

Android 深入浅出之 AudioFlinger

一 目的

本文承接 Audio 第一部分的 AudioTrack，通过 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，我们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。WHY？因为 AT 是 AF 的客户端，而 AF 是 Android 系统中 Audio 管理的中枢。AT 我们分析的是按流程方法，那么以 AT 为切入点的话，AF 的分析也应该是流程分析了。

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 我自己感觉是不够的。因为我发现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。所以，针对 Android 声音策略上的分析，我会单独在以后来分析。

二 从 AT 切入到 AF

直接从头看代码是没法掌握 AF 的主干的，必须要有一个切入点，也就是用一个正常的调用流程来分析 AF 的处理流程。先看看 AF 的产生吧，这个 C/S 架构的服务者是如何产生的呢？

2.1 AudioFlinger 的诞生

AF 是一个服务，这个就不用我多说了吧？代码在 framework/base/media/mediaserver/Main_mediaServer.cpp 中。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());

    sp<IServiceManager> sm = defaultServiceManager();

    ....

    AudioFlinger::instantiate(); --->AF 的实例化

    AudioPolicyService::instantiate(); --->APS 的实例化

    ....

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

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

```

哇塞，看来这个程序的负担很重啊。没想到。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( //把 AF 实例加入系统服务  
        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();  
  
    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。我们暂且不管它。等分析到 Audio 策略再说。

好了，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我们可以播放了吗？

2.2 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。

--参加 AudioTrack 分析的 4.1 节

1. 创建

```
AudioTrack* lpTrack = new AudioTrack();

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,

                        sampleRate, format, channels, (AudioSystem::output_flags)flags);

status_t status = createTrack(streamType, sampleRate, format, channelCount,

                              frameCount, flags, sharedBuffer, output);
```

---->creatTrack 会和 AF 打交道。我们看看 createTrack 重要语句

```
const sp<IAudioFlinger>& 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> cb1k = track->getCb1k(); //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. start 和 write

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

```
void AudioTrack::start()

{

//果然啊...

    status_t status = mAudioTrack->start();

}
```

那 write 呢？我们之前讲了，AT 就是从共享 buffer 中：

- Lock 缓存
- 写缓存
- 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：

- Lock,

- 读缓存，写硬件
- Unlock

总之，我们知道了 AT 的调用 AF 的流程了。下面一个一个看。

2.3 AF 流程

1 createTrack

```
sp<IAudioTrack> AudioFlinger::createTrack(  
    pid_t pid, //AT 的 pid 号  
    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<IMemory>& sharedBuffer, //AT 传入的共享 buffer, 这里为空  
    int output, //这个是从 AuidoSystem 获得的对应 MUSIC 流类型的索引  
    status_t *status)  
{  
    sp<PlaybackThread::Track> track;  
    sp<TrackHandle> trackHandle;  
    sp<Client> client;  
    wp<Client> wclient;  
    status_t lStatus;  
  
    {  
        Mutex::AutoLock _l(mLock);  
        //根据 output 句柄, 获得线程?  
        PlaybackThread *thread = checkPlaybackThread_l(output);  
        //看看这个进程是不是已经是 AF 的客户了  
        //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);  
  
    }  
  
    //喔，还有一个 trackHandle，而且返回到 AF 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。说实话，我刚开始接触的时候，大脑因为常接触到这些眼生的东西而死机！大家先不要拘泥于这些东西，我会一一分析到的。先进入到 `checkPlaybackThread_l` 看看。

```
AudioFlinger::PlaybackThread *AudioFlinger::checkPlaybackThread_l(int output) const  
{  
  
    PlaybackThread *thread = NULL;  
  
    //看到这种 indexOfKey 的东西，应该立即能想到：  
  
    //喔，这可能是一个 map 之类的东西，根据 key 能找到实际的 value  
  
    if (mPlaybackThreads.indexOfKey(output) >= 0) {  
  
        thread = (PlaybackThread *)mPlaybackThreads.valueFor(output).get();  
  
    }  
  
    //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：

- 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，我们刚才的调用流程里边，也没看到哪有创建线程的地方呀。
- 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。

我们知道，output 的来源是由 AT 的 set 函数得到的：如下：

```
audio_io_handle_t output = AudioSystem::getOutput(  
  
(AudioSystem::stream_type)streamType, //MUSIC 类型  
  
    sampleRate, //8000  
  
format, //PCM-16  
  
channels, //2 两个声道  
  
(AudioSystem::output_flags)flags//0  
);
```

上面这几个参数后续不再提示了，大家知道这些值都是由 AT 做为切入点传进去的

然后它在调用 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。

//叫 handle 啊? 好像 linux 下这种叫法的很少，难道又是受 MS 的影响吗?

我们进到 AudioSystem::getOutput 看看。注意，大家想想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,  
  
    uint32_t samplingRate,  
  
    uint32_t format,  
  
    uint32_t channels,  
  
    output_flags flags)  
{  
  
    audio_io_handle_t output = 0;  
  
    if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) == 0 &&  
        ((stream != AudioSystem::VOICE_CALL && stream != AudioSystem::BLUETOOTH_SCO) ||  
         channels != AudioSystem::CHANNEL_OUT_MONO ||  
         (samplingRate != 8000 && samplingRate != 16000))) {  
  
        Mutex::Autolock _l(gLock);  
  
        //根据我们的参数，我们会走到这个里边来  
  
        //喔，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) {  
  
        //我晕，又到 AudioPolicyService (APS)
```

```
//由它去 getOutput

    const sp<IAudioPolicyService>& 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;

}
```

怎么办？需要到 APS 中才能找到 output 的信息？

没办法，硬着头皮进去吧。那先得看看 APS 是如何创建的。不过这个刚才已经说了，是和 AF 一块在那个 Main_mediaService.cpp 中实例化的。

位置在 framework/base/lib/libaudioflinger/ AudioPolicyService.cpp 中

```
AudioPolicyService::AudioPolicyService()

: BnAudioPolicyService(), mpPolicyManager(NULL)

{

// 下面两个线程以后再说

mTonePlaybackThread = new AudioCommandThread(String8(""));

mAudioCommandThread = new AudioCommandThread(String8("ApmCommandThread"));

#if (defined GENERIC_AUDIO) || (defined AUDIO_POLICY_TEST)

//喔，使用普适的 AudioPolicyManager，把自己 this 做为参数

//我们这里先使用普适的看看吧

    mpPolicyManager = new AudioPolicyManagerBase(this);

//使用硬件厂商提供的特殊的 AudioPolicyManager

//mpPolicyManager = createAudioPolicyManager(this);

}

}
```

我们看看 AudioManagerBase 的构造函数吧，在 framework/base/lib/audioFlinger/ AudioPolicyManagerBase.cpp 中。

```

AudioPolicyManagerBase::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,

                                                    &outputDesc->mSamplingRate,

                                                    &outputDesc->mFormat,

                                                    &outputDesc->mChannels,

                                                    &outputDesc->mLatency,

                                                    outputDesc->mFlags);

    openOutput 又交给 APS 的 openOutput 来完成了，真绕...

}

```

唉，看来我们还是得回到 APS，

```

audio_io_handle_t AudioPolicyService::openOutput(uint32_t *pDevices,

                                                uint32_t *pSamplingRate,

                                                uint32_t *pFormat,

                                                uint32_t *pChannels,

                                                uint32_t *pLatencyMs,

                                                AudioSystem::output_flags flags)

{

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

    //FT, FT, FT, FT, FT, FT, FT

    //绕了这么一个大圈子，竟然回到 AudioFlinger 中了吗？

    return af->openOutput(pDevices, pSamplingRate, (uint32_t *)pFormat, pChannels,

                        pLatencyMs, flags);

}

```

在我们再次被绕晕之后，我们回眸看看足迹吧：

- 在 `AudioTrack` 中，调用 `set` 函数
- 这个函数会通过 `AudioSystem::getOutput` 来得到一个 `output` 的句柄

- AS 的 `getOutput` 会调用 `AudioPolicyService` 的 `getOutput`
- 然后我们就没继续讲 `APS` 的 `getOutput` 了，而是去看看 `APS` 创建的东西
- 发现 `APS` 创建的时候会创建一个 `AudioManagerBase`，这个 `AMB` 的创建又会调用 `APS` 的 `openOutput`。
- `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？

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,
                                                uint32_t samplingRate,
                                                uint32_t format,
                                                uint32_t channels,
                                                AudioSystem::output_flags flags)
{
    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 stream,
                                                    uint32_t samplingRate,
                                                    uint32_t format,
                                                    uint32_t channels,
                                                    AudioSystem::output_flags flags)
{
    audio_io_handle_t output = 0;
    uint32_t latency = 0;
    // 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。反正，到这里，我们知道了，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,
```

```
        uint32_t *pSamplingRate,
        uint32_t *pFormat,
        uint32_t *pChannels,
        uint32_t *pLatencyMs,
        uint32_t flags)
{
    status_t status;

    PlaybackThread *thread = NULL;

    mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;

    uint32_t samplingRate = pSamplingRate ? *pSamplingRate : 0;
    uint32_t format = pFormat ? *pFormat : 0;
    uint32_t channels = pChannels ? *pChannels : 0;
    uint32_t latency = pLatencyMs ? *pLatencyMs : 0;

    Mutex::Autolock _l(mLock);

    //由 Audio 硬件 HAL 对象创建一个 AudioStreamOut 对象
    AudioStreamOut *output = mAudioHardware->openOutputStream(*pDevices,
                                                                (int *)&format,
                                                                &channels,
                                                                &samplingRate,
                                                                &status);

    mHardwareStatus = AUDIO_HW_IDLE;

    if (output != 0) {
        //创建一个 Mixer 线程
        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;
}
}
```

明白了，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,  
    int streamType,  
    uint32_t sampleRate,  
    int format,  
    int channelCount,  
    int frameCount,  
    uint32_t flags,  
    const sp<IMemory>& sharedBuffer,  
    int output,  
    status_t *status)  
{  
    sp<PlaybackThread::Track> track;  
    sp<TrackHandle> trackHandle;  
    sp<Client> client;  
    wp<Client> wclient;  
    status_t lStatus;  
    {  
        //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。
```

实在是...太绕了。再进去看看 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。

跟着 sourceinsight ctrl+鼠标左键就进入到下面这个函数。

下面这个函数的签名好长啊。这是为何？

原来 Android 的 C++类中大量定义了内部类。说实话，我之前几年的 C++的经验中基本没接触过这么频繁使用内部类的东东。--->当然，你可以说 STL 也大量使用了呀。

我们就把 C++的内部类当做普通的类一样看待吧，其实我感觉也没什么特殊的含义，和外部类是一样的，包括函数调用，public/private 之类的东西。这个和 JAVA 的内部类是大不一样的。

```
sp<AudioFlinger::PlaybackThread::Track> AudioFlinger::PlaybackThread::createTrack_l(  
    const sp<AudioFlinger::Client>& client,  
    int streamType,  
    uint32_t sampleRate,  
    int format,  
    int channelCount,  
    int frameCount,  
    const sp<IMemory>& sharedBuffer,  
    status_t *status)  
{  
    sp<Track> track;  
    status_t lStatus;  
    { // scope for mLock  
        Mutex::Autolock _l(mLock);  
//new 一个 track 对象  
//我有点愤怒了，Android 真是层层封装啊，名字取得也非常相似。  
//看看这个参数吧，注意 sharedBuffer 这个，此时的值应是 0  
        track = new Track(this, client, streamType, sampleRate, format,  
            channelCount, frameCount, 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

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
再去看看 `new Track`，我们一直还没找到共享内存在哪里创建的!!!

```
AudioFlinger::PlaybackThread::Track::Track(  
    const wp<ThreadBase>& thread,  
    const sp<Client>& client,  
    int streamType,  
    uint32_t sampleRate,  
    int format,  
    int channelCount,  
    int frameCount,  
    const sp<IMemory>& sharedBuffer)  
    : TrackBase(thread, client, sampleRate, format, channelCount, frameCount, 0,  
    sharedBuffer),  
    mMute(false), mSharedBuffer(sharedBuffer), mName(-1)  
{  
    // mCblk !=NULL?什么时候创建的??  
    //只能看基类 TrackBase，还是很愤怒，太多继承了。  
    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

```
AudioFlinger::ThreadBase::TrackBase::TrackBase(  
    const wp<ThreadBase>& thread,  
    const sp<Client>& client,  
    uint32_t sampleRate,  
    int format,
```

```
        int channelCount,
        int frameCount,
        uint32_t flags,
        const sp<IMemory>& sharedBuffer)
:   RefBase(),
    mThread(thread),
    mClient(client),
    mCblk(0),
    mFrameCount(0),
    mState(IDLE),
    mClientTid(-1),
    mFormat(format),
    mFlags(flags & ~SYSTEM_FLAGS_MASK)
{
    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。这个 client 是什么？就是我们在 CreateTrack 中创建的
那个 Client，我不想再说了。反正这里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());
    下面这个语法好怪啊。什么意思？？
    new(mCblk) audio_track_cblk_t();
//各位，这就是 C++语法中的 placement new。干啥用的啊？new 后面的括号中是一块 buffer，再
后面是一个类的构造函数。对了，这个 placement new 的意思就是在这块 buffer 中构造一个对象。
我们之前的普通 new 是没法让一个对象在某块指定的内存中创建的。而 placement new 却可以。
这样不就达到我们的目的了吗？搞一块共享内存，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？太牛牛牛牛牛了。怎么想到的？
    // clear all buffers
```

```

mCblk->frameCount = frameCount;

mCblk->sampleRate = sampleRate;

mCblk->channels = (uint8_t)channelCount;
}

```

好了，解决一个重大疑惑，跨进程数据共享的重要数据结构 `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-1 的返回值。

```

2.4 到底有少种对象

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。说实话，这里废点心血整个或者 `paste` 一个大的 UML 图未尝不可。但是我不太习惯用图说话，因为图我实在是记不住。那好吧。我们就用最简单的话语争取把目前出现的对象说清楚。

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也行吧。

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`。这个 `TrackHandle` 实际上是对真正干活的 `PlaybackThread::Track` 的一个跨进程支持的封装。什么意思？本来 `PlaybackThread::Track` 是真正在 AF 中干活的东西，不过为了支持跨进程的话，我们用 `TrackHandle` 对其进行了一下包转。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<PlaybackThread::Track>& 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<IMemory> 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：

- RecordThread:

```
RecordThread : public ThreadBase, public AudioBufferProvider
```

用于录音的线程。

- PlaybackThread:

```
class PlaybackThread : public ThreadBase
```

用于播放的线程

- MixerThread

```
MixerThread : public PlaybackThread
```

用于混音的线程，注意他是从 PlaybackThread 派生下来的。

- DirectoutputThread

```
DirectOutputThread : public PlaybackThread
```

直接输出线程，我们之前在代码里老看到 DIRECT_OUTPUT 之类的判断，看来最终和这个线程有关。

- 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。-----》FT，真的很麻烦。

ThreadBase 我们不说了，反正里边封装了一些有用的函数。

我们看看 PlayingThread 吧，里边由定义了内部类：

5 PlayingThread 的内部类 Track

我们知道，TrackHandle 构造用的那个 Track 是 PlayingThread 的 createTrack_1 得到的。

```
class Track : public TrackBase
```

晕喔，又来一个 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 中用得到。

个人感觉：上面这些东西，其实完完全全可以独立到不同的文件中，然后加一些注释说明。写这样的代码，要是我是 BOSS 的话，一定会很不爽。有什么意义吗？有什么好处吗？

2.5 AF 流程继续

好了，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？我记得我们是根据 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,
                             uint32_t *pSamplingRate,
                             uint32_t *pFormat,
                             uint32_t *pChannels,
                             uint32_t *pLatencyMs,
                             uint32_t flags)
{
    if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) ||
```

```

        (format != AudioSystem::PCM_16_BIT) ||

        (channels != AudioSystem::CHANNEL_OUT_STEREO)) {

            thread = new DirectOutputThread(this, output, ++mNextThreadId);

            //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 立体声

            //则创建 DirectOutputThread.

        } else {

            //可惜啊, 我们创建的是最复杂的 MixerThread

            thread = new MixerThread(this, output, ++mNextThreadId);
    
```

1. MixerThread

非常重要的工作线程, 我们看看它的构造函数。

```

AudioFlinger::MixerThread::MixerThread(const sp<AudioFlinger>& audioFlinger,
AudioStreamOut* output, int id)

:   PlaybackThread(audioFlinger, output, id),

    mAudioMixer(0)

{

    mType = PlaybackThread::MIXER;

    //混音器对象, 传进去的两个参数时基类 ThreadBase 的, 都为 0

    //这个对象巨复杂, 最终混音的数据都由它生成, 以后再说...

    mAudioMixer = new AudioMixer(mFrameCount, mSampleRate);

}
    
```

2. AT 调用 start

此时, AT 得到 IAudioTrack 对象后, 调用 start 函数。

```

status_t AudioFlinger::TrackHandle::start() {

    return mTrack->start();

} //果然, 自己又不干活, 交给 mTrack 了, 这个是 PlayintThread createTrack_1 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_1 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-1 了
//奇怪，我们之前在看 createTrack-1 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-1(this);

    return status;
}

看看这个 addTrack-1 函数

status_t AudioFlinger::PlaybackThread::addTrack-1(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;

    }

    //我靠，有戏啊！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;

}

```

让我们想想吧。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。等我们以后分析再说

            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

到这里, 大家是不是有种焕然一新的感觉? 恩, 对了, AF 的工作就是如此的精密, 每个部分都配合得丝丝入扣。不过对于我们看代码的人来说, 实在搞不懂这么做的好处----哈哈 有点扯远了。

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。

```
uint32_t AudioFlinger::MixerThread::prepareTracks_l(const SortedVector< wp<Track> >&
activeTracks, Vector< sp<Track> > *tracksToRemove)
{

    uint32_t mixerStatus = MIXER_IDLE;

    //得到活跃 track 个数, 这里假设就是我们创建的那个 AT 吧, 那么 count=1
    size_t count = activeTracks.size();

    float masterVolume = mMasterVolume;

    bool masterMute = mMasterMute;

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

        sp<Track> t = activeTracks[i].promote();

        Track* const track = t.get();

        //得到 placement new 分配的那个跨进程共享的对象

        audio_track_cblk_t* cblk = track->cblk();

        //设置混音器, 当前活跃的 track。

        mAudioMixer->setActiveTrack(track->name());

        if (cblk->framesReady() && (track->isReady() || track->isStopped()) &&
            !track->isPaused() && !track->isTerminated())

        {

            // compute volume for this track

//AT 已经 write 数据了。所以肯定会进到这里来。

            int16_t left, right;

            if (track->isMuted() || masterMute || track->isPausing() ||
```

```
mStreamTypes[track->type()].mute) {  
  
    left = right = 0;  
  
    if (track->isPausing()) {  
  
        track->setPaused();  
  
    }  
  
    //AT 设置的音量假设不为零，我们需要聆听声音！  
  
    //所以走 else 流程  
  
    } else {  
  
        // read original volumes with volume control  
  
        float typeVolume = mStreamTypes[track->type()].volume;  
  
        float v = masterVolume * typeVolume;  
  
        float v_clamped = v * cblk->volume[0];  
  
        if (v_clamped > MAX_GAIN) v_clamped = MAX_GAIN;  
  
        left = int16_t(v_clamped);  
  
        v_clamped = v * cblk->volume[1];  
  
        if (v_clamped > MAX_GAIN) v_clamped = MAX_GAIN;  
  
        right = int16_t(v_clamped);  
  
    //计算音量  
  
    }  
  
    //注意，这里对混音器设置了数据提供来源，是一个 track，还记得我们前面说的吗？Track 从  
    AudioBufferProvider 派生  
  
    mAudioMixer->setBufferProvider(track);  
  
    mAudioMixer->enable(AudioMixer::MIXING);  
  
    int param = AudioMixer::VOLUME;  
  
    //为这个 track 设置左右音量等  
  
    mAudioMixer->setParameter(param, AudioMixer::VOLUME0, left);  
  
    mAudioMixer->setParameter(param, AudioMixer::VOLUME1, right);  
  
    mAudioMixer->setParameter(  
  
        AudioMixer::TRACK,  
  
        AudioMixer::FORMAT, track->format());  
  
    mAudioMixer->setParameter(  

```

```
        AudioManager::TRACK,

        AudioManager::CHANNEL_COUNT, track->channelCount());

    mAudioMixer->setParameter(

        AudioManager::RESAMPLE,

        AudioManager::SAMPLE_RATE,

        int(cbk->sampleRate));

    } else {

        if (track->isStopped()) {

            track->reset();

        }

        //如果这个 track 已经停止了，那么把它加到需要移除的 track 队列 tracksToRemove 中去
        //同时停止它在 AudioManager 中的混音

        if (track->isTerminated() || track->isStopped() || track->isPaused()) {

            tracksToRemove->add(track);

            mAudioMixer->disable(AudioMixer::MIXING);

        } else {

            mAudioMixer->disable(AudioMixer::MIXING);

        }

    }

}

// remove all the tracks that need to be...

count = tracksToRemove->size();

return mixerStatus;

}
```

看明白了吗？`prepare_1`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 AudioManager::process(void* output)

{

    mState.hook(&mState, output); //hook? 难道是钩子函数?

}
```

晕乎，就这么简单的函数???

CTRL+左键，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？没办法了，只能分析 AudioMixer 类了。

4. AudioMixer

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

```
AudioMixer::AudioMixer(size_t frameCount, uint32_t sampleRate)
    :   mActiveTrack(0), mTrackNames(0), mSampleRate(sampleRate)
{
    mState.enabledTracks= 0;
    mState.needsChanged = 0;
    mState.frameCount   = frameCount;
    mState.outputTemp   = 0;
    mState.resampleTemp = 0;
    mState.hook         = process__nop; //process__nop, 是该类的静态函数
    track_t* t = mState.tracks;
    //支持 32 路混音。牛死了
    for (int i=0 ; i<32 ; i++) {
        t->needs = 0;
        t->volume[0] = UNITY_GAIN;
        t->volume[1] = UNITY_GAIN;
        t->volumeInc[0] = 0;
        t->volumeInc[1] = 0;
        t->channelCount = 2;
        t->enabled = 0;
        t->format = 16;
        t->buffer.raw = 0;
        t->bufferProvider = 0;
        t->hook = 0;
        t->resampler = 0;
        t->sampleRate = mSampleRate;
        t->in = 0;
        t++;
    }
}
```



```

//其中, mState 是在 AudioMixer.h 中定义的一个数据结构

//注意, source insight 没办法解析这个 mState, 因为... 见下面的注释。

struct state_t {

    uint32_t        enabledTracks;

    uint32_t        needsChanged;

    size_t          frameCount;

    mix_t           hook;

    int32_t         *outputTemp;

    int32_t         *resampleTemp;

    int32_t         reserved[2];

    track_t         tracks[32]; // --attribute--((aligned(32))); 《--把这里注释掉

//否则 source insight 会解析不了这个 state_t 类型

};

int                mActiveTrack;

uint32_t          mTrackNames; //names? 搞得像字符串, 实际是一个 int

const uint32_t    mSampleRate;

state_t           mState

```

好了, 没什么吗。hook 对应的可选函数实现有:

```

process__validate

process__nop

process__genericNoResampling

process__genericResampling

process__OneTrack16BitsStereoNoResampling

process__TwoTracks16BitsStereoNoResampling

```

AudioMixer 构造的时候, hook 是 process__nop, 有几个地方会改变这个函数指针的指向。这部分涉及到数字音频技术, 我就无力讲解了。我们看看最接近的函数 process__OneTrack16BitsStereoNoResampling

```

void AudioMixer::process__OneTrack16BitsStereoNoResampling(state_t* state, void* output)
{
    单 track, 16bit 双声道, 不需要重采样, 大部分是这种情况了

    const int i = 31 - __builtin_clz(state->enabledTracks);

    const track_t& t = state->tracks[i];

```

```
AudioBufferProvider::Buffer& b(t.buffer);

int32_t* out = static_cast<int32_t*>(output);

size_t numFrames = state->frameCount;

const int16_t v1 = t.volume[0];

const int16_t vr = t.volume[1];

const uint32_t vr1 = t.volumeRL;

while (numFrames) {

    b.frameCount = numFrames;

//获得 buffer

    t.bufferProvider->getNextBuffer(&b);

    int16_t const *in = b.i16;

    size_t outFrames = b.frameCount;

    if UNLIKELY--->不走这.

    else {

        do {

            //计算音量等数据，和数字音频技术有关。这里不说了

            uint32_t r1 = *reinterpret_cast<uint32_t const *>(in);

            in += 2;

            int32_t l = mulRL(1, r1, vr1) >> 12;

            int32_t r = mulRL(0, r1, vr1) >> 12;

            *out++ = (r<<16) | (l & 0xFFFF);

        } while (--outFrames);

    }

    numFrames -= b.frameCount;

//释放 buffer。

    t.bufferProvider->releaseBuffer(&b);

}

}
```

好像挺简单的啊，不就是把数据处理下嘛。这里注意下 `buffer`。到现在，我们还没看到取共享内存里 AT 端 `write` 的数据呐。

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
AudioFlinger::PlaybackThread::Track::getNextBuffer(AudioBufferProvider::Buffer*
buffer)
{
//一阵暗喜吧。千呼万唤始出来，终于见到 cblk 了

    audio_track_cblk_t* cblk = this->cblk();

    uint32_t framesReady;

    uint32_t framesReq = buffer->frameCount;

//哈哈，看看数据准备好了没，

    framesReady = cblk->framesReady();

    if (LIKELY(framesReady)) {

        uint32_t s = cblk->server;

        uint32_t bufferEnd = cblk->serverBase + cblk->frameCount;

        bufferEnd = (cblk->loopEnd < bufferEnd) ? cblk->loopEnd : bufferEnd;

        if (framesReq > framesReady) {

            framesReq = framesReady;

        }

        if (s + framesReq > bufferEnd) {

            framesReq = bufferEnd - s;

        }

        获得真实的数据地址

        buffer->raw = getBuffer(s, framesReq);

        if (buffer->raw == 0) goto getNextBuffer_exit;

        buffer->frameCount = framesReq;

        return NO_ERROR;

    }
}
```

```
getNextBuffer_exit:  
  
    buffer->raw = 0;  
  
    buffer->frameCount = 0;  
  
    return NOT_ENOUGH_DATA;  
  
}
```

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

```
void AudioFlinger::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); // 哼哼，调用 cblk 的 stepServer，更新  
    服务端的使用位置  
  
    return result;  
  
}
```

到这里，大伙应该都明白了吧。原来 **AudioTrack** 中 **write** 的数据，最终是这么被使用的呀！！
恩，看一个 **process_OneTrack16BitsStereoNoResampling** 不过瘾，再看看
process_TwoTracks16BitsStereoNoResampling。

```
void AudioMixer::process_TwoTracks16BitsStereoNoResampling(state_t* state, void*  
output)  
  
    int i;  
  
    uint32_t en = state->enabledTracks;  
  
    i = 31 - __builtin_clz(en);  
  
    const track_t& t0 = state->tracks[i];  
  
    AudioBufferProvider::Buffer& b0(t0.buffer);
```

```
en &= ~(1<<i);

i = 31 - __builtin_clz(en);

const track_t& t1 = state->tracks[i];

AudioBufferProvider::Buffer& b1(t1.buffer);

int16_t const *in0;

const int16_t v10 = t0.volume[0];

const int16_t vr0 = t0.volume[1];

size_t frameCount0 = 0;

int16_t const *in1;

const int16_t v11 = t1.volume[0];

const int16_t vr1 = t1.volume[1];

size_t frameCount1 = 0;

int32_t* out = static_cast<int32_t*>(output);

size_t numFrames = state->frameCount;

int16_t const *buff = NULL;

while (numFrames) {

    if (frameCount0 == 0) {

        b0.frameCount = numFrames;

        t0.bufferProvider->getNextBuffer(&b0);

        if (b0.i16 == NULL) {

            if (buff == NULL) {

                buff = new int16_t[MAX_NUM_CHANNELS * state->frameCount];

            }

            in0 = buff;

            b0.frameCount = numFrames;


```

```
    } else {  
        in0 = b0.i16;  
    }  
  
    frameCount0 = b0.frameCount;  
}  
  
if (frameCount1 == 0) {  
    b1.frameCount = numFrames;  
  
    t1.bufferProvider->getNextBuffer (&b1);  
  
    if (b1.i16 == NULL) {  
        if (buff == NULL) {  
            buff = new int16_t[MAX_NUM_CHANNELS * state->frameCount];  
        }  
  
        in1 = buff;  
  
        b1.frameCount = numFrames;  
    } else {  
        in1 = b1.i16;  
    }  
  
    frameCount1 = b1.frameCount;  
}  
  
size_t outFrames = frameCount0 < frameCount1?frameCount0:frameCount1;  
  
numFrames -= outFrames;  
frameCount0 -= outFrames;  
frameCount1 -= outFrames;  
  
do {  
    int32_t l0 = *in0++;  
    int32_t r0 = *in0++;  
  
    l0 = mul(l0, v10);  
    r0 = mul(r0, vr0);  
  
    int32_t l = *in1++;
```

```
int32_t r = *in1++;

l = mulAdd(1, v11, 10) >> 12;

r = mulAdd(r, vr1, r0) >> 12;

// clamping...

l = clamp16(l);

r = clamp16(r);

*out++ = (r<<16) | (l & 0xFFFF);

} while (--outFrames);

if (frameCount0 == 0) {

    t0.bufferProvider->releaseBuffer(&b0);

}

if (frameCount1 == 0) {

    t1.bufferProvider->releaseBuffer(&b1);

}

}

if (buff != NULL) {

    delete [] buff;

}

}
```

看不懂了吧？哈哈，知道有这回事就行了，专门搞数字音频的需要好好研究下了！

三 再论共享 audio_track_cblk_t

为什么要再论这个？因为我在网上找了下，有人说 audio_track_cblk_t 是一个环形 buffer，环形 buffer 是什么意思？自己查查！

这个吗，和我之前的工作经历有关系，某 BOSS 费尽心机想搞一个牛掰掰的环形 buffer，搞得我累死了。现在 audio_track_cblk_t 是环形 buffer？我倒想看看它是如何实现。

顺便我们要解释下，audio_track_cblk_t 的使用和我之前说的 Lock,读/写,Unlock 不太一样。为何？

- 第一因为我们没在 AF 代码中看到有缓冲 buffer 方面的 wait，MixThread 只有当没有数据的时候会 usleep 一下。
- 第二，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

我们集中到 `audio_track_cblk_t` 这个类，来看看写者是如何使用的。写者就是 `AudioTrack` 端，在这个类中，叫 `user`

- `framesAvailable`，看看是否有空余空间
- `buffer`，获得写空间起始地址
- `stepUser`，更新 `user` 的位置。

2. 读者的使用

读者是 `AF` 端，在这个类中加 `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;
    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。

- 唉，又要发挥下了。`volatile` 只是告诉编译器，这个单元的地址不要 `cache` 到 `CPU` 的缓冲中。也就是每次取值的时候都要到实际内存中去读，而且可能读内存的时候先要锁一下总线。防止其他 `CPU` 核执行的时候同时去修改。由于是跨进程共享的内存，这块内存存在两个进程都是能见到的，又锁总线了，又是同一块内存，`volatile` 当然保证了同步一致性。
- `loopStart` 和 `loopEnd` 这两个值是表示循环播放的起点和终点的，下面还有一个 `loopCount` 吗，表示循环播放次数的

那就分析下吧。

先看写者的那几个函数

4 写者分析

先用 `framesAvailable` 看看当前剩余多少空间，我们可以假设是第一次进来嘛。读者还在那 `sleep` 呢。

```
uint32_t audio_track_cblk_t::framesAvailable()
{
    Mutex::Autolock _l(lock);

    return framesAvailable_l();
}

uint32_t audio_track_cblk_t::framesAvailable_l()
{
    uint32_t u = this->user; 当前写者位置，此时也为 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。

5 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;  
            }  
        }  
    }  
  
    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;
}
```

6 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()
{
    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。太牛了!
```