_idc指定。

当前码流中，level\_idc = 0x1e = 30，因此码流的级别为3。

**(3) seq\_parameter\_set\_id**

表示当前的序列参数集的id。通过该id值，图像参数集pps可以引用其代表的sps中的参数。本句法元素的值应该在[0，31]。

注意：当编码器需要产生新的序列参数集时，应该使用新的 seq\_parameter\_set\_id,即使用新的序列参数集，而不是去改变原来的参数集中的内容

**(4) log2\_max\_frame\_num\_minus4**

用于计算MaxFrameNum的值。计算公式为MaxFrameNum = 2^(log2\_max\_frame\_num\_minus4 +4)。MaxFrameNum是frame\_num的上限值，frame\_num是图像序号的一种表示方法，在帧间编码中常用作一种参考帧标记的手段。

这个句法元素主要是为读取另一个句法元素 frame\_num 服务的，frame\_num 是最重要的句法元素之一，它标识所属图像的解码顺序。可以在句法表看到，fram-num的解码函数是 ue（v），函数中的 v 在这里指定：

v = log2\_max\_frame\_num\_minus4 + 4

从另一个角度看，这个句法元素同时也指明了 frame\_num 的所能达到的最大值：

MaxFrameNum = 2( log2\_max\_frame\_num\_minus4 + 4 )

变量 MaxFrameNum 表示 frame\_num 的最大值，在后文中可以看到，在解码过程中它也是一个非常重要的变量。

值得注意的是 frame\_num 是循环计数的，即当它到达 MaxFrameNum 后又从 0 重新开始新一轮的计数。解码器必须要有机制检测这种循环，不然会引起类似千年虫的问题，在图像的顺序上造成混乱。在第八章会详细讲述 H.264 检测这种循环的机制。

**(5) pic\_order\_cnt\_type**

表示解码picture order count(POC)的方法。POC是另一种计量图像序号的方式，与frame\_num有着不同的计算方法。该语法元素的取值为0、1或2。

指明了 poc (picture order count) 的编码方法，poc 标识图像的播放顺序。由于H.264 使用了 B 帧预测，使得图像的解码顺序并不一定等于播放顺序，但它们之间存在一定的映射关系。poc 可以由 frame-num 通过映射关系计算得来，也可以索性由编码器显式地传送。H.264 中一共定义了三种 poc 的编码方法，这个句法元素就是用来通知解码器该用哪种方法来计算 poc。而以下的几个句法元素是分别在各种方法中用到的数据。

在如下的视频序列中本句法元素不应该等于 2:

一个非参考帧的接入单元后面紧跟着一个非参考图像(指参考帧或参考场)的接入单元

两个分别包含互补非参考场对的接入单元后面紧跟着一个非参考图像的接入单元.

一个非参考场的接入单元后面紧跟着另外一个非参考场,并且这两个场不能构成一个互补场对

**(6) log2\_max\_pic\_order\_cnt\_lsb\_minus4**

用于计算MaxPicOrderCntLsb的值，该值表示POC的上限。计算方法为MaxPicOrderCntLsb = 2^(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4)。

**(7) max\_num\_ref\_frames**

用于表示参考帧的最大数目。解码器依照这个句法元素的值开辟存储区，这个存储区用于存放已解码的参考帧，H.264 规定最多可用 16 个参考帧，本句法元素的值最大为 16。值得注意的是这个长度以帧为单位，如果在场模式下，应该相应地扩展一倍。

**(8) gaps\_in\_frame\_num\_value\_allowed\_flag**

标识位，说明frame\_num中是否允许不连续的值。这个句法元素等于 1 时，表示允许句法元素 frame\_num 可以不连续。当传输信道堵塞严重时，编码器来不及将编码后的图像全部发出，这时允许丢弃若干帧图像。在正常情况下每一帧图像都有依次连续的frame\_num 值，解码器检查到如果 frame\_num 不连续，便能确定有图像被编码器丢弃。这时，解码器必须启动错误掩藏的机制来近似地恢复这些图像，因为这些图像有可能被后续图像用作参考帧。

当这个句法元素等于 0 时，表不允许 frame\_num 不连续，即编码器在任何情况下都不能丢弃图像。这时，H.264 允许解码器可以不去检查 frame\_num 的连续性以减少计算量。这种情况下如果依然发生 frame\_num 不连续，表示在传输中发生丢包，解码器会通过其他机制检测到丢包的发生，然后启动错误掩藏的恢复图像。

**(9) pic\_width\_in\_mbs\_minus1**

用于计算图像的宽度。单位为宏块个数，因此图像的实际宽度为:

frame\_width = 16 × (pic\_width\_in\_mbs\_minus1 + 1);

本句法元素加 1 后指明图像宽度，以宏块为单位：

PicWidthInMbs = pic\_width\_in\_mbs\_minus1 + 1

通过这个句法元素解码器可以计算得到亮度分量以像素为单位的图像宽度：

PicWidthInSamplesL = PicWidthInMbs \* 16

从而也可以得到色度分量以像素为单位的图像宽度：

PicWidthInSamplesC = PicWidthInMbs \* 8

以上变量 PicWidthInSamplesL、PicWidthInSamplesC 分别表示图像的亮度、色度分量以像素为单位的宽。

H.264 将图像的大小在序列参数集中定义，意味着可以在通信过程中随着序列参数集动态地改变图像的大小，在后文中可以看到，甚至可以将传送的图像剪裁后输出。

**(10) pic\_height\_in\_map\_units\_minus1**

使用PicHeightInMapUnits来度量视频中一帧图像的高度。PicHeightInMapUnits并非图像明确的以像素或宏块为单位的高度，而需要考虑该宏块是帧编码或场编码。PicHeightInMapUnits的计算方式为：

PicHeightInMapUnits = pic\_height\_in\_map\_units\_minus1 + 1;

本句法元素加 1 后指明图像高度：

PicHeightInMapUnits = pic\_height\_in\_map\_units\_minus1 + 1

PicSizeInMapUnits = PicWidthInMbs \* PicHeightInMapUnits

图像的高度的计算要比宽度的计算复杂，因为一个图像可以是帧也可以是场，从这个句法元素可以在帧模式和场模式下分别计算出出亮度、色度的高。值得注意的是，这里以 map\_unit 为单位，map\_unit的含义由后文叙述。

**(11) frame\_mbs\_only\_flag**

标识位，说明宏块的编码方式。当该标识位为0时，宏块可能为帧编码或场编码；该标识位为1时，所有宏块都采用帧编码。根据该标识位取值不同，PicHeightInMapUnits的含义也不同，为0时表示一场数据按宏块计算的高度，为1时表示一帧数据按宏块计算的高度。

按照宏块计算的图像实际高度FrameHeightInMbs的计算方法为：

FrameHeightInMbs = ( 2 − frame\_mbs\_only\_flag ) \* PicHeightInMapUnits

结合 map\_unit 的含义，这里给出上一个句法元素 pic\_height\_in\_map\_units\_minus1的进一步解析步骤：

当 frame\_mbs\_only\_flag 等于１，pic\_height\_in\_map\_units\_minus1 指的是一个 picture 中帧的高度；

当frame\_mbs\_only\_flag 等于０，pic\_heght\_in\_map\_units\_minus1 指的是一个 picture 中场的高度，所以可以得到如下以宏块为单位的图像高度：

FrameHeightInMbs = ( 2 – frame\_mbs\_only\_flag ) \* PicHeightInMapUnits

PictureHeightInMbs= ( 2 – frame\_mbs\_only\_flag ) \* PicHeightInMapUnits

**(12) mb\_adaptive\_frame\_field\_flag**

标识位，说明是否采用了宏块级的帧场自适应编码。当该标识位为0时，不存在帧编码和场编码之间的切换；当标识位为1时，宏块可能在帧编码和场编码模式之间进行选择。

指明本序列是否属于帧场自适应模式。mb\_adaptive\_frame\_field\_flag等于１时表明在本序列中的图像如果不是场模式就是帧场自适应模式，等于０时表示本序列中的图像如果不是场模式就是帧模式。。表 列举了一个序列中可能出现的编码模式：

a. 全部是帧，对应于 frame\_mbs\_only\_flag =1 的情况。

b. 帧和场共存。frame\_mbs\_only\_flag =0, mb\_adaptive\_frame\_field\_flag =0

c. 帧场自适应和场共存。frame\_mbs\_only\_flag =0, mb\_adaptive\_frame\_field\_flag =1

值得注意的是，帧和帧场自适应不能共存在一个序列中。

**(13) direct\_8x8\_inference\_flag**

标识位，用于B\_Skip、B\_Direct模式运动矢量的推导计算。

**(14) frame\_cropping\_flag**

标识位，说明是否需要对输出的图像帧进行裁剪。用于指明解码器是否要将图像裁剪后输出，如果是的话，后面紧跟着的四个句法元素分别指出左右、上下裁剪的宽度。

**(15) vui\_parameters\_present\_flag**

标识位，说明SPS中是否存在VUI信息。指明 vui 子结构是否出现在码流中，vui 的码流结构在附录中指明，用以表征视频格式等额外信息。

#### 2.2.4 PPS语法元素及其含义

除了序列参数集SPS之外，H.264中另一重要的参数集合为图像参数集Picture Paramater

Set(PPS)。通常情况下，PPS类似于SPS，在H.264的裸码流中单独保存在一个NAL Unit中，只是PPS NALUnit的nal\_unit\_type值为8；而在封装格式中，PPS通常与SPS一起，保存在视频文件的文件头中。在H.264的协议文档中，PPS的结构定义在7.3.2.2节中，具体的结构如下表所示：



其中的每一个语法元素及其含义如下：

**(1) pic\_parameter\_set\_id**

表示当前PPS的id。某个PPS在码流中会被相应的slice引用，slice引用PPS的方式就是在Slice header中保存PPS的id值。该值的取值范围为[0,255]。

**(2) seq\_parameter\_set\_id**

表示当前PPS所引用的激活的SPS的id。通过这种方式，PPS中也可以取到对应SPS中的参数。该值的取值范围为[0,31]。Pps引用sps的值。

**(3) entropy\_coding\_mode\_flag**

熵编码模式标识，该标识位表示码流中熵编码/解码选择的算法。对于部分语法元素，在不同的编码配置下，选择的熵编码方式不同。例如在一个宏块语法元素中，宏块类型mb\_type的语法元素描述符为“ue(v)| ae(v)”，在baseline profile等设置下采用指数哥伦布编码，在main profile等设置下采用CABAC编码。

标识位entropy\_coding\_mode\_flag的作用就是控制这种算法选择。当该值为0时，选择左边的算法，通常为指数哥伦布编码或者CAVLC；当该值为1时，选择右边的算法，通常为CABAC。

**(4) bottom\_field\_pic\_order\_in\_frame\_present\_flag**

标识位，用于表示另外条带头中的两个语法元素delta\_pic\_order\_cnt\_bottom和delta\_pic\_order\_cn是否存在的标识。这两个语法元素表示了某一帧的底场的POC的计算方法。

**(5) num\_slice\_groups\_minus1**

表示某一帧中slice group的个数。当该值为0时，一帧中所有的slice都属于一个slice group。slice group是一帧中宏块的组合方式，定义在协议文档的3.141部分。

**(6) num\_ref\_idx\_l0\_default\_active\_minus1、num\_ref\_idx\_l0\_default\_active\_minus1**

表示当Slice Header中的num\_ref\_idx\_active\_override\_flag标识位为0时，P/SP/B

slice的语法元素num\_ref\_idx\_l0\_active\_minus1和num\_ref\_idx\_l1\_active\_minus1的默认值。

**(7) weighted\_pred\_flag**

标识位，表示在P/SP slice中是否开启加权预测。

**(8) weighted\_bipred\_idc**

表示在B Slice中加权预测的方法，取值范围为[0,2]。0表示默认加权预测，1表示显式加权预测，2表示隐式加权预测。

**(9) pic\_init\_qp\_minus26和pic\_init\_qs\_minus26**

表示初始的量化参数。实际的量化参数由该参数、slice header中的slice\_qp\_delta/slice\_qs\_delta计算得到。

**(10) chroma\_qp\_index\_offset**

用于计算色度分量的量化参数，取值范围为[-12,12]。

**(11) deblocking\_filter\_control\_present\_flag**

标识位，用于表示Slice header中是否存在用于去块滤波器控制的信息。当该标志位为1时，slice header中包含去块滤波相应的信息；当该标识位为0时，slice header中没有相应的信息。

**(12) constrained\_intra\_pred\_flag**

若该标识为1，表示I宏块在进行帧内预测时只能使用来自I和SI类型宏块的信息；若该标识位0，表示I宏块可以使用来自Inter类型宏块的信息。

**(13) redundant\_pic\_cnt\_present\_flag**

标识位，用于表示Slice header中是否存在redundant\_pic\_cnt语法元素。当该标志位为1时，slice header中包含redundant\_pic\_cnt；当该标识位为0时，slice header中没有相应的信息。

#### 2.2.5 SEI 语法元素及其含义

"0x06"SEI type后一个字节为“0x05”（淡黄底色）是SEI payload type，即表征SEI payload分析遵循user\_data\_unregistered()语法。





**SEI payload size**

“0x05”后一个字节为“0x2F”（淡蓝底色）是SEI payload size，此时整个payload是47个字节。SEI payload uuid"0x2F"随后的16个字节即为uuid，此时uuid为：dc45e9bde6d948b7962cd820d923eeef

**SEI payload content**

由于payload size是47个字节，除去16字节的uuid，剩下31个字节的content。由于content是字符串，所以有结束符"0x00"，有效的30个字符内容是：

Zencoder Video Encoding System

**rbsp trailing bits**

47个payload字节后的"0x80"（灰底色）即是rbsp trailing bits，在user\_data\_unregistered()里面都是按字节写入的，所以此时的NAL unit结尾写入的字节一定是0x80。



补充增强信息（Supplemental Enhancement Information）是码流范畴里面的概念，提供了向视频码流中加入信息的办法，是H.264/H.265 视频压缩标准的特性之一。SEI 有基本的特征：并不是解码过程的必须项；有可能对解码过程（容错、纠错）有帮助；集成在视频码流中；

在视频内容的生成端、传输过程中，都可以插入SEI 信息。插入的信息，和其他视频内容一起经过传输链路到达了消费端。那么在SEI 中可以添加哪些信息呢？这里举几个例子，用户场景可以任意扩展：

传递编码器参数；

传递视频版权信息;

传递摄像头参数；

传递内容生成过程中的剪辑事件（引发场景切换）；

《ISO/IEC 14496-10:2014》Annex D.1.1提供了最大到181的payload类型处理规范，由于类型可以指定任意大小，给SEI的添加、处理创造了很大的自由空间。Ffmpeg命令行：./ffmpeg -I oceans.h264 -c:v copy -bsf:v h264\_metadata=sei\_user\_data='086f3693-b7b3-4f2c-9653-21492feee5b8+hello' oceans.sei.h264

Ffmpeg函数实现：libavcodec/h264\_metadata\_bsf.c。

libx264支持多种SEI类型数据写入，常用的仍然是SEI\_USER\_DATA\_UNREGISTERED，具体的写入函数x264\_sei\_version\_write()位于libx264/encoder/set.c中。

#### 2.2.6解析SDP中包含的H.264的SPS和PPS

用RTP传输H264的时候,需要用到sdp协议描述,其中有两项:Sequence Parameter Sets (SPS)和Picture Parameter Set(PPS)需要用到,那么这两项从哪里获取呢?答案是从H264码流中获取.在H264码流中,都是以"0x00 0x00 0x01"或者"0x000x00 0x00 0x01"为开始码的,找到开始码之后,使用开始码之后的第一个字节的低5位判断是否为7(sps)或者8(pps),

及data[4] & 0x1f == 7 ||data[4] & 0x1f ==8.然后对获取的nal去掉开始码之后进行base64编码,得到的信息就可以用于sdp.sps和pps需要用逗号分隔开来.SDP中的H.264的SPS和PPS串，包含了初始化H.264解码器所需要的信息参数，包括编码所用的profile，level，图像的宽和高，deblock滤波器等。

这里需要特别提一下这两个参数pic\_width\_in\_mbs\_minus1 = 21，pic\_height\_in\_mbs\_minus1 = 17分别表示图像的宽和高，以宏块（16x16）为单位的值减1

因此，实际的宽为 (21+1)\*16 = 352 高为 (17+1)\*16 = 288

#### 2.2.7 RTP打包H.264

H264的NALU打包成RTP包的模式（下面是用到的两种模式）

（1）、一个NALU打包成一个RTP包，只需要在一个12字节的RTP包头后添加去掉开始码的NALU即可 （这种模式在一个NALU的大小小于MTU时使用）。

（2）、一个NALU打包成几个RTP包（FU\_A模式），在12个字节的RTP头后面加上一个字节的 FU indicator和一个字节的FU header。FU indicator前3位是NALU头的前3位，后5位是28（十进制）， FU header第1位标记RTP包是否为NALU的第一片，第2位标记RTP包是否为NALU的最后一片。第3位是保 留位，后5位是NALU头的type位。

可能的结构类型分别有:

1. 单一 NAL 单元模式

即一个 RTP 包仅由一个完整的 NALU 组成. 这种情况下 RTP NAL 头类型字段和原始的 H.264的 NALU 头类型字段是一样的. 对于 NALU 的长度小于 MTU 大小的包, 一般采用单一 NAL 单元模式.

对于一个原始的 H.264 NALU 单元常由 [Start Code] [NALU Header] [NALU Payload] 三部分组成, 其中 Start Code 用于标示这是一个 NALU 单元的开始, 必须是 “00 00 00 01” 或 “00 00 01”, NALU 头仅一个字节, 其后都是 NALU 单元内容.

打包时去除 “00 00 01” 或 “00 00 00 01” 的开始码, 把其他数据封包的 RTP 包即可.

如有一个 H.264 的 NALU 是这样的:

[00 00 00 01 67 42 A0 1E 23 56 0E 2F … ]

这是一个序列参数集 NAL 单元. [00 00 00 01] 是四个字节的开始码, 67 是 NALU 头, 42 开始的数据是 NALU 内容.

封装成 RTP 包将如下:

[ RTP Header ] [ 67 42 A0 1E 23 56 0E 2F ]

即只要去掉 4 个字节的开始码就可以了.

2. 组合封包模式

即可能是由多个 NAL 单元组成一个 RTP 包. 分别有4种组合方式: STAP-A, STAP-B, MTAP16, MTAP24. 那么这里的类型值分别是 24, 25, 26 以及 27.

其次, 当 NALU 的长度特别小时, 可以把几个 NALU 单元封在一个 RTP 包中.

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| RTP Header |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

|STAP-A NAL HDR | NALU 1 Size | NALU 1 HDR |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| NALU 1 Data |

: :

+ +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| | NALU 2 Size | NALU 2 HDR |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| NALU 2 Data |

: :

| +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| :...OPTIONAL RTP padding |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

3. 分片封包模式

用于把一个 NALU 单元封装成多个 RTP 包. 存在两种类型 FU-A 和 FU-B. 类型值分别是 28 和 29. 而当 NALU 的长度超过 MTU 时, 就必须对 NALU 单元进行分片封包. 也称为 Fragmentation Units (FUs).

### 2.3 H.264码流结构图

在H.264 中，句法元素共被组织成 序列、图像、片、宏块、子宏块五个层次。H.264中出现的一些概念从大到小排序依次是：序列、图像、片组、片、NALU、宏块、亚宏块、块、像素。





在这样的结构中，每一层的头部和它的数据部分形成管理与被管理的强依赖关系，头部的句法元素是该层数据的核心，而一旦头部丢失，数据部分的信息几乎不可能再被正确解码出来，尤其在序列层及图像层。



在 H.264 中，分层结构最大的不同是取消了序列层和图像层，并将原本属于序列和图像头部的大部分句法元素游离出来形成序列和图像两级参数集，其余的部分则放入片层。

参数集是一个独立的数据单位，不依赖于参数集外的其他句法元素。一个参数集不对应某一个特定的图像或序列，同一序列参数集可以被多个图像参数集引用，同理，同一个图像参数集也可以被多个图像引用。只在编码器认为需要更新参数集的内容时，才会发出新的参数集。





F

图像被划分为条带。条带由一系列的宏块组成，当使用宏块自适应帧/场解码时则由一系列宏块对组成，每个宏块对包含2个宏块。。每个宏块均包含一个 16×16 的亮度阵列，当视频格式不是单色时，还包含和两个相应的色度阵列。如果没有使用宏块自适应帧/场解码，每个宏块代表图像中的一个空间矩形区域。



宏块类型：https://www.cnblogs.com/ranson7zop/p/7603766.html



slice\_type 表示条带的编码类型



当 nal\_unit\_type 等于 5（IDR 图像）时， slice\_type 应等于 2、 4、 7 或 9。

当 num\_ref\_frames 等于 0 时， slice\_type 应等于 2、 4、 7 或 9。

field\_pic\_flag 等于 1 表示该条带是一个编码场的条带。field\_pic\_flag 等于 0 表示该条带是一个编码帧的条 带。当 field\_pic\_flag 不存在时，应推定其值为 0。

解码过程的概述如下：

— 8.1 中定义 NAL 单元的解码过程。

— 8.2 节定义了使用条带层及条带层以上各层的语法元素进行解码的过程。

— 与图像顺序排列有关的变量和函数在8.2.1中推导。(仅对图像中的某条带解码时才需要调用)。

— 与宏块到条带组映射有关的变量和函数在8.2.2中推导。（仅对图像中的某条带解码时才需要调用）。

— 当使用条带数据分割时，组合多个条带数据分割的方法在8.2.3定义。

— 在当前图像的frame\_num不等于PrevRefFrameNum，且不等于( PrevRefFrameNum + 1 ) % MaxFrameNum

时，frame\_num上的空隙的解码过程应在解码图像中的所有条带之前，按照8.2.5.2的规定执行。

— 在 P、 SP 或B 条带的解码开始阶段，需要执行8.2.4节定义的参考图像列表重建过程，从而可以得到参

考图像列表0 (RefPicList0)，当解码B条带时，得到参考图像列表1（RefPicList1）。

— 当当前图像是参考图像，且在所有本图像中的条带都已解码后，8.2.5中的已解码图像的标记过程描述

当前图像如何在后续图像解码过程的帧间预测过程中使用。

— 8.3，8.4，8.5，8.6 和 8.7 定义使用宏块层及宏块层以上层的语法元素进行解码的过程。

— 除8.3节所定义的I\_PCM宏块以外，I 和SI宏块的帧内预测过程都以帧内预测得到的样点作为输出。8.3节直接定义了I\_PCM宏块的重建过程。输出的结果是在去块效应滤波过程之前的重建样点。

— P和B宏块的帧间预测过程在8.4中定义，帧间预测结果作为输出。

— 去块效应滤波前变换系数的解码过程和图像重建过程在8.5中定义。此过程产生I和B宏块以及P条带中 的P宏块样点。输出结果为去块效应滤波前的重建样点。

— 8.6定义了SP条带和SI宏块中的P宏块的解码过程。该过程产生SP、SI宏块中的P宏块的样点值。输出结 果是去块效应滤波之前的重建样点。

— 8.7定义了对靠近块和宏块边缘的重建图像进行的去块效应滤波的过程，输出是已解码样点。

1帧（一幅图像） = 1~N个片（slice） //也可以说1到多个片为一个片组

1个片 = 1~N个宏块（Marcroblock）

1个宏块 = 8\*16 或者16X16的等YUV数据（原始视频采集数据）

#### 参考链接：

1.H264格式 详细介绍

https://blog.csdn.net/Frakie\_Kwok/article/details/77989024

2 .介绍sps pps

https://blog.csdn.net/sdsszk/article/details/83509325

3.H264基本原理

<https://blog.csdn.net/garrylea/article/details/78536775>

### 2.4 H.264 slice格式

参考链接：https://www.jianshu.com/p/19fa110c2383

#### 2.4.1 slice header解析

Slice header中主要保存了当前slice的一些全局的信息，slice body中的宏块在进行解码时需依赖这些信息。其中比较常见的一些语法元素有：

· first\_mb\_in\_slice: 当前slice中包含的第一个宏块在整帧中的位置；

· slice\_type：当前slice的类型；

slice\_type Name of slice\_type 0 P (P slice) 1 B (B slice) 2 I (I slice) 3 SP (SP slice) 4 SI (SI slice) 5 P (P slice) 6 B (B slice) 7 I (I slice) 8 SP (SP slice) 9 SI (SI slice)

IDR 图像时, slice\_type 等于 2, 4, 7, 9。

· pic\_parameter\_set\_id：当前slice所依赖的pps的id；范围 0 到 255。

· colour\_plane\_id:当标识位separate\_colour\_plane\_flag为true时，colour\_plane\_id表示当前的颜色分量，0、1、2分别表示Y、U、V分量。

· frame\_num:表示当前帧序号的一种计量方式。

· field\_pic\_flag:场编码标识位。当该标识位为1时表示当前slice按照场进行编码；该标识位为0时表示当前\* slice按照帧进行编码。

· bottom\_field\_flag:底场标识位。该标志位为1表示当前slice是某一帧的底场；为0表示当前slice为某一帧的顶场。

· idr\_pic\_id:表示IDR帧的序号。某一个IDR帧所属的所有slice，其idr\_pic\_id应保持一致。该值的取值范围为[0,65535]。

· pic\_order\_cnt\_lsb:表示当前帧序号的另一种计量方式。

· delta\_pic\_order\_cnt\_bottom:表示顶场与底场POC差值的计算方法，不存在则默认为0；

· slice\_qp\_delta：用于计算当前slice内所使用的初始qp值。

#### 2.4.2 slice body解析

从文档中我们可以看出，Slice\_data结构中独立的语法元素并不多，主要只有以下几个：

· cabac\_alignment\_one\_bit：表示如果码流启用了CABAC算法，那么码流在这里必须使用若干个比特1实现字节对齐。

· mb\_skip\_run和mb\_skip\_flag：这两个语法元素都用于表示宏块结构是否可以被跳过。“跳过”的宏块指的是，在帧间预测的slice中，当图像区域平坦时，码流中跳过这个宏块的所有数据，不进行传输，只通过这两个语法元素进行标记。在解码端，跳过的宏块通过周围已经重建的宏块来进行恢复。mb\_skip\_run用于熵编码使用CAVLC时，用一个语法元素表示连续跳过的宏块的个数；mb\_skip\_flag用于熵编码使用CABAC时，表示每一个宏块是否被跳过。

· mb\_field\_decoding\_flag：标识位，用于在帧场自适应的码流中标识某个宏块是帧模式还是场模式。

· end\_of\_slice\_flag：在CABAC模式下的一个标识位，表示是否到了slice的末尾。

一个Slice的body部分主要是一个个的宏块结构Macroblock组成，此外还存在一些辅助的信息。

从文档中我们可以看出，Slice\_data结构中独立的语法元素并不多，主要只有以下几个：

1.cabac\_alignment\_one\_bit：表示如果码流启用了CABAC算法，那么码流在这里必须使用若干个比特1实现字节对齐。

2.mb\_skip\_run和mb\_skip\_flag：这两个语法元素都用于表示宏块结构是否可以被跳过。“跳过”的宏块指的是，在帧间预测的slice中，当图像区域平坦时，码流中跳过这个宏块的所有数据，不进行传输，只通过这两个语法元素进行标记。在解码端，跳过的宏块通过周围已经重建的宏块来进行恢复。mb\_skip\_run用于熵编码使用CAVLC时，用一个语法元素表示连续跳过的宏块的个数；mb\_skip\_flag用于熵编码使用CABAC时，表示每一个宏块是否被跳过。

3.mb\_field\_decoding\_flag：标识位，用于在帧场自适应的码流中标识某个宏块是帧模式还是场模式。

4.end\_of\_slice\_flag：在CABAC模式下的一个标识位，表示是否到了slice的末尾。

上述的几个语法元素毫无疑问仅仅占用了全部数据很少的一部分，其他大部分的数据都包含在宏块结构中，即macroblock\_layer()结构。

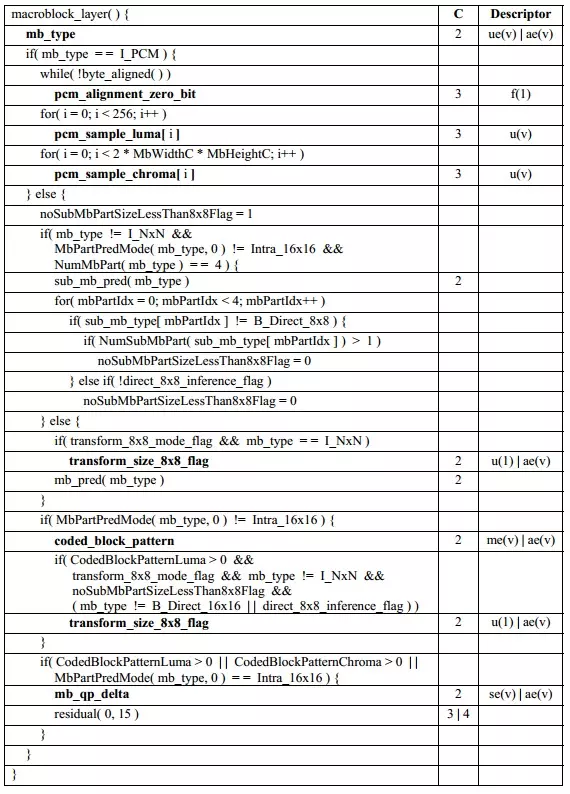
参考链接：<https://blog.csdn.net/liu0808/article/details/72453650>

#### 2.4.3 slice 宏块(Macroblock)结构解析

链接：https://www.jianshu.com/p/c38506f50eb7

宏块(Macroblock)结构

从上表中我们可以看出，一个Slice结构中宏块实际上占据了绝大部分。在标准中一个宏块的结构定义为下表：



(1). mb\_type:

在一个宏块中，最开始的语法元素为宏块的类型：mb\_type。从表中我们可以看出，根据mb\_type的值是否等于I\_PCM，整个解析方法分为两大类：PCM类型和非PCM类型，判断依据是当mb\_type为25时为I\_PCM模式，否则为非I\_PCM模式。

当这个宏块为I\_PCM模式时，宏块中以差分编码的形式保存宏块原始的像素值。此时存在如下几个语法元素：

pcm\_alignment\_zero\_bit：填充位，用比特0来填充直到按字节对齐；

pcm\_sample\_luma：256个亮度分量的差分像素值；

pcm\_sample\_chroma：若干个色度分量的差分像素值，实际数量由码流的颜色格式指定。例 如对于最常用的4:2:0格式的视频，共有128个色度像素值。除了mb\_type等于25时可以确定为I\_PCM格式之外，其他的mb\_type值可能根据帧类型（或slice类型）的不同而不同。比如对于I slice，mb\_type的非PCM模式可以选择0~24这些值之一；对于P slice，mb\_type只能取0~4这5个值；对于B slice，mb\_type可以取0~22这些值之一。目前我们所处理的码流全部由I帧构成，因此我们暂时只考虑I slice的情况。下图是标准文档中规定的I slice的mb\_type列表的一部分，完整列表在协议文档的中：

从上表中我们可以看出，mb\_type不仅仅表示了宏块的分割方式，还包含了一些其他的附加信息，如帧内预测模式、亮度和色度分量的coded\_block\_pattern。当帧内预测使用16×16模式时，宏块整个宏块的预测信息相同，因此不需要为各个子宏块分别指定预测模式，这样可以有效减少消耗的码流。

(2). transform\_size\_8x8\_flag

该语法元素为一个标识位，用于表示在环路滤波之前，预测残差的变换系数解码时依照的尺寸。当该标识位为1时，预测残差按照8×8像素块进行解码；当该标志位不存在或者为0时，预测残差按照4×4像素块进行解码。

(3). coded\_block\_pattern

coded\_block\_pattern语法元素常简称做cbp，用于表示当前宏块内的4个8×8子块编码对其中的哪个的残差系数进行编码。值得注意的是该语法元素仅仅在宏块为非I\_16x16模式时才存在，因为在I\_16x16模式时cbp的有关信息已经在mb\_type中体现。

(4). mb\_qp\_delta

mb\_qp\_delta表示宏块层的量化参数偏移值，取值范围为[-26, 25]。我们在前面已经在PPS中获取了整个序列的量化参数初始值（由pic\_init\_qp\_minus26计算），在slice header中获取slice层的量化参数偏移slice\_qp\_delta，因此每一个slice第一个宏块的量化参数可通过下面的公式计算：

QP0=pic\_init\_qp+26+slice\_qp\_delta+mb\_qp\_delta

从第二个宏块开始，每个宏块实际量化参数的计算方法为：

QPn=(QPm+mb\_qp\_delta+52)%52

在以H.264格式编码的视频码流中，宏块结构必然包含预测结构（I\_PCM模式除外），该结构中包含了像素块的预测模式等信息。对于不同预测模式的宏块，其预测结构是不同的。从上篇的宏块结构中，可以看出，对于部分模式，预测信息保存于mb\_pred()结构中，而对于另一部分模式则采用sub\_mb\_pred()结构。

在我们本系列的H.264分析器SimpleH264Analyzer项目中默认的全I帧测试码流中，我们所分析的第一个IDR帧的第一个宏块，其mb\_type为I\_NxN。实际上，对于除了I\_PCM模式之外的所有Intra宏块，其预测结构均采用mb\_pred()结构。

从表中可以看出，Intra预测模式的结构主要有两组，分别表示4×4和8×8模式，每一组包含两个元素，分别表示预测模式标识位和预测模式值，以及最后的色度分量预测模式。

prev\_intra4x4\_pred\_mode\_flag和prev\_intra8x8\_pred\_mode\_flag：表示帧内预测模式预测标识。如果该标识位为1，表示帧内预测模式的预测值就是实际的模式，否则就需要另外传递实际的帧内预测模式。

prev\_intra4x4\_pred\_mode\_flag和prev\_intra8x8\_pred\_mode\_flag：表示额外传递的实际帧内预测模式。

intra\_chroma\_pred\_mode：表示色度分量的预测模式，取值范围为[0,3]，分别代表DC、水平、垂直和平面模式。

#### 2.4.4 slice实列解析

https://yq.aliyun.com/users/v62pj2dkppzh6?spm=a2c4e.11153940.0.0.9d9e220e4uqKeP

H264流媒体协议格式中的AVCC及Annex B格式。在对H264进行编解码分析时，碰到一个完整视频帧被切分成多个Slice用于网络媒体传输的情况。Avcc是前面有长度，annexb前面有01开始位。

若完整的视频帧未被切分成多个slice时，可直接被解码成YUV或RGB数据，但是当切分成多个slice，需要根据first\_mb\_in\_slice计算每个帧片所在位置。



若完整的视频帧未被切分成多个slice时，可直接被解码成YUV或RGB数据，但是当切分成多个slice，需要根据first\_mb\_in\_slice计算每个帧片所在位置。

比如上图B帧中数据为0x00 0x00 0x00 0x01 0x41 0xE2，读取第6位转成二进制：1110 0010，与上0x80：0xB8&0x80=0x80,即表示为一帧的第一片(可查询指数哥伦布码，即第一个位为1)。接下来一包数据为0x00 0x00 0x00 0x01 0x41 0x63，则与上0x80为0x0，则表示不是第一片(不是第一个片时的值在上一包的基础上会出现递增)。可归纳规律（假设的数据）：

    第一包：0x00 0x00 0x00 0x01 0x41 0xE1

    第二包：0x00 0x00 0x00 0x01 0x41 0x62

    第三包：0x00 0x00 0x00 0x01 0x41 0x63

    第四包：0x00 0x00 0x00 0x01 0x41 0xE2

可看出第一包0xE1&0x80=0x80则是第一片，0x62&0x80=0x0则为第二包，直到遇到下一个0x80时为完整的一帧(第一包+第二包+第三包)。E最高位1开始，123代表包，直到到下一个最高位为1。

以上就是合并的基本思路，一般情况下I帧不会被切分成多个slice，而是以完整帧进行传输，P帧多数情况下存在切分的情况。

实际保存原始视频的图像数据的部分保存在其他的VCL层的NAL Units中。这部分数据在码流中称作是条带(Slice)。一个Slice包含一帧图像的部分或全部数据，换言之，一帧视频图像可以编码为一个或若干个Slice。一个Slice最少包含一个宏块，最多包含整帧图像的数据。在不同的编码实现中，同一帧图像中所构成的Slice数目不一定相同。在H.264中设计Slice的目的主要在于防止误码的扩散。因为不同的slice之间，其解码操作是独立的。某一个slice的解码过程所参考的数据（例如预测编码）不能越过slice的边界。

根据码流中不同的数据类型，H.264标准中共定义了5总Slice类型：

I slice: 帧内编码的条带；

P slice: 单向帧间编码的条带；

B slice: 双向帧间编码的条带；

SI slice: 切换I条带，用于扩展档次中码流切换使用；

SP slice: 切换P条带，用于扩展档次中码流切换使用；

在I slice中只包含I宏块，不能包含P或B宏块；在P和B slice中，除了相应的P和B类型宏块之外，还可以包含I类型宏块。

Slice header中主要保存了当前slice的一些全局的信息，slice body中的宏块在进行解码时需依赖这些信息。其中比较常见的一些语法元素有：

1.first\_mb\_in\_slice: 当前slice中包含的第一个宏块在整帧中的位置；

2.slice\_type：当前slice的类型；

3.pic\_parameter\_set\_id：当前slice所依赖的pps的id；

4.colour\_plane\_id:当标识位separate\_colour\_plane\_flag为true时，colour\_plane\_id表示当前的颜色分量，0、1、2分别表示Y、U、V分量。

5.frame\_num:表示当前帧序号的一种计量方式。

6.field\_pic\_flag:场编码标识位。当该标识位为1时表示当前slice按照场进行编码；该标识位为0时表示当前slice按照帧进行编码。

7.bottom\_field\_flag:底场标识位。该标志位为1表示当前slice是某一帧的底场；为0表示当前slice为某一帧的顶场。

5.idr\_pic\_id:表示IDR帧的序号。某一个IDR帧所属的所有slice，其idr\_pic\_id应保持一致。该值的取值范围为[0,65535]。

6.pic\_order\_cnt\_lsb:表示当前帧序号的另一种计量方式。

7.delta\_pic\_order\_cnt\_bottom:表示顶场与底场POC差值的计算方法，不存在则默认为0；

8.slice\_qp\_delta：用于计算当前slice内所使用的初始qp值。

#### 2.4.5 宏块MB

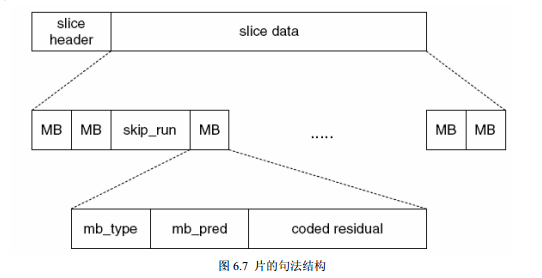


表 6.1 宏块的句法单元

|  |  |
| --- | --- |
| mb\_type | 确定该 MB 是帧内或帧间（P 或 B）编码模式，确定该 MB 分割的尺寸 |
| mb\_pred | 确定帧内预测模式（帧内宏块）确定表 0 或表 1 参考图 像，和每一宏块分割的差分编码的运动矢量（帧间宏块，除 8×8 宏块分割的帧内 MB） |
| sub\_mb\_pred | （只对 8×8MB 分割的帧内 MB）确定每一子宏块的子宏 块分割，每一宏块分割的表 0 和/或表 1 的参考图象；每一 宏块子分割的差分编码运动矢量。 |
| coded\_block\_pattern | 指出哪个 8×8 块（亮度和彩色）包含编码变换系数 |
| mb\_qp\_delta | 量化参数的改变值 |
| residual | 预测后对应于残差图象取样的编码变换系数 |

#### 2.4.6参考链接：

参考链接：<https://blog.csdn.net/liu0808/article/details/72453644>

### 2.6 sps pps slice header数据解析方法

#### 2.6.1 Sps

去除掉NALU开始标记后的67 42 00 1E F1 61 62 62，转换为二进制是01100111 01000010 00000000 00011110 11110001 01100001 01100010 01100010

01100111：NALU头

01000010 00000000 00011110：profile\_idc~level\_idc

1:哥伦布编码 seq\_parameter\_set\_id = 0 // Exp-Golomb解1

后面得1也是分别代表

log2\_max\_frame\_num\_minus4 = 0

pic\_order\_cnt\_type = 0

log2\_max\_pic\_order\_cnt\_lsb\_minus4 = 0

0001 011：Golomb解析max\_num\_ref\_frames，很明显前缀有3个0。所以codenum=2^3-1+read\_bits(011)=10

后面类似计算gaps\_in\_frame\_num\_value\_allowed\_flag = 0 // 0

pic\_width\_in\_mbs\_minus1 = 10 // Exp-Golomb解0001 011

pic\_height\_in\_map\_units\_minus1 = 8 // Exp-Golomb解00010 01

frame\_mbs\_only\_flag = 1 // 1

direct\_8x8\_inference\_flag = 0 // 0

frame\_cropping\_flag = 0 // 0

vui\_parameters\_present\_flag = 0 // 0

之前的这么多数据(除了NALU头之外的)都是seq\_parameter\_set\_data

根据Spec我们知道还有结尾补齐位，

seq\_parameter\_set\_rbsp( ) {

seq\_parameter\_set\_data( ) // 数据

rbsp\_trailing\_bits( ) // 按字节补齐

}

部分结果如下：

profile\_idc = 66 // 01000010

constraint\_set0\_flag = 0 // 0

constraint\_set1\_flag = 0 // 0

constraint\_set2\_flag = 0 // 0

constraint\_set3\_flag = 0 // 0

constraint\_set4\_flag = 0 // 0

constraint\_set5\_flag = 0 // 0

reserved\_zero\_2bits = 0 // 00

level\_idc = 30 // 00011110

以上都是写死，剩余都是指数哥伦布编码

指数哥伦布编码对数字20进行编码：

codeNum = 20

prefixLen = floor[log2(codeNum+1)] = floor[log2(21)]=4

surfix = codeNum+1-2^prefixLen=20+1-2^4=5=二进制的101

编码值=0000，1，0101(为方便观看，以逗号分隔了)

也就是说，哥伦布编码以中间的1为对称轴，前缀全写0，需要先算出一共要写几个0。然后再算后缀的信息位。至于解码，先计算前缀个数4个0，这样后缀的信息位是0101，也就是5。所以解码得出2^4-1+5=20

leadingZeroBits = −1

for (b = 0; !b; leadingZeroBits++)

b = read\_bits(1)

codeNum = 2^(leadingZeroBits) − 1 + read\_bits(leadingZeroBits)

#### 2.6.2 pps

去除掉NALU开始标记后的68 C8 A1 43 88，转换为二进制是

01101000 11001000 10100001 01000011 10001000。

解码结果

01101000:NALU头

pic\_parameter\_set\_id = 0 // Exp-Golomb解1

seq\_parameter\_set\_id = 0 // Exp-Golomb解1

entropy\_coding\_mode\_flag = 0 // 0

bottom\_field\_pic\_order\_in\_frame\_present\_flag = 0 // 0

num\_slice\_groups\_minus1 = 0 // Exp-Golomb解1

num\_ref\_idx\_l0\_default\_active\_minus1 = 9 // Exp-Golomb解000 1010

num\_ref\_idx\_l1\_default\_active\_minus1 = 9 // Exp-Golomb解0001 010

weighted\_pred\_flag = 0 // 0

weighted\_bipred\_idc = 0 // 00

pic\_init\_qp\_minus26 = 0 // Exp-Golomb解1

pic\_init\_qs\_minus26 = 0 // Exp-Golomb解1

chroma\_qp\_index\_offset = 0 // Exp-Golomb解1

deblocking\_filter\_control\_present\_flag = 0 // 0

constrained\_intra\_pred\_flag = 0 // 0

redundant\_pic\_cnt\_present\_flag = 0 // 0

#### 2.6.3 Slice

去除掉NALU开始标记后的65 88 84 02，转换为二进制是

01100101 10001000 10000100 00000010。

解码结果：

first\_mb\_in\_slice = 0 // Exp-Golomb解1

slice\_type = 7 // Exp-Golomb解0001000

pic\_parameter\_set\_id = 0 // Exp-Golomb解1

frame\_num = 0 // u(v)根据占用的位数

//(log2\_max\_frame\_num\_minus4 + 4)解出值 // 0000

也需要知道frame\_num有很多限制，比如在IDR当中必须为0，具体参见7.4.3 Slice header semantics

idr\_pic\_id = 0 // Exp-Golomb解1

pic\_order\_cnt\_lsb = 0 // u(v)根据占用的位数

//(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4)解出值 // 0000

### 2.7H264码流的两种形式：Annex B和AVCC

链接：<https://blog.csdn.net/romantic_energy/article/details/50508332>

1. Annex B

视频编码成的包叫做Network Abstraction Layer Units, 也简称为NALU、NAL，每个NALU包都可以被单独的解析和处理，每个NALU包的第一个字节包含了NALU类型，bit3-bit7包含的内容尤其重要(bit 0一定是off的，bit1-2指定了这个NALU是否被其他NALU引用)。NALU格式分为2类，VCL和non-VCL，总共有19种不同的NALU格式。

Annex B格式用开始码来解决这个问题，即给每个NALU加上前缀码：2个或者3个0x00,后面再加一个0x01, 如：0x000001或者0x00000001。

4字节类型的开始码在在连续的数据传输中非常有用，因为用字节来对齐、分割流数据，比如：用连续的31个bit0后接一个bit1来分割流数据，是很容易的。种类型的序列插入防竞争字节0x03实现的，那么插入防竞争字节后，0x000001变成了0x00000301。

Annex B格式通常用于实时的流格式，比如说传输流，通过无线传输的广播、DVD等。在这些格式中通常会周期性的重复SPS和PPS包，经常是在每一个关键帧之前，因此据此建立解码器可以一个随机访问的点，这样就可以加入一个正在进行的流，及播放一个已经在传输的流。

1. AVCC

另一个存储H.264流的方式是AVCC格式，在这种格式中，每一个NALU包都加上了一个指定其长度(NALU包大小)的前缀(in big endian format大端格式)，这种格式的包非常容易解析，但是这种格式去掉了Annex B格式中的字节对齐特性，而且前缀可以是1、2或4字节，这让AVCC格式变得更复杂了，指定前缀字节数(1、2或4字节)的值保存在一个头部对象中(流开始的部分)，这个头通常称为'extradata'或者'sequence header'，它的基本格式如下：

bits

8 version ( always 0x01 )

8 avc profile ( sps[0][1] )

8 avc compatibility ( sps[0][2] )

8 avc level ( sps[0][3] )

6 reserved ( all bits on )

2 NALULengthSizeMinusOne // 这个值是（前缀长度-1），值如果是3，那前缀就是4，因为4-1=3

3 reserved ( all bits on )

5 number of SPS NALUs (usually 1)

repeated once per SPS:

16 SPS size

variable SPS NALU data

8 number of PPS NALUs (usually 1)

repeated once per PPS

16 PPS size

variable PPS NALU data

AVCC格式的一个优点是在开始配置解码器的时候可以跳到流的中间播放，这种格式通常用于可以被随机访问的多媒体数据，如存储在硬盘的文件。也因为这个特性，MP4、MKV通常用AVCC格式来存储。

## 3. RTP

### 3.1 RTP介绍

#### 3.1.1 RTP概述

RTP全名是Real-time Transport Protocol（实时传输协议）。它是IETF提出的一个标准，对应的RFC文档为RFC3550（RFC1889为其过期版本）。RTP为Internet上端到端的实时传输提供时间信息和流同步，但并不保证服务质量，服务质量由RTCP来提供。

RTP实时传输协议（Real-time Transport Protocol，RTP）是在Internet上处理多媒体数据流的一种网络协议，利用它能够在一对一（单播）或者一对多（multicast，多播） 的网络环境中实现传流媒体数据的实时传输RTP提供抖动补偿和数据无序到达检测的机制。由于IP网络的传输特性，数据的无序到达是很常见的。 RTP允许数据通过IP组播的方式传送到多个目的地。RTP被认为是在IP网络中传输音频和视频的基本标准。RTP通常配合模板和负载格式使用。

#### 3.1.2 RTP传输介绍

RTP通常使用UDP来进行多媒体数据的传输，但如果需要的话可以使用TCP或者ATM等其它协议，整个RTP 协议由两个密切相关的部分组成：RTP数据协议和RTP控制协议，RTP标准定义了两个子协议，RTP和RTCP。RTP 使用偶数端口号接收发送数据，相应的RTCP则使用相邻的下一位奇数端口号。每一个多媒体流会建立一个RTP会话。一个会话包含带有RTP和RTCP端口号的IP地址。例如，音频和视频流使用分开的RTP会话，这样用户可以选择其中一个媒体流。形成会话的端口由其他协议（例如RTSP和SIP）来协商。RTP和RTCP使用UDP端口1024 - 65535。简单的多播音频会议。语音通信通过一个多播地址（一对多得IP地址类似于远程游戏）和一对端口来实现。一个用于音频数据（RTP），另一个用于控制包（RTCP）。

如果在一次会议中同时使用了音频和视频会议，这两种媒体将分别在不同的RTP会话中传送，每一个会话使用不同的传输地址（IP地址＋端口）。如果一个用户同时使用了两个会话，则每个会话对应的RTCP包都使用规范化名字CNAME（Canonical Name）。与会者可以根据RTCP包中的CNAME来获取相关联的音频和视频，然后根据RTCP包中的计时信息(Network time protocol)来实现音频和视频的同步。

**翻译器和混合器：**翻译器和混合器都是RTP级的中继系统。翻译器用在通过IP多播不能直接到达的用户区，例如发送者和接收者之间存在防火墙。当与会者能接收的音频编码格式不一样，比如有一个与会者通过一条低速链路接入到高速会议，这时就要使用混合器。在进入音频数据格式需要变化的网络前，混合器将来自一个源或多个源的音频包进行重构，并把重构后的多个音频合并，采用另一种音频编码进行编码后，再转发这个新的RTP包。从一个混合器出来的所有数据包要用混合器作为它们的同步源（SSRC，见RTP的封装）来识别，可以通过贡献源列表（CSRC表，见RTP的封装）可以确认谈话者。

**数据传输协议RTP**：用于实时传输数据。该协议提供的信息包括：时间戳（用于同步）、序列号（用于丢包和重排序检测）、以及负载格式（用于说明数据的编码格式）。

**控制协议RTCP**：用于QoS反馈和同步媒体流。相对于RTP来说，RTCP所占的带宽非常小，通常只有5%。

#### 3.1.3 参考链接

1.RTP协议全解析（H264码流和PS流）

https://blog.csdn.net/chen495810242/article/details/39207305

### 3.2 RTP数据格式

#### 3.2.1 RTP解析

RTP数据协议。RTP数据协议负责对流媒体数据进行封包并实现媒体流的实时传输，每一个RTP数据报都由头部（Header）和负载（Payload）两个部分组成，其中头部前12个字节的含义是固定的，而负载则可以是音频或者视频数据。

RTP数据报的头部格式如图所示：



**(1) V**：RTP协议的版本号，占2位，当前协议版本号为2

**(2) P**：填充标志，占1位，如果P=1，则在该报文的尾部填充一个或多个额外的八位组，它们不是有效载荷的一部分。

**(3) X**：扩展标志，占1位，如果X=1，则在RTP报头后跟有一个扩展报头

**(4)  CC**：CSRC计数器，占4位，指示CSRC 标识符的个数

**(5)  M**: 标记，占1位，不同的有效载荷有不同的含义，对于视频，标记一帧的结束；对于音频，标记会话的开始。Marker标志

**(6)  PT**: 有效荷载类型，占7位，用于说明RTP报文中有效载荷的类型，如GSM音频、JPEM图像等,在流媒体中大部分是用来区分音频流和视频流的，这样便于客户端进行解析。

**(7) 序列号**：占16位，用于标识发送者所发送的RTP报文的序列号，每发送一个报文，序列号增1。这个字段当下层的承载协议用UDP的时候，网络状况不好的时候可以用来检查丢包。同时出现网络抖动的情况可以用来对数据进行重新排序，序列号的初始值是随机的，同时音频包和视频包的sequence是分别记数的。

**(8)时戳(Timestamp)**：占32位，必须使用90 kHz 时钟频率。时戳反映了该RTP报文的第一个八位组的采样时刻。接收者使用时戳来计算延迟和延迟抖动，并进行同步控制。在marker结束之前的时间戳都一样。

**(9)  同步信源(SSRC)标识符**：占32位，用于标识同步信源。该标识符是随机选择的，参加同一视频会议的两个同步信源不能有相同的SSRC。同步源就是指RTP包流的来源。在同一个RTP会话中不能有两个相同的SSRC值。该标识符是随机选取的 RFC1889推荐了MD5随机算法。

**(10) 特约信源(CSRC)标识符**：每个CSRC标识符占32位，可以有0～15个。每个CSRC标识了包含在该RTP报文有效载荷中的所有特约信源。CSRC标识紧跟在RTP固定头部之后，用来表示RTP数据报的来源，RTP协议允许在同一个会话中存在多个数据源，它们可以通过RTP混合器合并为一个数据源。例如，可以产生一个CSRC列表来表示一个电话会议，该会议通过一个RTP混合器将所有 讲话者的语音数据组合为一个RTP数据源。CSRC不是必须的，个数由CC决定。其他都是固定的，计算字节数需要注意。

注：基本的RTP说明并不定义任何头扩展本身，如果遇到X=1，需要特殊处理

#### 3.2.2 RTCP解析

RTP需要RTCP为其服务质量提供保证，因此下面介绍一下RTCP的相关知识。

RTCP的主要功能是：服务质量的监视与反馈、媒体间的同步，以及多播组中成员的标识。在RTP会话期 间，各参与者周期性地传送RTCP包。RTCP包中含有已发送的数据包的数量、丢失的数据包的数量等统计资料，因此，各参与者可以利用这些信息动态地改变传输速率，甚至改变有效载荷类型。RTP和RTCP配合使用，它们能以有效的反馈和最小的开销使传输效率最佳化，因而特别适合传送网上的实时数据。

RTCP也是用UDP来传送的，但RTCP封装的仅仅是一些控制信息，因而分组很短，所以可以将多个RTCP分组封装在一个UDP包中。RTCP有如下五种分组类型:

**SR：**发送端报告，所谓发送端是指发出RTP数据报的应用程序或者终端，发送端同时也可以是接收端。

**RR：**接收端报告，所谓接收端是指仅接收但不发送RTP数据报的应用程序或者终端。

**SDES：**源描述，主要功能是作为会话成员有关标识信息的载体，如用户名、邮件地址、电话号码等，此外还具有向会话成员传达会话控制信息的功能。

**BYE：**通知离开，主要功能是指示某一个或者几个源不再有效，即通知会话中的其他成员自己将退出会话。

**APP：**由应用程序自己定义，解决了RTCP的扩展性问题，并且为协议的实现者提供了很大的灵活性。



上述五种分组的封装大同小异，下面只讲述SR类型，而其它类型请参考RFC3550。发送端报告分组SR（Sender Report）用来使发送端以多播方式向所有接收端报告发送情况。SR分组的主要内容有：相应的RTP流的SSRC，RTP流中最新产生的RTP分组的时间戳和NTP，RTP流包含的分组数，RTP流包含的字节数。SR包的封装如图3所示。



**版本（V）**：同RTP包头域。

**填充（P）**：同RTP包头域。

**接收报告计数器（RC）**：5比特，该SR包中的接收报告块的数目，可以为零。

**包类型（PT）**：8比特，SR包是200。

**长度域（Length）**：16比特，其中存放的是该SR包以32比特为单位的总长度减一。

**同步源（SSRC）**：SR包发送者的同步源标识符。与对应RTP包中的SSRC一样。NTP Timestamp（Network time protocol）SR包发送时的绝对时间值。NTP的作用是同步不同的RTP媒体流。

**RTP Timestamp**：与NTP时间戳对应，与RTP数据包中的RTP时间戳具有相同的单位和随机初始值。

**Sender’s packet count**：从开始发送包到产生这个SR包这段时间里，发送者发送的RTP数据包的总数. SSRC改变时，这个域清零。

**Sender`s octet count**：从开始发送包到产生这个SR包这段时间里，发送者发送的净荷数据的总字节数（不包括头部和填充）。发送者改变其SSRC时，这个域要清零。

**同步源n的SSRC标识符**：该报告块中包含的是从该源接收到的包的统计信息。

**丢失率（Fraction Lost）：**表明从上一个SR或RR包发出以来从同步源n(SSRC\_n)来的RTP数据包的丢失率。

**累计的包丢失数目**：从开始接收到SSRC\_n的包到发送SR,从SSRC\_n传过来的RTP数据包的丢失总数。

收到的扩展最大序列号：从SSRC\_n收到的RTP数据包中最大的序列号，

**接收抖动（Interarrival jitter）**：RTP数据包接受时间的统计方差估计

**上次SR时间戳（Last SR,LSR）**：取最近从SSRC\_n收到的SR包中的NTP时间戳的中间32比特。如果目前还没收到SR包，则该域清零。

上次SR以来的延时（Delay since last SR,DLSR）：上次从SSRC\_n收到SR包到发送本报告的延时。

RTP设计成允许应用自动扩展，连接数可从几个到上千个。例如，音频会议中，数据流量是内在限制的，因为同一时刻只有一两个人说话；对组播，给定连接数据率仍是常数，独立于连接数，但控制流量不是内在限制的。如每个参加者以固定速率发送接收报告，控制流量将随参加者数量线性增长，因此，速率必须按比例下降。

一旦确认地址有效，如后来标记成未活动，地址的状态应仍保留，地址应继续计入共享RTCP带宽地址的总数中，时间要保证能扫描典型网络分区，建议为30分钟。注意，这仍大于RTCP报告间隔最大值的五倍。

这个规范定义了除必需的CNAME外的几个源描述项，如NAME（人名）和EMAIL（电子邮件地址）。它也为定义新特定应用RTCP包类型的途径。给附加信息分配控制带宽应引起注意，因为它将降低接收报告和CNAME发送的速率而损害协议的性能。建议分配给单个参加者用于携带附加信息的RTCP带宽不要超过20%。而且并没有有意让所有SDES项包含在每个应用中。

### 3.3 RTP会话

#### 3.3.1 RTP会话过程

当应用程序建立一个RTP会话时，应用程序将确定一对目的传输地址。目的传输地址由一个网络地址和一对端口组成，有两个端口：一个给RTP包，一个给RTCP包，使得RTP/RTCP数据能够正确发送。RTP数据发向偶数的UDP端口，而对应的控制信号RTCP数据发向相邻的奇数UDP端口（偶数的UDP端口＋1），这样就构成一个UDP端口对。 RTP的发送过程如下，接收过程则相反。

1)RTP协议从上层接收流媒体信息码流（如H.263），封装成RTP数据包；RTCP从上层接收控制信息，封装成RTCP控制包。

2)RTP将RTP 数据包发往UDP端口对中偶数端口；RTCP将RTCP控制包发往UDP端口对中的接收端口。

### 3.4 RTP荷载H264码流



荷载格式定义三个不同的基本荷载结构，接收者可以通过RTP荷载的第一个字节后5位（如图2）识别荷载结构。

1)单个NAL单元包：荷载中只包含一个NAL单元。NAL头类型域等于原始 NAL单元类型,即在范围1到23之间

2)聚合包：本类型用于聚合多个NAL单元到单个RTP荷载中。本包有四种版本,单时间聚合包类型A (STAP-A)，单时间聚合包类型B (STAP-B)，多时间聚合包类型(MTAP)16位位移(MTAP16), 多时间聚合包类型(MTAP)24位位移(MTAP24)。赋予STAP-A, STAP-B, MTAP16, MTAP24的NAL单元类型号分别是 24,25, 26, 27

3)分片单元：用于分片单个NAL单元到多个RTP包。现存两个版本FU-A，FU-B,用NAL单元类型 28，29标识

常用的打包时的分包规则是：如果小于MTU采用单个NAL单元包，如果大于MTU就采用FUs分片方式。

因为常用的打包方式就是单个NAL包和FU-A方式，所以我们只解析这两种。

#### 3.4.12.1、单个NAL单元包

定义在此的NAL单元包必须只包含一个。这意味聚合包和分片单元不可以用在单个NAL 单元包中。并且RTP序号必须符合NAL单元的解码顺序。NAL单元的第一字节和RTP荷载头第一个字节重合。打包H264码流时，只需在帧前面加上12字节的RTP头即可。



#### IMG_2563.4.2分片单元（FU-A）

分片只定义于单个NAL单元不用于任何聚合包。NAL单元的一个分片由整数个连续NAL单元字节组成。每个NAL单元字节必须正好是该NAL单元一个分片的一部分。相同NAL单元的分片必须使用递增的RTP序号连续顺序发送(第一和最后分片之间没有其他的RTP包）。相似，NAL单元必须按照RTP顺序号的顺序装配。

当一个NAL单元被分片运送在分片单元(FUs)中时，被引用为分片NAL单元。STAPs,MTAPs不可以被分片。 FUs不可以嵌套。 即, 一个FU 不可以包含另一个FU。运送FU的RTP时戳被设置成分片NAL单元的NALU时刻。

图 4 表示FU-A的RTP荷载格式。FU-A由1字节的分片单元指示（如图5），1字节的分片单元头（如图6），和分片单元荷载组成。



S: 1 bit 当设置成1,开始位指示分片NAL单元的开始。当跟随的FU荷载不是分片NAL单元荷载的开始，开始位设为0。

E: 1 bit 当设置成1, 结束位指示分片NAL单元的结束，即, 荷载的最后字节也是分片NAL单元的最后一个字节。当跟随的 FU荷载不是分片NAL单元的最后分片,结束位设置为0。

R: 1 bit 保留位必须设置为0，接收者必须忽略该位

打包时，原始的NAL头的前三位为FU indicator的前三位，原始的NAL头的后五位为FU header的后五位。

取一段码流分析如下：

80 60 01 0f 00 0e 10 00 00 0000 00 7c 85 88 82 €`..........|???

00 0a 7f ca 94 05 3b 7f 3e 7f fe 14 2b 27 26 f8 ...??.;.>.?.+'&?

89 88 dd 85 62 e1 6d fc 33 01 38 1a 10 35 f2 14 ????b?m?3.8..5?.

84 6e 21 24 8f 72 62f0 51 7e 10 5f 0d 42 71 12 ?n!$?rb?Q~.\_.Bq.

17 65 62 a1 f1 44 dc df 4b 4a 38 aa 96 b7 dd 24 .eb??D??KJ8????$

前12字节是RTP Header

7c是FU indicator

85是FU Header

FU indicator（0x7C）和FU Header（0x85）换成二进制如下

0111 1100 1000 0101

按顺序解析如下：

0                            是F

11                          是NRI

11100                     是FU Type，这里是28，即FU-A

1                            是S，Start，说明是分片的第一包

0                            是E，End，如果是分片的最后一包，设置为1，这里不是

0                            是R，Remain，保留位，总是0

00101                     是NAl Type，这里是5，说明是关键帧

打包时，FUindicator的F、NRI是NAL Header中的F、NRI，Type是28；FU Header的S、E、R分别按照分片起始位置设置，Type是NAL Header中的Type。

解包时，取FU indicator的前三位和FU Header的后五位，即0110 0101（0x65）为NAL类型。

### 3.5 RTP荷载PS流

针对H264 做如下PS 封装：每个IDR NALU 前一般都会包含SPS、PPS 等NALU，因此将SPS、PPS、IDR 的NALU 封装为一个PS 包，包括ps 头，然后加上PS system header，PS system map，PES header+h264 raw data。所以一个IDR NALU PS 包由外到内顺序是：PSheader| PS system header | PS system Map | PES header | h264 raw data。对于其它非关键帧的PS 包，就简单多了，直接加上PS头和PES 头就可以了。顺序为：PS header | PES header | h264raw data。以上是对只有视频video 的情况，如果要把音频Audio也打包进PS 封装，也可以。当有音频数据时，将数据加上PES header 放到视频PES 后就可以了。顺序如下：PS 包=PS头|PES(video)|PES(audio)，再用RTP 封装发送就可以了。

        GB28181 对RTP 传输的数据负载类型有规定（参考GB28181 附录B），负载类型中96-127

        RFC2250 建议96 表示PS 封装，建议97 为MPEG-4，建议98 为H264

        即我们接收到的RTP 包首先需要判断负载类型，若负载类型为96，则采用PS 解复用，将音视频分开解码。若负载类型为98，直接按照H264 的解码类型解码。

        注：此方法不一定准确，取决于打包格式是否标准

PS 包中的流类型（stream type）的取值如下：

1)        MPEG-4 视频流： 0x10；

2)        H.264 视频流： 0x1B；

3)        SVAC 视频流： 0x80；

4)        G.711 音频流： 0x90；

5)        G.722.1 音频流： 0x92；

6)        G.723.1 音频流： 0x93；

7)        G.729 音频流： 0x99；

8)     SVAC音频流： 0x9B。

### 3.6 RTP的时间戳 同步

首先，了解几个基本概念：

时间戳单位：时间戳计算的单位不是秒之类的单位，而是由采样频率所代替的单位，这样做的目的就是 为了是时间戳单位更为精准。比如说一个音频的采样频率为8000Hz，那么我们可以把时间戳单位设为1 / 8000。

时间戳增量：相邻两个RTP包之间的时间差（以时间 戳单位为基准）。

采样频率：每秒钟抽取样本的次数，例如音频的采样率一般为8000Hz

帧率： 每秒传输或者显示帧数，例如25f/s

RTP包头的第2个32Bit即为RTP包的时间戳，Time Stamp ，占32位。

时间戳反映了RTP分组中的数据的第一个字节的采样时刻。在一次会话开始时的时间戳初值也是随机选择的。即使是没有信号发送时，时间戳的数值也要随时间不断的增加。接收端使用时间戳可准确知道应当在什么时间还原哪一个数据块，从而消除传输中的抖动。时间戳还可用来使视频应用中声音和图像同步。

在RTP协议中并没有规定时间戳的粒度，这取决于有效载荷的类型。因此RTP的时间戳又称为媒体时间戳，以强调这种时间戳的粒度取决于信号的类型。例如， 对于8kHz采样的话音信号，若每隔20ms构成一个数据块，则一个数据块中包含有160个样本（0.02×8000=160）。因此每发送一个RTP分 组，其时间戳的值就增加160。

官方的解释看懂没？没看懂？没关系，我刚开始也没看懂，那就听我的解释吧。

首先，时间戳就是一个值，用来反映某个数据块的产生（采集）时间点的， 后采集的数据块的时间戳肯定是大于先采集的数据块的。有了这样一个时间戳，就可以标记数据块的先后顺序。

第二，在实时流传输中，数据采集后立刻传递到RTP 模块进行发送，那么，其实，数据块的采集时间戳就直接作为RTP包的时间戳。

第三，如果用RTP来传输固定的文件，则这个时间戳 就是读文件的时间点，依次递增。这个不再我们当前的讨论范围内，暂时不考虑。

第四，时间戳的单位采用的是采样频率的倒数，例如采 样频率为8000Hz时，时间戳的单位为1 / 8000 ，在Jrtplib库中，有设置时间戳单位的函数接口，而ORTP库中根据负载类型直接给定了时间戳的单位（音频负载1/8000，视频负载1 /90000）

第五，时间戳增量是指两个RTP包之间的时间间隔， 详细点说，就是发送第二个RTP包相距发送第一个RTP包时的时间间隔（单位是时间戳单位）。

如果采样频率为90000Hz，则由上面讨论可知，时间戳单位为1/90000，我们就假设1s钟被划分了90000个时间块，那么，如果每秒发送25 帧，那么，每一个帧的发送占多少个时间块呢？当然是 90000/25 = 3600。因此，我们根据定义“时间戳增量是发送第二个RTP包相距发送第一个RTP包时的时间间隔”，故时间戳增量应该为3600。

下面说一下播放器如何对编码器发送的RTP包的时间戳做处理。这里的编码器是指发送RTP包的服务器，一般是指网络摄像机或RTSP服务器。

播放器本地建立了一个系统时钟，这个时钟一般根据系统的CPU时间来计算出来，当播放开始，时钟的时间为0。时间戳有分绝对时间和相对时间，收到的RTP包的时间戳为绝对时间，播放器需要把这个RTP时间戳转为播放器本地的相对时间戳，相对时间戳以0为开始，时间戳决定了一帧播放或渲染的时刻。当播放开始时，时钟开始运行，时间戳一直增加，播放器会用系统时钟的时间跟视频帧的时间戳（相对时间戳）进行对比，根据它们的大小关系决定是否马上渲染这帧还是要等待。但是，编码器发过来的时间戳只是一个参考值，有可能是不准的，为什么呢？编码器打时间戳基于的时钟跟播放器的本地时钟是可能不一样的，不同的编码器有不同的打时间戳算法，我只说一下常见的做法，一般地，编码器打时间戳是基于一个简单的规则：每发一个视频帧，PTS就递增一个固定的时间增量。根据TS流和Rtp协议的规定，视频的时钟频率是90000，也就是时间单位是1/90000秒。如果帧率是25帧，则相邻两帧的PTS增量就是3600，也就是每发一个帧，时间戳就增加3600，但是实际上两帧的时间增量应该根据它们实际经过的时间，而不是一个固定的值。实际上，编码器每秒采集图像的帧率并不是绝对固定的（取决于物理时钟的精度以及CPU处理的速度），有一定误差，平均采集帧率是25，但可能编码器某1秒采集了25帧，某一秒采集了24帧，假如都是按固定增量的方式递增时间戳，过一段时间后，编码器的时间戳跟播放器的时钟的时间就有很大差异，可能导致播放不流畅，或者缓冲了很多帧延时加大了。

   所以，解码器不能盲目相信编码器的时间戳，发现时间戳跟本地的时间差异太远需要尽快调整，应该实现一个动态适应的机制：当发现接收缓冲区里的帧数比较多时，要加快渲染速度；当缓冲区的帧数在一个比较低的水平（1-4），则稳定渲染速度

在Jrtplib中好像不需要自己管理时间戳的递增，由库内部管理。但在ORTP中每次数据的发送都需要自己传入时间戳的值，即自己 需要每次发完一个RTP包后，累加时间戳增量，不是很方便，这就需要自己对RTP的时间戳有比较深刻地理解，我刚开始就是因为没搞清楚，随时设置时间戳增 量导致传输一直有问题，困扰我好久。

多媒体通信同步方法，主要有时间戳同步法、同步标记法、多路复用同步法三种。下面主要讨论时间戳同步法，特别是RTP时间戳同步。内容包括RTP媒体间同步的实现，为什么需要RTCP的NTP时间来实现媒体间同步？没有RTCP，能实现RTP媒体间的同步吗？DirectShow时间戳和RTP时间 戳的区别，MPEG2-TS流的时间戳等。本文只简单讨论时间戳同步的原理，不涉及具体的实现方法，如音频帧和视频帧时间戳的计算方法，怎样根据时间戳去做音视频的呈现等。

根据RTP规范，不同的RTP媒体流是分开传输的，且使用各自独立的时间戳进行同步。假设在一次视频点播中，传输两路RTP媒体流，一路视频，一 路音频。根据视频帧时间戳，可以实现视频流内同步，这很好理解，通过视频帧时间戳可以计算出相邻视频帧的时间间隔，也就是视频帧之间的相对时间关系很容易 通过时间戳来确定，按照这个间隔去呈现视频，就可以获得较好的效果。同理，音频流也可以实现自身的同步。

那么音频和视频这两路媒体间如何实现同步呢？我们只使用音视频的RTP时间戳，看能否实现媒体间的同步。音视频的RTP时间戳的增长速率一般是不同的，但没关系，知道了具体的单位后，两者是可以通过单位换算联系起来的。如下图：

现在来看，这种方法好像可以实现同步，因为音视频被映射到同一个时间轴上了，音频和视频帧间的相对关系很清楚。慢着，RTP规范要求时间戳的初 始值应该是一个随机值，那么假设音频帧时间戳的初始值是随机值1234，视频帧时间戳的初始值是随机值5678，看起来应该是下面这样：

这么做合适吗？我们把音频帧时间戳1234和视频帧时间戳5678对应到绝对时间轴的0上，我们这么做的理由是什么？你可能会说，因为那是第一 个音频帧和第一个视频帧，所以可以对应到同一个点上，在第一幅图中我们就是这么做的，把音频帧时间戳0和视频帧时间戳0对应到绝对时间轴的0上。但是 RTP规范并没有规定第一个视频帧的时间戳和第一个音频帧的时间戳必须或者应该对应到绝对时间轴的同一个点上，从整个RTP规范中不能直接得出这样的结 论，也推导不出这样的结论。

我们上面两幅图所做的转换是不正确的，为什么呢？因为在做转换时，隐含了一个假设，我们想当然地认为这个假设是成立的，实际上它并不总是成立。这个假设就是第一个视频帧和第一个音频帧的时间戳应该对应到同一个点上，即无论它们时间戳是多少，都应该在同一时间播放。

仅仅使用RTP时间戳是无法实现媒体间同步的，根本的原因是音频时间轴和视频时间轴是完全独立的，通过音频帧和视频帧的时间戳，无法确定一个视频帧和一个音频帧的相对时间关系，也就是无法把它们都准确定位在绝对时间轴上，只能准确定位一个。

要实现RTP媒体间同步，需要借助于RTCP，在RTCP的SR包中，包含有<NTP时间，RTP时间戳>对，音频帧RTP时间戳和视 频帧RTP时间戳通过<NTP时间，RTP时间戳>对，都可以准确定位到绝对时间轴NTP上，音频帧和视频帧的相对时间关系就可以确定下来 了。

上面提到，我们的那个隐含的假设并不总是成立，那就是说它有成立的时候。那是不是说当它成立时，我们就可以不用RTCP来做媒体间同步了？答案是，基本上可以这么认为。

例如，对于RTP实时流，在发送端媒体间就同步的很好，在接收端只需做少许处理，不需要RTCP，就可以实现媒体间同步。当然，这只是少数例外。因为RTP规范并不包括这个假设，所以我们还是按照RTP规范来做吧。

下面说一下DirectShow和MPEG2-TS的时间戳。DirectShow中的时间戳和RTP中的时间戳，除了单位不一样，计算方法不一样 外，本质的区别就是DirectShow中的音频帧和视频帧时间戳使用的是同一个时间轴，所以不需要借助其他的东西，仅仅使用音频帧时间戳和视频帧时间戳 就可以实现媒体间同步。MPEG2-TS流中也有时间戳，它的时间戳和RTP及DirectShow的时间戳都不同，TS流中的音频帧和视频帧时间戳使用 的也是同一个时间轴，TS流中的音频和视频是复用的，这在一定程度上就起到了同步的作用，所以它并不是在每个帧上都打时间戳，比如它的PTS时间戳就是每 隔0.1秒一个，缺失的时间戳是通过其他时间戳插值计算出来的。

### 3.7 RTP打包方式

RFC3984 给出了3 中不同的RTP 打包方案：

（1）Single NALU Packet:在一个RTP 包中只封装一个NALU，在本文中对于小于 1400字节的NALU 便采用这种打包方案。  
       （2）Aggregation Packet:在一个RTP 包中封装多个NALU，对于较小的NALU 可以采用这种打包方案，从而提高传输效率。   
       （3）Fragmentation Unit:一个NALU 封装在多个RTP包中，在本文中，对于大于1400字节的NALU 便采用这种方案进行拆包处理。

### 3.8 RTP时间戳和播放器处理RTP时间戳

RTP包头的第2个32Bit即为RTP包的时间戳，Time Stamp ，占32位。时间戳反映了RTP分组中的数据的第一个字节的采样时刻。在会话开始时的时间戳初值也是随机选择的。即使是没有信号发送时，时间戳的数值也要随时间不断的增加。接收端使用时间戳可准确知道应当在什么时间还原哪一个数据块，从而消除传输中的抖动。时间戳还可用来使视频应用中声音和图像同步。在RTP协议中并没有规定时间戳的粒度，这取决于有效载荷的类型。因此RTP的时间戳又称为媒体时间戳，以强调这种时间戳的粒度取决于信号的类型。例如，对于8kHz采样的话音信号，若每隔20ms构成一个数据块，则一个数据块中包含有160个样本（0.02×8000=160）。因此每发送一个RTP分组，其时间戳的值就增加160。

第一先，时间戳就是一个值，用来反映某个数据块的产生（采集）时间点的，后采集的数据块的时间戳肯定是大于先采集的数据块的。有了这样一个时间戳，就可以标记数据块的先后顺序。

 第二，在实时流传输中，数据采集后立刻传递到RTP模块进行发送，那么，其实，数据块的采集时间戳就直接作为RTP包的时间戳。

 第三，如果用RTP来传输固定的文件，则这个时间戳就是读文件的时间点，依次递增。这个不再我们当前的讨论范围内，暂时不考虑。

 第四，时间戳的单位采用的是采样频率的倒数，例如采样频率为8000Hz时，时间戳的单位为1 / 8000 ，在Jrtplib库中，有设置时间戳单位的函数接口，而ORTP库中根据负载类型直接给定了时间戳的单位（音频负载1/8000，视频负载1/90000）

第五，时间戳增量是指两个RTP包之间的时间间隔，详细点说，就是发送第二个RTP包相距发送第一个RTP包时的时间间隔（单位是时间戳单位）。

如果采样频率为90000Hz，则由上面讨论可知，时间戳单位为1/90000，我们就假设1s钟被划分了90000个时间块，那么，如果每秒发送25帧，那么，每一个帧的发送占多少个时间块呢？当然是 90000/25 = 3600。因此，我们根据定义“时间戳增量是发送第二个RTP包相距发送第一个RTP包时的时间间隔”，故时间戳增量应该为3600。

下面说一下播放器如何对编码器发送的RTP包的时间戳做处理。这里的编码器是指发送RTP包的服务器，一般是指网络摄像机或RTSP服务器。

播放器本地建立了一个系统时钟，这个时钟一般根据系统的CPU时间来计算出来，当播放开始，时钟的时间为0。时间戳有分绝对时间和相对时间，收到的RTP包的时间戳为绝对时间，播放器需要把这个RTP时间戳转为播放器本地的相对时间戳，相对时间戳以0为开始，时间戳决定了一帧播放或渲染的时刻。当播放开始时，时钟开始运行，时间戳一直增加，播放器会用系统时钟的时间跟视频帧的时间戳（相对时间戳）进行对比，根据它们的大小关系决定是否马上渲染这帧还是要等待。但是，编码器发过来的时间戳只是一个参考值，有可能是不准的，为什么呢？编码器打时间戳基于的时钟跟播放器的本地时钟是可能不一样的，不同的编码器有不同的打时间戳算法，我只说一下常见的做法，一般地，编码器打时间戳是基于一个简单的规则：每发一个视频帧，PTS就递增一个固定的时间增量。根据TS流和Rtp协议的规定，视频的时钟频率是90000，也就是时间单位是1/90000秒。如果帧率是25帧，则相邻两帧的PTS增量就是3600，也就是每发一个帧，时间戳就增加3600，但是实际上两帧的时间增量应该根据它们实际经过的时间，而不是一个固定的值。实际上，编码器每秒采集图像的帧率并不是绝对固定的（取决于物理时钟的精度以及CPU处理的速度），有一定误差，平均采集帧率是25，但可能编码器某1秒采集了25帧，某一秒采集了24帧，假如都是按固定增量的方式递增时间戳，过一段时间后，编码器的时间戳跟播放器的时钟的时间就有很大差异，可能导致播放不流畅，或者缓冲了很多帧延时加大了。

所以，解码器不能盲目相信编码器的时间戳，发现时间戳跟本地的时间差异太远需要尽快调整，应该实现一个动态适应的机制：当发现接收缓冲区里的帧数比较多时，要加快渲染速度；当缓冲区的帧数在一个比较低的水平（1-4），则稳定渲染速度。

## 4. RTSP

### 4.1 RTSP概述

实时流协议RTSP（Real Time Streaming Protocol）最早由Real Networks和Netscape公司共同提出，它位于RTP和RTCP之上，其目的是希望通过IP网络有效地传输多媒体数据。RTSP在体系结构上位于RTP和RTCP之上，它使用TCP或RTP完成数据传输。HTTP与RTSP相比，HTTP传送HTML，而RTP传送的是多媒体数据。HTTP请求由客户机发出，服务器作出响应；使用RTSP时，客户机和服务器都可以发出请求，即RTSP可以是双向的。

作为一个应用层协议，RTSP提供了一个可供扩展的框架，它的意义在于使得实时流媒体数据的受控和点播变得可能。总的说来，RTSP是一个流媒体表示协议，主要用来控制具有实时特性的数据发送，但它本身并不传输数据，而是必须依赖于下层传输协议所提供的某些服务。RTSP可以对流媒体提供诸如播放、 暂停、快进等操作，它负责定义具体的控制消息、操作方法、状态码等，此外还描述了与RTP间的交互操作。

RTSP在制定时较多地参考了HTTP/1.1协议，甚至许多描述与HTTP/1.1完全相同。RTSP之所以特意使用与HTTP/1.1类似的语 法和操作，在很大程度上是为了兼容现有的Web基础结构，正因如此，HTTP/1.1的扩展机制大都可以直接引入到RTSP中。

由RTSP控制的媒体流集合可以用表示描述（Presentation Description）来定义，所谓表示是指流媒体服务器提供给客户机的一个或者多个媒体流的集合，而表示描述则包含了一个表示中各个媒体流的相关信 息，如数据编码/解码算法、网络地址、媒体流的内容等。

虽然RTSP服务器同样也使用标识符来区别每一流连接会话（Session），但RTSP连接并没有被绑定到传输层连接（如TCP等），也就是说在 整个RTSP连接期间，RTSP用户可打开或者关闭多个对RTSP服务器的可靠传输连接以发出RTSP 请求。此外，RTSP连接也可以基于面向无连接的传输协议（如UDP等）。

RTSP协议目前支持以下操作：

检索媒体：允许用户通过HTTP或者其它方法向媒体服务器提交一个表示描述。如表示是组播的，则表示描述就包含用于该媒体流的组播地址和端口号；如果表示是单播的， 为了安全在表示描述中应该只提供目的地址。

邀请加入：媒体服务器可以被邀请参加正在进行的会议，或者在表示中回放媒体，或者在表示中录制全部媒体或其子集，非常适合于分布式教学。，会议中几方可轮流按远程控制按钮。

添加媒体：通知用户新加入的可利用媒体流，这对现场讲座来讲显得尤其有用。与HTTP/1.1类似，RTSP请求也可以交由代理、通道或者缓存来进行处理。　如服务器告诉用户可获得附加媒体内容，对现场讲座显得尤其有用。

### 4.2 RTSP状态

RTSP控制通过单独协议发送的流，与控制通道无关。例如，RTSP控制可通过TCP连接，而数据流通过UDP。因此，即使媒体服务器没有收到请求，数据也会继续发送。在连接生命期，单个媒体流可通过不同TCP连接顺序发出请求来控制。所以，服务器需要维持能联系流与RTSP请求的连接状态。RTSP中很多方法与状态无关，但下列方法在定义服务器流资源的分配与应用上起着重要的作用：

SETUP：让服务器给流分配资源，启动RTSP连接。

PLAY与RECORD：启动SETUP 分配流的数据传输。

PAUSE：临时停止流，而不释放服务器资源。

TEARDOWN：释放流的资源，RTSP连接停止。

标识状态的RTSP方法使用连接头段识别RTSP连接，为响应SETUP请求，服务器连接产生连接标识。

### 4.3 RTSP协议消息

#### 4.3.1 RTSP消息格式

RTSP的消息有两大类 --- 请求消息(request), 回应消息(response)。  
请求消息：  
方法 URI RTSP版本 CR LF   
消息头 CR LF CR LF   
消息体 CR LF

如下：

DESCRIBE rtsp://192.168.1.211 RTSP/1.0

CSeq: 1

Accept: application/sdp

User-Agent: magnus-fc

其中方法包括OPTION回应中所有的命令,URI是接受方的地址,例如:rtsp://192.168.20.136。RTSP版本一般都是 RTSP/1.0。每行后面的CR LF表示回车换行，需要接受端有相应的解析，最后一个消息头需要有两个CR LF。

回应消息：  
RTSP版本 状态码 解释 CR LF   
消息头 CR LF CR LF   
消息体 CR LF

如下：

RTSP/1.0 200 OK

CSeq: 1

Server: GrandStream Rtsp Server V100R001

Content-Type: application/sdp

Content-length: 256

Content-Base: rtsp://192.168.1.211/0

v=0

o=StreamingServer 3331435948 1116907222000 IN IP4 192.168.1.211

s=h264.mp4

c=IN IP4 0.0.0.0

t=0 0

a=control:\*

m=video 0 RTP/AVP 96

a=control:trackID=0

a=rtpmap:96 H264/90000

m=audio 0 RTP/AVP 97

a=control:trackID=1

a=rtpmap:97 G726-16/8000

其中RTSP版本一般都是RTSP/1.0, 状态码是一个数值, 200表示成功, 解释是与状态码对应的文本解释.

#### 4.3.2 rtsp交互过程

简单的rtsp交互过程:

C表示rtsp客户端, S表示rtsp服务端

1. C->S:OPTION request //询问S有哪些方法可用

1. S->C:OPTION response //S回应信息中包括提供的所有可用方法

2. C->S:DESCRIBE request //要求得到S提供的媒体初始化描述信息

2. S->C:DESCRIBE response //S回应媒体初始化描述信息，主要是sdp

3. C->S:SETUP request //设置会话的属性，以及传输模式，提醒S建立会话

3. S->C:SETUP response //S建立会话，返回会话标识符，以及会话相关信息

4. C->S:PLAY request //C请求播放

4. S->C:PLAY response //S回应该请求的信息

S->C:发送流媒体数据

5. C->S:TEARDOWN request //C请求关闭会话

5. S->C:TEARDOWN response //S回应该请求

上述的过程是标准的、友好的rtsp流程，但实际的需求中并不一定按部就班来。 其 中第3和4步是必需的！第一步，只要服务器客户端约定好，有哪些方法可用，则option请求可以不要。第二步，如果我们有其他途径得到媒体初始化描述信 息（比如http请求等等），则我们也不需要通过rtsp中的describe请求来完成。第五步，可以根据系统需求的设计来决定是否需要。

#### 4.3.3 rtsp中常用方法

OPTION

目的是得到服务器提供的可用方法:

OPTIONS rtsp://192.168.20.136:5000/xxx666 RTSP/1.0

CSeq: 1 //每个消息都有序号来标记，第一个包通常是option请求消息

User-Agent: VLC media player (LIVE555 Streaming Media v2005.11.10)

服务器的回应信息包括提供的一些方法,例如:

RTSP/1.0 200 OK

Server: UServer 0.9.7\_rc1

Cseq: 1 //每个回应消息的cseq数值和请求消息的cseq相对应

Public: OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, SCALE,GET\_PARAMETER //服务器提供的可用的方法

DESCRIBE

C向S发起DESCRIBE请求,为了得到会话描述信息(SDP):

DESCRIBE rtsp://192.168.20.136:5000/xxx666 RTSP/1.0

CSeq: 2

token:

Accept: application/sdp

User-Agent: VLC media player (LIVE555 Streaming Media v2005.11.10)

服务器回应一些对此会话的描述信息(sdp):

RTSP/1.0 200 OK

Server: UServer 0.9.7\_rc1

Cseq: 2

x-prev-url: rtsp://192.168.20.136:5000

x-next-url: rtsp://192.168.20.136:5000

x-Accept-Retransmit: our-retransmit

x-Accept-Dynamic-Rate: 1

Cache-Control: must-revalidate

Last-Modified: Fri, 10 Nov 2006 12:34:38 GMT

Date: Fri, 10 Nov 2006 12:34:38 GMT

Expires: Fri, 10 Nov 2006 12:34:38 GMT

Content-Base: rtsp://192.168.20.136:5000/xxx666/

Content-Length: 344

Content-Type: application/sdp

v=0 //以下都是sdp信息

o=OnewaveUServerNG 1451516402 1025358037 IN IP4 192.168.20.136

s=/xxx666

u=http:///

e=admin@

c=IN IP4 0.0.0.0

t=0 0

a=isma-compliance:1,1.0,1

a=range:npt=0-

m=video 0 RTP/AVP 96 //m表示媒体描述，下面是对会话中视频通道的媒体描述

a=rtpmap:96 MP4V-ES/90000

a=fmtp:96 profile-level-id=245;config=000001B0F5000001B509000001000000012000C888B0E0E0FA62D089028307 a=control:trackID=0 //trackID＝0表示视频流用的是通道0

SETUP

客户端提醒服务器建立会话,并确定传输模式:

SETUP rtsp://192.168.20.136:5000/xxx666/trackID=0 RTSP/1.0

CSeq: 3

Transport: RTP/AVP/TCP;unicast;interleaved=0-1

User-Agent: VLC media player (LIVE555 Streaming Media v2005.11.10)

//uri中 带有trackID＝0，表示对该通道进行设置。Transport参数设置了传输模式，包的结构。接下来的数据包头部第二个字节位置就是 interleaved，它的值是每个通道都不同的，trackID＝0的interleaved值有两个0或1，0表示rtp包，1表示rtcp包，接 受端根据interleaved的值来区别是哪种数据包。

服务器回应信息:

RTSP/1.0 200 OK

Server: UServer 0.9.7\_rc1

Cseq: 3

Session: 6310936469860791894 //服务器回应的会话标识符

Cache-Control: no-cache

Transport: RTP/AVP/TCP;unicast;interleaved=0-1;ssrc=6B8B4567

PLAY

客户端发送播放请求:

PLAY rtsp://192.168.20.136:5000/xxx666 RTSP/1.0

CSeq: 4

Session: 6310936469860791894

Range: npt=0.000- //设置播放时间的范围

User-Agent: VLC media player (LIVE555 Streaming Media v2005.11.10)

服务器回应信息:

RTSP/1.0 200 OK

Server: UServer 0.9.7\_rc1

Cseq: 4

Session: 6310936469860791894

Range: npt=0.000000-

RTP-Info: url=trackID=0;seq=17040;rtptime=1467265309

//seq和rtptime都是rtp包中的信息

Range——播放时间

Range: npt=0.0-end

Range:clock=20100318T021919.35Z-20100318T031919.80Z

Scale——播放速度

Scale: 1.0

5.5 TEARDOWN

客户端发起关闭请求:

TEARDOWN rtsp://192.168.20.136:5000/xxx666 RTSP/1.0

CSeq: 5

Session: 6310936469860791894

User-Agent: VLC media player (LIVE555 Streaming Media v2005.11.10)

服务器回应:

RTSP/1.0 200 OK

Server: UServer 0.9.7\_rc1

Cseq: 5

Session: 6310936469860791894

Connection: Close

以上方法都是交互过程中最为常用的, 其它还有一些重要的方法如get/set\_parameter,pause,redirect等等

### 4.4 SDP协议解析

sdp的格式：

v=<version>

o=<username> <session id> <version> <network type> <address type> <address>

s=<session name>

i=<session description>

u=<URI>

e=<email address>

p=<phone number>

c=<network type> <address type> <connection address>

b=<modifier>:<bandwidth-value>

t=<start time> <stop time>

r=<repeat interval> <active duration> <list of offsets from start-time>

z=<adjustment time> <offset> <adjustment time> <offset> ....

k=<method>

k=<method>:<encryption key>

a=<attribute>

a=<attribute>:<value>

m=<media> <port> <transport> <fmt list>

v = （协议版本）

o = （所有者/创建者和会话标识符）

s = （会话名称）

i = \* （会话信息）

u = \* （URI 描述）

e = \* （Email 地址）

p = \* （电话号码）

c = \* （连接信息）

b = \* （带宽信息）

z = \* （时间区域调整）

k = \* （加密密钥）

a = \* （0 个或多个会话属性行）

时间描述：

t = （会话活动时间）

r = \* （0或多次重复次数）

媒体描述：

m = （媒体名称和传输地址）

i = \* （媒体标题）

c = \* （连接信息 — 如果包含在会话层则该字段可选）

b = \* （带宽信息）

k = \* （加密密钥）

a = \* （0 个或多个媒体属性行）

SDP一会话描述协议一描述SAP、sIP和RTSR会话的协议,是一种文件描述协议,是由服务器生成的描述媒体文件编码信息以及所在服务器的链接等的信息。在多媒体会话

中sDP传送有关媒体流的信息,使会话描述的参人方加人会话。sDP主要用于Intemet网中,但也可以在其它网络环境下使用。SDP十分通用,可描述其它网络环境中的会话,但主要用

于Intemet中。在Intemet环境下,sDP有两个主要目的:一是表明会话存在,二是传送足够信息给接收方,以便能加人、参加该会话。SDP所传达的信息包括:会话名称和目的,会话

活动时间,组成会话媒体种类,接收这些媒体的控制信息(如地址、端口、格式、带宽和会议管理人员资料等)。

总结：在RTSP交互过程中，只要在客户端发出Describe请求的时候，服务端回应的时候会有SDP消息发出，用SDP来描述会话情况和内容，方便客户端能够加入该会话，

参考链接：

1.SDP协议详细总结

https://blog.csdn.net/weixin\_42462202/article/details/90710599

### 4.6 资源预订协议RSVP协议

由于音频和视频数据流比传统数据对网络的延时更敏感，要在网络中传输高质量的音频、视频信息，除带宽要求之外，还需其他更多的条件。RSVP(ResourceReserveProtocol)是正在开发的Internet上的资源预订协议，使用RSVP预留一部分网络资源（即带宽），能在一定程度上为流媒体的传输提供QoS。在某些试验性的系统如网络视频会议工具vic中就集成了RSVP。

RSVP运行在传输层，在IP上层。与ICMP和IGMP相比，它是一个控制协议。 RSVP协议的两个重要概念是流与预定。流是从发送者到一个或多个接收者的连接特征，通过IP包中"流标记"来认证。发送一个流前，发送者传输一个路径信息到目的接收方，这个信息包括源IP地址、目的IP地址和一个流规格。这个流规格是由流的速率和延迟组成的，这是流的QoS需要的。接收者实现预定后，基于接收者的模式能够实现一种分布式解决方案。

　 RSVP领域的发展是非常迅速的，但目前并没有在任何一种网络上得到证实，它的应用只是局限在测试的小Intranet网络上。因为RSVP的预定必须建立在完全流方式的基础上，其可扩展性问题倍受关注。

## AAC音频格式

### 5.1 音频初步介绍

DTS：全称为Digital TheaterSystems(数字影院系统)，是一种有损多声道家庭影院音频格式，但它用了很高的码率进行编码，通常为768-1536kbps，能够营造出比AC3更好的影院效果。两者都是基于多声道的编码技术，通用5.1声道（三组全频立体声）--左/右前置，中置/低音，左/右后置环绕。其中低音单元专门重放120HZ以下的低频，以增强效果。DTS的数据流量通常为768/1536Kbps,压缩比只有3：1，其从提高数字空间的利用率着手，使信息数据得以充分利用。理论上它的声场无论在真实性、细腻性、连续性、宽广性和层次性方面优于AC3。

AC3, 全称为Audio Coding version 3，是Dolby实验室所发展的有损音频编码格式。AC3最被广泛应用于5.1声道，是Dolby Pro Logic的继承者，不同的地方在于AC3提供6个独立的声道而Pro Logic混合其环绕声道。AC3普及度很高，以384-448 kbps的码率应用于LaserDisc和DVD，也经常以640 kbps的码率广泛用在电影院。AC3通常为384/448Kbps,采用了高压缩编码（12：1），通过大副删除在理论上认为多余的细节信号，从而达到减少数据量的目的。实际上AC3的编码技术相当优秀，在如此低的数据流量下带给观众的依然是较为真实的声音体验。

M4A：.M4A 是MPEG4音频标准的文件的扩展名。在MPEG4标准中提到，普通的MPEG4文件扩展名是.mp4。自从Apple开始在它的iTunes以及 iPod中使用m4a以区别MPEG4的视频和音频文件以来，.m4a这个扩展名变得流行了。目前，几乎所有支持MPEG4音频的软件都支持.m4a。最常用的.m4a文件是使用AAC编码格式的（文件），不过其他的格式，比如Apple Lossless甚至mp3也可以被放在.m4a容器里。可以安全的把只包含音频的.mp4 文件的扩展名改成.m4a，以便让它能在你喜欢的播放器里播放，反之亦然。

M4B：M4B格式是一种可以在iPod/iPhone及iPad上播放的称为“有声读物”的音频文件格式。主要优点：M4B有声读物是一种为“音频书籍”专门设计的格式。支持书签是它的主要优点。它允许书在任何时候暂停和恢复播放，恢复播放时，音频将从暂停的点开始。这一特点显然是非常重要的，因为有声读物的时间可能长很多。因此，它更经常用于有声读物和podcasts（播客）。

M4R：.M4R iPhone 手机的铃声格式

M4V：.M4V 是一个标准视频文件格式，由苹果公司创造。此种格式为 iPod 、 iPhone 和 PlayStation Portable 所使用，同时此格式基于 MPEG-4 编码第二版。M4V是一种应用于网络视频点播网站和移动手持设备的视频格式，是MP4格式的一种特殊类型，其后缀常为.MP4或.M4V，其视频编码采用H264或H264/AVC，音频编码采用AAC。采用H264高清编码，相比于传统On2 VP6、H263、Divx等，能够以更小的体积实现更高的清晰度。

### 5.2 AAC 音频格式

ACC 的特点：提升的压缩率：可以以更小的档大小获得更高的音质；支持多声道：可提供最多48 个全音域声道；更高的分辨率：最高支持96KHz 的取样频率；提升的解碼效率：译码播放所占的资源更少。AAC 是个大家族，目前已经制定了如下的9 种规格，以适应不同场合的需要：

MPEG-2 AAC LC 低复杂度规格（Low Complexity）--比较简单，没有增益控制，但提高了

编码效率，在中等码率的编码效率以及音质方面，都能找到平衡点

MPEG-2 AAC Main 主规格

MPEG-2 AAC SSR 可变采样率规格（Scaleable Sample Rate）

MPEG-4 AAC LC 低复杂度规格（Low Complexity）------现在的手机比较常见的MP4文件中的音频部份就包括了该规格音频文件

MPEG-4 AAC Main 主规格 ------包含了除增益控制之外的全部功能，其音质最好

MPEG-4 AAC SSR 可变采样率规格（Scaleable Sample Rate）

MPEG-4 AAC LTP 长时期预测规格（Long Term Predicition）

MPEG-4 AAC LD 低延迟规格（Low Delay）

MPEG-4 AAC HE 高效率规格（High Efficiency）-----这种规格适合用于低码率编码

（1）Main（主规格）包含了除增益控制（Gain Control）之外的全部功能，其音质最好。

（2）低复杂度规格（Low Complexity）则使用比较简单的时域噪声修整（Temporal Noise Shaping，TNS）模块，也缺少了预测（Predict ion）和增益控制（Gain Control）模块，以此降低复杂度，提高了编码效率。SSR（Scalable Sampling Rate，可变取样率规格）和LC 规格大体相似，但多了增益控制功能。

（3）MPEG-4 AAC LTP（Long TermPrediction，长时期预测规格）、MPEG-4 AAC LD（Low Delay，低延迟规格）和MPEG－4 AAC HE（High Efficiency，高效率规格）都是用在低码率下编码，尤其是MPEG－4 AAC HE 规格由于高效率高音质（低码率下而言），再加上有Nero AAC 编码器的支持，所以被越来越多地应用到低码率编码种。事实上，Main 规格和LC 规格的音质相差并不大，但是编码效率却相差明显，所以目前使用地最多的当数LC 规格。无怪乎很多编码器甚至只支持LC 规格了！

和MP3 一样，炙手可热的AAC 格式也催生了大量的编码器。包括商业性质的和完全免费的，技术保密的和源码开放的，对消费者提供的和对开发者提供的，各种各样的编码器都有。其中主要的编码器有：（1）FhG：Fraunhofer IIS 研发的权威编码器，拥有很好的音质，可惜现在不对外提供了！

（2）Nero AAC：可能是目前最完美的AAC 编码器了，同时支持LCAAC 和HE AAC 规格。这个编码器是商业性质的，随着包含了Nero Digital 的Nero 6 一起发布，可能很多有刻录机的计算机里已经安装了该软件。Nero AAC 编码器提供了质量最好的VBR LC AAC 格式，同时HE AAC 规格保证了它在低码率下也有不俗的表现。此外，这个编码器还支持多声道编码！

（3）QuickTime／iTune：苹果公司的两款多媒体软件都提供了AAC 编码功能，其编码技术来自杜比实验室（Dolby Laboratories）。起初，它们的编码器功能比较简单，只支持双声道的CBR 模式LC AAC 格式，不过最近苹果刚刚在QuickTime Pro 版中提供了VBR 编码。本文后面将介绍的iTune 尽管还不具备VBR 模式编码，但是它却是目前音质最好的中码率编码器！

（4）FAAC：最好的命令行编码器，只支持LC 规格。其表现中规中矩，最近几个月新版本的发布将其音质提升了一个台阶。这个免费的编码器已经完全能和商业性质的编码器相媲美了，是搭配EAC 的不二选择。

（5）Psytel：这是最早期的命令行编码器，曾经很是风光，其音质在以前是最好的，就是编码速度很慢。可惜的是其作者现在为Nero 工作了，这个编码器也从此停止了更新。因为有相似的FAAC，所以不建议使用它。

（6）CodingTechnologies：该公司的AAC 编码技术已经被Real 公司加入到他们的产品中了。最新的RealProducer 10 和RealPlayer 10 都包含了这个AAC 编码器。

（7）HHI／zPlane （Compaact！）：Compaact！是一款新出的AAC 编码器，和Nero 一样，也是商业性质的。它的特性相当诱人，支持LC 和Main 规格，支持CBR 和VBR 模式，支持多声道高取样率（24bi t／96KHz），还支持命令行操作！

（8）Dolby AAC：杜比公司的AAC 编码器，因为着重在网络广播，所以低码率下优势明显。据称它编码的48kbps 立体声音质比其它AAC 编码器好20％，甚至64kbps 的Dolby AAC 可媲美128kbps 的MP3！

前使用最多的是LC和HE(适合低码率)。流行的Nero AAC编码程序只支持LC，HE，HEv2这三种规格，编码后的AAC音频，规格显示都是LC。HE其实就是AAC（LC）+SBR技术，HEv2就是AAC（LC）+SBR+PS技术；

AAC原始码流（又称为“裸流”）是由一个一个的ADTS frame组成的。其中每个ADTS frame之间通过syncword（同步字）进行分隔。同步字为0xFFF（二进制“111111111111”）。AAC码流解析的步骤就是首先从码流中搜索0x0FFF，分离出ADTS frame；然后再分析ADTS frame的首部各个字段。

AAC能够在一条音轨中包括48条全带宽（直到96khz）音频声道，加上15条低频增强（LFE，限制到120Hz）声道，直到15条数据流并且更多。杜比也参与了AAC的开发，MPEG规格的一部分，及考虑继承MP3。AAC能输出AC-3的任何码率，胜过AC-3，压缩率更高，但技术上更加复杂。AAC在5.1声道中以400 Kbps和双声道中以180 Kbps输出，就已经能提供良好的聆听效果。

#### 5.2.1参考链接

1.ADTS头解析

https://blog.csdn.net/jay100500/article/details/52955232

### 5.3 AAC音频文件格式

#### 5.3.1 ADIF ＆ ADTS

AAC的音频文件格式有ADIF ＆ ADTS：

ADIF：Audio Data Interchange Format 音频数据交换格式。这种格式的特征是可以确定的找到这个音频数据的开始，不需进行在音频数据流中间开始的解码，即它的解码必须在明确定义的开始处进行。故这种格式常用在磁盘文件中。

ADTS：Audio Data Transport Stream 音频数据传输流。这种格式的特征是它是一个有同步字的比特流，解码可以在这个流中任何位置开始。它的特征类似于mp3数据流格式。

简单说，ADTS可以在任意帧解码，也就是说它每一帧都有头信息。ADIF只有一个统一的头，所以必须得到所有的数据后解码。且这两种的header的格式也是不同的，目前一般编码后的和抽取出的都是ADTS格式的音频流。

#### 5.3.2 ADIF

AC的ADIF格式见下图：

IMG_256

ADIF 的头信息：



ADIF头信息位于AAC文件的起始处，接下来就是连续的 raw data blocks。

 组成ADIF头信息的各个域如下所示：

#### 5.3.3 ADTS

ADTS格式如下：

AAC的ADTS的一般格式见下图：

IMG_257

 图中表示出了ADTS一帧的简明结构，其两边的空白矩形表示一帧前后的数据。

ADTS 的固定头信息：



syncword ：总是0xFFF, 代表一个ADTS帧的开始, 用于同步.解码器可通过0xFFF确定每个ADTS的开始位置.因为它的存在，解码可以在这个流中任何位置开始, 即可以在任意帧解码。

ID：MPEG Version: 0 for MPEG-4，1 for MPEG-2

Layer：always: '00'

protection\_absent：Warning, set to 1 if there is no CRC and 0 if there is CRC

profile：表示使用哪个级别的AAC，如01 Low Complexity(LC) -- AAC LCprofile的值等于 Audio Object Type的值减1.profile = MPEG-4 Audio Object Type - 1



sampling\_frequency\_index：采样率的下标



channel\_configuration：声道数，比如2表示立体声双声道



ADTS的可变头信息：



aac\_frame\_length：一个ADTS帧的长度包括ADTS头和AAC原始流。frame length, this value must include 7 or 9 bytes of header length:

aac\_frame\_length = (protection\_absent == 1 ? 7 : 9) + size(AACFrame)

protection\_absent=0时, header length=9bytes

protection\_absent=1时, header length=7bytes

adts\_buffer\_fullness：0x7FF 说明是码率可变的码流。

number\_of\_raw\_data\_blocks\_in\_frame：表示ADTS帧中有number\_of\_raw\_data\_blocks\_in\_frame + 1个AAC原始帧。所以说number\_of\_raw\_data\_blocks\_in\_frame == 0 表示说ADTS帧中有一个AAC数据块。

(一个AAC原始帧包含一段时间内1024个采样及相关数据)

（1）帧同步目的在于找出帧头在比特流中的位置，13818-7规定，aac ADTS格式的帧头同步字为12比特的“1111 1111 1111”.

（2）ADTS的头信息为两部分组成，其一为固定头信息，紧接着是可变头信息。固定头信息中的数据每一帧都相同，而可变头信息则在帧与帧之间可变。

AAC文件处理流程:

(1)判断文件格式，确定为ADIF或ADTS

(2)若为ADIF，解ADIF头信息，跳至第6步。

(3)若为ADTS，寻找同步头。

(4)解ADTS帧头信息。

(5)若有错误检测，进行错误检测。

(6)解块信息。

(7)解元素信息。

AAC解码流程图:



在主控模块开始运行后，主控模块将AAC比特流的一部分放入输入缓冲区，通过查找同步字得到一帧的起始，找到后，根据ISO/IEC 13818-7所述的语法开始进行Noisless Decoding(无噪解码)，无噪解码实际上就是哈夫曼解码，通过反量化(Dequantize)、联合立体声（JointStereo），知觉噪声替换（PNS）,瞬时噪声整形（TNS），反离散余弦变换（IMDCT），频段复制（SBR）这几个模块之后，得出左右声道的PCM码流，再由主控模块将其放入输出缓冲区输出到声音播放设备。

### 5.4 AAC 与 MP3比较

作为一种高压缩比的音频压缩算法，远胜MP3；在音质方面，由于采用多声道，和使用低复杂性的描述方式，使其比几乎所有的传统编码方式在同规格的情况下更胜一筹。一般来说，AAC可以在对比MP3文件缩小30%的前题下提供更好的音质。AAC是目前唯一一个，能够在所有的EBU试听测试项目的获得“优秀”的网络广播格式。

AAC与MP3规格对比

比特率：AAC – 最高超过400kbps /MP3 – 32~320kbps

采样率：AAC – 最高96kHz / MP3– 最高48kHz

声道数：AAC – （5.1）六声道 / MP3 – 两声道

采样精度：AAC – 最高32bit / MP3– 最高16bit

AAC格式扩展名

.AAC – 基于MPEG-2的音频编码技术，属于传统的AAC编码。

.MP4 – 基于MPEG-4的音频编码技术。

.M4A – 苹果(Apple)公司对纯音频MP4文件采用的扩展名，本质和音频MP4相同。（包含视频的MP4文件为”.M4V”）

## AVI封装格式

### 8.1 AVI概述

AVI是音频视频交错（Audio Video Interleaved）的英文缩写，它是Microsoft公司开发的一种符合RIFF文件规范的数字音频与视频文件格式。AVI格式允许视频和音频交错在一起同步播放，支持256色和RLE压缩，但AVI文件并未限定压缩标准，因此，AVI文件格式只是作为控制界面上的标准，不具有兼容性，用不同压缩算法生成的AVI文件，必须使用相应的解压缩算法才能播放出来。

在介绍AVI文件前，我们要先来看看RIFF文件结构。AVI文件采用的是RIFF文件结构方式，RIFF(Resource Interchange File Format,资源互换文件格式)是微软公司定义的一种用户管理Windows环境中多媒体数据的文件格式，波形音频wave,MIDI和数字视频AVI都采用这种格式存储。RIFF文件的实际数据中，使用了列表（List）和块（Chunk）的形式来组织。列表可以嵌套列表和块。整个RIFF文件可以看成一个数据库，其数据块ID为RIFF ，称为RIFF块。一个RIFF文件中只允许存在一个RIFF块。RIFF块中包含一系列的子块，其中有一种子块的ID为“List”，称为LIST块，LIST块中可以再包含一系列的子块，但除了LIST块外的其他所有的子块都不能再包含子块。 RIFF和LIST块分别比普通的数据块多了一个被称为形式类型（Form Type）或者列表类型（List Type）的数据域。整体就是RIFF-len-”avi”-LIST-len-”hdrl”-avih-strlaud-strh-strfh264-strl-strh-strf

RIFF块（RIFF结构）

“RIFF”表示字符串（Chunk ID）

RIFF文件大小（数据块的大小）

形式类型或者列表类型“AVI”,”WAVE” …….



RIFF文件大小=实际数据长度+4（形式类型或者列表类型的大小）。 也就是说RIFF文件大小不包括“RIFF”域和“文件大小”域本身的大小（共8个字节），如文件大小域数据为28 69 6B 00,则实际为0x006b6928，转换为十进制为7039272，实际在Windows系统下看到的文件大小为7039280，即换算成实际大小时要加上8。可能文件大小值是低字节在前。

LIST块（LIST结构）

“LIST”表示字符串（Chunk ID）

LIST块大小

形式类型或者列表类型

LIST实际数据



LIST块大小域=实际的列表数据+4（形式类型或者列表类型域的大小）也就是说listSize值不包括“LIST”域和listSize域本身的大小。如该位置数据为5E 00 00 00 ,即0x0000005E，转换成十进制为94，实际总长为102bytes,即换算成列表总长度是要加上8。但是有一点要特别注意的是，当前List具体包括到哪里，可能会有List嵌套。如截图中，选中的部分为一个hdrl的List；其中包含两个strl的List，一个音频，一个视频；每个strl List又包含一个strh和一个strf。

Chunk块（Chunk结构，普通块）

Chunk ID——表示块类型的四字符码

Chunk 块大小——记录了整个块的大小

Chunk数据

Chunk块大小=实际的块数据长度，而不包括ckID域和ckSize域本身的大小



从00000000-000007F3为一个WindowsAVI文件的信息区部分：

它是文件的第一个LIST块。在它的内部记录着整个文件的系统构成，如告诉播放软件“我是一个AVI文件”，“在我体内有几个数据流”，“每个数据流包含着什么数据——图像，声音或其他”，“如果是图像数据流，那么它的大小，颜色，压缩方式，播放速度等等是怎样规定的”，“如果是声音数据流，那么它的压缩方式，播放效果等等又将有何规定”……在这些信息区中还有多个附属的LIST块，也就是前面提到的“子块”，它们用来记录每个数据流的全部信息。而这些附属LIST块与数据流之间保持着一一对应的关系，即第一个附属LIST块对应与00号数据流；第二个附属LIST块对应01号数据流……要想解释数据流，我们必须先了解AVI文件中数据块是什么。在AVI文件中，数据块是被放置在数据区中的一个有起始标识（由“数据流识别码”和“数据块存储方式识别码”组成，请参见对数据区部分的说明），并指明大小和数据内容的数据段，那么，数据流就是那些相互之间具有练习的同种数据类型的数据块集合。

00000000-00000003、多媒体文件识别码：RIFF

00000004-00000007、文件大小（10EDICH字节）——8字节

00000008-0000000B、AVI文件识别码

0000000C-0000000F、第一个LIST块识别码

00000010-00000013、第一个LIST块的大小（168H字节）

00000014-00000017、hdrl部分识别码，代表后面的数据记录着此文件的格式

00000018-0000001B、hdrl部分所包含的avih块识别码，此模块记录着本文件的初始化信息

0000001C-0000001F、avi块大小（38H字节）

00000020-00000023、每帧画面显示所维持多少个百万分之一秒，本例为1046h，即6667百万分之一秒约0.07秒。所以在播放此文件时，你看到的画面约每秒15帧

### 8.2 AVI Layout(AVI文件层次划分)

AVI Layout(AVI文件层次划分)



**1. 信息块——**包括文件的通用信息，定义数据格式，所用的压缩算法等参数

**2. 数据块——**包含实际数据流，即图像和声音序列数据。这是文件的主体，也是决定文件容量的主要部分。视频文件的大小等于该文件的数据率乘以该视频播放的时间长度

**3. 索引块——**索引块包含数据块列表好它们在文件中的位置，以提供文件内数据随机存取能力。

#### 6.2.1 信息块

AVI文件时目前使用的最复杂的RIFF文件，他能同时存储同步表现的音频视频数据。AVI的RIFF块的形式类型是AVI，它包含3个子块，如下所述。

1. 信息块：ID为“hdrl”的LIST块，定义AVI文件的数据格式 “hdrl”LIST块包含两个子块，一个是ID为“avih”的子块和一个是ID为”strl”的LIST块。

**“avih”块结构：**用于记录AVI文件的全局信息，比如流的数量，视频图像的宽和高等：

typedef struct

{

DWORD ChunID; // 必须为'avih'

DWORD ChunkSize; //本数据结构的大小，不包括最初的8個位元組（ID和Size兩個域）

DWORD dwMicroSecPerFrame ; //显示每帧所需的时间ns，定义avi的显示速率

DWORD dwMaxBytesPerSec; //最大的数据传输率

DWORD dwPaddingGranularity; //记录块的长度需为此值的倍数，通常是2048

DWORD dwFlages; //AVI文件的特殊属性，如是否包含索引块，音视频数据是否交叉存储

DWORD dwTotalFrame; //文件中的总帧数

DWORD dwInitialFrames; //说明在开始播放前需要多少桢

DWORD dwStreams; //文件中包含的数据流个数

DWORD dwSuggestedBufferSize; //建议使用的缓冲区的大小，

//通常为存储一桢图像以及同步声音所需要的数据之和

DWORD dwWidth; //图像宽

DWORD dwHeight; //图像高

DWORD dwReserved[4]; //保留值

}MainAVIHeader;

接着就是”avih”块的数据部分：一个或多个“strl”子列表

1. 文件中有多少个流，这里就对应有多少个“strl”子列表）。

2. 每个“strl”字列表至少包含一个“strh”块和一个“strf”块，

3. “strd”块（保存编解码器需要的一些配置信息）和“strn”块（保存流的名字）是可选的。

4. 注意：“strl”子列表出现的顺序与媒体流的编号是对应的，比如第一个“strl”字列表说明的是第一个流（Stream 0）,第二个“strl”字列表说明的是第二个流（Stream 1），以此类推。

**“strh”块结构**：用于说明这个“strl”LIST块对应的数据流的头信息：

typedef struct

{

　FOURCC fccType; //4字节，表示数据流的种类，vids 表示视频数据流，auds 音频数据流

　FOURCC fccHandler; //4字节 ，表示数据流解压缩的驱动程序代号

　DWORD dwFlags; //数据流属性

　WORD wPriority; //此数据流的播放优先级

　WORD wLanguage; //音频的语言代号

　DWORD dwInitalFrames; //说明在开始播放前需要多少桢

　DWORD dwScale; //数据量，视频每桢的大小或者音频的采样大小

　DWORD dwRate; //dwScale /dwRate = 每秒的采样数

　DWORD dwStart; //数据流开始播放的位置，以dwScale为单位

　DWORD dwLength; //数据流的数据量，以dwScale为单位

　DWORD dwSuggestedBufferSize; //建议缓冲区的大小

　DWORD dwQuality; //解压缩质量参数，值越大，质量越好

　DWORD dwSampleSize; //音频的采样大小

　RECT rcFrame; //视频图像所占的矩形

}AVIStreamHeader;

**“strf” 块结构：**“strf”子块紧跟在“strh”子块之后，其结构是“strh”子块的类型而定，如下所述：

如果strh子块是视频数据流，则strf子块的内容是一个BITMAPINFO结构，如下。

typedef struct tagBITMAPINFO

{

　BITMAPINFOHEADER bmiHeader;

　RGBQUAD bmiColors[1]; //颜色表

}BITMAPINFO;

typedef struct tagBITMAPINFOHEADER

{

　DWORD biSize;

　LONG biWidth;

　LONG biHeight;

　WORD biPlanes;

　WORD biBitCount;

　DWORD biCompression;

　DWORD biSizeImage;

　LONG biXPelsPerMeter;

　LONG biYPelsPerMeter;

　DWORD biClrUsed;

　DWORD biClrImportant;

}BITMAPINFOHEADER;

如果strh子块是音频数据流，则strf子块的内容是一个WAVEFORMAT结构，如下：

typedef struct

{

　WORD wFormatTag;

　WORD nChannels; //声道数

　DWORD nSamplesPerSec; //采样率

　DWORD nAvgBytesPerSec; //WAVE声音中每秒的数据量

　WORD nBlockAlign; //数据块的对齐标志

　WORD biSize; //此结构的大小

}WAVEFORMAT

**strd”块结构：**“strd”子块紧跟在strf子块后，存储供压缩驱动程序使用的参数，不一定存在，也没有固定的结构。

“strl”List块定义的AVI数据流依次将“hdrl”LIST块中的数据流头结构与“movi”LIST块中的数据联系在一起，第一个数据流头结构用于数据流0，第二个用于数据流1，依次类推。

#### 6.2.2 数据块

二. 数据块：ID为“movi”的LIST块，包含AVI的音视频序列数据

用于保存真正的媒体流数据（视频图像帧数据或音频采样数据等）。保存方式为：

1. 将数据块直接嵌套在“movi”列表里面

2. 将几个数据块分组成一个“rec”列表后再编排进“movi”列表

（注意：在读取AVI文件内容时，建议将一个“rec”列表中的所有数据块一次性读出）

但是，当文件中包含有多个流的时候，数据块与数据块之间如何来区别呢？于是数据块使用了一个四字符吗来表征它的类型，这个四字符码由2个字节的类型吗和2个字节的流编号组成。

“db”——非压缩视频

“dc”——压缩视频

“pc”——改用新的调色板

“wb”——音缩视频

比如:

第一个流（Stream 0）是音频，则表征音频数据块的四字符码为“00wb”;

第二个流（Steam 1）是视频，则表征视频数据块的四字符码为“01db”或“01dc”。

对于视频数据来说，在AVI数据序列中间还可以定义一个新的调色板，每个改变的调色板数据块永“xxpc”来表征，新的调色板使用一个数据结构AVIPALCHANGE来定义。（注意：如果一个流的调色板中途改变，则应在这个流格式的描述中，也及时AVISTREMAHEADER结构的dwFlags中包含一个AVISF\_VIDEO\_PALCHANGES标记）另外，文字数据块可以使用随意的类型码表征。

#### 6.2.3 索引块

三. 索引块：ID为“idxl”的子块，定义“movi”LIST块的索引数据，是可选块。

最后紧跟在“hdr”列表块和“movi”列表块之后的，就是AVI文件可选的索引块。这个索引块为AVI文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于“movi”列表，也可能相对于AVI文件开头）。索引块使用一个四字符码“idxl”来表征，索引信息使用一个数据结构AVIOLDINDEXl来定义。

typedef struct \_avioldindex {

FOURCC fcc; // 必须为‘idx1’

DWORD cb; // 本数据结构的大小，不包括最初的8个字节（fcc和cb两个域）

struct \_avioldindex\_entry {

DWORD dwChunkId; // 表征本数据块的四字符码

DWORD dwFlags; // 说明本数据块是不是关键帧、是不是‘rec ’列表等信息

DWORD dwOffset; // 本数据块在文件中的偏移量

DWORD dwSize; // 本数据块的大小

} aIndex[]; // 这是一个数组！为每个媒体数据块都定义一个索引信息

} AVIOLDINDEX;

注意：如果一个AVI文件包含有索引块，则应在AVI信息头的描述中，也及时AVIMAINHEADER结构的dwFlags中包含一个AVI\_HASINDEX标记

还有一种特殊的数据块，用一个四字符码“JUNK”来表征，它用于内部数据的对齐（填充），应用程序应该忽略这些数据块的实际意义。

#### 6.2.4 索引解释

假如要将avi文件分成头、中、尾的话，RRFT开始到movi段之间应该算是头，movi算是中间，index就算是尾部了。index其实就是movi中的内容的索引，作用是在拖动视频进度条时，能让解码器迅速定位到要找的视频帧。索引以“idx1”开头，后面跟着的是整个索引的大小，其基本结构是：

struct AVI\_index\_entry{  
     unsigned char dwChunkId[4];//表征本数据块的四字符码比如01db压缩视频 00wb音频  
     DWORD dwFlags;//说明本数据块是不是关键帧、是不是‘rec’列表等信息  
     DWORD dwOffset;//本数据块在文件中的偏移量  
     DWORD dwSize;//本数据块的大小  
};

idx1”就是索引的头

30 5b 00 00是索引的大小

大家看看实例的第二行，它们对应avi文件中的每一帧或者没一段音频：

30 30 64 63（“00dc”）是数据结构中的dwChunkId，当该段数据时音频时取（“01wb")

10 00 00 00 是dwFlags，关键帧取值为10 00 00 00，否则取值为00 00 00 00

04 00 00 00 是dwOffset;//本数据块在文件中的偏移量

BF 0E 00 00 是dwSize，对应帧的大小

　　在最初合成avi视频的时候，我是没有添加index段的，视频一样可以播放。可是当你要拖动视频时，视频不会立刻跳到你所要的位置，而是加快播放速度，直到到达目标位置。

目前遇到一个问题：当我将索引加进去avi以后，播放器需要有一个缓冲的过程，有点像用迅雷下载电影，可是没等电影下完，改过后缀就去播放的状况。问题解决以后再来补充吧。

经过对avi视频的详细分析，发现音频视频的交叉有如下规律：

15帧 vedio

1段 audio

15帧 vedio

1段 audio

16帧 vedio

1段 audio

15帧 vedio

1段 audio

15帧 vedio

1段 audio

16帧 vedio

1段 audio

也就是基本上是每15段vedio就有一段的audio，可是每经过两个15段的vedio循环，就要有16段的vedio才能有一段audio。另外还有一个规律就是，每一段audio大小都是一定的4096字节。这是某个特定文件的交叉规律，个人认为音视频交叉的频率跟帧率和音频频率有关，只要是按一定的比率来交叉就行，例如上面按每30帧视频放一段音频，然后音频的长度变为8192应该可以，有空再试试。

#### 6.2.5 参考链接

1.AVI文件格式详解

<https://blog.csdn.net/chenyonken/article/details/79174500>

2.avi文件格式解析

<https://blog.csdn.net/occupy8/article/details/40678471>

## MP4封装格式

### 7.1 MP4概述

最开始MP4指的是音频（MP3的升级版），即MPEG-2 AAC标准。随后MP4概念被转移到视频上，对应的是MPEG-4标准。而现在我们流行的叫法，多半是指能播放MPEG-4标准编码格式视频的播放器。但是这篇文章介绍的内容跟上面这些都无关，我们要讨论的是MP4文件封装格式，对应的标准为ISO/IEC 14496-12，即信息技术 视听对象编码的第12部分：ISO 基本媒体文件格式（Information technology Coding of audio-visual objects Part 12: ISO base media file format）。ISO/IEC组织指定的标准一般用数字表示，ISO/IEC 14496即MPEG-4标准。

MP4视频文件封装格式是基于QuickTime容器格式定义的，因此参考QuickTime的格式定义对理解MP4文件格式很有帮助。MP4文件格式是一个十分开放的容器，几乎可以用来描述所有的媒体结构，MP4文件中的媒体描述与媒体数据是分开的，并且媒体数据的组织也很自由，不一定要按照时间顺序排列，甚至媒体数据可以直接引用其他文件。同时，MP4也支持流媒体。MP4目前被广泛用于封装h.264视频和AAC音频，是高清视频的代表。

#### 参考链接

1.MP4文件格式

<https://www.cnblogs.com/ranson7zop/p/7889272.html>

2.MP4文件格式解析

<https://www.cnblogs.com/boonya/p/8572052.html>

3.mp4文件格式解析

<https://www.jianshu.com/p/529c3729f357>

### 7.2 MP4结构

MP4文件中的所有数据都装在box（QuickTime中为atom）中，也就是说MP4文件由若干个box组成，每个box有类型和长度，可以将box理解为一个数据对象块。box中可以包含另一个box，这种box称为container box。一个MP4文件首先会有且只有一个“ftyp”类型的box，作为MP4格式的标志并包含关于文件的一些信息；之后会有且只有一个“moov”类型的box（Movie Box），它是一种container box，子box包含了媒体的metadata信息；MP4文件的媒体数据包含在“mdat”类型的box（Midia Data Box）中，该类型的box也是container box，可以有多个，也可以没有（当媒体数据全部引用其他文件时），媒体数据的结构由metadata进行描述。

下面是一些概念：

**track**表示一些sample的集合，对于媒体数据来说，track表示一个视频或音频序列。

**hint track** 这个特殊的track并不包含媒体数据，而是包含了一些将其他数据track打包成流媒体的指示信息。

**sample**对于非hint track来说，video sample即为一帧视频，或一组连续视频帧，audio sample即为一段连续的压缩音频，它们统称sample。对于hint track，sample定义一个或多个流媒体包的格式。

**sample table**指明sampe时序和物理布局的表。

**chunk** 一个track的几个sample组成的单元。

在本文中，我们不讨论涉及hint的内容，只关注包含媒体数据的本地MP4文件。下图为一个典型的MP4文件的结构树。



首先需要说明的是，box中的字节序为网络字节序，也就是大端字节序（Big-Endian），简单的说，就是一个32位的4字节整数存储方式为高位字节在内存的低端。Box由header和body组成，其中header统一指明box的大小和类型，body根据类型有不同的意义和格式。

标准的box开头的4个字节（32位）为box size，该大小包括box header和box body整个box的大小，这样我们就可以在文件中定位各个box。如果size为1，则表示这个box的大小为large size，真正的size值要在largesize域上得到。（实际上只有“mdat”类型的box才有可能用到large size。）如果size为0，表示该box为文件的最后一个box，文件结尾即为该box结尾。（同样只存在于“mdat”类型的box中。）

size后面紧跟的32位为box type，一般是4个字符，如“ftyp”、“moov”等，这些box type都是已经预定义好的，分别表示固定的意义。如果是“uuid”，表示该box为用户扩展类型。如果box type是未定义的，应该将其忽略。

**File Type Box（ftyp）**：该box有且只有1个，并且只能被包含在文件层，而不能被其他box包含。该box应该被放在文件的最开始，指示该MP4文件应用的相关信息。“ftyp” body依次包括1个32位的major brand（4个字符），1个32位的minor version（整数）和1个以32位（4个字符）为单位元素的数组compatible brands。这些都是用来指示文件应用级别的信息。该box的字节实例如下：

IMG_256

**Movie Box（moov）**：该box包含了文件媒体的metadata信息，“moov”是一个container box，具体内容信息由子box诠释。同File Type Box一样，该box有且只有一个，且只被包含在文件层。一般情况下，“moov”会紧随“ftyp”出现。一般情况下（限于篇幅，本文只讲解常见的MP4文件结构），“moov”中会包含1个“mvhd”和若干个“trak”。其中“mvhd”为header box，一般作为“moov”的第一个子box出现（对于其他container box来说，header box都应作为首个子box出现）。“trak”包含了一个track的相关信息，是一个container box。下图为部分“moov”的字节实例，其中红色部分为box header，绿色为“mvhd”，黄色为一部分“trak”。



**Movie Header Box（mvhd）：**“mvhd”结构如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 mvhd大小 |
| box type | 4 | box类型 “mvhd” |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| creation time | 4 | 创建时间（相对于UTC时间1904-01-01零点的秒数） |
| modification time | 4 | 修改时间 |
| time scale | 4 | 文件媒体在1秒时间内的刻度值，可以理解为1秒长度的时间单元数 |
| duration | 4 | 该track的时间长度，用duration和time scale值可以计算track时长，比如audio track的time scale = 8000, duration = 560128，时长为70.016，video track的time scale = 600, duration = 42000，时长为70 |
| rate | 4 | 推荐播放速率，高16位和低16位分别为小数点整数部分和小数部分，即[16.16] 格式，该值为1.0（0x00010000）表示正常前向播放 |
| volume | 2 | 与rate类似，[8.8] 格式，1.0（0x0100）表示最大音量 |
| reserved | 10 | 保留位 |
| matrix | 36 | 视频变换矩阵 |
| pre-defined | 24 |  |
| next track id | 4 | 下一个track使用的id号 |

“mvhd”的字节实例如下图，各字段已经用颜色区分开：



**Track Box（trak）：**“trak”也是一个container box，其子box包含了该track的媒体数据引用和描述（hint track除外）。一个MP4文件中的媒体可以包含多个track，且至少有一个track，这些track之间彼此独立，有自己的时间和空间信息。“trak”必须包含一个“tkhd”和一个“mdia”，此外还有很多可选的box（略）。其中“tkhd”为track header box，“mdia”为media box，该box是一个包含一些track媒体数据信息box的container box。 “trak”的部分字节实例如下图，其中黄色为“trak”box的头，绿色为“tkhd”，蓝色为“edts”（一个可选box），红色为一部分“mdia”。



**Track Header Box（tkhd）：**“tkhd”结构如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 **tkhd大小** |
| box type | 4 | box类型 “**tkhd”** |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 | 按位或操作结果值，预定义如下：  0x000001 track\_enabled，否则该track不被播放；  0x000002 track\_in\_movie，表示该track在播放中被引用；  0x000004 track\_in\_preview，表示该track在预览时被引用。  一般该值为7，如果一个媒体所有track均未设置track\_in\_movie和track\_in\_preview，将被理解为所有track均设置了这两项；对于hint track，该值为0 |
| creation time | 4 | 创建时间（相对于UTC时间1904-01-01零点的秒数） |
| modification time | 4 | 修改时间 |
| track id | 4 | id号，不能重复且不能为0 |
| reserved | 4 | 保留位 |
| duration | 4 | track的时间长度 |
| reserved | 8 | 保留位 |
| layer | 2 | 视频层，默认为0，值小的在上层 |
| alternate group | 2 | track分组信息，默认为0表示该track未与其他track有群组关系 |
| volume | 2 | [8.8] 格式，如果为音频track，1.0（0x0100）表示最大音量；否则为0 0要不音量为0要不就是指定track视频 |
| reserved | 2 | 保留位 |
| matrix | 36 | 视频变换矩阵 |
| width | 4 | 宽 |
| height | 4 | 高，均为 [16.16] 格式值，与sample描述中的实际画面大小比值，用于播放时的展示宽高 |

“tkhd”的字节实例如下图，各字段已经用颜色区分开：



**Media Box（mdia）：** “mdia”也是个container box，其子box的结构和种类还是比较复杂的。先来看一个“mdia”的实例结构树图。

总体来说，“mdia”定义了track媒体类型以及sample数据，描述sample信息。一般“mdia”包含一个“mdhd”，一个“hdlr”和一个“minf”，其中“mdhd”为media header box，“hdlr”为handler reference box，“minf”为media information box。下面依次看一下这几个box的结构。

**Media Header Box（mdhd）：**“mdhd”结构如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 |
| box type | 4 | box类型 |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| creation time | 4 | 创建时间（相对于UTC时间1904-01-01零点的秒数） |
| modification time | 4 | 修改时间 |
| time scale | 4 | 同前表 |
| duration | 4 | track的时间长度 |
| language | 2 | 媒体语言码。最高位为0，后面15位为3个字符（见ISO 639-2/T标准中定义） |
| pre-defined | 2 |  |

“mdhd”的字节实例如下图，各字段已经用颜色区分开：

IMG_256

**Handler Reference Box（hdlr）**：“hdlr”解释了媒体的播放过程信息，该box也可以被包含在meta box（meta）中。“hdlr”结构如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 |
| box type | 4 | box类型 |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| pre-defined | 4 |  |
| handler type | 4 | 在media box中，该值为4个字符：  “vide”— video track  “soun”— audio track  “hint”— hint track |
| reserved | 12 |  |
| name | 不定 | track type name，以‘\0’结尾的字符串 |

“hdlr”的字节实例如下图，各字段已经用颜色区分开：



**Media Information Box（minf）**：“minf”存储了解释track媒体数据的handler-specific信息，media handler用这些信息将媒体时间映射到媒体数据并进行处理。“minf”中的信息格式和内容与媒体类型以及解释媒体数据的media handler密切相关，其他media handler不知道如何解释这些信息。“minf”是一个container box，其实际内容由子box说明。一般情况下，“minf”包含一个header box，一个“dinf”和一个“stbl”，其中，header box根据track type（即media handler type）分为“vmhd”、“smhd”、“hmhd”和“nmhd”，“dinf”为data information box，“stbl”为sample table box。下面分别介绍。下图为“minf”部分字节实例，其中红色为box header，蓝色为“smhd”，绿色为“dinf”，黄色为一部分“stbl”。



**Media Information Header Box（vmhd、smhd、hmhd、nmhd）**

Video Media Header Box（vmhd）

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 |
| box type | 4 | box类型 |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| graphics mode | 4 | 视频合成模式，为0时拷贝原始图像，否则与opcolor进行合成 |
| opcolor | 2×3 | ｛red，green，blue｝ |

Sound Media Header Box（smhd）

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 |
| box type | 4 | box类型 |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| balance | 2 | 立体声平衡，[8.8] 格式值，一般为0，-1.0表示全部左声道，1.0表示全部右声道 |
| reserved | 2 |  |

Hint Media Header Box（hmhd）

略

Null Media Header Box（nmhd）

非视音频媒体使用该box，略。

**Data Information Box（dinf）**：“dinf”解释如何定位媒体信息，是一个container box。“dinf”一般包含一个“dref”，即data reference box；“dref”下会包含若干个“url”或“urn”，这些box组成一个表，用来定位track数据。简单的说，track可以被分成若干段，每一段都可以根据“url”或“urn”指向的地址来获取数据，sample描述中会用这些片段的序号将这些片段组成一个完整的track。一般情况下，当数据被完全包含在文件中时，“url”或“urn”中的定位字符串是空的。

“dref”的字节结构如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 字节数 | 意义 |
| box size | 4 | box大小 |
| box type | 4 | box类型 |
| version | 1 | box版本，0或1，一般为0。（以下字节数均按version=0） |
| flags | 3 |  |
| entry count | 4 | “url”或“urn”表的元素个数 |
| “url”或“urn”列表 | 不定 |  |

“url”或“urn”都是box，“url”的内容为字符串（location string），“urn”的内容为一对字符串（name string and location string）。当“url”或“urn”的box flag为1时，字符串均为空。

下面是一个“dinf”的字节实例图。其中黄色为“dinf”的box header，由红色部分我们知道包含的“url”或“urn”个数为1，红色后面为“url”box的内容。紫色为“url”的box header（根据box type我们知道是个“url”），绿色为box flag，值为1，说明“url”中的字符串为空，表示track数据已包含在文件中。

IMG_256

**Sample Table Box（stbl）**： “stbl”几乎是普通的MP4文件中最复杂的一个box了，首先需要回忆一下sample的概念。sample是媒体数据存储的单位，存储在media的chunk中，chunk和sample的长度均可互不相同，如下图所示。



“stbl”包含了关于track中sample所有时间和位置的信息，以及sample的编解码等信息。利用这个表，可以解释sample的时序、类型、大小以及在各自存储容器中的位置。“stbl”是一个container box，其子box包括：sample description box（stsd）、time to sample box（stts）、sample size box（stsz或stz2）、sample to chunk box（stsc）、chunk offset box（stco或co64）、composition time to sample box（ctts）、sync sample box（stss）等。

“stsd”必不可少，且至少包含一个条目，该box包含了data reference box进行sample数据检索的信息。没有“stsd”就无法计算media sample的存储位置。“stsd”包含了编码的信息，其存储的信息随媒体类型不同而不同。

**Sample Description Box（stsd）**

box header和version字段后会有一个entry count字段，根据entry的个数，每个entry会有type信息，如“vide”、“sund”等，根据type不同sample description会提供不同的信息，例如对于video track，会有“VisualSampleEntry”类型信息，对于audio track会有“AudioSampleEntry”类型信息。视频的编码类型、宽高、长度，音频的声道、采样等信息都会出现在这个box中。



**Time To Sample Box（stts）**

“stts”存储了sample的duration，描述了sample时序的映射方法，我们通过它可以找到任何时间的sample。“stts”可以包含一个压缩的表来映射时间和sample序号，用其他的表来提供每个sample的长度和指针。表中每个条目提供了在同一个时间偏移量里面连续的sample序号，以及samples的偏移量。递增这些偏移量，就可以建立一个完整的time to sample表。



**Sample Size Box（stsz）**

“stsz” 定义了每个sample的大小，包含了媒体中全部sample的数目和一张给出每个sample大小的表。这个box相对来说体积是比较大的。四个字节其实就是每一帧得大小。



**Sample To Chunk Box（stsc）**

用chunk组织sample可以方便优化数据获取，一个thunk包含一个或多个sample。“stsc”中用一个表描述了sample与chunk的映射关系，查看这张表就可以找到包含指定sample的thunk，从而找到这个sample。



**Sync Sample Box（stss）**

“stss”确定media中的关键帧。对于压缩媒体数据，关键帧是一系列压缩序列的开始帧，其解压缩时不依赖以前的帧，而后续帧的解压缩将依赖于这个关键帧。“stss”可以非常紧凑的标记媒体内的随机存取点，它包含一个sample序号表，表内的每一项严格按照sample的序号排列，说明了媒体中的哪一个sample是关键帧。如果此表不存在，说明每一个sample都是一个关键帧，是一个随机存取点。四个字节其实和fps值也有关系。



**Chunk Offset Box（stco）**

“stco”定义了每个thunk在媒体流中的位置。位置有两种可能，32位的和64位的，后者对非常大的电影很有用。在一个表中只会有一种可能，这个位置是在整个文件中的，而不是在任何box中的，这样做就可以直接在文件中找到媒体数据，而不用解释box。需要注意的是一旦前面的box有了任何改变，这张表都要重新建立，因为位置信息已经改变了。



主要归纳为：co64指定每个chunk在媒体流中的位置。Mdata数据即为数据，占绝大空间。Stsc指定chunk中包含多少sample。Stsz指定每个sample得大小。Stss指定关键帧sample得位置即fps值。Stts即dts-sample映射。



ftyp：表明文件类型。

mdat：具体的媒体数据。

moov box：包含一系列的子box， 通过解析这些子box，获得编码信息、数据信息等。

Free Space Box（free或skip）**：**“free”中的内容是无关紧要的，可以被忽略。该box被删除后，不会对播放产生任何影响。

Meida Data Box（mdat）：该box包含于文件层，可以有多个，也可以没有（当媒体数据全部为外部文件引用时），用来存储媒体数据。数据直接跟在box type字段后面，具体数据结构的意义需要参考metadata（主要在sample table中描述）。普通MP4文件的结构就讲完了，可能会比较乱，下面这张图是常见的box的树结构图，可以用来大致了解MP4文件的构造。

### 7.3 MP4术语解释及知识补充

box：由唯一类型标识符和长度定义的面向对象的构件

container box：用来容纳一组相关box的box，container box通常都不是fullbox

chunk：同一轨道的一组连续的采样

hint track：不包含媒体数据，但包含了将一个或多个轨打包到流频道的指示

media data box：用来容纳实体数据的box

movie box：子box定义了元数据（metadata）的容器box

sample：与单个时间戳相关联的所有数据，video sample即为一帧视频，或一组连续视频帧，audio sample即为一段连续的压缩音频

sample description：定义和描述轨中的采样的格式的结构

sample table：指明sampe时序和物理布局的表

track：按时间排序的相关的采样，对于媒体数据来说，track表示一个视频或音频序列

整个box以box header开头，box header中包含了box的大小（size）和类型（type）等信息。其中，size指明了整个box所占用的大小，包括header部分，如果box很大(例如存放具体视频数据的mdat box)，超过了uint32的最大数值，size就被设置为1，并用接下来的8位uint64的largesize来存放大小。box中的字节序为网络字节序，也就是大端字节序（Big-Endian）。

“stbl”包含了关于track中sample所有时间和位置的信息，以及sample的编解码等信息。利用这个表，可以解释sample的时序、类型、大小以及在各自存储容器中的位置。“stbl”是一个container box，其子box包括：sample description box（stsd）、time to sample box（stts）、sample size box（stsz或stz2）、sample to chunk box（stsc）、chunk offset box（stco或co64）、composition time to sample box（ctts）、sync sample box（stss）等。“stsd”必不可少，且至少包含一个条目，该box包含了data reference box进行sample数据检索的信息。没有“stsd”就无法计算media sample的存储位置。“stsd”包含了编码的信息，其存储的信息随媒体类型不同而不同。

### 7.4 Mp4info工具各个参数介绍：

mp4文件基本信息

audio信息：

smplrate：sample rate(采样率)。

channel：通道个数。

bitrate：比特率。

audiosamplenum：音频sample的个数。

video信息：

width、height：视频的宽/高。

bitrate：比特率(码率)，秒为单位。等于视频总的大小/时长。

frames：视频帧数。

fps：帧率(frame per second)。

total\_time：时间长度，ms为单位。等于duration/timescale。

timescale：时间的粒度，1000表示1000个单位为1s。

duration：时间粒度的个数。

videosamplenum：视频sample的个数。

### 7.5MP4中如何seek

我们需要seek到30s。  
需要做如下工作：

1. 使用timescale将目标时间标准化。timescale为90000，30\*90000=2700000。
2. 通过time-to-sample box(stts)找到指定track的给定时间之前的第一个sample number。2700000/3000 = 900。900应该是一帧大概占用多少timescale=90000/100。
3. 通过sync sample table(stss)查询sample number之前的第一个sync sample。对应为795的sample。
4. 通过sample-to-chunk table(stsc)查找到对应的chunk number。对应的chunk号是假设是400。
5. 通过chunk offset box(stco或co64)查找到对应chunk在文件中的起始偏移量。第14个chunk的offset是3481072。
6. 最后使用sample-to-chunk box(stsc)和sample size box(stsz)的信息计算出该chunk中需要读取的sample在文件中的起始偏移量，即完成 seek。

## FLV 封装格式

### 概述

FLV（Flash Video）是现在非常流行的流媒体格式，由于其视频文件体积轻巧、封装播放简单等特点，使其很适合在网络上进行应用，目前主流的视频网站无一例外地使用了FLV格式。另外由于当前浏览器与Flash Player紧密的结合，使得网页播放FLV视频轻而易举，也是FLV流行的原因之一。

参考链接：

1. flv格式详解+实例剖析

<https://www.cnblogs.com/lidabo/p/9018548.html>

1. FLV 封装格式解析

<https://blog.csdn.net/y_z_hyangmo/article/details/79208275>

1. FLV文件格式详解

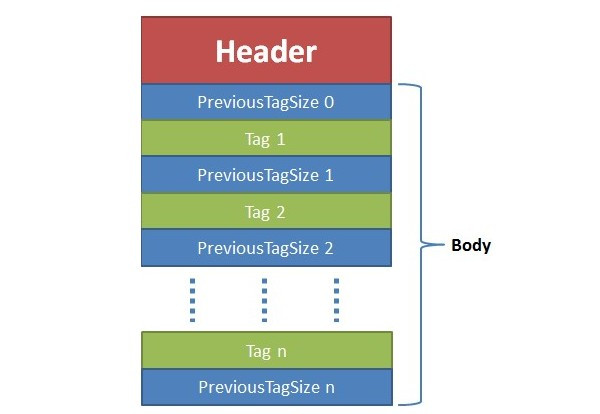
<https://blog.csdn.net/beyond702/article/details/78929334>

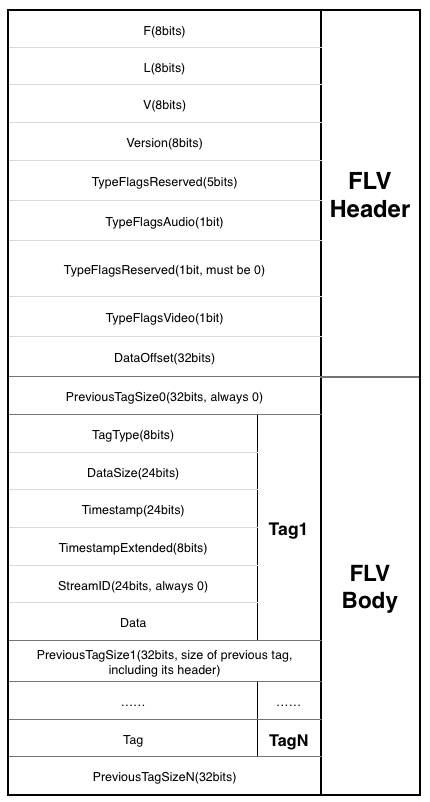
1. FLV文档

<https://www.adobe.com/content/dam/acom/en/devnet/flv/video_file_format_spec_v10_1.pdf>

### FLV文件格式

FLV是流媒体封装格式，我们可以将其数据看为二进制字节流。总体上看，FLV包括文件头（File Header）和文件体（File Body）两部分，其中文件体由一系列的Tag及Tag Size对组成。





#### flv header

Signature: 占三个字节，分别为字符 ‘F’, ‘L’, ‘V’。表明自己的身份。就是FLV文件。如果我们希望探测一个文件是不是FLV文件，检测头三个字节的内容就可以。

Version: 表明当前文件使用的FLV格式的版本，当前版本为Version 1, 所以这个字段就一直为固定值 0x01.

TypeFlags: TypeFlags字段的第0位和第2位用于指示文件包含视频内容和音频内容。

DataOffset: 记录FLV Header结构的大小，包含了DataOffset字段本身。我们计算一下就知道在Version 1当中这个字段的值为 9。

#### 2 flv body

PreviousTagSize: 这个就是上文提到的反向指针-“back-pointer”。指示之前一个TAG的大小。

#### 3 tag

tag type + tag data size + Timestamp + TimestampExtended + stream id + tag data

* type 1个字节。8为Audio,9为Video,18为scripts
* tag data size 3个字节。表示tag data的长度。从streamd id 后算起。
* Timestreamp 3个字节。时间戳
* TimestampExtended 1个字节。时间戳扩展字段
* stream id 3个字节。总是0
* tag data 数据部分

Tag分为三类

script

脚本Tag一般只有一个，是flv的第一个Tag，用于存放flv的信息，比如duration、audiodatarate、creator、width等。首先介绍下脚本的数据类型。所有数据都是以数据类型+（数据长度）+数据的格式出现的，数据类型占1byte，数据长度看数据类型是否存在，后面才是数据。

一般来说，该Tag Data结构包含两个AMF包。AMF（Action Message Format）是Adobe设计的一种通用数据封装格式，在Adobe的很多产品中应用，简单来说，AMF将不同类型的数据用统一的格式来描述。第一个AMF包封装字符串类型数据，用来装入一个“onMetaData”标志，这个标志与Adobe的一些API调用有，在此不细述。第二个AMF包封装一个数组类型，这个数组中包含了音视频信息项的名称和值。

| 值 | 类型 | 说明 |
| --- | --- | --- |
| 0 | Number type | 8 Bypte Double |
| 1 | Boolean type | 1 Bypte bool |
| 2 | String type | 后面2个字节为长度 |
| 3 | Object type |  |
| 4 | MovieClip type |  |
| 5 | Null type |  |
| 6 | Undefined type |  |
| 7 | Reference type |  |
| 8 | ECMA array type | 数组,类似Map |
| 10 | Strict array type |  |
| 11 | Date type |  |
| 12 | Long string type | 后面4个字节为长度 |

type=0x02对应String

size=0A=10

value=onMetaData 正好是10个字节。

Video

type=0x09=9。这里应该是一个video。

size=0x000030=48。长度为48。

timestreamp=0x000000。

TimestampExtended =0x00。

stream id =0x000000

我们看到数据部分：

视频信息+数据

视频信息，1个字节。

前4位为帧类型Frame Type

| **值** | **类型** |
| --- | --- |
| 1 | keyframe (for AVC, a seekable frame) 关键帧 |
| 2 | inter frame (for AVC, a non-seekable frame) |
| 3 | disposable inter frame (H.263 only) |
| 4 | generated keyframe (reserved for server use only) |
| 5 | video info/command frame |

后4位为编码ID (CodecID)

| **值** | **类型** |
| --- | --- |
| 1 | JPEG (currently unused) |
| 2 | Sorenson H.263 |
| 3 | Screen video |
| 4 | On2 VP6 |
| 5 | On2 VP6 with alpha channel |
| 6 | Screen video version 2 |
| 7 | AVC |

特殊情况

视频的格式(CodecID)是AVC（H.264）的话，VideoTagHeader会多出4个字节的信息，AVCPacketType 和CompositionTime。

* AVCPacketType 占1个字节

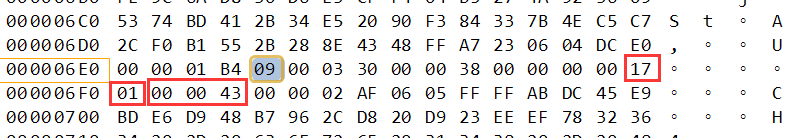
| **值** | **类型** |
| --- | --- |
| 0 | AVCDecoderConfigurationRecord(AVC sequence header) |
| 1 | AVC NALU |
| 2 | AVC end of sequence (lower level NALU sequence ender is not required or supported) |

**AVCDecoderConfigurationRecord**.包含着是H.264解码相关比较重要的sps和pps信息，再给AVC解码器送数据流之前一定要把sps和pps信息送出，否则的话解码器不能正常解码。而且在解码器stop之后再次start之前，如seek、快进快退状态切换等，都需要重新送一遍sps和pps的信息.AVCDecoderConfigurationRecord在FLV文件中一般情况也是出现1次，也就是第一个**video tag**.

* CompositionTime 占3个字节

| **条件** | **值** |
| --- | --- |
| AVCPacketType ==1 | Composition time offset |
| AVCPacketType !=1 | 0 |

我们看第一个video tag，也就是前面那张图。我们看到AVCPacketType =0。而后面三个字节也是0。说明这个tag记录的是AVCDecoderConfigurationRecord。包含sps和pps数据。  
再看到第二个video tag

sps pps

前面我们提到第一个video 一般存放的是sps和pps。这里我们具体解析下sps和pps内容。先看下存储的格式（图6）：

0x01+sps[1]+sps[2]+sps[3]+0xFF+0xE1+sps size+sps+01+pps size+pps

我们看到图7 。

sps[1]=0x64

sps[2]=00

sps[3]=0D

sps size=0x001B=27

跳过27个字节后，是0x01

pps size=0x0005=118

跳过5个字节，就到了back-pointers。

视频帧数据

解析出sps和pps tag后，后面的video tag就是真正的视频数据内容了

Audio

与视频格式类似

前4位为音频格式

| **值** | **类型** |
| --- | --- |
| 0 | Linear PCM, platform endian |
| 1 | ADPCM |
| 2 | MP3 |
| 3 | Linear PCM, little endian |
| 4 | Nellymoser 16-kHz mono |
| 5 | Nellymoser 8-kHz mono |
| 6 | Nellymoser |
| 7 | G.711 A-law logarithmic PCM |
| 8 | G.711 mu-law logarithmic PCM |
| 9 | reserved |
| 10 | AAC |
| 11 | Speex |
| 14 | MP3 8-Khz |
| 15 | Device-specific sound |

接着2位为采样率

值 类型

0 5.5-kHz

1 11-kHz

2 22-kHz

3 44-kHz

对于AAC总是3

接着1位为采样的长度

| **值** | **类型** |
| --- | --- |
| 0 | snd8Bit |
| 1 | snd16Bit |

压缩过的音频都是16bit

接着1位为音频类型

| **值** | **类型** |
| --- | --- |
| 0 | sndMono |
| 1 | sndStereo |

对于AAC总是1

### 10.3FLV分析工具

http://www.flvmeta.com/

yamdi：将flv转成带索引的flv，yamdi -i i.flv -o o.flv

flvlib： pip install flvlib, 查看索引信息：debug-flv --metadata file.flv

flvcheck：<http://www.adobe.com/products/adobe-media-server-family/tool-downloads.html>

## 11 http与音视频千丝万缕的关系

### 11.1 http介绍

### 11.2 http 请求过程

域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 --> 服务器响应http请求，浏览器得到html代码 --> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户。

关于HTTP协议可以参考以下：

HTTP协议漫谈 <http://kb.cnblogs.com/page/140611/>

HTTP协议概览 <http://www.cnblogs.com/vamei/archive/2013/05/11/3069788.html>

了解HTTP Headers的方方面面 <http://kb.cnblogs.com/page/55442/>

浏览器先向服务器发送请求，服务器接收到请求后，做相应的处理，然后封装好响应报文，再回送给浏览器。浏览器拿到响应报文后，再通过 浏览器引擎去渲染网页，解析DOM树，javascript引擎解析并执行脚本操作，插件去干插件该干的事儿...关于浏览器渲染、解析的原理，可以参考http://kb.cnblogs.com/page/129756/

### 11.3 HTTP详解

协议有有以下几种：

http/0.9: stateless

http/1.0: MIME, keep-alive (保持连接), 缓存

http/1.1: 更多的请求方法，更精细的缓存控制，持久连接(persistent connection) 比较常用

起始行中的请求方法有：

GET: 完整请求一个资源 （常用）

HEAD: 仅请求响应首部

POST：提交表单 （常用）

PUT: (webdav) 上传文件（但是浏览器不支持该方法）

DELETE：(webdav) 删除

OPTIONS：返回请求的资源所支持的方法的方法

TRACE: 追求一个资源请求中间所经过的代理（该方法不能由浏览器发出）

http请求消息

一个http请求代表客户端浏览器向服务器发送的数据。一个完整的http请求消息，包含一个请求行，若干个消息头（请求头），换行，实体内容

Accept 用于告诉服务器，客户机支持的数据类型（例如：Accept:text/html,image/\*）就是 告诉服务器端，接受那些MIME类型

Accept-Encoding 这个看来是接受那些压缩方式的文件

Accept-Charset：用于告诉服务器，客户机采用的编码格式

Accept-Lanague 告诉服务器能够发送哪些语言

Connection 告诉服务器支持keep-alive特性

Cookie 每次请求时都会携带上Cookie以方便服务器端识别是否是同一个客户端

Host 用来标识请求服务器上的那个虚拟主机，比如Nginx里面可以定义很多个虚拟主机那这里就是用来标识要访问那个虚拟主机。

User-Agent 用户代理，一般情况是浏览器，也有其他类型，如：wget curl 搜索引擎的蜘蛛等.客户机通过这个头告诉服务器，客户机的软件环境（操作系统，浏览器版本等）

If-Modified-Since 客户机通过这个头告诉服务器，资源的缓存时间

Referer 客户机通过这个头告诉服务器，它（客户端）是从哪个资源来访问服务器的（防盗链）

比如：

User-Agent：Wget/1.11.4 Red Hat Modied\r\n 用户代理wget

Accept：\*/\* \r\n 可接受MIME类型所有类型

Host：[www.XXX.com](http://www.XXX.com)\r\n 主机名

Connection：Keep-Alive\r\n 连接使用keep-alive特性

实体内容：

就是指浏览器端通过http协议发送给服务器的实体数据。例如：name=dylan&id=110

（get请求时，通过url传给服务器的值。post请求时，通过表单发送给服务器的值）

MIME（Multipurpose Internet Mail Extesions 多用途互联网邮件扩展）是一个互联网标准，它扩展了电子邮件标准，使其能够支持非ASCII字符、二进制格式附件等多种格式的邮件消息，这个标准被定义在RFC 2045、RFC 2046、RFC 2047、RFC 2048、RFC 2049等RFC中。 由RFC 822转变而来的RFC 2822，规定电子邮件标准并不允许在邮件消息中使用7位ASCII字符集以外的字符。正因如此，一些非英语字符消息和二进制文件，图像，声音等非文字消息都不能在电子邮件中传输。MIME规定了用于表示各种各样的数据类型的符号化方法。 此外，在万维网中使用的HTTP协议中也使用了MIME的框架，标准被扩展为互联网媒体类型。

MIME 遵循以下格式：major/minor 主类型/次类型 例如：

image/jpg

image/gif

text/html

video/quicktime

appliation/x-httpd-php

状态码

1xx: 信息性状态码

100, 101

2xx: 成功状态码

200：OK

3xx: 重定向状态码

301: 永久重定向, Location响应首部的值仍为当前URL，因此为隐藏重定向;

302: 临时重定向，显式重定向, Location响应首部的值为新的URL

304：Not Modified 未修改，比如本地缓存的资源文件和服务器上比较时，发现并没有修改，服务器返回一个304状态码，告诉浏览器，你不用请求该资源，直接使用本地的资源即可。

4xx: 客户端错误状态码

404: Not Found 请求的URL资源并不存在

5xx: 服务器端错误状态码

500: Internal Server Error 服务器内部错误

502: Bad Gateway 前面代理服务器联系不到后端的服务器时出现

504：Gateway Timeout 这个是代理能联系到后端的服务器，但是后端的服务器在规定的时间内没有给代理服务器响应

http响应信息

Location 这个头配合302状态吗，用于告诉客户端找谁

Content-Length 告诉浏览器，回送数据的长度

Connection 使用keep-alive特性，响应完成后，是否断开连接。close/Keep-Alive

Content-Encoding 使用gzip方式对资源压缩

Content-type MIME类型为html类型，字符集是 UTF-8.告诉浏览器，回送数据类型

Date 响应的日期

Server 使用的WEB服务器,服务器通过这个头，告诉浏览器服务器的类型

Transfer-Encoding:chunked 分块传输编码 是http中的一种数据传输机制，允许HTTP由网页服务器发送给客户端应用（通常是网页浏览器）的数据可以分成多个部分，分块传输编码只在HTTP协议1.1版本（HTTP/1.1）中提供

Vary 这个可以参考（http://blog.csdn.NET/tenfyguo/article/details/5939000）

X-Pingback 参考（http://blog.sina.com.cn/s/blog\_bb80041c0101fmfz.html）

Refresh 告诉浏览器，隔多长时间刷新

Content-Dispositio 告诉浏览器以下载的方式打开数据。例如： context.Response.AddHeader("Content-Disposition","attachment:filename=aa.jpg"); context.Response.WriteFile("aa.jpg");

Transfer-Encoding：告诉浏览器，传送数据的编码格式

ETag 缓存相关的头（可以做到实时更新）

Expries 告诉浏览器回送的资源缓存多长时间。如果是-1或者0，表示不缓存

Cache-Control 控制浏览器不要缓存数据 no-cache

Pragma 控制浏览器不要缓存数据 no-cache

状态行：例如： HTTP/1.1 200 OK （协议的版本号是1.1 响应状态码为200 响应结果为 OK）

实体内容（实体头）：响应包含浏览器能够解析的静态内容，例如：html，纯文本，图片等等信息

浏览器解析html代码，并请求html代码中的资源。浏览器拿到index.html文件后，就开始解析其中的html代码，遇到js/css/image等静态资源时，就向服务器端去请求下载（会使用多线程下载，每个浏览器的线程数不一样），这个时候就用上keep-alive特性了，建立一次HTTP连接，可以请求多个资源，下载资源的顺序就是按照代码里的顺序，但是由于每个资源大小不一样，而浏览器又多线程请求请求资源，所以从下图看出，这里显示的顺序并不一定是代码里面的顺序。

浏览器对页面进行渲染呈现给用户，最后，浏览器利用自己内部的工作机制，把请求到的静态资源和html代码进行渲染，渲染之后呈现给用户。自此一次完整的HTTP事务宣告完成.

### 11.4服务器端接收到http请求后生成html文件过程

1.nginx读取配置文件

Nginx在收到浏览器 GET / 请求时，会读取http请求里面的头部信息，根据Host来匹配 自己的所有的虚拟主机的配置文件的server\_name,看看有没有匹配的，有匹配那么就读取该虚拟主机的配置，发现如下配置：root /web/echo。通过这个就知道所有网页文件的就在这个目录下 这个目录就是/ 当我们http://www.linux178.com/时就是访问这个目录下面的文件，例如访问http://www.linux178.com/index.html,那么代表/web/echo下面有个文件叫index.html。index index.html index.htm index.php。通过这个就能得知网站的首页文件是那个文件，也就是我们在入http://www.linux178.com/ ，nginx就会自动帮我们把index.html（假设首页是index.php 当然是会尝试的去找到该文件，如果没有找到该文件就依次往下找，如果这3个文件都没有找到，那么就抛出一个404错误）加到后面，那么添加之后的URL是/index.php,然后根据后面的配置进行处理。

location ~ .\*\.php(\/.\*)\*$ {

root /web/echo;

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index.php;

astcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

include fastcgi\_params;

}这一段配置指明凡是请求的URL中匹配（这里是启用了正则表达式进行匹配） \*.php后缀的（后面跟的参数）都交给后端的fastcgi进程进行处理。

1. 把php文件交给fastcgi进程去处理

于是nginx把/index.php这个URL交给了后端的fastcgi进程处理，等待fastcgi处理完成后（结合数据库查询出数据，填充模板生成html文件）返回给nginx一个index.html文档，Nginx再把这个index.html返回给浏览器，于是乎浏览器就拿到了首页的html代码，同时nginx写一条访问日志到日志文件中去。

nginx是怎么找index.php文件的？当nginx发现需要/web/echo/index.php文件时，就会向内核发起IO系统调用，告诉内核，我需要这个文件,内核从/开始找到web目录，再在web目录下找到echo目录，最后在echo目录下找到index.php文件，于是把这个index.php从硬盘上读取到内核自身的内存空间，然后再把这个文件复制到nginx进程所在的内存空间，于是乎nginx就得到了自己想要的文件了。

寻找文件在文件系统层面是怎么操作的？比如nginx需要得到/web/echo/index.php这个文件。每个分区（像ext3 ext3等文件系统，block块是文件存储的最小单元 默认是4096字节）都是包含元数据区和数据区，每一个文件在元数据区都有元数据条目（一般是128字节大小），每一个条目都有一个编号，我们称之为inode（index node 索引节点），这个inode里面包含 文件类型、权限、连接次数、属主和数组的ID、时间戳、这个文件占据了那些磁盘块也就是块的编号（block，每个文件可以占用多个block,并且block不一定是连续的，每个block是有编号的）。假设

/ 在数据区占据 1、2号block ，/其实也是一个目录 里面有3个目录 web 111

web 占据 5号block 是目录 里面有2个目录 echo data

echo 占据 11号 block 是目录 里面有1个文件 index.php

index.php 占据 15 16号 block 是文件

内核拿到nginx的IO系统调用要获取/web/echo/index.php这个文件请求之后

① 内核读取元数据区 / 的inode，从inode里面读取/所对应的数据块的编号，然后在数据区找到其对应的块（1 2号块），读取1号块上的映射表找到web这个名称在元数据区对应的inode号

② 内核读取web对应的inode（3号），从中得知web在数据区对应的块是5号块，于是到数据区找到5号块，从中读取映射表，知道echo对应的inode是5号，于是到元数据区找到5号inode

③ 内核读取5号inode，得到echo在数据区对应的是11号块，于是到数据区读取11号块得到映射表，得到index.php对应的inode是9号

④ 内核到元数据区读取9号inode，得到index.php对应的是15和16号数据块，于是就到数据区域找到15 16号块，读取其中的内容，得到index.php的完整内容

### 11.5 HTTP文件断点续传的原理

对于断点续传，关键点是两个：

1. 终端知道当前的文件和上一次加载的文件是不是内容发生了变化，如果有变化，需要重新从offset 0 的位置开始下载

2. 终端记录好上次成功下载到的offset，告诉server端,server端支持从特定的offset 开始吐数据

对于关键点1，对于决定大部分产品的业务场景，可以通过前置业务接口解决；这里简单介绍一下：对于非下载工具类的产品，如视频APP（奇艺,优酷），视频播放前会请求相关业务的信息，主要返回片子叫什么名字，主要演员等等一些列信息，同时会返回一个对于播放最重要的信息——播放地址。 播放地址就是我们可以做文章的地方，如果《太子妃第一集》这个片子更新了（被广电要求减掉某个污的画面），可以后端系统让这个业务接口吐不同的播放地址/一个不同的url参数(?ver=1.1)/位置参数(#ver1.1)。这样纯天然的URL变化能纯天然的让终端认为不是同一个片子，而需要重新加载。没有业务接口的下载工具类的如何解决呢？

下载工具类的没有前置接口，可以使用HTTP 的ETAG来标识是否文件已经修改。

ETAG原理：如果URL上的资源内容改变，一个新的不一样的ETag就会被分配。用这种方法使用ETag即类似于指纹，并且他们能够被快速地被比较，以确定两个版本的资源是否相同。ETag的比较只对同一个URL有意义——不同URL上的资源的ETag值可能相同也可能不同，从他们的ETag的比较中无从推断。ETAG是HTTP的一个可选字段，且没有规范他的实现；实际上业内用的比较多的就是使用MD5签名的方式来生成(linux shell md5sum)

典型用法：

server端： Nginx >1.3.3 自带有ETAG的module , 当然同时也可以在业务代码里SetHeaders加一个ETAG字段

client端：

第一次请求时：

String etag = httpURLConnection.getHeaderField("ETag");

ETag: "b428eab9654aa7c87091e"

第二次请求（断点续传时）：

httpURLConnection.setRequestProperty(“If-None-Match”, "b428eab9654aa7c87091e");

If-None-Match: "b428eab9654aa7c87091e"

如果ETag值匹配，这就意味着资源没有改变，服务器便会发送回一个极短的响应，包含HTTP “304 未修改”的状态。304状态告诉客户端，它的缓存版本是最新的，并应该使用它。然而，如果ETag的值不匹配，这就意味着资源很可能发生了变化，那么，一个完整的响应就会被返回，包括资源的内容，就好像ETag没有被使用。这种情况下，客户端可以用新返回的资源和新的ETag替代先前的缓存版本。

对于一个C/C++程序员，第一时间会得出一个系统级实现方案：

1. 客户端传当前的offset

2. server端seek到文件特定的offset开始读取往http connection吐数据

不过我们深处在一个开放方案和标准不断完善的时代，不需要自己实现一个（这也是像我这样的C/C++研发工程师越来越没落的原因）,来看看HTTP协议是怎么解决这个问题的：

HTTP头Range字段：

Range : 用于客户端到服务器端的请求，可通过该字段指定下载文件的某一段大小，及其单位。典型的格式如：

Range: bytes=0-499 下载第0-499字节范围的内容   
Range: bytes=500-999 下载第500-999字节范围的内容   
Range: bytes=-500 下载最后500字节的内容   
Range: bytes=500- 下载从第500字节开始到文件结束部分的内容

**来个简单粗暴的例子**

curl --header "Range: bytes=0-20000" xxx.com/memcache.pdf -o part1  
curl --header "Range: bytes=20001-223651" xxx.com/memcache.pdf -o part2  
cat part1 part2 >> a.pdf

参考连接：<https://www.cnblogs.com/Creator/p/5490929.html>

### 11.6 四种常见的 POST 提交数据方式

其实就是将网页中form等数据封装到请求的body中数据的方式。

而url后面的参数一般是送给php等服务器参数，可理解为接口参数。

定义和用法

enctype 属性规定在发送到服务器之前应该如何对表单数据进行编码。默认地，表单数据会编码为 "application/x-www-form-urlencoded"。就是说，在发送到服务器之前，所有字符都会进行编码（空格转换为 "+" 加号，特殊符号转换为 ASCII HEX 值）。

参考链接：<https://www.cnblogs.com/softidea/p/5745369.html>

enctype属性值

|  |  |
| --- | --- |
| **值** | **描述** |
| application/x-www-form-urlencoded | 在发送前编码所有字符（默认） |
| multipart/form-data | 不对字符编码。在使用包含文件上传控件的表单时，必须使用该值。 |
| text/plain | 空格转换为 "+" 加号，但不对特殊字符编码。 |

我们知道，HTTP 协议是以 ASCII 码传输，建立在 TCP/IP 协议之上的应用层规范。规范把 HTTP 请求分为三个部分：状态行、请求头、消息主体。类似于下面这样：

BASH<method> <request-URL> <version>

<headers>

<entity-body>

协议规定 POST 提交的数据必须放在消息主体（entity-body）中，但协议并没有规定数据必须使用什么编码方式。实际上，开发者完全可以自己决定消息主体的格式，只要最后发送的 HTTP 请求满足上面的格式就可以。但是，数据发送出去，还要服务端解析成功才有意义。一般服务端语言如 php、python 等，以及它们的 framework，都内置了自动解析常见数据格式的功能。服务端通常是根据请求头（headers）中的 Content-Type 字段来获知请求中的消息主体是用何种方式编码，再对主体进行解析。所以说到 POST 提交数据方案，包含了 Content-Type 和消息主体编码方式两部分。

下面就正式开始介绍它们。

1.application/x-www-form-urlencoded

这应该是最常见的 POST 提交数据的方式了。浏览器的原生 <form> 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。

<form action="form\_action.asp" enctype="text/plain">

<p>First name: <input type="text" name="fname" /></p>

<p>Last name: <input type="text" name="lname" /></p>

<input type="submit" value="Submit" /></form>

此时Form提交的请求数据，抓包时看到的请求会是这样的内容（无关的请求头在本文中都省略掉了）：

BASHPOST http://www.example.com HTTP/1.1

Content-Type: application/x-www-form-urlencoded;charset=utf-8

title=test&sub%5B%5D=1&sub%5B%5D=2&sub%5B%5D=3

首先，Content-Type 被指定为 application/x-www-form-urlencoded；其次，提交的数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL 转码。大部分服务端语言都对这种方式有很好的支持。例如 PHP 中，$\_POST['title'] 可以获取到 title 的值，$\_POST['sub'] 可以得到 sub 数组。很多时候，我们用 Ajax 提交数据时，也是使用这种方式。例如 JQuery 和 QWrap 的 Ajax，Content-Type 默认值都是「application/x-www-form-urlencoded;charset=utf-8」。

2.multipart/form-data

这又是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，必须让 <form> 表单的enctype 等于 multipart/form-data。直接来看一个请求示例：

**BASH**POST http://www.example.com HTTP/1.1

Content-Type:multipart/form-data; boundary=----WebKitFormBoundaryrGKCBY7qhFd3TrwA

------WebKitFormBoundaryrGKCBY7qhFd3TrwA

Content-Disposition: form-data; name="text"

title

------WebKitFormBoundaryrGKCBY7qhFd3TrwA

Content-Disposition: form-data; name="file"; filename="chrome.png"

Content-Type: image/png

PNG ... content of chrome.png ...

------WebKitFormBoundaryrGKCBY7qhFd3TrwA--

首先生成了一个 boundary 用于分割不同的字段，为了避免与正文内容重复，boundary 很长很复杂。然后 Content-Type 里指明了数据是以 multipart/form-data 来编码，本次请求的 boundary 是什么内容。消息主体里按照字段个数又分为多个结构类似的部分，每部分都是以 --boundary 开始，紧接着是内容描述信息，然后是回车，最后是字段具体内容（文本或二进制）。如果传输的是文件，还要包含文件名和文件类型信息。消息主体最后以 --boundary-- 标示结束。关于 multipart/form-data 的详细定义，请前往 rfc1867 查看。这种方式一般用来上传文件，各大服务端语言对它也有着良好的支持。

上面提到的这两种 POST 数据的方式，都是浏览器原生支持的，而且现阶段标准中原生 <form> 表单也只支持这两种方式（通过 <form> 元素的 enctype 属性指定，默认为 application/x-www-form-urlencoded。其实 enctype 还支持 text/plain，不过用得非常少）。

随着越来越多的 Web 站点，尤其是 WebApp，全部使用 Ajax 进行数据交互之后，我们完全可以定义新的数据提交方式，给开发带来更多便利。

3.application/json

application/json 这个 Content-Type 作为响应头大家肯定不陌生。实际上，现在越来越多的人把它作为请求头，用来告诉服务端消息主体是序列化后的 JSON 字符串。由于 JSON 规范的流行，除了低版本 IE 之外的各大浏览器都原生支持 JSON.stringify，服务端语言也都有处理 JSON 的函数，使用 JSON 不会遇上什么麻烦。

这种方案，可以方便的提交复杂的结构化数据，特别适合 RESTful 的接口。各大抓包工具如 Chrome 自带的开发者工具、Firebug、Fiddler，都会以树形结构展示 JSON 数据，非常友好。但也有些服务端语言还没有支持这种方式，例如 php 就无法通过 $\_POST 对象从上面的请求中获得内容。这时候，需要自己动手处理下：在请求头中 Content-Type 为 application/json 时，从 php://input 里获得原始输入流，再 json\_decode 成对象。一些 php 框架已经开始这么做了。

4.text/xml

我的博客之前提到过 XML-RPC（XML Remote Procedure Call）。它是一种使用 HTTP 作为传输协议，XML 作为编码方式的远程调用规范。  
为什么不推荐采用application/x-www-form-urlencoded这种类型，它有如下问题：

测试困难,就别想通过postman这类工具测试，如果value中的对象也是一个对象，那么在构建这个参数时就非常困难，看下它的过程采用key1=value1&key2=value2这种形式将所有参数拼接起来，从一长串字符中想了解每个参数的含义没有个好眼力怕是不行。

value要进行编码，编码之后的对调试者不友好。

value是复杂对象的情况更加糟糕，一般只能通过程序来序列化得到参数，想手写基本不可能。

客户端调用复杂

参考链接：<https://www.cnblogs.com/softidea/p/5745369.html>

### 11.7 参考链接

1. 一次完整的HTTP请求过程

<https://www.cnblogs.com/xuxinstyle/p/9382506.html>

## 12 websocket与MPEG-DASH

### 12.1 websocket

#### 12.1.1 简介

WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。现在很多网站为了实现推送技术，所用的技术都是 Ajax 轮询。由浏览器对服务器发出HTTP请求，然后由服务器返回最新的数据给客户端的浏览器。这种传统的模式浏览器需要不断的向服务器发出请求，然而HTTP请求可能包含较长的头部，其中真正有效的数据可能只是很小的一部分，显然这样会浪费很多的带宽等资源。

软件通信有七层结构，下三层结构偏向与数据通信，上三层更偏向于数据处理，中间的传输层则是连接上三层与下三层之间的桥梁，每一层都做不同的工作，上层协议依赖与下层协议。基于这个通信结构的概念。

Socket 其实并不是一个协议，是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。当两台主机通信时，让 Socket 去组织数据，以符合指定的协议。TCP 连接则更依靠于底层的 IP 协议，IP 协议的连接则依赖于链路层等更低层次。

WebSocket 则是一个典型的应用层协议。

总的来说：Socket 是传输控制层协议，WebSocket 是应用层协议。

WebSocket 协议本质上是一个基于 TCP 的协议。为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息"Upgrade: WebSocket"表明这是一个申请协议升级的 HTTP 请求，服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了，双方就可以通过这个连接通道自由的传递信息，并且这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。

Websocket 使用 ws 或 wss 的统一资源标志符，类似于 HTTPS，其中 wss 表示在 TLS 之上的 Websocket。如：

ws://example.com/wsapi

wss://secure.example.com/

Websocket 使用和 HTTP 相同的 TCP 端口，可以绕过大多数防火墙的限制。默认情况下，Websocket 协议使用 80 端口；运行在 TLS 之上时，默认使用 443 端口。

Socket.IO 是一个封装了 Websocket、基于 Node 的 JavaScript 框架，包含 client 的 JavaScript 和 server 的 Node。其屏蔽了所有底层细节，让顶层调用非常简单。

#### 12.1.2 websocket握手请求

一个典型的Websocket握手请求如下：

客户端请求

GET / HTTP/1.1

Upgrade: websocket

Connection: Upgrade

Host: example.com

Origin: http://example.com

Sec-WebSocket-Key: sN9cRrP/n9NdMgdcy2VJFQ==

Sec-WebSocket-Version: 13

服务器回应

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: fFBooB7FAkLlXgRSz0BT3v4hq5s=

Sec-WebSocket-Location: ws://example.com/

Connection：必须设置 Upgrade，表示客户端希望连接升级。

Upgrade：必须设置 Websocket，表示希望升级到 Websocket 协议。

Sec-WebSocket-Key 是随机的字符串，服务器端会用这些数据来构造出一个 SHA-1 的信息摘要。把 “Sec-WebSocket-Key” 加上一个特殊字符串 “258EAFA5-E914-47DA-95CA-C5AB0DC85B11”，然后计算 SHA-1 摘要，之后进行 BASE-64 编码，将结果做为 “Sec-WebSocket-Accept” 头的值，返回给客户端。如此操作，可以尽量避免普通 HTTP 请求被误认为 Websocket 协议。

Sec-WebSocket-Version 表示支持的 Websocket 版本。RFC6455 要求使用的版本是 13，之前草案的版本均应当弃用。

 Origin 字段是可选的，通常用来表示在浏览器中发起此 Websocket 连接所在的页面，类似于 Referer。但是，与 Referer 不同的是，Origin 只包含了协议和主机名称。

其他一些定义在 HTTP 协议中的字段，如 Cookie 等，也可以在 Websocket 中使用。

# 三．音视频库使用说明

## 1.FFMPEG

### 1.1API学习整理

avcodec\_register\_all：注册全部编码器。有指定编码器得API。

avformat\_network\_init：网路库初始化。如果avformat调用网络功能需要使用。

av\_register\_all：注册avformat复用器。

avformat\_open\_input：打开视频文件。

### 1.2 ffmpeg问题

#### 1.2.1 ffmpeg添加mpeg ps流的pcm的解码支持

<https://blog.csdn.net/garefield/article/details/45113313>

ffmpeg原本不支持mpeg ps流，原因有两个。首先在编码时，ffmpeg没有在流中写入psm头；其次，在解码时，只在读文件头的时候判断了流的类型，而ps流中，流类型是在psm头中的，这样ffmpeg会读不到流类型。mpeg.c函数中mpegps\_read\_packet函数中做修改。

#### 1.2.2 ffmpeg加载慢

Vlc 调用ffmpeg 中avformat\_find\_stream\_info读取信息慢，可以修改probesize = 4096。

## Live555

### 分析live555客户端解析play等命令流程

首先RTSPClient 新建一个createNew，RTSPClient.cpp/RTSPClient 调用taskScheduler，新建task绑定incomingDataHandler，到handleResponseBytes，到handlePLAYResponse，等待接受到发送的play发送响应。具体rtsp交互流程可查看rtsp中相关章节。

### 2.2 rtsp时间戳计算方法

Rtsp的拖放和快放、慢放都是先pause，再play，根据play消息包中的range和scale参数来实现。

对应到live555中，pause时调用sendPauseCommand函数。Play时调用unsigned sendPlayCommand(MediaSubsession& subsession, responseHandler\* responseHandler,double start = 0.0f, double end = -1.0f, float scale = 1.0f, Authenticator\*authenticator = NULL)，在该函数中可以设置start，end（对应range，以秒为单位）和scale（对应快慢放）。end设为-1表示播放到文件结尾。在pause后的resume播放时，start可以设为-1，表示继续从当前位置播放；在seek时，start设为拖放到的播放时间点（以秒为单位）。

rtsp客户端对于每个收到的rtp数据包，可以根据时间戳调用doubleMediaSubsession::getNormalPlayTime(struct timeval const& presentationTime)获得当前相对文件开始（从0起始）的播放时间（秒）。

该函数的计算公式为：PlayStartTime+(currnpt-startnpt)\*scale

PlayStartTIme：从收到服务器play应答中的range域中提取。

Currnpt：当前rtp包的时间戳

Startnpt：play后收到的第一个数据包的npt。

rtspserver中的rtp包时间戳实现：

每收到一个play请求后（seek和resume的play请求同样如此），取当前时间作为时间戳。以后每个数据包的时间戳计算公式为：play后第一个包的时间戳+当前包与第一个包在文件中的相对时间间隔。注意在Play的响应包中音视频的seq和rtptime分别要取pause时的值，否则客户端的时间条指示会不对。

在live555的rtspserver中，收到resume的play请求时（range域start为-1），返回的应答包不会置range域start时间，导致客户端提取PlayStartTIme为0。为了避免这种问题，客户端在resume的play请求中最好置start为当前播放时间，这样其实类似于seek操作了。

在读取文件结束时（音频或视频结束），构造RTCP BYE包，通过RTCP通道往客户端发送一个BYE包，客户端会结束播放。