RGBX

The RGBX 32 bit RGB format is stored in memory as 8 red bits, 8 green bits, 8 blue bits, and 8 ignored bits. For example:

unsigned char \*pPixels;

pPixels[0] = 0x60; // red

pPixels[1] = 0x30; // green

pPixels[2] = 0xE0; // blue

pPixels += 4; // skip to next pixel

The RGBA\_Premultiplied 32 bit RGBA format is stored in memory as 8 pre-multiplied red bits, 8 pre-multiplied green bits, 8 pre-multiplied blue bits, and 8 alpha bits. For example:

unsigned char \*pPixels;

pPixels[0] = (0x60\*0x80)>>8; // premultiplied red

pPixels[1] = (0x30\*0x80)>>8; // premultiplied green

pPixels[2] = (0xE0\*0x80)>>8; // premultiplied blue

pPixels[3] = 0x80; // alpha

pPixels += 4; // skip to next pixel

NDK 对 Surface 的Native层封装的接口:

头文件:

android-ndk-r10e/platforms/android-21/arch-arm/usr/include/android/native\_window\_jni.h

android-ndk-r10e/platforms/android-21/arch-arm/usr/include/android/native\_window.h

实现库：

android-21/arch-arm64/usr/lib/libandroid.so

android-19/arch-arm/usr/lib/libandroid.so

SDK中源码位置是

frameworks/native/include/android/native\_window.h AnativeWindow\_setBuffersGeometry AnativeWindow\_lock ANativeWindow\_unlockAndPost

frameworks/base/native/android/native\_window.cpp

frameworks//native/include/android/native\_window\_jni.h ANativeWindow\_fromSurface

frameworks//base/native/android/native\_window.cpp

ANativeWindow完整结构 并没有导出到NDK 所有不能访问 ANativeWindow中的成员，NDK只能用native\_window\_jni.h或者native\_window.h中的函数，并且与windows.h定义的接口不一样(但是NDK中的也是直接调用ANativeWIndow.operate或者 windows.h中定义的函数)

if (ANativeWindow\_lock(pSurfaceWindow, &buffer, NULL) == 0)

ANativeWindow\_setBuffersGeometry(pSurfaceWindow, width, height, WINDOW\_FORMAT\_RGBX\_8888);

ANativeWindow\_unlockAndPost(pSurfaceWindow);

ANativeWindow\_fromSurface(env, objSurface);

ANativeWindow\_release(pSurfaceWindow);

ANativeWindow 支持的 颜色空间:

/\*

\* Pixel formats that a window can use.

\*/

enum {

WINDOW\_FORMAT\_RGBA\_8888 = 1,

WINDOW\_FORMAT\_RGBX\_8888 = 2,

WINDOW\_FORMAT\_RGB\_565 = 4,

};

/\*

\* Pixel formats that a window can use.

\*/

enum {

WINDOW\_FORMAT\_RGBA\_8888 = 1,

WINDOW\_FORMAT\_RGBX\_8888 = 2,

WINDOW\_FORMAT\_RGB\_565 = 4,

};

struct ANativeWindow;

typedef struct ANativeWindow ANativeWindow;

typedef struct ANativeWindow\_Buffer {

int32\_t width; // 水平方向 像素

int32\_t height; // 垂直方向 像素

int32\_t stride; // 严格每行要有多少个像素 可能 >= width

// The number of \*pixels\* that a line in the buffer takes in memory. This may be >= width.

int32\_t format; // Buffer的颜色空间 ANativeWindow只支持buffer是RGBA/X8888 RGB565的颜色空间

void\* bits; // 指向Buffer

uint32\_t reserved[6]; // Do not touch.

} ANativeWindow\_Buffer;

void ANativeWindow\_acquire(ANativeWindow\* window); // 增加ANativeWindiw引用 ,防止ANativeWindow对象被删除

void ANativeWindow\_release(ANativeWindow\* window); // 释放 由 acquire 获得的引用

int32\_t ANativeWindow\_getWidth(ANativeWindow\* window); // window surface 的当前宽度 单位像素 返回负数 错误

int32\_t ANativeWindow\_getHeight(ANativeWindow\* window);// window surface 的当前高度 单位像素 返回负数 错误

int32\_t ANativeWindow\_getFormat(ANativeWindow\* window); // 返回当前 window surface的像素格式 返回负数 错误

/\*

\* 控制 the window buffers.的大小和颜色

\*

\* For all of these parameters, if 0 is supplied , then the window's base value will come back in force.

??? 如果参数提供0 ,那么会设置为原初始值

\* buffer的宽高 有区别与显示宽高 如果不一致的话 会被缩放显示

\* width and height必须同时为0或者同时都不为0

\*

\*/

int32\_t ANativeWindow\_setBuffersGeometry(ANativeWindow\* window,

int32\_t width, int32\_t height, int32\_t format);

int32\_t ANativeWindow\_lock(ANativeWindow\* window, ANativeWindow\_Buffer\* outBuffer,

ARect\* inOutDirtyBounds);

// 上锁下一个即将显示的ANativeWindow\_Buffer(window's next drawing surface)并返回outBuffer

// inOutDirtyBounds 是输出输入参数 输入调用者想修改(redraw)的区域范围 输出可能会修改(updated),代表调用者实际能修改的区域

int32\_t ANativeWindow\_unlockAndPost(ANativeWindow\* window);

// 抛出post一个新的 ANativeWindow\_Buffer去显示

// 解锁有lock函数上锁的 AnativeWindow\_Buffer (window's drawing surface )

native\_window\_jni.h

/\*\*

\* 返回 对象关联的 ANativeWindow对象(Native层)

\* 返回的时候 已经\_acquire 所以之后必须\_release

\*/

ANativeWindow\* ANativeWindow\_fromSurface(JNIEnv\* env, jobject surface);

系统SDK 中的ANativeWindow 定义和接口 使用

可以查看android/frameworks/av/servers/camera/libcameraservice/device1/CameraHardwareInterface.h 的使用

或者android/system/core/include/system/window.h 的定义

ANativeWindow 支持如下操作,但不应该直接调用 perform(window, NATIVE\_WIDOW\_<operations>) 而是调用上面的函数接口

struct ANativeWindow

{

int (\*perform)(struct ANativeWindow\* window,

int operation, ... );

\*

\* The valid operations are:

\* NATIVE\_WINDOW\_SET\_USAGE

\* NATIVE\_WINDOW\_CONNECT (deprecated)

\* NATIVE\_WINDOW\_DISCONNECT (deprecated)

\* NATIVE\_WINDOW\_SET\_CROP (private)

\* NATIVE\_WINDOW\_SET\_BUFFER\_COUNT

\* NATIVE\_WINDOW\_SET\_BUFFERS\_GEOMETRY (deprecated)

\* NATIVE\_WINDOW\_SET\_BUFFERS\_TRANSFORM

\* NATIVE\_WINDOW\_SET\_BUFFERS\_TIMESTAMP

\* NATIVE\_WINDOW\_SET\_BUFFERS\_DIMENSIONS

\* NATIVE\_WINDOW\_SET\_BUFFERS\_FORMAT

\* NATIVE\_WINDOW\_SET\_SCALING\_MODE (private)

\* NATIVE\_WINDOW\_LOCK (private)

\* NATIVE\_WINDOW\_UNLOCK\_AND\_POST (private)

\* NATIVE\_WINDOW\_API\_CONNECT (private)

\* NATIVE\_WINDOW\_API\_DISCONNECT (private)

\* NATIVE\_WINDOW\_SET\_BUFFERS\_USER\_DIMENSIONS (private)

\* NATIVE\_WINDOW\_SET\_POST\_TRANSFORM\_CROP (private)

\*

\*/

int (\*dequeueBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\*\* buffer, int\* fenceFd);

int (\*queueBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer, int fenceFd);

int (\*cancelBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer, int fenceFd);

}

部分命令不支持 ANativeWindow.perform ，而是调用专门的操作函数.eg ANativeWindow.dequeueBuffer AnativeWindow.cancelBuffer

下面是SDK中操作，定义在 window.h ( 比NDK中的支持操作要多)

设置 操作区域

native\_window\_set\_crop(pre-transformed buffer pixel coordinates)

=> window->perform(window, NATIVE\_WINDOW\_SET\_CROP, crop);

native\_window\_set\_post\_transform\_crop(post-transformed pixel coordinates)

=> window->perform(window, NATIVE\_WINDOW\_SET\_POST\_TRANSFORM\_CROP, crop);

设置buffer数目

native\_window\_set\_buffer\_count

=> window->perform(window, NATIVE\_WINDOW\_SET\_BUFFER\_COUNT, bufferCount);

设置buffer大小和格式

static inline int native\_window\_set\_buffers\_geometry( struct ANativeWindow\* window, int w, int h, int format)

=> window->perform(window, NATIVE\_WINDOW\_SET\_BUFFERS\_GEOMETRY, w, h, format);

设置buffer大小

static inline int native\_window\_set\_buffers\_dimensions( struct ANativeWindow\* window,int w, int h)

=> window->perform(window, NATIVE\_WINDOW\_SET\_BUFFERS\_DIMENSIONS,w, h);

设置buffer格式

static inline int native\_window\_set\_buffers\_format(struct ANativeWindow\* window, int format)

=> window->perform(window, NATIVE\_WINDOW\_SET\_BUFFERS\_FORMAT, format);

设置放大模式

static inline int native\_window\_set\_scaling\_mode(struct ANativeWindow\* window,int mode)

=> window->perform(window, NATIVE\_WINDOW\_SET\_SCALING\_MODE, mode);

获取一个 ANativeWindowBuffer

static inline int native\_window\_dequeue\_buffer\_and\_wait(ANativeWindow \*anw, struct ANativeWindowBuffer\*\* anb)

=> return anw->dequeueBuffer\_DEPRECATED(anw, anb);

头文件：

frameworks /av/include/ndk/NdkMediaFormat.h

SDK源码实现：

frameworks/av/media/ndk/

NdkMediaFormat.cpp NdkMediaMuxer.cpp NdkMediaCrypto.cpp

NdkMediaCodec.cpp NdkMediaExtractor.cpp NdkMediaDrm.cpp

还有错误号 NdkMediaError.h

没有导出MediaCodecInfo MediaCodecList

NDK ./android-21 之后才支持 才导出了下面的接口

所以AndroidMainfest.xml中必须 <uses-sdk android:minSdkVersion="22" minSdkVersion >=21

~~Eclipse中删除.cproject和project中的cdt 重新加载工程,最后重新add Native Supprot来更换声明的头文件~~

或者在工程属性 – C/C++ Genrnal Paths and Symbols添加头文件目录

#include "media/NdkMediaCodec.h"

#include "media/NdkMediaFormat.h"

#include "media/NdkMediaExtractor.h"

1. 获得AMediaFormat中的某个属性字段的值，不同属性字段类型不一样，如String/int32等

AMediaFormat \*format

AMediaFormat\_getString(format, AMEDIAFORMAT\_KEY\_MIME, &mime);

AMediaFormat\_getInt32(format, AMEDIAFORMAT\_KEY\_WIDTH , &width );

定义如下：(NdkMediaFormat.h) 除了get方法也有set方法

bool AMediaFormat\_getInt32(AMediaFormat\*, const char \*name, int32\_t \*out);

bool AMediaFormat\_getInt64(AMediaFormat\*, const char \*name, int64\_t \*out);

bool AMediaFormat\_getFloat(AMediaFormat\*, const char \*name, float \*out)

bool AMediaFormat\_getString(AMediaFormat\*, const char \*name, const char \*\*out);

void AMediaFormat\_setBuffer(AMediaFormat\*, const char\* name, void\* data, size\_t size);

2.获得所有字段(相当于把AMediaFormat转换成字符串)

const char\* AMediaFormat\_toString(AMediaFormat\*);

3. 常用属性字段

extern const char\* AMEDIAFORMAT\_KEY\_AAC\_PROFILE; ACC profile类型

extern const char\* AMEDIAFORMAT\_KEY\_BIT\_RATE; 比特率

extern const char\* AMEDIAFORMAT\_KEY\_CHANNEL\_COUNT; 通道数

extern const char\* AMEDIAFORMAT\_KEY\_COLOR\_FORMAT; 颜色空间

extern const char\* AMEDIAFORMAT\_KEY\_DURATION; 时长

extern const char\* AMEDIAFORMAT\_KEY\_HEIGHT; 高

extern const char\* AMEDIAFORMAT\_KEY\_WIDTH; 宽

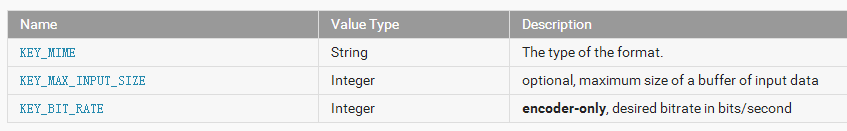
extern const char\* AMEDIAFORMAT\_KEY\_FRAME\_RATE; 编码参数：帧率

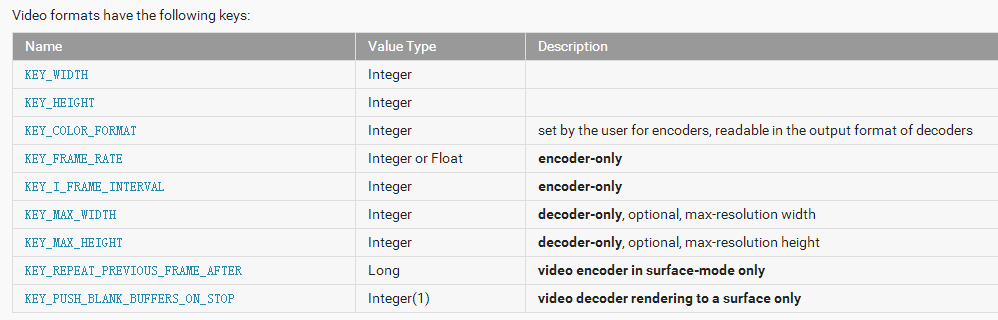
extern const char\* AMEDIAFORMAT\_KEY\_I\_FRAME\_INTERVAL; 编码参数：I帧间隔

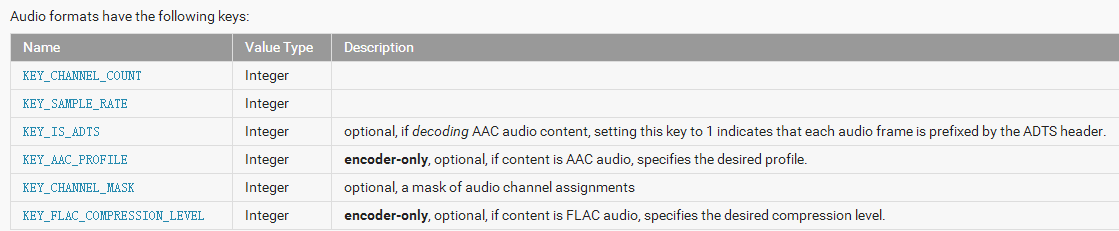
extern const char\* AMEDIAFORMAT\_KEY\_MAX\_INPUT\_SIZE; 输入数据的一个buffer长度 设置(configure)Codec Decoder的Buffer大小

extern const char\* AMEDIAFORMAT\_KEY\_MAX\_HEIGHT; 解码参数： 可以设置最大的分辨率 MTK平台测试 ，视频没有被截断??

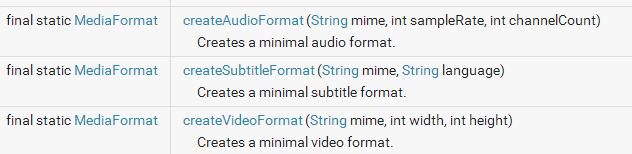
extern const char\* AMEDIAFORMAT\_KEY\_MAX\_WIDTH; 解码参数：







4. 创建 音频/视频/字幕的 MediaFormat



5.先创建指定类型的解码器，然后用MediaFormat来配置MediaCodec的参数

String mimeType = "video/avc";

MediaCodec decoder = MediaCodec.createDecoderByType(mimeType); // 先创建video/avc的解码器

MediaFormat format = MediaFormat.createVideoFormat(mimeType, 1920, 1080);

byte[] header\_sps = { 0, 0, 0, 1, 103, 100, 0, 40, -84, 52, -59, 1, -32, 17, 31, 120, 11, 80, 16, 16, 31, 0, 0, 3, 3, -23, 0, 0, -22, 96, -108 };

byte[] header\_pps = { 0, 0, 0, 1, 104, -18, 60, -128 };

format.setByteBuffer("csd-0", ByteBuffer.wrap(header\_sps));

format.setByteBuffer("csd-1", ByteBuffer.wrap(header\_pps));

format.setInteger(MediaFormat.KEY\_MAX\_INPUT\_SIZE, 1920 \* 1080);

format.setInteger("durationUs", 63446722);

decoder.configure(format, surface, null, 0); // 用MediaFormat来配置MediaCodec参数

decoder.start();

6. 读取mp4文件 获得sps pps参数

在H264中，SPS和PPS存在于NALU header中，而在MP4文件中，SPS和PPS存在于AVCDecoderConfigurationRecord， 首先要定位avcC.

H264 sps pps 参数 可以通过MediaExtra 得到的AMediaFormat得到 csd-0: data, csd-1: data

ACC 也有类似参数 在 csd-0:data

AMediaExtractor\* extract = AMediaExtractor\_new();

media\_status\_t err = AMediaExtractor\_setDataSourceFd(extract, nwc->fd, 0 , LONG\_MAX);

AMediaFormat \*format = AMediaExtractor\_getTrackFormat(extract, i); // 获取某一个轨道流Track的格式AMediaFormat

unsigned char \* sps = NULL ; size\_t sps\_length = 0;

unsigned char \* pps = NULL; size\_t pps\_length = 0;

AMediaFormat\_getBuffer(format, "csd-0",(void\*\*)&sps , &sps\_length);

AMediaFormat\_getBuffer(format, "csd-1",(void\*\*)&pps , &sps\_length);

pps sps参数 应该从容器中获得(文件) 或者NAL-header(ES流)

String mimeType = "video/avc";

MediaCodec decoder = MediaCodec.createDecoderByType(mimeType);

MediaFormat format = MediaFormat.createVideoFormat(mimeType, 1920, 1080);

byte[] header\_sps = { 0, 0, 0, 1, 103, 100, 0, 40, -84, 52, -59, 1, -32, 17, 31, 120, 11, 80, 16, 16, 31, 0, 0, 3, 3, -23, 0, 0, -22, 96, -108 };

byte[] header\_pps = { 0, 0, 0, 1, 104, -18, 60, -128 };

format.setByteBuffer("csd-0", ByteBuffer.wrap(header\_sps));

format.setByteBuffer("csd-1", ByteBuffer.wrap(header\_pps));

format.setInteger(MediaFormat.KEY\_MAX\_INPUT\_SIZE, 1920 \* 1080);

format.setInteger("durationUs", 63446722);

decoder.configure(format, surface, null, 0);

decoder.start();

7. 编解码器对象的状态

编解码器对象存在于三种状态之一：Stopped, Executing 或 Released。

Stoped状态实际是由三种状态的集成：Uninitialized, Configured以及 Error，

Executing状态的执行时通过三个子状态：Flushed, Running 以及 End-of-Stream。

创建一个编解码器的时候，它的状态是处于Uninitialized状态。

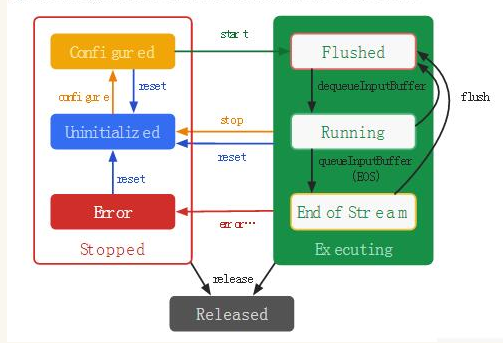
调用configure(…)方法配置它，以此进入Configured 状态。

调用start()方法转入Executing 状态。

在这个状态下你可以通过上述buffer队列操作过程数据。

调用stop()方法返回编解码器的Uninitialized 状态，因此这个编解码器需要再次configured 。当你使用完编解码器后，你必须调用release()方法释放其资源.

在极少情况下编解码器可能会遇到错误并进入Error 状态。这个错误可能是在队列操作时返回一个错误的值或者有时候产生了一个异常导致的。通过调用reset()方法使编解码器再次可用（Uninitialized）。你可以在任何状态调用reset()方法使编解码器返回Uninitialized 状态。否则，调用release()方法进入最终的Released 状态。



[Android 用MediaCodec实现视频硬解码](http://blog.csdn.net/halleyzhang3/article/details/11473961) <http://blog.csdn.net/halleyzhang3/article/details/11473961>

对YUV数据通过MediaCodec进行编码成H264 , 然后直接用UDP DatagramSocket DatagramPacket send到VLC 没有用rtsp协议

创建 MediaCodec编解码器

int width = 1280;

int height = 720;

int framerate = 20;

int bitrate = 2500000;

mediaCodec = MediaCodec.createEncoderByType("video/avc");

MediaFormat mediaFormat = MediaFormat.createVideoFormat("video/avc", width, height);

mediaFormat.setInteger(MediaFormat.KEY\_BIT\_RATE, bitrate);

mediaFormat.setInteger(MediaFormat.KEY\_FRAME\_RATE, framerate);

mediaFormat.setInteger(MediaFormat.KEY\_COLOR\_FORMAT, MediaCodecInfo.CodecCapabilities.COLOR\_FormatYUV420Planar);

mediaFormat.setInteger(MediaFormat.KEY\_I\_FRAME\_INTERVAL, 1); //关键帧间隔时间 单位s

mediaCodec.configure(mediaFormat, null, null, MediaCodec.CONFIGURE\_FLAG\_ENCODE);

mediaCodec.start();

获得所有的buffer

ByteBuffer[] inputBuffers = mediaCodec.getInputBuffers();

ByteBuffer[] outputBuffers = mediaCodec.getOutputBuffers();

获得当前空闲的buffer索引

int inputBufferIndex = mediaCodec.dequeueInputBuffer(-1);

把YUV数据拷贝给到空闲buffer(格式要是NV21),并把该buffer入队(queue)

ByteBuffer inputBuffer = inputBuffers[inputBufferIndex];

inputBuffer.clear();

inputBuffer.put(yuv420);

mediaCodec.queueInputBuffer(inputBufferIndex, 0, yuv420.length, 0, 0);

获得编码后的数据(NALU)

MediaCodec.BufferInfo bufferInfo = new **MediaCodec.BufferInfo**();

int outputBufferIndex = mediaCodec.**dequeueOutputBuffer**(bufferInfo,0);

while (outputBufferIndex >= 0) // 在Camera的 onPreviewFrame 中调用这个

{

ByteBuffer outputBuffer = outputBuffers[outputBufferIndex];

byte[] outData = new byte[**bufferInfo.size**];

**outputBuffer.get**(outData); // 从编码后的buffer中 把 所有数据放到outData

if(m\_info != null)

{

System.arraycopy(outData, 0, output, pos, outData.length);

pos += outData.length;

}

else //保存pps sps 只有开始时 第一个帧里有， 保存起来后面用

{

ByteBuffer spsPpsBuffer = ByteBuffer.wrap(outData); // 根据byte[] 创建一个ByteBuffer

if (spsPpsBuffer.getInt() == 0x00000001) // NALU 分隔 NALU header之前 一定是 00 00 00 01

{

m\_info = new byte[outData.length];

System.arraycopy(outData, 0, m\_info, 0, outData.length); // 这一帧数据保存到m\_info中

}

else

{

return -1;

}

}

mediaCodec.releaseOutputBuffer(outputBufferIndex, false);

outputBufferIndex = mediaCodec.dequeueOutputBuffer(bufferInfo, 0);

} // while

if(output[4] == 0x65) //key frame 编码器生成关键帧时只有 00 00 00 01 65 没有pps sps， 要加上

{

System.arraycopy(output, 0, yuv420, 0, pos);

System.arraycopy(m\_info, 0, output, 0, m\_info.length);

System.arraycopy(yuv420, 0, output, m\_info.length, pos); /// 也就是 output = m\_info + output ???

pos += m\_info.length;

}

[android硬编码h264-MediaCodec](http://blog.csdn.net/zqf_office/article/details/34180267) http://blog.csdn.net/zqf\_office/article/details/34180267

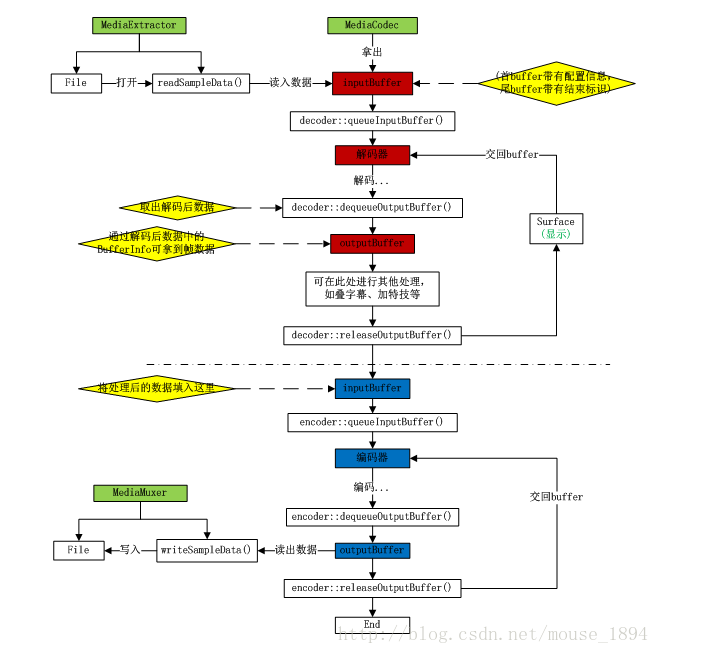
1. buf = **new** **byte**[camWidth \* camHeight \* 3 / 2];
2. cam.addCallbackBuffer(buf);
3. cam.setPreviewCallbackWithBuffer(**this**);

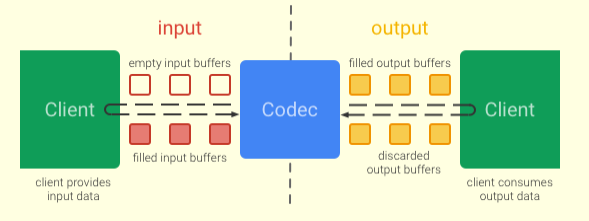
初始化编码器：

1. mediaCodec = MediaCodec.createEncoderByType("Video/AVC");
2. MediaFormat mediaFormat = MediaFormat.createVideoFormat(type, width, height);
3. mediaFormat.setInteger(MediaFormat.KEY\_BIT\_RATE, 125000);
4. mediaFormat.setInteger(MediaFormat.KEY\_FRAME\_RATE, 15);
5. mediaFormat.setInteger(MediaFormat.KEY\_COLOR\_FORMAT, MediaCodecInfo.CodecCapabilities.COLOR\_FormatYUV420Planar);
6. mediaFormat.setInteger(MediaFormat.KEY\_I\_FRAME\_INTERVAL, 5);
7. mediaCodec.configure(mediaFormat, **null**, **null**, MediaCodec.CONFIGURE\_FLAG\_ENCODE);
8. cam.startPreview();
9. mediaCodec.start();

喂数据了，这里的数据是来自摄像头的：

1. **public** **void** onFrame(**byte**[] buf, **int** offset, **int** length, **int** flag) {
2. ByteBuffer[] inputBuffers = mediaCodec.**getInputBuffers**();
3. ByteBuffer[] outputBuffers = mediaCodec.**getOutputBuffers**();
4. **int** inputBufferIndex = mediaCodec.dequeueInputBuffer(-1);
5. **if** (inputBufferIndex >= 0)
6. ByteBuffer inputBuffer = **inputBuffers**[**inputBufferIndex**];
7. inputBuffer.clear();
8. inputBuffer.put(buf, offset, length);
9. **mediaCodec.queueInputBuffer**(inputBufferIndex, 0, length, 0, 0);
10. }
11. MediaCodec.BufferInfo bufferInfo = **new** MediaCodec.BufferInfo();
12. **int** outputBufferIndex = **mediaCodec.dequeueOutputBuffer**(bufferInfo,0);
13. **while** (outputBufferIndex >= 0) {
14. ByteBuffer outputBuffer = outputBuffers[outputBufferIndex];
15. **if** (frameListener != **null**)
16. frameListener.onFrame(outputBuffer, 0, length, flag);
17. **mediaCodec.releaseOutputBuffer**(outputBufferIndex, **false**);
18. outputBufferIndex = mediaCodec.dequeueOutputBuffer(bufferInfo, 0);  // 获得下一个 while循环继续





它通过异步方式处理数据，并且使用了一组输入输出buffers。

在简单层面，你请求（或接收）到一个空的输入buffer，向里面填满数据并将它传递给编解码器处理。

这个编解码器将使用完这些数据并向所有空的输出buffer中的一个填充这些数据。

最终，你请求（或接受）到一个填充了数据的buffer,你可以使用其中的数据内容，并且在使用完后将其释放回编解码器

encoder::dequeueInputBuffer encoder::queueInputBuffer 用到的输入buffer队列 跟 输出的 buffer队列 不一样 Codec编/解码完成后 会给到输出buffer队列

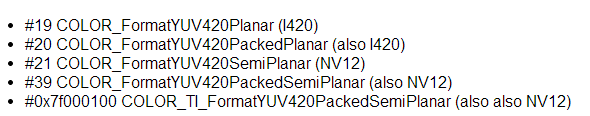
encoder :: dequeueOutputBuffer encoder:: releaseOutputBuffer 来获得编/解码后的数据

mMediaFormat.setInteger(MediaFormat.KEY\_COLOR\_FORMAT, MediaCodecInfo.CodecCapabilities.COLOR\_FormatSurface);

MediaCodecInfo.CodecCapabilities

|  |  |  |  |
| --- | --- | --- | --- |
| int | [COLOR\_Format16bitRGB565](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_Format16bitRGB565) | | 16 bits per pixel RGB color format, with 5-bit red & blue and 6-bit green component. |
| int | [COLOR\_Format24bitBGR888](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_Format24bitBGR888) | | 24 bits per pixel RGB color format, with 8-bit red, green & blue components. |
| int | [COLOR\_Format32bitABGR8888](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_Format32bitABGR8888) | | 32 bits per pixel RGBA color format, with 8-bit red, green, blue, and alpha components. |
| int | [COLOR\_FormatSurface](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatSurface) |

|  |  |  |
| --- | --- | --- |
| int | [COLOR\_FormatYUV420PackedPlanar](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420PackedPlanar) | *This constant was deprecated in API level 23. Use* [*COLOR\_FormatYUV420Flexible*](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420Flexible)*.* |
| int | [COLOR\_FormatYUV420PackedSemiPlanar](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420PackedSemiPlanar) | *This constant was deprecated in API level 23. Use* [*COLOR\_FormatYUV420Flexible*](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420Flexible)*.* |
| int | [COLOR\_FormatYUV420Planar](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420Planar) | *This constant was deprecated in API level 23. Use* [*COLOR\_FormatYUV420Flexible*](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420Flexible)*.* |
| int | [COLOR\_FormatYUV420SemiPlanar](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420SemiPlanar) | *This constant was deprecated in API level 23. Use* [*COLOR\_FormatYUV420Flexible*](file:///D:\Program%20Files%20(x86)\Android\android-sdk\docs\reference\android\media\MediaCodecInfo.CodecCapabilities.html#COLOR_FormatYUV420Flexible)*.* |



数据类型 与 Buffer

编解码器处理三种类型的数据：**压缩数据，原始音频数据，原始视频数据**。上述三种数据都可以通过ByteBuffers进行处理，但是你需要"原始视频数据"提供一个Surface来为提高编解码体验（译者注：显示视频图像）。**Surface直接使用本地视频数据buffers，而不是通过映射或复制的方式**(？？只在native 不会映射或拷贝到java？？)；因此，这样做的显得更加高效。通常在使用Surface的时候你不能够直接访问原始视频数据，{但是你可以使用ImageReader类来访问不可靠的解码后（或原始）的视频帧}。这可能仍然**比使用ByteBuffers更加有效率**，一些原始的buffers可能已经映射到了 direct ByteBuffers。当使用ByteBuffer模式，你可以通过使用Image类和getInput/OutputImage(int)来访问到原始视频数据帧。

**下面分压缩buffer 原始audio buffer 和 原始video buffer**

**Compressed Buffers**

压缩Buffers

针对视频类型是一个压缩的单帧。

针对音频数据通常是一个单个可访问单元(一个编码后的音频区段通常包含由特定格式类型决定的几毫秒音频数据)，但这种通常也不是十分严格，一个buffer可能包含多个可访问的音频单元。

在这两种情况下，buffers通常不开始或结束于任意的字节边界，而是结束于帧/可访问单元的边界。

#### 原始音频Buffers、Raw Audio Buffers

原始的音频数据buffers包含整个PCM音频帧数据

获得 某个解码后buffer中 指定通道的pcm数据

short[] getSamplesForChannel(MediaCodec codec, int bufferId, int channelIx) {

ByteBuffer outputBuffer = codec.getOutputBuffer(bufferId);

MediaFormat format = codec.getOutputFormat(bufferId);

ShortBuffer samples = outputBuffer.order(ByteOrder.nativeOrder()).asShortBuffer();

int numChannels = formet.getInteger(MediaFormat.KEY\_CHANNEL\_COUNT);

if (channelIx < 0 || channelIx >= numChannels) {

return null;

}

short[] res = new short[samples.remaining() / numChannels];

for (int i = 0; i < res.length; ++i) {

res[i] = samples.get(i \* numChannels + channelIx);

}

return res;

}

**Raw Video Buffers** 原始视频Buffers

    ByteBuffer模式下视频buffers的的展现是由他们的 color format确定的。你可以通过调用 getCodecInfo().getCapabilitiesForType(…).colorFormats方法获得其支持的颜色格式数组。视频编解码器可能支持三种类型的颜色格式：

* **native raw video format** 本地原始视频格式：被COLOR\_FormatSurface标记，其可与输入或输出Surface一起使用。
* **flexible YUV buffers**  灵活的YUV buffers(例如COLOR\_FormatYUV420Flexible)：这些与输入/输出Surface一起使用,以及在ByteBuffer模式中，通过调用getInput/OutputImage(int)方法
* 其他，具体的格式：通常只在ByteBuffer模式下被支持。有些颜色格式是特定供应商指定的。被定义在 MediaCodecInfo.CodecCapabilities中。颜色格式是一个很灵活的格式，你仍然可以使用 getInput/OutputImage(int)方法。

In Android 4.3 (API 18), MediaCodec was expanded to include a way to provide input through a Surface (via the createInputSurface method). This allows input to come from camera preview or OpenGL ES rendering. Android 4.3 was also the first release with MediaCodec tests in CTS, which helps ensure consistent behavior between devices.

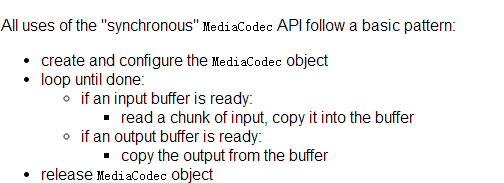
Android 4.3 also introduced MediaMuxer, which allows the output of the AVC codec (a raw H.264 elementary stream) to be converted to .MP4 format, with or without an associated audio stream.

Android 5.0 (API 21) introduced "asynchronous mode", which allows an app to provide a callback method that executes as buffers become available

1. Android4.3/API18 ,MediaCodec引入方法来 使得输入可以来自Surface(createInputSurface)，这使得输入可以来自camera预览或者OpenGL ES 显示(rendering)

2. Android4.3/API18 同时引入 MediaMuxer ，允许AVC编码器输出(H264裸流)到MediaMuxer，转换成MP4格式，可以带音频流或者不带

3. Android5.0 提供异步模式 ，使得buffers有效的时候会回调方法/接口



As of Android 4.4 (API 19), there is still no common input format. Nvidia Tegra 3 devices like the Nexus 7 (2012), and Samsung Exynos devices like the Nexus 10, want COLOR\_FormatYUV420Planar. Qualcomm Adreno devices like the Nexus 4, Nexus 5, and Nexus 7 (2013) want COLOR\_FormatYUV420SemiPlanar. TI OMAP devices like the Galaxy Nexus want COLOR\_TI\_FormatYUV420PackedSemiPlanar. (This is based on the format that is returned first when the AVC codec is queried.)

在Android4.4/API19，没有通用的输入格式：

Nvidia Tegra 3 平台 需要COLOR\_FormatYUV420Planar

Qualcomm Adreno平台 需要COLOR\_FormatYUV420SemiPlanar

TI OMAP平台 需要COLOR\_TI\_FormatYUV420PackedSemiPlanar

A more portable, and more efficient, approach is to use the API 18 Surface input API, demonstrated in the CameraToMpegTest sample. The down side of this is that you have to operate in RGB rather than YUV, which is a problem for image processing software. If you can implement the image manipulation in a fragment shader, perhaps by converting between RGB and YUV before and after your computations, you can take advantage of code execution on the GPU.

一个可移植高效的方法是使用API18 Surface输入接口 ，坏处是你必须处理RGB而不是YUV数据(图像软件处理时候可能遇上这个问题)

Note that the MediaCodec decoders may produce data in ByteBuffers using one of the above formats or in a proprietary format. For example, devices based on Qualcomm SoCs commonly use OMX\_QCOM\_COLOR\_FormatYUV420PackedSemiPlanar32m (#2141391876 / 0x7FA30C04).

MediaCodec解码器可能提供数据(在ByteBuffers中) 使用上述格式或者专有格式，

比如高通平台提供OMX\_QCOM\_COLOR\_FormatYUV420PackedSemiPlanar32m

Surface input uses COLOR\_FormatSurface, also known as OMX\_COLOR\_FormatAndroidOpaque (#2130708361 / 0x7F000789). For the full list, see OMX\_COLOR\_FORMATTYPE in OMX\_IVCommon.h.

As of API 21 you can work with an Image object instead. See the MediaCodec getInputImage() and getOutputImage() calls.

Surface输入使用COLOR\_FormatSurface，即OMX\_COLOR\_FormatAndroidOpaque，可以从OMX\_IVCommon.h查看所有颜色格式OMX\_COLOR\_FORMATTYPE

API21 可以使用 Image，接口是MediaCodec getInputImage() and getOutputImage()

Android MediaCodec stuff -- bigflake的博客

<http://bigflake.com/mediacodec/>

<http://www.polarxiong.com/archives/Android-YUV_420_888编码Image转换为I420和NV21格式byte数组.html>

<http://www.polarxiong.com/archives/Android-MediaCodec视频文件硬件解码-高效率得到YUV格式帧-快速保存JPEG图片-不使用OpenGL.html>

Image ImageFormat YUV\_420\_888 <http://www.polarxiong.com/archives/Android-Image类浅析-结合YUV_420_888.html>

使用 COLOR\_FormatYUV420Flexible 解码得到的ImageFormat是YUV\_420\_888的 ，通过codec. getOutputImage

也就是如果是输入解码的话，也要提供给codec.getInputImage

Android官方多媒体API Mediacodec翻译

<http://www.th7.cn/Program/Android/201604/819543.shtml>

Camera2参考代码

<https://github.com/googlesamples/android-Camera2Basic>

用了Image YUV420Flexible来实现编解码的demo

<http://blog.csdn.net/gh_home/article/details/52143102>

异步处理Buffer

<http://www.cnblogs.com/roger-yu/p/5635494.html>

MediaExtractor, MediaSync, MediaMuxer, MediaCrypto, MediaDrm, Image, Surface AudioTrack

mediaFormat.setInteger(MediaFormat.KEY\_MAX\_INPUT\_SIZE, 8192);这个参数是pcm buffer大小，可以根据自己的作适当修改,