

Audiotrack and audioflinger

<https://blog.csdn.net/zyuanyun/article/details/60890534>

<https://www.cnblogs.com/CoderTian/p/6446187.html>

MultiMedia 负责音视频的编解码，MultiMedia 将解码后的数据通过 AudioTrack 输出，而 AudioRecord 采集的录音数据交由 MultiMedia 进行编码。MediaPlayer 可以播放多种格式的音源，如 mp3、flac、wma、ogg、wav 等，而 AudioTrack 只能播放解码后的 PCM 数据流。

传输延迟（latency）的概念：传输延迟表示一个周期的音频数据的传输时间。可能有些读者一脸懵逼，一个周期的音频数据，这又是啥？我们再引入周期（period）的概念：Linux ALSA 把数据缓冲区划分为若干个块，dma 每传输完一个块上的数据即发出一个硬件中断，cpu 收到中断信号后，再配置 dma 去传输下一个块上的数据；一个块即是一个周期，周期大小（periodSize)即是一个数据块的帧数。再回到传输延迟（latency），传输延迟等于周期大小除以采样率，即 latency = periodSize / sampleRate。



getMinBufferSize() 接口，字面意思是返回最小数据缓冲区的大小，它是声音能正常播放的最低保障，从函数参数来看，返回值取决于采样率、采样深度、声道数这三个属性。MODE\_STREAM 模式下，应用程序重点参考其返回值然后确定分配多大的数据缓冲区。如果数据缓冲区分配得过小，那么播放声音会频繁遭遇 underrun，underrun 是指生产者（AudioTrack）提供数据的速度跟不上消费者（AudioFlinger::PlaybackThread）消耗数据的速度，反映到现实的后果就是声音断续卡顿，严重影响听觉体验。





**AudioFlinger 概述**

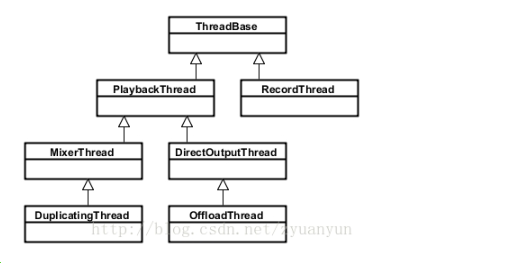
从 Android 7.0 开始，AudioFlinger 在系统启动时由 audioserver 加载, 详见 frameworks/av/media/audioserver/main\_audioserver.cpp：main\_audioserver.cpp 编译生成的可执行文件存放在 /system/bin/audioserver，系统启动时由 init 进程运行，详见 frameworks/av/media/audioserver/audioserver.rc：AudioFlinger 服务启动后，其他进程可以通过 ServiceManager 来获取其代理对象 IAudioFlinger，通过 IAudioFlinger 可以向 AudioFlinger 发出各种服务请求，从而完成自己的音频业务。



可以归纳出 AudioFlinger 响应的服务请求主要有：

获取硬件设备的配置信息

* 音量调节
* 静音操作
* 音频模式切换
* 音频参数设置
* 输入输出流设备管理
* 音频流管理



ThreadBase：PlaybackThread 和 RecordThread 的基类

RecordThread：录制线程类，由 ThreadBase 派生

PlaybackThread：回放线程基类，同由 ThreadBase 派生

MixerThread：混音回放线程类，由 PlaybackThread 派生，负责处理标识为 AUDIO\_OUTPUT\_FLAG\_PRIMARY、AUDIO\_OUTPUT\_FLAG\_FAST、AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER 的音频流，MixerThread 可以把多个音轨的数据混音后再输出

DirectOutputThread：直输回放线程类，由 PlaybackThread 派生，负责处理标识为 AUDIO\_OUTPUT\_FLAG\_DIRECT 的音频流，这种音频流数据不需要软件混音，直接输出到音频设备即可

DuplicatingThread：复制回放线程类，由 MixerThread 派生，负责复制音频流数据到其他输出设备，使用场景如主声卡设备、蓝牙耳机设备、USB 声卡设备同时输出

OffloadThread：硬解回放线程类，由 DirectOutputThread 派生，负责处理标识为 AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 的音频流，这种音频流未经软件解码的（一般是 MP3、AAC 等格式的数据），需要输出到硬件解码器，由硬件解码器解码成 PCM 数据

PlaybackThread 中有个极为重要的函数 threadLoop()，当 PlaybackThread 被强引用时，threadLoop() 会真正运行起来进入循环主体，处理音频流数据相关事务，

bool AudioFlinger::PlaybackThread::threadLoop()->prepareTracks\_l()->threadLoop\_mix ->threadLoop\_write()：

PlaybackThread 实例与输出流设备是一一对应的，比方说 OffloadThread 只会将音频数据输出到 compress\_offload 设备中，MixerThread(with FastMixer) 只会将音频数据输出到 low\_latency 设备中。

从 Audio HAL 中，我们通常看到如下 4 种输出流设备，分别对应着不同的播放场景：

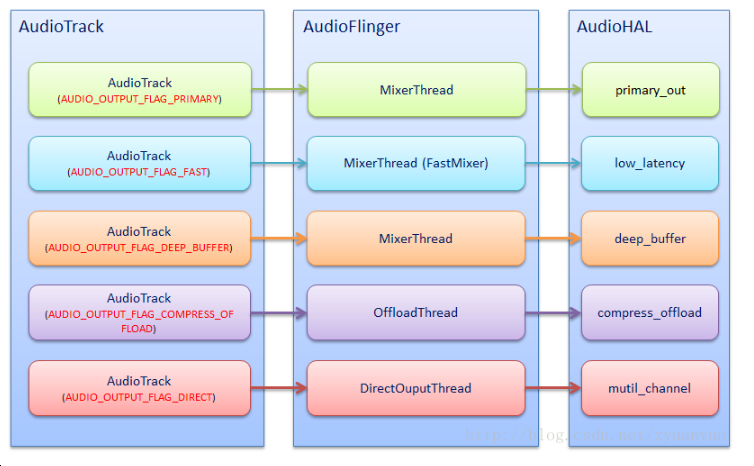
primary\_out：主输出流设备，用于铃声类声音输出，对应着标识为 AUDIO\_OUTPUT\_FLAG\_PRIMARY 的音频流和一个 MixerThread 回放线程实例

low\_latency：低延迟输出流设备，用于按键音、游戏背景音等对时延要求高的声音输出，对应着标识为 AUDIO\_OUTPUT\_FLAG\_FAST 的音频流和一个 MixerThread 回放线程实例

deep\_buffer：音乐音轨输出流设备，用于音乐等对时延要求不高的声音输出，对应着标识为 AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER 的音频流和一个 MixerThread 回放线程实例

compress\_offload：硬解输出流设备，用于需要硬件解码的数据输出，对应着标识为 AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 的音频流和一个 OffloadThread 回放线程实例

可能有人产生这样的疑问：既然 primary\_out 设备一直保持打开，那么能耗岂不是很大？这里阐释一个概念：输出流设备属于逻辑设备，并不是硬件设备。所以即使输出流设备一直保持打开，只要硬件设备不工作，那么就不会影响能耗。那么硬件设备什么时候才会打开呢？答案是 PlaybackThread 将音频数据写入到输出流设备时。



输出流设备决定了它对应的 PlaybackThread 是什么类型。怎么理解呢？意思是说：只有支持了该类型的输出流设备，那么该类型的 PlaybackThread 才有可能被创建。举个例子：只有硬件上具备硬件解码器，系统才建立 compress\_offload 设备，然后播放 mp3 格式的音乐文件时，才会创建 OffloadThread 把数据输出到 compress\_offload 设备上；反之，如果硬件上并不具备硬件解码器，系统则不应该建立 compress\_offload 设备，那么播放 mp3 格式的音乐文件时，通过 MixerThread 把数据输出到其他输出流设备上。

系统启动时，会检查并保存输入输出流设备的支持信息；播放器在播放 mp3 文件时，首先看 compress\_offload 设备是否支持了，如果支持，那么不进行软件解码，直接把数据标识为 AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD；如果不支持，那么先进行软件解码，然后把解码好的数据标识为 AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER，前提是 deep\_buffer 设备是支持了的；如果 deep\_buffer 设备也不支持，那么把数据标识为 AUDIO\_OUTPUT\_FLAG\_PRIMARY。

系统启动时，就已经打开 primary\_out、low\_latency、deep\_buffer 这三种输出流设备，并创建对应的 MixerThread 了；而此时 DirectOutputThread 与 OffloadThread 不会被创建，直到标识为 AUDIO\_OUTPUT\_FLAG\_DIRECT/AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 的音频流需要输出时，才开始创建 DirectOutputThread/OffloadThread 和打开 direct\_out/compress\_offload 设备。

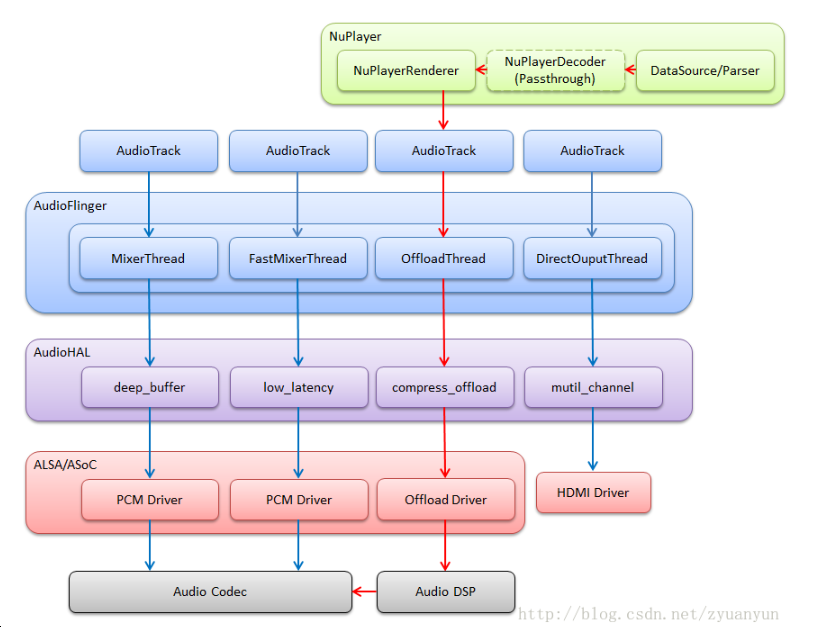
如：AudioPolicyManager::AudioPolicyManager(AudioPolicyClientInterface \*clientInterface)-> mpClientInterface->openOutput/openInput

其中 mpClientInterface->openOutput() 最终会调用到 AudioFlinger::openOutput()：打开输出流设备，并创建 PlaybackThread 对象:

AudioFlinger::openOutput-> p<PlaybackThread> thread = openOutput\_l{ // 分配全局唯一的 audio\_io\_handle\_t，可以理解它是回放线程的索引号}// outHwDev->openOutputStream{打开音频输出流设备，HAL 层根据 flags 选择打开相关类型的输出流设备}// thread = new OffloadThread{ AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 音频流，创建 OffloadThread 实例}// mPlaybackThreads.add(\*output, thread){ // 把 audio\_io\_handle\_t 和 PlaybackThread 添加到键值对向量 mPlaybackThreads 中键值对向量 mPlaybackThreads 中，由于 audio\_io\_handle\_t 和 PlaybackThread 是一

一对应的关系，所以拿到一个 audio\_io\_handle\_t，就能找到它对应的 PlaybackThread所以可以理解 audio\_io\_handle\_t 为 PlaybackThread 的索引号}.

OffloadThread 的使用率还是很高的，为什么不让 OffloadThread/compress\_offload 设备也进入待命状态呢？



Offload 音频流是未经 NuPlayerDecoder 进行解码的（NuPlayerDecoder 设置了 Passthrough 模式），所以必须把这些音频流数据送到 DSP，让 DSP 对其解码，解码后的 PCM 数据再送到 Codec 输出。

AudioTrack 构造函数有个 offloadInfo 的参数，NuPlayer DataSource/Parser 解析 mp3、flac 等文件得到数据编码信息，并在构造 AudioTrack 实例时作为参数传入，AudioFlinger 将基于这些编码信息打开 compress\_offload 设备。

NuPlayer Offload Playback 初始化流程:

1. NuPlayer根据源解析器和解码器创建的类型获取文件格式元数据,对于compress offload, 创建DecoderPassThrough，使软件解码器进入直通模式而不执行解码。
2. Nuplayer 调用CanOffloadStream, CanOffloadStream调用 APM API isOffloadSupported决定是否可以offload
3. NuPlayer创建一个新的NuPlayerRenderer，然后创建一个新的AudioTrack。 根据NuPlayer从APM收到的信息，它设置标志FLAG\_OFFLOAD\_AUDIO，传递给NuPlayerRenderer。
4. AudioTrack然后调用getOutput（），它从APM获取配置文件信息。然后APM调用openOutput（），它转到AudioFlinger。基于AudioPlayer先前设置的AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD标志，AudioFlinger创建offload thread，AudioFlinger还调用*open\_output\_stream（）进*入HAL。然后，HAL创建offload callback thread来通知write.
5. AudioTrack 创建track为这个流
6. NuplayerRenderer 向Audiosink发送一个开始命令，Audiosink调用start\_output\_stream().Hal 发送合适的MIXER Control到BE platform driver来配置输出设备。
7. HAL 打开compression driver 并且通过IOCTL SNDRV\_COMPRESS\_SET\_PARAMS发送codec和stream参数到compression driver。Compression driver 适当打开ASM session ，ASM driver分配buffers为这个session。

AudioFlinger 音频流管理由 AudioFlinger::PlaybackThread::Track 实现，Track 与 AudioTrack 是一对一的关系，一个 AudioTrack 创建后，那么 AudioFlinger 会创建一个 Track 与之对应；PlaybackThread 与 AudioTrack/Track 是一对多的关系，一个 PlaybackThread 可以挂着多个 Track。

AudioTrack 创建后，AudioPolicyManager 根据 AudioTrack 的输出标识和流类型，找到对应的输出流设备和 PlaybackThread（如果没有找到的话，则系统会打开对应的输出流设备并新建一个 PlaybackThread），然后创建一个 Track 并挂到这个 PlaybackThread 下面。

PlaybackThread 有两个私有成员向量与此强相关：

mTracks：该 PlaybackThread 创建的所有 Track 均添加保存到这个向量中

mActiveTracks：只有需要播放（设置了 ACTIVE 状态）的 Track 会添加到这个向量中；PlaybackThread 会从该向量上找到所有设置了 ACTIVE 状态的 Track，把这些 Track 数据混音后写到输出流设备

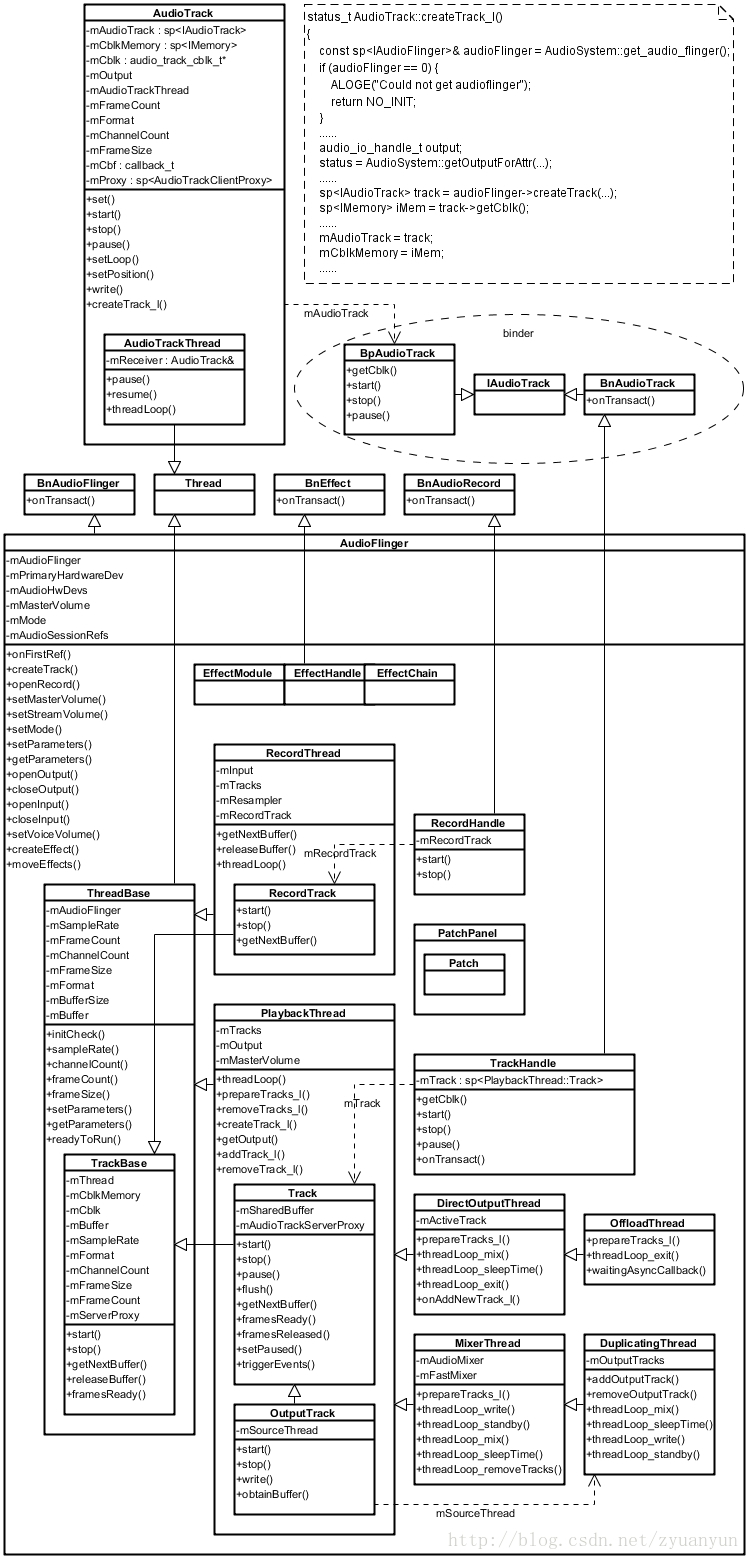
音频流控制最常用的三个接口：

AudioFlinger::PlaybackThread::Track::start：开始播放：把该 Track 置 ACTIVE 状态，然后添加到 mActiveTracks 向量中，最后调用 AudioFlinger::PlaybackThread::broadcast\_l() 告知 PlaybackThread 情况有变

AudioFlinger::PlaybackThread::Track::stop：停止播放：把该 Track 置 STOPPED 状态，最后调用 AudioFlinger::PlaybackThread::broadcast\_l() 告知 PlaybackThread 情况有变

AudioFlinger::PlaybackThread::Track::pause：暂停播放：把该 Track 置 PAUSING 状态，最后调用 AudioFlinger::PlaybackThread::broadcast\_l() 告知 PlaybackThread 情况有变

AudioFlinger::PlaybackThread::threadLoop() 得悉情况有变后，调用 prepareTracks\_l() 重新准备音频流和混音器：ACTIVE 状态的 Track 会添加到 mActiveTracks，此外的 Track 会从 mActiveTracks 上移除出来，然后重新准备 AudioMixer。



AudioFlinger::PlaybackThread::Track：音频流管理类，创建一块匿名共享内存用于 AudioTrack 与 AudioFlinger 之间的数据交换（方便起见，这块匿名共享内存，以后均简单称为 FIFO），同时实现 start()、stop()、pause() 等音频流常用控制手段；

AudioFlinger::TrackHandle：Track 对象只负责音频流管理业务，对外并没有提供跨进程的 Binder 调用接口，而应用进程又需要对音频流进行控制，所以需要一个对象来代理 Track 的跨进程通讯，这个角色就是 TrackHandle，AudioTrack 通过它与 Track 交互

AudioTrack：Android 音频系统对外提供的一个 API 类，负责音频流数据输出；每个音频流对应着一个 AudioTrack 实例，不同输出标识的 AudioTrack 会匹配到不同的 AudioFlinger::PlaybackThread；AudioTrack 与 AudioFlinger::PlaybackThread 之间通过 FIFO 来交换音频数据，AudioTrack 是 FIFO 生产者，AudioFlinger::PlaybackThread 是 FIFO 消费者

AudioTrack::AudioTrackThread：数据传输模式为 TRANSFER\_CALLBACK 时，需要创建该线程，它通过调用 audioCallback 回调函数主动从用户进程处索取数据并填充到 FIFO 上；数据传输模式为 TRANSFER\_SYNC 时，则不需要创建这个线程，因为用户进程会持续调用 AudioTrack.write() 填充数据到 FIFO；数据传输模式为 TRANSFER\_SHARED 时，也不需要创建这个线程，因为用户进程会创建一块匿名共享内存，并把要播放的音频数据一次性拷贝到这块匿名共享内存上了

IAudioTrack：IAudioTrack 是链结 AudioTrack 与 AudioFlinger 的桥梁；它在 AudioTrack 端的对象是 BpAudioTrack，在 AudioFlinger 端的对象是 BnAudioTrack，从图中不难看出，AudioFlinger::TrackHandle 继承自 BnAudioTrack，而 AudioFlinger::TrackHandle 恰恰是AudioFlinger::PlaybackThread::Track 的代理对象，所以 AudioTrack 得到 IAudioTrack 实例后，就可以调用 IAudioTrack 的接口与 AudioFlinger::PlaybackThread::Track 交互

**audio\_io\_handle\_t**：

当打开输出流设备及创建 PlaybackThread 时，系统会分配一个全局唯一的值作为 audio\_io\_handle\_t，并把 audio\_io\_handle\_t 和 PlaybackThread 添加到键值对向量 mPlaybackThreads 中，由于 audio\_io\_handle\_t 和 PlaybackThread 是一一对应的关系，因此拿到一个 audio\_io\_handle\_t，就能遍历键值对向量 mPlaybackThreads 找到它对应的 PlaybackThread，可以简单理解 audio\_io\_handle\_t 为 PlaybackThread 的索引号或线程 id。由于 audio\_io\_handle\_t 具有 PlaybackThread 索引特性，所以应用进程想获取 PlaybackThread 某些信息的话，只需要传入对应的 audio\_io\_handle\_t 即可。

**AudioTrack 构造过程**

1. 如果 cbf（audioCallback 回调函数）非空，那么创建 AudioTrackThread 线程（在set函数里创建）处理 audioCallback 回调函数（MODE\_STREAM 模式时，cbf 为空）；
2. 根据 streamType（流类型）、flags（输出标识）等参数调AudioSystem::getOutputForAttr()；经过一系列的调用，进入 AudioPolicyManager::getOutputForDevice()：

2.1如果输出标识置了 AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 或 AUDIO\_OUTPUT\_FLAG\_DIRECT，那么最终调用 AudioFlinger::openOutput() 打开输出标识对应的输出流设备并创建相应的 PlaybackThread，保存该 PlaybackThread 对应的 audio\_io\_handle\_t 给 AudioTrack；

2.2如果输出标识是其他类型，那么根据策略选择一个输出流设备和 PlaybackThread，并保存该 PlaybackThread 对应的 audio\_io\_handle\_t 给 AudioTrack；系统启动时，就已经打开 primary\_out、low\_latency、deep\_buffer 这三种输出流设备，并创建对应的 PlaybackThread 了；

3.通过 Binder 机制调用 AudioFlinger::createTrack()（注意 step2 中 AudioTrack 已经拿到一个 audio\_io\_handle\_t 了，此时把这个 audio\_io\_handle\_t 传入给 createTrack()）：

3.1根据传入的 audio\_io\_handle\_t 找到它对应的 PlaybackThread；

3.2PlaybackThread 新建一个音频流管理对象 Track；Track 构造时会分配一块匿名共享内存用于 AudioFlinger 与 AudioTrack 的数据交换缓冲区（FIFO）及其控制块（audio\_track\_cblk\_t），并创建一个 AudioTrackServerProxy 对象（PlaybackThread 将使用它从 FIFO 上取得可读数据的位置）；

3.3最后新建一个 Track 的通讯代理 TrackHandle，并以 IAudioTrack 作为返回值给 AudioTrack

4.通过 IAudioTrack 接口，取得 AudioFlinger 中的 FIFO 控制块（audio\_track\_cblk\_t），由此再计算得到 FIFO 的首地址；

5. 创建一个 AudioTrackClientProxy 对象（AudioTrack 将使用它从 FIFO 上取得可用空间的位置）；

**AudioTrack 由此建立了和 AudioFlinger 的全部联系工作：**

通过 IAudioTrack 接口可以控制该音轨的状态，例如 start、stop、pause;

持续写入数据到 FIFO 上，实现音频连续播放;

通过 audio\_io\_handle\_t，可以找到它对应的 PlaybackThread，从而查询该 PlaybackThread 的相关信息，如所设置的采样率、格式等等;

AudioTrack::AudioTrack(

audio\_stream\_type\_t streamType, // 音频流类型：如 Music、Voice-Call、DTMF、Alarm 等等

uint32\_t sampleRate, // 采样率：如 16KHz、44.1KHz、48KHz 等等

audio\_format\_t format, // 音频格式：如 PCM、MP3、AAC 等等

audio\_channel\_mask\_t channelMask, // 声道数：如 Mono（单声道）、Stereo（双声道）

const sp<IMemory>& sharedBuffer, // 共享内存缓冲区：数据模式是 MODE\_STATIC 时使用，数据模式是 MODE\_STREAM 时为空

audio\_output\_flags\_t flags, // 输出标识位，详见 AUDIO\_OUTPUT\_FLAG 描述

callback\_t cbf, // 回调函数

void\* user, // 回调函数的参数

uint32\_t notificationFrames,

int sessionId,

transfer\_type transferType, // 数据传输类型

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

int uid,

pid\_t pid,

const audio\_attributes\_t\* pAttributes,

bool doNotReconnect)

: mStatus(NO\_INIT),

mIsTimed(false),

mPreviousPriority(ANDROID\_PRIORITY\_NORMAL),

mPreviousSchedulingGroup(SP\_DEFAULT),

mPausedPosition(0),

mSelectedDeviceId(AUDIO\_PORT\_HANDLE\_NONE)

{

mStatus = set(streamType, sampleRate, format, channelMask,

0 /\*frameCount\*/, flags, cbf, user, notificationFrames,

sharedBuffer, false /\*threadCanCallJava\*/, sessionId, transferType, offloadInfo,

uid, pid, pAttributes, doNotReconnect);

}

-> AudioTrack::set->**new** AudioTrackThread{如果 cbf 非空，那么创建 AudioTrackThread 线程处理 audioCallback 回调函数}//createTrack\_l()-> **const** sp<IAudioFlinger>& audioFlinger = AudioSystem::get\_audio\_flinger(){获取 IAudioFlinger，通过 binder 请求 AudioFlinger 服务}//AudioSystem::getOutputForAttr{ AudioSystem::getOutputForAttr() 经过一系列的调用，进入 AudioPolicyManager::getOutputForDevice();如果输出标识置了AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 或 AUDIO\_OUTPUT\_FLAG\_DIRECT;那么最终调用 AudioFlinger::openOutput() 打开输出标识对应的输出流设备并创建相关的；PlaybackThread，保存该 PlaybackThread 对应的 audio\_io\_handle\_t 给 AudioTrack；如果输出标识是其他类型，那么根据策略选择一个输出流设备和 PlaybackThread，并保存该PlaybackThread 对应的 audio\_io\_handle\_t 给 AudioTrack

}// sp<IAudioTrack> track = audioFlinger->createTrack{向 AudioFlinger 发出 createTrack 请求}// sp<IMemory> iMem = track->getCblk();{AudioFlinger 创建 Track 对象时会分配一个 FIFO，这里获取 FIFO 的控制块}//**void** \*iMemPointer = iMem->pointer();{// 匿名共享内存首地址}// mAudioTrack = track; {// 保存 AudioFlinger::PlaybackThread::Track 的代理对象 }IAudioTrack mCblkMemory = iMem; {// 保存匿名共享内存首地址}// **new** AudioTrackClientProxy{当 mSharedBuffer 为空，意味着音轨数据模式为 MODE\_STREAM，那么创建 AudioTrackClientProxy 对象}//**new** StaticAudioTrackClientProxy{当 mSharedBuffer 非空，意味着音轨数据模式为 MODE\_STATIC，那么创建 StaticAudioTrackClientProxy 对象}.sp<IAudioTrack> AudioFlinger::createTrack-> PlaybackThread \*thread = checkPlaybackThread\_l(output);{ 根据传入来的 audio\_io\_handle\_t，找到对应的 PlaybackThread}//thread->createTrack\_l{在 PlaybackThread 上创建一个音频流管理对象 Track，createTrack\_l除了为AudioTrack在AudioFlinger中申请一个Track外,还会建立两者之间的IAudioTrack桥梁

}//trackHandle = new TrackHandle(**track**);{ 创建 Track 的通讯代理 TrackHandle 并返回它}

我们看看 Track 的构造过程，主要分析数据 FIFO 及它的控制块是如何分配的：

AudioFlinger::PlaybackThread::Track::Track；

{

// 检查 FIFO 控制块（audio\_track\_cblk\_t）是否分配好了，上面代码并未分配 audio\_track\_cblk\_t

// 因此只可能是构造 TrackBase 时分配的，等下再看看 TrackBase 的构造函数

// 数据传输模式为 MODE\_STREAM 模式，创建一个 AudioTrackServerProxy 对象

// PlaybackThread 将持续使用它从 FIFO 上取得可读数据的位置

// 数据传输模式为 MODE\_STATIC 模式，创建一个 StaticAudioTrackServerProxy 对象

为 Track 分配一个名称，AudioMixer 会根据 TrackName 找到对应的 Track

}

AudioFlinger::ThreadBase::TrackBase::TrackBase

{

// 分配一块匿名共享内存

// 这里，在匿名共享内存首地址上构造了一个 audio\_track\_cblk\_t 对象

// 这样 AudioTrack 与 AudioFlinger 都能访问这个 audio\_track\_cblk\_t 对象了

// 数据传输模式为 MODE\_STREAM/TRANSFER\_SYNC 时，数据 FIFO 的分配,数据 FIFO 的首地址紧靠控制块（audio\_track\_cblk\_t）之后

//数据传输模式为 MODE\_STATIC/TRANSFER\_SHARED 时，直接指向 sharedBuffer, sharedBuffer 是应用进程分配的匿名共享内存，应用进程已经一次性把数据,写到 sharedBuffer 来了，AudioFlinger 可以直接从这里读取

}

**AudioTrack 实例构造后，应用程序接着可以写入音频数据了。**

AudioTrack：AudioTrack 在 FIFO 中找到一块可用空间，把用户传入的音频数据写入到这块可用空间上，然后更新写位置（对于 AudioFinger 来说，意味 FIFO 上有更多的可读数据了）；如果用户传入的数据量比可用空间要大，那么要把用户传入的数据拆分多次写入到 FIFO 中（AudioTrack 和 AudioFlinger 是不同的进程，AudioFlinger 同时也在不停地读取数据，所以 FIFO 可用空间是在不停变化的）

AudioFlinger：AudioFlinger 在 FIFO 中找到一块可读数据块，把可读数据拷贝到目的缓冲区上，然后更新读位置（对于 AudioTrack 来说，意味着 FIFO 上有更多的可用空间了）；如果FIFO 上可读数据量比预期的要小，那么要进行多次的读取，才能积累到预期的数据量（AudioTrack 和 AudioFlinger 是不同的进程，AudioTrack 同时也在不停地写入数据，所以 FIFO 可读的数据量是在不停变化的）

**AudioTrack 写数据的代码:**

obtainBuffer() 在 FIFO 中找到一块可用区间，memcpy() 把用户传入的音频数据拷贝到这个可用区间上，releaseBuffer() 更新写位置。

AudioTrack::write

**AudioFlinger 消费数据的流程:**

我们把焦点放在“如何从 FIFO 读取数据”节点上。我们以 DirectOutputThread/OffloadThread 为例说明（MixerThread 读数据也是类似的过程，只不过是在 AudioMixer 中进行的，）

**void** AudioFlinger::DirectOutputThread::threadLoop\_mix()

{

// getNextBuffer() 从 FIFO 上获取可读数据块;

// memcpy() 把 FIFO 可读数据拷贝到 mSinkBuffer 目的缓冲区

// releaseBuffer() 更新 FIFO 读位置

// 对于 AudioTrack 来说，意味着 FIFO 上有更多的可用空间

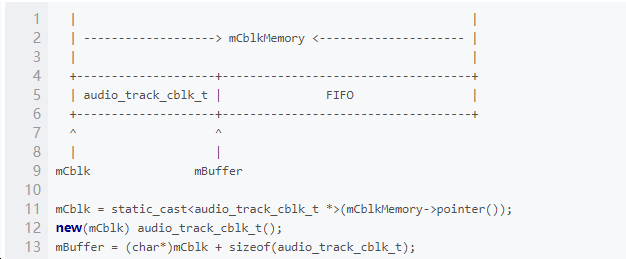
}

**环形 FIFO 管理：**

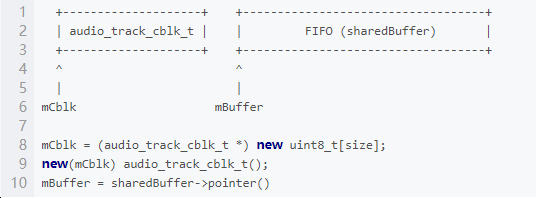
AudioTrack 与 AudioFlinger 不在同一个进程上，怎么保证读写指针的线程安全读写指针越过 FIFO 后，怎么处理?

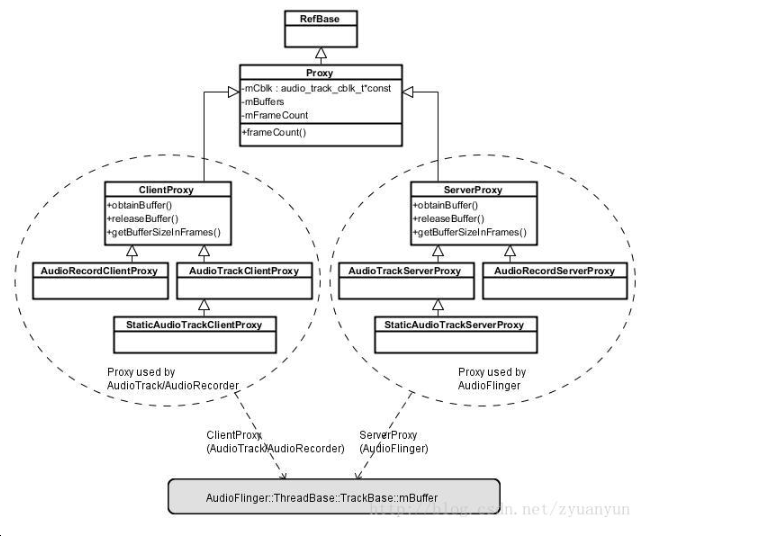
AudioTrack 写数据完成后，需要同步状态给 AudioFlinger，让 AudioFlinger 知道当前有可读数据了，而 AudioFlinger 读数据完成后，也需要同步状态给 AudioTrack，让 AudioTrack 知道当前有可用空间了；这里采取什么同步机制??

* MODE\_STREAM 模式下的匿名共享内存结构：



* MODE\_STATIC 模式下的匿名共享内存结构：





AudioTrackClientProxy：MODE\_STREAM 模式下，生产者 AudioTrack 使用它在 FIFO 中找到可用空间的位置.

AudioTrackServerProxy：MODE\_STREAM 模式下，消费者 AudioFlinger::PlaybackThread 使用它在 FIFO 中找到可读数据的位置

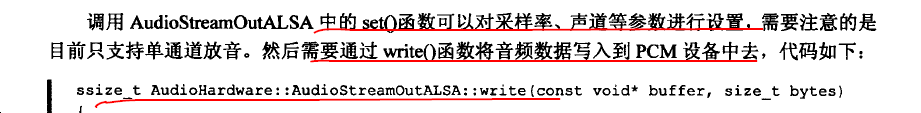
StaticAudioTrackClientProxy：MODE\_STATIC 模式下，生产者 AudioTrack 使用它在 FIFO 中找到可用空间的位置

StaticAudioTrackServerProxy：MODE\_STATIC 模式下，消费者 AudioFlinger::PlaybackThread 使用它在 FIFO 中找到可读数据的位置

AudioRecordClientProxy：消费者 AudioRecord 使用它在 FIFO 中找到可读数据的位置

AudioTrackServerProxy：生产者 AudioFlinger::RecordThread 使用它在 FIFO 中找到可用空间的位置

这里这个 Proxy 的功能大家应该能想到了吧，就是把参数封装后发送给 Binder 驱动，然后执行一系列 IPC 操作最后再取出结果返回。

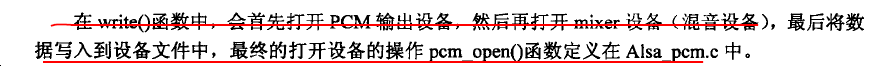


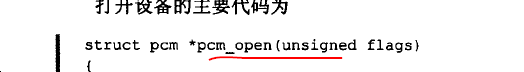
**{**



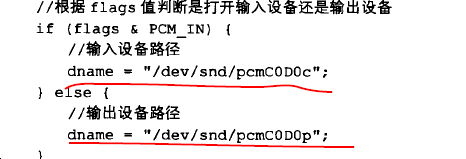


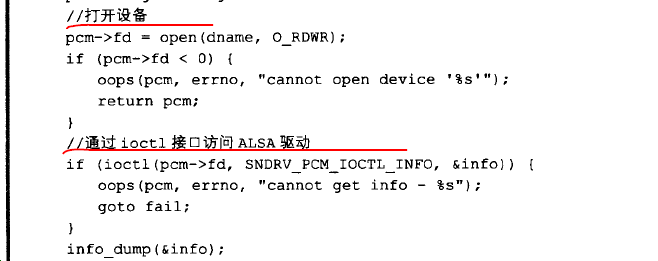
**}**



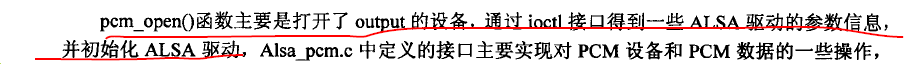


**{**



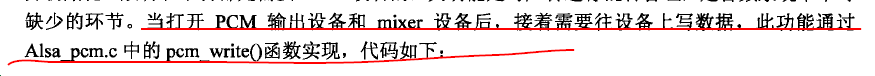


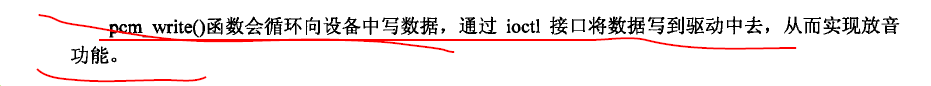
**}**











<https://blog.csdn.net/weijory/article/details/62422869>

audio HAL与[kernel](https://www.baidu.com/s?wd=kernel&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)联动过程中几个关键的函数进行分析和总结：

select\_device, 2.enable\_snd\_device, 3.enable\_audio\_route,4.pcm\_open,5.pcm\_write

播放音乐的时候AudioFlinger往hal层写数据时会调用到Hal层的out\_write函数，我们以out\_write函数为入口，层层分析上面所列的几个关键函数:

static ssize\_t out\_write(struct audio\_stream\_out \*stream, const void \*buffer,size\_t bytes)

{

...

if (out->standby)

{//第一次调用out\_write时需要打开pcm stream设备

out->standby = false;

...

ret = start\_output\_stream(out);//打开pcm stream设备

...

}

...

ret = pcm\_write(out->pcm, (void \*)buffer, bytes);//调用alsalib函数，写入音频数据

...

return bytes;//返回写入数据的字节数

}

在打开pcm stream设备的时候会调用到select\_device和pcm\_open函数:

int start\_output\_stream(struct stream\_out \*out)

{

...

out->pcm\_device\_id = platform\_get\_pcm\_device\_id(out->usecase, PCM\_PLAYBACK);

//根据out->usecase找到pcm逻辑设备编号

...

select\_devices(adev, out->usecase);//打开设备通路

...

if (out->usecase != USECASE\_AUDIO\_PLAYBACK\_OFFLOAD)

{

out->pcm = pcm\_open(adev->snd\_card, out->pcm\_device\_id,

PCM\_OUT | PCM\_MONOTONIC, &out->config);

//打开/dev/snd/pcmC0D0p 设备节点

...

}...

}

1、select\_devices函数

int select\_devices(struct audio\_device \*adev, audio\_usecase\_t uc\_id)

{

...

usecase = get\_usecase\_from\_list(adev, uc\_id);//根据uc\_id获取当前的usecase

...

if ((usecase->type == VOICE\_CALL) || (usecase->type == VOIP\_CALL) ||

(usecase->type == PCM\_HFP\_CALL))

{

out\_snd\_device = platform\_get\_output\_snd\_device(adev->platform,

usecase->stream.out->devices);

in\_snd\_device = platform\_get\_input\_snd\_device(adev->platform,

usecase->stream.out->devices);//如果当前是通话的状态下的话，获取输入和输出设备

usecase->devices = usecase->stream.out->devices;

} else {

...

if (usecase->type == PCM\_PLAYBACK) {

usecase->devices = usecase->stream.out->devices;

in\_snd\_device = SND\_DEVICE\_NONE;

out\_snd\_device = platform\_get\_output\_snd\_device(adev->platform,

usecase->stream.out->devices);

...//如果当前是音乐播放的话只选择输出设备

} else if (usecase->type == PCM\_CAPTURE) {

usecase->devices = usecase->stream.in->device;

out\_snd\_device = SND\_DEVICE\_NONE;

in\_snd\_device=platform\_get\_input\_snd\_device(adev->platform, AUDIO\_DEVICE\_NONE);

...//如果当时是录音的话只选择输入设备

}

}

...

if (out\_snd\_device != SND\_DEVICE\_NONE) {

...

enable\_snd\_device(adev, out\_snd\_device, false);//打开输出设备通路

}

if (in\_snd\_device != SND\_DEVICE\_NONE) {

....

enable\_snd\_device(adev, in\_snd\_device, false);//打开输入设备通路

}

...

usecase->in\_snd\_device = in\_snd\_device;

usecase->out\_snd\_device = out\_snd\_device;//更新当前的输入和输出设备

enable\_audio\_route(adev, usecase, true);//使能audio route

...

}

2、enable\_snd\_device和 enable\_audio\_route函数

打开设备通路的流程：

int enable\_snd\_device(struct audio\_device \*adev,

                             snd\_device\_t snd\_device,

                             bool \_\_unused update\_mixer)

{

....

if(platform\_get\_snd\_device\_name\_extn(adev->platform, snd\_device, device\_name) < 0 ) {

        ALOGE("%s: Invalid sound device returned", \_\_func\_\_);

        return -EINVAL;

}//根据snd\_device名字在device\_table上查找device\_name，如[SND\_DEVICE\_OUT\_HANDSET] = "handset"等

...

audio\_route\_apply\_and\_update\_path(adev->audio\_route, device\_name);

//跟当前的device\_name在 mixer\_paths.xml上找到对应的设备通路并打开

...

}

Audio\_route.c (\system\media\audio\_route)

int audio\_route\_apply\_and\_update\_path(struct audio\_route \*ar, const char \*name)

{

if (audio\_route\_apply\_path(ar, name) < 0) {

//更新声卡中所有控件的值

return -1;

}

return audio\_route\_update\_path(ar, name, false /\*reverse\*/);

//将更新后声卡控件的值写入到“/dev/snd/controlC0”中

}

对于一条设备通路对应一个mixer\_path结构体

struct mixer\_path {

char \*name;

unsigned int size;

unsigned int length;//这条通路中控件的个数

struct mixer\_setting \*setting;//通路中各个控件的值

};

struct mixer\_setting {

unsigned int ctl\_index;//控件的编号

unsigned int num\_values;

int \*value;

};

struct audio\_route {

struct mixer \*mixer;

unsigned int num\_mixer\_ctls;

struct mixer\_state \*mixer\_state; // 这里面保存着整个声卡中每个控件的初始值

unsigned int mixer\_path\_size;

unsigned int num\_mixer\_paths;

struct mixer\_path \*mixer\_path;// 这里面保存mixer\_paths.xml的所有设备通路值

};

如：<path name="speaker">

        <ctl name="Spk HP Switch" value="Spk" />

        <ctl name="Ext Spk PA Mode" value="Mode\_2" />

        <ctl name="MI2S\_RX Channels" value="One" />

        <ctl name="RX1 MIX1 INP1" value="RX1" />

        <ctl name="RX1 MIX1 INP2" value="RX2" />

        <ctl name="HPHL" value="Switch" />

</path>

int audio\_route\_apply\_path(struct audio\_route \*ar, const char \*name)

{

struct mixer\_path \*path;

...

path = path\_get\_by\_name(ar, name);//根据当前设备名字找到该设备通路

...

path\_apply(ar, path);//根据当前使能通路控件值更新声卡中控件的状态，即mixer\_state

return 0;

}

static int path\_apply(struct audio\_route \*ar, struct mixer\_path \*path)

{

...

for (i = 0; i < path->length; i++) {

ctl\_index = path->setting[i].ctl\_index;//通路控件的id

ctl = index\_to\_ctl(ar, ctl\_index);

type = mixer\_ctl\_get\_type(ctl);

if (!is\_supported\_ctl\_type(type))

continue;

...

memcpy(ar->mixer\_state[ctl\_index].new\_value, path->setting[i].value,

path->setting[i].num\_values \* sizeof(int));//更新声卡中该id对应的空间的值

}

return 0;

}

static int audio\_route\_update\_path(struct audio\_route \*ar, const char \*name, bool reverse)

{

...

path = path\_get\_by\_name(ar, name);//找出当前设备通路

...

i = reverse ? (path->length - 1) : 0;

end = reverse ? -1 : (int32\_t)path->length;

while (i != end) {

...

ctl\_index = path->setting[i].ctl\_index;//找出设备通路中控件id

struct mixer\_state \* ms = &ar->mixer\_state[ctl\_index];//根据控件id找到当前控件的值

...

/\* if any value has changed, update the mixer \*/

for (j = 0; j < ms->num\_values; j++) {

if (ms->old\_value[j] != ms->new\_value[j]) {

if (type == MIXER\_CTL\_TYPE\_ENUM)

mixer\_ctl\_set\_value(ms->ctl, 0, ms->new\_value[0]);

//将值写入到底层控件中

else

mixer\_ctl\_set\_array(ms->ctl, ms->new\_value, ms->num\_values);

memcpy(ms->old\_value, ms->new\_value, ms->num\_values \* sizeof(int));

break;

}

}

i = reverse ? (i - 1) : (i + 1);

}

return 0;

}

int mixer\_ctl\_set\_value(struct mixer\_ctl \*ctl, unsigned int id, int value)

{

struct snd\_ctl\_elem\_value ev;

...

memset(&ev, 0, sizeof(ev));

ev.id.numid = ctl->info->id.numid;

ret = ioctl(ctl->mixer->fd, SNDRV\_CTL\_IOCTL\_ELEM\_READ, &ev);//读当前所有音频控件的值

...

return ioctl(ctl->mixer->fd, SNDRV\_CTL\_IOCTL\_ELEM\_WRITE, &ev);//将新的控件的值写入到/dev/snd/controlC0中去

使能audio route的流程：

int enable\_audio\_route(struct audio\_device \*adev,

struct audio\_usecase \*usecase,

bool \_\_unused update\_mixer)

{

...

platform\_add\_backend\_name(mixer\_path, snd\_device);//在通路名上添加设备名，用于定制不同的audio route

...

audio\_route\_apply\_and\_update\_path(adev->audio\_route, mixer\_path);//打开audio route，其流程与之前的打开设备通路的一样

...

}

以上设备通路和audio route打开的过程中，最终都走到了mixer\_ctl\_set\_value函数，这个函数里面利用底层提供给用户的空间的设备节点来控制底层的音频控件，其联动流程如下：首先，我们在使用 snd\_card\_create函数创建声卡的时候，会调用snd\_ctl\_create函数创建control逻辑设备，

int snd\_ctl\_create(struct snd\_card \*card)

{

static struct snd\_device\_ops ops = {

.dev\_free = snd\_ctl\_dev\_free,

.dev\_register = snd\_ctl\_dev\_register,

.dev\_disconnect = snd\_ctl\_dev\_disconnect,

};//这个ops很关键

if (snd\_BUG\_ON(!card))

return -ENXIO;

return snd\_device\_new(card, SNDRV\_DEV\_CONTROL, card, &ops);//把control设备放入到card->devices中去

static const struct file\_operations snd\_ctl\_f\_ops =

{

.owner =THIS\_MODULE,

.read =snd\_ctl\_read,

.open =snd\_ctl\_open,

.release =snd\_ctl\_release,

.llseek =no\_llseek,

.poll =snd\_ctl\_poll,

.unlocked\_ioctl =snd\_ctl\_ioctl,

.compat\_ioctl =snd\_ctl\_ioctl\_compat,

.fasync =snd\_ctl\_fasync,

};

在用户空间中 mixer\_ctl\_set\_value函数读写底层control节点，其实就是通过snd\_ctl\_ioctl函数来实现的：

static long snd\_ctl\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

...

case SNDRV\_CTL\_IOCTL\_ELEM\_READ:

return snd\_ctl\_elem\_read\_user(card, argp);

case SNDRV\_CTL\_IOCTL\_ELEM\_WRITE:

return snd\_ctl\_elem\_write\_user(ctl, argp);

...

}

最终，snd\_ctl\_elem\_read\_user和snd\_ctl\_elem\_write\_user函数都分别调用了snd\_kcontrol的get和put函数，实现了对dapm kcontrol控件的控制。

3、pcm\_open

Pcm\_open函数主要是在start\_output\_stream中会调用到。

Pcm中的参数，以deep buffer中的参数来说明：

struct pcm\_config pcm\_config\_deep\_buffer = {

    .channels = 2,

    .rate = DEFAULT\_OUTPUT\_SAMPLING\_RATE,//48000

    .period\_size = DEEP\_BUFFER\_OUTPUT\_PERIOD\_SIZE,//960，硬件一次消耗的数据

    .period\_count = DEEP\_BUFFER\_OUTPUT\_PERIOD\_COUNT,//8

    .format = PCM\_FORMAT\_S16\_LE,//16 bits = 2bytes(即2字节)

    .start\_threshold = DEEP\_BUFFER\_OUTPUT\_PERIOD\_SIZE / 4,

    .stop\_threshold = INT\_MAX,

    .avail\_min = DEEP\_BUFFER\_OUTPUT\_PERIOD\_SIZE / 4,

};

1 frame = channels \* format = 2\*2字节 = 4字节 （32bits）

Period\_size = DEEP\_BUFFER\_OUTPUT\_PERIOD\_SIZE 即 960frames = 960\*32字节

struct pcm \*pcm\_open(unsigned int card, unsigned int device,

                     unsigned int flags, struct pcm\_config \*config)

{

struct snd\_pcm\_hw\_params params;//硬件参数

struct snd\_pcm\_sw\_params sparams;//软件参数

...

pcm = calloc(1, sizeof(struct pcm));

pcm->config = \*config;

snprintf(fn, sizeof(fn), "/dev/snd/pcmC%uD%u%c", card, device,

             flags & PCM\_IN ? 'c' : 'p');//P即为底层对应的dai\_link

pcm->flags = flags;

pcm->fd = open(fn, O\_RDWR);//打开底层设备节点

---------------------

if (ioctl(pcm->fd, SNDRV\_PCM\_IOCTL\_HW\_PARAMS, ¶ms)) {

        oops(pcm, errno, "cannot set hw params");

        goto fail\_close;

}//将硬件参数参数写到设备节点中。

---------------------

  if (ioctl(pcm->fd, SNDRV\_PCM\_IOCTL\_SW\_PARAMS, &sparams)) {

        oops(pcm, errno, "cannot set sw params");

        goto fail;

  }//将软件参数写入到底层设备节点中

/dev/snd/pcmC%uD%u%c 节点底层相关的操作函数在：Pcm\_native.c (\linux\android\kernel\sound\core)：

const struct file\_operations snd\_pcm\_f\_ops[2] = {

{

.owner =THIS\_MODULE,

.write =snd\_pcm\_write,

.aio\_write =snd\_pcm\_aio\_write,

.open =snd\_pcm\_playback\_open,

.release =snd\_pcm\_release,

.llseek =no\_llseek,

.poll =snd\_pcm\_playback\_poll,

.unlocked\_ioctl =snd\_pcm\_playback\_ioctl,

.compat\_ioctl =snd\_pcm\_ioctl\_compat,

.mmap =snd\_pcm\_mmap,

.fasync =snd\_pcm\_fasync,

.get\_unmapped\_area =snd\_pcm\_get\_unmapped\_area,

},

4、pcm\_write函数

int pcm\_write(struct pcm \*pcm, const void \*data, unsigned int count)

{

struct snd\_xferi x;

...

x.buf = (void\*)data;

x.frames = count / (pcm->config.channels \*

                        pcm\_format\_to\_bits(pcm->config.format) / 8);

//计算出帧数

for (;;) {

   if (!pcm->running) {

      if (ioctl(pcm->fd, SNDRV\_PCM\_IOCTL\_PREPARE))

         return oops(pcm, errno, "cannot prepare channel");

         if (ioctl(pcm->fd, SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES, &x))

            return oops(pcm, errno, "cannot write initial data");

         pcm->running = 1;

         return 0;

   }

   if (ioctl(pcm->fd, SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES, &x)) {

         pcm->running = 0；

         return oops(pcm, errno, "cannot write stream data");

   }

   return 0;

}

}

底层相关操作函数：

snd\_pcm\_playback\_ioctl1() -> SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES -> snd\_pcm\_lib\_write() ->snd\_pcm\_lib\_write1() -> snd\_pcm\_lib\_write\_transfer()

具体流程如下：

在snd\_pcm\_playback\_ioctl1函数中，根据SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES参数，将用户空间传下来的数据传递给snd\_pcm\_lib\_write()函数：

if (copy\_from\_user(&xferi, \_xferi, sizeof(xferi)))

return -EFAULT;

result = snd\_pcm\_lib\_write(substream, xferi.buf, xferi.frames);

snd\_pcm\_lib\_write()函数由通过snd\_pcm\_lib\_write1()函数调用snd\_pcm\_lib\_write\_transfer（）将size大小的数据buf传送给DSP：

return snd\_pcm\_lib\_write1(substream, (unsigned long)buf, size, nonblock,snd\_pcm\_lib\_write\_transfer);

static snd\_pcm\_sframes\_t snd\_pcm\_lib\_write1(struct snd\_pcm\_substream \*substream,

   unsigned long data,

   snd\_pcm\_uframes\_t size,

   int nonblock,

   transfer\_f transfer)

{

...

while (size > 0) {

...

err = transfer(substream, appl\_ofs, data, offset, frames);

//这个transfer即为snd\_pcm\_lib\_write\_transfer，在其中会调用substream->ops->copy(substream, -1, hwoff, buf, frames) 拷贝数据

...

if (runtime->status->state == SNDRV\_PCM\_STATE\_PREPARED &&

snd\_pcm\_playback\_hw\_avail(runtime) >= (snd\_pcm\_sframes\_t)runtime->start\_threshold) {

err = snd\_pcm\_start(substream);

...

}

}

...

}

下面看一下snd\_pcm\_lib\_write\_transfer和snd\_pcm\_start两个函数。

static int snd\_pcm\_lib\_write\_transfer(struct snd\_pcm\_substream \*substream,

     unsigned int hwoff,

     unsigned long data, unsigned int off,

     snd\_pcm\_uframes\_t frames)

{

struct snd\_pcm\_runtime \*runtime = substream->runtime;

...

char \_\_user \*buf = (char \_\_user \*) data + frames\_to\_bytes(runtime, off);

if (substream->ops->copy) {

if ((err = substream->ops->copy(substream, -1, hwoff, buf, frames)) < 0)

//调用substeam的copy函数拷贝数据到dai上去

return err;

} else {

char \*hwbuf = runtime->dma\_area + frames\_to\_bytes(runtime, hwoff);

if (copy\_from\_user(hwbuf, buf, frames\_to\_bytes(runtime, frames)))

    //直接将数据拷贝到硬件去

return -EFAULT;

}

return 0;

}

int snd\_pcm\_start(struct snd\_pcm\_substream \*substream)

{

return snd\_pcm\_action(&snd\_pcm\_action\_start, substream,

     SNDRV\_PCM\_STATE\_RUNNING);

}

其中，

static struct action\_ops snd\_pcm\_action\_start = {

.pre\_action = snd\_pcm\_pre\_start,

.do\_action = snd\_pcm\_do\_start,

.undo\_action = snd\_pcm\_undo\_start,

.post\_action = snd\_pcm\_post\_start

};

在snd\_pcm\_start函数中有以下流程：

snd\_pcm\_start-> snd\_pcm\_action -> snd\_pcm\_action\_single

在snd\_pcm\_action\_single中会分别执行：

ops->pre\_action(substream, state);

ops->do\_action(substream, state);

即snd\_pcm\_pre\_start() 和snd\_pcm\_do\_start() 函数，在snd\_pcm\_do\_start函数中会执行：

substream->ops->trigger(substream, SNDRV\_PCM\_TRIGGER\_START); 即触发数据拷贝。

Substream的ops都是在platform中实现的，如在snd\_soc\_dai\_link中，我们设置了platform\_name  = "msm-pcm-dsp.0"

则可以找到其对应的ops函数在：

Msm-pcm-q6-v2.c (\kernel\sound\soc\msm\qdsp6v2)

static struct snd\_pcm\_ops msm\_pcm\_ops = {

.open           = msm\_pcm\_open,

.copy= msm\_pcm\_copy,

.hw\_params= msm\_pcm\_hw\_params,

.close          = msm\_pcm\_close,

.ioctl          = snd\_pcm\_lib\_ioctl,

.prepare        = msm\_pcm\_prepare,

.trigger        = msm\_pcm\_trigger,

.pointer        = msm\_pcm\_pointer,

.mmap= msm\_pcm\_mmap,

};

结合上面的流程，用户空间pcm\_write函数分别调用了底层的msm\_pcm\_copy和msm\_pcm\_trigger函数，在msm\_pcm\_copy函数中使用了q6asm\_write函数向DSP中写入数据，q6asm\_write和msm\_pcm\_trigger中最终都是通过apr\_send\_pkt来发送数据的。