直播技术系列之二 播放质量控制

当成功拉取音视频流并解码之后，需要直接渲染视频、播放音频，这是需要有一个算法（或者称为策略）更为贴切来控制这一过程，观众看到的才是流畅、同步的音视频。

之所以记录这一块的知识，主要是这段时间基本上都被这个问题折腾。这个播放算法和业务其实是贴的很紧的，所以我现今位置没有找到介绍说如何流畅、同步播放的算法。在此我试图将我看到过的、实现过的一些算法描述下来。如果有人看了之后交流下还有哪些更好的实现就更好了。

既然是与业务相关紧密的，那可以先看看这种最简单的场景。

单路播放

从一个rtmp地址拉取到流，然后解压出图片、PCM数据，再播放出来。大体流程是这样的。

Rtmp流

解码器

渲染、播放

上图其实是漏掉了我们这里讨论的主题，播放质量控制。因为收到的流是网络流，所以接收每帧音视频数据之间的时间间隔是浮动的，甚至会浮动比较大（即使不考虑网络抖动）。因此在实际的应用中都会加入一个播放质量控制的处理过程。上图则变成下面这个样子。

Rtmp流

播放质量控制

渲染、播放

解码器

一般的播放质量控制包含两个模块：数据缓冲队列、定时播放线程。

播放质量控制的目的就是按照帧率定时提供数据给渲染、播放模块，这个定时越准确，则音视频播放的效果就越好。

在播放质量控制模块中，最核心的概念当属时间戳，这个时间戳一般从两个地方获得：

将视频、音频解码之后，可以获得解码数据的PTS。当打开媒体文件如FLV、MP3、MP4之类的时候，我一般使用PTS作为显示的时间戳。

流媒体网络协议（RTMP、RTSP）在获取到一帧数据时会有对应的时间戳。当从这些网络协议中拉流时，我一般使用这个时间戳作为显示的时间戳。因为在这种情况下，解码出来的PTS并不一定正确。

数据缓冲队列用于缓冲解码器解码出来的数据。有了这个缓冲队列才能应对网络的抖动以及解码或者其他处理导致的抖动。

有同事因为这个缓冲队列和我说可以将播放质量控制放在解码器前面，这样能节省很多很多的内存（解码前数据与解码后数据的差）。我认真想了下为什么要把播放质量控制放在解码器之后而不是之前。首先，几年前我刚刚接触音视频播放这一块时看到前辈写的代码就是放在解码器之后的，这种方式用了几年，没发现有大问题。然后，赶紧算下缓冲一路视频数据（音频数据较小）需要多少内存。以1080P、25fps为例，解码后为RGB（故意算高的），缓冲50帧（2秒）数据需要300MB左右的内存，是有点高，但我还是能够接受（这可能是因为以前都在服务器上开发，不太担心内存，而且根据我的经验，一般这个值是上限，实际直播中不会缓冲2秒数据再播放）。而如果播放质量控制模块放在解码器之前，则解码器产生的抖动谁来处理，在解码H264中有B帧或者编码使用的是变码率时这个问题会比较突出。而且在解码之后，存在很大的可能性需要加上其他音视频处理，这些处理中有些是很耗时间的。基于以上原因，我一般都将播放质量控制放在解码器之后。

数据缓冲队列一般满足以下几点需求：

1. 支持FIFO的数据队列
2. 支持设置队列个数上限
3. 支持多个用户从队列中拉取数据，而且都能拉取到完整的数据流
4. 当无人拉取数据时，再往队列中插入数据将导致老的数据出队列

如果没有第3点，则当有多个消费则同时从播放控制模块中拉取数据时，可能导致数据流被拆分给不同的消费者。如有序号为1、2、3、4、5、6的帧，可能用户一拉取到1、2、5，而用户二拉取到3、4、6。

如果不实现第4点，当拉取数据的用户后启动时，可能拉取到的前面一段数据就是过时的数据。

接下来看下定时播放线程。一般在播放质量控制中都会使用一个线程作为缓冲队列的消费者。不断从缓冲队列中拉取数据，并在应该渲染、播放的时候回调给真实的消费者。这里假设的是播放质量控制模块使用的是推送音视频数据的模式。实践经验是对于简单的应用场景，推送的模式会简单很多。

整个播放控制模块如图：

定时线程

Get

Frame

渲染

编码发送等

Push

播放控制模块

|  |
| --- |
| Void doPlayCtrlWork()  {  //这里使用的是计算当前帧时间戳与第一帧时间戳差值  //再与当前系统时间与第一帧显示时系统时间的差值比较的方法  //之所以使用第一帧与当前帧的差值  //而不是根据前后两帧的时间戳差值来确定需要Sleep多久后显示下一帧  //是因为系统Sleep定时器的不准确，可能导致偶尔出现视频的卡顿以及数据堆积  //特别是在机器的CPU使用率较高时，这种现象更加明显  LONGLONG first\_play\_time = -1;  Int delay = DELAY\_TIME;  Frame\* pCur = NULL;  Int first\_frame\_time = -1;  While(isRunning)  {  If(pCur==NULL)  {  pCur = frameList.get();  }  If(pCur!=NULL)  {  Int frame\_time = pCur->getTimestamp();  InterlockedCompareAndExchange(&first\_frame\_time, -1, frame\_time);  LONGLONG now = now();  InterlockedCompareAndExchange(&first\_play\_time, -1, now);  If(now-first\_play\_time-delay >= frame\_time - first\_frame\_time)  {  Callbackdata(pCur);  pCur = NULL;  }  }  Sleep(N);  }  } |

表格 1 流畅播放的伪代码

在定时线程中，实现的代码大概是这样的。

上面的伪代码只处理了视频或者音频的播放流畅性问题，没有处理音视频之间的同步问题。而且在碰到网络抖动导致数据帧延迟到达的时间超过DELAY\_TIME时，将无法处理。

当处理的媒体流中同时存在音频流与视频流，则此时必须考虑音视频的同步问题。

既然如此，我们下面加上音视频同步来考虑如何实现定时播放线程。

在音画同步中我们首先考虑较简单也较普遍的情况，音频数据时间戳与视频数据时间戳使用相同的时间基准（或者称为起点时间）。这种情况是最为普遍与自然的情况，比如当一个摄像头拍摄画面并同时采集声音时，当前时间就是音频与视频公用的时间基准，只要产生音频数据的线程与产生视频数据的线程都能够准确获取当前时间，则播放端很容易根据这个事实做到同步播放音视频。

Video

Audio

上图描绘了音频与视频采样时间戳的关系，因此不管视频、音频的采样点是否一致，只要才要的时间间隔不大，而且可以准确获取采样时的时间戳，播放端就可以根据这个时间戳做到同步播放。

基于音视频时间戳使用相同的时间基准这一假设，播放端需要做到音画同步将变得十分简单。只需对表格1中的代码进行简单的修改即可实现。音频的播放控制线程与视频的播放控制线程同时试图修改first\_frame\_time变量，而最终将是数据更早到达的线程去设置first\_frame\_time变量。接下来的播放控制，不需要做任何修改，此时音画就是同步的了。

在更复杂的情况中，音频的采集与视频的采集使用不同的设备，使用不同的时间基准，此时要做到音画同步播放将不是那么容易。在这种情况下，应对尽量在数据没有进过缓冲队列之前对音视频进行一次同步处理，并打上新的时间戳。因为如果音频或者视频数据进入缓冲队列之后，数据的即时性将变得更差，则音频与视频的时间戳将差异更大。摄像机使用的其实就是这种做法，从硬件采集的数据直接使用相同时间基准的时间戳，不进入任何的缓冲队列，这样的时间戳是即时的，因此播放端能够重新同步播放。

以上就是我所知道关于简单的单路视频的播放质量控制的所有内容了。