CSlunatic

【转】打包AAC码流到FLV文件

AAC编码后数据打包到FLV很简单。

1. FLV音频Tag格式

字节位置 意义

0x08, // 0, TagType

0xzz, 0xzz, 0xzz, // 1-3, DataSize,

0xzz, 0xzz, 0xzz, 0xzz, // 4-6, 7 TimeStamp | TimeStampExtend

0x00, 0x00, 0x00, // 8-10, StreamID

0xzz, // 11, AudioTag Header

0x0b, // 12, AACPacketType (如果不是AAC编码 没有这个字节)

0xzz ... 0xzz // 音频数据

2. AudioTagHeader

音频Tag头一般由一个字节定义（AAC用两个字节），第一个字节的定义如下：

音频格式 4bits | 采样率 2bits | 采样精度 1bits | 声道数 1bits|

音频格式 4bits

0x00 = Linear PCM, platform endian

0x01 = ADPCM

0x02 = MP3

0x03 = Linear PCM, little endian

0x04 = Nellymoser 16-kHz mono

0x05 = Nellymoser 8-kHz mono

0x06 = Nellymoser

0x07 = G.711 A-law logarithmic PCM

0x08 = G.711 mu-law logarithmic PCM

导航

博客园
首页
新随笔
联系
订阅XML
管理

公告

昵称：CSlunatic
园龄：5年9个月
粉丝：35
关注：0
+加关注

< 2019年1月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

0x09 = reserved
0x0A = AAC
0x0B = Speex
0x0E = MP3 8-Khz
0x0F = Device-specific sound

采样率 2bits
0 = 5.5-kHz
1 = 11-kHz
2 = 22-kHz
3 = 44-kHz
对于AAC总是3，这里看起来FLV不支持48K AAC，其实不是的，后面还是可以定义为48K。

采样精度 1bits
0 = snd8Bit
1 = snd16Bit
压缩过的音频都是16bit

声道数 1bits
0 = sndMono
1 = sndStereo
对于AAC总是1

综上，如果是AAC 48K 16比特精度 双声道编码，该字节为 0b1010 1111 = 0xAF。

看第2个字节，如果音频格式AAC（0x0A），AudioTagHeader中会多出1个字节的数据AACPacketType，这个字段来表示AACAUDIODATA的类型：
0x00 = AAC sequence header，类似h.264的sps,pps，在FLV的文件头部出现一次。
0x01 = AAC raw，AAC数据

3. AAC Sequence header
AAC sequence header定义AudioSpecificConfig，AudioSpecificConfig包含着一些更加详细的音频信息，它的定义在ISO14496-3中1.6.2.1。

Linux网络编程(18)
Linux文件系统(4)
MacOS(4)
uboot(4)
图像分析(7)
网页设计(9)
音视频(28)
硬件设计(3)
杂文(9)

随笔档案

2018年11月 (2)
2018年9月 (2)
2018年8月 (2)
2018年6月 (2)
2017年8月 (1)
2017年7月 (2)
2017年6月 (1)
2017年5月 (3)
2017年4月 (2)
2017年2月 (7)
2017年1月 (3)
2016年12月 (2)
2016年11月 (10)
2016年10月 (1)
2016年9月 (1)
2016年8月 (4)
2016年7月 (3)
2016年6月 (1)
2016年5月 (3)
2016年1月 (2)
2015年7月 (1)
2015年6月 (4)
2015年5月 (4)
2015年4月 (2)
2015年3月 (2)
2015年2月 (1)
2015年1月 (1)
2014年12月 (4)
2014年10月 (2)
2014年9月 (2)
2014年8月 (4)
2014年7月 (1)
2014年6月 (11)
2014年5月 (12)
2014年4月 (22)
2014年3月 (8)
2014年2月 (2)
2014年1月 (2)
2013年12月 (3)
2013年11月 (2)
2013年10月 (2)

简化的AudioSpecificConfig 2字节定义如下:

AAC Profile 5bits | 采样率 4bits | 声道数 4bits | 其他 3bits |

AAC Profile 5bits，参考ISO-14496-3 Object Profiles Table

AAC Main 0x01

AAC LC 0x02

AAC SSR 0x03

...

（为什么有些文档看到profile定义为4bits，实际验证是5bits）

采样率 4bits

Value samplingFrequencyIndex

0x00 96000

0x01 88200

0x02 64000

0x03 48000

0x04 44100

0x05 32000

0x06 24000

0x07 22050

0x08 16000

0x09 12000

0x0A 11025

0x0B 8000

0x0C reserved

0x0D reserved

0x0E reserved

0x0F escape value

声道数 4bits

0x00 - defined in audioDecderSpecificConfig

0x01 单声道（center front speaker）

0x02 双声道（left, right front speakers）

0x03 三声道（center, left, right front speakers）

0x04 四声道（center, left, right front speakers, rear surround speakers）

0x05 五声道（center, left, right front

2013年8月 (16)
2013年7月 (7)
2013年6月 (6)
2013年5月 (6)
2013年4月 (6)
2013年3月 (4)

最新评论

1. Re:Linux usb子系统（一）：子系统架构
写的真的好！整个流程梳理的很清晰
--Eagle-jie
2. Re:Linux SD/MMC/SDIO驱动分析
这篇文章分析的太透彻了，现在SDIO确实是比较常用的。
--kangear
3. Re:Live555 分析（一）：类介绍
学习了
--qin2013
4. Re:U-Boot 目录结构和编译过程
！！！！！！！！！！！！！！！
赞！！！！！！！！！！
--晨初

阅读排行榜

1. ALSA音频工具amixer,aplay,arecord(17271)
2. 【转】如何在VMware上安装macOS Sierra 10.12(16587)
3. Linux SD/MMC/SDIO驱动分析(12437)
4. 【转】Alsa音频编程【精华】(8280)
5. Linux usb子系统（三）：通过usbfs操作设备的用户空间驱动(6571)

评论排行榜

1. U-Boot 目录结构和编译过程(1)
2. Live555 分析（一）：类介绍(1)
3. Linux usb子系统（一）：子系统架构(1)
4. Linux SD/MMC/SDIO驱动分析(1)

推荐排行榜

1. Linux--start_kernel()函数分析(2)
2. U-Boot 目录结构和编译过程(2)
3. U-Boot 启动过程和源码分析(第一阶段)(1)
4. LINUX下 Udev详解(1)
5. 【转】做生意和打工的区别(1)

speakers, left surround, right surround rear speakers)
0x06 5.1声道（center, left, right front speakers, left surround, right surround rear speakers, front low frequency effects speaker)
0x07 7.1声道（center, left, right center front speakers, left, right outside front speakers, left surround, right surround rear speakers, front low frequency effects speaker)
0x08-0x0F - reserved

其他3bits设置为0即可。

AAC-LC， 48000，双声道 这样的设置 Sequence header 为 0b 00010 0011 0010 000 = 0x11 0x90。
因此 AAC Sequence header的整个音频Tag包为 0x08, 00 00 04, 00 00 00 00, 00 00 00, AF 00 11 90 | 00 00 00 0F

AAC Sequence header这个音频包有些FLV文件里面没有也可以正确解码。但对于RTMP播放，必须要在发送第一个音频数据包前发送这个header包。

4. AAC音频包
结构为：0x08, 3字节包长度，4字节时间戳，00 00 00，AF 01 N字节AAC数据 | 前包长度
其中编码后AAC纯数据长度为N，3字节包长度 = N + 2

前包长度 = 11 + 3字节包长度 = 11 + N + 2 = 13 + N。

FLV格式非常简单，头信息数据量很少，适合网络传输，因此被广泛的应用。
1. H264 NALU结构
　h264 NALU：0x00 00 00 01 | nalu_type(1字节)| nalu_data (N 字节) |

0x00 00 00 01 | ...

起始码(4字节) 类型 数据 下一个NALU起始码

H264 NALU固定以 0x00 00 00 01为起始，NALU_data部分不会出现这个起始码；

在找到下一个起始码之前，当前NALU数据长度不知；

NALU_type 1字节，定义为：1比特禁止位 | 2比特 重要性指示位 | 5比特 类型

固定为0 11重要 不能少 1-12 由h264使用

00不重要 可以丢弃

几个常用Nalu_type:

0x67 (0 11 00111) SPS 非常重要 type = 7

0x68 (0 11 01000) PPS 非常重要 type = 8

0x65 (0 11 00101) IDR帧 关键帧 非常重要 type = 5

0x41 (0 10 00001) P帧 重要 type = 1

0x01 (0 00 00001) B帧 不重要 type = 1

0x06 (0 00 00110) SEI 不重要 type = 6

2. FLV tag

前面讲过FLV文件就是由无数个Tag组成的，Tag有Video Tag, Audio Tag和Script Tag.

A/V Tag里面存储的就是音视频编码数据，Script Tag里面是一些码流描述信息。

理论上来说，不解析Script tag也可以对A/V Tag完整解码。tag的固定格式是：

Tag Type(1字节) | DataSize(3字节) | TimeStamp(3字节) | TimeStampExtended (1字节)| StreamID (3) | ...

下面将分别介绍各种NALU封到tag里面的

结构。

2. 一般Video tag

```
                                    字节位置
意义
0x09,                           // 0,
TagType
0xzz, 0xzz, 0xzz,               // 1-3,
DataSize,
0xzz, 0xzz, 0xzz, 0xzz,     // 4-6, 7
TimeStamp | TimeStampExtend
0x00, 0x00, 0x00,               // 8-10,
StreamID

0xz7,                            // 11,
FrameType | CodecID
0x01,                            // 12,
AVCPacketType
0x00, 0x00, 0x00,                // 13-15,
CompositionTime

0xzz, 0xzz, 0xzz, 0xzz,         // 16-19,
NaluLength    NBytes
0xzz, ...., ...., 0xzz,             // NBytes,
NaluData

0xzz, 0xzz, 0xzz, 0xzz.         // N+1-
N+3, PreviousTagSize
```

其中 0xzz的意思是该字节根据实际情况付不同的值

2.1 DataSize[0,1,2] = NaluLength + 5 + 4;

5 是 AVCPacket头5比特, （FrameType+CodecID | AVCPacketType | CompositionTime）

4 是写入NaluLength

2.2 对于一个裸h264流，没有时间戳的概念，可以默认以25fps，即40ms一帧数据。

```
int cts = 0;
TimeStamp[0,1,2]   = cts[0,1,2];
TimeStampExtend[0] = cts[3];
cts += 40;
```

2.3 if(nalu_type == IDR)
FrameType | CodecID = 0x17;
else
FrameType | CodecID = 0x27;

2.4 NaluLength就是nalu长度，然后紧跟N字节的Nalu数据。

2.5 PreviousTagSize在这里计算最为方便，PreviousTagSize = 11 + 5 + 4 + NaluLength

11 是video tag头数据 （TagType到StreamID）

IDR，I，P，B帧的NALU都是这个结构

3. SPS/PPS NALU

SPS和PPS在FLV里面称为序列头信息sequence header，它的AVCPacketType为0x00

字节位置 意义

```
0x09,                       // 0, TagType
0xzz, 0xzz, 0xzz,           // 1-3, DataSize,
0x00, 0x00, 0x00, 0x00,     // 4-6, 7; TimeStamp | TimeStampExtend
0x00, 0x00, 0x00,           // 8-10, StreamID

                            // AVC video tage header 5Bytes
0x17,                       // 11, FrameType | CodecID
0x00,                       // 12, AVCPacketType
0x00, 0x00, 0x00,           // 13-15, CompositionTime

                            // AVCDecoderConfigurationRecord 6 Bytes
0x01,                       // 16, ConfigurationVersion
0xzz,                       // 17,
```

```
AVC Profile
0x00,                       // 18, profile_compatibility
0xzz,                       // 19, AVC Level
0xFF,                       // 20, lengthSizeMinusOne,
                            // reserved 6bits | NAL unit length-1, commonly be 3
0xzz,                       // 21, numOfSequenceParameterSets,
                            // reserved 3bits | SPS count, commonly be 1

0xzz, 0xzz,                 // 22-23, SPS0 Length N0 Byte
0xzz, ...., 0xzz,           // N0 Byte SPS0 Data
0xzz, 0xzz,                 // SPSm Length Nm Byte (如果存在) 循环存放最多31个SPS
0xzz, ...., 0xzz,           // Nm Byte SPSm Data

0xzz,                       // PPS count
0xzz, 0xzz,                 // PPS0 Length
0xzz, ...., 0xzz,           // N0 Byte PPS0 Data
0xzz, 0xzz,                 // PPSm Length Nm Byte (如果存在) 循环存放最多255个PPS
0xzz, ...., 0xzz,           // Nm Byte PPSm Data

0xzz, 0xzz, 0xzz, 0xzz. // N+1-N+3, PreviousTagSize
```

3.1 在H.264码流里面reserved bit一般为0; 而在FLV码流里面reserved bit定义为1

3.2 在H.264里面 SPS和PPS是对立的NALU，但是在FLV里面会把他们统一写在一

个Video Tag里面。

而且这个tag必须是FLV里面第一个Video Tag,否则接收到其他video tag也没法解码.

为了防止SPS,PPS数据丢失，有些编码器会在每个IDR帧之前重复发SPS,PPS。这些SPS其实是一样的。

但也不排除有些变态的编码器前后的SPS会不同，比较标准容许这样做。

这样就需要首先遍历一边h264码流，将其中不同的SPS,PPS提起出来，先记录下来，然后再统一写到FLV。

也可以大胆一点接收到第一个SPS和第一个PPS后就结束这个遍历，就当作码流里面只有一个SPS和一个PPS。

3.3 DataSize=5 + // AVC video tag header (FrameType + CodecID | .. CompositionTime)
6 + //
AVCDecoderConfigurationRecord
SPSCount*2 +
// 每个SPS长度2字节
各个 SPSDataLength +
// 所有SPS数据长度和
1 +
// PPS个数
PPSCount*2
+ // 每个PPS长度2字节
各个 PPSDataLength;
// 所有PPS数据长度和

3.4 AVC Profile和 AVC Level就等于SPS NALU里面第1字节和第3字节 （第0字节为NaluType）

3.5 lengthSizeMinusOne，这个定义没有理解，不知道低2比特是什么含义，看到很多文档里面就直接设为0b11, 所有这个字节为0xFF

3.6 numOfSequenceParameterSets, 低5比特是SPS个数，H.264标准里面定义最多SPS个数为255,这里只有31。

不知道会不会存在问题，当然一般情况下就一个SPS,该值为 0xE1 (0b111

00001)

3.7 每个SPS，PPS数据长度都用两个字节来表述，

3.8 这个tag的 PreviousTagSize = 11 + DataSize。 11 是Video tag （TagType到StreamID）

4. FLV头

```
'F', 'L', 'V',                          // 0-2 FLV file Signature, also can be 'f''l''v'
0x01,                                   // FLV version,
0x0z,                                   // AV tag Enable.  0x05 AV both, 0x03 audio only, 0x01 video only
0x00, 0x00, 0x00, 0x09,                 // Length of this header.
0x00, 0x00, 0x00, 0x00.                 // PreviousTagLength.
```

5. SEI NALU

SEI是H.264里面的附加增强信息NALU，他对解析解码没有帮助，但提供一些编码器控制参数等信息。

FLV没有一个Tag单独包含SEI数据，它把SEI数据和紧随其后那个视频NALU数据打在同一个Video Tag里面。

包含SEI数据的VideoTag结构如下

```
                                字节位置    意义
0x09,                           // 0,       TagType
0xzz, 0xzz, 0xzz,               // 1-3,     DataSize,
0xzz, 0xzz, 0xzz, 0xzz,         // 4-6, 7;  TimeStamp | TimeStampExtend
0x00, 0x00, 0x00,               // 8-10,    StreamID

0x27,                           // 11,      FrameType | CodecID
0x01,                           // 12,      AVCPacketType
```

```
0x00, 0x00, 0x00,              // 13-
15,  CompositionTime

0xzz, 0xzz, 0xzz, 0xzz,        // 16-19,
SEILength  NBytes
0xzz, ...., ...., 0xzz,        //
NBytes,  SEIData

0xzz, 0xzz, 0xzz, 0xzz,        //
NaluLength  NBytes
0xzz, ...., ...., 0xzz,        //
NBytes,  NaluData

0xzz, 0xzz, 0xzz, 0xzz.    //
PreviousTagSize
```

5.1 DataSize[0,1,2] = （NaluLength + 5 + 4） + （SEILength + 4）;

6. 得到NALU代码

```
// 输入： H264_fp 264文件指针
// 输出： 找到的Nalu长度,
        *nalu_type返回找到的NALU类型
int h264_get_nalu(FILE *h264_fp,
uint8_t *nalu_type) {
   int start_pos = -1;
   int nalu_size = 0;
   int zero_num = 0;
   uint8_t tmp;
   while(!feof(h264_fp)){
      fread(&tmp, 1, 1, h264_fp);
      if(tmp == 0) zero_num++;
      else if(tmp == 1) {
         if(zero_num >= 3) {
            if(start_pos == -1) {
               start_pos =
ftell(h264_fp);
               fread(nalu_type, 1, 1,
h264_fp);
            } else {
               nalu_size =
ftell(h264_fp) - start_pos - 4;
               fseek(h264_fp, start_pos,
0);
               break;
```

```
            }
        }
    } else
        zero_num = 0;
  }
  return nalu_size;
}
```

Overview

Flash Video(简称**FLV**),是一种流行的网络格式。目前国内外大部分视频分享网站都是采用的这种格式.

File Structure

从整个文件上开看,FLV是由**The FLV header** 和 **The FLV File Body** 组成.

1.The FLV header

Field	Type	Comment
Signature	UI8	Signature byte always 'F' (0x46)
Signature	UI8	Signature byte always 'L' (0x4C)
Signature	UI8	Signature byte always 'V' (0x56)
Version	UI8	File version (for example, 0x01 for FLV version 1)
TypeFlagsReserved	UB[5]	Shall be 0
TypeFlagsAudio	UB[1]	1 = Audio tags are present

Type Flags Reserved	UB [1]	Shall be 0
Type Flags Video	UB [1]	1 = Video tags are present
Data Offset	UI32	The length of this header in bytes

Signature: FLV 文件的前3个字节为固定的'**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.

Version: 第4个字节表示flv版本号.

Flags: 第5个字节中的第0位和第2位,分别表示 video 与 audio 存在的情况.(1表示存在,0表示不存在)

DataOffset：最后4个字节表示FLV header 长度.

2.The FLV File Body

Field	**Type**	**Comment**
PreviousTagSize0	UI32	Always 0
Tag1	FLVTAG	First tag
PreviousTagSize1	UI32	Size of previous tag, including its header, in bytes. For FLV version1,

		this value is 11 plus the DataSize of the previous tag.
Tag2	FLV TAG	Second tag
...	...	...
PreviousTagSizeN-1	UI32	Size of second-to-last tag, including its header, in bytes.
TagN	FLV TAG	Last tag
PreviousTagSizeN	UI32	Size of last tag, including its header, in bytes

FLV header之后,就是 **FLV File Body**.

FLV File Body是由一连串的back-pointers + tags构成.back-pointers就是4个字节数据,表示前一个tag的size.

FLV Tag Definition

FLV文件中的数据都是由一个个TAG组成,TAG里面的数据可能是video、audio、scripts.

下表是TAG的结构:

1.FLVTAG

Field	**Type**	**Comment**
Reserved	UB[2]	Reserved for FMS, should be

		0
Filter	UB [1]	Indicates if packets are filtered. 0 = No pre-processing required. 1 = Pre-processing (such as decryption) of the packet is required before it can be rendered. Shall be 0 in unencrypted files, and 1 for encrypted tags. See Annex F. FLV Encryption for the use of filters.
TagType	UB [5]	Type of contents in this tag. The following types are defined: 8 = audio 9 = video 18 = script data
DataSize	UI24	Length of the message. Number of bytes after StreamID to end of tag (Equal to length of the tag – 11)
Times	UI2	Time in

tamp	4	milliseconds at which the data in this tag applies. This value is relative to the first tag in the FLV file, which always has a timestamp of 0.
TimestampExtended	UI8	Extension of the Timestamp field to form a SI32 value. This field represents the upper 8 bits, while the previous Timestamp field represents the lower 24 bits of the time in milliseconds.
StreamID	UI24	Always 0.
AudioTagHeader	IF TagType == 8 AudioTagHeader	
VideoTagHeader	IF TagType == 9	

	VideoTagHeader	
EncryptionHeader	IF Filter == 1 EncryptionTagHeader	
FilterParams	IF Filter == 1 FilterParams	
Data	IF TagType == 8 AUDIODATA IF TagType == 9 VIDEO	Data specific for each media type.

	DATA IF Tag Type == 18 SCRIPT DATA	

TagType: TAG中第1个字节中的前5位表示这个TAG中包含数据的类型,8 = audio,9 = video,18 = script data.

DataSize:StreamID之后的数据长度.

Timestamp和**TimestampExtended**组成了这个TAG包数据的PTS信息,记得刚开始做FVL demux的时候，并没有考虑TimestampExtended的值,直接就把Timestamp默认为是PTS，后来发生的现 象就是画面有跳帧的现象,后来才仔细看了一下文档发现真正数据的PTS是PTS= Timestamp | TimestampExtended<<24.

StreamID之后的数据就是每种格式的情况不一样了，接下格式进行详细的介绍.

Audio Tags

如果TAG包中的**TagType**==8时，就表示这个TAG是audio。

StreamID之后的数据就表示是**AudioTagHeader**，**AudioTagHeader**结构如下:

Field	**Type**	**Comment**
Sound Format	UB [4]	Format of SoundData. The following values

		are defined: 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 Formats 7, 8, 14, and 15 are reserved. AAC is supported in Flash Player 9,0,115,0 and higher. Speex is supported in Flash Player 10 and higher.
Sound Rate	UB [2]	Sampling rate. The following values are defined: 0 = 5.5 kHz 1 = 11 kHz 2 = 22 kHz 3 = 44 kHz

Sound Size	UB [1]	Size of each audio sample. This parameter only pertains to uncompressed formats. Compressed formats always decode to 16 bits internally. 0 = 8-bit samples 1 = 16-bit samples
Sound Type	UB [1]	Mono or stereo sound 0 = Mono sound 1 = Stereo sound
AACPacketType	IF SoundFormat == 10 UI8	The following values are defined: 0 = AAC sequence header 1 = AAC raw

AudioTagHeader的头1个字节，也就是接跟着**StreamID**的1个字节包含着音频类型、采样率等的基本信息.表里列的十分清楚.

AudioTagHeader之后跟着的就是**AUDIODATA**数据了，也就是audio payload 但是这里有个特例，如果音频格式（SoundFormat）是10 = 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。

Field	Type	Comment
Data	IF AACPacketType==0 AudioSpecificConfig	The AudioSpecificConfig is defined in ISO14496-3. Note that this is not the same as the contents of the esds box from an MP4/F4V file.
	ELSE IF AACPacketType== 1 Raw AAC frame data in UI8 [ ]	audio payload

AAC sequence header也就是包含了**AudioSpecificConfig**，**AudioSpecificConfig**包含着一些更加详细音频的信息，AudioSpecificConfig的定义在**ISO14496-3**中**1.6.2.1 AudioSpecificConfig**，这里就不详细贴了。而且在**ffmpeg**中有对AudioSpecificConfig解析的函数，**ff_mpeg4audio_get_config**(),可以对比的看一下，理解更深刻。

AAC raw 这种包含的就是音频ES流了，也就是audio payload.

在FLV的文件中，一般情况下 **AAC sequence header** 这种包只出现1次，而且是第一个audio tag，为什么要提到这种tag，因为当时在做FLVdemux的时候，如果是AAC的音频，需要在每帧AAC ES流前边添加7个字节**ADST**头,**ADST**在音频的格式中会

详细解读，这是解码器通用的格式，就是AAC的纯ES流要打包成ADST格式的AAC文件，解码器才能正常播放.就是在打包ADST的时候，需要**samplingFrequencyIndex**这个信息，samplingFrequencyIndex最准确的信息是在**AudioSpecificConfig**中，所以就对AudioSpecificConfig进行解析并得到了samplingFrequencyIndex。

到这步你就完全可以把FLV 文件中的音频信息及数据提取出来，送给音频解码器正常播放了。

Video Tags

如果TAG包中的**TagType**==9时，就表示这个TAG是video.

StreamID之后的数据就表示是**VideoTagHeader**，**VideoTagHeader**结构如下：

Field	**Type**	**Comment**
Frame Type	UB [4]	Type of video frame. The following values are defined: 1 = key frame (for AVC, a seekable frame) 2 = inter frame (for AVC, a non-seekable frame) 3 = disposable inter frame (H.263 only) 4 = generated key frame (reserved for server use only) 5 = video info/command frame
Cod	UB	Codec Identifier.

ecID	[4]	The following values are defined: 2 = Sorenson H.263 3 = Screen video 4 = On2 VP6 5 = On2 VP6 with alpha channel 6 = Screen video version 2 7 = AVC
AVC PacketType	IF CodecID == 7 UI8	The following values are defined: 0 = AVC sequence header 1 = AVC NALU 2 = AVC end of sequence (lower level NALU sequence ender is not required or supported)
Composition Time	IF CodecID == 7 SI24	IF AVCPacketType == 1 Composition time offset ELSE 0 See ISO 14496-12, 8.15.3 for an explanation of composition times. The offset in an FLV file is always in milliseconds.

VideoTagHeader的头1个字节，也就是接跟着**StreamID**的1个字节包含着视频帧类型及视频CodecID最基本信息.表里列的十分清楚.

VideoTagHeader之后跟着的就是**VIDEODATA**数据了，也就是video payload.当然就像音频AAC一样，这里也有特例就是如果视频的格式是AVC（H.264）的话，**VideoTagHeader**会多出4个字节的信息.

AVCPacketType 和 CompositionTime。AVCPacketType 表示接下来 **VIDEODATA**（**AVCVIDEOPACKET**）的内容：

IF AVCPacketType == 0 **AVCDecoderConfigurationRecord**（AVC sequence header）
IF AVCPacketType == 1 **One or more NALUs (Full frames are required)**

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.

AVCDecoderConfigurationRecord的定义在**ISO 14496-15**, 5.2.4.1中，这里不在详细贴，

SCRIPTDATA

如果TAG包中的**TagType**==18时，就表示这个TAG是*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.

Field	Type	Comment

Type	UI8	Type of the ScriptDataValue. The following types are defined: 0 = Number 1 = Boolean 2 = String 3 = Object 4 = MovieClip (reserved, not supported) 5 = Null 6 = Undefined 7 = Reference 8 = ECMA array 9 = Object end marker 10 = Strict array 11 = Date 12 = Long string
ScriptDataValue	IF Type == 0 DOUBLE IF Type == 1 UI8 IF Type == 2 SCRIPTDATASTRING IF Type == 3 SCRIPTDATA	Script data value. The Boolean value is (ScriptDataValue ≠ 0).

	OBJECT IF Type == 7 UI16 IF Type == 8 SCRIPTDATAECMA ARRAY IF Type == 10 SCRIPTDATASTRICTARRAY IF Type == 11 SCRIPTDATADATE IF Type == 12 SCRIPTDATALONGSTRING	

类型在FLV的官方文档中都有详细介绍.

onMetaData

onMetaData 是*SCRIPTDATA*中对我们来说十分重要的信息，结构如下表：

Property Name	**Type**	**Comment**
audiocodecid	Number	Audio codec ID used in the file (see E.4.2.1 for available SoundFormat values)
audiodatarate	Number	Audio bit rate in kilobits per second
audiodelay	Number	Delay introduced by the audio codec in seconds
audiosamplerate	Number	Frequency at which the audio stream is replayed
audiosamplesize	Number	Resolution of a single audio sample
canSeekToEnd	Boolean	Indicating the last video frame is a key frame
creationdate	String	Creation date and time
dura	Num	Total duration of

tion	ber	the file in seconds
filesize	Number	Total size of the file in bytes
framerate	Number	Number of frames per second
height	Number	Height of the video in pixels
stereo	Boolean	Indicating stereo audio
videocodecid	Number	Video codec ID used in the file (see E.4.3.1 for available CodecID values)
videodatarate	Number	Video bit rate in kilobits per second
width	Number	Width of the video in pixels

这里面的duration、filesize、视频的width、height等这些信息对我们来说很有用.

keyframes

当时在做flv demux的时候，发现官方的文档中并没有对*keyframe*s index做描述，但是flv的这种结构每个tag又不像TS有同步头，如果没有*keyframe*s index 的话，seek及快进快退的效果会非常差，因为需要一个tag一个tag的顺序读取。后来通过网络查一些资料，发现了一个*keyframes*的信息藏在*SCRIPTDATA*中。

keyframes几乎是一个**非官方**的标准，也就是民间标准.在网上已经很难看到flv文件格式，但是metadata里面不包含 keyframes 项目的视频．两个常用的操作metadata的工

具是flvtool2和FLVMDI，都是把keyframes作为一个默认的元信息项目.在FLVMDI的主页(http://www.buraks.com/flvmdi/)上有描述：

keyframes: (Object) This object is added only if you specify the /k switch. 'keyframes' is known to FLVMDI and if /k switch is not specified, 'keyframes' object will be deleted.
'keyframes' object has 2 arrays: 'filepositions' and 'times'. Both arrays have the same number of elements, which is equal to the number of key frames in the FLV. Values in times array are in 'seconds'. Each correspond to the timestamp of the n'th key frame. Values in filepositions array are in 'bytes'. Each correspond to the fileposition of the nth key frame video tag (which starts with byte tag type 9).

也就是说**keyframes**中包含着2个内容'filepositions' and 'times'分别指的是关键帧的文件位置和关键帧的PTS.通过**keyframes**可以建立起自己的Index，然后再seek和快进快退的操作中，快速有效的跳转到你想要找的关键帧的位置进行处理。

rtmpdump可以下载rtmp流并保存成flv文件。
如果要对流中的音频或视频单独处理，需要根据flv协议分别提取。
简单修改rtmpdump代码，增加相应功能。
1 提取音频:
rtmpdump程序在Download函数中循环下载:

```
....
 do
 {
....
nRead = RTMP_Read(rtmp, buffer, bufferSize);
....
}while(!RTMP_ctrlC && nRead > -1 &&
```

```
RTMP_IsConnected(rtmp) &&
!RTMP_IsTimedout(rtmp));
....
```

原程序是收到后写文件，生成flv。
现在，在写之前分别提取音视频，提取音频比较简单，直接分析buffer(参考RTMP_Write函数里的方法).
注意的是，rtmpdump里用的是RTMP_Read来接收，注意它的参数。为了方便，也可以直接用RTMP_ReadPacket。后面的视频使用RTMP_ReadPacket来接收并处理。

```
int RTMP_Write2(RTMP *r, const char
*buf, int size)
{
 RTMPPacket *pkt = &r->m_write;
 char *pend, *enc;
 int s2 = size, ret, num;


 if (size < 11) {
  /* FLV pkt too small */
  return 0;
 }


 if (buf[0] == 'F' && buf[1] == 'L' &&
buf[2] == 'V')
  {
   buf += 13;
   s2 -= 13;
  }


 pkt->m_packetType = *buf++;
 pkt->m_nBodySize =
AMF_DecodeInt24(buf);
 buf += 3;
 pkt->m_nTimeStamp =
AMF_DecodeInt24(buf);
 buf += 3;
 pkt->m_nTimeStamp |= *buf++ <<
24;
 buf += 3;
```

```
 s2 -= 11;


 if (((pkt->m_packetType ==
RTMP_PACKET_TYPE_AUDIO
            || pkt->m_packetType ==
RTMP_PACKET_TYPE_VIDEO) &&
          !pkt->m_nTimeStamp) || pkt-
>m_packetType ==
RTMP_PACKET_TYPE_INFO)
  {
    pkt->m_headerType =
RTMP_PACKET_SIZE_LARGE;
    if (pkt->m_packetType ==
RTMP_PACKET_TYPE_INFO)
     pkt->m_nBodySize += 16;
  }
 else
  {
    pkt->m_headerType =
RTMP_PACKET_SIZE_MEDIUM;
  }


BYTE outbuf2[640];
int nLen2 = 640;


AVManager::GetInstance()-
>Decode((BYTE*)(pkt->m_body+1),
pkt->m_nBodySize-1, outbuf2, nLen2);
//实际音频内容为pkt->m_body+1，大小
是pkt->m_nBodySize-1。这里的声音是
speex编码。
为什么跳过第一字节，可以参考:
http://bbs.rosoo.net/thread-16488-1-
1.html

evt_OnReceivePacket((char*)outbuf2,
nLen2);//回调出来



RTMPPacket_Free(pkt);
pkt->m_nBytesRead = 0;
```

2
视频处理
可以参考rtmpsrv.c
把nRead = RTMP_Read(rtmp, buffer, bufferSize);改成：

```
RTMPPacket pc = { 0 }, ps = { 0 };
 bool bFirst = true;
while (RTMP_ReadPacket(rtmp, &pc))
{
if (RTMPPacket_IsReady(&pc))
 {
    if (pc.m_packetType == RTMP_PACKET_TYPE_VIDEO && RTMP_ClientPacket(rtmp, &pc))
   {
      bool bIsKeyFrame = false;
    if (result == 0x17)//I frame
   {
      bIsKeyFrame = true;
   }
   else if (result == 0x27)
   {
      bIsKeyFrame = false;
   }
static unsigned char const start_code[4] = {0x00, 0x00, 0x00, 0x01};
fwrite(start_code, 1, 4, pf );
//int ret = fwrite(pc.m_body + 9, 1, pc.m_nBodySize-9, pf);


if( bFirst) {


//AVCsequence header


//ioBuffer.put(foredata);


//获取sps
```

```
int spsnum = data[10]&0x1f;

int number_sps = 11;

int count_sps = 1;

while (count_sps<=spsnum){

int spslen =
(data[number_sps]&0x000000FF)<<8
|(data[number_sps+1]&0x000000FF);

number_sps += 2;

fwrite(data+number_sps, 1, spslen, pf
);
fwrite(start_code, 1, 4, pf );

//ioBuffer.put(data,number_sps,
spslen);
//ioBuffer.put(foredata);

number_sps += spslen;

count_sps ++;

}

//获取pps

int ppsnum = data[number_sps]&0x1f;
```

```
int number_pps = number_sps+1;

int count_pps = 1;

while (count_pps<=ppsnum){

int ppslen =
(data[number_pps]&0x000000FF)
<<8|data[number_pps+1]&0x000000F
F;

number_pps += 2;

//ioBuffer.put(data,number_pps,ppslen
);

//ioBuffer.put(foredata);

fwrite(data+number_pps, 1, ppslen, pf
);
fwrite(start_code, 1, 4, pf );

number_pps += ppslen;

count_pps ++;

}

bFirst =false;

} else {
```

```
//AVCNALU

int len =0;

int num =5;

//ioBuffer.put(foredata);

while(num<pc.m_nbodysize)
 {

len =(data[num]&0x000000FF)<<24|
(data[num+1]&0x000000FF)<<16|
(data[num+2]&0x000000FF)
<<8|data[num+3]&0x000000FF;

num = num+4;

//ioBuffer.put(data,num,len);

//ioBuffer.put(foredata);

fwrite(data+num, 1, len, pf );
fwrite(start_code, 1, 4, pf );

num = num + len;

}

}
```

```
 }
}
```

嵌入式QQ交流群：127085086

分类：Linux程序设计,Linux网络编程,音视频

CSlunatic
关注 - 0
粉丝 - 35
+加关注

0 0

« 上一篇：USB Video Class及其实现
» 下一篇：【转】AAC ADTS格式分析
posted on 2016-11-08 14:16 CSlunatic 阅读(1609) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

发表评论

昵称：
兜兜有糖的博客
评论内容：

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]