Android 深入浅出之 Audio

第一部分 AudioTrack 分析

一 目的

本文的目的是通过从 Audio 系统来分析 Android 的代码，包括 Android 自定义的那套机制和一些常见类的使用，比如 Thread，MemoryBase 等。

分析的流程是：

- 先从 API 层对应的某个类开始，用户层先要有一个简单的使用流程。
- 根据这个流程，一步步进入到 JNI，服务层。在此过程中，碰到不熟悉或者第一次见到的类或者方法，都会解释。也就是深度优先的方法。

1.1 分析工具

分析工具很简单，就是 sourceinsight 和 android 的 API doc 文档。当然还得有 android 的源代码。我这里是基于 froyo 的源码。

注意，froyo 源码太多了，不要一股脑的加入到 sourceinsight 中，只要把 framwork 目录下的源码加进去就可以了，后续如要用的话，再加别的目录。

二 Audio 系统

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：

- AudioManager：这个主要是用来管理 Audio 系统的
- AudioTrack：这个主要是用来播放声音的
- 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，主要是策略上的问题。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。

三 AudioTrack（JAVA 层）

JAVA 的 AudioTrack 类的代码在：

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

3.1 AudioTrack API 的使用例子

先看看使用例子，然后跟进去分析。至于 AudioTrack 的其他使用方法和说明，需要大家自己去看 API 文档了。

```
//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 8K 个点
    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。
```

这里需要解释下两个东西：

1 AudioTrack.MODE_STREAM 的意思:

AudioTrack 中有 MODE_STATIC 和 MODE_STREAM 两种分类。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。这个和我们在 socket 中发送数据一样，应用层从某个地方获取数据，例如通过编解码得到 PCM 数据，然后 write 到 audiotrack。

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。

而 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 StreamType

这个在构造 AudioTrack 的第一个参数中使用。这个参数和 Android 中的 AudioManager 有关系，涉及到手机上的音频管理策略。

Android 将系统的声音分为以下几类常见的（未写全）：

- STREAM_ALARM：警告声
- STREAM_MUSCI：音乐声，例如 music 等
- STREAM_RING：铃声
- STREAM_SYSTEM：系统声音
- STREAM_VOCIE_CALL：电话声音

为什么要分这么多呢？以前在台式机上开发的时候很少知道有这么多的声音类型，不过仔细思考下，发现这样做是有道理的。例如你在听 music 的时候接到电话，这个时候 music 播放肯定会停止，此时你只能听到电话，如果你调节音量的话，这个调节肯定只对电话起作用。当电话打完了，再回到 music，你肯定不用再调节音量了。

其实系统将这几种声音的数据分开管理，所以，这个参数对 AudioTrack 来说，它的含义就是告诉系统，我现在想使用的是哪种类型的声音，这样系统就可以对应管理他们了。

3.2 分析之 getMinBufferSize

AudioTrack 的例子就几个函数。先看看第一个函数：

```
AudioTrack.getMinBufferSize(8000,//每秒 8K 个点
    AudioFormat.CHANNEL_CONFIGURATION_STEREO,//双声道
AudioFormat.ENCODING_PCM_16BIT);
----->AudioTrack.JAVA
//注意，这是个 static 函数
static public int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat)
{
        int channelCount = 0;
        switch(channelConfig) {
        case AudioFormat.CHANNEL_OUT_MONO:
        case AudioFormat.CHANNEL_CONFIGURATION_MONO:
            channelCount = 1;
            break;
        case AudioFormat.CHANNEL_OUT_STEREO:
        case AudioFormat.CHANNEL_CONFIGURATION_STEREO:
            channelCount = 2;--->看到了吧，外面名字搞得这么酷，其实就是指声道数
            break;
        default:
            loge("getMinBufferSize(): Invalid channel configuration.");
            return AudioTrack.ERROR_BAD_VALUE;
        }
    //目前只支持 PCM8 和 PCM16 精度的音频
        if ((audioFormat != AudioFormat.ENCODING_PCM_16BIT)
            && (audioFormat != AudioFormat.ENCODING_PCM_8BIT)) {
            loge("getMinBufferSize(): Invalid audio format.");
            return AudioTrack.ERROR_BAD_VALUE;
        }
      //ft，对采样频率也有要求，太低或太高都不行，人耳分辨率在 20HZ 到 40KHZ 之间
        if ( (sampleRateInHz < 4000) || (sampleRateInHz > 48000) ) {
            loge("getMinBufferSize(): " + sampleRateInHz +"Hz is not a supported sample
rate.");
            return AudioTrack.ERROR_BAD_VALUE;
        }
       //调用 native 函数，够烦的，什么事情都搞到 JNI 层去。
        int size = native_get_min_buff_size(sampleRateInHz, channelCount, audioFormat);
        if ((size == -1) || (size == 0)) {
            loge("getMinBufferSize(): error querying hardware");
```

```
            return AudioTrack.ERROR;
        }
        else {
            return size;
        }
```

native_get_min_buff_size--->在 framework/base/core/jni/android_media_track.cpp 中实现。（不了解 JNI 的一定要学习下，否则只能在 JAVA 层搞，太狭隘了。）最终对应到函数

```
static jint android_media_AudioTrack_get_min_buff_size(JNIEnv *env,  jobject thiz,
    jint sampleRateInHertz, jint nbChannels, jint audioFormat)
    {//注意我们传入的参数是:
    //sampleRateInHertz = 8000
    //nbChannels = 2;
    //audioFormat = AudioFormat.ENCODING_PCM_16BIT
    int afSamplingRate;
    int afFrameCount;
    uint32_t afLatency;
    //下面涉及到 AudioSystem，这里先不解释了，
    //反正知道从 AudioSystem 那查询了一些信息
    if (AudioSystem::getOutputSamplingRate(&afSamplingRate) != NO_ERROR) {
        return -1;
    }
    if (AudioSystem::getOutputFrameCount(&afFrameCount) != NO_ERROR) {
        return -1;
    }

    if (AudioSystem::getOutputLatency(&afLatency) != NO_ERROR) {
        return -1;
    }
    //音频中最常见的是 frame 这个单位，什么意思？经过多方查找，最后还是在 ALSA 的 wiki 中
    //找到解释了。一个 frame 就是 1 个采样点的字节数*声道。为啥搞个 frame 出来？因为对于多
//声道的话，用 1 个采样点的字节数表示不全，因为播放的时候肯定是多个声道的数据都要播出来
//才行。所以为了方便，就说 1 秒钟有多少个 frame，这样就能抛开声道数，把意思表示全了。
    // Ensure that buffer depth covers at least audio hardware latency
    uint32_t minBufCount = afLatency / ((1000 * afFrameCount)/afSamplingRate);
    if (minBufCount < 2) minBufCount = 2;
    uint32_t minFrameCount =
     (afFrameCount*sampleRateInHertz*minBufCount)/afSamplingRate;
```

```
//下面根据最小的 framecount 计算最小的 buffersize
int minBuffSize = minFrameCount
            * (audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1)
            * nbChannels;
    return minBuffSize;
}
```

getMinBufSize 函数完了后，我们得到一个满足最小要求的缓冲区大小。这样用户分配缓冲区就有了依据。下面就需要创建 AudioTrack 对象了

3.3 分析之 new AudioTrack

先看看调用函数：

```
    AudioTrack trackplayer = new AudioTrack(
    AudioManager.STREAM_MUSIC,
    8000,
    AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
    AudioFormat.ENCODING_PCM_16BIT,
    bufsize,
    AudioTrack.MODE_STREAM);//
其实现代码在 AudioTrack.java 中。
public AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int audioFormat,
            int bufferSizeInBytes, int mode)
    throws IllegalArgumentException {
        mState = STATE_UNINITIALIZED;

        // 获得主线程的 Looper，这个在 MediaScanner 分析中已经讲过了
        if ((mInitializationLooper = Looper.myLooper()) == null) {
            mInitializationLooper = Looper.getMainLooper();
        }

     //检查参数是否合法之类的，可以不管它
        audioParamCheck(streamType, sampleRateInHz, channelConfig, audioFormat, mode);
   //我是用 getMinBufsize 得到的大小，总不会出错吧?
        audioBuffSizeCheck(bufferSizeInBytes);

        // 调用 native 层的 native_setup，把自己的 WeakReference 传进去了
     //不了解 JAVA WeakReference 的可以上网自己查一下，很简单的
        int initResult = native_setup(new WeakReference<AudioTrack>(this),
```

```
                mStreamType, 这个值是 AudioManager.STREAM_MUSIC
                mSampleRate, 这个值是 8000
               mChannels, 这个值是 2
               mAudioFormat,这个值是 AudioFormat.ENCODING_PCM_16BIT
                mNativeBufferSizeInBytes, //这个是刚才 getMinBufSize 得到的
                mDataLoadMode);DataLoadMode 是 MODE_STREAM
        ....
    }
```

上面函数调用最终进入了 JNI 层 android_media_AudioTrack.cpp 下面的函数

```
static int
android_media_AudioTrack_native_setup(JNIEnv *env, jobject thiz, jobject weak_this,
        jint streamType, jint sampleRateInHertz, jint channels,
        jint audioFormat, jint buffSizeInBytes, jint memoryMode)
{
    int afSampleRate;
    int afFrameCount;
   下面又要调用一堆东西，烦不烦呐？具体干什么用的，以后分析到 AudioSystem 再说。
    AudioSystem::getOutputFrameCount(&afFrameCount, streamType);
   AudioSystem::getOutputSamplingRate(&afSampleRate, streamType);

   AudioSystem::isOutputChannel(channels);
    popCount 是统计一个整数中有多少位为 1 的算法
  int nbChannels = AudioSystem::popCount(channels);

    if (streamType == javaAudioTrackFields.STREAM_MUSIC) {
        atStreamType = AudioSystem::MUSIC;
    }
   int bytesPerSample = audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1;
    int format = audioFormat == javaAudioTrackFields.PCM16 ?
            AudioSystem::PCM_16_BIT : AudioSystem::PCM_8_BIT;
    int frameCount = buffSizeInBytes / (nbChannels * bytesPerSample);
    //上面是根据 Buffer 大小和一个 Frame 大小来计算帧数的。
    // AudioTrackJniStorage，就是一个保存一些数据的地方，这
    //里边有一些有用的知识，下面再详细解释
    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
       lpTrack->set(
            atStreamType,// stream type
            sampleRateInHertz,
            format,// word length, PCM
            channels,
            frameCount,
            0,// flags
            audioCallback,
          &(lpJniStorage->mCallbackData),//callback, callback data (user)
            0,// notificationFrames == 0 since not using EVENT_MORE_DATA to feed the
AudioTrack
            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
            true);// thread can call Java

    } else if (memoryMode == javaAudioTrackFields.MODE_STATIC) {
         //如果是 static 模式，需要用户一次性把数据写进去，然后
       //再由 audioTrack 自己去把数据读出来，所以需要一个共享内存
    //这里的共享内存是指 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);
          lpTrack->set(
            atStreamType,// stream type
            sampleRateInHertz,
            format,// word length, PCM
            channels,
            frameCount,
            0,// flags
            audioCallback,
```

```
          &(lpJniStorage->mCallbackData),//callback, callback data (user));
            0,// notificationFrames == 0 since not using EVENT_MORE_DATA to feed the
AudioTrack
            lpJniStorage->mMemBase,// shared mem
            true);// thread can call Java
    }

    if (lpTrack->initCheck() != NO_ERROR) {
        LOGE("Error initializing AudioTrack");
        goto native_init_failure;
    }
  //又来这一招，把 C++AudioTrack 对象指针保存到 JAVA 对象的一个变量中
//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，尤其是 Android 封装的一套共享内存的机制。这里一并讲解，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。

AudioTrackJniStorage 的代码很简单。

```
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, "AudioTrack Heap Base");
        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 也是 Android 搞的一套基于 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
对 Binder 机制不了解的，可以参考:
http://blog.csdn.net/Innost/archive/2011/01/08/6124685.aspx
  有好几个构造函数，我们看看我们使用的:
MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags, char const * name)
    : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
      mDevice(0), mNeedUnmap(false)
{
    const size_t pagesize = getpagesize();
    size = ((size + pagesize-1) & ~(pagesize-1));
    //创建共享内存，ashmem_create_region 这个是系统提供的,可以不管它
    //设备上打开的是/dev/ashmem 设备，而 Host 上打开的是一个 tmp 文件
    int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
    mapfd(fd, size);//把刚才那个 fd 通过 mmap 方式得到一块内存
    //不明白得去 man mmap 看看
    mapfd 完了后，mBase 变量指向内存的起始位置，mSize 是分配的内存大小，mFd 是
    ashmem_create_region 返回的文件描述符

}
MemoryHeapBase 提供了一下几个函数，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。

```
唉，这个估计是一个在 MemoryHeapBase 上的方便类吧？因为我看见了 offset
那么估计这个类就是一个能返回当前 Buffer 中写位置（就是 offset）的方便类
这样就不用用户到处去计算读写位置了。
class MemoryBase : public BnMemory
{
```

```
public:
    MemoryBase(const sp<IMemoryHeap>& heap, ssize_t offset, size_t size);
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset, size_t* size) const;
protected:
    size_t getSize() const { return mSize; }
    ssize_t getOffset() const { return mOffset; }
    const sp<IMemoryHeap>& getHeap() const { return mHeap; }
};
```

好了，明白上面两个 MemoryXXX，我们可以猜测下大概的使用方法了。

- BnXXX 端先分配 BnMemoryHeapBase 和 BnMemoryBase，
- 然后把 BnMemoryBase 传递到 BpXXX
- 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。

3.4 分析之 play 和 write

JAVA 层到这一步后就是调用 play 和 write 了。JAVA 层这两个函数没什么内容，都是直接转到 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
//从 int 类型直接转换成指针。要是以后 ARM 变成 64 位平台了，看 google 怎么改!
    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(JNIEnv *env,  jobject thiz,
                                            jshortArray javaAudioData,
                                            jint offsetInShorts,
jint sizeInShorts,
                                            jint javaAudioFormat) {
```

```
    return (android_media_AudioTrack_native_write(env, thiz,
                                                  (jbyteArray) javaAudioData,
                                                  offsetInShorts*2, sizeInShorts*2,
                                                  javaAudioFormat)
            / 2);
}
烦人，又根据 Byte 还是 Short 封装了下，最终会调到重要函数 writeToTrack 去
jint writeToTrack(AudioTrack* pTrack, jint audioFormat, jbyte* data,
                  jint offsetInBytes, jint sizeInBytes) {
      ssize_t written = 0;
    // regular write() or copy the data to the AudioTrack's shared memory?
    if (pTrack->sharedBuffer() == 0) {
    //创建的是流的方式，所以没有共享内存在 track 中
    //还记得我们在 native_setup 中调用的 set 吗？流模式下 AudioTrackJniStorage 可没创建
    //共享内存
        written = pTrack->write(data + offsetInBytes, sizeInBytes);
    } else {
        if (audioFormat == javaAudioTrackFields.PCM16) {
            // writing to shared memory, check for capacity
            if ((size_t)sizeInBytes > pTrack->sharedBuffer()->size()) {
                sizeInBytes = pTrack->sharedBuffer()->size();
            }
           //看见没？STATIC 模式的，就直接把数据拷贝到共享内存里
          //当然，这个共享内存是 pTrack 的，是我们在 set 时候把 AudioTrackJniStorage 的
        //共享设进去的
            memcpy(pTrack->sharedBuffer()->pointer(),
             data + offsetInBytes, sizeInBytes);
            written = sizeInBytes;
        } else if (audioFormat == javaAudioTrackFields.PCM8) {
           PCM8 格式的要先转换成 PCM16


    }
    return written;
}
```

到这里，似乎很简单啊，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。反正 JNI 这层是再看不出什么有意思的东西了。

四 AudioTrack（C++层）

接上面的内容，我们知道在 JNI 层，有以下几个步骤：

- new 了一个 AudioTrack
- 调用 set 函数，把 AudioTrackJniStorage 等信息传进去
- 调用了 AudioTrack 的 start 函数
- 调用 AudioTrack 的 write 函数

那么，我们就看看真正干活的的 C++AudioTrack 吧。

AudioTrack.cpp 位于 framework\base\libmedia\AudioTrack.cpp

4.1 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
那我们看看 set 函数把。
status_t AudioTrack::set(
        int streamType,
        uint32_t sampleRate,
        int format,
        int channels,
        int frameCount,
        uint32_t flags,
        callback_t cbf,
```

```
        void* user,
        int notificationFrames,
        const sp<IMemory>& sharedBuffer,
        bool threadCanCallJava)
{
   ...前面一堆的判断，等以后讲 AudioSystem 再说
   audio_io_handle_t output =
    AudioSystem::getOutput((AudioSystem::stream_type)streamType,
            sampleRate, format, channels, (AudioSystem::output_flags)flags);
   //createTrack? 看来这是真正干活的
    status_t status = createTrack(streamType, sampleRate, format, channelCount,
                                  frameCount, flags, sharedBuffer, 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);
       }
   return NO_ERROR;
}
看看真正干活的 createTrack
status_t AudioTrack::createTrack(
        int streamType,
        uint32_t sampleRate,
        int format,
        int channelCount,
        int frameCount,
        uint32_t flags,
        const sp<IMemory>& sharedBuffer,
        audio_io_handle_t output)
{
    status_t status;
    //啊，看来和 audioFlinger 挂上关系了呀。
    const sp<IAudioFlinger>& audioFlinger = AudioSystem::get_audio_flinger();

  //下面这个调用最终会在 AudioFlinger 中出现。暂时不管它。
    sp<IAudioTrack> track = audioFlinger->createTrack(getpid(),
                                                      streamType,
```

```
                                                    sampleRate,
                                                    format,
                                                    channelCount,
                                                    frameCount,
                                                    ((uint16_t)flags) << 16,
                                                    sharedBuffer,
                                                    output,
                                                    &status);

    //看见没，从 track 也就是 AudioFlinger 那边得到一个 IMemory 接口
//这个看来就是最终 write 写入的地方
    sp<IMemory> cblk = track->getCblk();
    mAudioTrack.clear();
    mAudioTrack = track;
    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());
    mCblk->out = 1;

    mFrameCount = mCblk->frameCount;
    if (sharedBuffer == 0) {
    //终于看到 buffer 相关的了。注意我们这里的情况
    //STREAM 模式没有传入共享 buffer，但是数据确实又需要 buffer 承载。
    //反正 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);
    }
    return NO_ERROR;
}
```

还记得我们说 MemoryXXX 没有同步机制，所以这里应该有一个东西能体现同步的，那么我告诉大家，就在 audio_track_cblk_t 结构中。它的头文件在 framework/base/include/private/media/AudioTrackShared.h
实现文件就在 AudioTrack.cpp 中

```
audio_track_cblk_t::audio_track_cblk_t()
//看见下面的 SHARED 没？都是表示跨进程共享的意思。这个我就不跟进去说了
//等以后介绍同步方面的知识时，再细说
    : lock(Mutex::SHARED), cv(Condition::SHARED), user(0), server(0),
```

```
    userBase(0), serverBase(0), buffers(0), frameCount(0),
    loopStart(UINT_MAX), loopEnd(UINT_MAX), loopCount(0), volumeLR(0),
    flowControlFlag(1), forceReady(0)
{
}
```

到这里，大家应该都有个大概的全景了。

- AudioTrack 得到 AudioFlinger 中的一个 IAudioTrack 对象，这里边有一个很重要的数据结构 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，这个线程干嘛的呢？还不知道
- 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（其实 AudioFlinger 是一个服务，在 mediaservice 中运行）接收数据，并最终写到音频设备中。

那我们先看看 AudioTrackThread 干什么了。

```
调用的语句是:
mAudioTrackThread = new AudioTrackThread(*this, threadCanCallJava);
AudioTrackThread 从 Thread 中派生，这个内容在深入浅出 Binder 机制讲过了。
反正最终会调用 AudioTrackAThread 的 threadLoop 函数。
先看看构造函数
AudioTrack::AudioTrackThread::AudioTrackThread(AudioTrack& receiver, bool bCanCallJava)
    : Thread(bCanCallJava), mReceiver(receiver)
{  //mReceiver 就是 AudioTrack 对象
  // bCanCallJava 为 TRUE
}
```

这个线程的启动由 AudioTrack 的 start 函数触发。

```
void AudioTrack::start()
{
  //start 函数调用 AudioTrackThread 函数触发产生一个新的线程，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()
{
```

```
  //太恶心了，又调用 AudioTrack 的 processAudioBuffer 函数
  return mReceiver.processAudioBuffer(this);
}
bool AudioTrack::processAudioBuffer(const sp<AudioTrackThread>& thread)
{
Buffer audioBuffer;
    uint32_t frames;
    size_t writtenSize;
      ...回调 1
         mCbf(EVENT_UNDERRUN, mUserData, 0);
      ...回调 2 都是传递一些信息到 JNI 里边
         mCbf(EVENT_BUFFER_END, mUserData, 0);
         // Manage loop end callback
    while (mLoopCount > mCblk->loopCount) {
        mCbf(EVENT_LOOP_END, mUserData, (void *)&loopCount);
    }
  //下面好像有写数据的东西
      do {
       audioBuffer.frameCount = frames;
//获得 buffer,
       status_t err = obtainBuffer(&audioBuffer, 1);
        size_t reqSize = audioBuffer.size;
//把 buffer 回调到 JNI 那去，这是单独一个线程，而我们还有上层用户在那不停
//地 write 呢，怎么会这样?
        mCbf(EVENT_MORE_DATA, mUserData, &audioBuffer);
         audioBuffer.size = writtenSize;
         frames -= audioBuffer.frameCount;
       releaseBuffer(&audioBuffer); //释放 buffer，和 obtain 相对应，看来是 LOCK 和 UNLOCK
操作了
    }
    while (frames);
   return true;
}
```

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) {
         //哈哈，太好了，这个函数没往里边写数据
        AudioTrack::Buffer* pBuff = (AudioTrack::Buffer*)info;
        pBuff->size = 0;
     }
```

从代码上看，本来 google 考虑是异步的回调方式来写数据，可惜发现这种方式会比较复杂，尤其是对用户开放的 JAVA AudioTrack 会很不好处理，所以嘛，偷偷摸摸得给绕过去了。

太好了，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。

让我们看看 write 吧。

4.2 write

```
ssize_t AudioTrack::write(const void* buffer, size_t userSize)
{
  够简单，就是 obtainBuffer，memcpy 数据，然后 releasBuffer
眯着眼睛都能想到，obtainBuffer 一定是 Lock 住内存了，releaseBuffer 一定是 unlock 内存了
     do {
        audioBuffer.frameCount = userSize/frameSize();
        status_t err = obtainBuffer(&audioBuffer, -1);
         size_t toWrite;
         toWrite = audioBuffer.size;
         memcpy(audioBuffer.i8, src, toWrite);
         src += toWrite;
        }
        userSize -= toWrite;
        written += toWrite;
        releaseBuffer(&audioBuffer);
    } while (userSize);

    return written;
}
obtainBuffer 太复杂了，不过大家知道其大概工作方式就可以了
status_t AudioTrack::obtainBuffer(Buffer* audioBuffer, int32_t waitCount)
{
   //恕我中间省略太多，大部分都是和当前数据位置相关，
    uint32_t framesAvail = cblk->framesAvailable();
     cblk->lock.lock();//看见没，lock 了
     result = cblk->cv.waitRelative(cblk->lock, milliseconds(waitTimeMs));
```

```
//我发现很多地方都要判断远端的 AudioFlinger 的状态，比如是否退出了之类的，难道
//没有一个好的方法来集中处理这种事情吗?
      if (result == DEAD_OBJECT) {
        result = createTrack(mStreamType, cblk->sampleRate, mFormat, mChannelCount,
          mFrameCount, mFlags, mSharedBuffer,getOutput());
        }
    //得到 buffer
    audioBuffer->raw = (int8_t *)cblk->buffer(u);
  return active ? status_t(NO_ERROR) : status_t(STOPPED);
}
在看看 releaseBuffer
void AudioTrack::releaseBuffer(Buffer* audioBuffer)
{
    audio_track_cblk_t* cblk = mCblk;
    cblk->stepUser(audioBuffer->frameCount);
}
uint32_t audio_track_cblk_t::stepUser(uint32_t frameCount)
{
    uint32_t u = this->user;

    u += frameCount;
     if (out) {
          if (bufferTimeoutMs == MAX_STARTUP_TIMEOUT_MS-1) {
            bufferTimeoutMs = MAX_RUN_TIMEOUT_MS;
        }
    } else if (u > this->server) {
         u = this->server;
    }

    if (u >= userBase + this->frameCount) {
        userBase += this->frameCount;
    }
   this->user = u;
  flowControlFlag = 0;
  return u;
}
```

奇怪了，releaseBuffer 没有 unlock 操作啊？难道我失误了？

再去看看 obtainBuffer?为何写得这么晦涩难懂？

原来在 obtainBuffer 中会某一次进去 lock,再某一次进去可能就是 unlock 了。没看到 obtainBuffer 中到处有 lock,unlock,wait 等同步操作吗。一定是这个道理。难怪写这么复杂。还使用了少用的 goto 语句。

唉，有必要这样吗!

五 AudioTrack 总结

通过这一次的分析，我自己觉得有以下几个点：

- AudioTrack 的工作原理，尤其是数据的传递这一块，做了比较细致的分析，包括共享内存，跨进程的同步等，也能解释不少疑惑了。
- 看起来，最重要的工作是在 AudioFlinger 中做的。通过 AudioTrack 的介绍，我们给后续深入分析 AudioFlinger 提供了一个切入点

工作原理和流程嘛，再说一次好了，JAVA 层就看最前面那个例子吧，实在没什么说的。

- 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。
- 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
- 用户一次次得 write，那 AudioTrack 无非就是把数据 memcpy 到共享 buffer 中咯
- 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。