<https://blog.csdn.net/pain_pain_pain_pain/article/details/52385767>

Android 6.0音频系统剖析（完全版）

* [rinswindqin](http://km.oa.com/user/rinswindqin)
* 2015年12月15日 10:53
* 浏览(433)
* 已收藏(54)
* 评论(5)
* 分享

# 1.   音频系统的总体架构

## 1.1. 概述

Audio系统是Android的一个重要的组成部分，所包含的组件如下图所示：

如上图所示，音频系统包括以下几个子模块：

l  AudioRecord和AudioTrack

这两个JAVA组件在android SDK中公开，提供了音频的播放和录制功能；

l  AudioManager、AudioSystem和AudioService

AudioManager、AudioSystem这两个组件封装了对AudioService的访问，向开发者提供了设置音频模式、音量、切换蓝牙耳机、对耳机和麦克风静音、开启振动模式等功能；

l  AudioFlinger

它是Audio系统的工作引擎，管理着系统中的输入输出音频流，并承担音频数据的混音，以及读写Audio硬件以实现数据的输入输出等工作；

l  AudioPolicy

它是Audio系统的策略控制中心，具有掌管系统中声音设备的选择和切换、音量控制等功能。

## 1.2. mediaserver进程

AudioTrack、AudioRecord工作在应用程序的进程中，仅仅是一个轻量级的对象，通过IBinder模式与AudioFlinger、AudioPolicy两大服务进行交互，对音频的所有操作，都是在AF和AP所在的进程中完成；而AF、AP两大服务，则是工作在系统的一个名叫mediaserver的进程中，如下图所示：

       mediaserver进程的代码在frameworks/av/media/mediaserver文件夹中，我们来看它的Android.mk文件：

LOCAL\_PATH:= $(call my-dir)

......................

include $(CLEAR\_VARS)

LOCAL\_SRC\_FILES:= \

         main\_mediaserver.cpp

LOCAL\_SHARED\_LIBRARIES := \

         libaudioflinger \

         .......................

         libradioservice

LOCAL\_STATIC\_LIBRARIES := \

        libicuandroid\_utils \

        libregistermsext

LOCAL\_C\_INCLUDES := \

.............

LOCAL\_MODULE:= mediaserver

LOCAL\_32\_BIT\_ONLY := true

include $(BUILD\_EXECUTABLE)

可以看出，mediaserver是一个native的进程，在系统启动的时候，将会由Init进程读取init.rc文件，获取这个进程的信息，并启动mediaserver作为一个常驻进程：

service media /system/bin/mediaserver

    class main

    user media

    group audio camera inet net\_bt net\_bt\_admin net\_bw\_acct drmrpc mediadrm

    ioprio rt 4

AP和AF两个服务，就是依附于mediaserver进程：

int main(int argc \_\_unused, char\*\* argv)

{

    ...................

    } else {

        .............

        sp<ProcessState> proc(ProcessState::self());

        sp<IServiceManager> sm = defaultServiceManager();

        ALOGI("ServiceManager: %p", sm.get());

   // 初始化AF服务

        AudioFlinger::instantiate();

        MediaPlayerService::instantiate();

        ResourceManagerService::instantiate();

        CameraService::instantiate();

   // 初始化AP服务

        AudioPolicyService::instantiate();

        SoundTriggerHwService::instantiate();

        RadioService::instantiate();

        registerExtensions();

        ProcessState::self()->startThreadPool();

        IPCThreadState::self()->joinThreadPool();

    }

}

## 1.3. 相关概念

在分析AudioTrack之前，有一些Android音频相关的概念需要讲解

1）数据加载类型

AudioTrack有两种数据加载类型：MODE\_STATIC和MODE\_STREAM。

这两种加载类型的不同点在于何时将音频数据写到AuidoTrack中去。

MODE\_STATIC模式下，需要在AudioTrack start之前，将这个缓冲区中的数据一次性write到AudioTrack 中去，后续就不需要再写入了。这种模式适合小文件的播放；

MODE\_STREAM模式下，需要在播放的过程中，不断调用AudioTrack的write方法，将音频数据写入到AudioTrack中。这种模式适合大文件，或者实时音频流的播放。

2）音频流类型

Android定义了几种不同的音频流类型，便于对音频进行管理和分类。这些类型定义在

Frameworks/base/media/com/android/media/AudioManager.java中，包括以下几种类型：

STREAM\_VOICE\_CALL --  电话通话音

STREAM\_SYSTEM -- android系统声音

STREAM\_RING -- 电话铃音

STREAM\_MUSIC -- 音乐

STREAM\_ALARM -- 警告音

STREAM\_NOTIFICATION -- 通知音

STREAM\_BLUETOOTH\_SCO -- 通话时连接到蓝牙耳机上播放出来的电话音

STREAM\_DTMF -- 通话时发送DTMF信号时的声音

STREAM\_TTS -- 语音识别

3）PCM数据格式的相关概念

PCM是一种未经过压缩的音频格式，包括PCM\_8和PCM\_16两种类型，PCM\_8的频点大小为1个字节，PCM\_16的频点大小为2个字节；

还有个采样率的概念，每秒采样多少个频点，以Hz为单位，比如采样率为8000Hz，说明每秒采样8000个点。

4）帧（Frame）

声音原本是模拟数据，对声音进行数字化处理，就是间隔一段时间采样一次，每次采样出来的数据称为一帧(frame)，因为每次采样的格式不一样，且声音可能是单声道，也可能有左右两个声道，所以不同的采样情况下，一帧的大小也不一样。

一个帧的大小和声音数据格式、单、双声道有关，格式为PCM8，单声道，则帧的大小为1个字节，如果格式为PCM16，双声道，则帧大小为4个字节。

Frame size = (PCM8 ? 1 : 2 ) \* (mono ? 1: 2) Bytes

AudioTrack中缓冲区的大小就是用frame count来表示，这个值标识了这个缓冲区能够容纳多少帧。frame count乘以frame size，才是缓冲区所占的字节数。

# 2.   音频播放过程剖析

## 2.1. 一个使用AudioTrack播放语音文件的例子

接下来给出一段利用AudioTrack播放PCM文件的代码，从这段代码出发，对音频播放过程中，音频系统的各个子模块是如何协调工作的，进行分析：

// 设置播放源

File sdCard = Environment.getExternalStorageDirectory();

File file = new File(sdCard , "test.pcm");

         Uri musicUri = Uri.fromFile(file);

// 通过getMinBufferSize，根据文件的采样参数，来获取最小缓冲区的大小

         int bufSize = AudioTrack.getMinBufferSize(16000,

                                                               AudioFormat.CHANNEL\_OUT\_MONO,

                                                               AudioFormat.ENCODING\_PCM\_16BIT);

// 创建一个AudioTrack对象，流类型为STREAM\_MUSIC，采样率为16000Hz，单声道，采样类型为PCM\_16，数据加    // 载类型为MODE\_STREAM

         AudioTrack aut = new AudioTrack(AudioManager.STREAM\_MUSIC,

                                                               16000, AudioFormat.CHANNEL\_OUT\_MONO,

                                                               AudioFormat.ENCODING\_PCM\_16BIT, bufSize

                                                               AudioTrack.MODE\_STREAM);

// 开始播放

         aut.play();

         byte[] buffer = new byte[bufSize ];

         FileInputStream stream = null;

         try {

                  stream = new FileInputStream(file);

                  int size = 0;

                  while ((size = stream.read(buffer)) > 0) {

           // MODE\_STREAM类型需要不断从文件中读取音频数据，并写入AudioTrack

                           aut.write(buffer, 0, size);

                  }

         } catch (Exception e) {

         } finally {

                  if (stream != null) {

                           stream.close();

                  }

｝

// 关闭AudioTrack

         aut.stop();

         aut.release();

通过这段代码，我们初始化了一个AudioTrack，用于播放SD卡中的一个test.pcm文件，定义采样率为16000Hz，使用单声道播放，音频文件的格式为PCM\_16Bit，数据加载类型为MODE\_STREAM。这段代码不断从音频文件中读出PCM\_16BIT的声音数据，写入到AudioTrack中，写入完成后，关闭AudioTrack，并调用release释放资源。

看了这段代码，我们可能会问这么几个问题：

l  getMinBufferSize是怎么确定AudioTrack内部缓冲的大小的？

l  AudioTrack初始化时，各个模块都做了什么事情？建立了一些什么对象？各个对象之间有什么关系？

l  通过AudioTrack.write方法写入的音频数据，写入到哪块内存中？如何写入这块内存中？这块内存是何时创建的？这些数据是由哪个子模块消费的？这个子模块是如何访问这块内存并从中拉取数据的？

l  音频数据是如何传递给音频硬件并从音频设备中播放出来的？

l  Android设备一般有好几个音频设备，比如听筒、扬声器、普通耳机、蓝牙耳机等等，Android系统怎么知道当前该用哪个音频设备播放这个文件？

l  write方法会阻塞么？

l  AudioTrack.stop调用过后，音乐会马上停止么？AudioTrack.release调用过后呢？

接下来我们会深入到AudioTrack之下，看看在播放过程中做了一些什么事情，之后再回来看这几个问题。

## 2.2. AudioTrack初始化时做了什么事情

### 2.2.1.    getMinBufferSize

我们注意到，在MODE\_STREAM模式下，并没有为写入数据创建一块内存，却调用了getMinBufferSize来计算最小需要一块多大的缓冲区供我们写入数据，从这里我们得知，在AudioTrack初始化的时候，肯定有个模块根据入参传入的缓冲区大小，为此次音频播放分配内存。

在这之前，先看看这个最小的缓冲区大小是如何确定的。

static public int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat) {

        // 计算声道数

        int channelCount = 0;

        switch(channelConfig) {

        case AudioFormat.CHANNEL\_OUT\_MONO:

        case AudioFormat.CHANNEL\_CONFIGURATION\_MONO:

            channelCount = 1;

            break;

       .................

        // 判断是否支持参数传入的采样率

        if ( (sampleRateInHz < SAMPLE\_RATE\_HZ\_MIN) || (sampleRateInHz > SAMPLE\_RATE\_HZ\_MAX) ) {

            loge("getMinBufferSize(): " + sampleRateInHz + " Hz is not a supported sample rate.");

            return ERROR\_BAD\_VALUE;

        }

        // 在native层实现

        int size = native\_get\_min\_buff\_size(sampleRateInHz, channelCount, audioFormat);

        ..............

        return size;

    }

AudioTrack的JNI代码在frameworks/base/core/jni中，看看android\_media\_AudioTrack.cpp中的实现：

static jint android\_media\_AudioTrack\_get\_min\_buff\_size(JNIEnv \*env,  jobject thiz,

    jint sampleRateInHertz, jint channelCount, jint audioFormat) {

size\_t frameCount;

    // 实际上调用了AudioTrack Native端的接口，获取硬件支持的最小帧数

    const status\_t status = AudioTrack::getMinFrameCount(&frameCount, AUDIO\_STREAM\_DEFAULT,

            sampleRateInHertz);

    if (status != NO\_ERROR) {

        ALOGE("AudioTrack::getMinFrameCount() for sample rate %d failed with status %d",

                sampleRateInHertz, status);

        return -1;

    }

    const audio\_format\_t format = audioFormatToNative(audioFormat);

if (audio\_is\_linear\_pcm(format)) {

   // 如果是PCM数据，还需要乘以声道数量，以及每个频点的大小（PCM\_16为2字节）

        const size\_t bytesPerSample = audio\_bytes\_per\_sample(format);

        return frameCount \* channelCount \* bytesPerSample;

    } else {

        return frameCount;

    }

}

还得看AudioTrack::getMinFrameCount函数：

status\_t AudioTrack::getMinFrameCount(

        size\_t\* frameCount,

        audio\_stream\_type\_t streamType,

        uint32\_t sampleRate)

{

    ...................

    uint32\_t afSampleRate;

status\_t status;

// 通过stream类型，获取输出的最大采样率

    status = AudioSystem::getOutputSamplingRate(&afSampleRate, streamType);

    if (status != NO\_ERROR) {

        ALOGE("Unable to query output sample rate for stream type %d; status %d",

                streamType, status);

        return status;

    }

size\_t afFrameCount;

// 通过stream类型，获取硬件内部缓冲的大小

    status = AudioSystem::getOutputFrameCount(&afFrameCount, streamType);

    if (status != NO\_ERROR) {

        ALOGE("Unable to query output frame count for stream type %d; status %d",

                streamType, status);

        return status;

    }

uint32\_t afLatency;

// 通过stream类型，获取硬件的延迟事件

    status = AudioSystem::getOutputLatency(&afLatency, streamType);

    if (status != NO\_ERROR) {

        ALOGE("Unable to query output latency for stream type %d; status %d",

                streamType, status);

        return status;

}

// 计算最小帧数

    \*frameCount = calculateMinFrameCount(afLatency, afFrameCount, afSampleRate, sampleRate, 1.0f);

    ..................

}

getMinBufSize会综合考虑硬件的情况（诸如是否支持采样率，硬件本身的延迟情况等）后，得出一个最小缓冲区的大小。一般我们分配的缓冲大小会是它的整数倍，这个缓冲区大小，将作为参数传入AudioTrack的构造方法中，AudioTrack将依赖这个值，为音频播放的流程分配内存。

### 2.2.2.    AudioTrack

从分析getMinBufSize这个方法可以看出，AudioTrack这个JAVA类的大部分功能，都是由Native端的AudioTrack这个C++类来实现的。这是很自然的事情，因为AudioPolicy和AudioFlinger这两大服务，都是Native服务，所以必定存在一个Native的AudioTrack类与它们进行交互，而Android SDK为了给上层应用提供JAVA接口，便通过JNI将AudioTrack Native类的各个方法，映射到了JAVA层。

AudioTrack JAVA的构造方法如下所示：

 public AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int audioFormat,

            int bufferSizeInBytes, int mode)

    throws IllegalArgumentException {

        this(streamType, sampleRateInHz, channelConfig, audioFormat,

                bufferSizeInBytes, mode, AudioSystem.AUDIO\_SESSION\_ALLOCATE);

    }

这个方法的参数为：

streamType ：音频流类型

sampleRateInHz ： 采样率

channelConfig ： 单声道/双声道

audioFormat ： 采样类型

bufferSizeInBytes ： 缓冲区大小

Mode ： 数据加载类型

继续看构造方法调用了另外一个构造方法：

 public AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int audioFormat,

            int bufferSizeInBytes, int mode, int sessionId)

    throws IllegalArgumentException {

        // mState already == STATE\_UNINITIALIZED

        this((new AudioAttributes.Builder())

                    .setLegacyStreamType(streamType)

                    .build(),

                (new AudioFormat.Builder())

                    .setChannelMask(channelConfig)

                    .setEncoding(audioFormat)

                    .setSampleRate(sampleRateInHz)

                    .build(),

                bufferSizeInBytes,

                mode, sessionId);

    }

这个构造方法将参数打包进AudioAttributes类中，AudioAttributes类是6.0上新增的一个类，用于描述音频的一些属性，包括：

l  usage :

这段音频用来做什么，AudioAttributes中定义了一些常量，比如USAGE\_MEDIA，表示音频用于音乐、电影等；USAGE\_VOICE\_COMMUNICATION，表示音频用于语音通话，比如电话或者VOIP；USAGE\_VOICE\_COMMUNICATION\_SIGNALLING，表示电话过程中的消息音，比如DTMP、忙音等；USAGE\_ALARM，表示闹钟...... 具体可以查看AudioAttributes.java

l  Content type :

这段音频包含了哪一种数据：CONTENT\_TYPE\_MUSIC，表示音频类型为音乐；CONTENT\_TYPE\_MOVIE，代表电影音轨；CONTENT\_TYPE\_SONIFICATION，代表一些伴随用户操作的短促音频，比如通知音、闹钟、按键音、游戏人物获得奖励时的提示音等；CONTENT\_TYPE\_SPEECH，代表文本转换后的语音......

Native层中也有一个结构体audio\_attributes\_t，与JAVA层的AudioAttributes类相对应。

查看这个构造方法的代码，发现它又调用了另外一个构造方法：

  public AudioTrack(AudioAttributes attributes, AudioFormat format, int bufferSizeInBytes,

            int mode, int sessionId)

                    throws IllegalArgumentException {

        ..............

        // 由native层来实现

        int initResult = native\_setup(new WeakReference<AudioTrack>(this), mAttributes,

                mSampleRate, mChannelMask, mChannelIndexMask, mAudioFormat,

                mNativeBufferSizeInBytes, mDataLoadMode, session);

        ..............

    }

从此处开始，进入了native层：

static jint

android\_media\_AudioTrack\_setup(JNIEnv \*env, jobject thiz, jobject weak\_this,

        jobject jaa,

        jint sampleRateInHertz, jint channelPositionMask, jint channelIndexMask,

        jint audioFormat, jint buffSizeInBytes, jint memoryMode, jintArray jSession) {

...................

// 创建Native AudioTrack

sp<AudioTrack> lpTrack = new AudioTrack();

...................

    // initialize the native AudioTrack object

    status\_t status = NO\_ERROR;

    switch (memoryMode) {

case MODE\_STREAM:

  // 当数据加载类型为MODE\_STREAM类型时

        status = lpTrack->set(

                AUDIO\_STREAM\_DEFAULT,// stream type, but more info conveyed in paa (last argument)

                sampleRateInHertz,

                format,// word length, PCM

                nativeChannelMask,

                frameCount,

                AUDIO\_OUTPUT\_FLAG\_NONE,

                audioCallback, &(lpJniStorage->mCallbackData),//callback, callback data (user)

                0,// notificationFrames == 0 since not using EVENT\_MORE\_DATA to feed the AudioTrack

                0,// shared mem

                true,// thread can call Java

                sessionId,// audio session ID

                AudioTrack::TRANSFER\_SYNC,

                NULL,                         // default offloadInfo

                -1, -1,                       // default uid, pid values

                paa);

        break;

    case MODE\_STATIC:

        if (!lpJniStorage->allocSharedMem(buffSizeInBytes)) {

            ALOGE("Error creating AudioTrack in static mode: error creating mem heap base");

            goto native\_init\_failure;

        }

   // 当数据加载类型为MODE\_STATIC类型时

        status = lpTrack->set(

                AUDIO\_STREAM\_DEFAULT,// stream type, but more info conveyed in paa (last argument)

                sampleRateInHertz,

                format,// word length, PCM

                nativeChannelMask,

                frameCount,

                AUDIO\_OUTPUT\_FLAG\_NONE,

                audioCallback, &(lpJniStorage->mCallbackData),//callback, callback data (user));

                0,// notificationFrames == 0 since not using EVENT\_MORE\_DATA to feed the AudioTrack

                lpJniStorage->mMemBase,// shared mem

                true,// thread can call Java

                sessionId,// audio session ID

                AudioTrack::TRANSFER\_SHARED,

                NULL,                         // default offloadInfo

                -1, -1,                       // default uid, pid values

                paa);

        break;

    default:

        ALOGE("Unknown mode %d", memoryMode);

        goto native\_init\_failure;

    }

.................

    setAudioTrack(env, thiz, lpTrack);

.        ..................

}

从代码中可以看出，两种不同的数据加载类型，调用Native AudioTrack的set方法的参数是不同的；这个set方法是关键，首先我们看它的参数：

status\_t AudioTrack::set(

        audio\_stream\_type\_t streamType,

        uint32\_t sampleRate,

        audio\_format\_t format,

        audio\_channel\_mask\_t channelMask,

        size\_t frameCount,

        audio\_output\_flags\_t flags,

        callback\_t cbf,

        void\* user,

        uint32\_t notificationFrames,

        const sp<IMemory>& sharedBuffer,

        bool threadCanCallJava,

        int sessionId,

        transfer\_type transferType,

        const audio\_offload\_info\_t \*offloadInfo,

        int uid,

        pid\_t pid,

        const audio\_attributes\_t\* pAttributes,

        bool doNotReconnect)

参数列表：

streamType ：音频流类型

sampleRate： 采样率

format： 采样类型，PCM\_8还是PCM\_16

channelMask： 单声道/双声道

frameCount： 缓冲区大小，以帧数为单位

flags ： 音频输出标志，见system/media/audio/include/system/audio.h中的定义，取值为AUDIO\_OUTPUT\_FLAGA\_\*

cbf ： JNI的回调函数

user ： 回调函数的参数

notificationFrames : 如果不是采用EVENT\_MORE\_DATA给AudioTrack填充数据，就取值为0

sharedBuffer ： 外部缓冲区，数据加载类型为MODE\_STATIC的时候会设置，MODE\_STREAM时为NULL

threadCanCallJava ： 线程是否能够回调JAVA层代码

sessionId ： 会话ID

transfer\_type ： 定义数据如何传递给AudioTrack，见AudioTrack.h中的TRANSFER\_\*

offloadInfo ：

Uid ： 取-1代表当前用户ID

Pid ： 取-1代表当前进程ID

pAttributes ： 音频属性

以MODE\_STREAM类型为例，JNI调用AudioTrack Native的set函数时，实参是这么配置的：

1) streamType设置为AUDIO\_STREAM\_DEFAULT，是因为更多信息可以通过audio\_attributes\_t参数获取

2) sampleRate的值由调用者设置

3) format的值由调用者设置

4) channelMask的值由调用者设置

5) frameCount的值由调用者设置

6) flags 为AUDIO\_OUTPUT\_FLAG\_NONE

7) cbf 和user设置为JAVA层传下来的回调函数

8) notificationFrames 设置为0，代表不使用EVENT\_MORE\_DATA的方式拉取数据

9) sharedBuffer 设置为NULL，代表MODE\_STREAM类型

10) threadCanCallJava 设置为true

11) sessionId 设置为JAVA层传递下来的值，为AUDIO\_SESSION\_ALLOCATE

12) Transfer\_type 设置为TRANSFER\_SYNC，代表数据持续写入到缓冲区中

13) offloadInfo 设置为NULL

14) Uid和pid都设置为-1，代表使用当前进程的uid和pid

15) pAttributes 由JAVA层传递进来的AudioAttributes对象，在JNI中转换为audio\_attributes\_t对象，包含了content\_type和usage等参数

接下来看看set方法：

status\_t AudioTrack::set(

        ..............)

{

    // 首先进行一些参数有效性判断和类变量赋值

    ..................

if (cbf != NULL) {

    // 如果允许回调，那么创建AudioTrackThread类来处理

        mAudioTrackThread = new AudioTrackThread(\*this, threadCanCallJava);

        mAudioTrackThread->run("AudioTrack", ANDROID\_PRIORITY\_AUDIO, 0 /\*stack\*/);

        // thread begins in paused state, and will not reference us until start()

    }

    // 调用createTrack\_l方法来创建IAudioTrack代理

status\_t status = createTrack\_l();

    ......................

    return NO\_ERROR;

}

这个方法的主体是在createTrack\_l函数中实现的。

status\_t AudioTrack::createTrack\_l()

{

    // 获取AF服务的代理端，通过IBinder访问服务端

    const sp<IAudioFlinger>& audioFlinger = AudioSystem::get\_audio\_flinger();

    if (audioFlinger == 0) {

        ALOGE("Could not get audioflinger");

        return NO\_INIT;

    }

    if (mDeviceCallback != 0 && mOutput != AUDIO\_IO\_HANDLE\_NONE) {

        AudioSystem::removeAudioDeviceCallback(mDeviceCallback, mOutput);

    }

    audio\_io\_handle\_t output;

    audio\_stream\_type\_t streamType = mStreamType;

    audio\_attributes\_t \*attr = (mStreamType == AUDIO\_STREAM\_DEFAULT) ? &mAttributes : NULL;

    status\_t status;

// getOutputForAttr通过audio\_attributes\_t选择底层的一个output，并返回它的唯一标识，以及具体的stream type,;

// 这个方法实际上是由AudioPolicy来完成的，这个以后再分析

    status = AudioSystem::getOutputForAttr(attr, &output,

                                           (audio\_session\_t)mSessionId, &streamType, mClientUid,

                                           mSampleRate, mFormat, mChannelMask,

                                           mFlags, mSelectedDeviceId, mOffloadInfo);

    if (status != NO\_ERROR || output == AUDIO\_IO\_HANDLE\_NONE) {

        ALOGE("Could not get audio output for session %d, stream type %d, usage %d, sample rate %u, format %#x,"

              " channel mask %#x, flags %#x",

              mSessionId, streamType, mAttributes.usage, mSampleRate, mFormat, mChannelMask, mFlags);

        return BAD\_VALUE;

    }

    // 获取硬件的延时时间

    status = AudioSystem::getLatency(output, &mAfLatency);

    if (status != NO\_ERROR) {

        ALOGE("getLatency(%d) failed status %d", output, status);

        goto release;

    }

    ALOGV("createTrack\_l() output %d afLatency %u", output, mAfLatency);

     // 获取硬件支持的frame count

    status = AudioSystem::getFrameCount(output, &mAfFrameCount);

    if (status != NO\_ERROR) {

        ALOGE("getFrameCount(output=%d) status %d", output, status);

        goto release;

    }

    // 获取硬件支持的采样率

    status = AudioSystem::getSamplingRate(output, &mAfSampleRate);

    if (status != NO\_ERROR) {

        ALOGE("getSamplingRate(output=%d) status %d", output, status);

        goto release;

    }

    if (mSampleRate == 0) {

        mSampleRate = mAfSampleRate;

        mOriginalSampleRate = mAfSampleRate;

    }

    // MODE\_STREAM模式使用的是AUDIO\_OUTPUT\_FLAG\_NONE和TRANSFER\_SYNC

    if ((mFlags & AUDIO\_OUTPUT\_FLAG\_FAST) && !((

            // either of these use cases:

            // use case 1: shared buffer

            (mSharedBuffer != 0) ||

            // use case 2: callback transfer mode

            (mTransfer == TRANSFER\_CALLBACK) ||

            // use case 3: obtain/release mode

            (mTransfer == TRANSFER\_OBTAIN)) &&

            // matching sample rate

            (mSampleRate == mAfSampleRate))) {

        ALOGW("AUDIO\_OUTPUT\_FLAG\_FAST denied by client; transfer %d, track %u Hz, output %u Hz",

                mTransfer, mSampleRate, mAfSampleRate);

        // once denied, do not request again if IAudioTrack is re-created

        mFlags = (audio\_output\_flags\_t) (mFlags & ~AUDIO\_OUTPUT\_FLAG\_FAST);

}

...................

    size\_t temp = frameCount;   // temp may be replaced by a revised value of frameCount,

                                // but we will still need the original value also

int originalSessionId = mSessionId;

// 通过IPC调用AF中的createTrack方法

    sp<IAudioTrack> track = audioFlinger->createTrack(streamType,

                                                      mSampleRate,

                                                      mFormat,

                                                      mChannelMask,

                                                      &temp,

                                                      &trackFlags,

                                                      mSharedBuffer,

                                                      output,

                                                      tid,

                                                      &mSessionId,

                                                      mClientUid,

                                                      &status);

    ALOGE\_IF(originalSessionId != AUDIO\_SESSION\_ALLOCATE && mSessionId != originalSessionId,

            "session ID changed from %d to %d", originalSessionId, mSessionId);

    if (status != NO\_ERROR) {

        ALOGE("AudioFlinger could not create track, status: %d", status);

        goto release;

    }

    ALOG\_ASSERT(track != 0);

    // AF在createTrack时，创建了一块共享内存，以及这块内存的控制对象

    sp<IMemory> iMem = track->getCblk();

    if (iMem == 0) {

        ALOGE("Could not get control block");

        return NO\_INIT;

    }

    void \*iMemPointer = iMem->pointer();

    if (iMemPointer == NULL) {

        ALOGE("Could not get control block pointer");

        return NO\_INIT;

    }

    // invariant that mAudioTrack != 0 is true only after set() returns successfully

    if (mAudioTrack != 0) {

        IInterface::asBinder(mAudioTrack)->unlinkToDeath(mDeathNotifier, this);

        mDeathNotifier.clear();

}

// 将这个IAudioTrack对象保存为一个类变量

    mAudioTrack = track;

    mCblkMemory = iMem;

    IPCThreadState::self()->flushCommands();

    // AudioTrack端也需要通过这个控制对象来访问共享内存

    audio\_track\_cblk\_t\* cblk = static\_cast<audio\_track\_cblk\_t\*>(iMemPointer);

    mCblk = cblk;

    // note that temp is the (possibly revised) value of frameCount

    if (temp < frameCount || (frameCount == 0 && temp == 0)) {

        // In current design, AudioTrack client checks and ensures frame count validity before

        // passing it to AudioFlinger so AudioFlinger should not return a different value except

        // for fast track as it uses a special method of assigning frame count.

        ALOGW("Requested frameCount %zu but received frameCount %zu", frameCount, temp);

    }

    frameCount = temp;

    .......................

    // reset server position to 0 as we have new cblk.

    mServer = 0;

    // update proxy

if (mSharedBuffer == 0) {

        // Proxy对象是用于AudioTrack端获取、释放、管理共享内存的对象，在之后分析AF的时候再讲解

        // MODE\_STREAM模式下创建的是AudioTrackClientProxy类对象

        mStaticProxy.clear();

        mProxy = new AudioTrackClientProxy(cblk, buffers, frameCount, mFrameSize);

    } else {

        mStaticProxy = new StaticAudioTrackClientProxy(cblk, buffers, frameCount, mFrameSize);

        mProxy = mStaticProxy;

    }

    // 设置左右声道的音量

    mProxy->setVolumeLR(gain\_minifloat\_pack(

            gain\_from\_float(mVolume[AUDIO\_INTERLEAVE\_LEFT]),

            gain\_from\_float(mVolume[AUDIO\_INTERLEAVE\_RIGHT])));

   .................

release:

    AudioSystem::releaseOutput(output, streamType, (audio\_session\_t)mSessionId);

    if (status == NO\_ERROR) {

        status = NO\_INIT;

    }

    return status;

}

在set方法中，AudioTrack Native通过IBinder访问了AudioFlinger服务，通过该服务提供的createTrack方法得到了一个IAudioTrack对象，这个IAudioTrack对象是个什么东西呢，在接下来讲解AudioFlinger的时候再分析。AudioTrack Native将这个对象的指针保存在mAudioTrack中，AudioTrack Native的很多方法，都是对这个IAudioTrack对象提供的方法进行的包装。

除此之外，AudioTrack还通过IAudioTrack获取了一个audio\_track\_cblk\_t指针，这个对象用于控制共享内存，但是这个对象是由谁创建的呢？AudioTrack所在的进程和AudioFlinger是两个不同的进程，他们是如何同时访问这同一个对象的呢？这个也留到之后再讲。

mSharedBuffer == 0代表当前模式是MODE\_STREAM，在这个模式下，由创建了一个AudioTrackClientProxy对象，这个对象包装了audio\_track\_cblk\_t对象，是用于在AudioTrack所在的进程中，为AudioTrack的write操作获取、释放内存缓冲区，这个类还有几个派生自同一个父类的兄弟类，用于在不同情况下，在不同进程中获取、释放内存缓冲区，这些类的实现也留到之后再讲。

我们还注意到，有一个名为AudioSystem::getOutputForAttr的Native方法被调用，这个方法又是用来做什么的呢？它的参数中有一个audio\_attributes\_t对象，是JAVA层传入的AudioAttribute对象转换成的native对象，其中有这个音频对应的usage和content type等属性。getOutputForAttr这个方法在AudioSystem中定义，但是实际上是通过IPC调用AudioPolicy服务中的同名方法来实现的，这个实现留到分析AudioPolicy的时候再讲。这里我们只需要知道，这个方法是通过音频属性、当前终端支持的硬件，和当前系统的状态，来为这次音频播放指定具体的输出流，声音将通过哪个具体的音频设备发出。这个接口返回了一个audio\_io\_handle\_t，是底层对应的输出流的唯一标识。

总结一下，AudioTrack在初始化的时候做了些什么事情呢？

AudioTrack JAVA通过JNI封装了对AudioTrack Native的调用，通过IPC访问AudioFlinger服务，获取了一个IAudioTrack，又通过IPC访问AudioPolicy，为音频播放事件指定了一个具体的输出流并返回其唯一标识，还通过IAudioTrack获得了共享内存的操作对象，把这些步骤体现在序列图中：

  这样，AudioTrack创建时做了些什么事情就一目了然了。说白了现在也看不出来它到底做了什么事情，更核心的事务是由AudioPolicy和AudioFlinger来实现的，这也的确是Android的一种很常见的功能实现方式，应用程序没有权限去访问一些具体的硬件设备，系统就通过IBinder服务，提供了一些接口给应用层使用，而核心的功能，都是在系统的一些拥有对应权限的native服务中实现。要对AudioTrack播放音频的逻辑进行深入分析，必须得分析到这两大服务中。

### 2.2.3.    AudioFlinger

前面分析AudioTrack的时候，我们知道这个对象通过IPC访问了AudioFlinger，获取了一个IAudioTrack对象，而几乎所有AudioTrack的方法，都是对这个对象的方法的封装。

从Android IBinder的框架可知，应用层持有的IBinder对象，其实都是一个Proxy端，例如这个IAudioTrack对象就是一个BpAudioTrack对象，仅仅是一个轻量级的对象，通过IBinder驱动，作为工作在AudioFlinger所在的服务中的一个Bn端对象在应用进程中的一个代理，接下来我们就要分析这个Bn端到底是哪个对象。

AudioFlinger在源代码中所在的位置为frameworks/av/servies/audioflinger文件夹。

我们来看AudioFlinger中与IAudioTrack相关的一些类：

TrackHandle类继承自BnAudioTrack，是刚才我们提到的，工作在AudioFlinger所在的mediaserver进程中的Bn端对象，AudioTrack通过BnAudioTrack与之进行交互，跨进程调用其接口。但是我们看TrackHandle的实现，发现它很简单：

AudioFlinger::TrackHandle::TrackHandle(const sp<AudioFlinger::PlaybackThread::Track>& track)

    : BnAudioTrack(),

      mTrack(track)

{

}

AudioFlinger::TrackHandle::~TrackHandle() {

    mTrack->destroy();

}

sp<IMemory> AudioFlinger::TrackHandle::getCblk() const {

    return mTrack->getCblk();

}

status\_t AudioFlinger::TrackHandle::start() {

    return mTrack->start();

}

void AudioFlinger::TrackHandle::stop() {

    mTrack->stop();

}

void AudioFlinger::TrackHandle::flush() {

    mTrack->flush();

}

void AudioFlinger::TrackHandle::pause() {

    mTrack->pause();

}

....................

TrackHandle也是个轻量级的类对象，封装了对Track类的调用。

Track派生自TrackBase，TrackBase又派生自AudioBufferProvider类，从这个类的名字可以看出，是一种“音频数据提供者”，为声音的播放提供数据，这个名字，是否表示AudioFlinger将通过这个“数据提供者”获取到应用端写入AudioTrack的数据呢？答案是肯定的，AudioMixer这个类，在进行混音操作的时候，会将Track作为“数据提供者”，从Track中拉取数据，参与混音，不过这是如何实现的，留到之后讲解共享内存的时候再说。

之前我们保留的一个问题，在AudioTrack初始化的时候，AudioFlinger到底建立了哪些对象，现在就有答案了：

对于每一个AudioTrack，AudioFlinger都为止创建了一个Track对象，并用TrackHandle对象对它进行包装，通过IPC提供给AudioTrack访问。

我们来看看AudioFlinger::createTrack方法，看看Track和TrackHandle是怎么创建的：

sp<IAudioTrack> AudioFlinger::createTrack(

        audio\_stream\_type\_t streamType,

        uint32\_t sampleRate,

        audio\_format\_t format,

        audio\_channel\_mask\_t channelMask,

        size\_t \*frameCount,

        IAudioFlinger::track\_flags\_t \*flags,

        const sp<IMemory>& sharedBuffer,

        audio\_io\_handle\_t output,

        pid\_t tid,

        int \*sessionId,

        int clientUid,

        status\_t \*status)

{

    sp<PlaybackThread::Track> track;

    sp<TrackHandle> trackHandle;

    sp<Client> client;

    status\_t lStatus;

    int lSessionId;

    // 首先会进行一些参数的有效性判断

    // client AudioTrack::set already implements AUDIO\_STREAM\_DEFAULT => AUDIO\_STREAM\_MUSIC,

    // but if someone uses binder directly they could bypass that and cause us to crash

    if (uint32\_t(streamType) >= AUDIO\_STREAM\_CNT) {

        ALOGE("createTrack() invalid stream type %d", streamType);

        lStatus = BAD\_VALUE;

        goto Exit;

    }

    // further sample rate checks are performed by createTrack\_l() depending on the thread type

    if (sampleRate == 0) {

        ALOGE("createTrack() invalid sample rate %u", sampleRate);

        lStatus = BAD\_VALUE;

        goto Exit;

    }

    // further channel mask checks are performed by createTrack\_l() depending on the thread type

    if (!audio\_is\_output\_channel(channelMask)) {

        ALOGE("createTrack() invalid channel mask %#x", channelMask);

        lStatus = BAD\_VALUE;

        goto Exit;

    }

    // further format checks are performed by createTrack\_l() depending on the thread type

    if (!audio\_is\_valid\_format(format)) {

        ALOGE("createTrack() invalid format %#x", format);

        lStatus = BAD\_VALUE;

        goto Exit;

    }

if (sharedBuffer != 0 && sharedBuffer->pointer() == NULL) {

   // 对MODE\_STATIC类型进行数据缓冲的有效性判断

        ALOGE("createTrack() sharedBuffer is non-0 but has NULL pointer()");

        lStatus = BAD\_VALUE;

        goto Exit;

    }

    {

        Mutex::Autolock \_l(mLock);

  // 通过output唯一标识，获取对应的PlaybackThread 线程对象

        PlaybackThread \*thread = checkPlaybackThread\_l(output);

        if (thread == NULL) {

            ALOGE("no playback thread found for output handle %d", output);

            lStatus = BAD\_VALUE;

            goto Exit;

        }

        pid\_t pid = IPCThreadState::self()->getCallingPid();

   // 获取Client对象

        client = registerPid(pid);

        PlaybackThread \*effectThread = NULL;

        if (sessionId != NULL && \*sessionId != AUDIO\_SESSION\_ALLOCATE) {

            lSessionId = \*sessionId;

            // check if an effect chain with the same session ID is present on another

            // output thread and move it here.

            for (size\_t i = 0; i < mPlaybackThreads.size(); i++) {

                sp<PlaybackThread> t = mPlaybackThreads.valueAt(i);

                if (mPlaybackThreads.keyAt(i) != output) {

                    uint32\_t sessions = t->hasAudioSession(lSessionId);

                    if (sessions & PlaybackThread::EFFECT\_SESSION) {

                        effectThread = t.get();

                        break;

                    }

                }

            }

        } else {

            // if no audio session id is provided, create one here

            lSessionId = nextUniqueId();

            if (sessionId != NULL) {

                \*sessionId = lSessionId;

            }

        }

        ALOGV("createTrack() lSessionId: %d", lSessionId);

  // Track实际上是由PlaybackThread 类来创建的

        track = thread->createTrack\_l(client, streamType, sampleRate, format,

                channelMask, frameCount, sharedBuffer, lSessionId, flags, tid, clientUid, &lStatus);

        LOG\_ALWAYS\_FATAL\_IF((lStatus == NO\_ERROR) && (track == 0));

        // we don't abort yet if lStatus != NO\_ERROR; there is still work to be done regardless

        // move effect chain to this output thread if an effect on same session was waiting

        // for a track to be created

        if (lStatus == NO\_ERROR && effectThread != NULL) {

            // no risk of deadlock because AudioFlinger::mLock is held

            Mutex::Autolock \_dl(thread->mLock);

            Mutex::Autolock \_sl(effectThread->mLock);

            moveEffectChain\_l(lSessionId, effectThread, thread, true);

        }

        // Look for sync events awaiting for a session to be used.

        for (size\_t i = 0; i < mPendingSyncEvents.size(); i++) {

            if (mPendingSyncEvents[i]->triggerSession() == lSessionId) {

                if (thread->isValidSyncEvent(mPendingSyncEvents[i])) {

                    if (lStatus == NO\_ERROR) {

                        (void) track->setSyncEvent(mPendingSyncEvents[i]);

                    } else {

                        mPendingSyncEvents[i]->cancel();

                    }

                    mPendingSyncEvents.removeAt(i);

                    i--;

                }

            }

        }

        setAudioHwSyncForSession\_l(thread, (audio\_session\_t)lSessionId);

    }

    if (lStatus != NO\_ERROR) {

        // remove local strong reference to Client before deleting the Track so that the

        // Client destructor is called by the TrackBase destructor with mClientLock held

        // Don't hold mClientLock when releasing the reference on the track as the

        // destructor will acquire it.

        {

            Mutex::Autolock \_cl(mClientLock);

            client.clear();

        }

        track.clear();

        goto Exit;

    }

    // Track对象创建成功过后，创建TrackHandle对象，对Track进行包装，通过IPC可以将代理返回给AT端

    // return handle to client

    trackHandle = new TrackHandle(track);

Exit:

    \*status = lStatus;

    return trackHandle;

}

在代码中我们可以看到，AF通过checkPlaybackThread\_l函数，来获取了一个PlaybackThread类对象。这个对象是个线程，从名字可以看出，它是负责播放声音的工作线程，Track对象是通过其createTrack方法来获得的。

PlaybackThread类有几个子类，如下面这个类图所示：

PlaybackThread是回放线程，在这个线程运行的过程中，循环地从每个处于活动状态的Track中拉取数据并进行处理，最终把数据通过AudioHAL提供的接口，写入到具体的硬件设备中，这样声音就得以播放。

MixerThread，混音线程，也是音频播放的常用线程，将这个线程维护的所有活动状态的Track中的音频数据拉取出来，进行混音操作后，再输出到音频设备中。

DirectOutputThread：直接输出线程，它会选择一路音频流后将数据直接输出，由于没有混音的操作，这样可以减少很多延时。

DuplicatingThread：多路输出线程，它从MixerThread派生，意味着它也能够混音。它最终会把混音后的数据写到多个输出中，也就是一份数据会有多个接收者。这就是Duplicate的含义。目前在蓝牙A2DP设备输出中使用。

下面是checkPlaybackThread\_l函数的实现：

AudioFlinger::PlaybackThread \*AudioFlinger::checkPlaybackThread\_l(audio\_io\_handle\_t output) const

{

    return mPlaybackThreads.valueFor(output).get();

}

我们发现它通过output从mPlaybackThreads数组中获取了一个已经存在的工作线程对象，这些工作线程是在哪里创建的呢？在AudioFlinger.cpp中发现了一个方法openOutput\_l：

sp<AudioFlinger::PlaybackThread> AudioFlinger::openOutput\_l(audio\_module\_handle\_t module,

                                                            audio\_io\_handle\_t \*output,

                                                            audio\_config\_t \*config,

                                                            audio\_devices\_t devices,

                                                            const String8& address,

                                                            audio\_output\_flags\_t flags)

{

    // 获取与这次音频播放配套的HAL接口

    AudioHwDevice \*outHwDev = findSuitableHwDev\_l(module, devices);

    if (outHwDev == NULL) {

        return 0;

    }

    audio\_hw\_device\_t \*hwDevHal = outHwDev->hwDevice();

    .............

// 获取一个AudioStreamOut 对象，它是音频输出设备的抽象，同时通过出参传出一个audio\_io\_handle\_t 指针

// 作为这个输出设备的唯一标识

    AudioStreamOut \*outputStream = NULL;

    status\_t status = outHwDev->openOutputStream(

            &outputStream,

            \*output,

            devices,

            flags,

            config,

            address.string());

    mHardwareStatus = AUDIO\_HW\_IDLE;

    if (status == NO\_ERROR) {

        // 按照flag来创建对应的工作线程对象

        PlaybackThread \*thread;

        if (flags & AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD) {

            thread = new OffloadThread(this, outputStream, \*output, devices, mSystemReady);

            ALOGV("openOutput\_l() created offload output: ID %d thread %p", \*output, thread);

        } else if ((flags & AUDIO\_OUTPUT\_FLAG\_DIRECT)

                || !isValidPcmSinkFormat(config->format)

                || !isValidPcmSinkChannelMask(config->channel\_mask)) {

            thread = new DirectOutputThread(this, outputStream, \*output, devices, mSystemReady);

            ALOGV("openOutput\_l() created direct output: ID %d thread %p", \*output, thread);

        } else {

      // 混音线程对象

            thread = new MixerThread(this, outputStream, \*output, devices, mSystemReady);

            ALOGV("openOutput\_l() created mixer output: ID %d thread %p", \*output, thread);

        }

        // 创建完毕后，将此线程对象加入mPlaybackThreads数组，并将audio\_io\_handle\_t 指针作为它的索引

        mPlaybackThreads.add(\*output, thread);

        return thread;

    }

    return 0;

}

这个函数是什么时候调用的呢？还记得AudioSystem::getOutputForAttr么，也许这个方函数就是在通过AudioAttribute获取output的时候创建的，是这样么？在之后分析AudioPolicy的时候再讨论。

工作线程对象是怎么创建Track对象的呢？接下来看看PlaybackThread::createTrack\_l：

sp<AudioFlinger::PlaybackThread::Track> AudioFlinger::PlaybackThread::createTrack\_l(

        .............)

{

    size\_t frameCount = \*pFrameCount;

sp<Track> track;

......................

    lStatus = initCheck();

    if (lStatus != NO\_ERROR) {

        ALOGE("createTrack\_l() audio driver not initialized");

        goto Exit;

    }

    { // scope for mLock

        Mutex::Autolock \_l(mLock);

..........................

        if (!isTimed) {

            track = new Track(this, client, streamType, sampleRate, format,

                              channelMask, frameCount, NULL, sharedBuffer,

                              sessionId, uid, \*flags, TrackBase::TYPE\_DEFAULT);

        } else {

            track = TimedTrack::create(this, client, streamType, sampleRate, format,

                    channelMask, frameCount, sharedBuffer, sessionId, uid);

        }

.....................

// 新建立的Track对象，加入mTracks数组

mTracks.add(track);

......................

    }

    lStatus = NO\_ERROR;

Exit:

    \*status = lStatus;

    return track;

}

如代码所示，createTrack\_l创建了一个Track对象，并保存到mTracks数组中。

经过上面的一系列处理，AudioFlinger为AudioTrack指定了一个工作线程对象，创建了一个Track对象并保存在工作线程的mTracks数组中，并使用一个TrackHandle对象包装了这个Track对象，并将其Bp端返回给AudioTrack端，这样，AudioTrack初始化时的工作就基本上完成了。总结一下，这个流程可以由以下图来表示：

### 2.2.4.    AudioPolicy

在AudioTrack初始化的过程中，Audio Policy Service扮演了一个什么角色呢？在之前我们提到，AudioTrack::createTrack\_l函数的执行过程中，调用了AudioSystem::getOutputForAttr函数，这个函数实际上是由Audio Policy来实现的，现在我们来分析一下这个函数：

status\_t AudioSystem::getOutputForAttr(const audio\_attributes\_t \*attr,

                                        audio\_io\_handle\_t \*output,

                                        audio\_session\_t session,

                                        audio\_stream\_type\_t \*stream,

                                        uid\_t uid,

                                        uint32\_t samplingRate,

                                        audio\_format\_t format,

                                        audio\_channel\_mask\_t channelMask,

                                        audio\_output\_flags\_t flags,

                                        audio\_port\_handle\_t selectedDeviceId,

                                        const audio\_offload\_info\_t \*offloadInfo)

{

    const sp<IAudioPolicyService>& aps = AudioSystem::get\_audio\_policy\_service();

    if (aps == 0) return NO\_INIT;

    return aps->getOutputForAttr(attr, output, session, stream, uid,

                                 samplingRate, format, channelMask,

                                 flags, selectedDeviceId, offloadInfo);

}

这个函数第一句就是向ServiceManager获取IAudioPolicyService的代理对象，然后调用了它的getOutputForAttr函数。

从这个函数的名字可以看出，它的意图是通过创建AudioTrack时传入的AudioAttribute参数（包括音频的usage和content type等信息），来为这次音频播放事件指定一个Output，返回这个Output的唯一标识（audio\_io\_handle\_t ），之后就可以通过这个唯一标识对Output进行一系列操作。

Audio Policy Service在定位这个Output的过程中，做了一些什么判断，创建了一些什么对象，与AudioFlinger之间有什么交互呢？这需要我们分析一下getOutputForAttr函数。

在这之前，先看看Audio Policy Service这个模块中有一些什么对象：

从图中可以看出，AudioPolicyService类继承自BnAudioPolicyService，是实际完成这个IPC调用的类，但它依然是个轻量级的包装类，所有工作都是由AudioPolicyInterface的子类，AudioPolicyManager类来完成的，所以我们来看看AudioPolicyManager::getOutputForAttr：

status\_t AudioPolicyManager::getOutputForAttr(const audio\_attributes\_t \*attr,

                                              audio\_io\_handle\_t \*output,

                                              audio\_session\_t session,

                                              audio\_stream\_type\_t \*stream,

                                              uid\_t uid,

                                              uint32\_t samplingRate,

                                              audio\_format\_t format,

                                              audio\_channel\_mask\_t channelMask,

                                              audio\_output\_flags\_t flags,

                                              audio\_port\_handle\_t selectedDeviceId,

                                              const audio\_offload\_info\_t \*offloadInfo)

{

    .................

    // 1、通过Audio Attribute来获取stream Type

    \*stream = streamTypefromAttributesInt(&attributes);

    ...............

    // 2、通过Audio Attribute来获取routing strategy

routing\_strategy strategy = (routing\_strategy) getStrategyForAttr(&attributes);

// 3、通过routing strategy来指定音频数据需要写入的设备

    audio\_devices\_t device = getDeviceForStrategy(strategy, false /\*fromCache\*/);

    if ((attributes.flags & AUDIO\_FLAG\_HW\_AV\_SYNC) != 0) {

        flags = (audio\_output\_flags\_t)(flags | AUDIO\_OUTPUT\_FLAG\_HW\_AV\_SYNC);

    }

    ALOGV("getOutputForAttr() device 0x%x, samplingRate %d, format %x, channelMask %x, flags %x",

          device, samplingRate, format, channelMask, flags);

// 4、通过音频设备类型，以及stream类型，获取对应的Output

    \*output = getOutputForDevice(device, session, \*stream,

                                 samplingRate, format, channelMask,

                                 flags, offloadInfo);

    if (\*output == AUDIO\_IO\_HANDLE\_NONE) {

        mOutputRoutes.removeRoute(session);

        return INVALID\_OPERATION;

    }

    return NO\_ERROR;

}

获取Output的过程中，AudioPolicyManager做了这么几件事

A)通过usage、content type等音频属性，获取对应的stream Type

B)通过usage、content type等音频属性，获取对应的routing strategy

C)通过routing strategy来获取audio\_devices\_t，也就是音频数据需要写入的具体的音频设备

D)已知输出的音频设备类型、当前音频播放事件的stream type，获取一个Output

这个流程中出现了几个概念需要讲解一下

l  routing\_strategy

表示音频播放的路由策略的分类，定义在audiopolicy模块的RoutingStrategy.h中，比如STRATEGY\_MEDIA,播放音乐时的策略；比如STRATEGY\_PHONE，电话通话音的策略；STRATEGY\_SONIFICATION，通知音的策略......

一个音频播放的事件需要选择哪种路由策略，是由音频数据的用途、内容，以及当前系统的audio mode等属性来决定的。

l  audio\_devices\_t

定义在system/media/audio/include/system/audio.h中，列举了Android终端所能够支持的所有的音频输入、输出硬件类型，例如：

AUDIO\_DEVICE\_OUT\_EARPIECE ： 机身听筒（输出）

AUDIO\_DEVICE\_OUT\_SPEAKER ：机身扬声器（输出）

AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET ：有线耳机（输出）

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO ： 蓝牙HFP输出

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP ： 蓝牙A2DP媒体输出

AUDIO\_DEVICE\_OUT\_TELEPHONY\_TX ：电话通话音输出

AUDIO\_DEVICE\_IN\_BUILTIN\_MIC ： 机身麦克风（输入）

AUDIO\_DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET ： 蓝牙HFP输入

AUDIO\_DEVICE\_IN\_REMOTE\_SUBMIX ： 远程声音（比如wifi display）

..........

一些设备类型，比如AUDIO\_DEVICE\_OUT\_ALL\_A2DP，是几个同类型设备进行二进制“|”操作得到的结果：

AUDIO\_DEVICE\_OUT\_ALL\_A2DP = (AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP |

                                 AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |

                                 AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER)

Output只是一个抽象的概念，代表音频输出流，这个输出流必须得绑定到一个具体的硬件设备，通过驱动将音频数据写入硬件的缓冲区，才能进行播放。从audio.h定义的所有设备看来，Android系统所支持的输入、输出音频设备非常多，Output是如何知道该向哪个物理设备写数据的呢？

而且，并非每个Android终端都有那么多物理设备，这个和终端厂商的硬件设计有关系，Android的音频系统怎么知道一个终端支持哪些物理设备呢？

接下来就来分析这个问题。

Android Policy这个模块有一个配置文件，需要厂商跟进不同的硬件版本来进行配置，就是Android终端系统分区中的audio\_policy.conf文件，路径为/system/etc/audio\_policy.conf。

我们以LG的N5为例，看看它的配置文件：

# 定义了当前终端的全局音频配置

global\_configuration {

  # 支持哪些输出设备

  attached\_output\_devices AUDIO\_DEVICE\_OUT\_EARPIECE|AUDIO\_DEVICE\_OUT\_SPEAKER|AUDIO\_DEVICE\_OUT\_TELEPHONY\_TX

  # 默认输出设备是扬声器

  default\_output\_device AUDIO\_DEVICE\_OUT\_SPEAKER

  # 支持哪些输入设备

  attached\_input\_devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC|AUDIO\_DEVICE\_IN\_BACK\_MIC|AUDIO\_DEVICE\_IN\_REMOTE\_SUBMIX|AUDIO\_DEVICE\_IN\_TELEPHONY\_RX

  speaker\_drc\_enabled TRUE

}

#定义了当前终端的所有音频硬件模块

audio\_hw\_modules {

  # 定义了一个名为primary 的HwModule，是当前终端的主音频模块

  primary {

    # 以下定义这个HwModule的所有输出设备，保存在mOutputProfiles中

outputs {

  # 名为primary 的IOProfile，以及它的属性，它是个Source

      primary {

        sampling\_rates 44100|48000

        channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        # 这个IOProfile的声音数据支持用哪些设备输出

        devices AUDIO\_DEVICE\_OUT\_EARPIECE|AUDIO\_DEVICE\_OUT\_SPEAKER|AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET|AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE|AUDIO\_DEVICE\_OUT\_ALL\_SCO|AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL

  # 这个HwModule的默认Source

        flags AUDIO\_OUTPUT\_FLAG\_PRIMARY

      }

  # 名为deep\_buffer 的IOProfile，以及它的属性

      deep\_buffer {

         sampling\_rates 8000|11025|12000|16000|22050|24000|32000|44100|48000

         channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

         formats AUDIO\_FORMAT\_PCM\_16\_BIT

         devices AUDIO\_DEVICE\_OUT\_SPEAKER|AUDIO\_DEVICE\_OUT\_EARPIECE|AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET|AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE|AUDIO\_DEVICE\_OUT\_ALL\_SCO|AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL

         flags AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER

      }

...........

 # 名为voice\_tx  的IOProfile，以及它的属性，用于电话输入

      voice\_tx {

        sampling\_rates 8000|16000|48000

        channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO|AUDIO\_CHANNEL\_OUT\_MONO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_OUT\_TELEPHONY\_TX

      }

}

# 以下定义这个HwModule的所有输入设备，保存在mInputProfiles中

inputs {

  # 名为primary 的IOProfile，以及它的属性，它是个Sink

      primary {

        sampling\_rates 8000|11025|12000|16000|22050|24000|32000|44100|48000

        channel\_masks AUDIO\_CHANNEL\_IN\_MONO|AUDIO\_CHANNEL\_IN\_STEREO|AUDIO\_CHANNEL\_IN\_FRONT\_BACK

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC|AUDIO\_DEVICE\_IN\_WIRED\_HEADSET|AUDIO\_DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET|AUDIO\_DEVICE\_IN\_VOICE\_CALL|AUDIO\_DEVICE\_IN\_BACK\_MIC

      }

      voice\_rx {

        sampling\_rates 8000|16000|48000

        channel\_masks AUDIO\_CHANNEL\_IN\_STEREO|AUDIO\_CHANNEL\_IN\_MONO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_IN\_TELEPHONY\_RX

      }

    }

  }

  # 定义了一个名为a2dp的HwModule，用于蓝牙耳机

  a2dp {

    outputs {

      a2dp {

        sampling\_rates 44100

        channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_OUT\_ALL\_A2DP

      }

    }

    inputs {

      a2dp {

        sampling\_rates 44100|48000

        channel\_masks AUDIO\_CHANNEL\_IN\_MONO|AUDIO\_CHANNEL\_IN\_STEREO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_IN\_BLUETOOTH\_A2DP

      }

    }

  }

  # 定义了一个名为usb的HwModule，用于usb声卡

  usb {

    outputs {

      usb\_accessory {

        sampling\_rates 44100

        channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY

      }

      usb\_device {

        sampling\_rates dynamic

        channel\_masks dynamic

        formats dynamic

        devices AUDIO\_DEVICE\_OUT\_USB\_DEVICE

      }

    }

    inputs {

      usb\_device {

        sampling\_rates dynamic

        channel\_masks dynamic

        formats dynamic

        devices AUDIO\_DEVICE\_IN\_USB\_DEVICE

      }

    }

  }

  ............

}

 AudioPolicyManager类在初始化的时候对audio\_policy.conf进行解析，创建了一些对象，可以结合前文的Audio Policy模块的类图进行分析：

l  AudioPort

Audio port是Android Framework内部，以及Audio HAL中使用的一个概念，Audio Port可以充当Source和Sink两种角色，前者是音频数据的生产者，后者是音频数据的消费者；Audio Port有3种类型：

n  AUDIO\_PORT\_TYPE\_SESSION：比如一个AudioTrack

n  AUDIO\_PORT\_TYPE\_DEVICE：一个具体的设备，比如AUDIO\_DEVICE\_OUT\_SPEAKER

n  AUDIO\_PORT\_TYPE\_MIX：一个混音器，比如一个PlaybackThread output

l  HwModule

定义了一个Android终端支持的音频硬件模块，例如N5的配置文件中，就定义了primary、A2DP、usb三个HwModule。Primary用于使用机身音频设备、蓝牙耳机HFP、有线耳机等设备进行媒体播放、录制以及电话通话等；A2DP用于使用蓝牙耳机A2DP进行媒体播放和录制；usb用于USB声卡。

AudioPolicyManager类将解析出来的所有HwModule对象存储在mHwModules集合中。

l  IOProfile

每个HwModule都有一个或者多个IO，每个IO都有自己的属性，AudioPolicy定义了IOProfile类来描述每个IO的属性，比如角色(source还是sink，前者是音频数据的生产者，后者是消费者）、采样率、声道数、采样类型、以及支持采用哪些音频设备进行播放（如果是source）、或者支持用哪些音频设备进行声音采集（如果是sink）。HwModule将所有的IOProfile保存在mOutputProfiles和mInputProfiles两个矢量中。

N5就定义了多个IOProfile，例如一个名为primary的用于音频播放的IOProfile：

primary {

        sampling\_rates 44100|48000

        channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

        formats AUDIO\_FORMAT\_PCM\_16\_BIT

        devices AUDIO\_DEVICE\_OUT\_EARPIECE|AUDIO\_DEVICE\_OUT\_SPEAKER|AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET|AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE|AUDIO\_DEVICE\_OUT\_ALL\_SCO|AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL

        flags AUDIO\_OUTPUT\_FLAG\_PRIMARY

      }

看看它的devices节点，这个IOProfile允许连到听筒、扬声器、有线耳机、头戴式耳机、蓝牙HFP设备、AUX设备等物理设备上播放。

l  DeviceDescriptor

Android\_policy.conf定义了当前系统的所有音频模块，以及每个模块的IOProfile，我们注意到，比如N5的primary这个音频模块，有一个名为primary的用于音频播放的IOProfile，它支持了N多种音频物理设备，但是这些设备，终端并不一定都支持，还有为何每次播放音乐的时候，都是从扬声器发出的声音？这说明总有个默认设备吧？

其实，在配置文件中也对这些属性进行了定义，比如N5就有个全局音频配置

 定义了当前终端的全局音频配置

global\_configuration {

  # 支持哪些输出设备

  attached\_output\_devices AUDIO\_DEVICE\_OUT\_EARPIECE|AUDIO\_DEVICE\_OUT\_SPEAKER|AUDIO\_DEVICE\_OUT\_TELEPHONY\_TX

  # 默认输出设备是扬声器

  default\_output\_device AUDIO\_DEVICE\_OUT\_SPEAKER

  # 支持哪些输入设备

  attached\_input\_devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC|AUDIO\_DEVICE\_IN\_BACK\_MIC|AUDIO\_DEVICE\_IN\_REMOTE\_SUBMIX|AUDIO\_DEVICE\_IN\_TELEPHONY\_RX

  speaker\_drc\_enabled TRUE

}

attached\_output\_devices 定义了裸机状态下的输出设备，有机身话筒、扬声器、电话发送端；attached\_input\_devices 定义了裸机状态下的所有输入设备，有前、后置麦克风、remote submix、电话接收端；default\_output\_device定义了默认输出设备，这里为扬声器。

每个输出、输入设备设备，都被AudioPolicyManager抽象为一个DeviceDescriptor，输出设备保存在mAvailableOutputDevices中，输入设备保存在mAvailableInputDevices。在AudioPolicy为音频播放、音频录制选择路由的时候，都会在mAvailableOutputDevices或者mAvailableInputDevices中进行遍历，看终端当前哪些设备处于有效状态，才能进一步建立到具体设备的路由。

确定了终端的音频能力之后，AudioPolicyManager开始为终端建立输入流和输出流。

AudioPolicyManager::AudioPolicyManager(AudioPolicyClientInterface \*clientInterface)

..............

{

   .................

    // 这里就是解析conf文件的时机

    mDefaultOutputDevice = new DeviceDescriptor(AUDIO\_DEVICE\_OUT\_SPEAKER);

    if (ConfigParsingUtils::loadAudioPolicyConfig(AUDIO\_POLICY\_VENDOR\_CONFIG\_FILE,

                 mHwModules, mAvailableInputDevices, mAvailableOutputDevices,

                 mDefaultOutputDevice, mSpeakerDrcEnabled) != NO\_ERROR) {

        if (ConfigParsingUtils::loadAudioPolicyConfig(AUDIO\_POLICY\_CONFIG\_FILE,

                                  mHwModules, mAvailableInputDevices, mAvailableOutputDevices,

                                  mDefaultOutputDevice, mSpeakerDrcEnabled) != NO\_ERROR) {

            ALOGE("could not load audio policy configuration file, setting defaults");

            defaultAudioPolicyConfig();

        }

    }

    audio\_devices\_t outputDeviceTypes = mAvailableOutputDevices.types();

    audio\_devices\_t inputDeviceTypes = mAvailableInputDevices.types() & ~AUDIO\_DEVICE\_BIT\_IN;

for (size\_t i = 0; i < mHwModules.size(); i++) {

    // 根据HwModule的名称来选择对应的HAL接口，加载不同的HAL动态库

        mHwModules[i]->mHandle = mpClientInterface->loadHwModule(mHwModules[i]->mName);

        if (mHwModules[i]->mHandle == 0) {

            ALOGW("could not open HW module %s", mHwModules[i]->mName);

            continue;

        }

   // 打开每个输出流，并为之选择输出设备

        for (size\_t j = 0; j < mHwModules[i]->mOutputProfiles.size(); j++)

        {

      // 遍历这个HwModule的输出端IOProfile

            const sp<IOProfile> outProfile = mHwModules[i]->mOutputProfiles[j];

            if (outProfile->mSupportedDevices.isEmpty()) {

                ALOGW("Output profile contains no device on module %s", mHwModules[i]->mName);

                continue;

            }

            // 跳过AUDIO\_OUTPUT\_FLAG\_DIRECT类型的IOProfile

            if ((outProfile->mFlags & AUDIO\_OUTPUT\_FLAG\_DIRECT) != 0) {

                continue;

            }

      // 选择这个IOProfile支持的，且在系统的mAvailableOutputDevices中的输出设备，attach到这个输出流

            audio\_devices\_t profileType = outProfile->mSupportedDevices.types();

            if ((profileType & mDefaultOutputDevice->type()) != AUDIO\_DEVICE\_NONE) {

                profileType = mDefaultOutputDevice->type();

            } else {

                // chose first device present in mSupportedDevices also part of

                // outputDeviceTypes

                for (size\_t k = 0; k  < outProfile->mSupportedDevices.size(); k++) {

                    profileType = outProfile->mSupportedDevices[k]->type();

                    if ((profileType & outputDeviceTypes) != 0) {

                        break;

                    }

                }

            }

            if ((profileType & outputDeviceTypes) == 0) {

          // 如果一个IOProfile所支持的所有设备，系统都不支持，则丢弃

                continue;

            }

      // 在AudioPolicy中，输出流用SwAudioOutputDescriptor来抽象；

      // 在这里创建了输出流对象并与对应的IOProfile绑定

            sp<SwAudioOutputDescriptor> outputDesc = new SwAudioOutputDescriptor(outProfile,

                                                                                 mpClientInterface);

      // 为这个输出流对象填充属性

            outputDesc->mDevice = profileType;

            audio\_config\_t config = AUDIO\_CONFIG\_INITIALIZER;

            config.sample\_rate = outputDesc->mSamplingRate;

            config.channel\_mask = outputDesc->mChannelMask;

            config.format = outputDesc->mFormat;

            audio\_io\_handle\_t output = AUDIO\_IO\_HANDLE\_NONE;

      // mpClientInterface其实就是AudioFlinger，调用了它的openOutput来打开这个输出流

            status\_t status = mpClientInterface->openOutput(outProfile->getModuleHandle(),

                                                            &output,

                                                            &config,

                                                            &outputDesc->mDevice,

                                                            String8(""),

                                                            &outputDesc->mLatency,

                                                            outputDesc->mFlags);

            if (status != NO\_ERROR) {

          // 如果这个输出流打不开，则丢弃

                ALOGW("Cannot open output stream for device %08x on hw module %s",

                      outputDesc->mDevice,

                      mHwModules[i]->mName);

            } else {

                outputDesc->mSamplingRate = config.sample\_rate;

                outputDesc->mChannelMask = config.channel\_mask;

                outputDesc->mFormat = config.format;

                for (size\_t k = 0; k  < outProfile->mSupportedDevices.size(); k++) {

                    audio\_devices\_t type = outProfile->mSupportedDevices[k]->type();

                    ssize\_t index =

                            mAvailableOutputDevices.indexOf(outProfile->mSupportedDevices[k]);

                    // give a valid ID to an attached device once confirmed it is reachable

                    if (index >= 0 && !mAvailableOutputDevices[index]->isAttached()) {

                        mAvailableOutputDevices[index]->attach(mHwModules[i]);

                    }

                }

                if (mPrimaryOutput == 0 &&

                        outProfile->mFlags & AUDIO\_OUTPUT\_FLAG\_PRIMARY) {

              // 定义系统的主输出流

                    mPrimaryOutput = outputDesc;

                }

                                    ALOGE("add output %08x -- %s -- %d",

                      outputDesc->mDevice, outProfile->mName.string(), output);

          // 输出流创建完毕，保存这个输出流对象

                addOutput(output, outputDesc);

          // 为这个输出流选择具体的输出设备

                setOutputDevice(outputDesc,

                                outputDesc->mDevice,

                                true);

            }

        }

  // 打开每个输入流，并为之选择具体的音频采集设备

        for (size\_t j = 0; j < mHwModules[i]->mInputProfiles.size(); j++)

        {

      .......................

        ｝

    }

    ................

}

AudioPolicyManager的初始化时机是AudioPolicyService初始化的时候，是在系统启动时，mediaserver进程被init进程拉起来的时候，所以，在系统启动的时候，AudioPolicy就为整个系统创建好了输入流、输出流，并为输入流输出流绑定了具体的音频设备。

AudioPolicyManager的mpClientInterface，其实就是AudioFlinger，代码中通过mpClientInterface指针调用了openOutput函数，在之前我们分析AudioFlinger的时候，分析过这个函数，这个函数调用了HAL的接口打开了一个输出流，并创建了一个工作线程，返回了一个audio\_io\_handle\_t作为输出流的唯一标识，在分析这个函数的时候我们预留了一个问题，这个函数是在什么时候创建的，现在我们知道答案了，在mediaserver这个后台服务进程启动的时候，AudioPolicy service初始化的时候，调用了这个函数。这样，audio\_io\_handle\_t是由HAL分配的句柄，在AudioPolicy中，标识了一个SwAudioOutputDescriptor输出流对象，在AudioFlinger中，标识了一个工作线程对象。

输出流创建成功过后，addOutput函数将这个对象保存在了mOutputs集合中。

接下来有个关键函数叫做setOutputDevice，从函数名称上来看，是为了建立这个输出流和具体的设备之间的音频路由，我们来看看代码：

uint32\_t AudioPolicyManager::setOutputDevice(const sp<AudioOutputDescriptor>& outputDesc,

                                             audio\_devices\_t device,

                                             bool force,

                                             int delayMs,

                                             audio\_patch\_handle\_t \*patchHandle,

                                             const char\* address)

{

    ALOGV("setOutputDevice() device %04x delayMs %d", device, delayMs);

    AudioParameter param;

    uint32\_t muteWaitMs;

    if (outputDesc->isDuplicated()) {

        muteWaitMs = setOutputDevice(outputDesc->subOutput1(), device, force, delayMs);

        muteWaitMs += setOutputDevice(outputDesc->subOutput2(), device, force, delayMs);

        return muteWaitMs;

    }

    // no need to proceed if new device is not AUDIO\_DEVICE\_NONE and not supported by current

    // output profile

    if ((device != AUDIO\_DEVICE\_NONE) &&

            ((device & outputDesc->supportedDevices()) == 0)) {

  // 如果输出流并不支持这个目标设备，返回

        return 0;

    }

    // filter devices according to output selected

    device = (audio\_devices\_t)(device & outputDesc->supportedDevices());

    audio\_devices\_t prevDevice = outputDesc->mDevice;

    ALOGV("setOutputDevice() prevDevice 0x%04x", prevDevice);

    if (device != AUDIO\_DEVICE\_NONE) {

        outputDesc->mDevice = device;

    }

    muteWaitMs = checkDeviceMuteStrategies(outputDesc, prevDevice, delayMs);

// 在哪些情况下不改变这个输出流的音频路由呢？要同时满足3个条件

// 1、目标设备为AUDIO\_DEVICE\_NONE类型或者新设置的设备与这个输出流之前的路由设备一致

// 2、force == false

// 3、这个输出流已经连接建立了一条有效的路由

    if ((device == AUDIO\_DEVICE\_NONE || device == prevDevice) &&

        !force &&

        outputDesc->mPatchHandle != 0) {

        ALOGV("setOutputDevice() setting same device 0x%04x or null device", device);

        return muteWaitMs;

    }

    ALOGV("setOutputDevice() changing device");

    // 开始为这个输出流设置路由

    if (device == AUDIO\_DEVICE\_NONE) {

        resetOutputDevice(outputDesc, delayMs, NULL);

} else {

  // address这个属性，可以用于区分不同的蓝牙设备，因为系统可能有很多个蓝牙设备，各个设备地址不一样

        DeviceVector deviceList = (address == NULL) ?

                mAvailableOutputDevices.getDevicesFromType(device)

                : mAvailableOutputDevices.getDevicesFromTypeAddr(device, String8(address));

   // 必须首先判断系统是否支持这种设备

        if (!deviceList.isEmpty()) {

            struct audio\_patch patch;

            outputDesc->toAudioPortConfig(&patch.sources[0]);

            patch.num\_sources = 1;

            patch.num\_sinks = 0;

            for (size\_t i = 0; i < deviceList.size() && i < AUDIO\_PATCH\_PORTS\_MAX; i++) {

                deviceList.itemAt(i)->toAudioPortConfig(&patch.sinks[i]);

                patch.num\_sinks++;

            }

            ssize\_t index;

            if (patchHandle && \*patchHandle != AUDIO\_PATCH\_HANDLE\_NONE) {

                index = mAudioPatches.indexOfKey(\*patchHandle);

            } else {

                index = mAudioPatches.indexOfKey(outputDesc->mPatchHandle);

            }

            sp< AudioPatch> patchDesc;

            audio\_patch\_handle\_t afPatchHandle = AUDIO\_PATCH\_HANDLE\_NONE;

            if (index >= 0) {

                patchDesc = mAudioPatches.valueAt(index);

                afPatchHandle = patchDesc->mAfPatchHandle;

            }

      // 建立音频路由

            status\_t status = mpClientInterface->createAudioPatch(&patch,

                                                                   &afPatchHandle,

                                                                   delayMs);

            ALOGV("setOutputDevice() createAudioPatch returned %d patchHandle %d"

                    "num\_sources %d num\_sinks %d",

                                       status, afPatchHandle, patch.num\_sources, patch.num\_sinks);

            if (status == NO\_ERROR) {

                if (index < 0) {

                    patchDesc = new AudioPatch(&patch, mUidCached);

                    addAudioPatch(patchDesc->mHandle, patchDesc);

                } else {

                    patchDesc->mPatch = patch;

                }

                patchDesc->mAfPatchHandle = afPatchHandle;

                patchDesc->mUid = mUidCached;

                if (patchHandle) {

                    \*patchHandle = patchDesc->mHandle;

                }

                outputDesc->mPatchHandle = patchDesc->mHandle;

                nextAudioPortGeneration();

                mpClientInterface->onAudioPatchListUpdate();

            }

        }

.................

    }

    // update stream volumes according to new device

    applyStreamVolumes(outputDesc, device, delayMs);

    return muteWaitMs;

}

如果输出流支持目标设备，且系统也支持目标设备，AudioPolicy便在输出流和设备之间建立一个音频路由，叫做Audio Patch。

以N5为例，终端的输出流最终达到了以下状态：

回过头来看看，在AudioTrack的创建过程中，AudioPolicy到底担当了一种什么样的角色？

AudioPolicy在初始化的时候，就根据终端的配置，创建好了对应的输出流和工作线程，并为它们分配了一个唯一标识，并将输出流和具体的音频输出设备建立了路由。

在AudioTrack创建的时候，需要通过具体的音频属性，来选择输出流，通过调用AudioPolicy提供的getOutputForAttr函数，在AudioPolicy中绑定到一个输出流，在AudioFlinger中，绑定到了一个工作线程，返回了一个output句柄，与输出流、工作线程进行一一对应，绑定这个输出流的流程为：

l  通过Audio Attribute，获取音频播放的stream Type

l  通过Audio Attribute，获取音频播放的routing strategy（路由策略）

l  通过路由策略，来获取音频播放的音频设备

l  遍历AudioPolicy的所有输出流，选择一个与当前设备建立了音频路由的输出流，返回其句柄

l  之后就可以通过这个句柄定位到工作线程，通过工作线程来创建Track对象

AudioPolicy要为这次音频播放事件定位该使用哪个输出设备的时候，除了会考察当前设置的路由策略以外，还和系统的“Force Use”策略、音频模式（audio mode)、当前系统的available output devices、外连设备（比如有线耳机、蓝牙HFP、A2DP等设备、车载多媒体设备等）的连接情况有关。

“Force Use”策略，简单地说就是我们是否设置了强制使用某个设备进行输出，可以通过AudioManager里面提供的一些方法来设置，比如：

AudioManager.setSpeakerphoneOn ：切换到扬声器

AudioManager.setBluetoothScoOn ： 切换到蓝牙耳机HFP

除此之外还有很多未公开的切换对象，比如A2DP、HDMI、USB声卡等等。

“Force Use”除了强制使用的输出目标设备之外，还设置了这个“Force Use”所应用的场景，比如setSpeakerphoneOn 就是针对电话通过、VOIP通话等场景，Force Use A2DP就是针对播放音乐的场景。

最后来我们来分析一下通过路由策略选择设备的函数：

audio\_devices\_t Engine::getDeviceForStrategy(routing\_strategy strategy) const

{

        ..........................

  case STRATEGY\_PHONE:

        ......................

        // 首先分析当前系统是否强制在电话通话的场景下使用了某些输出设备

        switch (mForceUse[AUDIO\_POLICY\_FORCE\_FOR\_COMMUNICATION]) {

 // 如果设置了强制切换到蓝牙SCO情况

        case AUDIO\_POLICY\_FORCE\_BT\_SCO:

            if (!isInCall() || strategy != STRATEGY\_DTMF) {

          // 系统的音频模式未处于通话状态下，判断当前系统中的OUT\_BLUETOOTH\_SCO\_CARKIT是否有效

                 // 如果有效就选择这个设备

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT;

                if (device) break;

            }

                    // 接下来尝试选择BLUETOOTH\_SCO\_HEADSET设备

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET;

            if (device) break;

                    // 接下来尝试选择BLUETOOTH\_SCO设备

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO;

            if (device) break;

            // 如果设置了FORCE\_BT\_SCO，却没有任何一个蓝牙HFP连接，就进入FORCE\_NONE的情景

        default:    // FORCE\_NONE 也就是没有设置电话场景下的Force use

            // 如果没有在电话中，尝试在STREAM\_VOICE\_CALL与A2DP设备间建立音频路由

            if (!isInCall() &&

                    (mForceUse[AUDIO\_POLICY\_FORCE\_FOR\_MEDIA] != AUDIO\_POLICY\_FORCE\_NO\_BT\_A2DP) &&

                    (outputs.getA2dpOutput() != 0)) {

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES;

                if (device) break;

            }

      // 尝试选择有线耳机、头戴式耳机等设备

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE;

            if (device) break;

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET;

            if (device) break;

      // 尝试选择USB声卡设备

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_USB\_DEVICE;

            if (device) break;

            if (!isInCall()) {

                            // audio mode未处于电话状态下，尝试选择以下设备（不知道到底是什么设备）

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_DGTL\_DOCK\_HEADSET;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_ANLG\_DOCK\_HEADSET;

                if (device) break;

            }

                    // 尝试选择耳机输出

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_EARPIECE;

            if (device) break;

            // 如果连机身耳机都不支持，就选择系统的默认输出设备（在auido\_policy.conf里面配置的）

            device = mApmObserver->getDefaultOutputDevice()->type();

            if (device == AUDIO\_DEVICE\_NONE) {

                ALOGE("getDeviceForStrategy() no device found for STRATEGY\_PHONE");

            }

            break;

    // 如果强制使用扬声器

        case AUDIO\_POLICY\_FORCE\_SPEAKER:

            // 在电话通话中，A2DP设备有效的情况下，尝试使用A2DP设备

            if (!isInCall() &&

                    (mForceUse[AUDIO\_POLICY\_FORCE\_FOR\_MEDIA] != AUDIO\_POLICY\_FORCE\_NO\_BT\_A2DP) &&

                    (outputs.getA2dpOutput() != 0)) {

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER;

                if (device) break;

            }

            if (!isInCall()) {

           // audio mode未处于电话状态下，尝试选择以下设备（不知道到底是什么设备）

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_USB\_DEVICE;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_DGTL\_DOCK\_HEADSET;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL;

                if (device) break;

                device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_ANLG\_DOCK\_HEADSET;

                if (device) break;

            }

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_LINE;

            if (device) break;

            // 尝试使用扬声器输出

            device = availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_SPEAKER;

            if (device) break;

      // 如果系统不支持扬声器，就选择系统默认输出设备

            device = mApmObserver->getDefaultOutputDevice()->type();

            if (device == AUDIO\_DEVICE\_NONE) {

                ALOGE("getDeviceForStrategy() no device found for STRATEGY\_PHONE, FORCE\_SPEAKER");

            }

            break;

        }

    break;

    .....................

}

通过对这个方法的分析，我们可以看出，音频设备也是有优先级的。

AudioPolicy就分析得差不多了，这个模块到底在AudioTrack的初始化过程中起什么作用呢？其实就是为了选择一个具体的输出流，定位一个工作线程，与一个具体的音频播放设备建立路由，还有，加载对应的HAL动态库和接口。

最后用一个序列图来梳理一下getOutputForAttr的流程：

**2.3. 共享内存和CB对象**

AudioTrack初始化完成过后，我们就可以向AudioTrack中写数据了。现在我们需要思考，数据写入的目标内存是哪里，由哪个模块创建的？在AudioTrack初始化时获取了一个audio\_track\_cblk\_t对象，这个对象用来做什么？在这一节中我们就来分析一下这些问题。

来看看Track对象的构造函数：

AudioFlinger::PlaybackThread::Track::Track(

            ...................)

// 调用父类的构造方法

    :   TrackBase(thread, client, sampleRate, format, channelMask, frameCount,

                  (sharedBuffer != 0) ? sharedBuffer->pointer() : buffer,

                  sessionId, uid, flags, true /\*isOut\*/,

                  (type == TYPE\_PATCH) ? ( buffer == NULL ? ALLOC\_LOCAL : ALLOC\_NONE) : ALLOC\_CBLK,

                  type),

    .................

{

    .....................

if (sharedBuffer == 0) {

// MODE\_STREAM模式，创建一个AudioTrackServerProxy对象来操作缓冲区

        mAudioTrackServerProxy = new AudioTrackServerProxy(mCblk, mBuffer, frameCount,

                mFrameSize, !isExternalTrack(), sampleRate);

    } else {

        mAudioTrackServerProxy = new StaticAudioTrackServerProxy(mCblk, mBuffer, frameCount,

                mFrameSize);

    }

    mServerProxy = mAudioTrackServerProxy;

    mName = thread->getTrackName\_l(channelMask, format, sessionId);

    if (mName < 0) {

        ALOGE("no more track names available");

        return;

    }

    // only allocate a fast track index if we were able to allocate a normal track name

    if (flags & IAudioFlinger::TRACK\_FAST) {

      ..................

    }

}

Track类继承自TrackBase，所以在初始化的时候也调用了TrackBase的构造函数：

AudioFlinger::ThreadBase::TrackBase::TrackBase(

           .............)

    :  ...............

{

..........................

size\_t size = sizeof(audio\_track\_cblk\_t);

// 计算共享内存的大小

    size\_t bufferSize = (buffer == NULL ? roundup(frameCount) : frameCount) \* mFrameSize;

if (buffer == NULL && alloc == ALLOC\_CBLK) {

// ALLOC\_CBLK模式，加上CP结构体的大小

        size += bufferSize;

    }

if (client != 0) {

        // 通过Client对象持有的MemoryDealer对象，分配共享内存

        mCblkMemory = client->heap()->allocate(size);

        if (mCblkMemory == 0 ||

                (mCblk = static\_cast<audio\_track\_cblk\_t \*>(mCblkMemory->pointer())) == NULL) {

            ALOGE("not enough memory for AudioTrack size=%u", size);

            client->heap()->dump("AudioTrack");

            mCblkMemory.clear();

            return;

        }

    } else {

        // this syntax avoids calling the audio\_track\_cblk\_t constructor twice

        mCblk = (audio\_track\_cblk\_t \*) new uint8\_t[size];

        // assume mCblk != NULL

    }

    // construct the shared structure in-place.

    if (mCblk != NULL) {

        / 这句代码是关键，placement new关键字，用于在mCblk这块共享内存上创建一个对象，这个对象可以被访问这块共     // 享内存的多个进程访问

  new(mCblk) audio\_track\_cblk\_t();

        switch (alloc) {

        ...............

        case ALLOC\_CBLK:

            // clear all buffers

            if (buffer == NULL) {

          // MODE\_STREAM模式时，需要获取用于读、写数据的共享内存的首地址并保存在mBuffer 指针中

                mBuffer = (char\*)mCblk + sizeof(audio\_track\_cblk\_t);

                memset(mBuffer, 0, bufferSize);

            } else {

                mBuffer = buffer;

            }

            break;

        case ALLOC\_LOCAL:

            mBuffer = calloc(1, bufferSize);

            break;

        case ALLOC\_NONE:

            mBuffer = buffer;

            break;

        }

    }

}

 看这一句代码：

mCblkMemory = client->heap()->allocate(size);

Heap函数返回了一个MemoryDealer对象，通过其allocate方法创建了一块共享内存，并将其首地址保存在mCblk中，内存的大小在之前计算出来了，是创建AudioTrack时指定的内存大小，加上audio\_track\_cblk\_t结构体的大小。

在此之后，有一段很奇怪的代码：

  new(mCblk) audio\_track\_cblk\_t();

这是C++中的placement new；一般的new关键字只能在堆上面分配对象，如果要在一块指定的非堆内存上分配对象，就必须使用placement new关键字。

placement new的用法，new后面的括号里是内存，紧接其后的是一个类的构造函数，表示在括号里指定的内存中创建一个对象。使用placement new，使得audio\_track\_cblk\_t创建在共享内存上，它就自然而然地能被多个进程看见并使用了。

CB对象主要是对音频播放、录制流程中的音频数据的生产者、消费者的活动进行协调，以音频播放为例，AudioTrack作为音频数据的生产者，持有一个ClientProxy对象，通过CB对象来移动写指针，找到共享内存中允许写入数据的空闲缓冲区的位置并写入音频数据；AudioFlinger作为音频数据的消费者，持有一个ServerProxy对象，通过CP对象来移动读指针，找到缓冲区中已写入数据的首地址，读取数据，并移动读指针到新的位置。CB对象将共享内存作为环状缓冲来维护:

如图所示，共享内存中被分配来容纳数据的缓冲区长度为mFrameCount个frame，缓冲区的首部填充了一个CB对象，它通过front指向缓冲区中已写入的数据的首地址，代表读数据的位置；通过rear指向缓冲区中已经写入的数据的末尾，代表新的数据将从哪个位置开始写入；红色的部分，也就是rear - front，代表已经写入的数据有多少个frame，蓝色的部分为空闲缓冲区。

front指针不能超过rear指针的位置，front和rear指针在到达缓冲区的末尾时，将回到0位置，这样就达到了环状缓冲的目的，能够有效地利用有限的内存空间，供生产者和消费者使用。

audio\_track\_cblk\_t类中有一个AudioTrackShareStreaming对象，它提供了mRear和mFront两个参数，是否就是途中的front指针和rear指针呢：

其实不是，rear和front是根据mRear和mFront两个参数，以及缓冲区的最大frame数计算出来的。对于这个观点，我们需要分析ClientProxy和ServerProxy的相关函数。

首先看生产者端，也就是AudioTrack端：

status\_t ClientProxy::obtainBuffer(Buffer\* buffer, const struct timespec \*requested,

        struct timespec \*elapsed)

{

   ......................

    for (;;) {

       ................

        int32\_t front;

        int32\_t rear;

  // 使用AudioTrack播放音频的时候mIsOut为true

        if (mIsOut) {

            front = android\_atomic\_acquire\_load(&cblk->u.mStreaming.mFront);

            rear = cblk->u.mStreaming.mRear;

        } else {

           ........................

        }

  // 通过rear - front确定当前缓冲区中已经写入的数据的长度

        ssize\_t filled = rear - front;

        // 进行有效性判断

        if (!(0 <= filled && (size\_t) filled <= mFrameCount)) {

            ..................

        }

        // 计算空闲的数据缓冲区大小

        size\_t avail = mIsOut ? mFrameCount - filled : filled;

        if (avail > 0) {

            // 'avail' may be non-contiguous, so return only the first contiguous chunk

            size\_t part1;

            if (mIsOut) {

                       // 这句代码就是关键了，通过mRear指针以及缓冲区总大小，计算出写数据的位置

                rear &= mFrameCountP2 - 1;

                           // 计算rear到缓冲区末尾之间的空间的大小

                part1 = mFrameCountP2 - rear;

            } else {

                front &= mFrameCountP2 - 1;

                part1 = mFrameCountP2 - front;

            }

            if (part1 > avail) {

                part1 = avail;

            }

                    // 如果part1大于用户希望申请的大小，那么以后者为准

            if (part1 > buffer->mFrameCount) {

                part1 = buffer->mFrameCount;

            }

                    // 返回写数据的缓冲区的位置，以及允许写入的最大帧数

            buffer->mFrameCount = part1;

            buffer->mRaw = part1 > 0 ?

                    &((char \*) mBuffers)[(mIsOut ? rear : front) \* mFrameSize] : NULL;

            buffer->mNonContig = avail - part1;

            mUnreleased = part1;

            status = NO\_ERROR;

                     // 如果缓冲区分配成功，返回给用户

            break;

        }

// 如果没有空闲的缓冲区，那么通过入参判断是否需要等待，需不需要超时

        struct timespec remaining;

        const struct timespec \*ts;

        switch (timeout) {

        ...............

        }

        int32\_t old = android\_atomic\_and(~CBLK\_FUTEX\_WAKE, &cblk->mFutex);

        if (!(old & CBLK\_FUTEX\_WAKE)) {

            if (measure && !beforeIsValid) {

                clock\_gettime(CLOCK\_MONOTONIC, &before);

                beforeIsValid = true;

            }

            errno = 0;

                     // 通过Futex，等待消费者将缓冲区中的数据消费，产生空闲空闲区域供生产者使用

            (void) syscall(\_\_NR\_futex, &cblk->mFutex,

                    mClientInServer ? FUTEX\_WAIT\_PRIVATE : FUTEX\_WAIT, old & ~CBLK\_FUTEX\_WAKE, ts);

            ..................

            }

        }

    }

    ...........................

    return status;

}

AudioTrack将对obtainBuffer获取的缓冲区进行写操作，之后调用releaseBuffer函数释放缓冲区，看看这个函数的实现：

void ClientProxy::releaseBuffer(Buffer\* buffer)

{

    // 写入了多少数据？

    size\_t stepCount = buffer->mFrameCount;

    if (stepCount == 0 || mIsShutdown) {

        // prevent accidental re-use of buffer

        buffer->mFrameCount = 0;

        buffer->mRaw = NULL;

        buffer->mNonContig = 0;

        return;

    }

    mUnreleased -= stepCount;

    audio\_track\_cblk\_t\* cblk = mCblk;

    // Both of these barriers are required

if (mIsOut) {

    // 更新mRear的位置

        int32\_t rear = cblk->u.mStreaming.mRear;

        android\_atomic\_release\_store(stepCount + rear, &cblk->u.mStreaming.mRear);

    } else {

        int32\_t front = cblk->u.mStreaming.mFront;

        android\_atomic\_release\_store(stepCount + front, &cblk->u.mStreaming.mFront);

    }

}

从这个函数中可以看出，释放缓冲的时候，mRear加上了写入的帧数，也就是说，随着AudioTrack不断obtain和release，mRear会不断增加，很容易就超过了缓冲区的最大帧数。所以，这个mRear并非直接指向缓冲区的写位置，而是需要在obtainBuffer中，通过“rear &= mFrameCountP2 - 1”这个公式来计算出写位置的（mFrameCountP2 是供读写的共享内存的总帧数，进行了一次向上舍入），这样rear到达了缓冲区末尾，会回到缓冲区的首部，达到环状缓冲的效果。

当共享内存里面没有剩余空间的时候，代表消费者消费音频数据的速度太慢，如果入参配置了需要等待，就通过Futex来进行等候，直到消费者消费了音频数据，通过Futex来通知生产者，生产者将会判断剩余空间是否满足消费者的要求，如果不满足，继续等待。

AudioFlinger作为消费者，需要通过ServerProxy来读取缓冲区中的数据，下面我们来看ServerProxy::obtainBuffer：

status\_t ServerProxy::obtainBuffer(Buffer\* buffer, bool ackFlush)

{

    ...................

    {

    audio\_track\_cblk\_t\* cblk = mCblk;

    int32\_t front;

    int32\_t rear;

    // AudioTrackServerProxy的mIsOut为true

    if (mIsOut) {

        int32\_t flush = cblk->u.mStreaming.mFlush;

        rear = android\_atomic\_acquire\_load(&cblk->u.mStreaming.mRear);

        front = cblk->u.mStreaming.mFront;

        if (flush != mFlush) {

            ...................

            // 处理Client端的flush事件

.                 ...................

            front = newFront;

        }

    } else {

        front = android\_atomic\_acquire\_load(&cblk->u.mStreaming.mFront);

        rear = cblk->u.mStreaming.mRear;

}

// 缓冲区中已经写入的数据长度

    ssize\_t filled = rear - front;

    // pipe should not already be overfull

    if (!(0 <= filled && (size\_t) filled <= mFrameCount)) {

        ALOGE("Shared memory control block is corrupt (filled=%zd); shutting down", filled);

        mIsShutdown = true;

    }

    if (mIsShutdown) {

        goto no\_init;

    }

    // don't allow filling pipe beyond the nominal size

    size\_t availToServer;

if (mIsOut) {

        // 计算缓冲区中消费者能够读取的帧数

        availToServer = filled;

        mAvailToClient = mFrameCount - filled;

    } else {

        availToServer = mFrameCount - filled;

        mAvailToClient = filled;

    }

    // 'availToServer' may be non-contiguous, so return only the first contiguous chunk

    size\_t part1;

if (mIsOut) {

        // 计算读取数据的位置，如果front到达了缓冲区末尾，将味道缓冲区头部

        front &= mFrameCountP2 - 1;

        part1 = mFrameCountP2 - front;

    } else {

        rear &= mFrameCountP2 - 1;

        part1 = mFrameCountP2 - rear;

    }

if (part1 > availToServer) {

    // 不能超过剩余帧数

        part1 = availToServer;

    }

    size\_t ask = buffer->mFrameCount;

    if (part1 > ask) {

        part1 = ask;

    }

    // 将读缓冲的首地址和允许读取的最大帧数返回给消费者

    buffer->mFrameCount = part1;

    buffer->mRaw = part1 > 0 ?

            &((char \*) mBuffers)[(mIsOut ? front : rear) \* mFrameSize] : NULL;

    buffer->mNonContig = availToServer - part1;

    ........................

    return NO\_INIT;

}

ServerProxy::releaseBuffer：

void ServerProxy::releaseBuffer(Buffer\* buffer)

{

    ..................

audio\_track\_cblk\_t\* cblk = mCblk;

// AudioTrackServerProxy的mIsOut为true

if (mIsOut) {

        // 移动mFront到新的位置

        int32\_t front = cblk->u.mStreaming.mFront;

        android\_atomic\_release\_store(stepCount + front, &cblk->u.mStreaming.mFront);

    } else {

        int32\_t rear = cblk->u.mStreaming.mRear;

        android\_atomic\_release\_store(stepCount + rear, &cblk->u.mStreaming.mRear);

    }

......................

    if (!mIsOut || (mAvailToClient + stepCount >= minimum)) {

        // 消费数据之后，如果空余出来的缓冲区大于生产者需要的缓冲区大小，则通知生产者

        int32\_t old = android\_atomic\_or(CBLK\_FUTEX\_WAKE, &cblk->mFutex);

        if (!(old & CBLK\_FUTEX\_WAKE)) {

            (void) syscall(\_\_NR\_futex, &cblk->mFutex,

                    mClientInServer ? FUTEX\_WAKE\_PRIVATE : FUTEX\_WAKE, 1);

        }

    }

}

AudioFlinger通过front指针，来获取读位置，将数据从共享内存中消费，并移动front指针来指向未消费的数据的位置。

总之，在AudioTrack播放音频的过程中，在AudioFlinger为止创建对应的Track对象的时候，创建了一块长度由AudioTrack的调用者指定的共享内存。每个AudioTrack都对应了一块共享内存，作为生产者，将数据写入这块共享内存，并由AudioFlinger作为消费者，将数据从共享内存中拉取出来。

**2.4. Play时候做了些什么**

AudioTrack初始化完毕后，在写入数据之前，调用了play方法，它调用了native\_start方法，这个方法是个JNI函数，自然我们得看android\_media\_AudioTrack.cpp中的实现：

static void

android\_media\_AudioTrack\_start(JNIEnv \*env, jobject thiz)

{

    sp<AudioTrack> lpTrack = getAudioTrack(env, thiz);

    if (lpTrack == NULL) {

        jniThrowException(env, "java/lang/IllegalStateException",

            "Unable to retrieve AudioTrack pointer for start()");

        return;

    }

    lpTrack->start();

}

不出意外，依然是由AudioTrack Native来实现的:

status\_t AudioTrack::start()

{

    .........................

    status\_t status = NO\_ERROR;

    if (!(flags & CBLK\_INVALID)) {

        status = mAudioTrack->start();

        if (status == DEAD\_OBJECT) {

            flags |= CBLK\_INVALID;

        }

    }

    ................

    return status;

}

通过之前的分析我们知道mAudioTrack的被代理对象是TrackHandle对象，这个对象包装了AudioTrack在AudioFlinger中对应PlaybackThread线程里的一个Track对象，来看看Track的start函数：

status\_t AudioFlinger::PlaybackThread::Track::start(AudioSystem::sync\_event\_t event \_\_unused,

                                                    int triggerSession \_\_unused)

{

......................

   sp<ThreadBase> thread = mThread.promote();

    .................

if (thread != 0) {

.......................

        PlaybackThread \*playbackThread = (PlaybackThread \*)thread.get();

        ..............................

  // 调用了这个Track所依附的工作线程的addTrack\_l函数

        status = playbackThread->addTrack\_l(this);

        if (status == INVALID\_OPERATION || status == PERMISSION\_DENIED) {

            triggerEvents(AudioSystem::SYNC\_EVENT\_PRESENTATION\_COMPLETE);

            //  restore previous state if start was rejected by policy manager

            if (status == PERMISSION\_DENIED) {

                mState = state;

            }

        }

        ...............

    } else {

        status = BAD\_VALUE;

    }

    return status;

}

继续分析：

status\_t AudioFlinger::PlaybackThread::addTrack\_l(const sp<Track>& track)

{

    status\_t status = ALREADY\_EXISTS;

    track->mRetryCount = kMaxTrackStartupRetries;

    if (mActiveTracks.indexOf(track) < 0) {

        if (track->isExternalTrack()) {

      // 一般情况下都是External Track

            TrackBase::track\_state state = track->mState;

            mLock.unlock();

      // 由Audio Policy负责启动对应的输出流，mId是输出流的句柄，在之前调用getOutputForAttr时获取到的

            status = AudioSystem::startOutput(mId, track->streamType(),

                                              (audio\_session\_t)track->sessionId());

            mLock.lock();

            // abort track was stopped/paused while we released the lock

            if (state != track->mState) {

                if (status == NO\_ERROR) {

                    mLock.unlock();

                    AudioSystem::stopOutput(mId, track->streamType(),

                                            (audio\_session\_t)track->sessionId());

                    mLock.lock();

                }

                return INVALID\_OPERATION;

            }

            // abort if start is rejected by audio policy manager

            if (status != NO\_ERROR) {

                return PERMISSION\_DENIED;

            }

        }

        track->mFillingUpStatus = track->sharedBuffer() != 0 ? Track::FS\_FILLED : Track::FS\_FILLING;

        track->mResetDone = false;

        track->mPresentationCompleteFrames = 0;

// 把这个Track加入到mActiveTracks集合中

        mActiveTracks.add(track);

        ..................

    }

    onAddNewTrack\_l();

    return status;

}

前面我知道Track被工作线程创建的时候，被加入了PlaybackThread::mTracks集合中，在Play的时候，又把它加入到了mActiveTracks集合中，也就是说，在这个集合中的Track对象是“active”的，不在这个集合中，却在mTracks集合中的，是不处于“active状态的”，工作线程只会从所有的“active”的Track中消费数据。一个AudioTrack进行play的时候，对应的Track对象便被其依附的工作线程设置为active状态。

在将Track对象add到mActiveTracks集合之前，还调用了一个AudioSystem::startOutput方法，这个方法是由AudioPolicy来实现的，从字面意思来看，是为了启动一个输出流。

从之前分析AudioTrack的初始化的时候，我们知道，通过getOutputForAttr获取了一个输出流的audio\_io\_handle\_t 句柄，这个句柄在AudioPolicy中对应了一个输出流对象，在AudioFlinger中对应了一个工作线程。现在startOutput就准备对这个输出流发送“启动”命令了，看看做了一些什么事情：

status\_t AudioPolicyManager::startOutput(audio\_io\_handle\_t output,

                                             audio\_stream\_type\_t stream,

                                             audio\_session\_t session)

{

    // 通过句柄定位到输出流对象

    ssize\_t index = mOutputs.indexOfKey(output);

    if (index < 0) {

        ALOGW("startOutput() unknown output %d", output);

        return BAD\_VALUE;

    }

    sp<SwAudioOutputDescriptor> outputDesc = mOutputs.valueAt(index);

..................

    uint32\_t delayMs = 0;

// 这个输出流对象是个source，充当数据生产者，所以要调用startSource（sink是对应的音频输出设备）

    status\_t status = startSource(outputDesc, stream, newDevice, &delayMs);

    if (status != NO\_ERROR) {

        mOutputRoutes.decRouteActivity(session);

        return status;

}

..............

    return status;

}

startSource方法启动了这个输出流，这个方法的步骤为：

l  查看这个输出流是否需要强制更新路由

l  调用setOutputDevice函数

setOutputDevice函数在之前分析过，主要的目的是建立一条输出流与具体的音频输出设备之间的路由。

当然，在某些情况下，不需要重新建立输出路由，在哪些情况下不改变这个输出流的音频路由呢？要同时满足3个条件

l  目标设备为AUDIO\_DEVICE\_NONE类型或者新设置的设备与这个输出流之前的路由设备一致

l  force == false

l  这个输出流已经连接建立了一条有效的路由

下面用一个序列图总结一下:

到现在为止，AudioTrack在AudioPolicyServcie中对应的输出流对象，已经和具体的输出设备进行了绑定，并且设置了输出音量；在AudioFlinger中对应的PlaybackThread，已经把对应的Track对象放入了“Active track”集合中，就等待向AudioTrack中写入数据了。

2.5. 数据是如何写入的

2.5.1.    AudioTrack如何写数据

从例子中可知，在MODE\_STREAM这种数据填充模式下，AudioTrack不断读取文件，没读出一部分PCM数据，就通过write方法写入到AudioTrack中。

ssize\_t AudioTrack::write(const void\* buffer, size\_t userSize, bool blocking)

{

    ...................

    size\_t written = 0;

    Buffer audioBuffer;

// userSize是调用者希望写入的音频数据字节数，循环直到userSize为0为止

while (userSize >= mFrameSize) {

        // 计算userSize对应的帧数

        audioBuffer.frameCount = userSize / mFrameSize;

// 通过obtainBuffer函数获取内存缓冲区，并根据blocking变量决定是否在共享内存的空闲空间不够时进行阻塞等待

        status\_t err = obtainBuffer(&audioBuffer,

                blocking ? &ClientProxy::kForever : &ClientProxy::kNonBlocking);

        if (err < 0) {

            if (written > 0) {

                break;

            }

            return ssize\_t(err);

        }

              // 这次obtainBuffer获取的缓冲区的真实大小（单位为字节数）

        size\_t toWrite = audioBuffer.size;

              // 向缓冲区填充数据

        memcpy(audioBuffer.i8, buffer, toWrite);

              // 定位剩余数据的首地址

        buffer = ((const char \*) buffer) + toWrite;

        // 计算剩余数据的字节数

        userSize -= toWrite;

   // 计算写入的总数据字节数

        written += toWrite;

              // 释放内存缓冲

        releaseBuffer(&audioBuffer);

    }

    return written;

}

AudioTrack的obtainBuffer函数调用了AudioTrackClientPorxy对象的obtainBuffer函数，这个函数在之前分析CB对象的时候已经讲过，它通过CB对象的rear指针，在共享内存中寻找到一块空闲空间的首地址供写入数据。

在AudioTrackClientProxy对象访问缓冲区的时候，由于共享内存的大小限制，当前空闲的内存大小可能会小于待写入的数据字节数，所以分配的缓冲区大小不足以写入所有的数据。在这种情况下，write方法会循环调用obtainBuffer方法，每次写入一部分数据，写完后调用releaseBuffer来更新CB对象的rear指针位置，如果共享内存中空闲内存大小为0，还会进入等待状态，直到消费者从共享内存中消费了一部分数据，有空闲内存待分配为止。

下图为AudioTrack端消费数据的流程：

2.5.2.    AudioFlinger如何消费数据

上一节我们了解到，AudioTrack作为生产者，在不断地向共享内存中写数据。共享内存不是无限的，一个音频文件的PCM数据字节数，一般情况下都会远大于共享内存的大小，如果没有一个消费者不断从共享内存中消费数据，那么，很快共享内存就被生产者填满了，AudioTrack调用obtainBuffer函数时也会被阻塞。

消费者是谁呢？之前我们笼统地把消费者当作AudioFlinger服务，那么到底是AudioFlinger的哪个组件不断从共享内存中消费数据的呢？答案是PlaybackThread。

既然PlaybackThread是个线程，那么它肯定有一个具体的执行函数，在线程的生命周期之内，不断对共享内存中的数据进行消费。PlaybackThread是Native层的线程类ThreadBase的子类，实现了threadLoop作为其工作函数。这个线程是什么时候开始执行的呢？PlaybackThread重写了onFirstRef函数，在函数体中执行了run函数：

void AudioFlinger::PlaybackThread::onFirstRef()

{

    run(mThreadName, ANDROID\_PRIORITY\_URGENT\_AUDIO);

}

线程便开始执行，调用threadLoop方法。

bool AudioFlinger::PlaybackThread::threadLoop()

{

Vector< sp<Track> > tracksToRemove;

..................

    checkSilentMode\_l();

    // 循环直到线程被中止

    while (!exitPending())

    {

             ................

            if ((!mActiveTracks.size() && systemTime() > mStandbyTimeNs) ||

                                   isSuspended()) {

                // 如果没有任何一个Track的数据可读，或者处于暂停状态，置硬件为standby状态

                if (shouldStandby\_l()) {

                    threadLoop\_standby();

                    mStandby = true;

                }

          ............................

            }

            // 这个函数是关键，用于判断mActiveTracks集合中的各个Track是否有数据要处理

      // 并将一些状态变为不活动状态的Track添加到tracksToRemove中，以便循环结束后从mActiveTracks中移除

            mMixerStatus = prepareTracks\_l(&tracksToRemove);

           ....................

        } // mLock scope ends

        if (mBytesRemaining == 0) {

            mCurrentWriteLength = 0;

      // 如果mActiveTracks中的一个或者多个Track的数据可读，则使用AudioMixer进行混音

            if (mMixerStatus == MIXER\_TRACKS\_READY) {

               threadLoop\_mix();

            } else if ((mMixerStatus != MIXER\_DRAIN\_TRACK)

                        && (mMixerStatus != MIXER\_DRAIN\_ALL)) {

                // 没有数据需要混合，则休眠

                threadLoop\_sleepTime();

                if (mSleepTimeUs == 0) {

                    mCurrentWriteLength = mSinkBufferSize;

                }

            }

            // 混叠之后的数据保存在mMixerBuffer 这个缓冲区中

            if (mMixerBufferValid) {

                void \*buffer = mEffectBufferValid ? mEffectBuffer : mSinkBuffer;

                audio\_format\_t format = mEffectBufferValid ? mEffectBufferFormat : mFormat;

                       // 如果没设置音效，拷贝数据到mSinkBuffer中

                memcpy\_by\_audio\_format(buffer, format, mMixerBuffer, mMixerBufferFormat,

                        mNormalFrameCount \* mChannelCount);

            }

            ........................

        }

  ......................

        if (!waitingAsyncCallback()) {

            // mSleepTimeUs == 0 means we must write to audio hardware

            if (mSleepTimeUs == 0) {

                ssize\_t ret = 0;

                if (mBytesRemaining) {

              // 将数据写入具体的硬件设备，进行播放

                    ret = threadLoop\_write();

                    ....................

                } else if ((mMixerStatus == MIXER\_DRAIN\_TRACK) ||

                        (mMixerStatus == MIXER\_DRAIN\_ALL)) {

                    threadLoop\_drain();

                }

              .......................

                } else {

                  // 如果没有数据要写，休眠一段时间

                ATRACE\_BEGIN("sleep");

                usleep(mSleepTimeUs);

                ATRACE\_END();

            }

        }

        // 如果部分Track变为不活动状态，从mActiveTracks中移除

        threadLoop\_removeTracks(tracksToRemove);

        tracksToRemove.clear();

   ....................

}

...................

    return false;

}

从工作线程的执行函数可以看出，这个线程承担着消费各个处于活动状态的Track中的数据，进行混音操作，并写入到音频硬件中播放的任务，主要流程大概是这样的：

l  判断是否需要设置standby状态；

l  调用prepareTracks\_l函数判断是否有处于活动状态的Track，这些Track的数据是否准备好，是否有一些Track重新处于不活动状态；

l  如果有一个或者多个Track的数据可读，便进行混音操作，将混音后的数据拷贝到一个缓冲区中；

l  将混音后的数据写入硬件，进行播放；

在threadLoop方法中有几个threadLoop\_\*函数，每个函数都在线程的工作过程中承担一些核心工作，比如：

l  threadLoop\_standby

这个函数用于将线程和音频硬件设置为StandBy状态，这个状态可以翻译成“待命状态”，意思是当前没有处于活动状态的Track，没有数据可写，所以工作线程将休眠一段时间，音频硬件也暂时处于不工作的状态，直到有数据要写入硬件，才会解除待命状态。

l  threadLoop\_mix

当prepareTracks\_l被调用后，如果当前有处于活动状态的Track，其共享内存中的音频数据可读，那么，工作线程就会调用这个函数，使用AudioMixer对数据进行混音操作。

l  threadLoop\_write

负责将混音后的数据写入到音频硬件中，它调用了NBAIO（Non-block Audio I/O ）的接口来写入数据。

l  threadLoop\_removeTracks

负责将prepareTracks\_l函数执行过程中重新标记为不活动的Track，从,mActivieTracks中移除

接下来我们来看看，prepareTracks\_l是怎么确定能否进行混音操作的，又是如何将Track重置为不活动状态的，见如下的伪代码：

count = mActiveTracks.size；

for (int i = 0; i < count; i++) {

     sp<Track> track = mActiveTracks[i];

     int frameReady = track->framesReady;

     If   framesReady > min Size

         AND track->isReady

         AND !track->isPaused

         AND !track->isTerminated

     Then

         开始准备AudioMixer;

         mAudioMixer->setBufferPorvider(track);

         mAudioMixer->enable(name);

         计算左右声道音量；

         mAudioMixer->setParameter(左声道音量)；

         mAudioMixer->setParameter(右声道音量)；

         mAudioMixer->setParameter(format)；

         mAudioMixer->setParameter(channelMask)；

         mAudioMixer->setParameter(sample Rate)；

.         ............

         mixerStatus = MIXER\_TRACKS\_READY;

    Else

         tracksToRemove->add(track);

         mAudioMixer->disable(name);

    End if

}

removeTracks\_l(\*tracksToRemove);

return mixerStatus;

prepareTracks\_l将对mActiveTracks集合中所有Track进行遍历，查看每个Track是否依然处于活动状态，判断标准有4条：

l  Track的共享内存中的rear - front大于一个最小值

l  Track处于“ready”状态

l  Track不处于pause状态

l  Track不处于terminate状态

如果这个Track同时满足这4个条件，那么Track依然是活动的，工作线程能够对Track的数据进行消费，接下来工作线程将对AudioMixer混音器对象进行配置，并将Track设置为混音器对象的BufferProvider，为混音器设置音量、数据格式、声道数、采样率等数据，并允许混音器访问这个Track；这里并没有进行实际的混音操作。

如果Track不满足其中任意一个条件，那么，这个Track将被添加到tracksToRemove中，循环结束后，removeTracks函数将会把这个Track从mActiveTracks中移除。

AudioMixer是如何获取Track的共享内存中的数据呢？我们先来看看这张类图：

AudioMixer最多能够支持32个Track混音，每个Track在AuidoMixer中对应一个track\_t对象，这个依赖于一个AudioBufferProvider对象来读取数据，Track是这个类的子类。这样，AudioMixer可以通过track\_t对象调用Track所封装的getNextBuffer、releaseBuffer函数来消费Track的共享内存中的数据。

之前分析AudioTrack的时候得知，AudioTrack通过一个AudioTrackClientProxy类对象来获取、释放写缓冲，AudioTrackClientProxy是Proxy类的子类，访问audio\_track\_cblk\_t对象的rear指针，从而寻找到共享内存中空闲空间的首地址和空间的长度，向共享内存中写入数据，履行AudioTrack作为生产者的职责。生产者写完数据后，AudioTrackClientProxy还会提供方法，移动rear指针指向已写入的数据的末尾，以便下一次写入数据。

Track对象依赖于AudioTrackServerProxy对象来获取共享内存中已经被AudioTrack写入的数据的首地址，以及可读的数据的长度，AudioTrackServerProxy和AudioTrackClientProxy同属于Proxy类的子类，同样实现了obtainBuffer和releaseBuffer方法，只是实现不一致，ServerProxy主要是访问audio\_track\_cblk\_t对象的front指针，消费完毕后，ServerProxy还要负责移动front指针到新的读数据的位置。

我们用一个系列图来描述一下AudioMixer通过Track对象来消费数据的过程：

2.6. Stop和Release时都做了些什么

AndroidTrack JAVA的release方法的实现如下：

public void release() {

        // even though native\_release() stops the native AudioTrack, we need to stop

        // AudioTrack subclasses too.

        try {

            stop();

        } catch(IllegalStateException ise) {

            // don't raise an exception, we're releasing the resources.

        }

        native\_release();

        mState = STATE\_UNINITIALIZED;

    }

它首先会尝试调用stop，在调用native\_release，这个方法会映射到JNI中的对应函数：

static void android\_media\_AudioTrack\_release(JNIEnv \*env,  jobject thiz) {

    // 清除AudioTrack Native对象的全局指针

    sp<AudioTrack> lpTrack = setAudioTrack(env, thiz, 0);

    if (lpTrack == NULL) {

        return;

    }

    lpTrack->stop();

    // 释放JNI data

    AudioTrackJniStorage\* pJniStorage = (AudioTrackJniStorage \*)env->GetLongField(

        thiz, javaAudioTrackFields.jniData);

    // reset the native resources in the Java object so any attempt to access

    // them after a call to release fails.

    env->SetLongField(thiz, javaAudioTrackFields.jniData, 0);

    if (pJniStorage) {

        Mutex::Autolock l(sLock);

        audiotrack\_callback\_cookie \*lpCookie = &pJniStorage->mCallbackData;

        //ALOGV("deleting pJniStorage: %x\n", (int)pJniStorage);

        while (lpCookie->busy) {

            if (lpCookie->cond.waitRelative(sLock,

                                            milliseconds(CALLBACK\_COND\_WAIT\_TIMEOUT\_MS)) !=

                                                    NO\_ERROR) {

                break;

            }

        }

        sAudioTrackCallBackCookies.remove(lpCookie);

        // 删除初始化时创建的全局引用

        env->DeleteGlobalRef(lpCookie->audioTrack\_class);

        env->DeleteGlobalRef(lpCookie->audioTrack\_ref);

        delete pJniStorage;

    }

}

static sp<AudioTrack> setAudioTrack(JNIEnv\* env, jobject thiz, const sp<AudioTrack>& at)

{

    Mutex::Autolock l(sLock);

    sp<AudioTrack> old =

            (AudioTrack\*)env->GetLongField(thiz, javaAudioTrackFields.nativeTrackInJavaObj);

    if (at.get()) {

        at->incStrong((void\*)setAudioTrack);

    }

if (old != 0) {

    // 引用计数-1

        old->decStrong((void\*)setAudioTrack);

    }

    env->SetLongField(thiz, javaAudioTrackFields.nativeTrackInJavaObj, (jlong)at.get());

return old;

}

当调用setAudioTrack清除AudioTrack对象的全局指针的时候，调用了AudioTrack的decStrong，导致其强引用计数-1，而setAudioTrack返回的强指针，将在android\_media\_AudioTrack\_release函数执行完毕后，也将被析构，导致指向的AudioTrack对象的强引用计数减为0，这个时候，AudioTrack Native对象达到了其生命周期的末尾，被析构。我们来看它的析构函数：

AudioTrack::~AudioTrack()

{

    if (mStatus == NO\_ERROR) {

        // Make sure that callback function exits in the case where

        // it is looping on buffer full condition in obtainBuffer().

        // Otherwise the callback thread will never exit.

        stop();

        if (mAudioTrackThread != 0) {

            mProxy->interrupt();

            mAudioTrackThread->requestExit();   // see comment in AudioTrack.h

            mAudioTrackThread->requestExitAndWait();

            mAudioTrackThread.clear();

        }

        // No lock here: worst case we remove a NULL callback which will be a nop

        if (mDeviceCallback != 0 && mOutput != AUDIO\_IO\_HANDLE\_NONE) {

            AudioSystem::removeAudioDeviceCallback(mDeviceCallback, mOutput);

        }

        IInterface::asBinder(mAudioTrack)->unlinkToDeath(mDeathNotifier, this);

        mAudioTrack.clear();

        mCblkMemory.clear();

        mSharedBuffer.clear();

        IPCThreadState::self()->flushCommands();

        ALOGV("~AudioTrack, releasing session id %d from %d on behalf of %d",

                mSessionId, IPCThreadState::self()->getCallingPid(), mClientPid);

        AudioSystem::releaseAudioSessionId(mSessionId, mClientPid);

    }

}

调用了sp<T>::clear方法:

template<typename T>

void sp<T>::clear() {

    if (m\_ptr) {

        m\_ptr->decStrong(this);

        m\_ptr = 0;

    }

}

m\_ptr指向被sp引用的对象，也就是BpAudioTrack对象，这个时候这个代理端对象的引用计数为0，导致对象被析构。

BpAudioTrack对象倒是被析构了，但是BnAudioTrack对象呢？我们知道每个AudioTrack，AudioFlinger都会给它创建一个TrackHandle对象，一个Track对象，放着它们不管可是会造成内存泄漏的。BpAudioTrack的引用计数归0，导致这个对象被回收的过程中，肯定对它所代理的对象TrackHandle进行了一些操作，从而避免内存泄漏。

首先我们来看看和BpAudioTrack相关的类的类图：

BpAudioTrack和BnAudioTrack，都是RefBase类的子类，RefBase对类本身的引用计数进行了一些维护工作，提供了incStrong和decStrong函数来增加或者减少类对象的引用计数，当一个强指针指向类对象的时候，引用计数加1，强指针超过其生命周期过后，会调用decStrong将对象的引用计数减1，如果发现类对象的强引用计数将被减为0，会调用类对象的一个名为onLastStrongRef方法。当类对象的引用计数为0的时候，将会通过delete关键字回收这个对象。

BpAudioTrack对象在引用计数归0的时候，对它所代理的对象做了一些什么操作呢？通过上面的类图，我们发现这个类的继承树中，有一个BpRefBase类，这个类重写了RefBase基类的onLastStrongRef方法：

void BpRefBase::onLastStrongRef(const void\* /\*id\*/)

{

    if (mRemote) {

        mRemote->decStrong(this);

    }

}

mRemote是个IBinder对象，其实是个BpBinder，在IBinder框架中，代理了一个BBinder对象，BpAudioTrack在强引用计数归0的时候，将会调用父类的onLastStrongRef方法，通过IPC将其代理的BBinder对象的强引用计数减1。

BBinder是个什么对象呢？看看类图，其实就是TrackHandle对象。

所以，BpAudioTrack的强引用计数归0的时候，TrackHandle对象的强引用计数也会减1。在AudioFlinger中，没有任何一个sp指针指向这个TrackHandle对象，只有在AudioTrack通过IPC调用createTrack的时候，才会导致TrackHandle的强引用计数加一。所以，在这次decStrong过后，TrackHandle对象的强引用计数将归0，导致这个对象被析构。

TrackHandle的析构函数中会对Track对象进行一些清除工作，见下图：

Track的terminate函数将被调用，并调用了Track所依附的工作线程的removeTrack\_l方法，将这个Track从工作线程的mTracks集合中删除掉：

void AudioFlinger::PlaybackThread::removeTrack\_l(const sp<Track>& track)

{

track->triggerEvents(AudioSystem::SYNC\_EVENT\_PRESENTATION\_COMPLETE);

// 仅仅从mTracks中移除了，什么时候从mActiveTracks中移除呢？

    mTracks.remove(track);

    deleteTrackName\_l(track->name());

    // redundant as track is about to be destroyed, for dumpsys only

    track->mName = -1;

    ..................

}

通过前面的分析我们知道，除了mTracks集合持有这个Track对象的强引用之外，还有mActivteTracks这个集合，也持有它的强引用，从mTracks集合中移除的时候，Track对象的引用计数不会归0，对象依然存在。

什么时候这个Track对象会从mActivteTracks集合中删除呢？来看看MixerThread:: prepareTracks\_l函数，这里用伪代码来分析：

prepareTracks\_l 函数：

  .................

  If

1、共享内存中还有未消费数据

2、track->isReady() == TRUE

3、track->isPause() == FALSE

4、track->isTerminate() == FALSE

  Then

继续消费数据

  Else

未消费数据被丢弃，并将track对象添加到待删除的Track集合tracksToRemove中

  End if

  ..............

  将tracksToRemove集合中的所有track对象从mActiveTracks集合中删除

也就是说，如果这个track不满足if中的某个条件，就会进入else分支，导致track对象被添加到tracksToRemove中，并在prepareTracks\_l的末尾，从mActivityTracks集合中删除：

void AudioFlinger::PlaybackThread::removeTracks\_l(const Vector< sp<Track> >& tracksToRemove)

{

    size\_t count = tracksToRemove.size();

    if (count > 0) {

        for (size\_t i=0 ; i<count ; i++) {

            const sp<Track>& track = tracksToRemove.itemAt(i);

      // 遍历tracksToRemove，从mActiveTracks中删除

            mActiveTracks.remove(track);

            mWakeLockUids.remove(track->uid());

            mActiveTracksGeneration++;

            ALOGV("removeTracks\_l removing track on session %d", track->sessionId());

            sp<EffectChain> chain = getEffectChain\_l(track->sessionId());

            if (chain != 0) {

                ALOGV("stopping track on chain %p for session Id: %d", chain.get(),

                        track->sessionId());

                chain->decActiveTrackCnt();

            }

            if (track->isTerminated()) {

                removeTrack\_l(track);

            }

        }

    }

}

在调用PlaybackThread::destoryTrack\_l的时候，对Track对象调用了terminate函数，这个时候track->isTerminate() == TRUE，不满足if的条件，进入了我们所希望的else分支。

Track对象从mActiveTracks中移除过后，不再有任何全局的强指针指向这个对象，待所有相关的局部sp指针超过各自生命周期的时候，Track对象就因为强引用计数归0，而导致对象被析构。

所以AudioTrack release的时候，将会导致对应的BpAudioTrack、TrackHandle、Track对象被析构，这样就不会产生内存泄漏了。

在prepareTracks\_l中，如果满足了if的条件，工作线程将继续消费Track的共享内存中写入的音频数据，这个时候播放是不会停止的，如果不满足，Track对象就会被丢弃，对应的数据也就被丢弃了，此时播放就会立刻停止，哪怕音乐还没有放完。

那么，调用AudioTrack::stop的时候，播放会立刻停止么？If中有一个条件track->isReady() == TRUE，如果这个条件不满足，就会进入else分支，来看看isReady的实现：

bool AudioFlinger::PlaybackThread::Track::isReady() const {

    // 如果当前Track处于STOPED状态，也会返回true

    if (mFillingUpStatus != FS\_FILLING || isStopped() || isPausing()) {

        return true;

    }

    if (isStopping()) {

        if (framesReady() > 0) {

            mFillingUpStatus = FS\_FILLED;

        }

        return true;

    }

    if (framesReady() >= mFrameCount ||

            (mCblk->mFlags & CBLK\_FORCEREADY)) {

        mFillingUpStatus = FS\_FILLED;

        android\_atomic\_and(~CBLK\_FORCEREADY, &mCblk->mFlags);

        return true;

    }

    return false;

}

      看到了么？Stop的时候，将Track的状态置为STOPED，这种状态也满足“ready”的要求，所以，播放是不会立刻停止的。