[Android audio and video development and upgrade: audio and video hard decoding] 3. Audio and video playback: audio and video synchronization

Tutorial code: [Github Portal]

Table of contents

In this article you can learn

> The previous article mainly talked about the process of Android MediaCodec to realize hard decoding of audio and video, and built the basic decoding framework. This article will explain the specific audio and video rendering, including MediaCodec initialization, Surface initialization, AudioTrack initialization, audio and video data stream separation and extraction, etc., as well as the very important audio and video synchronization.

In the decoding process framework base class defined in the previous article, several virtual functions are reserved for subclasses to initialize their own things. In this article, let's take a look at how to implement it.

1. Audio and video data stream separation extractor

In the last article, I mentioned the audio and video data separation extractor many times. Before implementing the audio and video decoder subclass, I implemented this first.

Encapsulate Android native extractor

As mentioned before, Android natively comes with a MediaExtractor for audio and video data separation and extraction. Next, based on this, make a tool class MMExtractor that supports audio and video extraction:

```
class MMExtractor(path: String?) {

    /**音视频分离器*/
    private var mExtractor: MediaExtractor? = null

    /**音频通道索引*/
    private var mAudioTrack = -1

    /**视频通道索引*/
    private var mVideoTrack = -1

    /**当前帧时间戳*/
    private var mCurSampleTime: Long = 0

    /**开始解码时间点*/
    private var mStartPos: Long = 0

    init {
        //【1，初始化】
        mExtractor = MediaExtractor()
        mExtractor?.setDataSource(path)
    }

    /**
     * 获取视频格式参数
     */
    fun getVideoFormat(): MediaFormat? {
        //【2.1，获取视频多媒体格式】
        for (i in 0 until mExtractor!!.trackCount) {
            val mediaFormat = mExtractor!!.getTrackFormat(i)
            val mime = mediaFormat.getString(MediaFormat.KEY_MIME)
            if (mime.startsWith("video/")) {
                mVideoTrack = i
                break
            }
        }
        return if (mVideoTrack >= 0)
            mExtractor!!.getTrackFormat(mVideoTrack)
        else null
    }

    /**
     * 获取音频格式参数
     */
    fun getAudioFormat(): MediaFormat? {
        //【2.2，获取音频频多媒体格式】
        for (i in 0 until mExtractor!!.trackCount) {
            val mediaFormat = mExtractor!!.getTrackFormat(i)
            val mime = mediaFormat.getString(MediaFormat.KEY_MIME)
            if (mime.startsWith("audio/")) {
                mAudioTrack = i
                break
            }
        }
        return if (mAudioTrack >= 0) {
            mExtractor!!.getTrackFormat(mAudioTrack)
```

```
    } else null
}

/**
 * 读取视频数据
 */
fun readBuffer(byteBuffer: ByteBuffer): Int {
    //【3，提取数据】
    byteBuffer.clear()
    selectSourceTrack()
    var readSampleCount = mExtractor!!.readSampleData(byteBuffer, 0)
    if (readSampleCount < 0) {
        return -1
    }
    mCurSampleTime = mExtractor!!.sampleTime
    mExtractor!!.advance()
    return readSampleCount
}

/**
 * 选择通道
 */
private fun selectSourceTrack() {
    if (mVideoTrack >= 0) {
        mExtractor!!.selectTrack(mVideoTrack)
    } else if (mAudioTrack >= 0) {
        mExtractor!!.selectTrack(mAudioTrack)
    }
}

/**
 * Seek到指定位置，并返回实际帧的时间戳
 */
fun seek(pos: Long): Long {
    mExtractor!!.seekTo(pos, MediaExtractor.SEEK_TO_PREVIOUS_SYNC)
    return mExtractor!!.sampleTime
}

/**
 * 停止读取数据
 */
fun stop() {
    //【4，释放提取器】
    mExtractor?.release()
    mExtractor = null
}

fun getVideoTrack(): Int {
    return mVideoTrack
}

fun getAudioTrack(): Int {
    return mAudioTrack
}

fun setStartPos(pos: Long) {
```

```
        mStartPos = pos
    }

    /**
     * 获取当前帧时间
     */
    fun getCurrentTimestamp(): Long {
        return mCurSampleTime
    }
}
```

It's relatively simple, just paste the code directly.

There are 5 key parts. Let's briefly explain:

[1, initialization]

Very simple, two lines of code: create a new one, and then set the audio and video file path

```
mExtractor = MediaExtractor()
mExtractor?.setDataSource(path)
```

[2.1/2.2, get audio and video multimedia formats]

Audio and video are the same:

1) Traverse all the channels in the video file, generally two channels of audio and video;

2) Then obtain the encoding format of the corresponding channel, and determine whether it contains the encoding format starting with "video/" or "audio/";

3) Finally, through the obtained index, return the corresponding audio and video multimedia format information.

[3, extract data]

Focus on how to extract data:

1) The parameter in readBuffer(byteBuffer: ByteBuffer) is the buffer passed in by the decoder and used to store the data to be decoded.

2) In the selectSourceTrack() method, according to the currently selected channel (only one audio/video channel is selected at the same time), call mExtractor!!.selectTrack(mAudioTrack) to switch the channel correctly.

3) Then read the data:

```
var readSampleCount = mExtractor!!.readSampleData(byteBuffer, 0)
```

At this time, the size of the read audio and video data stream will be returned, and if it is less than 0, the data has been read.

4) Enter the next frame: first record the timestamp of the current frame, and then call advance to enter the next frame, then the read pointer will automatically move to the beginning of the next frame.

```
//记录当前帧的时间戳
mCurSampleTime = mExtractor!!.sampleTime
//进入下一帧
mExtractor!!.advance()
```

[4, release the extractor]

When the client exits decoding, it needs to call stop whether to extract related resources.

> Description: The seek(pos: Long) method is mainly used for skipping and quickly positioning the data to the specified playback position. However, in the video, except for the I frame, the PB frame needs to rely on other frames for decoding, so, Usually only seek to the I frame, but the I frame usually has a certain error with the specified playback position, so it is necessary to specify which key frame the seek is close to. There are the following three types:
> SEEK_TO_PREVIOUS_SYNC: the previous key frame of the skip position
> SEEK_TO_NEXT_SYNC: skip playback Position's next keyframe
> SEEK_TO_CLOSEST_SYNC: The closest keyframe to the skip position

At this point, you can understand why when we are watching videos, after dragging the progress bar and releasing it, the video usually moves forward at the position you released.

Encapsulates audio and video extractors

The tools encapsulated above can support audio and video data extraction. Next, we will use this tool to extract audio and video data respectively.

Let's review the extractor model defined in the previous article:

```
interface IExtractor {

    fun getFormat(): MediaFormat?

    /**
     * 读取音视频数据
     */
    fun readBuffer(byteBuffer: ByteBuffer): Int

    /**
     * 获取当前帧时间
     */
    fun getCurrentTimestamp(): Long

    /**
     * Seek到指定位置，并返回实际帧的时间戳
     */
    fun seek(pos: Long): Long

    fun setStartPos(pos: Long)

    /**
     * 停止读取数据
     */
    fun stop()
}
```

With the tools encapsulated above, everything becomes very simple, just do a proxy transfer.

> video extractor

```
class VideoExtractor(path: String): IExtractor {

    private val mMediaExtractor = MMExtractor(path)

    override fun getFormat(): MediaFormat? {
        return mMediaExtractor.getVideoFormat()
    }

    override fun readBuffer(byteBuffer: ByteBuffer): Int {
        return mMediaExtractor.readBuffer(byteBuffer)
    }

    override fun getCurrentTimestamp(): Long {
        return mMediaExtractor.getCurrentTimestamp()
    }

    override fun seek(pos: Long): Long {
        return mMediaExtractor.seek(pos)
    }

    override fun setStartPos(pos: Long) {
        return mMediaExtractor.setStartPos(pos)
    }

    override fun stop() {
        mMediaExtractor.stop()
    }
}
```

audio extractor

```
class AudioExtractor(path: String): IExtractor {

    private val mMediaExtractor = MMExtractor(path)

    override fun getFormat(): MediaFormat? {
        return mMediaExtractor.getAudioFormat()
    }

    override fun readBuffer(byteBuffer: ByteBuffer): Int {
        return mMediaExtractor.readBuffer(byteBuffer)
    }

    override fun getCurrentTimestamp(): Long {
        return mMediaExtractor.getCurrentTimestamp()
    }

    override fun seek(pos: Long): Long {
        return mMediaExtractor.seek(pos)
    }

    override fun setStartPos(pos: Long) {
        return mMediaExtractor.setStartPos(pos)
    }

    override fun stop() {
        mMediaExtractor.stop()
    }
}
```

2. Video playback

Let's first define a video decoder subclass that inherits BaseDecoder

```
class VideoDecoder(path: String,
                   sfv: SurfaceView?,
                   surface: Surface?): BaseDecoder(path) {
    private val TAG = "VideoDecoder"

    private val mSurfaceView = sfv
    private var mSurface = surface

    override fun check(): Boolean {
        if (mSurfaceView == null && mSurface == null) {
            Log.w(TAG, "SurfaceView和Surface都为空，至少需要一个不为空")
            mStateListener?.decoderError(this, "显示器为空")
            return false
        }
        return true
    }

    override fun initExtractor(path: String): IExtractor {
        return VideoExtractor(path)
    }

    override fun initSpecParams(format: MediaFormat) {
    }

    override fun configCodec(codec: MediaCodec, format: MediaFormat): Boolean {
        if (mSurface != null) {
            codec.configure(format, mSurface , null, 0)
            notifyDecode()
        } else {
            mSurfaceView?.holder?.addCallback(object : SurfaceHolder.Callback2 {
                override fun surfaceRedrawNeeded(holder: SurfaceHolder) {
                }

                override fun surfaceChanged(holder: SurfaceHolder, format: Int,
width: Int, height: Int) {
                }

                override fun surfaceDestroyed(holder: SurfaceHolder) {
                }

                override fun surfaceCreated(holder: SurfaceHolder) {
                    mSurface = holder.surface
                    configCodec(codec, format)
                }
            })

            return false
        }
        return true
    }

    override fun initRender(): Boolean {
        return true
    }

    override fun render(outputBuffers: ByteBuffer,
```

```
                        bufferInfo: MediaCodec.BufferInfo) {
    }

    override fun doneDecode() {
    }
}
```

In the last article, the decoding process framework is defined, and the subclass definition is very simple and clear. Just follow the steps and fill in the virtual functions reserved in the base class.

Check parameters

As you can see, video decoding supports two types of rendering surfaces, one is SurfaceView and the other is Surface. When in fact, the Surface is passed to MediaCodec in the end

1. SurfaceView should be a familiar View, and the most commonly used one is the display of MediaPlayer. Of course, you can also draw pictures, animations, etc.
2. Surface should not be very commonly used. In order to support the subsequent use of OpenGL to render video, it is supported in advance.

Generate data extractors

```
override fun initExtractor(path: String): IExtractor {
    return VideoExtractor(path)
}
```

Configure the decoder

The configuration of the decoder requires only one line of code:

```
codec.configure(format, mSurface , null, 0)
```

I don't know if you found out in the last article, in the method initCodec() of BaseDecoder to initialize the decoder, after calling the configCodec method, it will enter the waitDecode method and suspend the thread.

```
abstract class BaseDecoder(private val mFilePath: String): IDecoder {
    //省略其他
    ......

    private fun initCodec(): Boolean {
        try {
            val type = mExtractor!!.getFormat()!!.getString(MediaFormat.KEY_MIME)
            mCodec = MediaCodec.createDecoderByType(type)
            if (!configCodec(mCodec!!, mExtractor!!.getFormat()!!)) {
                waitDecode()
            }
            mCodec!!.start()

            mInputBuffers = mCodec?.inputBuffers
            mOutputBuffers = mCodec?.outputBuffers
        } catch (e: Exception) {
            return false
        }
        return true
    }
}
```

Initialize Surface

It is because of a problem that the creation of SurfaceView has a time process, and it is not available immediately. It needs to monitor its status through CallBack.

After the surface is initialized, configure the MediaCodec.

```
override fun surfaceCreated(holder: SurfaceHolder) {
    mSurface = holder.surface
    configCodec(codec, format)
}
```

If you use OpenGL to directly pass the surface in, you can directly configure the MediaCodec.

render

As mentioned above, the rendering of the video does not require the client to render manually. It only needs to provide the drawing surface surface, call releaseOutputBuffer, and set the two parameters to true. So, there is no need to do anything here.

```
mCodec!!.releaseOutputBuffer(index, true)
```

3. Audio playback

With the basics of the video player above, the audio player can be done in minutes.

```
class AudioDecoder(path: String): BaseDecoder(path) {
    /**采样率*/
    private var mSampleRate = -1

    /**声音通道数量*/
    private var mChannels = 1

    /**PCM采样位数*/
    private var mPCMEncodeBit = AudioFormat.ENCODING_PCM_16BIT

    /**音频播放器*/
    private var mAudioTrack: AudioTrack? = null

    /**音频数据缓存*/
    private var mAudioOutTempBuf: ShortArray? = null

    override fun check(): Boolean {
        return true
    }

    override fun initExtractor(path: String): IExtractor {
        return AudioExtractor(path)
    }

    override fun initSpecParams(format: MediaFormat) {
        try {
            mChannels = format.getInteger(MediaFormat.KEY_CHANNEL_COUNT)
            mSampleRate = format.getInteger(MediaFormat.KEY_SAMPLE_RATE)

            mPCMEncodeBit = if (format.containsKey(MediaFormat.KEY_PCM_ENCODING))
{
                format.getInteger(MediaFormat.KEY_PCM_ENCODING)
            } else {
                //如果没有这个参数，默认为16位采样
                AudioFormat.ENCODING_PCM_16BIT
            }
        } catch (e: Exception) {
        }
    }

    override fun configCodec(codec: MediaCodec, format: MediaFormat): Boolean {
        codec.configure(format, null , null, 0)
        return true
    }

    override fun initRender(): Boolean {
        val channel = if (mChannels == 1) {
            //单声道
            AudioFormat.CHANNEL_OUT_MONO
        } else {
            //双声道
            AudioFormat.CHANNEL_OUT_STEREO
        }

        //获取最小缓冲区
        val minBufferSize = AudioTrack.getMinBufferSize(mSampleRate, channel,
```

```
mPCMEncodeBit)

        mAudioOutTempBuf = ShortArray(minBufferSize/2)

        mAudioTrack = AudioTrack(
            AudioManager.STREAM_MUSIC,//播放类型：音乐
            mSampleRate, //采样率
            channel, //通道
            mPCMEncodeBit, //采样位数
            minBufferSize, //缓冲区大小
            AudioTrack.MODE_STREAM) //播放模式：数据流动态写入，另一种是一次性写入

        mAudioTrack!!.play()
        return true
    }

    override fun render(outputBuffer: ByteBuffer,
                        bufferInfo: MediaCodec.BufferInfo) {
        if (mAudioOutTempBuf!!.size < bufferInfo.size / 2) {
            mAudioOutTempBuf = ShortArray(bufferInfo.size / 2)
        }
        outputBuffer.position(0)
        outputBuffer.asShortBuffer().get(mAudioOutTempBuf, 0, bufferInfo.size/2)
        mAudioTrack!!.write(mAudioOutTempBuf!!, 0, bufferInfo.size / 2)
    }

    override fun doneDecode() {
        mAudioTrack?.stop()
        mAudioTrack?.release()
    }
}
```

The initialization process is the same as the video, and there are three differences:

1. Initialize the decoder

Audio does not need surface, pass null directly

```
codec.configure(format, null , null, 0)
```

2. The parameters obtained are different

Audio playback needs to obtain the sampling rate, the number of channels, the number of sampling bits, etc.

3. An audio renderer needs to be initialized: AudioTrack

Since the decoded data is PCM data, it can be played directly using AudioTrack. Initialize it in initRender()

.

- Configure mono and binaural based on the number of channels
- Calculate and obtain the minimum buffer according to the sampling rate, the number of channels, and the number of sampling bits

```
AudioTrack.getMinBufferSize(mSampleRate, channel, mPCMEncodeBit)
```

Create AudioTrack and start

```
mAudioTrack = AudioTrack(
            AudioManager.STREAM_MUSIC,//播放类型：音乐
            mSampleRate, //采样率
            channel, //通道
            mPCMEncodeBit, //采样位数
            minBufferSize, //缓冲区大小
            AudioTrack.MODE_STREAM) //播放模式：数据流动态写入，另一种是一次性写入

mAudioTrack!!.play()
```

4. Manually render audio data for playback

The last step is to write the decoded data into AudioTrack for playback.

One thing to note is that the decoded data needs to be converted from ByteBuffer to ShortBuffer, and the length of the Short data type is halved.

Fourth, call and play

Above, the audio and video playback process is basically realized. If there is no accident, the above audio and video decoder can be called on the page to realize the playback.

Just take a look at the page and related calls.

main_activity.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">
    <SurfaceView android:id="@+id/sfv"
                 app:layout_constraintTop_toTopOf="parent"
                 android:layout_width="match_parent"
                 android:layout_height="200dp"/>
</android.support.constraint.ConstraintLayout>
```

MainActivity.kt

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initPlayer()
    }

    private fun initPlayer() {
        val path = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest.mp4"

        //创建线程池
        val threadPool = Executors.newFixedThreadPool(2)

        //创建视频解码器
        val videoDecoder = VideoDecoder(path, sfv, null)
        threadPool.execute(videoDecoder)

        //创建音频解码器
        val audioDecoder = AudioDecoder(path)
        threadPool.execute(audioDecoder)

        //开启播放
        videoDecoder.goOn()
        audioDecoder.goOn()
    }
}
```

So far, the decoding and playback of audio and video are basically realized. But if you actually run the code, you will find out: **why are the video and audio out of sync?**

This leads to the next inevitable problem, which is audio and video synchronization.

5. Audio and video synchronization

Sync signal source

Since video and audio are two independent tasks running, the decoding speed of video and audio is different, and the decoded data may not be displayed immediately.

As I said in the first article, there are two important time parameters for decoding: PTS and DTS, which are used to represent rendering time and decoding time, respectively. PTS is needed here.

There are generally three times in the player, the audio time, the video time, and the other is the system time. There are three time sources that can be used to achieve synchronization in this way:

- video timestamp
- audio timestamp
- external timestamp

VideoPTS

Under normal circumstances, since humans are more sensitive to sound, and the PTS of video decoding is usually not continuous, while the PTS of audio is relatively continuous, if the video is used as the synchronization signal source, the sound will basically be abnormal, and the playback of the picture will also be abnormal. It will be like playing at double speed.

AudioPTS

Then in the remaining two options, it is a good choice to use the PTS of the audio as the synchronization source to adapt the picture to the audio.

However, it is not adopted here, but the system time is used as the synchronization signal source. Because if the audio PTS is used as the synchronization source, a more complex synchronization mechanism is required, and there are more couplings between the audio and video.

system time

The system time is very suitable as a unified signal source. The audio and video are independent of each other and do not interfere with each other, and at the same time, the basic consistency can be guaranteed.

Synchronize audio and video

To achieve synchronization between audio and video, there are two points to consider here:

1. Comparison

After the decoded data comes out, check the time gap between the PTS timestamp and the current system flow. If it is fast, it will be delayed, and if it is slow, it will be played directly.

2. Correction

When entering the pause or decoding, and resuming the playback, it is necessary to correct the elapsed time of the system, subtract the paused time, and restore the real elapsed time, that is, the elapsed time.

Look back at the BaseDecoder decoding process:

```
abstract class BaseDecoder(private val mFilePath: String): IDecoder {
    //省略其他
    ......

    /**
     * 开始解码时间，用于音视频同步
     */
    private var mStartTimeForSync = -1L

    final override fun run() {
        if (mState == DecodeState.STOP) {
            mState = DecodeState.START
        }
        mStateListener?.decoderPrepare(this)

        //【解码步骤：1. 初始化，并启动解码器】
        if (!init()) return

        Log.i(TAG, "开始解码")

        while (mIsRunning) {
            if (mState != DecodeState.START &&
                mState != DecodeState.DECODING &&
                mState != DecodeState.SEEKING) {
                Log.i(TAG, "进入等待：$mState")

                waitDecode()

                // ---------【同步时间矫正】-------------
                //恢复同步的起始时间，即去除等待流失的时间
                mStartTimeForSync = System.currentTimeMillis() - getCurTimeStamp()
            }

            if (!mIsRunning ||
                mState == DecodeState.STOP) {
                mIsRunning = false
                break
            }

            if (mStartTimeForSync == -1L) {
                mStartTimeForSync = System.currentTimeMillis()
            }

            //如果数据没有解码完毕，将数据推入解码器解码
            if (!mIsEOS) {
                //【解码步骤：2. 见数据压入解码器输入缓冲】
                mIsEOS = pushBufferToDecoder()
            }

            //【解码步骤：3. 将解码好的数据从缓冲区拉取出来】
            val index = pullBufferFromDecoder()
            if (index >= 0) {
                // ---------【音视频同步】-------------
                if (mState == DecodeState.DECODING) {
                    sleepRender()
```

```
                }
                //【解码步骤：4. 渲染】
                render(mOutputBuffers!![index], mBufferInfo)
                //【解码步骤：5. 释放输出缓冲】
                mCodec!!.releaseOutputBuffer(index, true)
                if (mState == DecodeState.START) {
                    mState = DecodeState.PAUSE
                }
            }
            //【解码步骤：6. 判断解码是否完成】
            if (mBufferInfo.flags == MediaCodec.BUFFER_FLAG_END_OF_STREAM) {
                Log.i(TAG, "解码结束")
                mState = DecodeState.FINISH
                mStateListener?.decoderFinish(this)
            }
        }
        doneDecode()
        release()
    }
}
```

> When should time synchronization be performed without considering pause and resume?

The answer is: after the data is decoded, before rendering.

After the decoder enters the decoding state, it goes to [Decoding step: 3. Pull the decoded data out of the buffer]. At this time, if the data is valid, enter the comparison.

```
// ---------【音视频同步】-------------
final override fun run() {

    //......

    //【解码步骤：3. 将解码好的数据从缓冲区拉取出来】
    val index = pullBufferFromDecoder()
    if (index >= 0) {
        // ---------【音视频同步】-------------
        if (mState == DecodeState.DECODING) {
            sleepRender()
        }
        //【解码步骤：4. 渲染】
        render(mOutputBuffers!![index], mBufferInfo)
        //【解码步骤：5. 释放输出缓冲】
        mCodec!!.releaseOutputBuffer(index, true)
        if (mState == DecodeState.START) {
            mState = DecodeState.PAUSE
        }
    }

    //......
}

private fun sleepRender() {
    val passTime = System.currentTimeMillis() - mStartTimeForSync
    val curTime = getCurTimeStamp()
    if (curTime > passTime) {
        Thread.sleep(curTime - passTime)
    }
}

override fun getCurTimeStamp(): Long {
    return mBufferInfo.presentationTimeUs / 1000
}
```

The principle of synchronization is as follows:

Before entering the decoding, obtain the current system time and store it in mStartTimeForSync. After a frame of data is decoded, calculate the distance between the current system time and mStartTimeForSync, that is, the time that has been played. If the PTS of the current frame is greater than the lost time, enter sleep, otherwise Render directly.

> Consider time correction in case of suspension

After entering the pause, since the system time keeps going, and the mStartTimeForSync does not accumulate with the system time, when the playback is resumed, the mStartTimeForSync is added to the paused time period.

But there are several ways to calculate it:

One is to record the paused time. When resuming, subtract the paused time from the system time, which is the paused time period, and then use mStartTimeForSync to add this paused time period, which is the new mStartTimeForSync;

The other is to use the system time when resuming playback, minus the PTS of the frame currently being played, and the resulting value is mStartTimeForSync.

Here the second

```
if (mState != DecodeState.START &&
    mState != DecodeState.DECODING &&
    mState != DecodeState.SEEKING) {
    Log.i(TAG, "进入等待：$mState")

    waitDecode()

    // ---------【同步时间矫正】-------------
    //恢复同步的起始时间，即去除等待流失的时间
    mStartTimeForSync = System.currentTimeMillis() - getCurTimeStamp()
}
```

So far, from decoding to playback, to audio and video synchronization, a simple player is done.

The next article will briefly introduce how to use the MediaMuxer provided by Android to encapsulate Mp4. It will not involve encoding and decoding, but only decapsulating and encapsulating **data** . > **Encapsulation**] Prepare for the whole process.