

Audio 是整个 Android 平台非常重要的一个组成部分，负责音频数据的采集和输出、音频流的控制、音频设备的管理、音量调节等，主要包括如下部分：

* **Audio Application Framework**：音频应用框架
  + AudioTrack：负责回放数据的输出，属 Android 应用框架 API 类
  + AudioRecord：负责录音数据的采集，属 Android 应用框架 API 类
  + AudioSystem： 负责音频事务的综合管理，属 Android 应用框架 API 类
* **Audio Native Framework**：音频本地框架
  + AudioTrack：负责回放数据的输出，属 Android 本地框架 API 类
  + AudioRecord：负责录音数据的采集，属 Android 本地框架 API 类
  + AudioSystem： 负责音频事务的综合管理，属 Android 本地框架 API 类
* **Audio Services**：音频服务
  + AudioPolicyService：音频策略的制定者，负责音频设备切换的策略抉择、音量调节策略等
  + AudioFlinger：音频策略的执行者，负责输入输出流设备的管理及音频流数据的处理传输
* **Audio HAL**：音频硬件抽象层，负责与音频硬件设备的交互，由 AudioFlinger 直接调用

与 Audio 强相关的有 MultiMedia，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。

播放声音可以使用 MediaPlayer 和 AudioTrack。两者的差别在于：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。从上面 Android 音频系统架构图来看：MediaPlayer 在 Native 层会创建对应的音频解码器和一个 AudioTrack，解码后的数据交由 AudioTrack 输出。所以 MediaPlayer 的应用场景更广，一般情况下使用它也更方便；**只有一些对声音时延要求非常苛刻的应用场景才需要用到 AudioTrack**。

AudioTrack Java API 两种数据传输模式：

| **Transfer Mode** | **Description** |
| --- | --- |
| MODE\_STATIC | 应用进程将回放数据一次性付给 AudioTrack，适用于数据量小、时延要求高的场景 |
| MODE\_STREAM | 用进程需要持续调用 write() 写数据到 FIFO，写数据时有可能遭遇阻塞（等待 AudioFlinger::PlaybackThread 消费之前的数据），基本适用所有的音频场景 |

AudioTrack Java API 音频流类型：**（与 Android 的音频管理策略有关）**

| **Stream Type** | **Description** |
| --- | --- |
| STREAM\_VOICE\_CALL | 电话语音 |
| STREAM\_SYSTEM | 系统声音 |
| STREAM\_RING | 铃声声音，如来电铃声、闹钟铃声等 |
| STREAM\_MUSIC | 音乐声音 |
| STREAM\_ALARM | 警告音 |
| STREAM\_NOTIFICATION | 通知音 |
| STREAM\_DTMF | DTMF 音（拨号盘按键音） |

AudioTrack Native API 四种数据传输模式：

| **Transfer Mode** | **Description** |
| --- | --- |
| TRANSFER\_CALLBACK | 在 AudioTrackThread 线程中通过 audioCallback 回调函数主动从应用进程那里索取数据，ToneGenerator 采用这种模式 |
| TRANSFER\_OBTAIN | 应用进程需要调用 obtainBuffer()/releaseBuffer() 填充数据，目前我还没有见到实际的使用场景 |
| TRANSFER\_SYNC | 应用进程需要持续调用 write() 写数据到 FIFO，写数据时有可能遭遇阻塞（等待 AudioFlinger::PlaybackThread 消费之前的数据），基本适用所有的音频场景；对应于 AudioTrack Java API 的 MODE\_STREAM 模式 |
| TRANSFER\_SHARED | 应用进程将回放数据一次性付给 AudioTrack，适用于数据量小、时延要求高的场景；对应于 AudioTrack Java API 的 MODE\_STATIC 模式 |

AudioTrack Native API 音频流类型：

| **Stream Type** | **Description** |
| --- | --- |
| AUDIO\_STREAM\_VOICE\_CALL | 电话语音 |
| AUDIO\_STREAM\_SYSTEM | 系统声音 |
| AUDIO\_STREAM\_RING | 铃声声音，如来电铃声、闹钟铃声等 |
| AUDIO\_STREAM\_MUSIC | 音乐声音 |
| AUDIO\_STREAM\_ALARM | 警告音 |
| AUDIO\_STREAM\_NOTIFICATION | 通知音 |
| AUDIO\_STREAM\_DTMF | DTMF 音（拨号盘按键音） |

AudioTrack Native API 输出标识：

| **AUDIO\_OUTPUT\_FLAG** | **Description** |
| --- | --- |
| AUDIO\_OUTPUT\_FLAG\_DIRECT | 表示音频流直接输出到音频设备，不需要软件混音，一般用于 HDMI 设备声音输出 |
| AUDIO\_OUTPUT\_FLAG\_PRIMARY | 表示音频流需要输出到主输出设备，一般用于铃声类声音 |
| AUDIO\_OUTPUT\_FLAG\_FAST | 表示音频流需要快速输出到音频设备，一般用于按键音、游戏背景音等对时延要求高的场景 |
| AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER | 表示音频流输出可以接受较大的时延，一般用于音乐、视频播放等对时延要求不高的场景 |
| AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD | 表示音频流没有经过软件解码，需要输出到硬件解码器，由硬件解码器进行解码 |

帧：

一个完整的声音单元（一个采样样本）

双声道，2个样本

5.1 声道，6个样本

帧的大小（一个完整的声音单元的数据量）等于声道数乘以采样深度。

FrameSize = channelCount \* bytesPerSample

传输延迟：

表示一个周期的音频数据的传输时间

周期：

Liunx ALSA把数据缓冲区划分若干个块，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，周期大小即是一个数据块的帧数，传输延迟等于周期大小除以采样率，即 latency = periodSize /sampleRate

音频重采样：（为了保证不失真）

把一个采样率的数据转换到另一个采样率的数据。Android原生系统上，音频硬件设备一般都工作在一个固定的采样率上（如48kHZ），因此所有音轨数据都需要重采样到这个固定的采样率上，然后再输出。

AudioPolicyService：

音频系统策略的制定者，负责音频设备切换的策略选择，音量调节策略等。

AudioFlinger:

音频系统策略的执行者，负责音频流设备的管理及音频流数据的处理传输。

AudioFlinger：

AudioResampler.cpp：重采样处理类，可进行采样率转换和声道转换；由录制线程 AudioFlinger::RecordThread 直接使用

AudioMixer.cpp：混音处理类，包括重采样、音量调节、声道转换等，其中的重采样复用了 AudioResampler；由回放线程AudioFlinger::MixerThread 直接使用

Effects.cpp：音效处理类

Tracks.cpp：音频流管理类，可控制音频流的状态，如 start、stop、pause

Threads.cpp：回放线程和录制线程类；回放线程从 FIFO 读取回放数据并混音处理，然后写数据到输出流设备；录制线程从输入流设备读取录音数据并重采样处理，然后写数据到 FIFO

AudioFlinger.cpp：AudioFlinger 对外提供的服务接口

AudioFlinger 对外提供的主要的服务接口如下：

| **Interface** | **Description** |
| --- | --- |
| sampleRate | 获取硬件设备的采样率 |
| format | 获取硬件设备的音频格式 |
| frameCount | 获取硬件设备的周期帧数 |
| latency | 获取硬件设备的传输延迟 |
| setMasterVolume | 调节主输出设备的音量 |
| setMasterMute | 静音主输出设备 |
| setStreamVolume | 调节指定类型的音频流的音量，这种调节不影响其他类型的音频流的音量 |
| setStreamMute | 静音指定类型的音频流 |
| setVoiceVolume | 调节通话音量 |
| setMicMute | 静音麦克风输入 |
| setMode | 切换音频模式：音频模式有 4 种，分别是 Normal、Ringtone、Call、Communicatoin |
| setParameters | 设置音频参数：往下调用 HAL 层相应接口，常用于切换音频通道 |
| getParameters | 获取音频参数：往下调用 HAL 层相应接口 |
| openOutput | 打开输出流：打开输出流设备，并创建 PlaybackThread 对象 |
| closeOutput | 关闭输出流：移除并销毁 PlaybackThread 上面挂着的所有的 Track，退出 PlaybackThread，关闭输出流设备 |
| openInput | 打开输入流：打开输入流设备，并创建 RecordThread 对象 |
| closeInput | 关闭输入流：退出 RecordThread，关闭输入流设备 |
| createTrack | 新建输出流管理对象： 找到对应的 PlaybackThread，创建输出流管理对象 Track，然后创建并返回该 Track 的代理对象 TrackHandle |
| openRecord | 新建输入流管理对象：找到 RecordThread，创建输入流管理对象 RecordTrack，然后创建并返回该 RecordTrack 的代理对象 RecordHandle |

可以归纳出 AudioFlinger 响应的服务请求主要有：

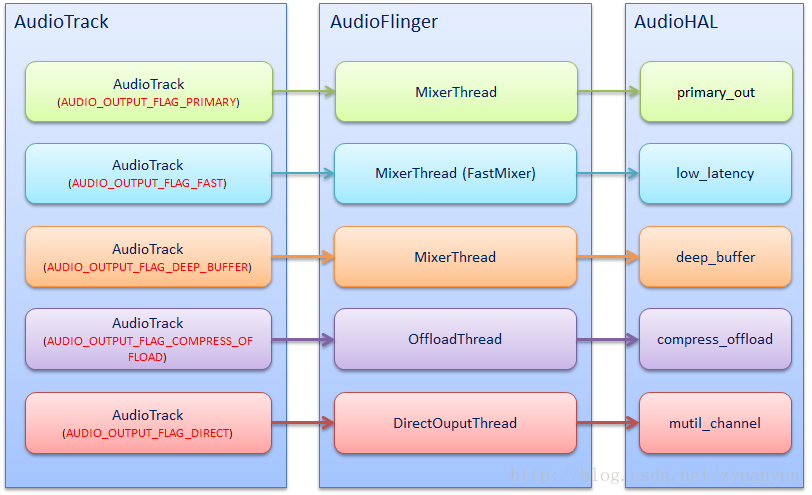
* 获取硬件设备的配置信息
* 音量调节
* 静音操作
* 音频模式切换
* 音频参数设置
* 输入输出流设备管理
* 音频流管理

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

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。

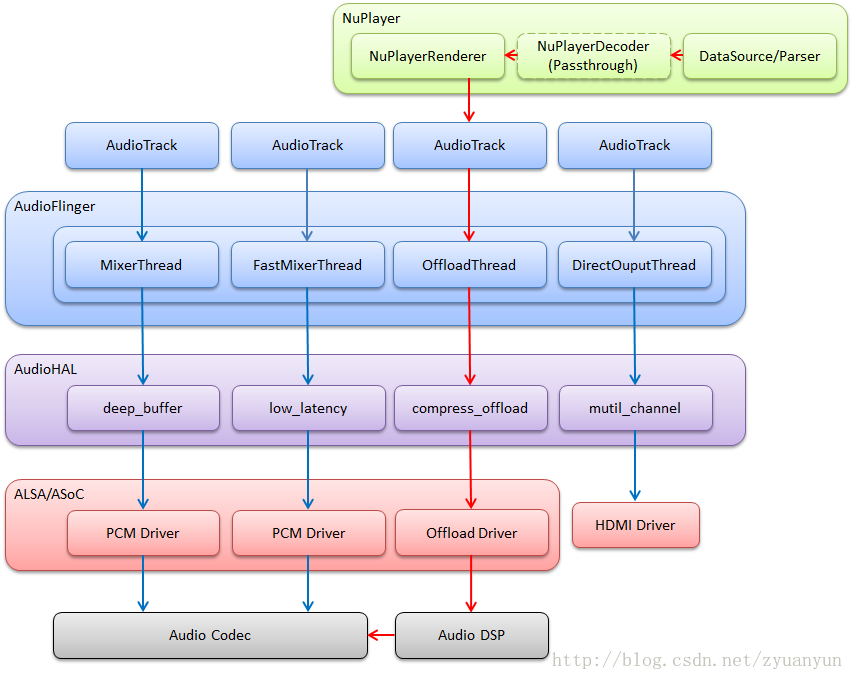
描述 AudioTrack、PlaybackThread、输出流设备三者的对应关系：



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。

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。

AudioFlinger OffloadThread



红色的是 Offload 音频流，它与其他音频流有什么本质的不同？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。

compress\_offload 设备，说白了，就是驱动 DSP 解码数据、Codec 输出声音。 而 DSP 要解码数据，首先得知道数据的编码信息，如编码器 codec\_id、采样率 sample\_rate、声道数 channel\_mask、比特率 bit\_rate 等信息，由于 DSP 并没有实现 DataSource/Parser 部件，不能自己解析数据的编码信息，所以得有“人”告诉它，这个“人”无疑是 compress\_offload 设备。

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。

到这里，大家明白了吗？每个 mp3/flac 文件的编码信息可能是不一样的，比如 a.mp3 文件的编码信息是 mp3&44.1KHZ&16bit… ，而 b.flac 文件的编码信息是 flac&48KHz&24bit…； 播放 a.mp3 时，AudioFlinger 打开一个配置为 mp3&44.1KHz&16bit… 的 compress\_offload 设备，接着播放 b.flac，就需要关闭之前的 compress\_offload 设备，重新打开一个配置为 flac&48KHz&24bit… 的 compress\_offload 设备。所以系统不会提前打开 compress\_offload 设备，只有等到播放 mp3、flac 时取到明确的数据编码信息，才基于这些编码信息打开 compress\_offload 设备。

编码信息包含很多条目，切换音源时，是否编码信息有一点点不一样，都需要重新打开 compress\_offload 设备呢？不能运行时更新信息到 DSP 吗？其实 stagefright 和 compress\_offload 是支持运行期更新某些信息的，也就是无缝切换，至于是哪些信息，依赖于 DSP 算法实现；有兴趣深入的可以参考 sendMetaDataToHal() 和 compress\_set\_gapless\_metadata() 。

**AudioTrack & AudioFlinger 相关类**

**AudioFlinger::PlaybackThread**：回放线程基类，不同输出标识的音频流对应不同类型的 PlaybackThread 实例。所有的 PlaybackThread 实例都会添加到 AudioFlinger.mPlaybackThreads 向量中；这个向量的定义： DefaultKeyedVector< audio\_io\_handle\_t, sp<PlaybackThread> > mPlaybackThreads;，audio\_io\_handle\_t 是与 PlaybackThread 是一一对应的，audio\_io\_handle\_t 在创建 PlaybackThread 时由系统分配，这个值是全局唯一的。

**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。个 MixerThread 往往挂着多个 Track 对象，这多个 Track 对象都会添加到 PlaybackThread.mTracks 向量中统一管理。

**AudioFlinger::TrackHandle**：Track 对象只负责音频流管理业务，代理 Track 的跨进程通讯，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。

**AudioTrack 构造过程**

构造一个 AudioTrack 实例时，系统大致流程：

1. 如果 cbf（audioCallback 回调函数）非空，那么创建 AudioTrackThread 线程处理 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）等参数调用 AudioSystem::getOutputForAttr()；经过一系列的调用，进入 AudioPolicyManager::getOutputForDevice()：
   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. 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；别忘了在 3.4. AudioFlinger 回放录制线程 小节中提到：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()）：
   1. 根据传入的 audio\_io\_handle\_t 找到它对应的 PlaybackThread；
   2. PlaybackThread 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. 最后新建一个 Track 的通讯代理 TrackHandle，并以 IAudioTrack 作为返回值给 AudioTrack（TrackHandle、BnAudioTrack、BpAudioTrack、IAudioTrack 的关系见上一个小节）；
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）；