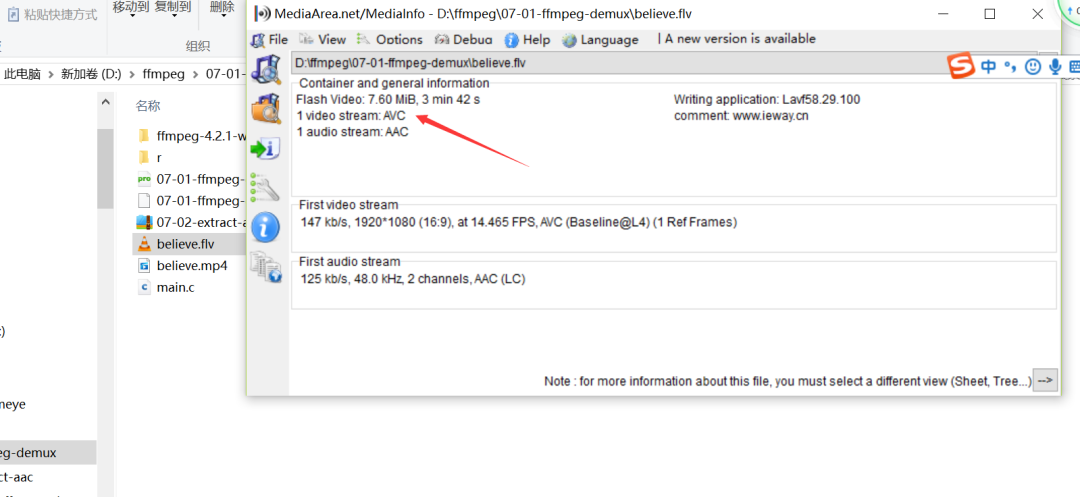
# H264之NALU解析

## **一、H264简介：**

H.264从1999年开始，到2003年形成草案，最后在2007年定稿有待核实。在ITU的标准⾥称 为H.264，在MPEG的标准⾥是MPEG-4的⼀个组成部分–MPEG-4 Part 10，⼜叫Advanced Video Codec，因此常常称为MPEG-4 AVC或直接叫AVC：



## **二、H264编码原理：**

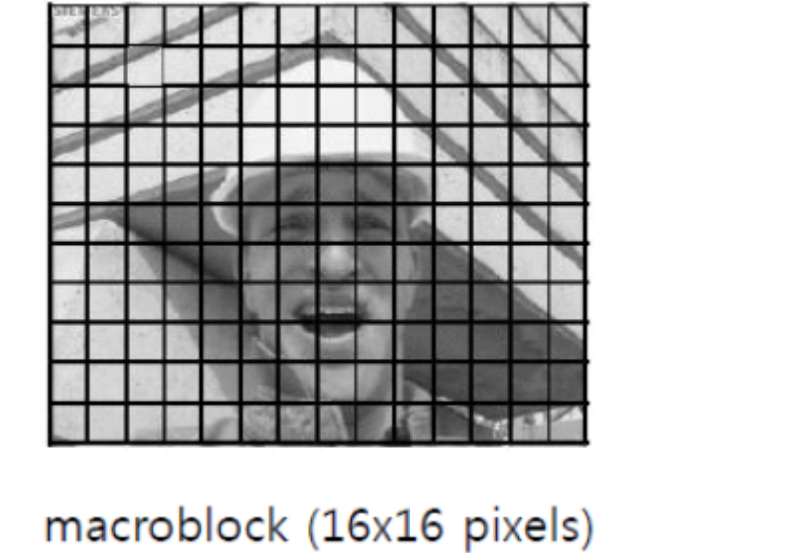
在⾳视频传输过程中，视频⽂件的传输是⼀个极⼤的问题；⼀段分辨率为1920\**1080，每个像素点为RGB占⽤3个字节，帧率是25的视频，对于传输带宽的要求是：1920\**108\**3*25/1024/1024=148.315MB/s，换成bps则意味着视频每秒带宽为 1186.523Mbps，这样的速率对于⽹络存储是不可接受的。因此视频压缩和编码技术应运⽽⽣。

这里普及基本知识哈：

**Mbps(Million bits per second)即“传输速率”，也叫“带宽”，咋们去营业厅开网线的时候会问你办几兆的宽带，那么这里说的“几兆的宽带”就是指多少Mbps，但是Mbps和MB/s是怎么换算的呢？**

* + **8Mbps=1MB/s；8Mbps换算成下载速度就是1MB/s，不过由于种种限制，实际情况中8M的宽带往往达不到1MB/s的下载速度，能达到800KB/s以上算是正常情况；所以上面计算的结果每秒带宽为 1186.523Mbps是这样来的：1Mbps代表每秒传输1,000,000位，即每秒传输1,000,000/8=125,000字节=125KB=0.125MB。148.315  /  0.125 = 1186.523**

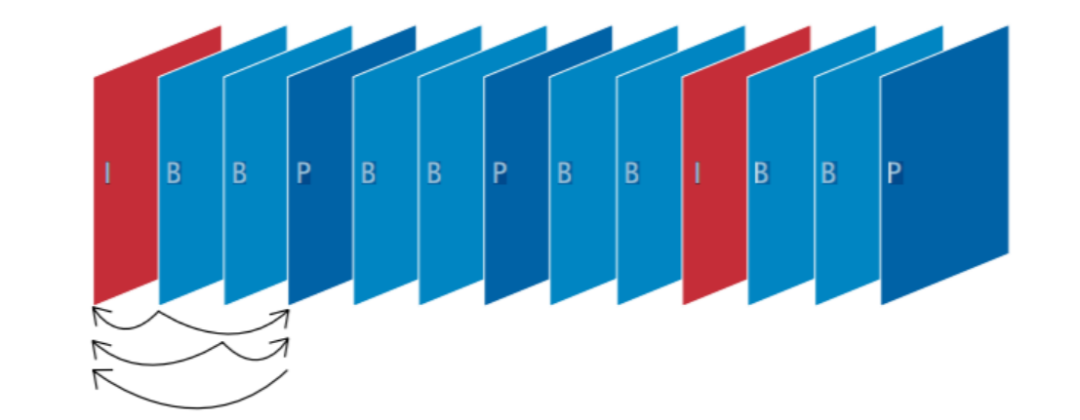
对于视频⽂件来说，视频由单张图⽚帧所组成，⽐如每秒25帧，但是图⽚帧的像素块之间存在 相似性，因此视频帧图像可以进⾏图像压缩；H264采⽤了16\*16的分块⼤⼩对视频帧图像 进⾏相似⽐较和压缩编码。如下图所示(这里涉及到空间压缩和帧间压缩不做详细介绍！)：



## **三、H264中I帧、P帧、B帧：**

这块知识介绍在之前的文章里面有介绍，为了知识的完整性，今天这里再次介绍一下。

H264使⽤帧内压缩和帧间压缩的⽅式提⾼编码压缩率；H264采⽤了独特的I帧、P帧和B帧策略 来实现，连续帧之间的压缩：



| **帧的分类** | **中文** | **意义** |
| --- | --- | --- |
| I帧 | 帧内编码帧 (intra picture) | I帧通常是每个GOP(MPEG所使用的一种视频压缩技术)的第一帧，经过适度地压缩，做为随机访问的参考点，可以当成图像。I帧可以看成是一个图像经过压缩后的产物。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帧 | 前向预测编码帧（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。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帧 | 双向预测帧(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。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。 |

总结：我们从上面的介绍可以发现，他们的压缩效率为：

* **压缩率大小：B > P > I**

补充说明：

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）。它的特点：

1、它是⼀个全帧压缩编码帧。它将全帧图像信息进⾏JPEG压缩编码及传输。

2、解码时仅⽤I帧的数据就可重构完整图像。

3、I帧描述了图像背景和运动主体的详情。

4、I帧不需要参考其他画⾯⽽⽣成。

5、I帧是P帧和B帧的参考帧(其质量直接影响到同组中以后各帧的质量)。

6、 I帧是帧组GOP的基础帧(如果为IDR则为第⼀帧),在⼀组中只有⼀个IDR帧，⼀个或多个I 帧(包括IDR帧)。

7、I帧不需要考虑运动⽮量。

8、 I帧所占数据的信息量⽐较⼤。

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。它的特点：

* P帧是I帧后⾯相隔1~2帧的编码帧。
* P帧采⽤运动补偿的⽅法传送它与前⾯的I或P帧的差值及运动⽮量(预测误差)。
* 解码时必须将I帧中的预测值与预测误差求和后才能重构完整的P帧图像。
* 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。它的特点：

1）B帧是由前⾯的I或P帧和后⾯的P帧来进⾏预测的。

2）B帧传送的是它与前⾯的I或P帧和后⾯的P帧之间的预测误差及运动⽮量。

3）B帧是双向预测编码帧。

4）B帧压缩⽐最⾼,因为它只反映两参考帧间运动主体的变化情况,预测⽐较准确。

5）B帧不是参考帧,不会造成解码错误的扩散。

注意：

I、B、P各帧是根据压缩算法的需要，是⼈为定义的,它们都是实实在在的物理帧。⼀般来 说，I帧的压缩率是7（跟JPG差不多），P帧是20，B帧可以达到50。可⻅使⽤B帧能节省⼤量 空间，节省出来的空间可以⽤来保存多⼀些I帧，这样在相同码率下，可以提供更好的画质。

## **四、H264编码结构解析：**

H264除了对视频压缩处理之外，为了方便网络传输，提供了对应的视频编码和分片策略；类似网络数据 封装成IP帧，在H264中将其称为组(GOP,gruop of pictures)、片(slice)、宏块(Macroblock)这些一起组成了H264的码流分层结构；H264将其组织成为序列(GOP)、图片(pictrue)、片(slice)、宏块(Macroblock)、子块(subblock)等五个层次。

补充说明一下：

在视频编码序列中，

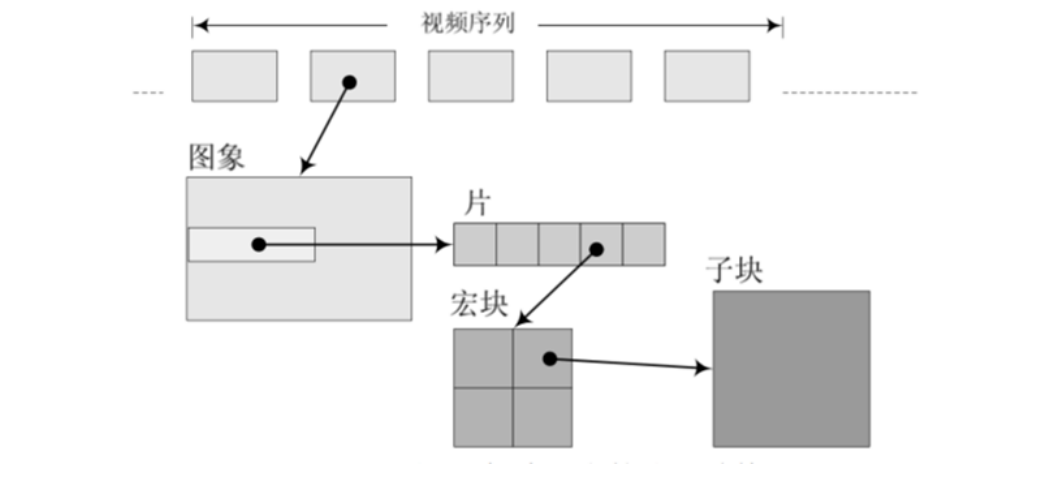
GOP即Group of picture（图像组），指两个I帧之间的距离，

Reference（参考周期）指两个P帧之间的距离。⼀个I帧所占⽤的字节数⼤于⼀个P帧，⼀个P 帧所占⽤的字节数⼤于⼀个B帧。

所以在码率不变的前提下，GOP值越⼤，P、B帧的数量会越多，平均每个I、P、B帧所占⽤的 字节数就越多，也就更容易获取较好的图像质量；Reference越⼤，B帧的数量越多，同理也 更容易获得较好的图像质量。

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，会影响到⼀个GOP中后续P、B 帧的图像质量，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。同时，由于P、B帧的复杂度⼤于I帧，所以过多的P、B帧会影响编码效率，使编码效率降低。另外，过⻓的GOP还会影响Seek操作的响应速度，由于P、B帧是由前⾯的I或P帧预测得到 的，所以Seek操作需要直接定位，解码某⼀个P或B帧时，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值越⻓，需要解码的预测帧就越多，seek响应的时间也越⻓。

GOP （图像组）主要⽤作形容⼀个IDR帧 到下⼀个IDR帧 之间 间隔了多少个帧：



H264将视频分为连续的帧进⾏传输，在连续的帧之间使⽤I帧、P帧和B帧。同时对于帧内⽽⾔，将图像分块为⽚、宏块和字块进⾏分⽚传输；通过这个过程实现对视频⽂件的压缩包装。

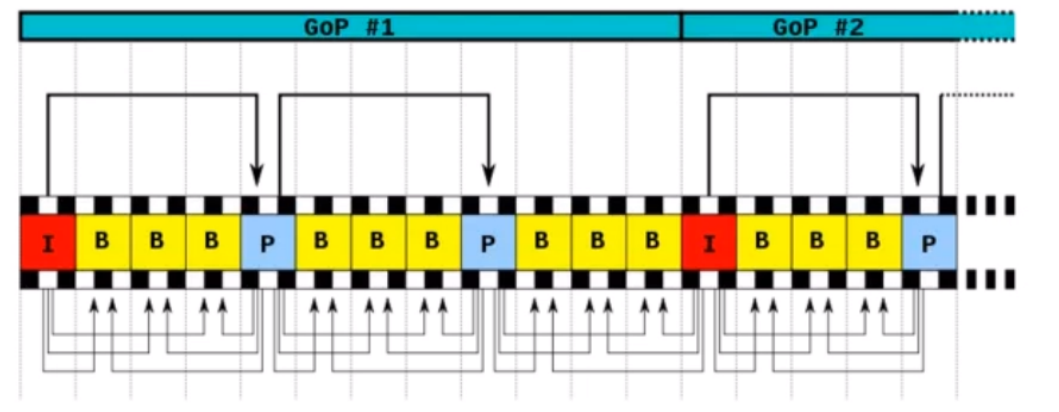
这里简单介绍一下什么是IDR帧？**IDR（Instantaneous Decoding Refresh，即时解码刷新）**；⼀个序列的第⼀个图像叫做 IDR 图像（⽴即刷新图像），IDR 图像都是 I 帧图像(不过I帧不一定是IDR帧哈！)。I和IDR帧都使⽤帧内预测。I帧不⽤参考任何帧，但是之后的P帧和B帧是有可能参考这个I帧之 前的帧的。IDR就不允许这样。⽐如（解码的顺序）：

**IDR1 P4 B2 B3 P7 B5 B6 I10 B8 B9 P13 B11 B12 P16 B14 B15   
这⾥的B8可以跨过I10去参考P7(解码的时候哈)  
  
原始图像：IDR1 B2 B3 P4 B5 B6 P7 B8 B9 I10**

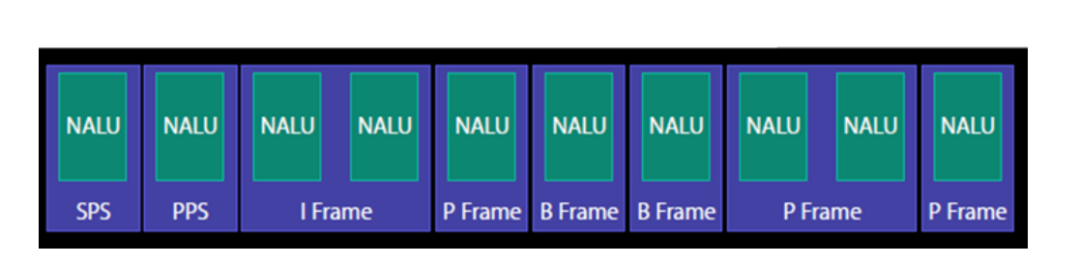
**IDR1 P4 B2 B3 P7 B5 B6 IDR8 P11 B9 B10 P14 B11 B12   
这⾥的B9就只能参照IDR8和P11，不可以 参考IDR8前⾯的帧**

**其核⼼作⽤是，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**。

下⾯是⼀个H264码流的举例（从码流的帧分析可以看出来B帧不能被当做参考帧，它的延迟比较大）：



## **五、NALU(Network Abstract Layer Unit)介绍：**



SPS：序列参数集，SPS中保存了一组编码视频序列(Coded video sequence)的全局参数。

PPS:图像参数集，对应的是一个序列中某一副图像或者某几副图像的参数。

I帧：帧内编码，可独立解码生成完整的图片

P帧：前向预测编码帧，需要参考其前⾯的⼀个I 或者B 来⽣成⼀张完整的图⽚。

B帧: 双向预测内插编码帧，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。

注意：发I帧之前，⾄少要发⼀次SPS和PPS。

下面我们开始介绍NALU结构:

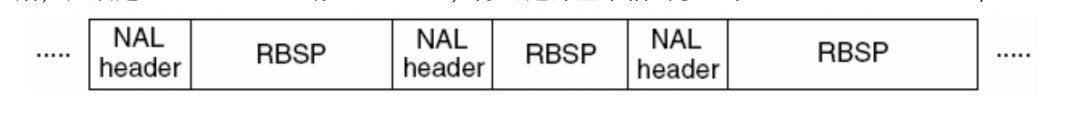
H.264原始码流(裸流)是由⼀个接⼀个NALU组成，它的功能分为两层，VCL(视频编码层)和 NAL(⽹络提取层)：

VCL：包括核⼼压缩引擎和块，宏块和⽚的语法级别定义，设计⽬标是尽可能地独⽴于⽹ 络进⾏⾼效的编码。

NAL：负责将VCL产⽣的⽐特字符串适配到各种各样的⽹络和多元环境中，覆盖了所有⽚级 以上的语法级别。

在VCL进⾏数据传输或存储之前，这些编码的VCL数据，被映射或封装进NAL单元。下面是一个NALU的组成：

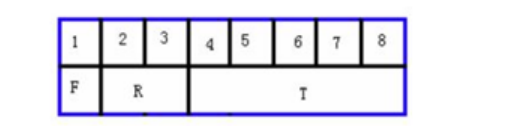
⼀个NALU = ⼀组对应于视频编码的NALU头部信息 + ⼀个原始字节序列负荷(RBSP,Raw Byte Sequence Payload).

NALU结构单元的主体结构如下所示；⼀个原始的H.264 NALU单元通常由[StartCode] [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"，除此之外基本相当于⼀个NAL header + RBSP：

注意：对于FFmpeg解复⽤后，MP4⽂件读取出来的packet是不带startcode，但TS⽂件读取出来 的packet带了startcode。

## **六、解析NALU:**

每个NAL单元是⼀个⼀定语法元素的可变⻓字节字符串，包括包含⼀个字节的头信息（⽤来表 示数据类型），以及若⼲整数字节的负荷数据。NALU头信息（⼀个字节）：



T为负荷数据类型，占5bit; nal\_unit\_type：这个NALU单元的类型,1～12由H.264使⽤，24～31由H.264以外的应⽤

R为重要性指示位，占2个bit;nal\_ref\_idc.：取00~11,似乎指示这个NALU的重要性,如00的NALU解码器可以丢弃它⽽不 影响图像的回放,0～3，取值越⼤，表示当前NAL越重要，需要优先受到保护。如果当前 NAL是属于参考帧的⽚，或是序列参数集，或是图像参数集这些重要的单位时，本句法元 素必需⼤于0。

最后的F为禁⽌位，占1bit；forbidden\_zero\_bit：在 H.264 规范中规定了这⼀位必须为 0。

H.264标准指出，当数据流是储存在介质上时，在每个NALU 前添加起始码：0x000001 或 0x00000001，⽤来指示⼀个NALU 的起始和终⽌位置：

在这样的机制下，在码流中检测起始码，作为⼀个NALU得起始标识，当检测到下⼀个起始 码时，当前NALU结束。

3字节的0x000001只有⼀种场合下使⽤，就是⼀个完整的帧被编为多个slice（⽚）的时 候，包含这些slice的NALU 使⽤3字节起始码。其余场合都是4字节0x00000001的。

比如说：

**0x00 00 00 01 67 …   
0x00 00 00 01 68 …  
0x00 00 00 01 65 …  
  
67：  
⼆进制：0110 0111  
00111 = 7（⼗进制）**

下面是nal\_unit\_type的组成：

| **nal\_unit\_type** | **NAL 单元和 RBSP 语法结构的内容** |
| --- | --- |
| 0 | 未指定 |
| 1 | ⼀个⾮IDR图像的编码条带 slice\_layer\_without\_partitioning\_rbsp ( ) |
| 2 | 编码条带数据分割块A slice\_data\_partition\_a\_layer\_rbsp( ) |
| 3 | 编码条带数据分割块B slice\_data\_partition\_b\_layer\_rbsp( ) |
| 4 | 编码条带数据分割块C slice\_data\_partition\_c\_layer\_rbsp( ) |
| 5 | IDR图像的编码条带(⽚) slice\_layer\_without\_partitioning\_rbsp ( ) |
| 6 | 辅助增强信息 (SEI) sei\_rbsp( ) |
| 7 | 序列参数集 seq\_parameter\_set\_rbsp( ) |
| 8 | 图像参数集 pic\_parameter\_set\_rbsp( ) |

| **9** | **访问单元分隔符 access\_unit\_delimiter\_rbsp( )** |
| --- | --- |
| 10 | 序列结尾 end\_of\_seq\_rbsp( ) |
| 11 | 流结尾 end\_of\_stream\_rbsp( ) |
| 12 | 填充数据 filler\_data\_rbsp( ) |
| 13 | 序列参数集扩展seq\_parameter\_set\_extension\_rbsp( ) |
| 14...18 | 保留 |
| 19 | 未分割的辅助编码图像的编码条带 slice\_layer\_without\_partitioning\_rbsp ( ) |
| 20...23 | 保留 |
| 24...31 | 未指定 |

## **七、H264 annexb模式：**

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都是mp4模式，没有startcode，SPS和PPS以及其它信息 被封装在container中，每⼀个frame前⾯4个字节是这个frame的⻓度。

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 实现如下：

 const AVBitStreamFilter \*bsfilter = av\_bsf\_get\_by\_name("h264\_mp4toannexb");   
   
 AVBSFContext \*bsf\_ctx = NULL;   
 // 2 初始化过滤器上下⽂   
 av\_bsf\_alloc(bsfilter, &bsf\_ctx); //AVBSFContext;  
  // 3 添加解码器属性  
  avcodec\_parameters\_copy(bsf\_ctx->par\_in, ifmt\_ctx->streams[videoindex]->cod ecpar);  
 av\_bsf\_init(bsf\_ctx);

下面是一个实际工程demo:

*#include <stdio.h>*  
*#include <libavutil/log.h>*  
*#include <libavformat/avio.h>*  
*#include <libavformat/avformat.h>*  
  
*#define ADTS\_HEADER\_LEN  7;*  
  
const int sampling\_frequencies[] = {  
    96000,  // 0x0  
    88200,  // 0x1  
    64000,  // 0x2  
    48000,  // 0x3  
    44100,  // 0x4  
    32000,  // 0x5  
    24000,  // 0x6  
    22050,  // 0x7  
    16000,  // 0x8  
    12000,  // 0x9  
    11025,  // 0xa  
    8000   // 0xb  
    // 0xc d e f是保留的  
};  
  
int adts\_header(char \* const p\_adts\_header, const int data\_length,  
                const int profile, const int samplerate,  
                const int channels)  
{  
  
    int sampling\_frequency\_index = 3; // 默认使用48000hz  
    int adtsLen = data\_length + 7;  
  
    int frequencies\_size = sizeof(sampling\_frequencies) / sizeof(sampling\_frequencies[0]);  
    int i = 0;  
    for(i = 0; i < frequencies\_size; i++)  
    {  
        if(sampling\_frequencies[i] == samplerate)  
        {  
            sampling\_frequency\_index = i;  
            break;  
        }  
    }  
    if(i >= frequencies\_size)  
    {  
        printf("unsupport samplerate:%d\n", samplerate);  
        return -1;  
    }  
  
    p\_adts\_header[0] = 0xff;         //syncword:0xfff                          高8bits  
    p\_adts\_header[1] = 0xf0;         //syncword:0xfff                          低4bits  
    p\_adts\_header[1] |= (0 << 3);    //MPEG Version:0 for MPEG-4,1 for MPEG-2  1bit  
    p\_adts\_header[1] |= (0 << 1);    //Layer:0                                 2bits  
    p\_adts\_header[1] |= 1;           //protection absent:1                     1bit  
  
    p\_adts\_header[2] = (profile)<<6;            //profile:profile               2bits  
    p\_adts\_header[2] |= (sampling\_frequency\_index & 0x0f)<<2; //sampling frequency index:sampling\_frequency\_index  4bits  
    p\_adts\_header[2] |= (0 << 1);             //private bit:0                   1bit  
    p\_adts\_header[2] |= (channels & 0x04)>>2; //channel configuration:channels  高1bit  
  
    p\_adts\_header[3] = (channels & 0x03)<<6; //channel configuration:channels 低2bits  
    p\_adts\_header[3] |= (0 << 5);               //original：0                1bit  
    p\_adts\_header[3] |= (0 << 4);               //home：0                    1bit  
    p\_adts\_header[3] |= (0 << 3);               //copyright id bit：0        1bit  
    p\_adts\_header[3] |= (0 << 2);               //copyright id start：0      1bit  
    p\_adts\_header[3] |= ((adtsLen & 0x1800) >> 11);           //frame length：value   高2bits  
  
    p\_adts\_header[4] = (uint8\_t)((adtsLen & 0x7f8) >> 3);     //frame length:value    中间8bits  
    p\_adts\_header[5] = (uint8\_t)((adtsLen & 0x7) << 5);       //frame length:value    低3bits  
    p\_adts\_header[5] |= 0x1f;                                 //buffer fullness:0x7ff 高5bits  
    p\_adts\_header[6] = 0xfc;      //‭11111100‬       //buffer fullness:0x7ff 低6bits  
    // 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。  
  
    return 0;  
}  
  
int main(int argc, char \*argv[])  
{  
    int ret = -1;  
    char errors[1024];  
  
    char \*in\_filename = NULL;  
    char \*aac\_filename = NULL;  
  
    FILE \*aac\_fd = NULL;  
  
    int audio\_index = -1;  
    int len = 0;  
  
  
    AVFormatContext \*ifmt\_ctx = NULL;  
    AVPacket pkt;  
  
    // 设置打印级别  
    av\_log\_set\_level(AV\_LOG\_DEBUG);  
  
    if(argc < 3)  
    {  
        av\_log(NULL, AV\_LOG\_DEBUG, "the count of parameters should be more than three!\n");  
        return -1;  
    }  
  
    in\_filename = argv[1];      // 输入文件  
    aac\_filename = argv[2];     // 输出文件  
  
    if(in\_filename == NULL || aac\_filename == NULL)  
    {  
        av\_log(NULL, AV\_LOG\_DEBUG, "src or dts file is null, plz check them!\n");  
        return -1;  
    }  
  
    aac\_fd = fopen(aac\_filename, "wb");  
    if (!aac\_fd)  
    {  
        av\_log(NULL, AV\_LOG\_DEBUG, "Could not open destination file %s\n", aac\_filename);  
        return -1;  
    }  
  
    // 打开输入文件  
    if((ret = avformat\_open\_input(&ifmt\_ctx, in\_filename, NULL, NULL)) < 0)  
    {  
        av\_strerror(ret, errors, 1024);  
        av\_log(NULL, AV\_LOG\_DEBUG, "Could not open source file: %s, %d(%s)\n",  
               in\_filename,  
               ret,  
               errors);  
        return -1;  
    }  
  
    // 获取解码器信息  
    if((ret = avformat\_find\_stream\_info(ifmt\_ctx, NULL)) < 0)  
    {  
        av\_strerror(ret, errors, 1024);  
        av\_log(NULL, AV\_LOG\_DEBUG, "failed to find stream information: %s, %d(%s)\n",  
               in\_filename,  
               ret,  
               errors);  
        return -1;  
    }  
  
    // dump媒体信息  
    av\_dump\_format(ifmt\_ctx, 0, in\_filename, 0);  
  
    // 初始化packet  
    av\_init\_packet(&pkt);  
  
    // 查找audio对应的steam index  
    audio\_index = av\_find\_best\_stream(ifmt\_ctx, AVMEDIA\_TYPE\_AUDIO, -1, -1, NULL, 0);  
    if(audio\_index < 0)  
    {  
        av\_log(NULL, AV\_LOG\_DEBUG, "Could not find %s stream in input file %s\n",  
               av\_get\_media\_type\_string(AVMEDIA\_TYPE\_AUDIO),  
               in\_filename);  
        return AVERROR(EINVAL);  
    }  
  
    // 打印AAC级别  
    printf("audio profile:%d, FF\_PROFILE\_AAC\_LOW:%d\n",  
           ifmt\_ctx->streams[audio\_index]->codecpar->profile,  
           FF\_PROFILE\_AAC\_LOW);  
  
    if(ifmt\_ctx->streams[audio\_index]->codecpar->codec\_id != AV\_CODEC\_ID\_AAC)  
    {  
        printf("the media file no contain AAC stream, it's codec\_id is %d\n",  
               ifmt\_ctx->streams[audio\_index]->codecpar->codec\_id);  
        goto failed;  
    }  
    // 读取媒体文件，并把aac数据帧写入到本地文件  
    while(av\_read\_frame(ifmt\_ctx, &pkt) >=0 )  
    {  
        if(pkt.stream\_index == audio\_index)  
        {  
            char adts\_header\_buf[7] = {0};  
            adts\_header(adts\_header\_buf, pkt.size,  
                        ifmt\_ctx->streams[audio\_index]->codecpar->profile,  
                        ifmt\_ctx->streams[audio\_index]->codecpar->sample\_rate,  
                        ifmt\_ctx->streams[audio\_index]->codecpar->channels);  
            fwrite(adts\_header\_buf, 1, 7, aac\_fd);  // 写adts header , ts流不适用，ts流分离出来的packet带了adts header  
            len = fwrite( pkt.data, 1, pkt.size, aac\_fd);   // 写adts data  
            if(len != pkt.size)  
            {  
                av\_log(NULL, AV\_LOG\_DEBUG, "warning, length of writed data isn't equal pkt.size(%d, %d)\n",  
                       len,  
                       pkt.size);  
            }  
        }  
        av\_packet\_unref(&pkt);  
    }  
  
failed:  
    // 关闭输入文件  
    if(ifmt\_ctx)  
    {  
        avformat\_close\_input(&ifmt\_ctx);  
    }  
    if(aac\_fd)  
    {  
        fclose(aac\_fd);  
    }  
  
    return 0;  
}

我们把mp4格式来转换输出h264格式，看看结果：

