MediaCodec.java createDecoderByType 对于音频 可以设置如下类型

audio/3gpp" - AMR narrowband audio

audio/amr-wb" - AMR wideband audio

**audio/mpeg**" - MPEG1/2 audio layer III

**audio/mp4a-latm**"- AAC audio (note, this is raw AAC packets, not packaged in LATM!)

**这个不是latm打包 是裸ACC数据,也就是要提供raw acc data才能解码 不能封装到latm**

audio/vorbis" - vorbis audio

audio/g711-alaw"- G.711 alaw audio

audio/g711-mlaw"- G.711 ulaw audio

如果是audio/mp4a-latm那么送到MediaCodec解码或者MediaCodec编码的数据 都是 裸ACC数据

可以查看 <http://blog.csdn.net/hiccupzhu/article/details/24370551> AAC元素信息

在AAC中，原始数据块的组成可能有六种不同的元素：

SCE: Single Channel Element单通道元素。单通道元素基本上只由一个ICS组成。一个原始数据块最可能由16个SCE组成。

CPE: Channel Pair Element 双通道元素，由两个可能共享边信息的ICS和一些联合立体声编码信息组成。一个原始数据块最多可能由16个SCE组成。

.....

FIL: Fill Element 填充元素。包含了一些扩展信息。如SBR，动态范围控制信息等。

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。（一个数据块有多个语法元素）**  
          **(7)解元素信息**。

<http://www.cnblogs.com/xiaoshubao/p/5368183.html> Android官方多媒体API Mediacodec翻译（一）

有些格式，特别是ACC音频和MPEG4，H.264和H.265视频格式要求以包含特定数量的构建数据buffers或者codec-specific数据为前缀的实际数据。当处理这样的压缩格式时，这些数据必须在start()方法后和任何帧数据之前提交给编解码器。这些数据必须在调用queueInputBuffer方法时用BUFFER\_FLAG\_CODEC\_CONFIG标记。

Codec-specific数据也可以被包含在传递给configure的ByteBuffer的格式里面，包含的keys是 "csd-0", "csd-1"等。这些keys通常包含在通过MediaExtractor获得的轨道MediaFormat中。这个格式中的Codec-specific数据将在接近start()方法时自动提交给编解码器；你不能显式地提交这些数据。如果这个格式不包含编解码器指定的数据，你也可以选择在这个编解码器中以这个格式所要求的并以正确的顺序传递特定数量的buffers来提交这些数据。还有，你也可以连接所有的codec-specific数据并作为一个单独的codec-config buffer提交。

注意：当编解码器被立即flushed 或start之后不久，并且在任何输出buffer或输出格式变化被返回前需要特别地小心，编解码器的code specific 数据可能会在flush过程中丢失。为保证编解码器的正常运行，你必须在刷新后通过buffers再次提交被标记为BUFFER\_FLAG\_CODEC\_CONFIG buffers的这些数据

编码器（或者产生压缩数据的编解码器）将在任何合法的输出buffer前创建并返回被标记为 codec-config flag的codec specific data 。包含codec-specific-data 的Buffers含有没有意义的时间戳。

/\*\*

304 \* This indicates that the (encoded) buffer marked as such contains

305 \* the data for a key frame.

306 \*

307 \* @deprecated Use {@link #BUFFER\_FLAG\_KEY\_FRAME} instead.

308 \*/

309 public static final int BUFFER\_FLAG\_SYNC\_FRAME = 1;

310

311 /\*\*

312 \* This indicates that the (encoded) buffer marked as such contains

313 \* the data for a key frame.

314 \*/

315 public static final int **BUFFER\_FLAG\_KEY\_FRAME** = 1;

316

317 /\*\*

318 \* This indicated that the buffer marked as such contains codec

319 \* initialization / codec specific data instead of media data.

320 \*/

321 public static final int **BUFFER\_FLAG\_CODEC\_CONFIG** = 2;

322

323 /\*\*

324 \* This signals the end of stream, i.e. no buffers will be available

325 \* after this, unless of course, {@link #flush} follows.

326 \*/

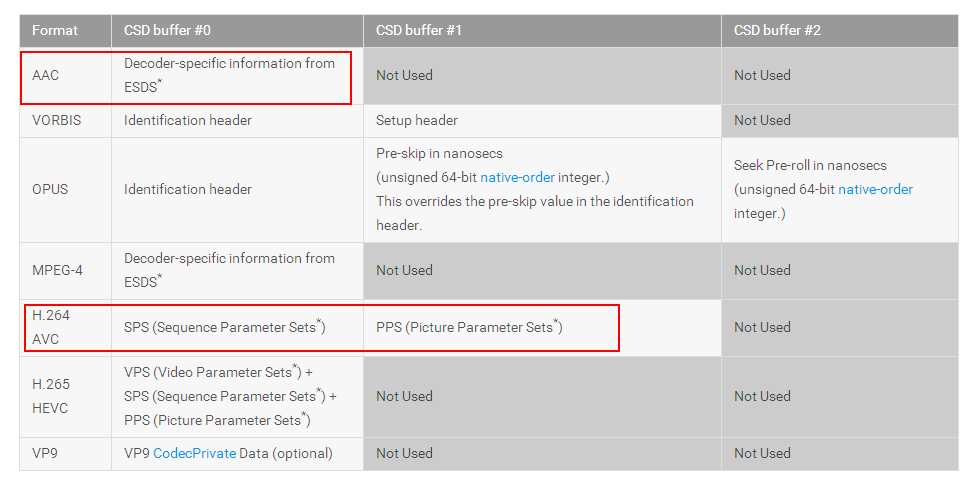
327 public static final int **BUFFER\_FLAG\_END\_OF\_STREAM** = 4;

**public** **final** **void** **[queueInputBuffer](http://androidxref.com/5.1.1_r6/s?refs=queueInputBuffer&project=frameworks)**(

[823](http://androidxref.com/5.1.1_r6/xref/frameworks/base/media/java/android/media/MediaCodec.java" \l "823) **int** [index](http://androidxref.com/5.1.1_r6/s?defs=index&project=frameworks),

[824](http://androidxref.com/5.1.1_r6/xref/frameworks/base/media/java/android/media/MediaCodec.java" \l "824) **int** [offset](http://androidxref.com/5.1.1_r6/s?defs=offset&project=frameworks), **int** [size](http://androidxref.com/5.1.1_r6/s?defs=size&project=frameworks), **long** [presentationTimeUs](http://androidxref.com/5.1.1_r6/s?defs=presentationTimeUs&project=frameworks), **int** [**flags**](http://androidxref.com/5.1.1_r6/s?defs=flags&project=frameworks))

Android 使用以下codec-specific数据buffers。{这些也被要求在轨道配置的格式轨道属性MediaMuxer中进行配置}。所有设置的参数以及被标记为（\*）的codec-specific-data必须以 "\x00\x00\x00\x01"字符开头。



ACC的csd-0来自acc,m4a文件或者mp4文件中的**ESDS**

(https://developer.android.com/reference/android/media/MediaCodec.html)

ACC 的 **ESDS** (Elementary Stream Descriptor).  原始流描述符

t csd-0 contains the ESDS (Elementary Stream Descriptor).

Looking at MakeAACCodecSpecificData in avc\_utils.cpp in the Android sources I see that the first 5 bits are the object type (2), the next 4 are the frequency index (4 = 48000), the next 4 are the channel config (2).  查看Android的源码 MakeAACCodecSpecificData @ avc\_utils.cpp

[This wiki page](http://wiki.multimedia.cx/index.php?title=Understanding_AAC) indicates that the remaining bits are for frame length, depends on core decoder, and extension flag. 下面的链接告诉我们剩余的bit的意义

<https://wiki.multimedia.cx/index.php?title=Understanding_AAC>

I'm using MediaFormat.createAudioFormat() which should setup the important elements of the ESDS.  使用MediaFormat 必须建立ESDS

<http://stackoverflow.com/questions/12942201/decoding-aac-using-mediacodec-api-on-android> 将ESDS转换成ADTS头

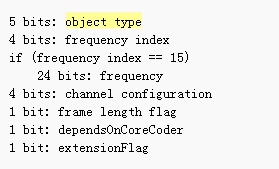
以及设置csd-0 ，把除去  ADTS header的ES送到解码器

csd-0 就是LATM中的 Audio Specific Config (LATM头 可以通过带外SDP config参数传递 也可以通过带内RTP包传输 每个LATM帧最开始)

<https://wiki.multimedia.cx/index.php?title=Understanding_AAC>

大部分AAC数据都会打包到MPEG-4 文件中，包含一个音频'trak' atom，'stsd' description atom，an 'mp4a' atom，an 'esds' atom

有些esds atom 包含了相关AAC流的 setup data ，包含了2个字节：



(如果不是LC，而是SBR，PS ALS 的话，channel configratio和frame lenght flag之间还有其他参数

见 avpriv\_mpeg4audio\_get\_config mpeg4audio.c@libavcodec)

frame length flag:

0: Each packet contains 1024 samples

1: Each packet contains 960 samples

比如 csd-0 = 0x12 0x10 

profile: 2 ACC LC

frequency: 4 44100 Hz 采样率

channel: 2 2个通道

frame length 0 1024samples

见 aacdec\_template.c @libavcodec

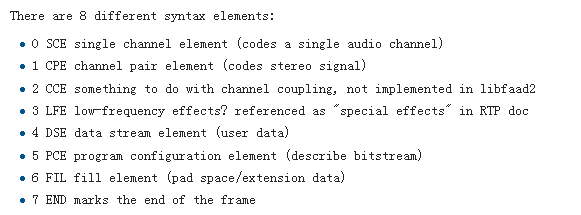
decode\_audio\_specific\_config

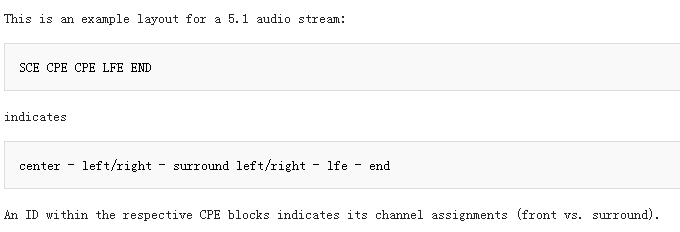
--> avpriv\_mpeg4audio\_get\_config 获取 object\_type sample\_rate

--> decode\_ga\_specific\_config (AOT\_AAC\_LC)

在MPEG-4文件中，AAC数据分成一系列不同长度的帧，构成一个AAC帧的块被叫做语法元素，叫做 syntax elements.( 语法元素)

读取帧流的最开始3bits来查找 element type 元素类型 ，然后解码这个元素，一共有8种元素：





ADTS的全称是Audio Data Transport Stream。是AAC音频的传输流格式

AAC音频格式有ADIF和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。

LC-AAC就是比较传统的AAC，相对而言，主要用于中高码率(>=80Kbps)

HE-AAC(相当于AAC+SBR)主要用于中低码(<=80Kbps)

HE-AACv2(相当于AAC+SBR+PS)主要用于低码率(<=48Kbps）

AAC帧格式及编码介绍 <http://blog.csdn.net/sunnylgz/article/details/7676340> 比较广地介绍了ACC的各种profile差别 以及编解码流程

AAC共有9种规格，以适应不同的场合的需要：

       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，有  Nero ACC 编码器支持

将AAC打包成ADTS格式 ADTS 格式头

<http://blog.csdn.net/bsplover/article/details/7426476>

根据 中国移动多媒体广播CMMB 的格式做 做ADTS 的头部

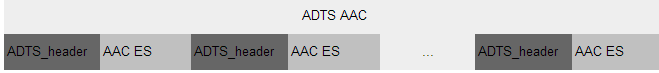
硬件采集 ---> PCM编码[原始量化点|样值] --->分帧[960|1024][生成原始帧raw frame|原始数据块] ---> 封装[ADTS|LATM(较主流)]

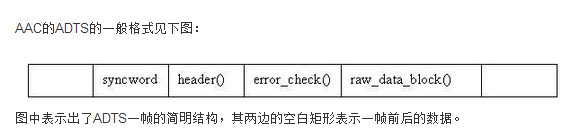
(??960|1024 是PCM样本数目 1024/44.1k = 播放时间 ??)

ADTS格式：short for ：audio data transform stream 对前述原始帧进行ADTS封装就形成ADTS帧，

ADTS帧：帧头+帧净荷组成，帧头存储了音频的相关参数信息：采样率，声道数，帧长度等关键信息[用帧净荷数据的解析解码]。

帧净荷主要有**原始帧组成[960|1024个样值]。可包含1~4个原始帧**





一般的AAC解码器都需要把AAC的ES流打包成ADTS的格式，一般是在AAC ES流前添加7个字节的ADTS header

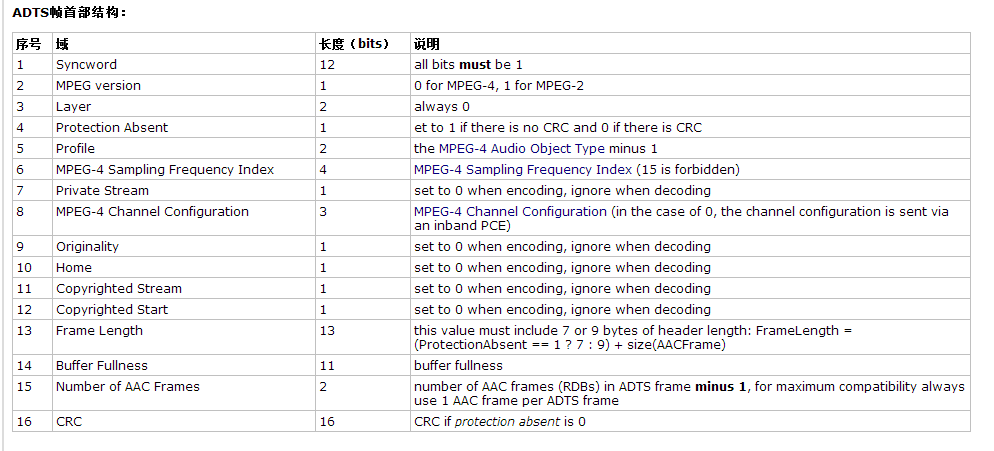
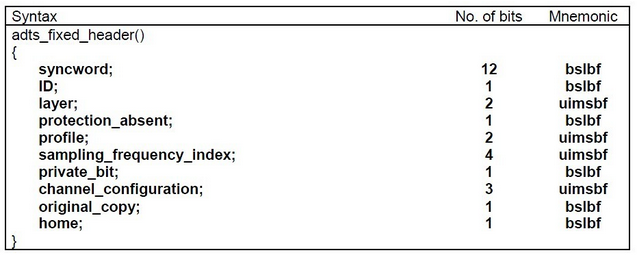
ADTS 头中相对有用的信息  采样率、声道数、帧长度

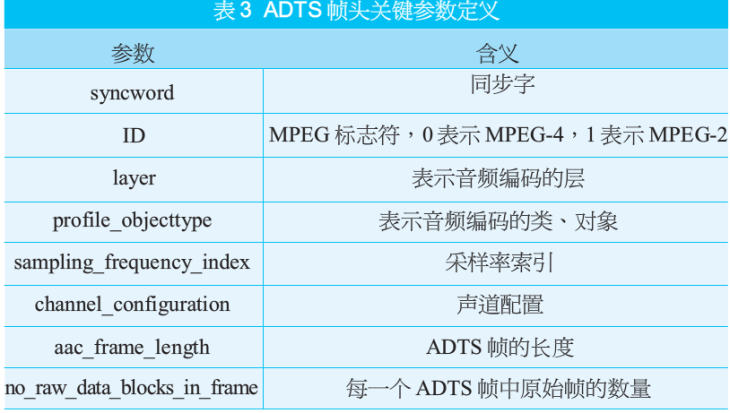
每一个带ADTS头信息的AAC流会清晰的告送解码器他需要的这些信息。

一般情况下ADTS的头信息都是7个字节，分为2部分：

adts\_fixed\_header(); 28bit

adts\_variable\_header(); ~28bit 28+28=56 56/8=7byte



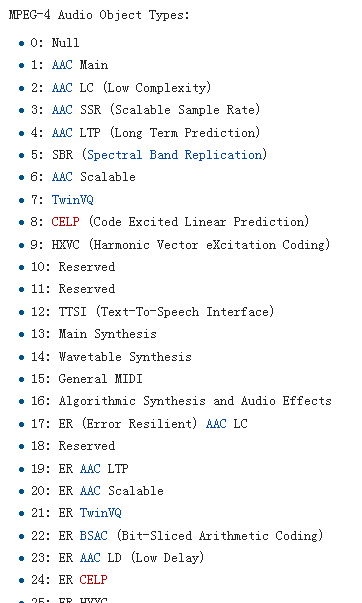
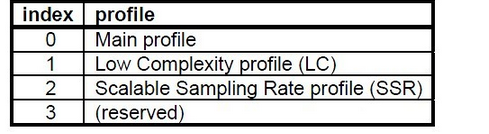
 ADTS可以作为MPEG-2(mp3文件 audio/mpeg ) 或者 MPEG-4(acc文件 audio/?)

**syncword**：同步头 总是0xFFF, all bits must be 1，代表着一个**ADTS帧的开始 1.5 个字节的F开头**

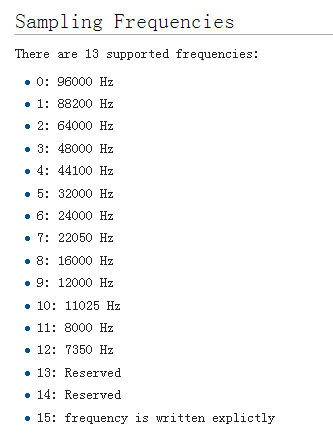
**ID**：MPEG Version: 0 for **MPEG-4**, 1 for **MPEG-2**

**Layer**：always: '00'

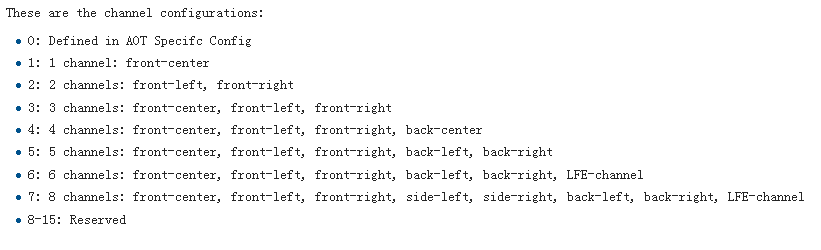
**profile**：表示使用哪个级别的AAC，有些芯片只支持AAC LC 。在**MPEG-2 AAC**中定义了3种： (???? 应该还有MPEG-4 ACC定义的HE???? 可以查看 <https://wiki.multimedia.cx/index.php?title=MPEG-4_Audio>)

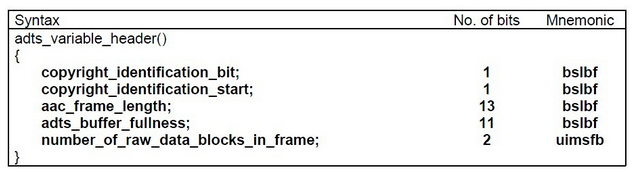
 (Audio Object Type -1 = index )

**sampling\_frequency\_index**：表示使用的采样率下标，通过这个下标在 **Sampling Frequencies[ ]**数组中查找得知采样率的值。



channel\_configuration: 表示声道数





**frame\_length** : 一个ADTS帧的长度 包括ADTS头和AAC原始流(raw data block.)

**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**

number\_of\_raw\_data\_blocks\_in\_frame == 0 表示说**ADTS帧中有一个AAC数据块**并不是说没有。(一个AAC原始帧包含一段时间内**1024个采样**及相关数据) ??? TOM

**???一个ACC原始帧 1024个采样 44100 / 1024 ~= 43 个原始帧 每秒 ???**

用ffmpeg给ACC原始流打包成ADTS格式

我们只需得到封装格式里面关于音频采样率、声道数、元数据长度、aac格式类型等信息。然后在每个AAC原始流前面加上个ADTS头就OK了

ff\_adts\_write\_frame\_header (代码省略)

ACC传输有两种头： ADTS/ADIF，libfaad还有LATM 和LOAS 头

ACC 的 TS流可以使用ADTS和LATM两种封装格式。

在ffmpeg的mpegtsenc中，用了一个amux的AVFormatContext，先把非ADTS的raw aac流写成ADTS或者LATM格式，然后再写入TS流

LATM 的全称为“Low-overhead MPEG-4 Audio TransportMultiplex”（低开销 音频 传输 复用）

MPEG-4 AAC制定的一种高效率的码流传输方式 ， MPEG-2 TS 流也采用LATM 作为AAC 音频码流的封装格式

LATM格式也以帧为单位，主要由AudioSpecificConfig（音频特定配置单元）与音频负载组成

**ADTS帧**转换为**LATM帧**

根据ADTS头的信息，生成StreamMuxConfig，将**ADTS中的原始帧**提取出来，前面加上PayloadLengthInfo做为**LATM的音频帧**

ADTS转换为LATM举例：

1 读取ADTS帧头信息，获得采样率，声道配置，aac算法方面的参数，帧长信息等

2 提取原始帧并保存

3 按照提取的ADTS头参照AAC标准，生成对应的LATM头,即：AudioSpecificConfig头

4.根据原始帧长度信息生成PayloadLengthInfo信息，再由PayloadLengthInfo信息与原始帧数据生成LATM音频负载，再与AudioSpecificConfig联合生成LATM帧。

**(TOM: 按这样的说法，LATM** 也就是换成 一个头 **AudioSpecifiConfig ，**然后加上**长度**PayloadLengthInfo和**原始帧**PayloadMux**)**

**LATM格式也以帧**为单位，主要由**AudioSpecificConfig（音频特定配置单元**）与**音频负载**组成

**1. AudioSpecificConfig** 描述了一个LATM 帧的信息

**2. 音频负载**主要 由若干子帧组成，每个子帧组成 PayloadLengthInfo（负载长度信息）和PayloadMux（负载净荷）组成。

AudioSpecificConfig 信息可以是带内传，也可以是带外传

带内传，就是指每一个LATM 帧，都含有一个AudioSpecificConfig 信息, 可适应音频编码信息不断变化的情况

带外传，则每一个LATM帧都不含有AudioSpecificConfig 信息，而通过其他方式把AudioSpecificConfig信息发送到解码端, AudioSpecificConfig 信 息一般是不变的, 节省音频传输码率

带内或带外传，由muxconfigPresent 标志位决定。

例如流媒体应用中，**muxconfigPresent 可设置为0，这样LATM帧中将不含有AudioSpecificConfig 信息**

**LATM帧**通过**RTP包**发送出去，

**AudioSpecificConfig** 可通过**SDP文件**一次性传送到解码端。

AudioSpecificConfig 主要参数



Audio Specific Config 引自：https://wiki.multimedia.cx/index.php?title=MPEG-4\_Audio  
The Audio Specific Config is the global header for MPEG-4 Audio: AudioSpecificConfig是MPEG-4音频的全局头部 这个就是“csi-0” ESDS (Elementary Stream Descriptor).

Audio -> AOTs(AAC规范定义的音频关键编码信息 Audio Object Types(AOT) ?? AudioSpecificConfig ) -> ESDS (MP4规范中定义的一个box类型,其中包含了AOTs）

Video -> SPS/PPS(H264规范定义的视频关键编码信息) -> AVCC（MP4规范中定义的一个box类型,其中包含了SPS/PPS)

mpeg-4文件有一个音频的trak atom–stsd–mp4a–esds 这个esds有aac流的设置信息

5 bits: object type 基本的object type用5个比特表示  
 if (object type == 31)  
     6 bits + 32: object type  
 4 bits: frequency index 4个比特，用来表示采样率表中的索引号  
 if (frequency index == 15)  
     24 bits: frequency  
 4 bits: channel configuration 4个比特 声道数  
 var bits: AOT Specific Config

ffmpeg 读写header的代码参考

libavcodec\aacenc.c put\_audio\_specific\_config()

libavcodec\mpeg4audio.c avpriv\_mpeg4audio\_get\_config()

从ADTS头得到AudioSpecialConfig

var profile:int = ((payload[2]&0xc0)>>6)+1;

var sample\_rate:int = (payload[2]&0x3c)>>2;

var channel:int = ((payload[2]&0x1)<<2)|((payload[3]&0xc0)>>6); // 上面分别从ADTS头得到profile sample\_rate channle

var config1:int = (profile<<3)|((sample\_rate&0xe)>>1); // 5 + 4 + 4 = 13 bit 两个字节就可以保存audioSpecilConfig

var config2:int = ((sample\_rate&0x1)<<7)|(channel<<3);

var aacSeqHeader:ByteArray = new ByteArray();

aacSeqHeader.writeByte(config1); aacSeqHeader 保存了 两个字节的AudioSpecialConfig

aacSeqHeader.writeByte(config2);

音频负载

由若干子帧组成，每个子帧由PayloadLengthInfo和PayloadMux组成，与ADTS帧净荷一样，音频负载主要包含原始帧数据。

由于每个原始帧没有起始信息，需要PayloadLengthInfo来描述

PayloadMux可以复用**多个节目**的音频流，每个节目可以有**多层音频流**，有**一个或多个原始帧**组成

**AAC打包成TS流通常有两种方式**，分别是先打包成**ADTS或LATM**。

ADTS的每一帧都有个帧头，在每个帧头信息都一样的状况下，会有很大的冗余。

LATM格式具有很大的灵活性，每帧的音频配置单元既可以带内传输，有可以带外传输。

正因为如此，LATM不仅适用于流传输还可以用于RTP传输，RTP传输时，若音频数据配置信息是保持不变，可以先通过SDP会话先传输**StreamMuxConfig（AudioSpecificConfig）**信息，由于LATM流由，一个包含了，一个或多个音频帧，的audioMuxElements序列组成。一个完整或部分完整的audioMuxElement可直接映射到一个RTP负载上。

一个audioMuxElement 一个音频复用元素：

**AudioMuxElement**(muxConfigPresent) <= 在一个RTP包或者 多个RTP包 负载上 ， SDP用cpresent参数对应了muxConfigPresent信息

{

if (muxConfigPresent)

{

useSameStreamMux;

if (!useSameStreamMux) // 使用相同的流复用

// cpresent/muxConfigPresent=1 带内传输时候， audioMuxElement应包括一个指示位useSameStreamMux

**StreamMuxConfig**(); // 发送 AudioSpecificConfig信息

}

if (audioMuxVersionA == 0)

{

for (i = 0; i <= **numSubFrames**; i++) // 子帧

{

**PayloadLengthInfo**(); // 每个子帧由PayloadLengthInfo和PayloadMux组成

**PayloadMux**(); [净荷]

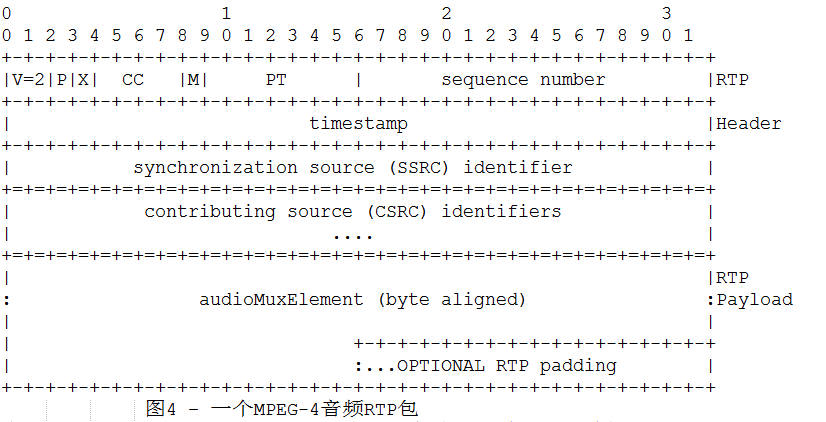
}.

}

}

1. muxConfigPresent(SDP用cpresent):该值为1(带内模式)，audioMuxElement应包括一个指示位"useSameStreamMux"并且可能包括一个音频压缩配置信息"StreamMuxConfig"。**UseSameStreamMux位表示是否前一帧中的StreamMuxConfig元素也应用于本帧**。如果useSameStreamMux位指示要使用前一帧的StreamMuxConfig，而前一帧已经丢失，则将无法对当前帧进行解码。因此，在带内模式下，StreamMuxConfig元素应根据网络条件重复传输。

2. muxConfigPresent(SDP用cpresent) 该值为0 (带外模式)，StreamMuxConfig元素需要通过带外方式传输。如果是SDP，则要使用MIME参数"config"



根据RFC3016，" 5.3 MPEG-4音频MIME类型登记" **SDP中可以给定的参数**：

MIME子类型名: MP4A-LATM

**cpresent:** 一个布尔值参数，表示音频负载配置数据是否已经复用到一个RTP负载

中（参见4.1）。0表示尚未复用，1表示已经复用。该参数的缺省值为1。

**config:** 一个16进制形式的8位字节串，可表示ISO/IEC 14496-3 [5] (参见4.1)

定义的MPEG-4音频负载配置数据"StreamMuxConfig"。该配置信息可按照MSB（最高有效

位）优先原则直接映射到8位字节串。配置数据的第一位应位于第一个8位组的MSB。在

最后一个8位组中,如果需要，应该在配置数据后跟随填充0。

5.4 **SDP usage of MPEG-4 Audio**

MIME媒体类型audio/MP4A-LATM串可以映射到SDP（RFC 2327）的字段上, 如下:

? MIME类型(**audio**)加入SDP"**m=**"中作为媒体名。

? MIME子类型(**MP4A-LATM**)加入SDP"**a=rtpmap**"作为编码名称 <----- TOM: 告诉我们SDP中字段如何填写

? 必需参数"rate"加入"a=rtpmap"的作为时钟速率。

? 可选参数"ptime"加入SDP "a=ptime"属性

? 可选参数"profile-level-id"加入"a=fmtp"行表示编码器能力。参数"object"

加入"a=fmtp" 属性，负载格式相关参数"bitrate", "cpresent"和 "config"

加入"a=fmtp"行。这些参数以分号分隔，按照“参数=值”的成对形式表示MIME媒体类型串。

下面是SDP中媒体表示的例子：

对于6 kb/s的CELP码流 (音频采样频率为8 kHz),

? m=audio 49230 RTP/AVP 96

? a=**rtpmap:**96 MP4A-LATM/8000

? a=**fmtp:**96 profile-level-id=9; object=8; **cpresent=0; (**音频配置数据没有复用到RTP负载中去 )

config=9128B1071070

? a=ptime:20

对于64 kb/s的AAC LC立体声码流(音频采样频率为24 kHz),

? m=audio 49230 RTP/AVP 96

? a=rtpmap:96 MP4A-LATM/24000

? a=fmtp:96 profile-level-id=1; bitrate=64000; cpresent=0;

? **config=**9122620000

在上面两个例子中，音频配置数据仅通过SDP进行了描述，并**没有复用到RTP负载中去**。

此外，"时钟速率（clock rate）"也设置为音频采样速率。

如果时钟速率设置为缺省值，并且必须要取得音频采样速率，则可通过解析参数"config"

来实现。举例如下：

? m=audio 49230 RTP/AVP 96

? a=rtpmap:96 MP4A-LATM/90000

? a=fmtp:96 object=8; cpresent=0; config=9128B1071070

下例显示RTP负载中的音频配置数据。

? m=audio 49230 RTP/AVP 96

? a=rtpmap:96 MP4A-LATM/90000

? a=fmtp:96 object=2; cpresent=1

使用rtp传输的latm封装的aac流，这个LATM格式头是什么样的

StreamMuxConfig +  多个 (PayloadLengthInfo + PayloadMux)

aac-lc主要包含了profileAndLevel、formatType、maxal-sduFrames、audioObjectType、**audioSpecificConfig**这5个参数，

还有其他很多的参数比如maxAudioObject、muxConfigPresent、**streamMuxConfig**、errorProtection\_SpecificConfig、EP\_DataPresent等等，

其中就前面5个是主要的参数主要用于aac-lc( id = 2 low complexity 低复杂)、后面几个参数通常是用于aac-ld (low delay id = 23 低延迟 )

AAC音频能力

[http://rg4.net/archives/1480.html](%20http:/rg4.net/archives/1480.html)

RTP 协议采用的AAC封装格式是LATM格式

AAC的LATM over RTP打包格式定义在RFC 3016。

SDP中几个参数含义：

object，就是AAC的AOT；

cpresent=0，表示StreamMuxConfig不出现在码流中；

config，就是StreamMuxConfig用base16进行编码。

每个RTP包的载荷，最前面是PayloadLengthInfo，每出现一个0xFF表示帧长度+255，直至非0xFF就是剩余的长度；然后就是PayloadMux即AAC的裸流

一共有2种AAC头格式，一种是StreamMuxConfig，另一种是AudioSpecificConfig

 当CMMB中音频压缩标准为AAC时，默认采用LATM封装。**StreamMuxConfig**采用**带外传输**。StreamMuxConifg中的若干默认参数如下：

       audioMuxVersion:0标志流语法版本号为0

       allStreamSameTiemFraming：1，标志复用到PayLoadMux()中的所有负载共享一个共同的时基

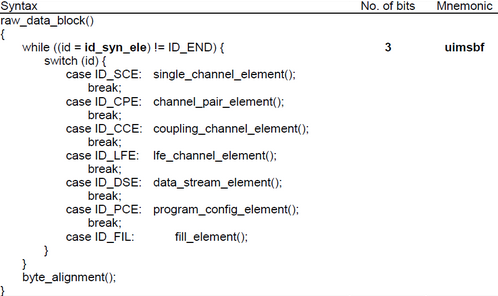
       umSubFrames:0  表示只有一个音频子帧.

       audioObjectType:2 AAC-LC

freameLengthType: 0 帧长度是可变的

       latmBufferFullness:0xFF 码率可变的码流

ACC raw data block 解码：



如果语法元素是 CPE(双通道的音频) 解码流程：

<https://wiki.multimedia.cx/index.php?title=Decoding_AAC_CPE>

（1）SCE: Single Channel Element单通道元素。单通道元素基本上只由一个ICS组成。一个原始数据块最可能由16个SCE组成。

（2）CPE: Channel Pair Element 双通道元素，由两个可能共享边信息的ICS和一些联合立体声编码信息组成。一个原始数据块最多可能由~~16个SCE~~(???TOM???)组成。

ICS：Individual Channel Stream 独立通道流

联合立体声（Joint Stereo）联合立体声的是对原来的取样进行的一定的渲染工作，使声音更”好听”些

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（Joint Stereo），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

AAC是一种VBR(可变码率)的基于块的编码，每个块解码为 **1024 time-domain samples （1024个时域采样点），每个帧的解码是独立的，不依赖于其他帧**

esds box = 长度(含长度本身(4个字节)) + “esds”标签(4个字节) + version&flag (4个字节) + 03 [ 04 [ 05 ] ] + 06

1. esds box中主要是存放Element Stream Descriptors（ESDs），该box的前四个字节为version&flag，一般为0x 00 00 00 00；

2. ESDs中可以分为三层，每层为包含关系，分别为MP4ESDescr，MP4DecConfigDescr，MP4DecSpecificDescr

3. 每层的结构都类似如下：

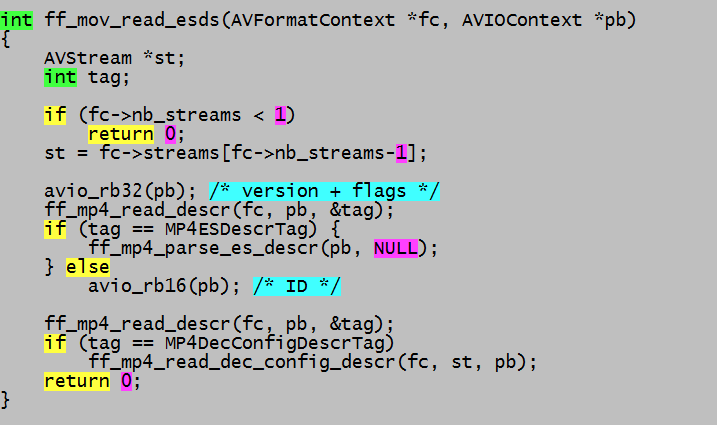
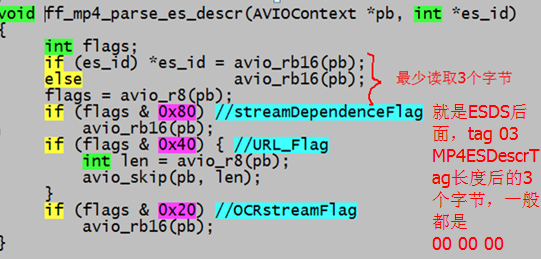
typedef esdsStruct{

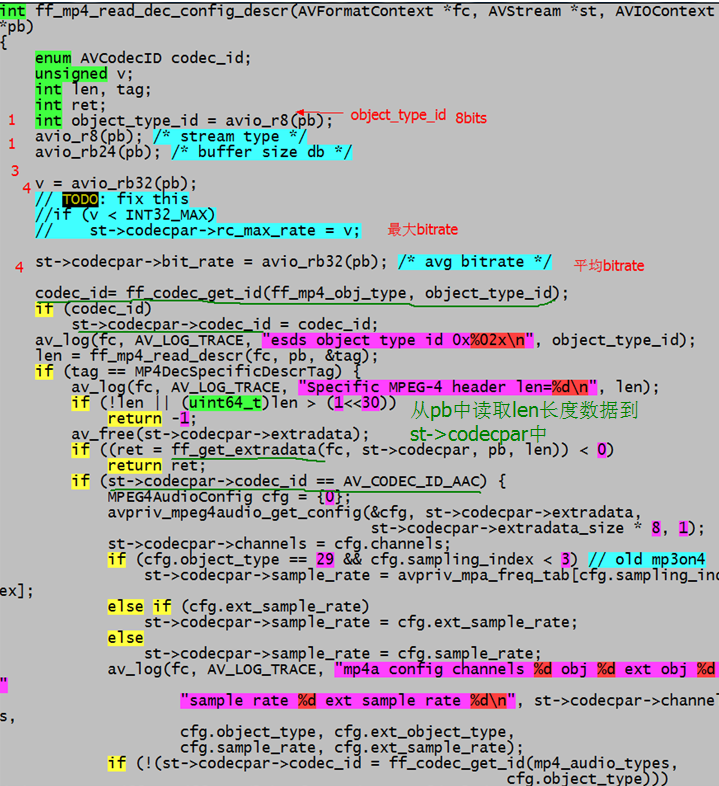
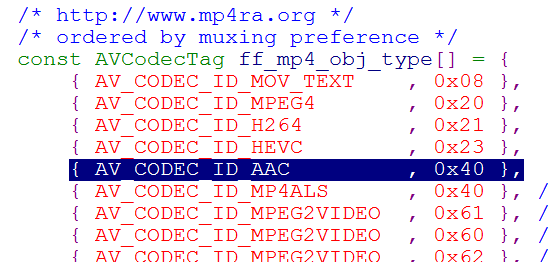
uint8\_t tag;

~~<不定长，最长4字节 >~~ size; // 起始标签后，为data size（不包含tag和size字节数），指示该层data的字节数 长度一个字节

uint8\_t[size] data;

}esdsStruct；

typedef struct MPEG4AudioConfig { 这个就是Audio Special Config @ libavcodec/mpeg4audio.h

int object\_type;

int sampling\_index;

int sample\_rate;

int chan\_config;

int sbr; ///< -1 implicit, 1 presence

int ext\_object\_type;

int ext\_sampling\_index;

int ext\_sample\_rate;

int ext\_chan\_config;

int channels;

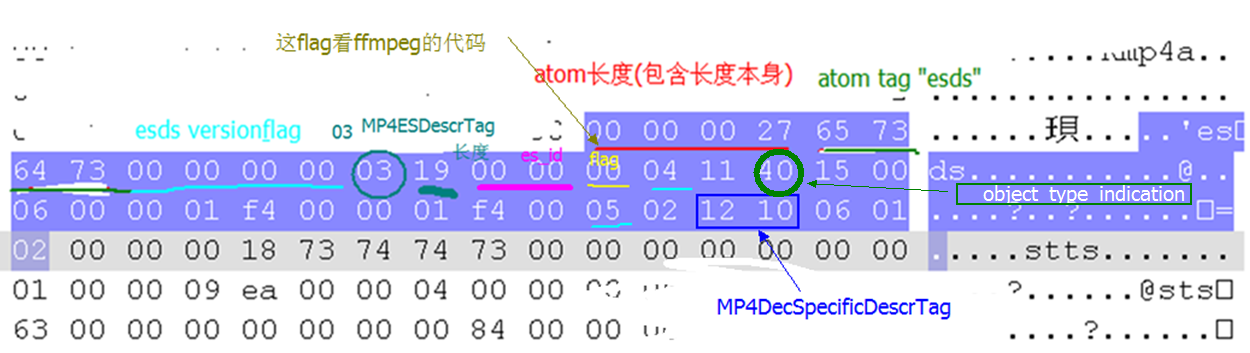
int ps; ///< -1 implicit, 1 presence

int frame\_length\_short;

} MPEG4AudioConfig;

ff\_mp4\_read\_dec\_config\_descr @libavformat/isom.c

--> avpriv\_mpeg4audio\_get\_config(&cfg, st->codecpar->extradata, st->codecpar->extradata\_size \* 8, 1)



libavformat/mov.c objectTypeIndication

ff\_mp4\_parse\_es\_descr

ff\_mp4\_read\_dec\_config\_descr

libavformat/isom.c

libavformat/isom.h

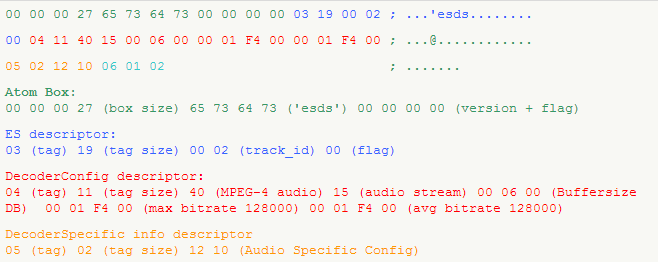
#define MP4ESDescrTag 0x03

#define MP4DecConfigDescrTag 0x04 (包含了一些最高bitrate 平均bitrate )

#define MP4DecSpecificDescrTag 0x05 (就是Audio Specif Config)

#define MP4SLDescrTag 0x06 <=与MP4ESDescrtTag同级关系

mp4 stsd/mp4a/esds atom的分析<http://doublescn.appspot.com/?p=1134002>





Audio Specific Config:

5 bits: object type

4 bits: frequency index

4 bits: channel configuration

1 bit: frame length flag

1 bit: dependsOnCoreCoder

1 bit: extensionFlag

Audio Object Types

MPEG-4 Audio Object Types:

0: Null

1: AAC Main

2: AAC LC (Low Complexity)

3: AAC SSR (Scalable Sample Rate)

4: ...

Sampling Frequencies

0: 96000 Hz

1: 88200 Hz

2: 64000 Hz

3: 48000 Hz

4: 44100 Hz

5: ...

Channel Configurations

0: Defined in AOT Specifc Config

1: 1 channel: front-center

2: 2 channels: front-left, front-right

3: 3 channels: front-center, front-left, front-right

4: ...

frame length flag:

0: Each packet contains 1024 samples

1: Each packet contains 960 samples

Example:

1210h = 00010010 00010000b = 00010(AAC LC) 010 0 (44100Hz) 0010 (strero) 0 (1024 samples) 00 ( ? )

const AVCodecTag ff\_mp4\_obj\_type[] = { // MP4DecConfigDescr 04 长度(1个字节) + data[0]( obj\_type) + 其他data

{ CODEC\_ID\_MOV\_TEXT , 0x08 },

{ CODEC\_ID\_MPEG4 , 0x20 },

{ CODEC\_ID\_H264 , 0x21 },

{ CODEC\_ID\_AAC , 0x40 },

{ CODEC\_ID\_MP4ALS , 0x40 }, /\* 14496-3 ALS \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x61 }, /\* MPEG2 Main \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x60 }, /\* MPEG2 Simple \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x62 }, /\* MPEG2 SNR \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x63 }, /\* MPEG2 Spatial \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x64 }, /\* MPEG2 High \*/

{ CODEC\_ID\_MPEG2VIDEO, 0x65 }, /\* MPEG2 422 \*/

{ CODEC\_ID\_AAC , 0x66 }, /\* MPEG2 AAC Main \*/

{ CODEC\_ID\_AAC , 0x67 }, /\* MPEG2 AAC Low \*/

{ CODEC\_ID\_AAC , 0x68 }, /\* MPEG2 AAC SSR \*/

{ CODEC\_ID\_MP3 , 0x69 }, /\* 13818-3 \*/

{ CODEC\_ID\_MP2 , 0x69 }, /\* 11172-3 \*/

{ CODEC\_ID\_MPEG1VIDEO, 0x6A }, /\* 11172-2 \*/

{ CODEC\_ID\_MP3 , 0x6B }, /\* 11172-3 \*/

{ CODEC\_ID\_MJPEG , 0x6C }, /\* 10918-1 \*/

{ CODEC\_ID\_PNG , 0x6D },

{ CODEC\_ID\_JPEG2000 , 0x6E }, /\* 15444-1 \*/

{ CODEC\_ID\_VC1 , 0xA3 },

{ CODEC\_ID\_DIRAC , 0xA4 },

{ CODEC\_ID\_AC3 , 0xA5 },

{ CODEC\_ID\_DTS , 0xA9 }, /\* mp4ra.org \*/

{ CODEC\_ID\_VORBIS , 0xDD }, /\* non standard, gpac uses it \*/

{ CODEC\_ID\_DVD\_SUBTITLE, 0xE0 }, /\* non standard, see unsupported-embedded-subs-2.mp4 \*/

{ CODEC\_ID\_QCELP , 0xE1 },

{ CODEC\_ID\_MPEG4SYSTEMS, 0x01 },

{ CODEC\_ID\_MPEG4SYSTEMS, 0x02 },

{ CODEC\_ID\_NONE , 0 },

};

mp3: mpegaudiodec\_template.c (libavcodec)

acc: aacdec\_template.c (libavcodec)