Android 系统分析	版本： <0.1>
	日期： <2010 年 11 月 8 日>
<文档标号 >	

谷歌、高通、三星 Audio 的系统分析

版本历史

日期	版本	描述	作者
2011 年 1 月 08 日	<0.1>	新建	王海斌

1. 概要 2

2. GOOGLE ANDROID AUDIO HAL 2

2.1 AUDIO 系统综述 2

2.2 AUDIO 系统和上层接口 4

2.2.1 Audio 系统的各个层次 4

2.2.2 media 库中的 Audio 框架部分 5

2.2.3 Audio Track 7

2.4.1 .1 AudioFlinger 本地代码 15

2.4.1.2 Audio 系统的 JNI 代码 45

2.4.1.3 Audio 系统的 Java 代码 45

2.4.2 AUDIO 的硬件抽象层 45

2.4.2.1 Audio 硬件抽象层（ HAL ）的接口定义 45

2.4.2.2 AudioFlinger 中自带 Audio 硬件抽象层实现 47

2.4.2.3 Audio 硬件抽象层的真正实现 56

2.4.3 AUDIO HAL 在 ECLAIR /FROYO 和 DONUT 的差异 56

2.4.4 ANDROID POLICY MANGER 58

2.2.5 高通 ANDROID 的 AUDIO HAL 实现方式 59

2.2.5.1 高通的 ANDROID 的 AUDIO 的架构 59

2.2.5.2 高通 ANDROID 的 AUDIO 的具体实现 60

2.2.6 三星 ANDROID 的 AUDIO HAL 的实现方式 69

2.2.6.1 三星 ANDROID 的架构 69

2.2.6.2 三星针对 C110 AUDIO 的 HAL 具体实现 70

参考资料 **85**

Android Audio 系统分析

1. 概要

本文主要分析 Google 的原始 Android 的 Audio 的架构，比较 Google 的 Android 在 1.6 Donut 和 2.2 Froyo 关于 Audio 差异，重点介绍 Audio 的 HAL 的架构以及高通如何针对自己的硬件平台（ QSD8650/QSD8650A ）来实现 Audio 的 HAL 和三星如何针对自己的硬件平台（ S5PC110）来实现 Audio 的 HAL 。

在 android 的 HAL 中，这部分与芯片的功能联系十分紧密，所以一般的芯片厂商都是按照 Google 定义的 HAL 架构进行开发和维护，终端厂商一般不需要改动或只需要轻微的调整而不涉及到架构和接口的改变。对于 Audio 基本上也遵循这个原则。

本文最后结合高通的平台、三星的平台及其他平台的现有方案提出了 LeOS 实现能够兼容不同硬件平台的 Audio 的 HAL 的设计思路和方案。

2. Google Android audio HAL

本章首先基于 Google 的 Android 1.6 （Donut）来分析 Google 的 Android 中 Audio 层次架构，然后分析介绍 Google Android 2.1/2.2(Eclair/Froyo) 与 1.6(Donut) 的 Audio 的差异，再详细的介绍 Audio 的 HAL部分。

2.1 Audio 系统综述

此处主要分析 Android 音频系统的输入 /输出环节，不涉及音频编解码的内容。 Android 音频系统从驱动程序、本地框架到 Java 框架都具有内容。 Android 的 Audio 系统不涉及编解码环节，只是负责上层系统和底层 Audio 硬件的交互，一般以 PCM 作为输入 /输出格式。

Audio 系统在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。 Android 的 Audio 系统的输入 /输出层次一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。

Audio 系统主要分成如下几个层次：

（1） media 库提供的 Audio 系统本地部分接口；

（2） AudioFlinger 作为 Audio 系统的中间层；

（3） Audio 的硬件抽象层（ HAL ）提供底层支持；

（4） Audio 接口通过 JNI 和 Java 框架提供给上层。

Audio 系统的各个层次接口主要提供了两方面功能：放音（ Track）和录音（ Recorder）。

Android 的 Audio 系统结构如图 1 所示。

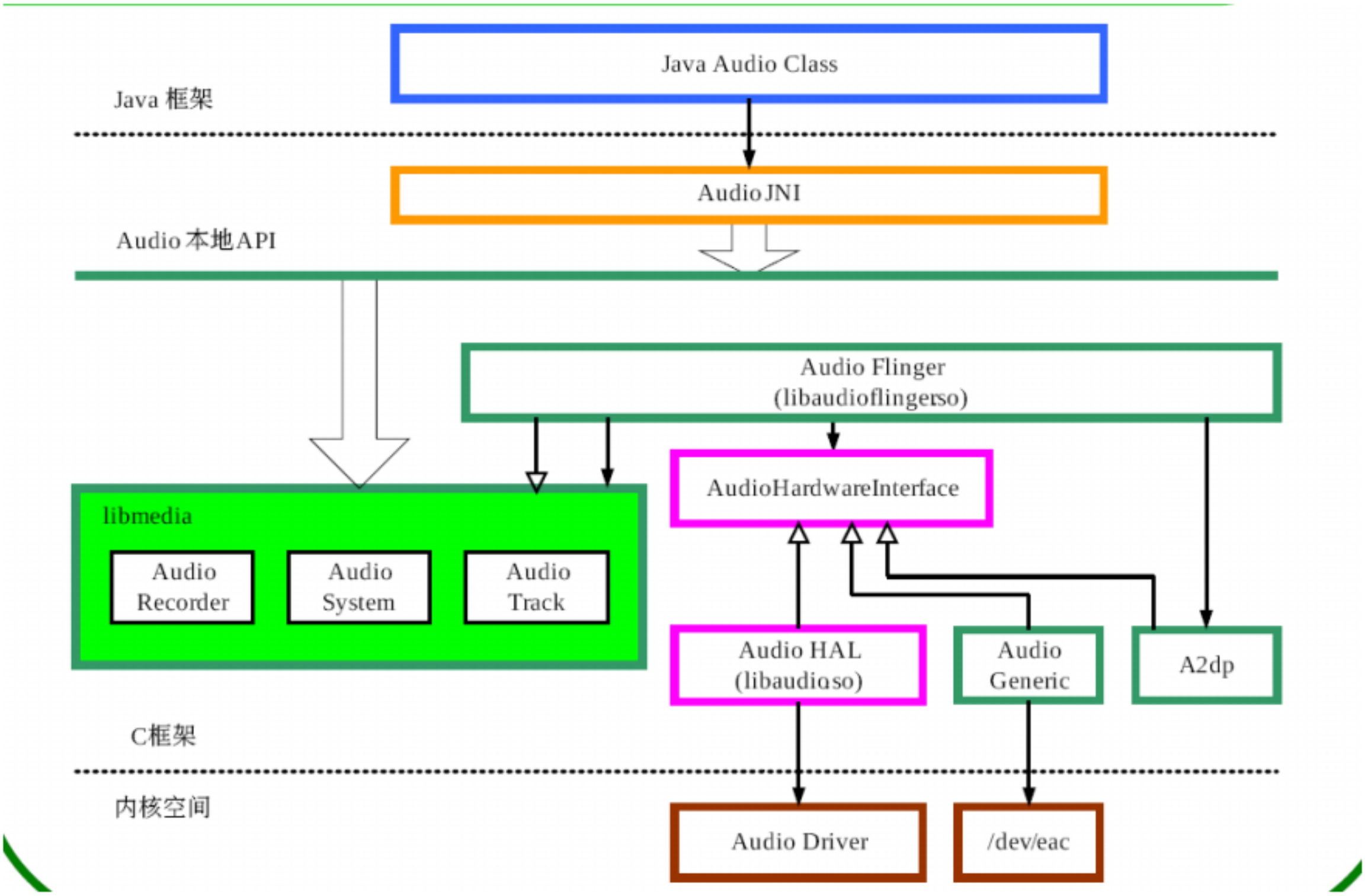

图 1 Android 的 Audio 系统结构

Android 的 Audio 系统的代码分布情况如下所示：

（1） Audio 的 Java 框架部分

代码路径： frameworks/base/media/java/android/media

与 Audio 相关的 Java 包是 android.media ，主要包含 AudioManager 和 Audio 系统的几个类。

（2） Audio 的 JNI 部分

代码路径： frameworks/base/core/jni

生成库 libandroid_runtime.so ， Audio 的 JNI 是其中的一个部分。

（3） Audio 的本地框架部分

头文件路径： frameworks/base/include/media/

源代码路径： frameworks/base/media/libmedia/

Audio 本地框架是 media 库的一部分，本部分内容被编译成库 libmedia.so ，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4） Audio Flinger

代码路径： frameworks/base/libs/audioflinger

这部分内容被编译成库 libaudioflinger.so ，它是 Audio 系统的本地服务部分。

（5） Audio 的硬件抽象层接口

头文件路径： hardware/libhardware_legacy/include/hardware_legacy/

C++文件路径： hardware/msm7k/libaudio/

Audio 硬件抽象层的实现在各个系统中可能是不同的，需要使用代码去继承相应的类并实现它们，作为 Android 系统本地框架层和驱动程序接口。

2.2 Audio 系统和上层接口

在 Android 中， Audio 系统自上而下由 Java 的 Audio 类、 Audio 本地框架类、 AudioFlinger 和 Audio 的硬件抽象层（HAL ）几个部分组成。

先看看 Audio 里边有哪些东西？通过 Android 的 SDK文档，发现主要有三个：

1) AudioManager ：这个主要是用来管理 Audio 系统的

2) AudioTrack ：这个主要是用来播放声音的

3) AudioRecord ：这个主要是用来录音的

其中 AudioManager 的理解需要考虑整个系统上声音的策略问题，例如来电话铃声，短信铃声等。一般看来，最简单的就是播放声音了。所以我们打算从 AudioTrack 开始分析。

2.2.1 Audio 系统的各个层次

Audio 系统的各层次情况如下所示。

Audio 本地框架类是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

AudioFlinger 继承 libmeida 中的接口 ，提供实现库 libaudiofilnger.so 。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so 。

Audio 使用 JNI 和 Java 对上层提供接口， JNI 部分通过调用 libmedia 库提供的接口来实现。

Audio 的硬件抽象层（ HAL ）提供到硬件的接口，供 AudioFlinger 调用。 Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

在 Android 的 Audio 系统中， 无论上层还是下层，都使用一个管理类和输出输入两个类来表示整个 Audio 系统，输出输入两个类负责数据通道。在各个层次之间具有对应关系，如表 1 所示。

	Audio 管理环节	Audio 输出	Audio 输入

Java 层	android.media.AudioSystem	android.media.AudioTrack	android.media.AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层（ HAL）	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

表 1 Android 各个层次的对应关系

2.2.2 media 库中的 Audio 框架部分

Android 的 Audio 系统的核心框架在 media 库中提供，对上面主要实现 AudioSystem 、AudioTrack 和 AudioRecorder 三个类。在 AudioSystem，AudioTrack 和 AudioRecorder 中，都可以获得 IAudioFlinger(sp<IAudioFlinger>& audioFlinger = AudioSystem::get_audio_flinger()) 。AudioTrack 和 AudioRecorder 通过 audioFlinger 创建 IAudioTrack 和 IAudioRecorder

提供了 IAudioFlinger 类接口，在这个类中，可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件在 frameworks/base/include/media/ 目录中，主要的头文件如下：

AudioSystem.h ：media 库的 Audio 部分对上层的总管接口；

IAudioFlinger.h ：需要下层实现的总管接口；

AudioTrack.h ：放音部分对上接口；

IAudioTrack.h ：放音部分需要下层实现的接口；

AudioRecorder.h ：录音部分对上接口；

IAudioRecorder.h ：录音部分需要下层实现的接口。

Ixxx 的接口通过 AudioFlinger 来实现，其他接口通过 JNI 向上层提供接口。

IAudioFlinger.h 、IAudioTrack.h 和 IAudioRecorder.h 这三个接口通过下层的继承来实现（即 AudioFlinger ）。AudioSystem.h 、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，它们既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。

meida 库中 Audio 部分的结构如图 2 所示。

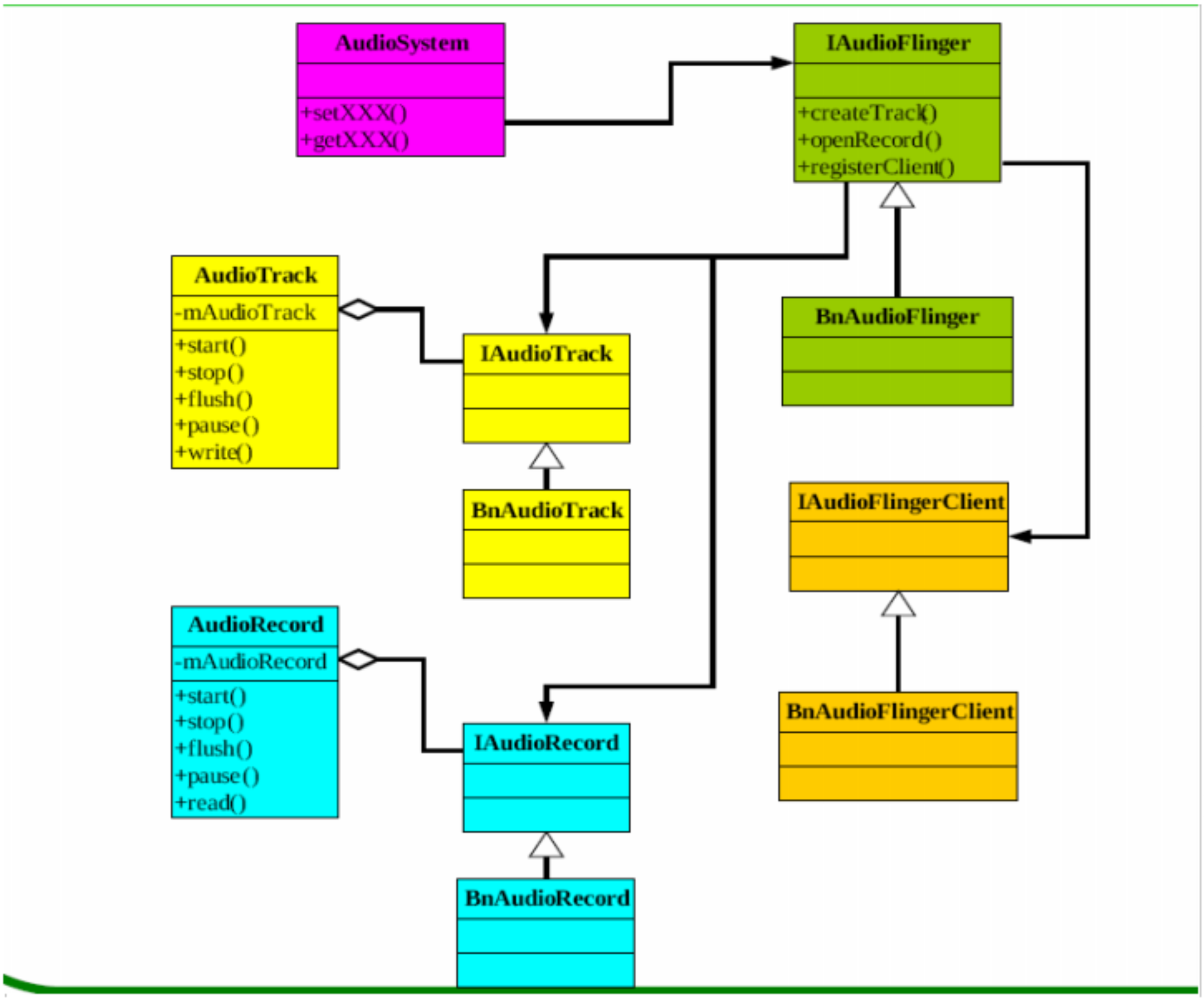

图 2 meida 库中 Audio 部分的结构

从功能上看，AudioSystem 负责的是 Audio 系统的综合管理功能，而 AudioTrack 和 AudioRecorder 分别负责音频数据的输出和输入，即播放和录制。

AudioSystem.h 中主要定义了一些枚举值和 set/get 等一系列接口。

在 Audio 系统的几个枚举值中， audio_routes 是由单独的位来表示的，而不是由顺序的枚举值表示，因此这个值在使用过程中可以使用 " 或"的方式。例如，表示声音可以既从耳机（ EARPIECE ）输出，也从扬声器（ SPEAKER ）输出，这样是否能实现，由下层提供支持。在这个类 中， set/get 等接口控制的也是相关的内容，例如 Audio 声音的大小、 Audio 的模式、路径等。

AudioTrack 是 Audio 输出环节的类，其中最重要的接口是 write() 。**********

AudioRecord 是 Audio 输入环节的类，其中最重要的接口为 read()。**********

AudioTrack 和 AudioRecord 的 read/write 函数的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（ PCM 数据）。这两个类还涉及 Auido 数据格式、通道数、帧数目等参数，可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

在 libmedia 库中提供的只是一个 Audio 系统框架， AudioSystem 、 AudioTrack 和 AudioRecord 分别调用下层的 IAudioFlinger 、IAudioTrack 和 IAudioRecord 来实现。另外的一个接口是 IAudioFlingerClient ，它作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IaudioFlinger 运行时信息。

AudioTrack 和 AudioRecorder 都具有 start, stop 和 pause 等接口。前者具有 write 接口，用于声音的播放，后者具有 read 接口，用于声音的录制。 AudioSystem 用于 Audio 系统的控制工作，主要包含一些 set 和 get 接口，是一个对上层的类 。

AudioSystem .h：

```
class AudioSystem
{
        public:
                enum stream_type { // Audio     流的类型
                        SYSTEM = 1,
                        RING = 2,
                        MUSIC = 3,
                        ALARM = 4,
                        NOTIFICA  TION = 5,
                        BLUETOOTH_SCO = 6,
                        ENFORCED_AUDIBLE = 7,
                        NUM_STREAM_TYPES
};
```

2.2.3 Audio Track

2.2.3.1 AudioTrack （Java 层）

JAVA的 AudioTrack 类的代码在：

framework\base\media\java\android\media\AudioTrack.java 中。

1. AudioTrack API 的使用例子

先看看使用例子，然后跟进去分析。

```
//      根据采样率，采样精度，单双声道来得到      frame 的大小。
int bufsize = AudioTrack.getMinBufferSize(8000,                    // 每秒 8000 个点 ( 采样率 )
                AudioFormat.CHANNEL_CONFIGURATION_STEREO,          // 双声道
                AudioFormat.ENCODING_PCM_16BIT);                   // 一个采样点 16 比特 -2 个字节
//      注意，按照数字音频的知识，这个算出来的是      一秒钟 buffer 的大小 。
//      创建 AudioTrack
AudioTrack trackplayer =            new AudioTrack (AudioManager.STREAM_MUSIC, 8000,
                AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
                AudioFormat.ENCODING_PCM_16BIT,
                        bufsize ,
              AudioTrack.MODE_STREAM);
```

trackplayer.play() ;// 开始

trackplayer.write(bytes_pkg, 0, bytes_pkg.length) ;// 往 track 中写数据

…:

trackplayer.stop();// 停止播放

trackplayer.release();// 释放底层资源。

这里需要解释：

AudioTrack 中有 MODE_STATIC 和 MODE_STREAM 两种分类。

1) STREAM 的意思是由用户在应用程序通过 write 方式把数据一次一次得写到 audiotrack 中。
这种方式的坏处就是总是在 JAVA 层和 Native 层交互，效率损失较大。

2) STATIC 的意思是一开始创建的时候，就把 音频数据放到一个固定的 buffer ，然后直接传给 audiotrack ，后续就不用一次次得 write 了。AudioTrack 会自己播放这个 buffer 中的数据。
这种方法对于铃声等 内存占用较小，延时要求较高的声音来说很适用 。

2. 分析之 getMinBufferSize // 注意，这是个 static 函数

```
AudioTrack.getMinBufferSize(8000,// 每秒 8K 个点
        AudioFormat.CHANNEL_CONFIGURATION_STEREO,// 双声道
        AudioFormat.ENCODING_PCM_16BIT);
```

// 调用 native 函数

native_get_min_buff_size---> 在 framework/base/core/jni/android_media_track.cpp 中实现。

// 音频中最常见的是 frame 这个单位 ------1 个采样点的字节数 * 声道 。

getMinBufSize 函数完了后，我们得到一个满足最小要求的缓冲区大小。

3. 分析之 new AudioTrack 先看看调用函数： (java)

```
AudioTrack trackplayer = new AudioTrack(
        AudioManager.STREAM_MUSIC, 8000,
            AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
        AudioFormat.ENCODING_PCM_16BIT,
        bufsize,
                AudioTrack.MODE_STREAM);// 一次一次写入
```

// 调用 native 层的 native_setup ，把 WeakReference 传进去了

int initResult = native_setup (new WeakReference<AudioTrack>(this).....

上面函数调用最终进入了 JNI 层 android_media_AudioTrack.cpp 函数，其中有

AudioTrackJniStorage* lpJniStorage = new AudioTrackJniStorage();

jclass clazz = env->GetObjectClass(thiz);

lpJniStorage->mCallbackData.audioTrack_class = (jclass)env->NewGlobalRef(clazz);

lpJniStorage->mCallbackData.audioTrack_ref = env->NewGlobalRef(weak_this);

lpJniStorage->mStreamType = atStreamType;

// 创建真正的 AudioTrack 对象

AudioTrack* lpTrack = new AudioTrack();

if (memoryMode == javaAudioTrackFields.MODE_STREAM) {

// 如果是 STREAM 流方式的话，把刚才那些参数设进去 (AudioTrack.cpp 的 set())

lpTrack->set(.... audioCallback , //callback

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

0, // 共享内存， STREAM 模式需要用户一次次写，所以就不用共享内存了

…..

} else if (memoryMode == javaAudioTrackFields.MODE_STATIC) {

// static 模式，需要用户一次性把数据写进去，然后由 audioTrack 把数据读出来 ，

// 所以需要一个共享内存 ， 指 C++AudioTrack 和 AudioFlinger 之间共享的内容

// 因为真正播放的工作是由 AudioFlinger 来完成的。

lpJniStorage-> allocSharedMem (buffSizeInBytes); //native AudioTrack 分配共享内存？

lpTrack->set(…; audioCallback, //callback

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

lpJniStorage->mMemBase,// shared mem

…:)

}

// 这样， Native 层的 AudioTrack 对象就和 JAVA层的 AudioTrack 对象关联起来了。

env->SetIntField(thiz, javaAudioTrackFields.nativeTrackInJavaObj, (int)lpTrack);

env->SetIntField(thiz, javaAudioTrackFields.jniData, (int)lpJniStorage);

1) AudioTrackJniStorage 详解

这个类其实就是一个辅助类，但是里边有一些知识很重要，尤其 Android 封装的一套共享内存的机制。把这块搞清楚了，我们就能轻松得在两个进程间进行内存的拷贝。

```
struct audiotrack_callback_cookie {
    jclass   audioTrack_class;
    jobject  audioTrack_ref;
};  cookie          其实就是把 JAVA中的一些东西保存了下，没什么特别的意义
class AudioTrackJniStorage {
public:
                sp<MemoryHeapBase>  mMemHeap;  //  这两个  Memory很重要
    sp<MemoryBase>  mMemBase;
    audiotrack_callback_cookie mCallbackData;
    int   mStreamType;
    bool                allocSharedMem  (int sizeInBytes) {
        mMemHeap = new MemoryHeapBase(sizeInBytes, 0, "...");
        mMemBase = new MemoryBase(mMemHeap, 0, sizeInBytes);
      //                注意用法，先弄一个   HeapBase，再把 HeapBase传入到 MemoryBase中去。
        return true;
    }
};
```

2) MemoryHeapBase

MemroyHeapBase是 Android 的一套基于 Binder 机制的对内存操作的类 。既然是 Binder 机制，那么肯定有一个服务端（ Bnxxx），一个代理端 Bpxxx。看看 MemoryHeapBase定义：

```
class MemoryHeapBase :        public virtual BnMemoryHeap
//      果然，从 BnMemoryHeap派生，那就是 Bn端。这样就和 Binder 挂上钩了
//      Bp 端调用的函数最终都会调到    Bn 这来
```

MemoryHeapBase提供了几个函数，可以获取共享内存的大小和位置 。

getBaseID ()---> 返回 mFd，如果为负数，表明刚才创建共享内存失败了

getBase() -> 返回 mBase，内存位置

getSize() -> 返回 mSize，内存大小

有了 MemoryHeapBase，又搞了一个 MemoryBase，这又是一个和 Binder 机制挂钩的类。这个类就是一个能返回当前 Buffer 中写位置（就是 offset ）的方便类

class MemoryBase : public BnMemory

明白上面两个 MemoryXXX，我们可以猜测下大概的使用方法了。

(1)BnXXX 端先分配 BnMemoryHeapBase和 BnMemoryBase，

(2) 然后把 BnMemoryBase传递到 BpXXX

(3)BpXXX 就可以使用 BpMemoryBase得到 BnXXX端分配的共享内存了。

注 意， 既然是进程间共享内存，那么 Bp 端肯定使用 memcpy之类的函数来操作内存 ，这些函数是没有同步保护的，而且 Android 也不可能在系统内部为这 种共享内存去做增加同步保护。所以看来后续在操作这些共享内存的时候，肯定存在一个跨进程的同步保护机制。我们在后面讲实际播放的时候会碰到。

另外， 这里的 SharedBuffer 最终会在 Bp 端也就是 AudioFlinger 那用到 。

3) 分析之 play 和 write

JAVA层就是调用 play 和 write 了。 JAVA层这两个函数没什么内容，直接转到 native 层干活了。

先看看 play 函数对应的 JNI 函数

```
static void android_media_AudioTrack_start(JNIEnv *env, jobject thiz) {
    //      看见没，从 JAVA那个 AudioTrack 对象获取保存的 C++层的 AudioTrack 对象指针
    AudioTrack *lpTrack = (AudioTrack *)env->GetIntField(
            thiz, javaAudioTrackFields.nativeTrackInJavaObj);
    lpTrack->start(); //          这个以后再说
}
```

下面是 write 。我们写的是 short 数组，

```
static jint
android_media_AudioTrack_native_write_short(...) {
    return (android_media_AudioTrack_native_write()/ 2);
}
```

根据 Byte 还是 Short 封装了下，最终会调到重要函数 writeToTrack

JAVA层的 AudioTrack ，无非就是调用 write 函数，而实际由 JNI 层的 C++ AudioTrack write 数据。

2.2.3.2 AudioTrack （C++ 层）

接上面的内容，我们知道在 JNI 层，有以下几个步骤：

1) new AudioTrack

2) 调用 set() ，把 AudioTrackJniStorage 等信息传进去

3) 调用了 AudioTrack->start()

4) 调用 AudioTrack->write()

那么，我们就看看真正干活的的 C++AudioTrack 吧。

AudioTrack.cpp 位于 AudioTrack.cpp

1. new AudioTrack() 和 set 调用 JNI 层调用的是最简单的构造函数：

```
AudioTrack::AudioTrack()
    :mStatus(NO_INIT) //          把状态初始化成 NO_INIT。Android 大量使用了设计模式中的 state 。
{
}
```

接下来调用 set 。我们看看 JNI 那 set 了什么

```
lpTrack->set(
        atStreamType, //              应该是 Music 吧
        sampleRateInHertz,  //8000
        format,  //                  应该是 PCM_16吧
        channels,  //                立体声 =2
        frameCount, //
        0,  // flags
        audioCallback,               //JNI 中的一个回调函数
        &(lpJniStorage->mCallbackData),          // 回调函数的参数
        0, //                 通知回调函数，表示 AudioTrack 需要数据，不过暂时没用上
        0,              // 共享 buffer 地址， stream 模式没有
        true);//                回调线程可以调 JAVA的东西
```

set 函数详解 ：#######三个点： (1)getOutput (2)createTrack (3)new AudioTrackThread

```
status_t AudioTrack::set() {
    ...     前面一堆的判断，等以后讲 AudioSystem 再说
    audio_io_handle_t output =
            AudioSystem::               getOutput()  ;
```

```
//createTrack                    ? 看来这是真正干活的
status_t status =                      createTrack   (...., output);
//cbf                是 JNI 传入的回调函数    audioCallback
if (cbf != 0) { //                     看来，怎么着也要创建这个线程了！
        mAudioTrackThread =                        new AudioTrackThread   (*this, threadCanCallJava);
}
```

看看真正干活的 createTrack

```
status_t AudioTrack::createTrack() {
        // 和 audioFlinger    挂上关系
    const sp<IAudioFlinger>& audioFlinger = AudioSystem::get_audio_flinger();
    sp<IAudioTrack> track = audioFlinger->createTrack();
        // 从 track  也就是 AudioFlinger   那边得到一个  IMemory 接口， 这个是最终  write  写入的地方
        sp<IMemory> cblk = track->getCblk();          //  从 audioFlinger    得到 buffer  ? YES!!!!
    mCblkMemory.clear();  //sp<XXX>                的 clear ， 就看着做是  delete XXX  吧
    mCblkMemory = cblk;
        mCblk = static_cast<audio_track_cblk_t*>(cblk->pointer());
    if (sharedBuffer == 0) {                    //  共享内存
        //buffer                相关。 STREAM模式没有传入共享  buffer ， 但是数据确实需要  buffer  承载。
                //AudioTrack   是没有创建  buffer ， 那只能是刚才从  AudioFlinger   中得到的  buffer  了。
            mCblk->buffers =                (char*)mCblk + sizeof(audio_track_cblk_t)              ;
        } else {
            mCblk->buffers =                sharedBuffer->pointer()       ;
            mCblk->stepUser(mCblk->frameCount);
        }
    }
```

MemoryXXX 没有同步机制，所以应该有一个东西能体现同步的，在 audio_track_cblk_t 结构中。它的头文件在 framework/base/include/private/media/AudioTrackShared.h

AudioTrack 得到 AudioFlinger 中的一个 IAudioTrack 对象，这里边有一个 audio_track_cblk_t ，它包括

一块缓冲区地址，包括一些进程间同步的内容，可能还有数据位置等内容。 AudioTrack 启动了一个线程，叫 AudioTrackThread 。AudioTrack 调用 write 函数，肯定是把数据写到那块共享缓冲了，然后 IAudioTrack 在另外 一个进程 AudioFlinger 中接收数据，并最终写到音频设备中。

那 AudioTrackThread 干什么了。

mAudioTrackThread = new AudioTrackThread (*this, threadCanCallJava);

反正最终会调用 AudioTrackAThread 的 threadLoop 函数。

先看看构造函数

```
AudioTrackThread(AudioTrack& receiver, bool bCanCallJava)
    : Thread(bCanCallJava), mReceiver(receiver)
    {    //          mReceiver 就是 AudioTrack 对象
        // bCanCallJava          为 TRUE
    }
```

AudioTrackThread 的启动由 AudioTrack 的 start 函数触发。

```
void AudioTrack::start() {
    //start          调用 AudioTrackThread 函数，执行 mAudioTrackThread 的 threadLoop
    sp<AudioTrackThread> t = mAudioTrackThread;
    t->          run("AudioTrackThread", THREAD_PRIORITY_AUDIO_CLIENT) ;
    //       让 AudioFlinger 中的 track 也 start
    status_t status =          mAudioTrack->start()      ;
}
bool AudioTrack::AudioTrackThread::threadLoop() {
        //          调用 AudioTrack 的 processAudioBuffer 函数
        return mReceiver.processAudioBuffer(this);
    }
```

难道真的有两处在 write 数据？看来必须得到 mCbf去看看了，传的是 EVENT_MORE_DATA 标志。

mCbf 由 set 的时候传入 C++的 AudioTrack ，实际函数是：

```
static void audioCallback(int event, void* user, void *info) {
    if (event == AudioTrack::EVENT_MORE_DATA) {
    //      这个函数没往里边写数据
    AudioTrack::Buffer* pBuff = (AudioTrack::Buffer*)info;
```

```
        pBuff->size = 0;
    }
```

看来就只有用户的 write 会真正的写数据了，这个 AudioTrackThread 除了通知一下，也没什么实际有意义的操作了。

2. writessize_t AudioTrack::write(const void* buffer, size_t userSize)

够简单，就是 obtainBuffer ，memcpy数据，然后 releasBuffer

2.2.3.3 AudioTrack 总结

看起来，最重要的工作是在 AudioFlinger 中做的。

1) AudioTrack 被 new 出来，然后 set 了一堆信息，同时会 通过 Binder 机制调用另外一端的 AudioFlinger ，得到 IAudioTrack 对象，通过它和 AudioFlinger 交互 。

2) 调用 start 函数后，会启动一个线程专门做 回调处理，代码里边也会有那种数据拷贝的回调，但是 JNI 层的回调 函数实际并没有往里边写数据，大家只要看 write 就可以了

3) 用户一次次得 write ，那 AudioTrack 无非就是把数据 memcpy到共享 buffer 中。可想而知， AudioFlinger 那一定有一个线程在 memcpy数据到音频设备中去。我们拭目以待。

2.2.4 AudioFlinger 本地代码

AudioFlinger 是 Audio 系统的中间层，在系统中起到服务作用，它主要作为 libmedia 提供的 Audio 部分接口的实现，其代码路径为：

frameworks/base/libs/audioflinger

AudioFlinger 的核心文件是 AudioFlinger.h 和 AudioFlinger.cpp ，提供了类 AudioFlinger ，这个类是一个 IAudioFlinger 的实现。

AudioFlinger 主要提供 createTrack() 创建音频的输出设备 IAudioTrack ， openRecord() 创建音频的输入设备 IAudioRecord 。另外包含的就是一个 get/set 接口，用于控制。

从工作的角度看， AudioFlinger 在初始化之后，首先获得放音设备，然后为混音器（ Mixer ）建立线程，接着建立放音设备线程，在线程中获得放音设备。 (不是很理解)

在 AudioFlinger 的 AudioResampler.h 中定义了一个音频重取样器工具类。这个音频重取样工具包含 3 种质量：低等质量（ LOW_QUALITY ）将使用线性差值算法实现；中等质量（ MED_QUALITY ）将使用立方差值算法实现；高等质量（ HIGH_ QUALITY ）将使用 FIR（有限阶滤波器）实现。 AudioResampler 中的 AudioResamplerOrder1 是线性实现， AudioResamplerCubic.* 文件提供立方实现方式， AudioResamplerSinc.* 提供 FIR 实现。

AudioMixer.h 和 AudioMixer.cpp 中实现的是一个 Audio 系统混音器，它被 AudioFlinger 调用，一般用于在声音输出之前的处理，提供多通道处理、声音缩放、重取样。 AudioMixer 调用了 AudioResampler 。

AudioFlinger 本身的实现通过调用下层的 Audio 硬件抽象层（ HAL ）的接口来实现具体的功能，各个接口之间具有对应关系。

2.2.4.1 目的

AudioTrack 作为 AF（AudioFlinger ）的客户端，来看看 AF 是如何完成工作的。在 AT（AudioTrack ）中，涉及到的都是流程方面的事务，而不是系统 Audio 策略上的内容。因为 AT 是 AF 的客户端，而 AF 是 Android 系统中 Audio 管理的中枢。对于分析 AT 来说，只要能把它的调用顺序（也就是流程说清楚就可以了），但是对于 AF 的话，简单的分析调用流程 是不够的。因为手机上的声音交互和管理是一件比较复杂的事情。举个简单例子，当听 music 的时候来电话了，声音处理会怎样？

虽然在 Android 中，还有一个叫 AudioPolicyService 的（APS）东西，但是它最终都会调用到 AF 中去，因为 AF 实际创建并管理了硬件设备。

2.2.4.2 从 AT 切入到 AF

1. AudioFlinger 的诞生

AF 是一个服务，代码在 framework/base/media/mediaserver/Main_mediaServer.cpp 中。

```
int main(int argc, char** argv)
{
        …..
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
        …..
}
```

为何 AF, APS 要和 MediaService 和 CameraService 都放到一个篮子里？看看 AF 的实例化静态函数，在 framework/base/libs/audioFlinger/audioFlinger.cpp 中

```
void AudioFlinger::instantiate() {
    defaultServiceManager()->addService(
        String16("media.audio_flinger"), new AudioFlinger() );
}
```

再来看看它的构造函数是什么做的。

```
AudioFlinger::AudioFlinger(): BnAudioFlinger(), // 初始化基类
```

```
    mAudioHardware(0),                    //audio 硬件的 HAL 对象
    mMasterVolume(1.0f),                  mMasterMute(false),          mNextThreadId(0)          {
mHardwareStatus                           =                            AUDIO_HW_IDLE;
// 创建代表 Audio 硬件的 HAL 对象
mAudioHardware = AudioHardwareInterface::create();  // 和硬件挂钩
mHardwareStatus                           =                            AUDIO_HW_INIT;
if (mAudioHardware->initCheck() == NO_ERROR) {
  setMode(AudioSystem::MODE_NORMAL);
  // 设置系统的声音模式等，其实就是设置硬件的模式
  setMasterVolume(1.0f);
  setMasterMute(false);
}
  }
```

AF 中经常有 setXXX 的函数，到底是干什么的呢？我们看看 setMode 函数。

```
  status_t AudioFlinger::setMode(int mode) {
    mHardwareStatus = AUDIO_HW_SET_MODE;
    status_t ret = mAudioHardware->setMode (mode);// 设置硬件的模式
    mHardwareStatus = AUDIO_HW_IDLE;
    return ret;
  }
```

当然， setXXX 还有些别的东西，但基本上都会涉及到硬件对象。 Android 系统启动的时候，看来 AF 也准备好硬件了。不过，创建硬件对象就代表可以播放了吗？

2. AT 调用 AF的流程

我这里简单的把 AT调用 AF的流程列一下，待会按这个顺序分析 AF的工作方式。

1) 创建

```
AudioTrack* lpTrack = new AudioTrack();  //JNI
lpTrack->set(...);
```

这个就进入到 C++的 AT 了。下面是 AT的 set 函数

```
  audio_io_handle_t output =
          AudioSystem::getOutput ((AudioSystem::stream_type)streamType,...);
  status_t status = createTrack(streamType, sampleRate, ..., sharedBuffer, output);
----->creatTrack 会和 AF 打交道。我们看看 createTrack 重要语句
  const sp &audioFlinger = AudioSystem::get_audio_flinger();
  // 下面很重要，调用 AF的 createTrack 获得一个 IAudioTrack 对象
  sp<IAudioTrack> track = audioFlinger->createTrack();
  sp<IMemory> cblk = track->getCblk();  // 获取共享内存的管理结构
```

总结一下创建的流程， AT 调用 AF 的 createTrack 获得一个 IAudioTrack 对象，然后从这个对象中获得共享内存的对象。

2) 2. start 和 write

看看 AT的 start ，估计就是调用 IAudioTrack 的 start 吧？

```
  void AudioTrack::start() {
```

status_t status = mAudioTrack->start();

}

那 write 呢?我们之前讲了， AT就是从共享 buffer 中:

l Lock 缓存

l 写缓存

l Unlock 缓存

注意，这里的 Lock 和 Unlock 是有问题的，什么问题呢 ?待会我们再说。按这种方式的话，那么 AF 一定是有一个线程在那也是：

l Lock ，

l 读缓存，写硬件

l Unlock

总之，我们知道了 AT 的调用 AF 的流程了。下面一个一个看。

3. AF流程

1) 1 createTrack--- 真正干活的地方

```
sp<IAudioTrack> AudioFlinger::createTrack(
    pid_t pid,   //AT                  的 pid 号(client)
    int streamType,     //MUSIC                  ，流类型
    uint32_t sampleRate,   //8000                  采样率
    int format,     //PCM_16                  类型
    int channelCount,      //2                  ，双声道
    int frameCount,      //                  需要创建的 buffer 可包含的帧数
    uint32_t flags,
    const sp& sharedBuffer,      //AT                  传入的共享 buffer ，这里为空 (Stream 模式 )
    int output,      //                  这个是从 AuidoSystem 获得的对应 MUSIC流类型的索引
    status_t *status)
      {
 {
  Mutex::Autolock _l(mLock);
  //            获得线程？ //  在 openOutput() 中 new MixerThread(mPlaybackThread.add)
  PlaybackThread *thread = checkPlaybackThread_l(output);
  //            看看这个进程是不是已经是 AF 的 client 了
     DefaultKeyedVector< pid_t pid, wp<Client> > mClients;  //map
  //            这里说明一下，由于是 C/S 架构，那么作为服务端的 AF 肯定有地方保存作为 C的 AT 的信息
  //            那么， AF 是根据 pid 作为客户端的唯一标示的 mClients 是一个类似 map的数据组织结构
  wclient = mClients.valueFor(pid);
  if (wclient != NULL) {
  } else {
```

```
        //              如果还没有这个客户信息，就创建一个，并加入到          map中去
        client = new Client(this, pid);
        mClients.add(pid, client);
    }
    //              从刚才找到的那个线程对象中创建一个          track
    track = thread->createTrack_l(client, streamType, sampleRate, format,
            channelCount, frameCount, sharedBuffer, &lStatus);
}
    //          还有一个  trackHandle  ，而且返回到  AT 端的是这个  trackHandle  对象
    trackHandle = new TrackHandle(track);
    return trackHandle;
}
```

这个 AF 函数中，突然冒出来了很多新类型的数据结构。先进入到 checkPlaybackThread_l 看看。

```
AudioFlinger::PlaybackThread *AudioFlinger::checkPlaybackThread_l(int output) const
{     DefaultKeyedVector< int output, sp<PlaybackThread> >  mPlaybackThreads;
    PlaybackThread *thread = NULL;
    //          看到  indexOfKey  的东西，应该想到：这可能是一个          map之类的东西，根据    key  能找到  value
    if (mPlaybackThreads.indexOfKey(output) >= 0) {
        thread = (PlaybackThread *)mPlaybackThreads.valueFor(output).get();
    } ##############//openOutput()              才会执行  mPlaybackThreads.add()       注意
    //          这个函数的意思是根据      output  值，从一堆线程中找到对应的那个线程
    return thread;
}
```

看到这里很疑惑啊：

l AF 的构造函数中没有创建线程，只创建了一个 audio 的 HAL对象

| 如果 AT是 AF 的第一个客户的话，刚才的调用流程里边，也没看到创建线程的地方呀。

l output 是个什么玩意儿？为什么会根据它作为 key 来找线程呢？

看来，我们得去 Output 的来源那看看了。 output 的来源是由 AT 的 set 函数得到的：如下：

```
audio_io_handle_t output = AudioSystem::getOutput(streamType, ...);
```

然后调用 AT 自己的 createTrack ，最终把这个 output 值传递到 AF 了。其中 audio_io_handle_t 类型就是一个 int 类型。

进到 AudioSystem::getOutput 看看。注意，这是系统的第一次调用 , 而且发生在 AudioTrack 那个进程里边。AudioSystem 的位置在 framework/base/media/libmedia/AudioSystem.cpp 中

```
audio_io_handle_t AudioSystem::getOutput(stream_type stream,...) {
    audio_io_handle_t output = 0;
    if ()
{
        Mutex::Autolock _l(gLock);
    //          根据我们的参数，会走到这个里边 .  又是从 map中找到 stream=music 的 output 。
    //          可惜啊，我们是第一次进来 , output 一定是 0
    output = AudioSystem::gStreamOutputMap.valueFor(stream);
}
if (output == 0) {
    //          到 audioPolicyService(APS),  由它去 getOutput #####  重点啊！！！！！
    const sp<>& aps = AudioSystem::get_audio_policy_service();
    output = aps->getOutput(stream, samplingRate, format, channels, flags);
    if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) == 0) {
        Mutex::Autolock _l(gLock);
        //          如果取到 output 了，再把 output 加入到 AudioSystem 维护的这个 map中去
        //          说白了，就是保存一些信息吗。免得下次又这么麻烦去骚扰 APS!
            AudioSystem::gStreamOutputMap.add(stream, output);
        }
    }
    return output;
}
```

需要到 APS 中才能找到 output 的信息？那先得看看 APS 是如何创建的。刚才看到是和 AF 一块在 Main_mediaService.cpp 中实例化的。

```
AudioPolicyService::AudioPolicyService()
        : BnAudioPolicyService() , mpPolicyManager(NULL)  {
```

```
mTonePlaybackThread = new AudioCommandThread(String8(""));
mAudioCommandThread = new AudioCommandThread(
                String8("ApmCommandThread"));
#if (defined GENERIC_AUDIO) || (defined AUDIO_POLICY_TEST)
    //          使用普适的  AudioPolicyManager  ，把自己  this  做为参数
    mpPolicyManager = new AudioPolicyManagerBase(this);
    //          使用硬件厂商提供的特殊的     AudioPolicyManager
    //mpPolicyManager = createAudioPolicyManager(this);
    }
}
```

看看 AudioManagerBase 的构造函数吧。

```
AudioPolicyManagerBase(AudioPolicyClientInterface *clientInterface)
        : mPhoneState(AudioSystem::MODE_NORMAL), mRingerMode(0),
          mMusicStopTime(0),  mLimitRingtoneVolume(false)
{
    mpClientInterface = clientInterface; //                  这个 client   就是 APS，刚才通过  this  传进来了
    AudioOutputDescriptor *outputDesc = new AudioOutputDescriptor();
    outputDesc->mDevice = (uint32_t)AudioSystem::DEVICE_OUT_SPEAKER;
    mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,...);
    // openOutput          又交给 APS的 openOutput 来完成了，真绕  ....
}
```

还是得回到 APS

```
audio_io_handle_t AudioPolicyService::openOutput(uint32_t *pDevices,                ··) {
    sp af = AudioSystem::get_audio_flinger();
    //      绕了这么一个大圈子，竟然回到     AudioFlinger   中了啊？？
    return af->openOutput(pDevices, pSamplingRate, ...);  //device                是 speaker
}
```

在我们再次被绕晕之后，我们回眸看看足迹吧：

1) 在 AudioTrack 中，调用 set 函数

2) 这个函数会通过 AudioSystem::getOutput 来得到一个 output 的句柄

3) AS 的 getOutput 会调用 AudioPolicyService 的 getOutput

4) APS创建的时候会创建一个 AudioManagerBase，AMB的创建又会调用 APS的 openOutput 。

5) APS的 openOutput 又会调用 AudioFlinger 的 openOutput

有一个疑问， AT 中 set 参数会和 APS构造时候最终传入到 AF 的 openOutput 一样吗？如果不一样，那么构造时候 openOutput 的又是什么参数呢？应该不一样把？？？

先放下这个悬念，我们继续从 APS的 getOutPut 看看。

```
audio_io_handle_t AudioPolicyService::getOutput(AudioSystem::stream_type stream,...)
{
    Mutex::Autolock _l(mLock);
    //          自己又不干活，由    AudioManagerBase 干活
    return mpPolicyManager->getOutput(stream, samplingRate, format, channels, flags);
}
```

进去看看吧

```
audio_io_handle_t AudioPolicyManagerBase::getOutput(AudioSystem::stream_type,...)
{
    // open a non direct output
    output = mHardwareOutput; //                这个是在哪里创建的？在      AMB构造的时候  ..
    return output;
}
```

具体 AMB的分析待以后 Audio 系统策略的时候我们再说吧。在 APS构造的时候会 open 一个 Output，而这个 Output 又会调用 AF的 openOutput 。

```
int AudioFlinger::openOutput(uint32_t *pDevices,                    ··) {
    ....
    Mutex::Autolock _l(mLock);
    //          由 Audio 硬件 HAL对象创建一个 AudioStreamOut 对象
    AudioStreamOut *output = mAudioHardware->openOutputStream(*pDevices, ...);
    mHardwareStatus = AUDIO_HW_IDLE;
    if (output != 0) {
```

```
    //          创建一个 Mixer 线程
    //          启机后， device 默认 speaker ， 创建 new MixerThread
    //new AudioTrack          ， set(), getOutput, openoutput, new MixerThread
    thread = new MixerThread(this, output, ++mNextThreadId);
}
//      终于找到了，把这个线程加入线程管理组织中
mPlaybackThreads.add(mNextThreadId, thread);
return mNextThreadId;
}
```

看来 AT 在调用 AF 的 createTrack 的之前， AF 已经在某个时候把线程创建好了，而且是一个 Mixer 类型的线程，看来和混音有关系呀。 这个似乎和我们开始设想的 AF 工作有点联系喔。 Lock，读缓存，写 Audio 硬件， Unlock 。可能都是在这个线程里边做的。

2) 继续 createTrack

```
AudioFlinger::createTrack(pid_t pid,..., const sp & sharedBuffer, int output,...) {
    ...
    {
        //          假设我们找到了对应的线程
        Mutex::Autolock _l(mLock);
        PlaybackThread *thread = checkPlaybackThread_l(output);
        //          晕，调用这个线程对象的 createTrack_l
        track = thread->createTrack_l(client, streamType, sampleRate, format,
                channelCount, frameCount, sharedBuffer, &lStatus);
    }
    trackHandle = new TrackHandle(track);
    return trackHandle          ；// 注意，这个对象是最终返回到 AT进程中的。
}
```

进去看看 thread->createTrack_l 吧。 _l 的意思是这个函数进入之前已经获得同步锁了。

```
AudioFlinger::PlaybackThread::createTrack_l() {
    {   // scope for mLock
        Mutex::Autolock _l(mLock);
```

//new 一个 track 对象， 注意 sharedBuffer ，此时的值应是 0

track = new Track(this, client, ..., sharedBuffer);

mTracks.add(track); // 把这个 track 加入到数组中，是为了管理用的。

}

lStatus = NO_ERROR;

return track;

}

看到这个数组的存在，我们应该能想到什么吗？这时已经有：

一个 MixerThread ，内部有一个数组保存 track 的 //MixerThread 继承自 playbackThread

看来，不管有多少个 AudioTrack ，最终在 AF 端都有一个 track 对象对应，而且这些所有的 track 对象都会由一个线程对象来处理。 ---- 难怪是 Mixer 啊#######（最多有 32 个）

再去看看 new Track ，我们一直还没找到共享内存在哪里创建的！！！

PlaybackThread::Track::Track() {

//mCblk !=NULL? 什么时候创建的？？只能看基类 TrackBase

if (mCblk != NULL) {

mVolume[0] = 1.0f;

mVolume[1] = 1.0f;

mStreamType = streamType;

mCblk->frameSize = AudioSystem::isLinearPCM(format) ? channelCount *

sizeof(int16_t) : sizeof(int8_t);

}

}

看看基类 TrackBase 干嘛了

TrackBase::TrackBase() {

size_t size = sizeof(audio_track_cblk_t);

size_t bufferSize = frameCount*channelCount*sizeof(int16_t);

if (sharedBuffer == 0) {

size += bufferSize;

}

// 调用 client 的 allocate 函数。 Client 是在 CreateTrack 中创建的 Client, 会创建一块共享内存

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

// 有了共享内存，但是还没有里边有同步锁的那个对象 audio_track_cblk_t

mCblk = static_cast<audio_track_cblk_t *>(mCblkMemory->pointer());

new(mCblk) audio_track_cblk_t();

// 这就是 C++语法中的 placement new 。new 后面的括号中是一块 buffer ，再后面是一个类的构造函数。placement new 就是在这块 buffer 中构造一个对象。普通 new 是没法让一个对象在某块指定的内存中创建的。这样，申请了一块共享内存，再在这块内存上创建一个对象。这样，这个对象不也就能在两个内存中共享了吗？怎么想到的？

// clear all buffers

mCblk->frameCount = frameCount;

mCblk->sampleRate = sampleRate;

mCblk->channels = (uint8_t)channelCount;

}

解决一个重大疑惑，跨进程数据共享数据结构 audio_track_cblk_t 是通过 placement new 在一块共享内存上来创建的。

回到 AF的 CreateTrack ，有这么一句话： trackHandle = new TrackHandle(track);

return trackHandle ；// 注意，这个对象是最终返回到 AT 进程中的。

trackHandle 的构造使用了 thread->createTrack_l 的返回值。

3. 到底有多少种对象

读到这里的人，一定会被异常多的 class 类型，内部类，继承关系搞疯掉。

1) AudioFlinger

class AudioFlinger : public BnAudioFlinger, public IBinder::DeathRecipient

AudioFlinger 类是代表整个 AudioFlinger 服务的类，其余所有的工作类都是通过内部类的方式在其中定义的。你把它当做一个壳子。

2) Client

Client 是描述 C/S 结构的 C 端的代表，也就算是一个 AT 在 AF 端的对等物吧。不过可不是 Binder 机制中的 BpXXX 喔。因为 AF是用不到 AT 的功能的。

class Client : public RefBase {

public:

sp<AudioFlinger> mAudioFlinger; // 代表 S 端的 AudioFlinger

sp<MemoryDealer> mMemoryDealer; // 每个 C 端使用的共享内存，通过它分配

pid_t mPid; //C 端的进程 id

};

3) TrackHandle

Trackhandle 是 AT 端调用 AF的 CreateTrack 得到的一个基于 Binder 机制的 Track 。TrackHandle 实际上是对真正干活的 PlaybackThread::Track 的跨进程支持的封装。本来 PlaybackThread::Track 是真正在 AF中干活的东西，不过为了支持跨进程，用 TrackHandle 对其进行了包转。这样在 AudioTrack 调用 TrackHandle 的功能，实际都由 TrackHandle 调用 PlaybackThread::Track 来完成了。可以认为是一种 Proxy 模式吧。

这个就是 AudioFlinger 异常复杂的一个原因！！！

```
class TrackHandle : public android::BnAudioTrack {
public:
    TrackHandle(const sp& track);
    virtual ~TrackHandle();
    virtual status_t  start();
    virtual void stop();
    virtual void flush();
    virtual void mute(bool);
    virtual void pause();
    virtual void setVolume(float left, float right);
    virtual sp getCblk() const;
    sp<PlaybackThread::Track> mTrack;
};
```

4) 线程类

AF 中有好几种不同类型的线程，分别有对应的线程类型：

(1)RecordThread ：

RecordThread : public ThreadBase, public AudioBufferProvider

用于录音的线程。

(2)PlaybackThread:

class PlaybackThread : public ThreadBase

用于播放的线程

(3)MixerThread

MixerThread : public PlaybackThread

用于混音的线程，注意他是从 PlaybackThread 派生下来的。

(4)DirectoutputThread

DirectOutputThread : public PlaybackThread

直接输出线程， DIRECT_OUTPUT之类的判断，最终和这个线程有关。

(5)DuplicatingThread ：

DuplicatingThread : public MixerThread

复制线程？而且从混音线程中派生？暂时不知道有什么用

这么多线程，都有一个共同的父类 ThreadBase，这个是 AF 对 Audio 系统单独定义的一个以 Thread 为基类的类。

5) PlayingThread 的内部类 Track

TrackHandle 构造用的那个 Track 是 PlayingThread 的 createTrack_l 得到的。

class Track : public TrackBase // TrackBase 是 ThreadBase 定义的内部类

class TrackBase : public AudioBufferProvider, public RefBase

基类 AudioBufferProvider 是一个对 Buffer 的封装，以后在 AF 读共享缓冲，写数据到硬件 HAL中。

4. AF 流程继续

这里终于在 AF 中的 createTrack 返回了 TrackHandle 。这个时候系统处于什么状态？ AF中的几个 Thread 我们之前说了，在 AF 启动的某个时间就已经起来了。我们就假设 AT 调用 AF 服务前，这个线程就已经启动了。

这个可以看代码就知道了：

```
void AudioFlinger::PlaybackThread::onFirstRef() {
    const size_t SIZE = 256;
    char buffer[SIZE];
    snprintf(buffer, SIZE, "Playback Thread %p", this);
    //onFirstRef ，实际是 RefBase 的一个方法，在构造 sp 的时候就会被调用
    // 下面的 run 就真正创建了线程并开始执行 threadLoop 了
    run(buffer, ANDROID_PRIORITY_URGENT_AUDIO);
}
```

到底执行哪个线程的 threadLoop ？是根据 output 句柄来查找线程的。

看看 openOutput 的实行，真正的线程对象创建是在那儿。

```
int AudioFlinger::openOutput(uint32_t *pDevices,...) {
    if () {
        //          如果 flags 没有设置直接输出标准，或者      format 不是 16bit ，或者声道数不是    2,
        //          则创建 DirectOutputThread    。
        thread = new DirectOutputThread(this, output, ++mNextThreadId);
    } else {
        //          创建的是最复杂的    MixerThread
        thread = new MixerThread(this, output, ++mNextThreadId);
    }
```

1) MixerThread

非常重要的工作线程，看看它的构造函数。

```
MixerThread::MixerThread(const sp<AudioFlinger>& audioFlinger,
        AudioStreamOut* output, int id)
: PlaybackThread(audioFlinger, output, id), mAudioMixer(0)
{
    mType = PlaybackThread::MIXER;
    //      混音器对象，传进去的两个参数是基类       ThreadBase 的，都为 0，最终混音的数据都由它生成
    mAudioMixer = new AudioMixer(mFrameCount, mSampleRate);
}
```

2) At 调用 start // 加到 active track 。start 才会触发 AudioTrackThread

此时， AT得到 IAudioTrack 对象后，调用 start 函数。

```
status_t AudioFlinger::TrackHandle::start() {
    return mTrack->start();
}
```

果然，自己不干活，交给 mTrack 了，这个是 PlayingThread createTrack_l 得到的 Track 对象

```
status_t AudioFlinger::PlaybackThread::Track::start()  {
    status_t status = NO_ERROR;
```

```
sp<ThreadBase> thread = mThread.promote();
//        这个 Thread 就是调用 createTrack_l 的那个 thread 对象，这里是 MixerThread
if (thread != 0) {
    Mutex::Autolock _l(thread->mLock);
    int state = mState;
    if (mState == PAUSED) {
        mState = TrackBase::RESUMING;
    } else {
        mState = TrackBase::ACTIVE;
    }
    //          把自己由加到 addTrack_l 了
    //          奇怪，我们之前在看 createTrack_l 的时候，不是已经有个 map保存创建的 track 了
    //          这里怎么又出现了一个类似的操作？
    PlaybackThread *playbackThread = (PlaybackThread *)thread.get();
    playbackThread->addTrack_l(this);  //                加到 active track 。
    return status;
}
```

看看这个 addTrack_l 函数

```
status_t AudioFlinger::PlaybackThread::addTrack_l(const sp<Track>& track)
{
    status_t status = ALREADY_EXISTS;
    // set retry count for buffer fill
    track->mRetryCount = kMaxTrackStartupRetries;
    if (mActiveTracks.indexOf(track) < 0) {
        mActiveTracks.add(track);//              啊，原来是加入到活跃 Track 的数组啊
        status = NO_ERROR;
    }
    //        看到这个 broadcast ，一定要想到：恩，在不远处有那么一个线程正等着这个 CV呢。
    mWaitWorkCV.broadcast();
```

```
    return status;
}
```

start() 是把某个 track 加入到 PlayingThread 的活跃 Track 队列，然后触发一个信号事件。由于这个事件是 PlayingThread 的内部成员变量，而 PlayingThread 又创建了一个线程，那么难道是那个线程在等待这个事件吗？这时候有一个活跃 track ，那个线程应该可以干活了吧？

这个线程是 MixerThread 。我们去看看它的线程函数 threadLoop 吧。

```
bool AudioFlinger::MixerThread::threadLoop() {
    int16_t* curBuf = mMixBuffer;
    Vector< sp<Track> > tracksToRemove;
    while (!exitPending()){
        processConfigEvents();
        //Mixer                 进到这个循环中来
        mixerStatus = MIXER_IDLE;
        { // scope for mLock
            Mutex::Autolock _l(mLock);
            const SortedVector< wp<Track> >& activeTracks = mActiveTracks;
            //                  每次都取当前最新的活跃      Track  数组
            //                  下面是预备操作，返回状态看看是否有数据需要获取
            mixerStatus = prepareTracks_l(activeTracks, &tracksToRemove);
        }
        //LIKELY            ，是 GCC的一个东西，可以优化编译后的代码      ， 就当做是 TRUE吧
        if (LIKELY(mixerStatus == MIXER_TRACKS_READY)) {
            // mix buffers...
            //                  调用混音器，把 buf 传进去，估计得到了混音后的数据了
            //curBuf                    是 mMixBuffer ， PlayingThread 的内部 buffer ，在某个地方已经创建好了，
            //                  缓存足够大
            mAudioMixer->process(curBuf);
            sleepTime = 0;
            standbyTime = systemTime() + kStandbyTimeInNsecs;
```

```
}
//          有数据要写到硬件中，肯定不能      sleep  了呀
if (sleepTime == 0) {
    //              把缓存的数据写到    outPut  中。这个  mOutput 是 AudioStreamOut
    //              由 Audio HAL 的那个对象创建得到
    int bytesWritten = (int)mOutput->write(curBuf, mixBufferSize);
    mStandby = false;
} else {
        usleep(sleepTime);//                  如果没有数据，那就休息吧     ..
}
```

3) MixerThread 核心

AF 的工作就是如此的精密。 MixerThread 的线程循环中，最重要的两个函数：

prepare_l 和 mAudioMixer->process ，我们一一来看看。

prepare_l 的功能是什么？根据当前活跃的 track 队列，来为混音器设置信息。可想而知，一个 track 必然在混音器中有一个对应的东西。我们待会分析 AudioMixer 的时候再详述。

为混音器准备好后，下面调用它的 process 函数

```
void AudioMixer::process(void* output) {
    mState.hook(&mState, output);    //hook                ？难道是钩子函数？
}
```

hook 是一个函数指针啊，在哪里赋值的？具体实现函数又是哪个？只能分析 AudioMixer 类了。

4) AudioMixer

AudioMixer 实现在 framework/base/libs/audioflinger/AudioMixer.cpp 中

process__OneTrack16BitsStereoNoResampling

// 单 track ， 16bit 双声道，不需要重采样 , 大部分是这种情况了

到现在，还没看到取共享内存里 AT 端 write 的数据。那只能到 bufferProvider 去看了。注意，这里用的是 AudioBufferProvider 基类，实际的对象是 Track 。它从 AudioBufferProvider 派生。我们用得是 PlaybackThread 的这个 Track

```
status_t PlaybackThread::Track::getNextBuffer(AudioBufferProvider::Buffer*) {
```

```
        //          千呼万唤始出来，终于见到      cblk   了
        audio_track_cblk_t* cblk = this->cblk();
        //          哈哈，看看数据准备好了没，
        framesReady = cblk->framesReady();
        if (LIKELY(framesReady)) {
        }
}
```

再看看释放缓冲的地方： releaseBuffer ，这个直接在 ThreadBase 中实现了

```
void ThreadBase::TrackBase::releaseBuffer(AudioBufferProvider::Buffer* buffer)
{
    buffer->raw = 0;
    mFrameCount = buffer->frameCount;
    step();
    buffer->frameCount = 0;
}
```

看看 step 吧。 mFrameCount表示我已经用完了这么多帧。

```
bool AudioFlinger::ThreadBase::TrackBase::step() {
    bool result;
    audio_track_cblk_t* cblk = this->cblk();
    result = cblk->stepServer(mFrameCount);
    //          调用 cblk 的 stepServer ，更新服务端的使用位置
    return result;
}
```

原来 AudioTrack 中 write 的数据，最终是这么被使用的呀！！！

2.2.4.3 再论共享 audio_track_cblk_t

audio_track_cblk_t 是一个环形 buffer ，环形 buffer 是？顺便解释下， audio_track_cblk_t 的使用和我之前说的 Lock，读/ 写， Unlock 不太一样。为何？

(1) 因为在 AF代码中．有缓冲 buffer 方面的 wait ， MixThread 只有当没有数据的时候会 usleep 一下。

(2) 如果有多个 track ，多个 audio_track_cblk_t 的话，假如又是采用 wait 信号的办法，那么由于 pthread 库缺乏 WaitForMultiObjects 的机制，那么到底该等哪一个？这个问题是我们之前在做跨平台同步库的一个重要难题。

1. 写者的使用

写者就是 AudioTrack 端，在这个类中，叫 user

buffer ，获得写空间起始地址

framesAvailable ，看看是否有空余空间

stepUser ，更新 user 的位置。

2. 读者的使用

读者是 AudioFlinger 端，在这个类中叫 server

framesReady ，获得可读的位置

stepServer ，更新读者的位置

看看这个类的定义：

```
struct audio_track_cblk_t {
    Mutex lock;  //                同步锁
    Condition cv;  //CV
    volatile uint32_t user;   //                    写者
    volatile uint32_t server;   //                    读者
    uint32_t userBase;   //                写者起始位置
    uint32_t serverBase;  //                读者起始位置
    void* buffers;   //                指向 FIFO 的起始地址
    uint32_t frameCount;
    // Cache line boundary
    uint32_t loopStart;    //                循环起始
    uint32_t loopEnd;    //                循环结束
    int loopCount;
    uint8_t out;   //                如果是 Track 的话， out 就是 1，表示输出。
}
```

注意这是 volatile ，跨进程的对象，看来这个 volatile 也是可以跨进程的嘛。

3. 写者分析

先用 frameavail 看看当前剩余多少空间，假设是第一次进来。读者还在那 sleep 呢。

```
uint32_t audio_track_cblk_t::framesAvailable() {
    Mutex::Autolock _l(lock);
    return framesAvailable_l();
}
int32_t audio_track_cblk_t::framesAvailable_l() {
    uint32_t u = this->user; //                  当前写者位置，此时为     0
    uint32_t s = this->server; //                  当前读者位置，此时为     0
    if (out) { out                  为 1
        uint32_t limit = (s < loopStart) ? s : loopStart;
        //          我们不设循环播放时间吗。所以      loopStart    是初始值   INT_MAX，所以  limit=0
        return limit + frameCount - u;
        //          返回  0+frameCount-0 ，也就是全缓冲最大的空间。假设       frameCount=1024   帧
    }
}
```

然后调用 buffer 获得其实位置， buffer 就是得到一个地址位置。

```
void* audio_track_cblk_t::buffer(uint32_t offset) const {
    return (int8_t *)this->buffers + (offset - userBase) * this->frameSize;
}
```

完了，我们更新写者，调用 stepUser

```
uint32_t audio_track_cblk_t::stepUser(uint32_t frameCount) {
    //framecount          ，表示我写了多少，假设这一次写了       512 帧
    uint32_t u = this->user;  //user                  位置还没更新呢，此时     u=0；
    u += frameCount;  //u                更新了，  u=512
    // Ensure that user is never ahead of server for AudioRecord
    if (out) {
        //          没甚，计算下等待时间
    }
    //userBase          还是初始值为  0，可惜啊，我们只写了    1024 的一半 ， 所以  userBase  加不了
    if (u >= userBase + this->frameCount) {
```

```
        userBase += this->frameCount;
        //          但是这句话很重要，    userBase 也更新了。根据   buffer   函数的实现来看，似乎把这个
        //          环形缓冲铺直了 ....   连绵不绝。
    }
    this->user = u;//              喔，  user 位置也更新为   512 了，但是  useBase 还是 0
    return u;
}
```

好了，假设写者这个时候 sleep 了，而读者起来了。

4. 读者分析

```
uint32_t audio_track_cblk_t::framesReady() {
    uint32_t u = this->user; //u                  为 512
    uint32_t s = this->server;//                  还没读呢，  s 为零
    if (out) {
        if (u < loopEnd) {
            return u - s;//loopEnd                    也是 INT_MAX，所以这里返回   512，表示有  512 帧可读了
        } else {
                Mutex::Autolock _l(lock);
            if (loopCount >= 0) {
                return (loopEnd - loopStart)*loopCount + u - s;
            } else {
                return UINT_MAX;
            }
        }
    } else {
        return s - u;
    }
}
```

使用完了，然后 stepServer

```
bool audio_track_cblk_t::stepServer(uint32_t frameCount) {
```

```
status_t err;
err = lock.tryLock();
uint32_t s = this->server;
s += frameCount; //          读了 512 帧了，所以  s=512
if (out) {
}
//        没有设置循环播放嘛，所以不走这个
if (s >= loopEnd) {
    s = loopStart;
    if (--loopCount == 0) {
        loopEnd = UINT_MAX;
        loopStart = UINT_MAX;
    }
}
//        一样啊，把环形缓冲铺直了
if (s >= serverBase + this->frameCount) {
    serverBase += this->frameCount;
}
this->server = s; //server                  为 512 了
cv.signal(); //              读者读完了。触发下写者吧。
        lock.unlock();
        return true;
    }
```

5. 真的是环形缓冲吗？

环形缓冲是这样一个场景，现在 buffer 共 1024 帧。

假设：写者先写到 1024 帧，读者读到 512 帧，那么，写者还可以从头写 512 帧。

所以，我们得回头看看 frameavail 是不是把这 512 帧算进来了。

```
    uint32_t audio_track_cblk_t::framesAvailable_l() {
        uint32_t u = this->user; //1024
```

```
        uint32_t s = this->server;//512
        if (out) {
            uint32_t limit = (s < loopStart) ? s : loopStart;
            return limit + frameCount - u;                    返回 512，用上了！
        }
    }
```

再看看 stepUser 这句话

```
    if (u >= userBase + this->frameCount) {u                为 1024， userBase 为 0， frameCount 为 1024
        userBase += this->frameCount;//                     好， userBase 也为 1024 了
    }
```

看看 buffer

```
    return (int8_t *)this->buffers + (offset - userBase) * this->frameSize;
    //offset          是外界传入的基于    user 的一个偏移量。
    //offset-userBase          ，得到的正式从头开始的那段数据空间。
```

2.2.5 Audio Policy

1. 目的

AudioPolicyService （APS）是个什么东西？为什么要有它的存在？下层的 Audio HAL 层又是怎么结合到 Android 中来的？更有甚者，问个实在问题：插入耳机后，声音又怎么从最开始的外放变成从耳机输出了？调节音量的时候到底是调节 Music 的还是调节来电音量呢？这些东西，在 AF 的流程中统统都没讲到。但是这些又是至关重要的。从策略（ Policy ）比流程更复杂和难懂。

1.1. AF 和 APS系统第一次起来后，到底干了什么。

1.2. 检测到耳机插入事件后， AF 和 APS的处理。

2. AF 和 APS的诞生

在 framework\base\media\MediaServer\Main_MediaServer 中：

```
int main(int argc, char** argv) {
        // 先创建 audioFlinger
    AudioFlinger::instantiate();
        // 再创建 audioPolicyManager
    AudioPolicyService::instantiate();
```

```
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

1) new AudioFlinger

instantiate 内部会 实 例化一个 对 象，那直接看 AF 的构造函数 .

```
AudioFlinger::AudioFlinger() : BnAudioFlinger(),
    mAudioHardware(0), mMasterVolume(1.0f), mMasterMute(false), mNextUniqueId(1)
{
// 注意 mAudioHardware 和 mNextUniqueId
mHardwareStatus = AUDIO_HW_IDLE;
// 创 建 audio 的 HAL代表
        mAudioHardware = AudioHardwareInterface::create();
mHardwareStatus = AUDIO_HW_INIT;
// 下面的 东西 不会使用 APS, APS还 没 创建。
if (mAudioHardware->initCheck() == NO_ERROR) {
    mMode = AudioSystem::MODE_NORMAL;
    setMode(mMode);
    setMasterVolume(1.0f);
    setMasterMute(false);
}
```

感 觉 上， AF 的构造函数就是 创 建了一个最重要的 AudioHardWare 的 HAL 代表。 好像是没干什 么 策略上的事情： AF 创 建了一个 AudioHardware 的 HAL 对 象。 注意整个系 统就 这 一个 AudioHardware 了。也就是 说，不管是 线控耳机， 蓝 牙耳机，麦克，外放等等，最后都会由 这 一个 HAL统一管理。

2) new AudioPolicyService

```
AudioPolicyService(): BnAudioPolicyService() , mpPolicyManager(NULL)
        // 策略管理器，很重要
{
    // start tone playback thread 播放 铃 声的 ?
    mTonePlaybackThread = new AudioCommandThread(String8(""));
    // start audio commands thread 音 频 命令 ?
```

```
mAudioCommandThread =          new AudioCommandThread(
                               String8("ApmCommandThread"));
//          注意 AudioPolicyManagerBase 的构造函数，把 this 传进 去了。
mpPolicyManager =              new AudioPolicyManagerBase(this)       ;
//          根据系 统属性来判断 摄 像机是否 强制使用声音。是防止 偷 拍吗 ？
property_get("ro.camera.sound.forced", value, "0");
mpPolicyManager->setSystemProperty("ro.camera.sound.forced", value);
}
```

这 里分析的是 Audio Policy ，而构造函数中又 创建了一个 AudioPolicyManagerBase ，而且不同厂商 还可以 实现 自己的 AudioPolicyManager ，看来 这个对 于音 频策略有至 关重要的作用了。

另外， AudioPolicyManagerBase 的构造函数可是把 APS传进 去了。

3) AudioPolicyManagerBase

```
AudioPolicyManagerBase(AudioPolicyClientInterface *clientInterface)
: mPhoneState(AudioSystem::MODE_NORMAL) //          这里有 电话 状 态
{           mpClientInterface = clientInterface;       //                  保存 APS对象
//          下面 这个意思就是把几 种 for_use 的情况使用的 设备 全部置 为 NONE
for (int i = 0; i < AudioSystem::NUM_FORCE_USE; i++) {
    mForceUse[i] = AudioSystem::FORCE_NONE;
}
//          目前可以的 输 出 设备，耳机和外放： OR操作符，最 终 mAvailableOutputDevices=0x3
mAvailableOutputDevices = AudioSystem::DEVICE_OUT_EARPIECE |
        AudioSystem::DEVICE_OUT_SPEAKER;
//          目前可用的 输入 设备，内置 MIC
mAvailableInputDevices = AudioSystem::DEVICE_IN_BUILTIN_MIC;
        // 创 建一个 AudioOutputDescriptor ，并 设 置它的 device 为 外设 0x2
AudioOutputDescriptor *outputDesc =                 new AudioOutputDescriptor()       ;
outputDesc->mDevice = (uint32_t)AudioSystem::DEVICE_OUT_SPEAKER;
//          调用 APS的 openOutput，得到一个 mHardwareOutput， 实际 就是 线 程 index
mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,
```

…);

addOutput(mHardwareOutput, outputDesc) ; //mOutputs 很重要

// 更新了 mHardwareOutput 对应 的输 出 设备， 发 命令 给 APS更新 对应 混音 线 程的 输出 设备

setOutputDevice(mHardwareOutput, AudioSystem::DEVICE_OUT_SPEAKER, true);

updateDeviceForStrategy();

}

------> mPhoneState(AudioSystem::MODE_NORMAL)

AudioSystem, Android 如何管理音 频系 统 的 关键 所在。位置在 framework\base\include \media\AudioSystem.h 中，定 义了大量的枚 举 之类的东西来表达 Google 对 音频 系统 的看法。

下面是 audio_mode 的定 义

```
enum audio_mode {
    MODE_INVALID = -2,   //              无效 mode
    MODE_CURRENT = -1,   //              当前 mode，和音 频设备 的切 换（路由）有 关
    MODE_NORMAL = 0,     //              正常 mode，没有 电话 和 铃声
    MODE_RINGTONE,       //              收到来 电信号了，此 时 会有 铃声
    MODE_IN_CALL,        //              电话 mode，这 里表示已 经 建立通 话 了
    NUM_MODES  // not a valid entry, denotes end-of-list
};
```

-------> AudioSystem::FORCE_NONE 和 AudioSystem::NUM_FORCE_USE

```
// device categories used for setForceUse()
enum forced_config {
    FORCE_NONE,
    FORCE_SPEAKER,
    FORCE_HEADPHONES,
    FORCE_BT_SCO,
    FORCE_BT_A2DP,
    FORCE_WIRED_ACCESSORY,
    FORCE_BT_CAR_DOCK,
    FORCE_BT_DESK_DOCK,
    NUM_FORCE_CONFIG,
```

```
        FORCE_DEFAULT = FORCE_NONE
    };
    // usages used for setForceUse()
    enum force_use {
        FOR_COMMUNICATION,  // FORCE_SPEAKER,  FORCE_BT_SCO
        FOR_MEDIA,  // FORCE_HEADPHONES, FORCE_BT_A2DP,  FORCE_WIRED_ACCESSORY
        FOR_RECORD, //FORCE_BT_SCO, FORCE_WIRED_ACCESSORY
        FOR_DOCK,  //BT_CAR_DOCK, BT_DESK_DOCK, WIRED_ACCESSORY
        NUM_FORCE_USE
    };
```

在 setSpeakerPhoneOn(bool on) 中：

```
public void setSpeakerphoneOn(boolean on){
    if (on) {
                    //  强制通 话使用  speaker
        AudioSystem.setForceUse(AudioSystem.FOR_COMMUNICATION,
                    AudioSystem.FORCE_SPEAKER);
        mForcedUseForComm = AudioSystem.FORCE_SPEAKER;
    } else {
        AudioSystem.setForceUse(AudioSystem.FOR_COMMUNICATION,
                    AudioSystem.FORCE_NONE);
        mForcedUseForComm = AudioSystem.FORCE_NONE;
    }
  }
```

-----> 输入输出设备

android 使用枚 举 audio_device 定义了很多 输入输出设备。

------>AudioOutputDescriptor *outputDesc = new AudioOutputDescriptor()]

注释： descriptor for audio outputs . Used to maintain current configuration of each opened audio output and keep track of the usage of this output by each audio stream type.

描述 audio 输出的，可以用来保存一些配置信息 。

跟踪音 频 stream 类型使用 这个 output 的一些情况。

------>mHardwareOutput = mpClientInterface->openOutput():

这里调用的是 APS的 openOutput，看看去：

audio_io_handle_t AudioPolicyService::openOutput(uint32_t *pDevices, ….)

{

sp<IAudioFlinger> af = AudioSystem::get_audio_flinger();

return af- >openOutput(pDevices, …);

}

------>AudioFlinger::openOutput()

int AudioFlinger::openOutput(uint32_t *pDevices, ….) {

// 传进 来的 值， *pDevices=0x2， 代表外放，其他都 为 0

Mutex::Autolock _l(mLock);

AudioStreamOut *output = mAudioHardware- >openOutputStream(*pDevices, ….) ;

mHardwareStatus = AUDIO_HW_IDLE;

if (output != 0) {

int id = nextUniqueId();

// 走哪个分支？ mAudioHardware->openOutputStream 确实会更改指 针对应 的 value 。

// 那几个 值变 成： format 为 PCM_16_BIT， channels 为 2， samplingRate 为 44100， 走 else

if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) ||

(format != AudioSystem::PCM_16_BIT) ||

(channels != AudioSystem::CHANNEL_OUT_STEREO)) {

thread = new DirectOutputThread(this, output, id, *pDevices);

} else {

// openOutput() ，就会在 AF 中创 建一个混音 线 程。所有外放的程序， 输出都由外放 stream

// 混音 线 程来工作 ； 所有耳机的程序， 输 出都由耳机 stream 混音 线 程完成 。

thread = new MixerThread(this, output, id, *pDevices);

}

}

mPlaybackThreads.add(id, thread); // 将混音 线 程加到 mPlaybackThreads

return id; // 返回混音 线 程的索引 .

```
    }
------>setOutputDevice(mHardwareOutput,...)
void AudioPolicyManagerBase::setOutputDevice(audio_io_handle_t output, uint32_t device, bool force, int
delayMs)
{
      //output=1, device              为 AudioSystem::DEVICE_OUT_SPEAKER, force   为 true
       AudioOutputDescriptor *outputDesc = mOutputs.valueFor(output);
       if (outputDesc->isDuplicated()) {
          setOutputDevice(outputDesc->mOutput1->mId, device, force, delayMs);
           setOutputDevice(outputDesc->mOutput2->mId, device, force, delayMs);
           return;
       }
       uint32_t prevDevice = (uint32_t)outputDesc->device();   //0x3,                          外放和耳机
       outputDesc->mDevice = device;  //                现在设置为外放
       //popCount          为 2，因为 device=0x2=0010
      //mHardwareOutput          实际上是 AF 返回的一个 线程索引，那 AMB怎么根据这样一个东西来管理所有的 线程呢？果然， 这里就比较了 output 是不是等于最初 创建的线程索引。 这就表明。 虽然只有这么一个 mHardwareOutput ，但实际上还是能够操作其他 output 的！
      if (output == mHardwareOutput && AudioSystem::popCount(device) == 2) {
        setStrategyMute(STRATEGY_MEDIA, true, output);
      //wait for the PCM output buffers to empty before proceeding with rest of the command
        usleep(outputDesc->mLatency*2*1000);
    }
      // do the routing              设置路由，新的 输出设备为 外放
    AudioParameter param = AudioParameter();
    param.addInt(        String8(AudioParameter::keyRouting)          , (int)device);
       mpClientInterface->setParameters          (mHardwareOutput, param.toString(), delayMs);
    // update stream volumes according to new device
    applyStreamVolumes(output, device, delayMs);
    // if changing from a combined headset + speaker route, unmute media streams
```

```
    if (output == mHardwareOutput && AudioSystem::popCount(prevDevice) == 2) {
        setStrategyMute(STRATEGY_MEDIA, false, output, delayMs);
    }
}
```

------>updateDeviceForStrategy()

```
    void AudioPolicyManagerBase::updateDeviceForStrategy() {
        for (int i = 0; i < NUM_STRATEGIES; i++) {
            mDeviceForStrategy[i] = getDeviceForStrategy((routing_strategy)i, false);
        }
    }
```

枚举：enum routing_strategy {

```
            STRATEGY_MEDIA,
            STRATEGY_PHONE,
            STRATEGY_SONIFICATION,
            STRATEGY_DTMF,
            NUM_STRATEGIES
        };
```

------>getDeviceForStrategy()

3. 总结

总结下吧， AF和 APS都创建完了，得到什么了吗？下面按先后顺序说说。

- AF创建了一个代表 HAL对象的东西
- APS创建了两个 AudioCommandThread，一个用来处理命令，一个用来播放 tone。
- APS 同时会创建 AudioManagerBase，做为系统默认的音频管理
- AMB集中管理了策略上面的事情，同时会在 AF 的 openOutput 中创建一个混音线程 。同时， AMB会更新一些策略上的安排。

另外，我们分析的 AMB是 Generic 的，但不同厂商可以实现自己的策略。例如我可以设置只要有耳机，所有类型声音都从耳机出。

2.4.1.2 Audio 系统的 JNI 代码

Android 的 Audio 部分通过 JNI 向 Java 层提供接口，在 Java 层可以通过 JNI 接口完成 Audio 系统的大部分操作。

Audio JNI 部分的代码路径为： frameworks/base/core/jni 。

其中，主要实现的 3 个文件为：android_media_AudioSystem.cpp 、android_media_AudioTrack.cpp 和 android_media_AudioRecord.cpp ，它们分别对应了 Android Java 框架中的 3 个类的支持：

android.media.AudioSystem ：负责 Audio 系统的总体控制；

android.media.AudioTrack ：负责 Audio 系统的输出环节；

android.media.AudioRecorder ：负责 Audio 系统的输入环节。

在 Android 的 Java 层中，可以对 Audio 系统进行控制和数据流操作，对于控制操作，和底层的处理基本一致；但是对于数据流操作，由于 Java 不支持指针，因此接口被封装成了另外的形式。例如，对于音频输出，android_media_AudioTrack 提供的是写字节和写短整型的接口类型。所定义的 JNI 接口 native_write_byte 和 native_write_short ，它们一般是通过调用 AudioTrack 的 write() 函数来完成的，只是在 Java 的数据类型和 C++ 的指针中做了一步转换。

2.4.1.3 Audio 系统的 Java 代码

Android 的 Audio 系统的相关类在 android.media 包中， Java 部分的代码路径为：

frameworks/base/media/java/android/media

Audio 系统主要实现了以下几个类：android.media.AudioSystem 、android.media.AudioTrack 、android.media.AudioRecorder 、android.media.AudioFormat 。前面的 3 个类和本地代码是对应的， AudioFormat 提供了一些 Audio 相关类型的枚举值。

注意：在 Audio 系统的 Java 代码中，虽然可以通过 AudioTrack 和 AudioRecorder 的 write() 和 read()接口，在 Java 层对 Audio 的数据流进行操作。但是，更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的输入 / 输出，而 Java 层只进行控制类操作，不处理数据流 。

2.4.2 Audio 的硬件抽象层

2.4.2.1 Audio 硬件抽象层（ HAL ）的接口定义

Audio 的硬件抽象层是 AudioFlinger 和 Audio 硬件的接口，在各个系统的移植过程中可以有不同的实现方式。Audio 硬件抽象层的接口路径为：

hardware/libhardware_legacy/include/hardware_legacy/

C++文件路径： hardware/msm7k/libaudio/

其中主要的文件为： AudioHardwareBase.h 和 AudioHardwareInterface.h 。

Android 中的 Audio 硬件抽象层可以基于 Linux 标准的 ALSA 或 OSS 音频驱动实现，也可以基于私有的 Audio 驱动接口来实现。在高通 QSD8K 平台上，采用私有的 Audio 驱动接口（设备节点名为 /dev/msm_audio_ctl 、/dev/msm_pcm_out 、/dev/msm_pcm_in 等，其功能基本上是类似于 ALSA或 OSS的 pcm driver ）。

在 AudioHardwareInterface.h 中的类： AudioStreamOut 、AudioStreamIn 和 AudioHardwareInterface 。AudioStreamOut 和 AudioStreamIn 的主要定义如下所示：

```
class AudioStreamOut{
public:
    virtual ~AudioStreamOut() = 0;
    virtual status_t              setVolume (float volume) = 0;
    virtual ssize_t               write  (const void* buffer,size_t bytes) = 0;
    //.....           省略部分内容
};
class AudioStreamIn {
public:
 virtual ~AudioStreamIn() = 0;
 virtual status_t            setGain  (float gain) = 0;
 virtual ssize_t           read (void* buffer, ssize_t bytes) = 0;
 //.....         省略部分内容
};
```

AudioStreamOut 和 AudioStreamIn 分别对应了 音频的输出环节和输入环节 ，其中负责数据流的接口分别是 wirte() 和 read()，参数是一块内存的指针和长度；另外还有一些设置和获取接口。

Audio 的硬件抽象层主体 AudioHardwareInterface 类的定义如下所示：

```
class AudioHardwareInterface
{
public:
    virtual status_t initCheck() = 0;
    virtual status_t setVoiceVolume(float volume) = 0;
    virtual status_t setMasterVolume(float volume) = 0;
    virtual status_t setRouting(int mode, uint32_t routes) = 0;
    virtual status_t getRouting(int mode, uint32_t* routes) = 0;
    virtual status_t setMode(int mode) = 0;
 virtual status_t getMode(int* mode) = 0;
 //.....          省略部分内容
 virtual AudioStreamOut*          openOutputStream  (//  打开输出流  ) =0;
 virtual AudioStreamIn*           openInputStream   (//    打开输入流  ) =0;
    static      AudioHardwareInterface* create()          ;      //  获取一个  AudioHardware Interface       类型的指针
};
```

在这个 AudioHardwareInterface 接口中，使用 openOutputStream() 和 openInputStream() 函数分别获取 AudioStreamOut 和 AudioStreamIn 两个类，它们作为音频输入 /输出设备来使用。

此外， AudioHardwareInterface.h 定义了 C 语言的接口来获取一个 AudioHardware Interface 类型的指针。

```
Extern "C" AudioHardwareInterface*    createAudioHardware(void);
```

如果实现一个 Android 的硬件抽象层，则需要实现 AudioHardwareInterface 、AudioStreamOut 和 AudioStreamIn 三个类，将代码编译成动态库 libauido.so 。AudioFlinger 会连接这个动态库， 并调用其中的 createAudioHardware() 函数来获取接口 。

在 AudioHardwareBase.h 中定义了类： AudioHardwareBase ， 它继承了 AudioHardwareInterface ， 显然继承这个接口也可以实现 Audio 的硬件抽象层。

提示： Android 系统的 Audio 硬件抽象层可以通过继承类 AudioHardwareInterface 来实现，其中分为控制部分和输入/输出处理部分。

2.4.2.2 AudioFlinger 中自带 Audio 硬件抽象层实现

在 AudioFlinger 中可以通过编译宏的方式选择使用哪一个 Audio 硬件抽象层。这些 Audio 硬件抽象层既可以作为参考设计，也可以在没有实际的 Audio 硬件抽象层（甚至没有 Audio 设备）时使用，以保证系统的正常运行。

在 AudioFlinger 的编译文件 Android.mk 中，具有如下的定义：

```
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
    LOCAL_STATIC_LIBRARIES += libaudiointerface
else
    LOCAL_SHARED_LIBRARIES += libaudio
endif
LOCAL_MODULE:= libaudioflinger
include $(BUILD_SHARED_LIBRARY)
```

定义的含义为：当宏 BOARD_USES_GENERIC_AUDIO 为 true 时，连接 libaudiointerface.a 静态库；当 BOARD_USES_GENERIC_AUDIO 为 false 时，连接 libaudio.so 动态库。在正常的情况下，一般是使用后者，即在另外的地方实现 libaudio.so 动态库，由 AudioFlinger 的库 libaudioflinger.so 来连接使用。

libaudiointerface.a 也在这个 Android.mk 中生成，通过编译 4 个源文件（AudioHardwareGeneric.cpp 、AudioHardwareStub.cpp 、AudioDumpInterface.cpp 、AudioHardwareInterface.cpp ），生成了 libaudiointerface.a 静态库。其中 AudioHardwareInterface.cpp 负责实现基础类和管理，而 AudioHardwareGeneric.cpp 、AudioHardwareStub.cpp 和 AudioDumpInterface.cpp 三个文件各自代表一种 Auido 硬件抽象层的实现。

AudioHardwareGeneric.cpp ：实现基于特定驱动的通用 Audio 硬件抽象层；

AudioHardwareStub.cpp ：实现 Audio 硬件抽象层的一个桩；

AudioDumpInterface.cpp ：实现输出到文件的 Audio 硬件抽象层。

在 AudioHardwareInterface.cpp 中，实现了 Audio 硬件抽象层的创建函数 AudioHardwareInterface::create() ，根据 GENERIC_AUDIO 、DUMP_FLINGER_OUT 等宏选择创建几个不同的 Audio 硬件抽象层，最后返回的接口均为 AudioHardwareInterface 类型的指针。

1) 用桩实现的 **Audio** 硬件抽象层

AudioHardwareStub.h 和 AudioHardwareStub.cpp 是一个 Android 硬件抽象层的桩实现方式。这个实现不操作实际的硬件和文件，它所进行的是空操作，在系统没有实际的 Audio 设备时使用这个实现，来保证系统的正常工作。如果使用这个硬件抽象层，实际上 Audio 系统的输入和输出都将为空。

在实现过程中，为了保证声音可以输入和输出，这个桩实现的主要内容是实现 AudioStreamOutStub 和 AudioStreamInStub 类的读/写函数。如下所示：

```
class AudioStreamOutStub : public AudioStreamOut {
public:
    virtual status_t    set(int format, int channelCount, uint32_t sampleRate);
    virtual uint32_t    sampleRate() const { return 44100; }
    virtual size_t      bufferSize() const { return 4096; }
    virtual int         channelCount() const { return 2; }
```

```
    virtual int         format() const { return AudioSystem::PCM_16_BIT; }
    virtual uint32_t    latency() const { return 0; }
    virtual status_t    setVolume(float volume) { return NO_ERROR; }
    virtual ssize_t     write(const void* buffer, size_t bytes);
    virtual status_t    standby();
    virtual status_t    dump(int fd, const Vector<String16>& args);
};

class AudioStreamInStub : public AudioStreamIn {
public:
    virtual status_t    set(int format, int channelCount, uint32_t sampleRate,
AudioSystem::audio_in_acoustics acoustics);
    virtual uint32_t    sampleRate() const { return 8000; }
    virtual size_t      bufferSize() const { return 320; }
    virtual int         channelCount() const { return 1; }
    virtual int         format() const { return AudioSystem::PCM_16_BIT; }
    virtual status_t    setGain(float gain) { return NO_ERROR; }
    virtual ssize_t     read(void* buffer, ssize_t bytes);
    virtual status_t    dump(int fd, const Vector<String16>& args);
    virtual status_t    standby() { return NO_ERROR; }
};
```

上面实际上使用了最简单的模式，只是用固定的参数，如缓存区大小、采样率、通道数等，以及将一些函数直接无错误返回。

此外使用 AudioHardwareStub 类继承 AudioHardwareBase, AudioHardwareBase 又继承 AudioHardwareInterface ，所以 AudioHardwareStub 也就是继承 AudioHardwareInterface 。如下所示：

```
class AudioHardwareStub : public  AudioHardwareBase
{
public:
                    AudioHardwareStub();
    virtual          ~AudioHardwareStub();
    virtual status_t    initCheck();
    virtual status_t    setVoiceVolume(float volume);
    virtual status_t    setMasterVolume(float volume);

    // mic mute
        virtual status_t    setMicMute(bool state) { mMicMute = state;  return  NO_ERROR; }
    virtual status_t    getMicMute(bool* state) { *state = mMicMute ; return NO_ERROR; }

    virtual status_t    setParameter(const char* key, const char* value)
          { return NO_ERROR; }

    // create I/O streams
    virtual AudioStreamOut* openOutputStream(
                        int format=0,
```

```
                int channelCount=0,
                uint32_t sampleRate=0,
                status_t *status=0);

    virtual AudioStreamIn* openInputStream(
                int inputSource,
                int format,
                int channelCount,
                uint32_t sampleRate,
                status_t *status,
                        AudioSystem::audio_in_acoustics acoustics);

protected:
    virtual status_t    doRouting() { return NO_ERROR; }
    virtual status_t    dump(int fd, const Vector<String16>& args);

        bool        mMicMute;
private:
        status_t        dumpInternals(int fd, const Vector<String16>& args);
};

// ----------------------------------------------------------------------------

};
```

在实现过程中，为了保证声音可以输入和输出，这个庄实现的主要内容是实现 AudioStreamOutStub 和 AudioStreamInStub 类的读写函数。如下所示：

```
ssize_t AudioStreamOutStub::write(const void* buffer, size_t bytes)
{
    // fake timing for audio output
    usleep(bytes * 1000000 / sizeof(int16_t) / channelCount() / sampleRate());
    return bytes;
}
ssize_t AudioStreamInStub::read(void* buffer, ssize_t bytes)
{
    // fake timing for audio input
    usleep(bytes * 1000000 / sizeof(int16_t) / channelCount() / sampleRate());
    memset(buffer, 0, bytes);
    return bytes;
}
```

2) Android 通用的 Audio 硬件抽象层

AudioHardwareGeneric.h 和 AudioHardwareGeneric.cpp 是 Android 通用的一个 Audio 硬件抽象层。与前面的桩实现

不同，这是一个真正能够使用的 Audio 硬件抽象层，但是它需要 Android 的一种特殊的声音驱动程序的支持。

与前面类似，使用 AudioStreamOutGeneric 、AudioHardwareInGeneric 、 AudioHardwareGeneric 这三个类分别继承 Audio 硬件抽象层的三个接口。如下所示：

```
class AudioStreamOutGeneric : public AudioStreamOut {
public:
                    AudioStreamOutGeneric() : mAudioHardware(0), mFd(-1) {}
    virtual          ~AudioStreamOutGeneric();

    virtual status_t    set(
            AudioHardwareGeneric *hw,
            int mFd,
            int format,
            int channelCount,
            uint32_t sampleRate);

    virtual uint32_t    sampleRate() const { return 44100; }
    virtual size_t      bufferSize() const { return 4096; }
    virtual int         channelCount() const { return 2; }
    virtual int         format() const { return AudioSystem::PCM_16_BIT; }
    virtual uint32_t    latency() const { return 20; }
    virtual status_t    setVolume(float volume) { return INVALID_OPERATION; }
    virtual ssize_t     write(const void* buffer, size_t bytes);
    virtual status_t    standby();
    virtual status_t    dump(int fd, const Vector<String16>& args);

private:
    AudioHardwareGeneric *mAudioHardware;
    Mutex   mLock;
    int     mFd;
};

class AudioStreamInGeneric : public AudioStreamIn {
public:
                    AudioStreamInGeneric() : mAudioHardware(0), mFd(-1) {}
    virtual          ~AudioStreamInGeneric();

    virtual status_t    set(
            AudioHardwareGeneric *hw,
            int mFd,
```

```
        int format,
        int channelCount,
        uint32_t sampleRate,
        AudioSystem::audio_in_acoustics acoustics);

    uint32_t    sampleRate() const { return 8000; }
    virtual size_t      bufferSize() const { return 320; }
    virtual int         channelCount() const { return 1; }
    virtual int         format() const { return AudioSystem::PCM_16_BIT; }
    virtual status_t    setGain(float gain) { return INVALID_OPERATION; }
    virtual ssize_t     read(void* buffer, ssize_t bytes);
    virtual status_t    dump(int fd, const Vector<String16>& args);
    virtual status_t    standby() { return NO_ERROR; }

private:
    AudioHardwareGeneric *mAudioHardware;
    Mutex   mLock;
    int     mFd;
};


class AudioHardwareGeneric : public AudioHardwareBase
{
public:
                    AudioHardwareGeneric();
    virtual           ~AudioHardwareGeneric();
    virtual status_t    initCheck();
    virtual status_t    setVoiceVolume(float volume);
    virtual status_t    setMasterVolume(float volume);

    // mic mute
    virtual status_t    setMicMute(bool state);
    virtual status_t    getMicMute(bool* state);

    virtual status_t    setParameter(const char* key, const char* value)
            { return NO_ERROR; }

    // create I/O streams
    virtual AudioStreamOut* openOutputStream(
            int format=0,
```

```
        int channelCount=0,
        uint32_t sampleRate=0,
        status_t *status=0);

    virtual AudioStreamIn* openInputStream(
        int inputSource,
        int format,
        int channelCount,
        uint32_t sampleRate,
        status_t *status,
        AudioSystem::audio_in_acoustics acoustics);

        void            closeOutputStream(AudioStreamOutGeneric* out);
        void            closeInputStream(AudioStreamInGeneric* in);
protected:
    virtual status_t        doRouting() { return NO_ERROR; }
    virtual status_t        dump(int fd, const Vector<String16>& args);

private:
    status_t                dumpInternals(int fd, const Vector<String16>& args);

    Mutex                   mLock;
    AudioStreamOutGeneric   *mOutput;
    AudioStreamInGeneric    *mInput;
    int                     mFd;
    bool                    mMicMute;
};
```

};

在 AudioHardwareGeneric.cpp 的实现中，使用的驱动程序是 /dev/eac，这是一个非标准程序，定义设备的路径如下所示：

static char const * const kAudioDeviceName = "/dev/eac";

对于 Linux 操作系统，这个驱动程序在文件系统中的节点主设备号为 10，次设备号自动生成。

提示： eac 是 Linux 中的一个 misc 驱动程序，作为 Android 的通用音频驱动，写设备表示放音，读设备表示录音。

在 audioHardwareGeneric 的构造函数中，打开这个驱动程序的设备节点。

```
AudioHardwareGeneric::AudioHardwareGeneric()
    : mOutput(0), mInput(0),  mFd(-1), mMicMute(false)
{
    mFd = ::open(kAudioDeviceName, O_RDWR);
```

```
}
```

这个音频设备是一个比较简单的驱动程序，没有很多设置接口，只是用读设备表示录音，写设备表示放音。放音和录音支持的都是 16 位的 PCM。

```
ssize_t AudioStreamOutGeneric::write(const void* buffer, size_t bytes)
{
    Mutex::Autolock _l(mLock);
    return ssize_t(::write(mFd, buffer, bytes));
}
ssize_t AudioStreamInGeneric::read(void* buffer, ssize_t bytes)
{
    // FIXME: remove logging
    LOGD("AudioStreamInGeneric::read(%p, %d) from fd %d", buffer, bytes, mFd);
    AutoMutex lock(mLock);
    if (mFd < 0) {
        LOGE("Attempt to read from unopened device");
        return NO_INIT;
    }
    return ::read(mFd, buffer, bytes);
}
```

虽然 AudioHardwareGeneric 是一个可以真正工作的 Audio 硬件抽象层，但是这种实现方式非常简单，不支持各种设置，参数也只能使用默认的。而且，这种驱动程序需要在 Linux 核心加入 eac 驱动程序的支持。

3) 提供 **Dump** 功能的 **Audio** 硬件抽象层

AudioDumpInterface.h 和 AudioDumpInterface.cpp 是一个提供了 Dump 功能的 Audio 硬件抽象层，它所起到的作用就是将输出的 Audio 数据写入到文件中。

AudioDumpInterface 本身支持 Audio 的输出功能，不支持输入功能。只实现了 AudioStreamOut ，没有实现 AudioStreamIn ，因此这个 Audio 硬件抽象层只支持输出功能，不支持输入功能。如下所示：

```
class AudioStreamOutDump : public AudioStreamOut {
public:
                AudioStreamOutDump( AudioStreamOut* FinalStream);
                ~AudioStreamOutDump();
                virtual ssize_t     write(const void* buffer, size_t bytes);

    virtual uint32_t    sampleRate() const { return mFinalStream->sampleRate(); }
    virtual size_t      bufferSize() const { return mFinalStream->bufferSize(); }
    virtual int         channelCount() const { return mFinalStream->channelCount(); }
    virtual int         format() const { return mFinalStream->format(); }
    virtual uint32_t    latency() const { return mFinalStream->latency(); }
    virtual status_t    setVolume(float volume)
                        { return mFinalStream->setVolume(volume); }
```

```
    virtual status_t    standby();
    virtual status_t    dump(int fd, const Vector<String16>& args) { return
     mFinalStream->dump(fd, args); }
    void                Close(void);

private:
    AudioStreamOut      *mFinalStream;
    FILE                *mOutFile;     // output file
};

class AudioDumpInterface : public AudioHardwareBase
{

public:
                    AudioDumpInterface(AudioHardwareInterface* hw);
    virtual AudioStreamOut* openOutputStream(
                        int format=0,
                        int channelCount=0,
                        uint32_t sampleRate=0,
                        status_t *status=0);
    virtual             ~AudioDumpInterface();

    virtual status_t    initCheck()
                        {return mFinalInterface->initCheck();}
    virtual status_t    setVoiceVolume(float volume)
                        {return mFinalInterface->setVoiceVolume(volume);}
    virtual status_t    setMasterVolume(float volume)
                        {return mFinalInterface->setMasterVolume(volume);}

// mic mute
virtual status_t    setMicMute(bool state)
                    {return mFinalInterface->setMicMute(state);}
virtual status_t    getMicMute(bool* state)
                    {return mFinalInterface->getMicMute(state);}

virtual status_t    setParameter(const char* key, const char* value)
                    {return mFinalInterface->setParameter(key, value);}
```

```
    virtual AudioStreamIn* openInputStream(int inputSource, int format, int
     channelCount,
          uint32_t sampleRate, status_t *status, AudioSystem::audio_in_acoustics
     acoustics)
      { return mFinalInterface->openInputStream(inputSource, format, channelCount,
     sampleRate, status, acoustics); }

    virtual status_t    dump(int fd, const Vector<String16>& args) { return
     mFinalInterface->dumpState(fd, args); }

protected:
    virtual status_t    doRouting() {return mFinalInterface->setRouting(mMode,
      mRoutes[mMode]);}

    AudioHardwareInterface  *mFinalInterface;
    AudioStreamOutDump      *mStreamOut;

};
};
```

输出文件的名称被定义为：

```
#define FLINGER_DUMP_NAME "/data/FlingerOut.pcm"
```

在 audioDumpInterface.cpp 的 AudioStreamOut 所实现的写函数中，写入的对象就是这个文件。如下所示：

```
ssize_t AudioStreamOutDump::write(const void* buffer, size_t bytes)
{
    ssize_t ret;

    ret = mFinalStream->write(buffer, bytes);
    if(!mOutFile && gFirst) {
        gFirst = false;
        // check if dump file exist
        mOutFile = fopen(FLINGER_DUMP_NAME, "r");
        if(mOutFile) {
            fclose(mOutFile);
            mOutFile = fopen(FLINGER_DUMP_NAME, "ab");
        }
        }
    if (mOutFile) {
        fwrite(buffer, bytes, 1, mOutFile);
```

```
    }
    return ret;
```

如果文件是打开的，则使用追加方式写入。因此使用这个 Audio 硬件抽象层时，播放的内容（ PCM）将全部被写入文件。而且这个类支持各种格式的输出，这取决于调用者的设置。

AudioDumpInterface 并不是为了实际的应用使用的，而是为了调试使用的类。当进行音频播放器调试时，有时无法确认是解码器的问题还是 Audio 输出单元的问题，这时就可以用这个类来替换实际的 Audio 硬件抽象层，将解码器输出的 Audio 的 PCM 数据写入文件中，由此可以判断解码器的输出是否正确。

提示：使用 AudioDumpInterface 音频硬件抽象层，可以通过 /data/FlingerOut.pcm 文件找到 PCM 的输出数据。

2.4.2.3 Audio 硬件抽象层的真正实现

实现一个真正的 Audio 硬件抽象层，需要完成的工作和实现以上的硬件抽象层类似。

例如：可以基于 Linux 标准的音频驱动： OSS（Open Sound System）或者 ALSA （Advanced Linux Sound Architecture ）驱动程序来实现。

对于 OSS 驱动程序，实现方式和前面的 AudioHardwareGeneric 类似，数据流的读 /写操作通过对 /dev/dsp 设备的读 /写来完成；区别在于 OSS 支持了更多的 ioctl 来进行设置，还涉及通过 /dev/mixer 设备进行控制，并支持更多不同的参数。

对于 ALSA 驱动程序，实现方式一般不是直接调用驱动程序的设备节点，而是先实现用户空间的 alsa-lib，然后 Audio 硬件抽象层通过调用 alsa-lib 来实现。

在实现 Audio 硬件抽象层时，对于系统中有多个 Audio 设备的情况，可由硬件抽象层自行处理 setRouting() 函数设定，例如，可以选择支持多个设备的同时输出，或者有优先级输出。对于这种情况，数据流一般来自 AudioStreamOut::write() 函数，可由硬件抽象层确定输出方法。对于某种特殊的情况，也有可能采用硬件直接连接的方式，此时数据流可能并不来自上面的 write()，这样就没有数据通道，只有控制接口。 Audio 硬件抽象层也是可以处理这种情况的。

2.4.3 audio HAL 在 Eclair/Froyo 和 Donut 的差异

Android 2.1/2.2 (Eclair/Froyo) 和 1.6 (Donut) 的音频系统架构层次分别见图 3 和图 4。

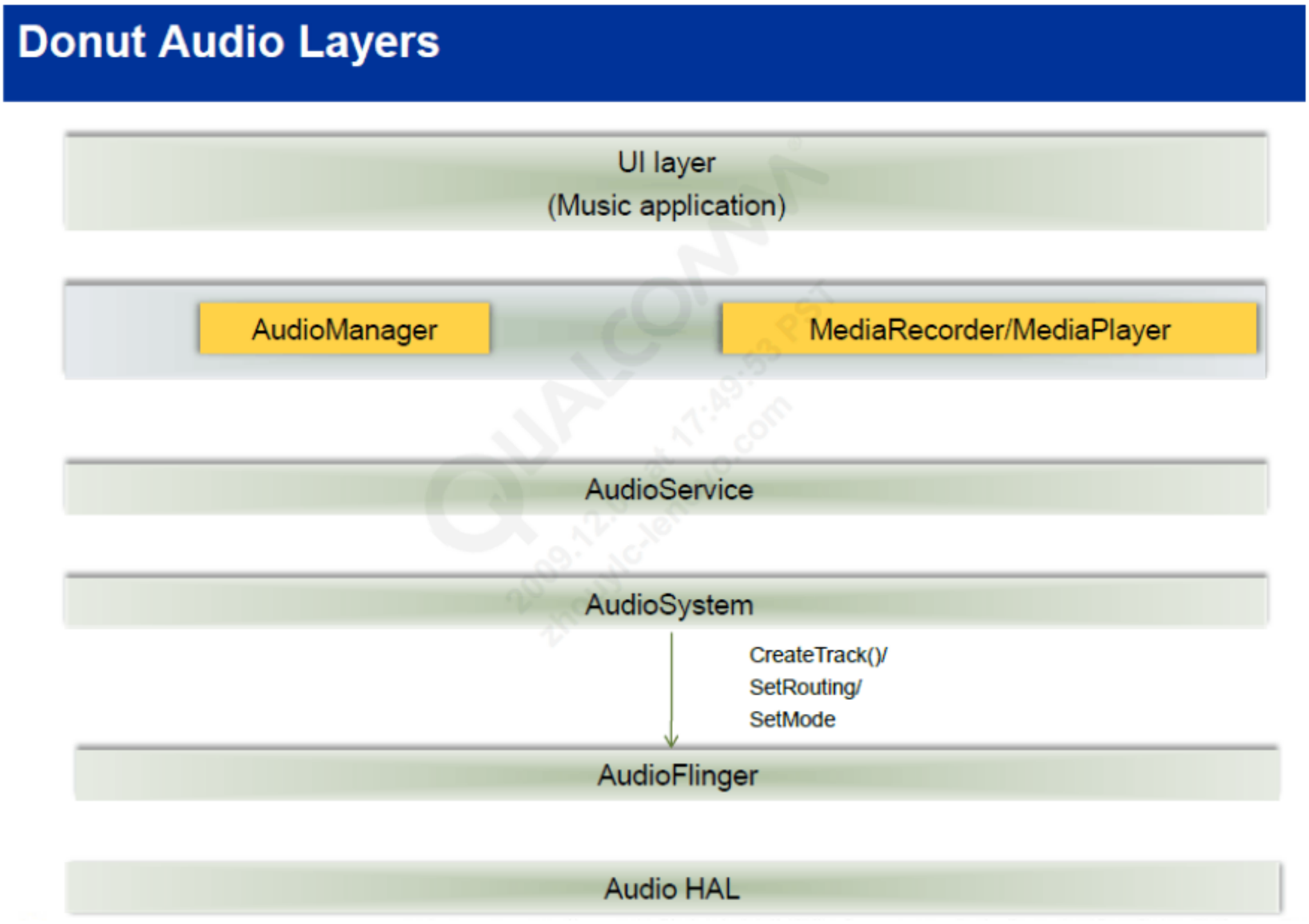

图 3

可见 Eclair/Froyo 相比于 Donut 在 audio HAL 上最大的差异是 增加了 AudioPolicyService/AudioPolicyManager 用于实现音频设备路由控制。在 hardware/msm7k/libaudio-qsd8k/ 下增加的文件有 AudioPolicyManager.h 和 AudioPolicyManager.cpp ，实现了类 AudioPolicyManager ，相应地编译出的 HAL 库除了 libaudio.so 以外增加了 libaudiopolicy.so 。

在 donut 上，音频 设备路由控制是分散在各种应用程序、 services 及 frameworks 中，例如 phone 应用、AudioManager/AudioService 、AudioFlinger ；而在 Eclair/Froyo 中，统一集中在 AudioPolicyManager （也称为 routing manager）进行。应用程序发送切换路由的 event 给 AudioManager ， AudioManager 再向 routing manager 发送请求，经由 AudioPolicyService 在 audio HAL 统一进行路由控制。 RoutingManager 也侦听来自 phone 应用等发出的广播 intents。

Eclair/Froyo 相比于 Donut 在音频系统的另一个变化是增加了对编码后的音频数据流（例如 MP3 数据流）的支持，也就是编码的数据流而不是 PCM 数据流可直接传给 HAL ，再由 HAL 传给硬件进行解码和播放，在高通平台上是支持硬件解码 MP3、AAC 等音频数据流的，但由于目前 Android 只支持 non-tunnel 模式音频播放，因此还无法用到。

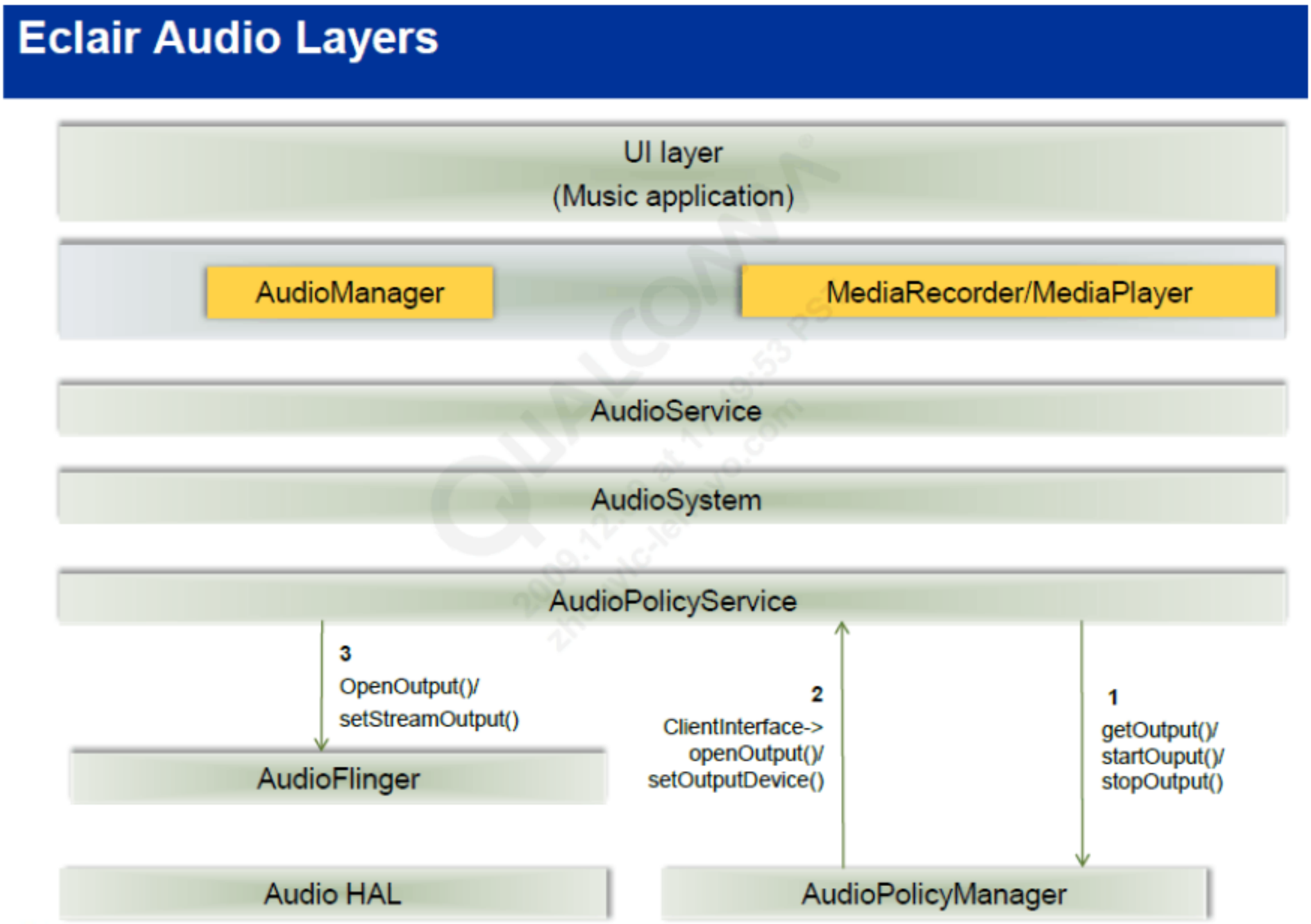

图 4

2.4.4　Android Policy Manger

Android　Policy　Manager（APM ）定义在多个音频实例（如电话通知、音频的播放、提示等）的并发规则。在　　APM 规则中有一条定义哪一个音频实例获得设备来播放，这里这个音频实例被看作一个任务而设备被看作一种资源，每个音频实例都要被分配一个设备才能播放　　.APM　就定义了一个优先级高的音频实例如何强制获得被某些优先级底音频实例使用的设备的规则，所有说ＡＰM就像一个资源管理者，能够使高优先级的任务从底优先级的任务获取资源，ＡＰM主要有以下职责，ＡＰM和其它模块的交互方式图 5

管理不同的输入输出设备接口，如ＨＡＬ，Ａ２ＤＰ

管理不同的输入输出设备，如扬声器，耳机等

基于特定数据流选择和定义合适的路由策略

管理每个数据流的声音的设置

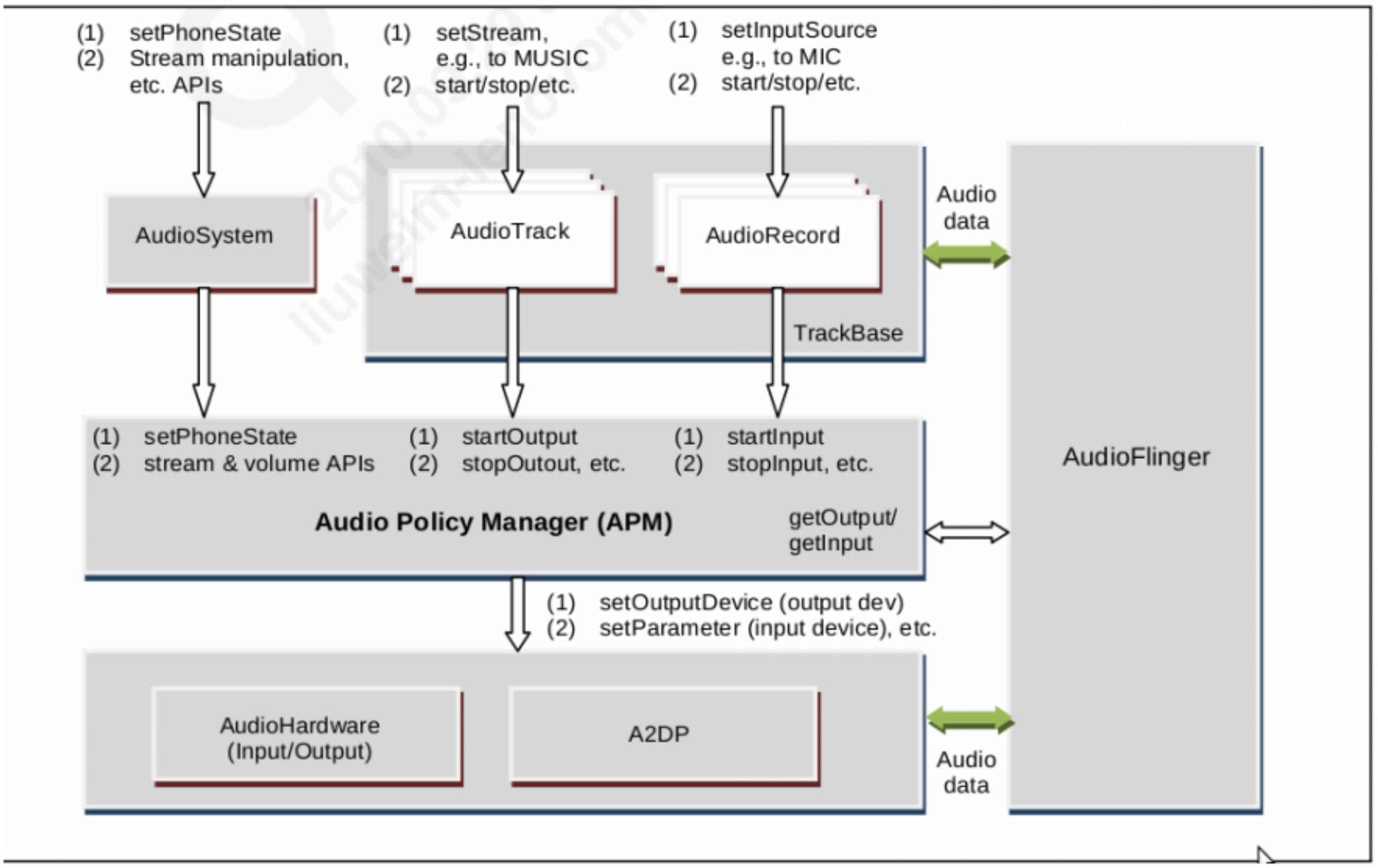

图 5 ＡＰM和其它模块的交互方式

2.2.5 高通 Android 的 Audio HAL 实现方式

2.2.5.1 高通的 Android 的 Audio 的架构

高通也是在遵循 Google 的框架来实现自己 Audio 的 HAL 。通过实现 AudioStreamOut, AudioStreamIn 和 audioHardwareBase（继承 AudioHardwareInterface ）来实现 HAL，如图 6。

2.2.5.2 高通 Android 的 Audio 的具体实现

1) **Audio HAL** 实现代码在 **hardware/msm7k/libaudio-qsd8k,** 主要文件是 **AudioHardware.h** 和 **AudioHardware.cpp**，编译生成库 **libaudio.so** 和 **libaudiopolicy.so**；

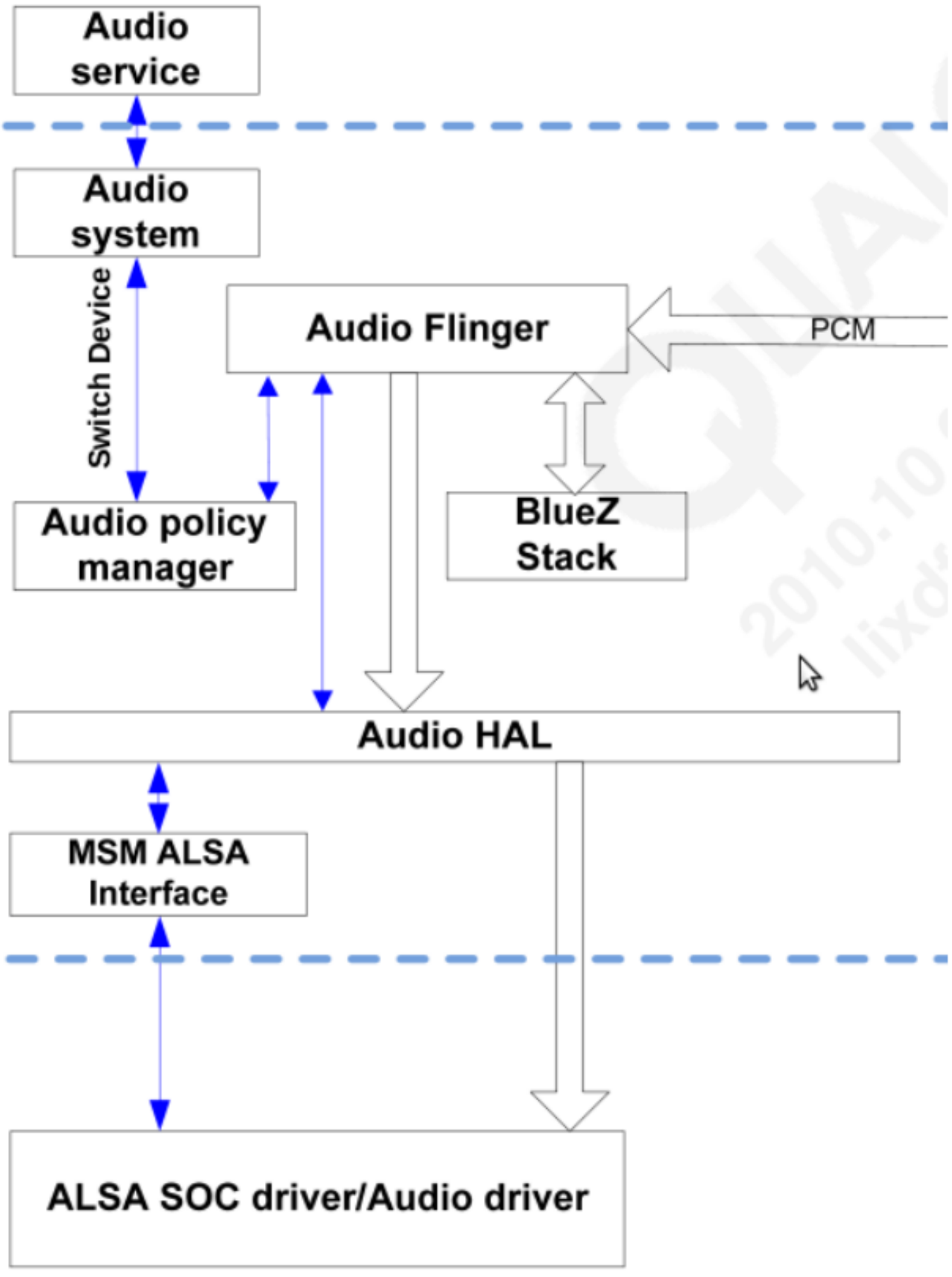

图 6

2) **Audio HAl** 的代码实现符合 **Google** 架构，是通过定义 **AudioStreamOutMSM7xx** 、**AudioStreamInMSM72xx** 、**AudioHardware** 这三个类，这三个类通过 **AudioStreamOut, AudioStreamIn** 和 **audioHardwareBase**（继承 **AudioHardwareInterface** ）来实现自己的 **HAL**，具体代码如下：

```
class AudioHardware : public  AudioHardwareBase

{

    class AudioStreamOutMSM72xx;

    class AudioStreamInMSM72xx;


public:

                    AudioHardware();

    virtual             ~AudioHardware();

    virtual status_t    initCheck();


    virtual status_t    setVoiceVolume(float volume);

    virtual status_t    setMasterVolume(float volume);


    virtual status_t    setMode(int mode);


    // mic mute

    virtual status_t    setMicMute(bool state);
```

```
virtual status_t    getMicMute(bool* state);



virtual status_t    setParameters(const String8& keyValuePairs);

virtual String8     getParameters(const String8& keys);



// create I/O streams

virtual AudioStreamOut* openOutputStream(

                        uint32_t devices,

                        int *format=0,

                        uint32_t *channels=0,

                        uint32_t *sampleRate=0,

                        status_t *status=0);



virtual AudioStreamIn* openInputStream(

                        uint32_t devices,

                        int *format,

                        uint32_t *channels,

                        uint32_t *sampleRate,

                        status_t *status,
```

```
                AudioSystem::audio_in_acoustics acoustics);


    virtual    void        closeOutputStream(AudioStreamOut* out);

    virtual    void        closeInputStream(AudioStreamIn* in);


    virtual size_t getInputBufferSize(uint32_t sampleRate, int format, int
    channelCount);


            void        clearCurDevice() { mCurSndDevice = -1; }


    protected:

        virtual status_t    dump(int fd, const Vector<String16>& args);


    private:

        status_t    doAudioRouteOrMute(uint32_t device);

        status_t    setMicMute_nosync(bool state);

        status_t    checkMicMute();

        status_t    dumpInternals(int fd, const Vector<String16>& args);

        uint32_t    getInputSampleRate(uint32_t sampleRate);

        bool        checkOutputStandby();
```

```
status_t    get_mMode();

status_t    get_mRoutes();

status_t    set_mRecordState(bool onoff);

status_t    get_snd_dev();

status_t    doRouting(AudioStreamInMSM72xx *input);

AudioStreamInMSM72xx*   getActiveInput_l();

size_t      getBufferSize(uint32_t sampleRate, int channelCount);


class AudioStreamOutMSM72xx : public AudioStreamOut {

public:

                    AudioStreamOutMSM72xx();

    virtual             ~AudioStreamOutMSM72xx();

            status_t    set(AudioHardware* mHardware,

                        uint32_t devices,

                        int *pFormat,

                        uint32_t *pChannels,

                        uint32_t *pRate);

virtual uint32_t    sampleRate() const { return mSampleRate; }

// Changed to 1024 from 4800

virtual size_t      bufferSize() const { return mBufferSize; }
```

```
        virtual uint32_t    channels() const { return mChannels; }

        virtual int         format() const { return AUDIO_HW_OUT_FORMAT; }

        virtual uint32_t    latency() const { return
(1000*AUDIO_HW_NUM_OUT_BUF*(bufferSize()/frameSize()))/sampleRate()+AUDIO_HW_OUT_LATENC
Y_MS; }

        virtual status_t    setVolume(float left, float right) { return
INVALID_OPERATION; }

    virtual ssize_t     write(const void* buffer, size_t bytes);

    virtual status_t    standby();

    virtual status_t    dump(int fd, const Vector<String16>& args);

        bool        checkStandby();

    virtual status_t    setParameters(const String8& keyValuePairs);

    virtual String8     getParameters(const String8& keys);

        uint32_t    devices() { return mDevices; }

    virtual status_t    getRenderPosition(uint32_t *dspFrames);

        virtual status_t    openDriver();


    private:

            AudioHardware* mHardware;

            int         mFd;

            int         mStartCount;
```

```
        int         mRetryCount;

        bool        mStandby;

        uint32_t    mDevices;

        uint32_t    mChannels;

        uint32_t    mSampleRate;

        size_t      mBufferSize;
};


class AudioStreamInMSM72xx : public AudioStreamIn {

public:

    enum input_state {

        AUDIO_INPUT_CLOSED,

        AUDIO_INPUT_OPENED,

        AUDIO_INPUT_STARTED

    };


                    AudioStreamInMSM72xx();

virtual             ~AudioStreamInMSM72xx();

        status_t    set(AudioHardware* mHardware,

                        uint32_t devices,
```

```
                int *pFormat,
                uint32_t *pChannels,
                uint32_t *pRate,
                AudioSystem::audio_in_acoustics acoustics);
    virtual size_t      bufferSize() const { return mBufferSize; }
    virtual uint32_t    channels() const { return mChannels; }
    virtual int         format() const { return mFormat; }
    virtual uint32_t    sampleRate() const { return mSampleRate; }
    virtual status_t    setGain(float gain) { return INVALID_OPERATION; }
    virtual ssize_t     read(void* buffer, ssize_t bytes);
    virtual status_t    dump(int fd, const Vector<String16>& args);
    virtual status_t    standby();
    virtual status_t    setParameters(const String8& keyValuePairs);
    virtual String8     getParameters(const String8& keys);
    virtual unsigned int  getInputFramesLost() const { return 0; }
            uint32_t    devices() { return mDevices; }
            int         state() const { return mState; }

private:
            AudioHardware* mHardware;
```

```
        int         mFd;

        int         mState;

        int         mRetryCount;

        int         mFormat;

        uint32_t    mChannels;

        uint32_t    mSampleRate;

        size_t      mBufferSize;

        AudioSystem::audio_in_acoustics mAcoustics;

        uint32_t    mDevices;

        bool mFirstread;

};



    static const uint32_t inputSamplingRates[];

    bool        mRecordState;

    bool        mInit;

    bool        mMicMute;

    bool        mBluetoothNrec;

    uint32_t    mBluetoothIdTx;

    uint32_t    mBluetoothIdRx;

    AudioStreamOutMSM72xx*  mOutput;
```

```
SortedVector <AudioStreamInMSM72xx*>    mInputs;


msm_bt_endpoint *mBTEndpoints;
int mNumBTEndpoints;
int mCurSndDevice;
uint32_t mVoiceVolume;
        bool        mDualMicEnabled;
        int         mTtyMode;

    friend class AudioStreamInMSM72xx;
        Mutex       mLock;
        uint32_t        mRoutes[AudioSystem::NUM_MODES];
};
```

2.2.6 三星 Android 的 Audio HAL 的实现方式

2.2.6.1 三星 Android 的架构

三星的 Audio HAL 是基于 ALSA 驱动（ / dev/audio/）,也是通过继承 AudioHardwareInterface, 实现 AudioStreamOut, AudioStreamIn 和 audioHardwareBase（继承 AudioHardwareInterface ）来实现 HAL 。具体架构如图 7。

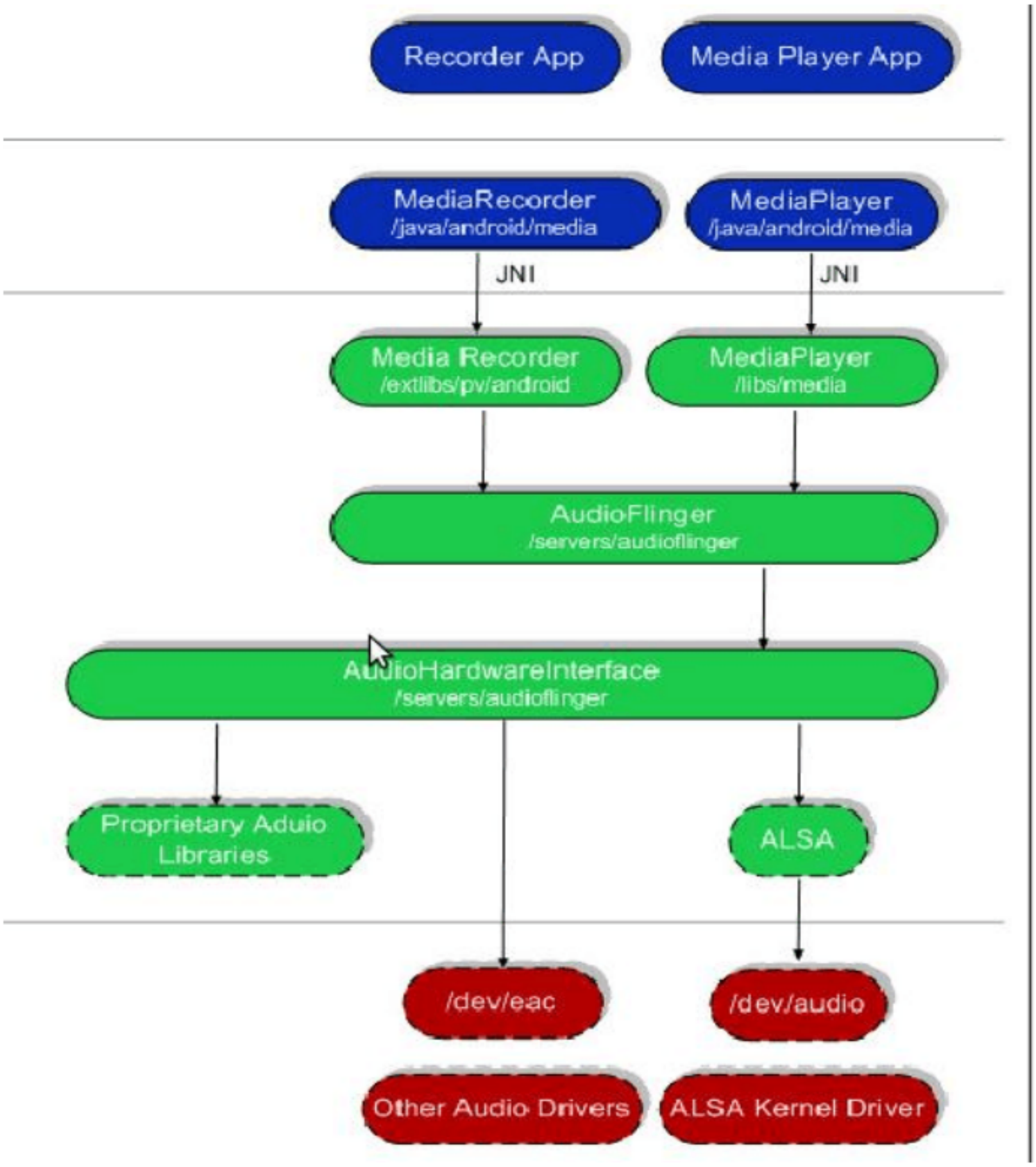

图 7

2.2.6.2 三星针对 C110 Audio 的 HAl 具体实现

1) **Audio HAL** 实现代码在 **external/libaudio/,**等同于高通的 **hardware/msm7k/libaudio-qsd8k/,** 核心代码是 **AudioHardwareALSA.h** 、**AudioHardwareALS.cpp,** 编译生成 **libaudio.so** 和 **libaudiopolicy.so,C110**平台 **Audio** 驱动只支持 **16** 位 **PCM** 数据格式，而高通平台则支持 **PCM** 、**AMR NB** 、**EVRC** 、**QCELP** 、**AAC** 等多种格式；

2) Audio HAl 的代码实现符合 Google 架构，是通过定义 AudioStreamOutALSA、AudioStreamInALSA、AudioHardwareALSA这三个类，这三个类通过 AudioStreamOut, AudioStreamIn 和 audioHardwareBase（继承 AudioHardwareInterface ）来实现自己的HAL，具体代码如下：

```
class AudioStreamOutALSA : public AudioStreamOut, public ALSAStreamOps

 {

    public:

                        AudioStreamOutALSA(AudioHardwareALSA *parent);

  virtual               ~AudioStreamOutALSA();



 status_t                           set(int *format,

                                    uint32_t *channelCount,

                                    uint32_t *sampleRate){

return ALSAStreamOps::set(format, channelCount, sampleRate);

 }

    virtual uint32_t        sampleRate() const

    {

        return ALSAStreamOps::sampleRate();

    }
```

```
virtual size_t          bufferSize() const

{

    return ALSAStreamOps::bufferSize();

}

//virtual int             channelCount() const;

virtual uint32_t            channels() const;


virtual int             format() const

{

    return ALSAStreamOps::format();

}


virtual uint32_t        latency() const;


        virtual ssize_t         write(const void *buffer, size_t bytes);

        virtual status_t        dump(int fd, const Vector<String16>& args);

        virtual status_t        setDevice(int mode, uint32_t newDevice, uint32_t
audio_mode);

    virtual status_t    setVolume(float left, float right); //Tushar: New arch
```

```
        status_t            setVolume(float volume);


        status_t            standby();

        bool                isStandby();

        virtual status_t    setParameters(const String8& keyValuePairs);

        virtual String8     getParameters(const String8& keys);


        virtual status_t    getRenderPosition(uint32_t *dspFrames);


private:

    AudioHardwareALSA       *mParent;

        bool                mPowerLock;

};


class AudioStreamInALSA : public AudioStreamIn, public ALSAStreamOps

{

    public:
```

```
                AudioStreamInALSA(AudioHardwareALSA *parent);

    virtual          ~AudioStreamInALSA();


status_t                    set(int *format,

                            uint32_t *channelCount,

                            uint32_t *sampleRate){

return ALSAStreamOps::set(format, channelCount, sampleRate);

}


  virtual uint32_t       sampleRate() const {

      return ALSAStreamOps::sampleRate();

  }


  virtual size_t         bufferSize() const

  {

      return ALSAStreamOps::bufferSize();

  }


  virtual uint32_t           channels() const

  {
```

```
        return ALSAStreamOps::channelCount();

    }


    virtual int             format() const

    {

        return ALSAStreamOps::format();

    }


    virtual ssize_t         read(void* buffer, ssize_t bytes);

    virtual status_t        dump(int fd, const Vector<String16>& args);

    virtual status_t        setDevice(int mode, uint32_t newDevice, uint32_t
audio_mode);


    virtual status_t        setGain(float gain);

     virtual status_t        standby();


    virtual status_t    setParameters(const String8& keyValuePairs);

    virtual String8     getParameters(const String8& keys);


   virtual unsigned int  getInputFramesLost() const { return 0; }
```

```
    private:
        AudioHardwareALSA *mParent;
};

class AudioHardwareALSA : public AudioHardwareBase
{
    public:
                        AudioHardwareALSA();
        virtual             ~AudioHardwareALSA();

        /**
         * check to see if the audio hardware interface has been initialized.
         * return status based on values defined in include/utils/Errors.h
         */
virtual status_t        initCheck();

/**
 * put the audio hardware into standby mode to conserve power. Returns
 * status based on include/utils/Errors.h
```

```
     */
    virtual status_t        standby();

    /** set the audio volume of a voice call. Range is between 0.0 and 1.0 */
    virtual status_t        setVoiceVolume(float volume);

    /**
     * set the audio volume for all audio activities other than voice call.
     * Range between 0.0 and 1.0. If any value other than NO_ERROR is returned,
     * the software mixer will emulate this capability.
     */
    virtual status_t        setMasterVolume(float volume);

    // mic mute
    virtual status_t        setMicMute(bool state);
    virtual status_t        getMicMute(bool* state);

    /** This method creates and opens the audio hardware output stream */
    virtual AudioStreamOut* openOutputStream(
                                uint32_t devices,
```

```
                int *format=0,
                uint32_t *channels=0,
                uint32_t *sampleRate=0,
                status_t *status=0);
    virtual    void        closeOutputStream(AudioStreamOut* out);

/** This method creates and opens the audio hardware input stream */
virtual AudioStreamIn* openInputStream(
                uint32_t devices,
                int *format,
                uint32_t *channels,
                uint32_t *sampleRate,
                status_t *status,
                AudioSystem::audio_in_acoustics acoustics);
    virtual    void        closeInputStream(AudioStreamIn* in);


protected:
    /**
```

```
        * doRouting actually initiates the routing. A call to setRouting

        * or setMode may result in a routing change. The generic logic calls

        * doRouting when required. If the device has any special requirements
these

        * methods can be overriden.

        */

        virtual status_t    doRouting();



        virtual status_t    dump(int fd, const Vector<String16>& args);



        friend class AudioStreamOutALSA;

        friend class AudioStreamInALSA;



        ALSAMixer           *mMixer;

            AudioStreamOutALSA *mOutput;

            AudioStreamInALSA  *mInput;



    private:

        Mutex               mLock;

    };
```

// ---

};

3) 此外在 **AudioHardwareALSA.h** 中还定义了类 **ALSAMixer** 、 **ALSAControl** 和 **ALSAStreamOps** 分别用来控制 **C110** 平新增的功能。

```
class AudioHardwareALSA;


// ---------------------------------------------------------------------


class ALSAMixer
{
    public:
                        ALSAMixer();
        virtual         ~ALSAMixer();
        bool            isValid()
{ return !!mMixer[SND_PCM_STREAM_PLAYBACK]; }
        status_t        setMasterVolume(float volume);
        status_t        setMasterGain(float gain);
```

```
        status_t        setVolume(uint32_t device, float volume);

        status_t        setGain(uint32_t device, float gain);



        status_t        setCaptureMuteState(uint32_t device, bool state);

        status_t        getCaptureMuteState(uint32_t device, bool *state);

        status_t        setPlaybackMuteState(uint32_t device, bool state);

        status_t        getPlaybackMuteState(uint32_t device, bool *state);


    private:

        snd_mixer_t         *mMixer[SND_PCM_STREAM_LAST+1];

};


class ALSAControl

{

 public:

                        ALSAControl(const char *device = "default");

        virtual         ~ALSAControl();


        status_t        get(const char *name, unsigned int &value, int
index = 0);
```

```
        status_t            set(const char *name, unsigned int value, int index
    = -1);


      private:

        snd_ctl_t           *mHandle;

};


class ALSAStreamOps

{

    protected:

        friend class AudioStreamOutALSA;

        friend class AudioStreamInALSA;


        struct StreamDefaults

        {

        const char *        devicePrefix;

        snd_pcm_stream_t    direction;      // playback or capture

        snd_pcm_format_t    format;

        int                 channels;

        uint32_t            sampleRate;

        unsigned int        latency;        // Delay in usec
```

```
    unsigned int        bufferSize;      // Size of sample buffer
};

                        ALSAStreamOps();
virtual                 ~ALSAStreamOps();

 status_t               set(int *format,
                        uint32_t *channels,
                        uint32_t *rate);
 virtual uint32_t       sampleRate() const;
 status_t               sampleRate(uint32_t rate);
 virtual size_t         bufferSize() const;
 virtual int            format() const;
int                     getAndroidFormat(snd_pcm_format_t format);

 virtual int            channelCount() const;
 status_t               channelCount(int channels);
uint32_t                        getAndroidChannels(int channels);

 status_t               open(int mode, uint32_t device);
```

```
        void                close();

        status_t            setSoftwareParams();

        status_t            setPCMFormat(snd_pcm_format_t format);

        status_t            setHardwareResample(bool resample);

    const char          *streamName();

    virtual status_t      setDevice(int mode, uint32_t device, uint32_t
audio_mode);



    const char          *deviceName(int mode, uint32_t device);



    void                setStreamDefaults(StreamDefaults *dev) {

        mDefaults = dev;

    }

    Mutex               mLock;



private:

    snd_pcm_t           *mHandle;

    snd_pcm_hw_params_t    *mHardwareParams;
```

```
        snd_pcm_sw_params_t    *mSoftwareParams;

        int                    mMode;

        uint32_t               mDevice;



        StreamDefaults         *mDefaults;

#ifdef SLSI_RESAMPLER

            uint32_t           mReqSampleRate;

#endif

    };
```

参考资料

1) 庄渭峰 高通 HAL分析 _20100929
2) 庄渭峰 SMDKC110分析 _20101020
3) 韩超/ 梁泉, 《 Android 系统原理及开发要点详解》, 2010
4) Qualcomm, Android Eclair Overview (Version A), 2009.12
5) Qualcomm, Android Policy Manager Quick Start Guide 2010.7
6) Qualcomm, QSD8x50_linux_Audio_Overview
7) Samsung, Audroid Android Open Source
8) Samsung, Android Intro