# [Android audio and video development and upgrade: audio and video hard decoding articles] 2. Audio and video hard decoding process: encapsulate the basic decoding framework

简 jianshu.com/p/ff65ef5207ce

## [Android audio and video development and upgrade: audio and video hard decoding] 2. Audio and video hard decoding process: encapsulate the basic decoding framework

**Tutorial code: [ Github Portal ]**

### Table of contents

### In this article you can learn

> This article mainly introduces the process of Android using hard decoding API to implement hard decoding, including MediaCodec input and output buffering, MediaCodec decoding process, decoding code encapsulation and explanation.

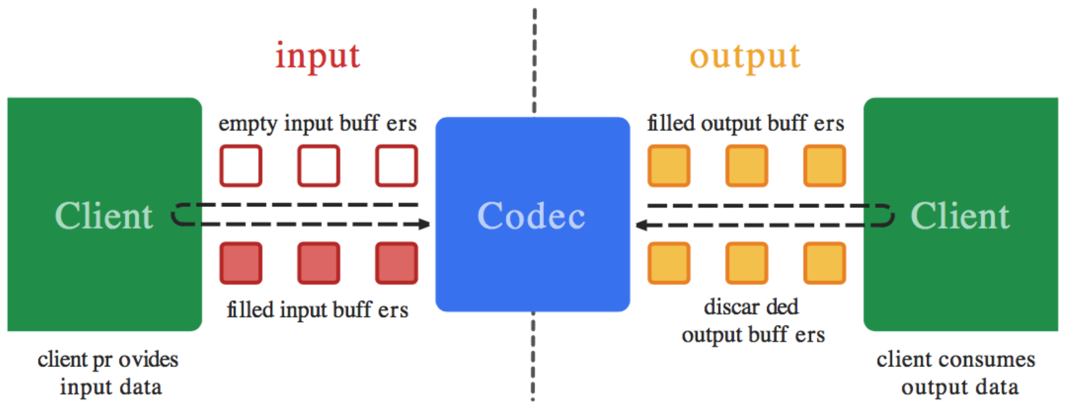
### 1. Introduction

MediaCodec is a codec interface introduced in Android 4.1 (api 16), and supports both audio and video encoding and decoding.

> Be sure to understand the next two pictures, because the subsequent code is written based on these two pictures.

data flow

First of all, let's take a look at the data flow of MediaCodec, which is also in the official Api document, and many articles will refer to it.



MediaCodec data stream

Taking a closer look, MediaCodec divides the data into two parts, input (left) and output (right), that is, two data buffers for input and output.

**input** : It is to input the data to be decoded (when decoding) or the data to be encoded (when encoding) to the client.

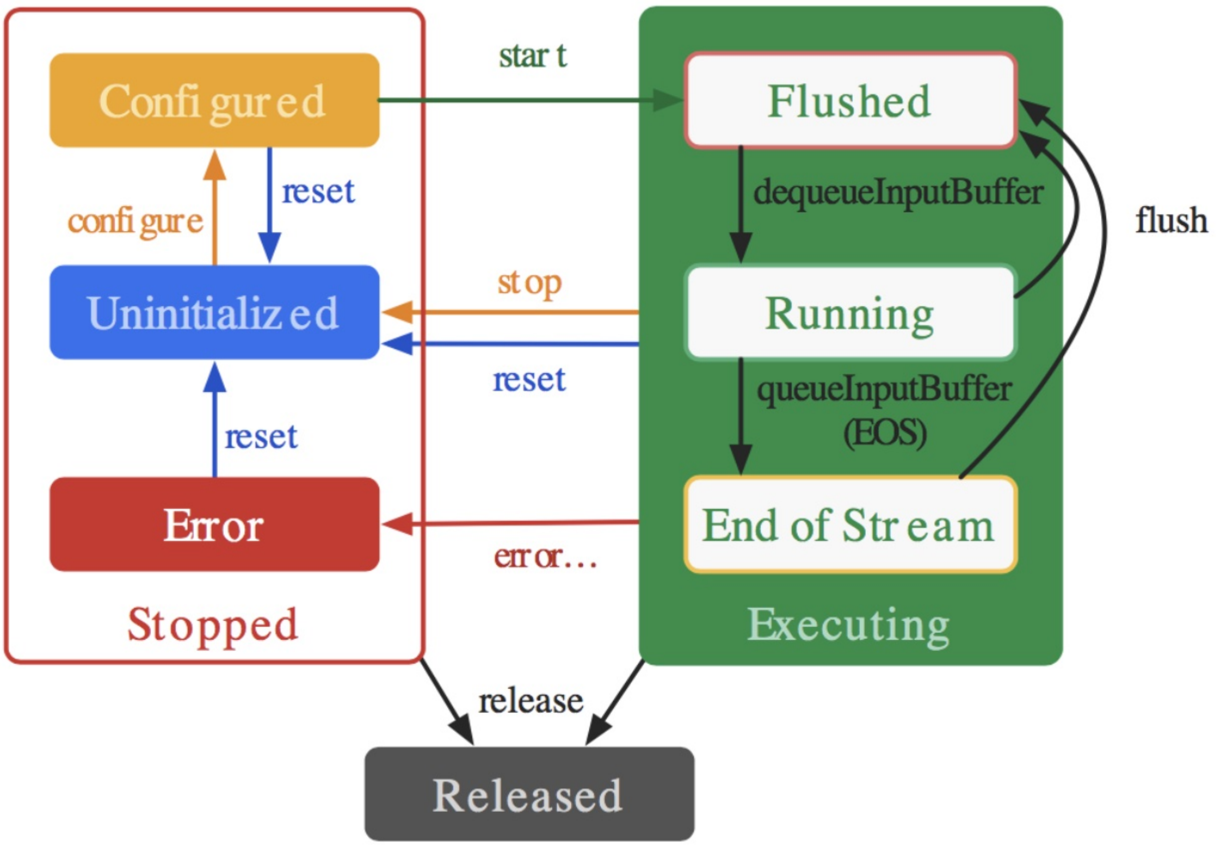
**output** : It is to output the decoded (decoding) or encoded (encoding) data to the client.

> MediaCodec uses an asynchronous way to process input and output data internally. MediaCodec will process the input data, fill it into the output buffer, and give it to the client for rendering or processing

**Note: After the client finishes processing the data, the output buffer must be released manually, otherwise, the MediaCodec output buffer will be occupied and the decoding cannot continue.**

state

It's still a state map from the official



MediaCodec state diagram

Take a closer look at this picture, and it is divided into three major states as a whole: Sotpped, Executing, and Released.

**Stoped** : Contains 3 small states: Error, Uninitialized, Configured.

> First, after creating a new MediaCodec, it will enter the Uninitialized state;
> secondly, after calling the configure method to configure the parameters, it will enter Configured;

**Executing** : It also contains 3 small states: Flushed, Running, and End of Stream.

> Again, after calling the start method, MediaCodec enters the Flushed state;
> then, after calling the dequeueInputBuffer method, it enters the Running state;
> finally, when the decoding/encoding ends, it enters the End of Stream (EOF) state.
> At this point, a video is processed.

**Released** : Finally, if you want to end the entire data processing process, you can call the release method to release all resources.

**So, what is the status of Flushed?**

As we can see from the figure, in the Running or End of Stream state, the flush method can be called to re-enter the Flushed state.

When we enter the End of Stream during the decoding process, the decoder will no longer receive input. At this time, we need to call the flush method to re-enter the state of receiving data.

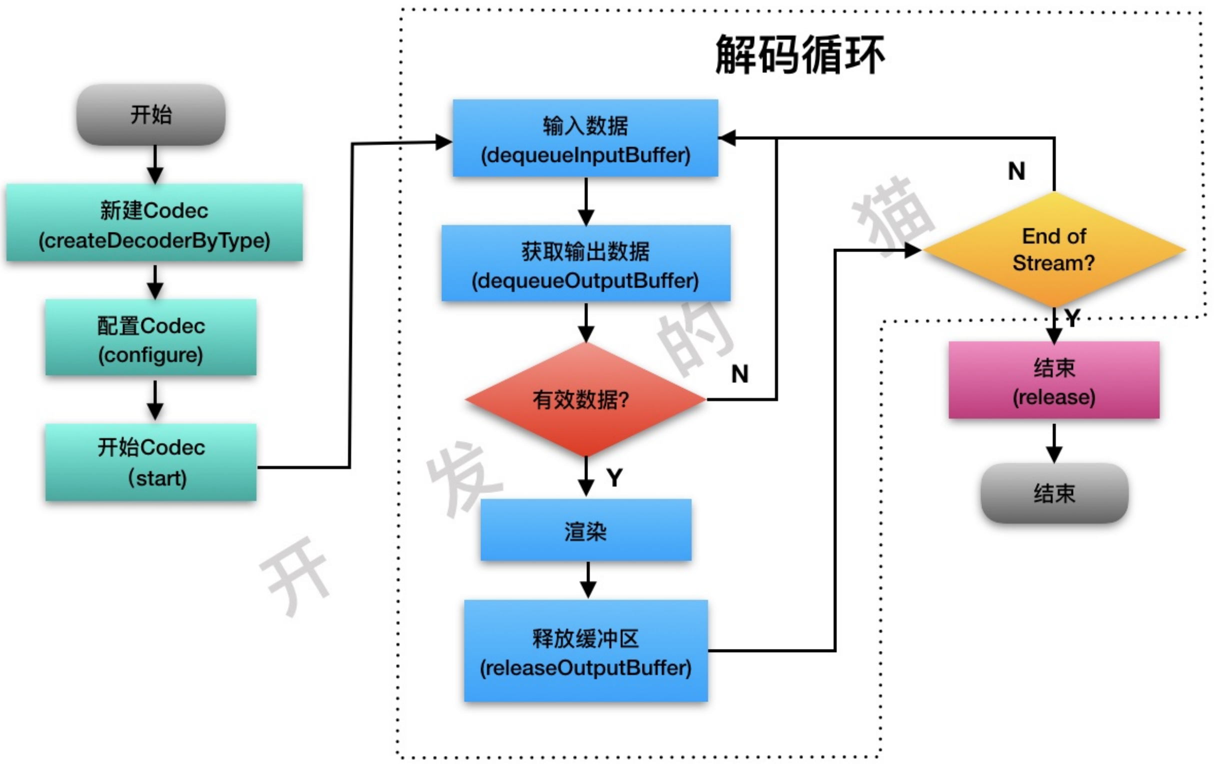
Or, we want to skip the video during playback. At this time, we need to Seek to the specified time point. At this time, we also need to call the flush method to clear the buffer, otherwise the decoding timestamp will be confused.

> Again, be sure to understand these two pictures well, because the subsequent code is written based on these two pictures.

## 2. Decoding process

MediaCodec has two working modes, asynchronous mode and synchronous mode. Here we use synchronous mode. For asynchronous mode, please refer to the official website example .

According to the official data flow diagram and state diagram, draw a most basic decoding process as follows:



解码循环
开始
新建Codec (createDecoderByType)
配置Codec (configure)
开始Codec (start)
输入数据 (dequeueInputBuffer)
获取输出数据 (dequeueOutputBuffer)
有效数据?
N
Y
渲染
释放缓冲区 (releaseOutputBuffer)
End of Stream?
N
Y
结束 (release)
结束
开发的猫

Decoding Flowchart

After initialization and configuration, enter the cyclic decoding process, continuously input data, then obtain the decoded data, and finally render it until all data decoding is completed (End of Stream).

## 3. Start decoding

According to the above flow chart, it can be found that the decoding process is basically the same regardless of audio or video, and the only difference lies in the two parts of [Configuration] and [Rendering].

## Define the decoder

Therefore, we abstract the entire decoding process into a decoding base class: BaseDecoder. In order to standardize the code and better expand, we first define a decoder: IDecoder, which inherits Runnable.

```
interface IDecoder: Runnable {

    /**
     * 暂停解码
     */
    fun pause()

    /**
     * 继续解码
     */
    fun goOn()

    /**
     * 停止解码
     */
    fun stop()

    /**
     * 是否正在解码
     */
    fun isDecoding(): Boolean

    /**
     * 是否正在快进
     */
    fun isSeeking(): Boolean

    /**
     * 是否停止解码
     */
    fun isStop(): Boolean

    /**
     * 设置状态监听器
     */
    fun setStateListener(l: IDecoderStateListener?)

    /**
     * 获取视频宽
     */
    fun getWidth(): Int

    /**
     * 获取视频高
     */
    fun getHeight(): Int

    /**
     * 获取视频长度
     */
    fun getDuration(): Long

    /**
     * 获取视频旋转角度
     */
    fun getRotationAngle(): Int
    /**
     * 获取音视频对应的格式参数
     */
    fun getMediaFormat(): MediaFormat?

    /**
     * 获取音视频对应的媒体轨道
     */
    fun getTrack(): Int

    /**
     * 获取解码的文件路径
     */
    fun getFilePath(): String
}
```

Defines some basic operations of the decoder, such as pause/continue/stop decoding, get the duration of the video, the width and height of the video, the decoding status, etc.

### Why inherit from Runnable?

> The synchronous mode decoding is used here, which requires continuous looping and pulling of data, which is a time-consuming operation. Therefore, we define the decoder as a Runnable, and finally put it into the thread pool for execution.

Next, inherit from IDecoder and define the base decoder BaseDecoder.

First look at the basic parameters:

```
abstract class BaseDecoder: IDecoder {
    //-------------线程相关-----------------------
    /**
     * 解码器是否在运行
     */
    private var mIsRunning = true

    /**
     * 线程等待锁
     */
    private val mLock = Object()

    /**
     * 是否可以进入解码
     */
    private var mReadyForDecode = false

    //---------------解码相关-----------------------
    /**
     * 音视频解码器
     */
    protected var mCodec: MediaCodec? = null

    /**
     * 音视频数据读取器
     */
    protected var mExtractor: IExtractor? = null

    /**
     * 解码输入缓存区
     */
    protected var mInputBuffers: Array<ByteBuffer>? = null

    /**
     * 解码输出缓存区
     */
    protected var mOutputBuffers: Array<ByteBuffer>? = null

    /**
     * 解码数据信息
     */
    private var mBufferInfo = MediaCodec.BufferInfo()

    private var mState = DecodeState.STOP

    private var mStateListener: IDecoderStateListener? = null

    /**
     * 流数据是否结束
     */
    private var mIsEOS = false

    protected var mVideoWidth = 0

    protected var mVideoHeight = 0
    //省略后面的方法
    ....
}
```

- First, we define thread-related resources, mIsRunning for judging whether to continue decoding, mLock for suspending threads, etc.
- Then, it is decoding related resources, such as MdeiaCodec itself, input and output buffers, decoding status, and so on.
- Among them, there is a decoding state DecodeState and audio and video data reader IExtractor.

## Define the decoding state

In order to facilitate the recording of the decoding state, an enumeration class is used here to represent

```
enum class DecodeState {
    /**开始状态*/
    START,
    /**解码中*/
    DECODING,
    /**解码暂停*/
    PAUSE,
    /**正在快进*/
    SEEKING,
    /**解码完成*/
    FINISH,
    /**解码器释放*/
    STOP
}
```

Defining audio and video data separators

As mentioned earlier, MediaCodec requires us to continuously feed data to the input buffer, so where does the data come from? It must be an audio and video file. The IExtractor here is used to extract the data stream in the audio and video file.

Android comes with an audio and video data reader MediaExtractor. Also, for the convenience of maintenance and scalability, we still set a reader IExtractor first.

```
interface IExtractor {
    /**
     * 获取音视频格式参数
     */
    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()
}
```

The most important method is readBuffer, which is used to read audio and video data streams

### Define the decoding process

Earlier, we only posted the parameter part of the decoder. Next, we posted the most important part, which is the decoding process part.

```
abstract class BaseDecoder: IDecoder {
    //省略参数定义部分，见上
    .......

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

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

        while (mIsRunning) {
            if (mState != DecodeState.START &&
                mState != DecodeState.DECODING &&
                mState != DecodeState.SEEKING) {
                waitDecode()
            }

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

            //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. 将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) {
                //【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) {
                mState = DecodeState.FINISH
                mStateListener?.decoderFinish(this)
            }
        }
        doneDecode()
        //【解码步骤：7. 释放解码器】
        release()
    }


    /**
     * 解码线程进入等待
     */
    private fun waitDecode() {
        try {
            if (mState == DecodeState.PAUSE) {
                mStateListener?.decoderPause(this)
            }
            synchronized(mLock) {
                mLock.wait()
            }
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * 通知解码线程继续运行
     */
    protected fun notifyDecode() {
        synchronized(mLock) {
            mLock.notifyAll()
        }
        if (mState == DecodeState.DECODING) {
            mStateListener?.decoderRunning(this)
        }
    }

    /**
     * 渲染
     */
    abstract fun render(outputBuffers: ByteBuffer,
                        bufferInfo: MediaCodec.BufferInfo)

    /**
     * 结束解码
     */
    abstract fun doneDecode()
}
```

In the run callback method of Runnable, the entire decoding process is integrated:

**[Decoding steps: 1. Initialize and start the decoder]**

```
abstract class BaseDecoder: IDecoder {
    //省略上面已有代码
    ......

    private fun init(): Boolean {
        //1.检查参数是否完整
        if (mFilePath.isEmpty() || File(mFilePath).exists()) {
            Log.w(TAG, "文件路径为空")
            mStateListener?.decoderError(this, "文件路径为空")
            return false
        }
        //调用虚函数，检查子类参数是否完整
        if (!check()) return false

        //2.初始化数据提取器
        mExtractor = initExtractor(mFilePath)
        if (mExtractor == null ||
            mExtractor!!.getFormat() == null) return false

        //3.初始化参数
        if (!initParams()) return false

        //4.初始化渲染器
        if (!initRender()) return false

        //5.初始化解码器
        if (!initCodec()) return false
        return true
    }

    private fun initParams(): Boolean {
        try {
            val format = mExtractor!!.getFormat()!!
            mDuration = format.getLong(MediaFormat.KEY_DURATION) / 1000
            if (mEndPos == 0L) mEndPos = mDuration

            initSpecParams(mExtractor!!.getFormat()!!)
        } catch (e: Exception) {
            return false
        }
        return true
    }

    private fun initCodec(): Boolean {
        try {
            //1.根据音视频编码格式初始化解码器
            val type = mExtractor!!.getFormat()!!.getString(MediaFormat.KEY_MIME)
            mCodec = MediaCodec.createDecoderByType(type)
            //2.配置解码器
            if (!configCodec(mCodec!!, mExtractor!!.getFormat()!!)) {
                waitDecode()
            }
            //3.启动解码器
            mCodec!!.start()

            //4.获取解码器缓冲区
            mInputBuffers = mCodec?.inputBuffers
            mOutputBuffers = mCodec?.outputBuffers
        } catch (e: Exception) {
            return false
        }
        return true
    }

    /**
     * 检查子类参数
     */
    abstract fun check(): Boolean

    /**
     * 初始化数据提取器
     */
    abstract fun initExtractor(path: String): IExtractor

    /**
     * 初始化子类自己特有的参数
     */
    abstract fun initSpecParams(format: MediaFormat)

    /**
     * 初始化渲染器
     */
    abstract fun initRender(): Boolean

    /**
     * 配置解码器
     */
    abstract fun configCodec(codec: MediaCodec, format: MediaFormat): Boolean
}
```

In the initialization method, it is divided into 5 steps. It looks complicated, but it is actually very simple.

1. Check if the parameters are complete: if the path is valid, etc.

2. Initialize Data Extractor: Initialize Extractor

3. Initialization parameters: extract some necessary parameters, duration, width, height, etc.

4. Initialize renderer: not required for video, AudioTracker for audio

5. Initialize Decoder: Initialize MediaCodec

   In initCodec(),

   ```
   val type = mExtractor!!.getFormat()!!.getString(MediaFormat.KEY_MIME)
   mCodec = MediaCodec.createDecoderByType(type)
   ```

When initializing MediaCodec:

1. First, obtain the encoding information MediaFormat of audio and video data through Extractor;
2. Then, query the encoding type in MediaFormat (such as video/avc, ie H264; audio/mp4a-latm, ie AAC);
3. Finally, call createDecoderByType to create the decoder.

**It should be noted that** since the initialization of audio and video is slightly different, several virtual functions are defined, and different things are handed over to subclasses to implement. The details will be explained in the next article [Audio and video playback: audio and video synchronization].

### [Decoding step: 2. Push data into the decoder input buffer]

Go directly to the pushBufferToDecoder method

```
abstract class BaseDecoder: IDecoder {
    //省略上面已有代码
    ......

    private fun pushBufferToDecoder(): Boolean {
        var inputBufferIndex = mCodec!!.dequeueInputBuffer(2000)
        var isEndOfStream = false

        if (inputBufferIndex >= 0) {
            val inputBuffer = mInputBuffers!![inputBufferIndex]
            val sampleSize = mExtractor!!.readBuffer(inputBuffer)
            if (sampleSize < 0) {
                //如果数据已经取完，压入数据结束标志：BUFFER_FLAG_END_OF_STREAM
                mCodec!!.queueInputBuffer(inputBufferIndex, 0, 0,
                    0, MediaCodec.BUFFER_FLAG_END_OF_STREAM)
                isEndOfStream = true
            } else {
                mCodec!!.queueInputBuffer(inputBufferIndex, 0,
                    sampleSize, mExtractor!!.getCurrentTimestamp(), 0)
            }
        }
        return isEndOfStream
    }
}
```

The following methods are called:

1. Query whether there is an available input buffer, returning the buffer index. The parameter 2000 is to wait for 2000ms. If you fill in -1, it will wait indefinitely.

```
var inputBufferIndex = mCodec!!.dequeueInputBuffer(2000)
```

2. Obtain the available buffer through the buffer index inputBufferIndex, and use the Extractor to extract the data to be decoded and fill it into the buffer.

```
val inputBuffer = mInputBuffers!![inputBufferIndex]
val sampleSize = mExtractor!!.readBuffer(inputBuffer)
```

3. Call queueInputBuffer to push data into the decoder.

```
mCodec!!.queueInputBuffer(inputBufferIndex, 0,
    sampleSize, mExtractor!!.getCurrentTimestamp(), 0)
```

> **Note** : If SampleSize returns -1, there is no more data.
> At this time, the last parameter of queueInputBuffer is to pass in the end tag MediaCodec.BUFFER_FLAG_END_OF_STREAM.

### [Decoding step: 3. Pull the decoded data out of the buffer]

Go directly to pullBufferFromDecoder()

```
abstract class BaseDecoder: IDecoder {
    //省略上面已有代码
    ......

    private fun pullBufferFromDecoder(): Int {
        // 查询是否有解码完成的数据，index >=0 时，表示数据有效，并且index为缓冲区索引
        var index = mCodec!!.dequeueOutputBuffer(mBufferInfo, 1000)
        when (index) {
            MediaCodec.INFO_OUTPUT_FORMAT_CHANGED -> {}
            MediaCodec.INFO_TRY_AGAIN_LATER -> {}
            MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED -> {
                mOutputBuffers = mCodec!!.outputBuffers
            }
            else -> {
                return index
            }
        }
        return -1
    }
}
```

First, call the dequeueOutputBuffer method to check whether there is available data that has been decoded, where mBufferInfo is used to obtain data frame information, and the second parameter is the waiting time. Here, wait for 1000ms, and fill in -1 to wait indefinitely.

```
var index = mCodec!!.dequeueOutputBuffer(mBufferInfo, 1000)
```

Second, determine the index type:

MediaCodec.INFO_OUTPUT_FORMAT_CHANGED: The output format changed

MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED: Input buffering changed

MediaCodec.INFO_TRY_AGAIN_LATER: No data available, will come back later

Greater than or equal to 0: there is available data, index is the output buffer index

### [Decoding step: 4. Rendering]

A virtual function render is called here, that is, the rendering is handed over to the subclass

### [Decoding step: 5. Release the output buffer]

Call the releaseOutputBuffer method to release the output buffer.

> Note: The second parameter is a boolean named render. This parameter is used to decide whether to display this frame of data during video decoding.

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

### [Decoding steps: 6. Determine whether decoding is complete]

Remember when we pushed data into the decoder, when sampleSize < 0, we pushed an end marker?

When this flag is received, the decoder knows that all data has been received. After all data is decoded, the end marker information will be added to the last frame of data, that is,

```
if (mBufferInfo.flags == MediaCodec.BUFFER_FLAG_END_OF_STREAM) {
    mState = DecodeState.FINISH
    mStateListener?.decoderFinish(this)
}
```

### [Decoding step: 7. Release the decoder]

After the while loop ends, release all resources. At this point, one decoding ends.

```
abstract class BaseDecoder: IDecoder {
    //省略上面已有代码
    ......

    private fun release() {
        try {
            mState = DecodeState.STOP
            mIsEOS = false
            mExtractor?.stop()
            mCodec?.stop()
            mCodec?.release()
            mStateListener?.decoderDestroy(this)
        } catch (e: Exception) {
        }
    }
}
```

Finally, other methods defined by the decoder (such as pause, goOn, stop, etc.) will not be described in detail. You can view the project source code.

**end**

I originally planned to include the audio and video playback parts in this article, but in the end I found that it was too long, not conducive to reading, and would be tiring to read. Therefore, if the actual playback part is integrated with the next one [Audio and Video Play: Audio and Video Synchronization], the content and length will be more reasonable.