rlandj

在吵吵闹闹中发发脾气，在油盐酱醋中惹惹鸡毛蒜皮，吃吃饭喝喝酒逛逛街旅旅行，没什么值得拼死奋斗努力巴结谁，甭惯着别人怠慢了自己，买不起的就不买，忘不掉的就不忘，活着图自己开心，累了让自己舒服，虚伪滚蛋，纠结去死.....想太多太累，做一个简单的人！

with ffmpeg to encode video for live streaming and for recording to files for on-demand playback

We've been doing some experimentation with ffmpeg to encode video for live streaming and for recording to files for on-demand playback. While I've been impressed by the capabilities of ffmpeg, I've found that its command-line processing is quite daunting. Here I provide a set of command-line options along with commentary on what is going on.

One of the most frustrating things I've found in trying to learn ffmpeg is that many examples online are quite out-of-date. It seems that ffmpeg has changed its command-line options fairly frequently. So many of the examples you'll find won't work.

The example that follows uses a snapshot of the ffmpeg subversion code from September 10, 2009.

Ready?

Are you sure?

Here is the command line:

```
ffmpeg -f rawvideo -pix_fmt yuv422p \
    -s 720x486 -r 29.97 \
    -i /tmp/vpipe \
    -ar 48000 -f s16le -ac 2 -i /tmp/apipe \
    -vol 4096 \
    -acodec libfaac -ac 2 -ab 192k -ar 44100 \
    -async 1 \
    -vcodec libx264 -vpre default -threads 0 \
    -croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
    -r 20 -g 45 -s 432x324 -b 1024k -bt 100k \
    -deinterlace \
    -y \
    '/tmp/encoding-0001.mp4' \
    -an \
    -vcodec libx264 -vpre default -threads 0 \
    -croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
    -r 20 -g 45 -s 432x324 -b 1024k -bt 100k \
    -deinterlace \
    -vglobal 1 \
    -f rtp rtp://127.0.0.1:9012 \
    -vn \
    -vol 4096 \
    -acodec libfaac -ac 2 -ab 192k -ar 44100 \
    -async 1 \
```

```
    -flags +global_header \
    -f rtp rtp://127.0.0.1:9014 \
    -newaudio \
    -an \
    -vcodec libx264 -vpre default -threads 0 \
    -croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
    -r 15 -g 45 -s 432x324 -b 768k -bt 50k \
    -deinterlace \
    -vglobal 1 \
    -f rtp rtp://127.0.0.1:9008 \
    -newvideo \
    -vn \
    -vol 4096 \
    -acodec libfaac -ac 2 -ab 128k -ar 44100 \
    -async 1 \
    -flags +global_header \
    -f rtp rtp://127.0.0.1:9010 \
    -newaudio
```

Don't be intimidated. We'll break this thing down line-by-line.

Specifying the input

```
ffmpeg -f rawvideo -pix_fmt yuv422p \
    -s 720x486 -r 29.97 \
    -i /tmp/vpipe \
```

Everything up to the -i option describes what is contained in the input file /tmp/vpipe. In this case, our input video is raw (uncompressed) frame data in the YUV 4:2:2 planar format, 720 pixels wide by 486 pixels high, with a frame rate of 29.97 frames per second. Note that these options must precede the -i option. If any of these options came after the -i, ffmpeg would think that they belonged to the next input file specified.

```
    -ar 48000 -f s16le -ac 2 -i /tmp/apipe \
```

Now we're telling ffmpeg what the audio looks like in the input file /tmp/apipe. We have a sample rate of 48000 samples per second; each sample is signed 16-bit, little endian, and there are 2 audio channels.

Specifying the output

The next set of options describes the output format, for both audio and video. I do not believe that the order within these options is critical, but I like to group them logically so that all the audio options are together and all the video options are together.

```
    -vol 4096 \
    -acodec libfaac -ac 2 -ab 192k -ar 44100 \
    -async 1 \
```

The -vol option indicates that we're going to adjust the audio levels. The baseline value is 256. I have not seen good documentation on the valid values here, but I've used values like 512, 1024, 2048, and 4096 to bump up the volume. Picking the right value here requires some experimentation and depends heavily on the levels in your original source.

We're going to output AAC audio using the libfaac encoder. ffmpeg has a built-in AAC encoder, but it doesn't seem to be as robust as that provided by libfaac. If you want AAC, and your version of ffmpeg was linked against libfaac, I would recommend that you use it.

The AAC will be 2 channels, 192kbps, with a sample rate of 44100 Hz.

Finally, the -async option indicates that ffmpeg should use audio/video sync method 1, which adjusts the start of the video and audio tracks, but does not do any time stretching over the course of the tracks.

```
    -vcodec libx264 -vpre default -threads 0 \
    -croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
    -r 20 -g 45 -s 432x324 -b 1024k -bt 100k \
    -deinterlace \
```

Here we are specifying our video output options. We're going to use H.264 compression, as provided by the libx264 library. The -vpre option means that we will use the default quality preset (note that when using libx264 to encode, you can specify two -vpre options. The first is quality, and the second is the profile to use (main, baseline, etc., with "main" being the default). I have not seen this documented very well.

The -threads 0 option instructs ffmpeg to use the optimal number of threads when encoding.

We are going to crop our video a few pixels around the border, as we were getting some noise around the edges of our source video.

The -r option specifies that our output will be 20 frames per second. The -g option is the "group of pictures" (GOP) size, which is the number of frames between keyframes. With a smaller number, your output will have more keyframes, which means that streaming clients will be able to recover more quickly if they drop packets for some

reason. It also will have a detrimental effect on file size.

The -s option specifies our frame size, the -b option specifies the desired bitrate, and the -bt option is the bitrate tolerance. ffmpeg will try to keep the video close to the desired bitrate, and the tolerance tells it how much leeway it has above and below the target bitrate.

Finally, we are deinterlacing the video, as the source was NTSC interlaced video. Without deinterlacing, you can see some very unpleasant "comb" artifacts in your digitized video.

```
-y \
'/tmp/encoding-0001.mp4' \
```

Here we specify the output file. The -y option instructs ffmpeg to overwrite the file without asking for confirmation. ffmpeg will infer the file format from the filename extension. Here it will be writing to an MPEG4 file.

Adding another output

Now we are going to add some specs for RTP output. We will send this RTP stream over the network to a Wowza server, which can convert the RTP to RTMP for playback in Flash clients.

Unlike when we wrote to an MPEG4 file, RTP requires that we break the audio and video into two separate streams.

```
-an \
-vcodec libx264 -vpre default -threads 0 \
-croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
-r 20 -g 45 -s 432x324 -b 1024k -bt 100k \
-deinterlace \
-vglobal 1 \
-f rtp rtp://127.0.0.1:9012 \
```

You'll notice that most of these specs are the same as what we specified for our MPEG4 output. But there are a few differences. Let's focus on those. The -an option tells ffmpeg to remove the audio stream from this output. The -vglobal 1 option instructs ffmpeg to use out-of-band global headers in the video stream (as opposed to in-band). This may help some players interpret the video stream.

Finally, we specify the output format as "rtp" with the -f option, and instead of a filename, we have a URL that indicates where ffmpeg should send the RTP packets.

Next we specify the audio output:

```
-vn \
-vol 4096 \
-acodec libfaac -ac 2 -ab 192k -ar 44100 \
-async 1 \
-flags +global_header \
-f rtp rtp://127.0.0.1:9014 \
-newaudio \
```

Again, this is very similar to the options we provided for our MPEG4 output, with some notable differences. First, we use the -vn option to indicate that this output does not contain video. The -flags +global_header is used to force ffmpeg to spit out some important audio specifications in the SDP it generates (you use an SDP file on the Wowza server to connect the RTMP stream to the RTP stream; Wowza needs to know all about the audio and video to interpret it properly).

Again, we specify the rtp format with the -f option, and we provide a URL. Note that the port number is different. It is customary for RTP streams to use two ports with an open port between them. The ports just after each of the RTP ports will be used for RTCP ports (in our example, ports 9013 and 9015 wil be used), where the receiver communicates back to the sender.

The last option, -newaudio, restores the audio stream that got killed off by the -an option earlier. Note that -newaudio is a special option; it only modifies the output which immediately precedes it. This is one of those cases where order of options is critical.

With these options, we are now writing to an MPEG4 file and streaming RTP simultaneously. But wait, there's more...

Adding another RTP stream

Our first RTP stream uses about 1200 Kbps with audio and video combined. Let's create an option for our bandwidth-deprived visitors to use.

We can tack on another pair of outputs, one for video and one for audio:

```
-an \
-vcodec libx264 -vpre default -threads 0 \
-croptop 6 -cropbottom 6 -cropleft 9 -cropright 9 \
-r 15 -g 45 -s 432x324 -b 256k -bt 50k \
-deinterlace \
-vglobal 1 \
```

```
-f rtp rtp://127.0.0.1:9008 \
-newvideo \
-vn \
-vol 4096 \
-acodec libfaac -ac 1 -ab 64k -ar 44100 \
-async 1 \
-flags +global_header \
-f rtp rtp://127.0.0.1:9010 \
-newaudio
```

There are a few differences between our second set of RTP outputs and the first:

- the video framerate is 15 (-r 15)
- the video bitrate is 256kbps and the tolerance is 50kbps (-b 256k -bt 50k)
- the audio is single channel (-ac 1)
- the audio is 64kbps (-ab 64k)
- the ports are different in the RTP URLs

You'll also notice that there is a -newvideo option after the video RTP URL. That's because our previous output was audio-only, due to the -vn option. Using the -newvideo option restores the video stream to this output. Without it, you'd have no audio (because of the -an), and no video (because of the -vn in the previous output).

So now we're writing to an MPEG4 file and streaming at two different bitrates simultaneously! Pretty sweet.

I won't get into the specifics here of how you get the SDP file from ffmpeg, put it on the Wowza server, and connect to the Wowza server with a flash-based player. There are lots of docs and notes in the Wowza forums on how to do that.

posted @ 2014-09-11 11:35 rlandj 阅读(1157) 评论(0) 编辑 收藏 举报