Bill_Hoo 的BLOG

发私信 加友情链接

博主的更多文章>>

原创 【基于libRTMP的流媒体直播之 AAC、H264 推送】

2014-09-24 11:46:23

标签：FLV 直播 H264 RTMP AAC

这段时间在捣腾基于 RTMP 协议的流媒体直播框架，其间参考了众多博主的文章，剩下一些细节问题自行琢磨也算摸索出个门道，现将自己认为比较恼人的 AAC 音频帧的推送和解析、H264 码流的推送和解析以及网上没说清楚的地方分享给各位。

RTMP 协议栈的实现，Bill 直接使用的 libRTMP，关于 libRTMP 的编译、基本使用方法，以及简单的流媒体直播框架，请参见博文[C++实现RTMP协议发送H.264编码及AAC编码的音视频]，言简意赅，故不再赘述。

言归正传，我们首先来看看 AAC 以及 H264 的推送。

不论向 RTMP 服务器推送音频还是视频，都需要按照 FLV 的格式进行封包。因此，在我们向服务器推送第一个 AAC 或 H264 数据包之前，需要首先推送一个音频 Tag [AAC Sequence Header] 以下简称“音频同步包”，或者视频 Tag [AVC Sequence Header] 以下简称“视频同步包”。

AAC 音频帧的推送

我们首先来看看音频 Tag，根据 FLV 标准 Audio Tags 一节的描述：

AUDIODATA

Field	Type	Comment
SoundFormat (see notes following table)	UB[4] 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	Format of SoundData Formats 7, 8, 14, and 15 are reserved for internal use AAC is supported in Flash Player 9,0,115,0 and higher. Speex is supported in Flash Player 10 and higher.
SoundRate	UB[2] 0 = 5.5-kHz 1 = 11-kHz 2 = 22-kHz 3 = 44-kHz	Sampling rate For AAC: always 3
SoundSize	UB[1] 0 = snd8Bit 1 = snd16Bit	Size of each sample. This parameter only pertains to uncompressed formats. Compressed formats always decode to 16 bits internally. 0 = snd8Bit 1 = snd16Bit
SoundType	UB[1] 0 = sndMono 1 = sndStereo	Mono or stereo sound For Nellymoser: always 0 For AAC: always 1
SoundData	UI8[size of sound data]	if SoundFormat == 10 AACAUDIODATA else Sound data—varies by format

AACAUDIODATA

Field	Type	Comment
AACPacketType	UI8	0: AAC sequence header 1: AAC raw
Data	UI8[n]	if AACPacketType == 0 AudioSpecificConfig else if AACPacketType == 1 Raw AAC frame data

我们可以将其简化并得到 *AAC* 音频同步包的格式如下：

AAC 音频同步包 4Bytes

AUDIODATA		Binaray	Hex.	简称
4bits SoundFormat		[AAC] 1010	0xA	2Bytes AACDecoderSpecific
2bits SoundRate		[44kHz] 11	0xF	
1bit SoundSize		[snd16bit] 1		
1bit SoundType		[sndStereo] 1		
3Bytes SoundData [AACAudioData]	1Byte AACPacketType	00000000	0x00	
	5bits AudioObjectType	[AAC LC] 00010	0x1210	2Bytes AudioSpecificConfig
	4bits SampleRateIndex	[44100] 0100		
	4bits ChannelConfig	[Stereo] 0010		
	1bit FrameLengthFlag	0		
	1bit dependOnCoreCoder	0		
	1bit extensionFlag	0		

音频同步包大小固定为 4 个字节。前两个字节被称为 [AACDecoderSpecificInfo]，用于描述这个音频包应当如何被解析。后两个字节称为 [AudioSpecificConfig]，更加详细的指定了音频格式。

[AACDecoderSpecificInfo] 俩字节可以直接使用 FAAC 库的 faacEncGetDecoderSpecificInfo 函数来获取，也可以根据自己的音频源进行计算。一般情况下，双声道，44kHz 采样率的 AAC 音频，其值为 0xAF00，示例代码：

```
int RtmpLiveEncoderImpl::getAacDecoderSpecificInfo(
    RtmpLiveCodecByteTy (&spec_info)[2], bool is_metadata){
    // AACDecoderSpecificInfo
    // AAC | sample rate 44kHz | sample size 16bit | channels stereo
    //  1010          11              1               1
    // 0x  A                         F
    spec_info[0] = 0xAF;
    spec_info[1] = is_metadata ? 0x00 : 0x01;

    return RtmpLiveCodecErr::NO_ERR;
}
```


根据 FLV 标准 不难得知，[AACDecoderSpecificInfo] 第 1 个字节高 4 位 |1010| 代表音频数据编码类型为 AAC，接下来 2 位 |11| 表示采样率为 44kHz，接下来 1 位 |1| 表示采样点位数 16bit，最低 1 位 |1| 表示双声道。其第二个字节表示数据包类型，0 则为 AAC 音频同步包，1 则为普通 AAC 数据包。

音频同步包的后两个字节 [AudioSpecificConfig] 的结构，援引其他博主图如下：

长度	字段	说明
5 bit	audioObjectType	编码结构类型，AAC-LC为2
4 bit	samplingFrequencyIndex	音频采样率索引值, 44100对应值4
4 bit	channelConfiguration	音频输出声道，2
	GASpecificConfig	该结构包含以下三项
1 bit	frameLengthFlag	标志位，用于表明IMDCT窗口长度, 0
1 bit	dependsOnCoreCoder	标志位，表明是否依赖于corecoder, 0
1 bit	extensionFlag	选择了AAC-LC, 这里必须为0

我们只需参照上述结构计算出对应的值即可。至此，4 个字节的音频同步包组装完毕，便可推送至 RTMP 服务器，示例代码如下：

```
RtmpLiveCodecByteTy body[4] = { 0 };
int i = 0;

// Get AacDecoderSpecificInfo.
char spec_info[2] = { 0 };
getAacDecoderSpecificInfo(spec_info, true);

body[i++] = spec_info[0];
body[i++] = spec_info[1];

// AudioSpecificConfig.
uint16_t audio_specific_config = 0;
audio_specific_config |= ((2 << 11) & 0xF800); // 2: AAC LC(Low Complexity).
audio_specific_config |= ((4 << 7) & 0x0780); // 4: 44kHz.
audio_specific_config |= ((2 << 3) & 0x78); // 2: Stereo.
audio_specific_config |= 0 & 0x07; // Padding: 000

body[i++] = (audio_specific_config >> 8) & 0xFF;
body[i++] = audio_specific_config & 0xFF;

if(!sendRtmpPacket(body, i, RTMP_PACKET_TYPE_AUDIO
    return RtmpLiveCodecErr::ENCODER_SEND_RTMP_PAC
}
```


网上有博主说音频采样率小于等于 44100 时 SamplingFrequencyIndex 应当选择 3（48kHz），Bill 测试发现采样率等于 44100 时设置标记为 3 或 4 均能正常推送并在客户端播放，不过我们还是应当按照标准规定的行事，故此处的 SamplingFrequencyIndex 选 4。

完成音频同步包的推送后，我们便可向服务器推送普通的 AAC 数据包，推送数据包时，[AACDecoderSpecificInfo] 则变为 0xAF01，向服务器说明这个包是普通 AAC 数据包。后面的

数据为 AAC 原始数据去掉前 7 个字节（若存在 CRC 校验，则去掉前 9 个字节），我们同样以一张简化的表格加以阐释：

AAC 音频数据包 (2 + N) Bytes

AUDIODATA		Binaray	Hex.	简称
4bits SoundFormat		[AAC] 1010	0xA	2Bytes AACDecoderSpecific
2bits SoundRate		[44kHz] 11	0xF	
1bit SoundSize		[snd16bit] 1		
1bit SoundType		[sndStereo] 1		
SoundData [AACAudioData]	1Byte AACPacketType	[AAC raw]	0x01	
	N Bytes AAC Raw Data Without Header N = [AAC Raw data length] - ([has CRC] ? 9 : 7)			

推送普通 AAC 数据包的示例代码：

```
int i = 0;
const size_t body_size = 2 + aac_raw_data_length - 7;
RtmpLiveCodecByteTy *body = new(std::nothrow) RtmpLiveCodecByteTy[body_size];
if(body == NULL){
    return RtmpLiveCodecErr::MEMORY_BAD_ALLOC;
}

// Get AACDecoderSpecificInfo.
char spec_info[2] = { 0 };
getAacDecoderSpecificInfo(spec_info, false);

body[i++] = spec_info[0];
body[i++] = spec_info[1];

// Aac raw data without 7bytes frame header.
memcpy(&body[i], data + 7, aac_raw_data_length - 7);

if(!sendRtmpPacket(body, body_size, RTMP_PACKET_TYPE_AUDIO, timestamp)){
    delete[] body;
    return RtmpLiveCodecErr::ENCODER_SEND_RTMP_PACKET_FAILED;
}
```


至此，我们便完成了 AAC 音频的推送流程。此时可尝试使用 VLC 或其他支持 RTMP 协议的播放器连接到服务器测试正在直播的 AAC 音频流。

H264 码流的推送

前面提到过，向 RTMP 服务器发送 H264 码流，需要按照 FLV 格式进行封包，并且首先需要发送视频同步包 [AVC Sequence Header]。我们依旧先阅读 FLV 标准 Video Tags 一节：

VIDEODATA

Field	Type	Comment
FrameType	UB[4]	1: keyframe (for AVC, a seekable frame) 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
CodecID	UB[4]	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
VideoData	If CodecID == 2 H263VIDEOPACKET If CodecID == 3 SCREENVIDEOPACKET If CodecID == 4 VP6FLVVIDEOPACKET If CodecID == 5 VP6FLVALPHAVIDEOPACKET If CodecID == 6 SCREENV2VIDEOPACKET if CodecID == 7 AVCVIDEOPACKET	Video frame payload or UI8 (see note following table)

AVCVIDEOPACKET

Field	Type	Comment
AVCPacketType	UI8	0: AVC sequence header 1: AVC NALU 2: AVC end of sequence (lower level NALU sequence ender is not required or supported)
CompositionTime	SI24	if AVCPacketType == 1 Composition time offset else 0
Data	UI8[n]	if AVCPacketType == 0 AVCDecoderConfigurationRecord else if AVCPacketType == 1 One or more NALUs (can be individual slices per FLV packets; that is, full frames are not strictly required) else if AVCPacketType == 2 Empty

由于视频同步包前半部分比较简单易懂，仔细阅读上述标准便可明白如何操作，故 Bill 不另作图阐释。由上图可知，我们的视频同步包 FrameType == 1，CodecID == 7，VideoData == AVCVIDEOPACKET，继续展开 AVCVIDEOPACKET，我们可以得到 AVCPacketType == 0x00，CompositionTime == 0x000000，Data == AVCDecoderConfigurationRecord。

因此构造视频同步包的关键点便是构造 AVCDecoderConfigurationRecord。同样，我们援引其他博主的图片来阐释这个结构的细节：

长度	字段	说明
8 bit	configurationVersion	版本号，1
8 bit	AVCProfileIndication	sps[1]
8 bit	profile_compatibility	sps[2]
8 bit	AVCLevelIndication	sps[3]
6 bit	reserved	111111
2 bit	lengthSizeMinusOne	NALUnitLength的长度-1，一般为3
3 bit	reserved	111
5 bit	numOfSequenceParameterSets	sps个数，一般为1
	sequenceParameterSetNALUnits	(sps_size + sps)的数组
8 bit	numOfPictureParameterSets	pps个数，一般为1
	pictureParameterSetNALUnit	(pps_size +pps)的数组

其中需要额外计算的是 H264 码流的 Sps 以及 Pps，这两个关键数据可以在开始编码 H264 的时候提取出来并加以保存，在需要时直接使用即可。具体做法请读者自行 Google 或参见 参考博文[2]，在此不再赘述。

当我们得到本次 H264 码流的 Sps 以及 Pps 的相关信息后，我们便可以完成视频同步包的组装，示例代码如下：

```
int RtmpLiveEncoderImpl::sendAVCSequenceHeader(
    const RtmpLiveVideoMetadataTy &meta_data){
        RtmpLiveCodecByteTy body[1024] = { 0 };
        int i = 0;

        //
        // FrameType and CodecID
        //
        body[i++] = 0x17; // FrameType: 1: key frame | CodecID: 7: AVC

        //
        // VideoData
        //
        body[i++] = 0x00; // AVCPacketType: AVC sequence header
        i += 3; // Composition time. 0x000000.

        // AVCDecoderConfigurationRecord.
        body[i++] = 0x01; // configurationVersion
        body[i++] = meta_data.sps_data[1]; // AVCProfileIndication
        body[i++] = meta_data.sps_data[2]; // profile_compatibility
        body[i++] = meta_data.sps_data[3]; // AVCLevelIndication
        body[i++] = 0xFF; // lengthSizeMinusOne, always 0xFF
```

```
        // Numbers of Sps.
        body[i++] = 0xE1; // 0xE1 & 0x1F == 1.

        // Sps data length
        body[i++] = meta_data.sps_length >> 8;
        body[i++] = meta_data.sps_length & 0xff;

        // Sps data
        memcpy(&body[i], meta_data.sps_data, meta_data.sps_length);
        i += meta_data.sps_length;

        // Numbers of Pps.
        body[i++] = 0x01; //0x01 & 0xFF == 1.

        // Pps data length
        body[i++] = meta_data.pps_length >> 8;
        body[i++] = meta_data.pps_length & 0xff;

        // Pps data
        memcpy(&body[i], meta_data.pps_data, meta_data.pps_length);
        i += meta_data.pps_length;

        if(!sendRtmpPacket(body, i, RTMP_PACKET_TYPE_VIDEO, 0)){
            return RtmpLiveCodecErr::ENCODER_SEND_RTMP_PACKET_FAILED;
        }

        return RtmpLiveCodecErr::NO_ERR;
    }
```


至此，视频同步包便构造完毕并推送给 RTMP 服务器。接下来只需要将普通 H264 码流稍加封装便可实现 H264 直播，下面我们来看一下普通视频包的组装过程。

回顾 FLV 标准 的 Video Tags 一节，我们可以得到 H264 普通数据包的封包信息，FrameType == （H264 I 帧 ? 1 : 2），CodecID == 7，VideoData == AVCVIDEOPACKET，继续展开，我们可以得到 AVCPacketType == 0x01，CompositionTime 此处仍然设置为 0x000000，具体原因 TODO(billhoo)，Data == H264 NALU Size + NALU Raw Data。

构造视频数据包的示例代码如下：

```
RtmpLiveCodecByteTy *body = new(std::nothrow) RtmpLiveCodecByteTy[nalu_size + 9];
if(body == NULL){
    return RtmpLiveCodecErr::MEMORY_BAD_ALLOC;
}

int i = 0;

//
// FrameType and CodecID
//
if(is_key_frame) {
    body[i++] = 0x17; // 1: I-frame  7: AVC
} else {
    body[i++] = 0x27; // 2: P-frame  7: AVC
}

//
// VideoData
//
body[i++] = 0x01; // AVCPacketType: AVC NALU
i += 3; // Composition time. 0x000000.
```

```
// NALU size
body[i++] = nalu_size >> 24;
body[i++] = nalu_size >> 16;
body[i++] = nalu_size >> 8;
body[i++] = nalu_size & 0xff;

// NALU data
memcpy(&body[i], data, nalu_size);

if(!sendRtmpPacket(body, i + nalu_size, RTMP_PACKET_TYPE_VIDEO, timestamp)){
    delete[] body;
    return RtmpLiveCodecErr::ENCODER_SEND_RTMP_PACKET_FAILED;
}
```


至此 H264 码流的整个推送流程便已完成，我们可以使用 VLC 或其他支持 RTMP 协议的播放器进行测试。

关于 AAC 音频帧及 H264 码流的时间戳

通过前文的步骤我们已经能够将 AAC 音频帧以及 H264 码流正常推送到 RTMP 直播服务器，并能够使用相关播放器进行播放。但播放的效果如何还取决于时间戳的设定。

在网络良好的情况下，自己最开始使用的音频流时间戳为 AAC 编码器刚输出一帧的时间，视频流时间戳为 H264 编码器刚编码出来一帧的时间，VLC 播放端就频繁报异常，要么是重新缓冲，要么直接没声音或花屏。在排除了推送步骤实现有误的问题后，Bill 发现问题出在时间戳上。

之后有网友说直播流的时间戳不论音频还是视频，在整体时间线上应当呈现递增趋势。由于 Bill 最开始的时间戳计算方法是按照音视频分开计算，而音频时戳和视频时戳并不是在一条时间线上，这就有可能出现音频时戳在某一个时间点比对应的视频时戳小， 在某一个时间点又跳变到比对应的视频时戳大，导致播放端无法对齐。

目前采用的时间戳为底层发送 RTMP 包的时间，不区分音频流还是视频流，统一使用即将发送 RTMP 包的系统时间作为该包的时间戳。目前局域网测试播放效果良好，音视频同步且流畅。

参考博文

[1][C++实现RTMP协议发送 H.264 编码及 AAC 编码的音视频]

[2][使用 libRtmp 进行 H264 与 AAC 直播]

[3][RTMP直播到FMS中的AAC音频直播]

文章评论

[1楼] leleniu 回复

2014-10-09 16:14:22

音视频时间戳怎么做的，求分享，谢谢。

[2楼]楼主 Bill_Hoo 回复

2014-10-28 11:29:09

回复 leleniu:[1楼]

您好，音视频时间戳均取自发送 RTMP 包的系统时间。

[3楼]楼主 Bill_Hoo 回复

2015-09-25 16:41:25

回复 leleniu:[1楼]

更新一下，以前使用的是发送时间，但后来发现可能因为内部线程的调度导致发送的时间戳产生些许误差，因此最后音视频时间戳均采用采集时间，并使用 RTMP 32位扩展时戳进行发送

发表评论

昵　称: 登录 快速注册

验证码: 请点击后输入验证码 博客过2级，无需填写验证码

内　容:

意见反馈