FFMPEG基础篇

2018-4-12 21:12

755

|  |
| --- |
|  |

从事音视频技术开发对FFmpeg都不会感到陌生，通过它可以完成音视频采集、编解码、转码、后处理以及流媒体服务等诸多的功能，可以说涵盖了音视频开发中绝大多数的领域。金山云多媒体SDK团队在移动直播、短视频等项目中遇到了许多问题，本文为《FFmpeg从入门到出家》系列的第一篇上半部分，由LiveVideoStack审校整理，希望能让大家对FFmpeg有更深入了解。

文 / 施雪梅

视频流媒体中程中视频数据的传输占据了绝大部分的带宽，如何提升编码效率，使用更少的带宽，提供更优质的画面质量，是音视频开发人员一直努力的重点。HEVC(High Efficiency Video Coding，也叫H.265)编码格式的推出，给这一方向带来了突破点，但由于其算法复杂度较高，前期未曾得到普遍应用，而随着移动设备计算能力的提高和越来越多的设备开始支持HEVC的硬件编/解码，直播平台也开始逐渐引入HEVC视频格式。

HEVC属视频编码层面标准，如果在视频流媒体中进行应用，还需要相应的封装格式和流媒体协议的支持。鉴于直播的大部分推拉流协议是基于RTMP的，本文主要介绍如何在RTMP协议中增加对HEVC视频编码格式的支持，其他协议或私有协议，可参考本文自行添加。此外，除推流端和播放端要做出修改，用到的RTMP Server部分也要同步进行相应修改，才能够保证HEVC在直播中的正常使用。

# 背景介绍

典型的直播框架通常包括三大部分，如下图所示：

1. 推流端：负责音视频数据的采集、处理、编码及封装后将数据推送至源站；
2. 服务端：涵盖源站和CDN，接收来自推流端的音视频数据，然后将数据分发至各播放端；
3. 播放端：从CDN拉取直播数据，解复用、解码后渲染音视频数据；

[](http://livevideostack.com/data/attachment/portal/201804/17/220319my44n2mu0ivvmhu4.jpg)

图1. 直播框架图

引入HEVC编码，涉及到的变动部分如上图中红色字体所标注：

1. 编码模块：需要支持HEVC格式的编解码，该部分不属于本文的介绍范畴，我们有在其它文章中介绍如何在iOS11上进行HEVC的硬编硬解，感兴趣的朋友可自行查阅；
2. 封装/传输模块：RTMP、HTTP-FLV流媒体协议需要增加对HEVC视频编码格式的支持，该部分是本文介绍的重点。

相信广大的音视频开发者对于FFmpeg并不陌生，由于它在多媒体处理上提供的强大功能以及开源易于修改维护的特性，使得其被广泛应用于各音视频相关软件中。但官方FFmpeg中没有对RTMP FLV中进行HEVC的相关扩展，这是因为FLV与RTMP是Adobe发行的标准，而Adobe暂停了对FLV与RTMP标准的更新，HEVC的相关扩展属于私有标准，所以为了减少国际上不必要的不兼容性麻烦，官方FFmpeg并不会对FLV与RTMP中扩展HEVC进行支持。经过CDN联盟讨论，我们制定了相关的协议扩展规范，并在FFmpeg中完成了相关代码实现。

本文后面介绍的就是如何在FFmpeg中，对RTMP进行HEVC扩展。如果您的开发工程中并没有用到FFmpeg，可直接阅读第四章节，也能够很轻松的在您的代码中增加这部分内容。

# FFmpeg简析

FFmpeg从无到有，发展至今，功能日益强大，代码也越来越多，很多初学者都被其众多的源文件、庞大的结构体和复杂的算法打消了继续学习的念头。本章节将从总体对FFmpeg进行简单的解析，教您如何阅读FFmpeg源码。

## 总体说明

FFmpeg包含如下类库：

* **libavformat**- 用于各种音视频封装格式的生成和解析，包括获取解码所需信息、读取音视频数据等功能。各种流媒体协议代码(如rtmpproto.c等)以及音视频格式的(解)复用代码(如flvdec.c、flvenc.c等)都位于该目录下。
* **libavcodec** - 音视频各种格式的编解码。各种格式的编解码代码(如aacenc.c、aacdec.c等)都位于该目录下。
* **libavutil** - 包含一些公共的工具函数的使用库，包括算数运算，字符操作等。
* **libswscale** - 提供原始视频的比例缩放、色彩映射转换、图像颜色空间或格式转换的功能。
* **libswresample** - 提供音频重采样，采样格式转换和混合等功能。
* **libavfilter** - 各种音视频滤波器。
* **libpostproc**- 用于后期效果处理，如图像的去块效应等。
* **libavdevice** - 用于硬件的音视频采集、加速和显示。
* 如果您之前没有阅读FFmpeg代码的经验，建议优先阅读libavformat、libavcodec以及libavutil下面的代码，它们提供了音视频开发的最基本功能，应用范围也是最广的。

## 常用结构

FFmpeg里面最常用的数据结构，按功能可大致分为以下几类(以下代码行数，以branch: origin/release/3.4为准)：

1. 封装格式

* **AVFormatContext** - 描述了媒体文件的构成及基本信息，是统领全局的基本结构体，贯穿程序始终，很多函数都要用它作为参数；
* **AVInputFormat** - 解复用器对象，每种作为输入的封装格式(例如FLV、MP4、TS等)对应一个该结构体，如libavformat/flvdec.c的ff\_flv\_demuxer；
* **AVOutputFormat**- 复用器对象，每种作为输出的封装格式（例如FLV, MP4、TS等）对应一个该结构体，如libavformat/flvenc.c的ff\_flv\_muxer；
* **AVStream** - 用于描述一个视频/音频流的相关数据信息。

2. 编解码

* **AVCodecContext** - 描述编解码器上下文的数据结构，包含了众多编解码器需要的参数信息；
* **AVCodec**- 编解码器对象，每种编解码格式(例如H.264、AAC等）对应一个该结构体，如libavcodec/aacdec.c的ff\_aac\_decoder。每个AVCodecContext中含有一个AVCodec；
* **AVCodecParameters** - 编解码参数，每个AVStream中都含有一个AVCodecParameters，用来存放当前流的编解码参数。

3. 网络协议

* **AVIOContext**- 管理输入输出数据的结构体；
* **URLProtocol** - 描述了音视频数据传输所使用的协议，每种传输协议(例如HTTP、RTMP)等，都会对应一个URLProtocol结构，如libavformat/http.c中的ff\_http\_protocol；
* **URLContext** - 封装了协议对象及协议操作对象。

4. 数据存放

* **AVPacket** - 存放编码后、解码前的压缩数据，即ES数据；
* **AVFrame** - 存放编码前、解码后的原始数据，如YUV格式的视频数据或PCM格式的音频数据等；

上述结构体的关系图如下所示(箭头表示派生出)：

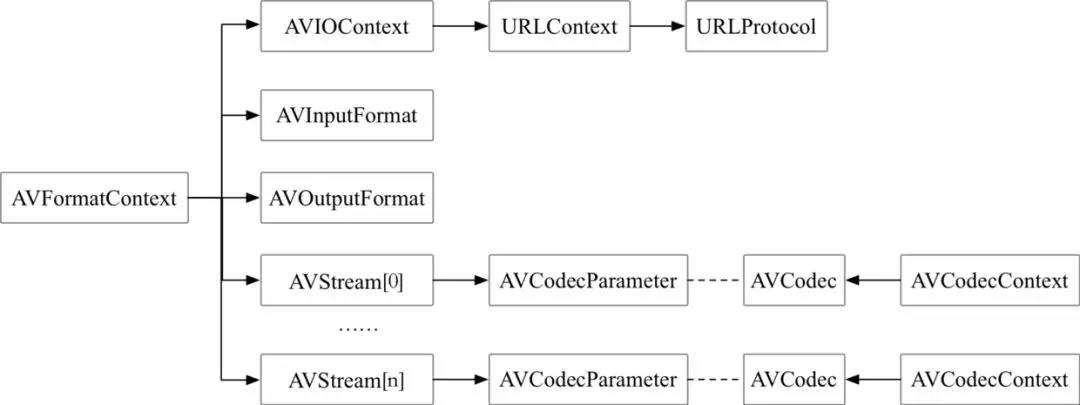
[](http://livevideostack.com/data/attachment/portal/201804/12/210745upzpc8bmm94pc2em.jpg)

图2. FFmpeg结构体关系图

## **代码结构**

下面这段代码完成了读取媒体文件中音视频数据的基本功能，本节以此为例，分析FFmpeg内部代码的调用逻辑。

char \*url = "http://192.168.1.105/test.flv";  
AVPacket pkt;  
int ret = 0;  
//注册复用器、编码器等  
av\_register\_all();  
avformat\_network\_init();  
//打开文件  
AVFormatContext \*fmtCtx = avformat\_alloc\_context();  
ret = avformat\_open\_input(&fmtCtx, url, NULL, NULL);  
ret = avformat\_find\_stream\_info(fmtCtx, NULL);  
//读取音视频数据  
while(ret >= 0)  
{  
  ret = av\_read\_frame(s, &pkt);  
}

### 注册

av\_register\_all函数的作用是注册一系列的(解)复用器、编/解码器等。它在所有基于FFmpeg的应用程序中几乎都是第一个被调用的，只有调用了该函数，才能使用复用器、编码器等。

static void register\_all(void)  
{  
avcodec\_register\_all();  
    /\* (de)muxers \*/  
    ……  
    REGISTER\_MUXDEMUX(FLV,              flv);  
    ……  
}REGISTER\_MUXDEMUX实际上调用的是av\_register\_input\_format和av\_register\_output\_format，通过这两个方法，将(解)复用器分别添加到了全局变量first\_iformat与first\_oformat链表的最后位置。

编/解码其注册过程相同，此处不再赘述。

### 文件打开

FFmpeg读取媒体数据的过程始于avformat\_open\_input，该方法中完成了媒体文件的打开和格式探测的功能。但FFmpeg是如何找到正确的流媒体协议和解复用器呢？可以看到avformat\_open\_input方法中调用了init\_input函数，在这里面完成了查找流媒体协议和解复用器的工作。

static intinit\_input(AVFormatContext \*s, const char \*filename,  
AVDictionary \*\*options)  
{  
int ret;  
    ……  
    if ((ret = s->io\_open(s, &s->pb, filename, AVIO\_FLAG\_READ | s->avio\_flags, options)) < 0)  
        return ret;  
    if (s->iformat)  
        return 0;  
    return av\_probe\_input\_buffer2(s->pb, &s->iformat, filename,  
                                 s, 0, s->format\_probesize);  
}

1.s->io\_open实际上调用的就是io\_open\_default，它最终调用到url\_find\_protocol方法。

static conststructURLProtocol \*url\_find\_protocol(const char \*filename)  
{  
constURLProtocol \*\*protocols;  
    ……  
    protocols = ffurl\_get\_protocols(NULL, NULL);  
    if (!protocols)  
        return NULL;  
    for (i = 0; protocols[i]; i++) {  
constURLProtocol \*up = protocols[i];  
        if (!strcmp(proto\_str, up->name)) {  
av\_freep(&protocols);  
            return up;  
        }  
        if (up->flags & URL\_PROTOCOL\_FLAG\_NESTED\_SCHEME &&  
!strcmp(proto\_nested, up->name)) {  
av\_freep(&protocols);  
            return up;  
        }  
    }  
av\_freep(&protocols);  
    return NULL;  
}ffurl\_get\_protocols可以得到当前编译的FFmpeg支持的所有流媒体协议，通过url的scheme和protocol->name相比较，得到正确的protocol。例如本例中URLProtocol最终指向了libavformat/http.c中的ff\_http\_protocol。

2.av\_probe\_input\_buffer2最终调用到av\_probe\_input\_format3，该方法遍历所有的解复用器，即first\_iformat链表中的所有节点，调用它们的read\_probe()函数计算匹配得分，函数最终返回计算找到的最匹配的解复用器。本例中AVInputFormat最终指向了libavformat/flvdec.c中的ff\_flv\_demuxer。

### 数据读取

av\_read\_frame作用是读取媒体数据中的每个音视频帧，该方法中最关键的地方就是调用了AVInputFormat的read\_packet()方法。AVInputFormat的read\_packet()是一个函数指针，指向当前的AVInputFormat的读取数据的函数。在本例中，AVInputFormat为ff\_flv\_demuxer，也就是说read\_packet最终指向了flv\_read\_packet。

# FLV文件结构解析

FLV(FLASH VIDEO)，是一种常用的文件封装格式，目前国内外大部分视频分享网站都是采用的这种格式。其标准定义为《Adobe Flash Video File Format Specification》。RTMP协议也是基于FLV视频格式的。

FLV的文件格式在该规范中已阐述清楚，本章节不再重复描述，而是结合下面的示例具体阐述如何分析FLV文件。

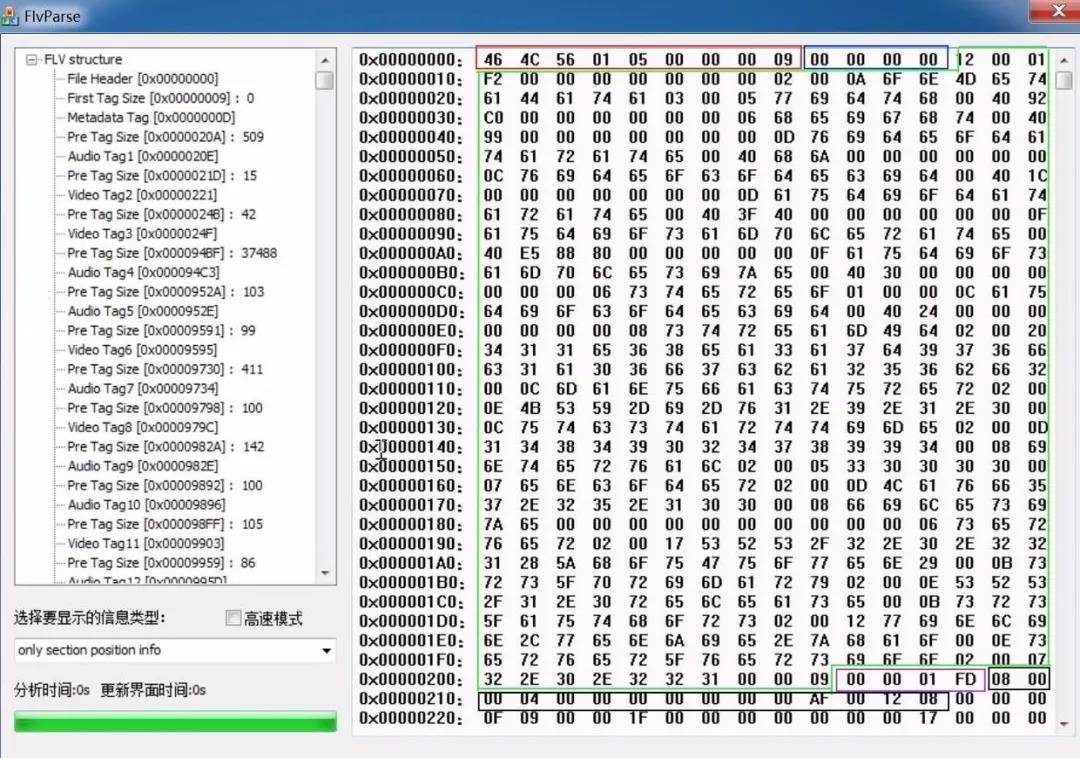
[](http://livevideostack.com/data/attachment/portal/201804/12/212240b5hg9d7g755hhzz2.jpg)

图3. FLV文件结构示例1

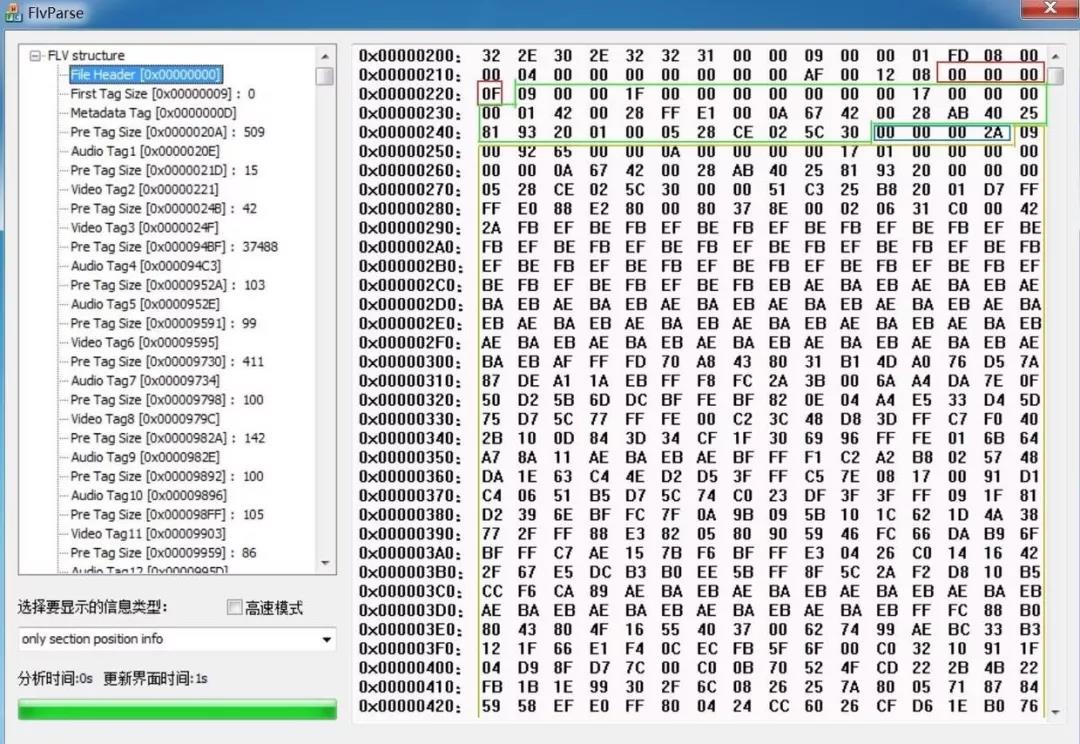
[](http://livevideostack.com/data/attachment/portal/201804/12/212304qiubar3kpaf47dpf.jpg)

图4. FLV文件结构示例2

FLV文件的分析工具有很多，这里给大家推荐FLV Parser这个小软件，通过它可以很容易的看到文件的组成结构。

## **文件结构**

从整个文件上看，FLV是由Header和File Body组成，如下图所示：

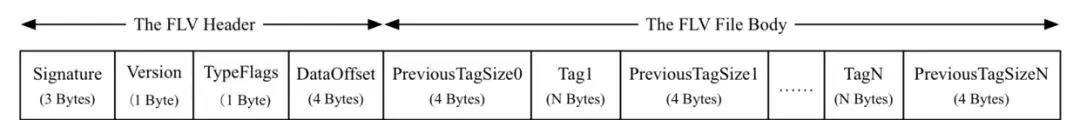
[](http://livevideostack.com/data/attachment/portal/201804/12/212322zgxy0eehng14s45h.jpg)

图5. FLV文件总体结构

1. **FLV Header** - 长度为9，其结构的标准定义参见标准定义见E.2 The FLV header；
2. **FLV File Body** - 由一连串的PreviousTagSize + Tag构成。previousTagSize是4个字节的数据，表示前一个tag的size。标准定义参见E.3 The FLV File Body。

以图3. FLV文件结构示例1为例分析整体结构：

1.位置0x00000000 - 0x00000008, 共9个字节，为FLV Header，其中：

* 0x00000000 - 0x00000002 : 0x46 0x4C 0x56分别表示字符'F''L''V'，用来标识这个文件是FLV格式的。在做格式探测的时候，如果发现前3个字节为“FLV”，就认为它是FLV文件；
* 0x00000003 : 0x01, 表示FLV版本号；
* 0x00000004 : 0x05, 转换为2进制是0000 0101，其中第0位为1，表示存在video，第2位为1，表示存在audio；
* 0x00000005 - 0x00000008 : 0x00 0x00 0x00 0x09，转十进制为9，表示FLV header的长度，当FLV 版本号为1时，该值通常为9。

2.位置0x00000009 - ，为FLV File Body：

* 0x00000009 - 0x0000000C : 0x00 0x00 0x00 0x00 PreviousTagSize0，转十进制为0，该值永远为0；
* 0x0000000D - 0x00000209 : 0x12 ... 0x09，共509个字节，为Tag1的具体内容；
* 0x0000020A - 0x0000020D : 0x00 0x00 0x01 0xFD，转十进制为509，表示它前面的Tag，即Tag1的长度为509；
* 0x0000020E - ：按照Tag + PreviousTagSize的结构依次递推，此处不再举例说明。

## **Tag定义**

FLV File Body是由一系列的PreviousTagSize + Tag组成，其中PreviousTagSize的长度为4个字节，用来表示前一个Tag的长度；Tag里面的数据可能是video、audio或者scripts，其定义参见E.4.1 FLV Tag，结构如下：

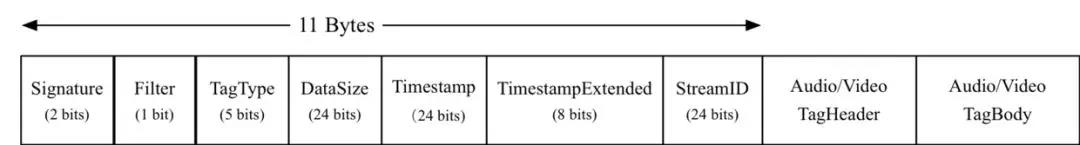
[](http://livevideostack.com/data/attachment/portal/201804/12/212359aovtszxvv7kntwet.jpg)

图6. FLV Tag 结构

以图3. FLV文件结构示例1为例分析Tag结构：

1. 位置0x0000020E : 0x08, 二进制为0000 1000，第5位为0, 表示为非加扰文件；低5位01000为8，说明这个Tag包含的数据类型为Audio；
2. 位置0x0000020F - 0x00000211 : 0x00 0x00 0x04，转十进制为4，说明Tag的内容长度为4，与该tag后面的previousTagSize(15) - 11相同；
3. 位置0x00000212 - 0x00000214 : 0x00 0x00 0x00，转十进制为0，说明当前Audio数据的时间戳为0；
4. 位置0x00000215 : 0x00，扩展时间戳为0，如果扩展时间戳不为0，那么该Tag的时间戳应为：Timestamp | TimestampExtended<<24；
5. 位置0x00000216 - 0x00000218 : 0x00 0x00 0x00，StreamID，总是0；
6. StreamID之后的数据每种格式的情况都不一样，下面会依次进行详细解读。

## **Audio Tags**

如果TAG包中的TagType等于8，表示该Tag中包含的数据类型为Audio。StreamID之后的数据就是AudioTagHeader，其定义详见E.4.2.1 AUDIODATA。结构如下：

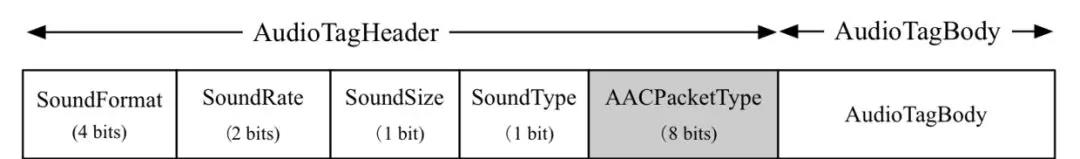
[](http://livevideostack.com/data/attachment/portal/201804/12/212426inn4zzot757xh65s.jpg)

图7. FLV Audio Tag结构

需要说明的是，通常情况下AudioTagHeader之后跟着的就是AUDIODATA数据了，但有个特例，如果音频编码格式为AAC，AudioTagHeader中会多出1个字节的数据AACPacketType，这个字段来表示AACAUDIODATA的类型：

* 0 = AAC sequence header
* 1 = AAC raw。

以图3. FLV文件结构示例为例分析AudioTag结构：

1.位置0x00000219 : 0xAF, 二进制表示为1010 1111：

* 高4位为1010，转十进制为10，表示Audio的编码格式为AAC；
* 第3、2位为11，转十进制为3，表示该音频的采样率为44KHZ；
* 第1位为1，表示该音频采样点位宽为16bits；
* 第0位为1，表示该音频为立体声。

2.位置0x0000021A : 0x00，十进制为0，并且Audio的编码格式为AAC，说明AACAUDIODATA中存放的是AAC sequence header；

3.位置0x0000021B - 0x0000021C : AUDIODATA数据，即AAC sequence header。

### AudioSpecificConfig

AAC sequence header中存放的是AudioSpecificConfig，该结构包含了更加详细的音频信息，《ISO-14496-3 Audio》中的1.6.2.1 章节对此作了详细定义。

通常情况下，AAC sequence header这种Tag在FLV文件中只出现1次，并且是第一个Audio Tag，它存放了解码AAC音频所需要的详细信息。

为什么AudioTagHeader中定义了音频的相关参数，我们还需要传递AudioSpecificConfig呢？

因为当SoundFormat为AAC时，SoundType须设置为1（立体声），SoundRate须设置为3（44KHZ），但这并不意味着FLV文件中AAC编码的音频必须是44KHZ的立体声。播放器在播放AAC音频时，应忽略AudioTagHeader中的参数，并根据AudioSpecificConfig来配置正确的解码参数。

## **Video Tag**

如果TAG包中的TagType等于9，表示该Tag中包含的数据类型为Video。StreamID之后的数据就是VideoTagHeader，其定义详见E.4.3.1 VIDEODATA，结构如下：

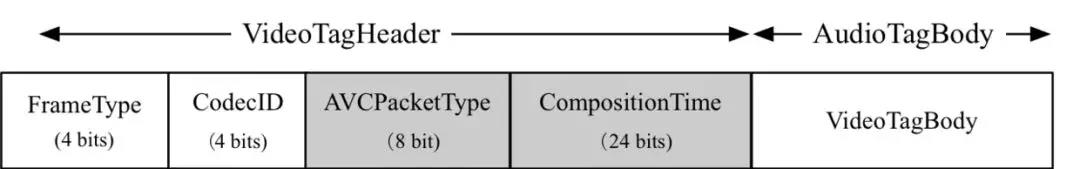
[](http://livevideostack.com/data/attachment/portal/201804/12/212201h074cra3r68jm80a.jpg)

图8. FLV Video Tag结构

VideoTagHeader之后跟着的就是VIDEODATA数据了，但是和AAC音频一样，它也存在一个特例，就是当视频编码格式为H.264的时候，VideoTagHeader会多出4个字节的信息，AVCPacketType和CompositionTime。

* AVCPacketType用来表示VIDEODATA的内容
* CompositonTime相对时间戳，如果AVCPacketType=0x01，为相对时间戳，其它均为0；

以图4. FLV文件结构示例2为例分析VideoTagHeader结构：

1.位置0x0000022C : 0x17, 二进制表示为0001 0111：

* 高4位为0001，转十进制为1，表示当前帧为关键帧；
* 低4位为0111，转十进制为7，说明当前视频的编码格式为AVC。

2.位置0x0000022D : 0x00，十进制为0，并且Video的编码格式为AVC，说明VideoTagBody中存放的是AVC sequence header；

3.位置0x0000022E - 0x00000230 : 转十进制为0，表示相对时间戳为0；

4.位置0x00000231 - 0x0000021C : VIDEODATA数据，即AVC sequence header。

### AVCDecoderConfigurationRecord

AVC sequence header中存放的是AVCDecoderConfigurationRecord，《ISO-14496-15 AVC file format》对此作了详细定义。它存放的是AVC的编码参数，解码时需设置给解码器后方可正确解码。

通常情况下，AVC sequence header这种Tag在FLV文件中只出现1次，并且是第一个Video Tag。

有关AVCDecoderConfigurationRecord结构的代码解析，可以参考中的ff\_isom\_write\_avcc方法。

### CompositionTime(相对时间戳)

相对时间戳的概念需要和PTS、DTS一起理解：

* DTS : Decode Time Stamp，解码时间戳，用于告知解码器该视频帧的解码时间；
* PTS : Presentation Time Stamp，显示时间戳，用于告知播放器该视频帧的显示时间；
* CTS : Composition Time Stamp，相对时间戳，用来表示PTS与DTS的差值。

如果视频里各帧的编码是按输入顺序依次进行的，则解码和显示时间相同，应该是一致的。但在编码后的视频类型中，如果存在B帧，输入顺序和编码顺序并不一致，所以才需要PTS和DTS这两种时间戳。视频帧的解码一定是发生在显示前，所以视频帧的PTS，一定是大于等于DTS的，因此CTS=PTS-DTS。

FLV Video Tag中的TimeStamp，不是PTS，而是DTS，视频帧的PTS需要我们通过DTS + CTS计算得到。

为什么Audio Tag不需要CompositionTime呢？

因为Audio的编码顺序和输入顺序一致，即PTS=DTS，所以它没有CompositionTime的概念。

## **Script Data Tags**

如果TAG包中的TagType等于18，表示该Tag中包含的数据类型为SCRIPT。

SCRIPTDATA 结构十分复杂，定义了很多格式类型，每个类型对应一种结构，详细可参考E.4.4 Data Tags

onMetaData是SCRIPTDATA中一个非常重要的信息，其结构定义可参考E.5 onMetaData。它通常是FLV文件中的第一个Tag，用来表示当前文件的一些基本信息: 比如视音频的编码类型id、视频的宽和高、文件大小、视频长度、创建日期等。