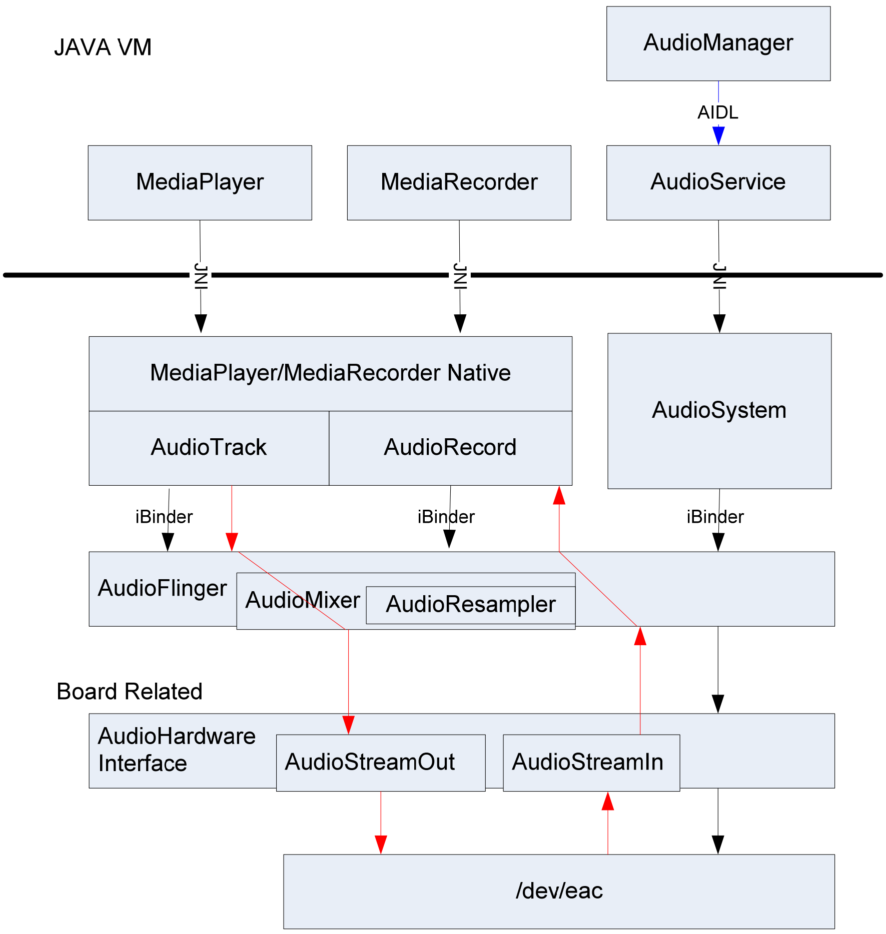
音频系统



初始化

在 system_init（运行在 Simulator 上）或者 main_Mediaserver 中，AudioFlinger 被创建，会生成一个 AudioHardwareInterface 实例（Android 定义的音频设备的一个抽象层），并且初始化音频系统的模式和路由信息如下：

```
mHardwareStatus = AUDIO_HW_IDLE;
mAudioHardware = AudioHardwareInterface::create();
mHardwareStatus = AUDIO_HW_INIT;
if (mAudioHardware->initCheck() == NO_ERROR) {
    // open 16-bit output stream for s/w mixer
    mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
    mOutput = mAudioHardware->openOutputStream(AudioSystem::PCM_16_BIT);
    mHardwareStatus = AUDIO_HW_IDLE;
    if (mOutput) {
        mSampleRate = mOutput->sampleRate();
        mChannelCount = mOutput->channelCount();
```

```
            mFormat = mOutput->format();
            mMixBufferSize = mOutput->bufferSize();
            mFrameCount = mMixBufferSize / mChannelCount / sizeof(int16_t);
            mMixBuffer = new int16_t[mFrameCount * mChannelCount];
            memset(mMixBuffer, 0, mMixBufferSize);
            mAudioMixer = new AudioMixer(mFrameCount, mSampleRate);
            // FIXME - this should come from settings
            setMasterVolume(1.0f);
            setRouting(AudioSystem::MODE_NORMAL, AudioSystem::ROUTE_SPEAKER,
AudioSystem::ROUTE_ALL);
            setRouting(AudioSystem::MODE_RINGTONE,
AudioSystem::ROUTE_SPEAKER, AudioSystem::ROUTE_ALL);
            setRouting(AudioSystem::MODE_IN_CALL, AudioSystem::ROUTE_EARPIECE,
AudioSystem::ROUTE_ALL);
            setMode(AudioSystem::MODE_NORMAL);
            mMasterMute = false;
        } else {
            LOGE("Failed to initialize output stream");
        }
    } else {
        LOGE("Couldn't even initialize the stubbed audio hardware!");
    }
```

在 SystemServer 启动的时候，会生成一个 AudioService 的实例，

```
            try {
                Log.i(TAG, "Starting Audio Service");
                ServiceManager.addService(Context.AUDIO_SERVICE,                    new
AudioService(context));
            } catch (Throwable e) {
                Log.e(TAG, "Failure starting Volume Service", e);
            }
```

AudioService 的构造函数会读取一些关于音频的配置信息，比如 Ringer 和 vibrate 信息，

```
    private void readPersistedSettings() {
        final ContentResolver cr = mContentResolver;

        mRingerMode        =        System.getInt(cr,        System.MODE_RINGER,
AudioManager.RINGER_MODE_NORMAL);
        mRingerModeAffectedStreams = System.getInt(mContentResolver,
                System.MODE_RINGER_STREAMS_AFFECTED,        1        <<
AudioSystem.STREAM_RING);

        mVibrateSetting = System.getInt(cr, System.VIBRATE_ON, 0);

        mMuteAffectedStreams = System.getInt(cr,
                System.MUTE_STREAMS_AFFECTED,
```

```
                    ((1        <<        AudioSystem.STREAM_MUSIC)|(1        <<
AudioSystem.STREAM_RING)|(1 << AudioSystem.STREAM_SYSTEM)));

        // Each stream will read its own persisted settings

        // Broadcast the sticky intent
        broadcastRingerMode();

        // Broadcast vibrate settings
        broadcastVibrateSetting(AudioManager.VIBRATE_TYPE_RINGER);
        broadcastVibrateSetting(AudioManager.VIBRATE_TYPE_NOTIFICATION);
    }
```

同时也会从底层音频系统读取模式和路由信息：

```
    private void readAudioSettings() {
        synchronized (mSettingsLock) {
            mMicMute = AudioSystem.isMicrophoneMuted();
            mMode = AudioSystem.getMode();
            for (int mode = 0; mode < AudioSystem.NUM_MODES; mode++) {
                mRoutes[mode] = AudioSystem.getRouting(mode);
            }
        }
    }
```

在 AudioSystem.cpp 第一次调用 get_audio_flinger 成功后，它会通过 binder 来监听运行在 media_server 进程中的 AudioFlinger 是否活着。

```
// establish binder interface to AudioFlinger service
const sp<IAudioFlinger>& AudioSystem::get_audio_flinger()
{
    Mutex::Autolock _l(gLock);
    if (gAudioFlinger.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.audio_flinger"));
            if (binder != 0)
                break;
            LOGW("AudioFlinger not published, waiting...");
            usleep(500000); // 0.5 s
        } while(true);
        if (gDeathNotifier == NULL) {
            gDeathNotifier = new DeathNotifier();
        } else {
            if (gAudioErrorCallback) {
                gAudioErrorCallback(NO_ERROR);
            }
```

```
            }
        binder->linkToDeath(gDeathNotifier);
        gAudioFlinger = interface_cast<IAudioFlinger>(binder);
    }
    LOGE_IF(gAudioFlinger==0, "no AudioFlinger!?");
    return gAudioFlinger;
}
```

到此，整个音频系统初始化完毕。

重新启动

如果 AudioFlinger 运行的 media_server 进程异常死掉，AudioSystem 会收到一个事件通知，

```
void AudioSystem::DeathNotifier::binderDied(const wp<IBinder>& who) {
    Mutex::Autolock _l(AudioSystem::gLock);
    AudioSystem::gAudioFlinger.clear();
    if (gAudioErrorCallback) {
        gAudioErrorCallback(DEAD_OBJECT);
    }
    LOGW("AudioFlinger server died!");
}
```

从而调用 android_media_AudioSystem.cpp 注册下来的回调函数，该函数又是通过 JNI 来调用 AudioService.java 注册下来的回调函数，在该函数中会发送 MSG_MEDIA_SERVER_DIED 消息，AudioService 会监听这个消息，这样 AudioService 就能知道 AudioFlinger 已不工作，它就接着调用 getMode 来尝试连接到重启后的 AudioFlinger。

```
                case MSG_MEDIA_SERVER_DIED:
                    Log.e(TAG, "Media server died.");
                    // Force creation of new IAudioflinger interface
                    mMediaServerOk = false;
                    AudioSystem.getMode();
                    break;
```

当连接成功后，AudioFlinger 会调用 android_media_AudioSystem.cpp 注册下来的回调函数，该函数又是通过 JNI 来调用 AudioService.java 注册下来的回调函数，在该函数中会发送 MSG_MEDIA_SERVER_STARTED 消息。接着 AudioService 就去配置底层音频系统，包括模式、路由、每一路流的音量大小和 Ringer 状态。

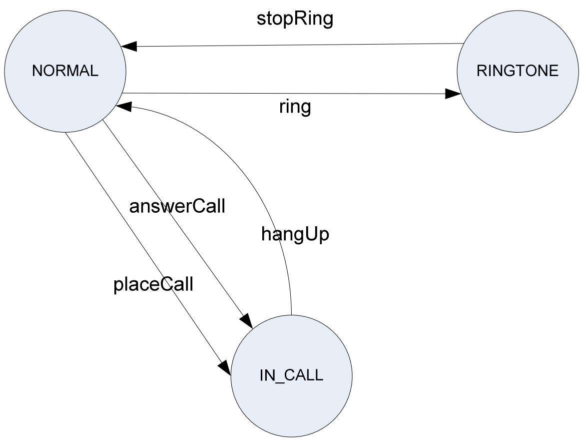
```
                    case MSG_MEDIA_SERVER_STARTED:
                    // Restore audio routing and stream volumes
                    applyAudioSettings();
                    for (int streamType = AudioSystem.NUM_STREAMS - 1; streamType
>= 0; streamType--) {
                        int volume;
                        VolumeStreamState streamState = mStreamStates[streamType];
                        if (streamState.muteCount() == 0) {
                            volume = streamState.mVolumes[streamState.mIndex];
```

```
                        } else {
                            volume = streamState.mVolumes[0];
                        }
                        AudioSystem.setVolume(streamType, volume);
                    }
                    setRingerMode();
```

Note: AudioSystem 的 Native 实现在 device/libs/android_runtime/android_media_AudioSystem.cpp 中。

模式

初始的时候音频系统是处于 MODE_NORMAL 模式的，下面是其模式状态变迁图：



问题：

当一个 Ringtone 放完了之后，理论上系统是否要自动切换回 NORMAL 模式而不是必须要主动调用 stopRing？我没找到相关 code。

路由信息

1. 当 HeadsetObserver 检测到有耳机插上来的时候，它会把音频系统的路由设置成均使用该耳机；当耳机被拔下来后，它会把音频系统的路由设置成缺省配置（即都通过扬声器）。

```
    private synchronized final void update(String newName, int newState) {
        if (newName != mHeadsetName || newState != mHeadsetState) {
            mHeadsetName = newName;
            mHeadsetState = newState;
            AudioManager audioManager = (AudioManager)
mContext.getSystemService(Context.AUDIO_SERVICE);

            if (mHeadsetState == 1) {
                audioManager.setRouting(AudioManager.MODE_NORMAL,
AudioManager.ROUTE_HEADSET,
                                         AudioManager.ROUTE_ALL);
                audioManager.setRouting(AudioManager.MODE_RINGTONE,
```

```
                                AudioManager.ROUTE_HEADSET | AudioManager.ROUTE_SPEAKER,
                                AudioManager.ROUTE_ALL);
                    audioManager.setRouting(AudioManager.MODE_IN_CALL, AudioManager.ROUTE_HEADSET,
                                AudioManager.ROUTE_ALL);
                } else {
                    audioManager.setRouting(AudioManager.MODE_NORMAL, AudioManager.ROUTE_SPEAKER,
                                AudioManager.ROUTE_ALL);
                    audioManager.setRouting(AudioManager.MODE_RINGTONE, AudioManager.ROUTE_SPEAKER,
                                AudioManager.ROUTE_ALL);
                    audioManager.setRouting(AudioManager.MODE_IN_CALL, AudioManager.ROUTE_EARPIECE,
                                AudioManager.ROUTE_ALL);
                }
                sendIntent();
            }
        }
```

PhoneApp 会接收到一个 ACTION_HEADSET_PLUG 的 Intent，往自身发送一个消息，处理如下：只是在没有蓝牙耳机或者未使用蓝牙耳机，而且该有线耳机是被拔掉的情况下，才把路由信息设置成 MODE_IN_CALL 走 ROUTE_SPEAKER。

```
                    case EVENT_WIRED_HEADSET_PLUG:
                        // Since the presence of a wired headset or bluetooth affects the
                        // speakerphone, update the "speaker" state.  We ONLY want to do
                        // this on the wired headset connect / disconnect events for now
                        // though, so we're only triggering on EVENT_WIRED_HEADSET_PLUG.
                        if (!isHeadsetPlugged() &&
                                (mBtHandsfree == null || !mBtHandsfree.isAudioOn())) {
                            // is the state is "not connected", restore the speaker state.
                            PhoneUtils.restoreSpeakerMode(getApplicationContext());
                        }
                        NotificationMgr.getDefault().updateSpeakerNotification();
                        break;
```

问题：

假设耳机插上来之前是通过蓝牙耳机在接听电话（或者听音乐）的，耳机插上来的时候系统就自动切换到使用耳机了，但是 PhoneApp 这个时候并不知道，它还以为在继续使用蓝牙耳机。但当耳机拔掉之后，怎么再切换回继续使用蓝牙耳机呢（系统默认是切换成 EARPIECE）？这个时候 PhoneApp 的状态应该是不对的。

2. 当在 Setting 里面把蓝牙耳机配对和 RFCOMM 连接上之后，BluetoothHeadsetService 会负责去和蓝牙耳机建立 SCO 连接，当连接完成之后 BluetoothHandsfree 会调用 AudioManager 的 setBluetoothScoOn 函数来通知音频系统去切换 MODE_IN_CALL 路由

信息到使用 ROUTE_BLUETOOTH。

```
/**
 * Sets audio routing to the Bluetooth headset on or off.
 *
 * @param on set <var>true</var> to route SCO (voice) audio to/from Bluetooth
 *           headset; <var>false</var> to route audio to/from phone earpiece
 */
public void setBluetoothScoOn(boolean on){
    setRouting(MODE_IN_CALL, on ? ROUTE_BLUETOOTH : ROUTE_EARPIECE,
ROUTE_ALL);
}
```

当蓝牙设备被关闭或者链接断掉的时候，BluetoothHeadsetService 会收到一个 DISABLED_ACTION 的 Intent，接着 BluetoothHandsfree 会调用 AudioManager 的 setBluetoothScoOn 函数来通知音频系统去切换 MODE_IN_CALL 路由信息到 ROUTE_EARPIECE。

问题：

setBluetoothScoOn 的实现在处理蓝牙设备被关闭的时候，是直接把路由信息改成到 ROUTE_EARPIECE，并没有恢复到使用蓝牙设备之前的信息状态。

Ringtone 是否需要在蓝牙耳机上播放呢？

音量控制

对外接口是 AudioManager

播放

音频系统对外的播放接口是 AudioTrack，每一路音频会对应一个 AudioTrack 的实例，它会通过 iBinder 来远程调用 AudioFlinger 的 createTrack 函数。

```
// create the track
sp<IAudioTrack> track = audioFlinger->createTrack(getpid(),
        streamType, sampleRate, format, channelCount, bufferCount, flags);
if (track == 0) {
    LOGE("AudioFlinger could not create track");
    return NO_INIT;
}
sp<IMemory> cblk = track->getCblk();
if (cblk == 0) {
    LOGE("Could not get control block");
    return NO_INIT;
}
```

而 AudioFlinger 的 createTrack 又会在内部生成一个 Track 实例，再将其包装成 TrackHandle 返回给 AudioTrack。

```
track = new Track(this, client, streamType, sampleRate, format,
```

```
            channelCount, bufferCount, channelCount == 1 ? mMixBufferSize>>1 : mMixBufferSize);
    mTracks.add(track);
    trackHandle = new TrackHandle(track);
    return trackHandle;
```

所以 AudioTrack 和 AudioFlinger 实际操作的都是 Track 实例，AudioTrack 通过它来执行控制操作（start/stop）和写入操作（write），AudioFlinger 则负责管理多个 Track（包括调用 AudioMixer 来混音）。两者之间的关系可以用生产者/消费者来类比，AudioTrack 是生产者，AudioFlinger 则是消费者。

AudioTrack

AudioTrack 的 start/stop 操作可以理解成一个开关，控制的是是否将与之对应的 Track 实例纳入 AudioFlinger 的管理中去，下面仅以 start 操作为例。

```
void AudioTrack::start()
{
    LOGV("start");
    if (mAudioTrackThread != 0) {
        mAudioTrackThread->mLock.lock();
    }

    if (android_atomic_or(1, &mActive) == 0) {
        setpriority(PRIO_PROCESS, 0, THREAD_PRIORITY_AUDIO_CLIENT);
        mActive = 1;
        mAudioTrack->start();
        if (mAudioTrackThread != 0) {
            mAudioTrackThread->run("AudioTrackThread", THREAD_PRIORITY_AUDIO_CLIENT);
        }
    }

    if (mAudioTrackThread != 0) {
        mAudioTrackThread->mLock.unlock();
    }
}

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

status_t AudioFlinger::Track::start()
{
    LOGV("start(%d)", mName);
    mAudioFlinger->addTrack(this);
    return NO_ERROR;
```

```
}

status_t AudioFlinger::addTrack(const sp<Track>& track)
{
    Mutex::Autolock _l(mLock);

    // here the track could be either new, or restarted
    // in both cases "unstop" the track
    if (track->isPaused()) {
        track->mState = TrackBase::RESUMING;
        LOGV("PAUSED => RESUMING (%d)", track->name());
    } else {
        track->mState = TrackBase::ACTIVE;
        LOGV("? => ACTIVE (%d)", track->name());
    }
    LOGV("mWaitWorkCV.broadcast");
    mWaitWorkCV.broadcast();

    if (mActiveTracks.indexOf(track) < 0) {
        // the track is newly added, make sure it fills up all its
        // buffers before playing. This is to ensure the client will
        // effectively get the latency it requested.
        track->mFillingUpStatus = Track::FS_FILLING;
        mActiveTracks.add(track);
        return NO_ERROR;
    }
    return ALREADY_EXISTS;
}
```

AudioTrack 的 write 则是往 audio_track_cblk_t 结构中写入数据。

```
ssize_t AudioTrack::write(const void* buffer, size_t userSize)
{
    LOGV("write %d bytes, mActive=%d", userSize, mActive);
    ssize_t written = 0;
    do {
        if (mPosition == 0) {
            status_t err = obtainBuffer(&mAudioBuffer, true);
            if (err < 0) {
                // out of buffers, return #bytes written
                if (err == status_t(NO_MORE_BUFFERS))
                    break;
                return ssize_t(err);
            }
        }
```

```
            size_t capacity = mAudioBuffer.size - mPosition;
            size_t toWrite = userSize < capacity ? userSize : capacity;

            memcpy(mAudioBuffer.i8 + mPosition, buffer, toWrite);
            buffer = static_cast<const int8_t*>(buffer) + toWrite;
            mPosition += toWrite;
            userSize -= toWrite;
            capacity -= toWrite;
            written += toWrite;

            if (capacity == 0) {
                mPosition = 0;
                releaseBuffer(&mAudioBuffer);
            }
        } while (userSize);

        return written;
}
```

AudioFlinger

AudioFlinger 对 Track 的管理是实现在 threadLoop 中的。先检测进入 standby 的超时是否到了，超时的话 AudioFlinger 会调用 AudioHardwareInterface 的 standby，这个是为省电考虑的。

```
        nsecs_t standbyTime = systemTime();

        do {
            enabledTracks = 0;
            { // scope for the lock
                Mutex::Autolock _l(mLock);
                const SortedVector< wp<Track> >& activeTracks = mActiveTracks;
                // put audio hardware into standby after short delay
                if UNLIKELY(systemTime() > standbyTime) {
                    // wait until we have something to do...
                    LOGD("Audio hardware entering standby\n");
                    mHardwareStatus = AUDIO_HW_STANDBY;
                    if (!mStandby) {
                        mAudioHardware->standby();
                        mStandby = true;
                    }
                    mHardwareStatus = AUDIO_HW_IDLE;
                    // we're about to wait, flush the binder command buffer
                    IPCThreadState::self()->flushCommands();
                    mWaitWorkCV.wait(mLock);
                    LOGD("Audio hardware exiting standby\n");
                    standbyTime = systemTime() + kStandbyTimeInNsecs;
```

```
            continue;
        }
```
如果未进 standby，接下来遍历所有当前 Active 的 Track 实例。
```
        // find out which tracks need to be processed
        size_t count = activeTracks.size();
        for (size_t i=0 ; i<count ; i++) {
            sp<Track> t = activeTracks[i].promote();
            if (t == 0) continue;

            Track* const track = t.get();
            audio_track_cblk_t* cblk = track->cblk();
            uint32_t u = cblk->user;
            uint32_t s = cblk->server;
            // The first time a track is added we wait
            // for all its buffers to be filled before processing it
            audioMixer().setActiveTrack(track->name());
```
当有某个 Track 的数据需要处理时（数据存储在 audio_track_cblk_t 结构中，其 user 域表明当前写入指针在 buffer 中的位置，server 域表明读取指针在 buffer 中的位置，所以只有当 user 大于 server 的时候说明有数据要处理），先计算该 Track 的 volume 信息，然后去配置针对该路 Track 的 Mixer 信息。
```
            if ((u > s) && (track->isReady(u, s) || track->isStopped()) &&
                    !track->isPaused())
            {
                // compute volume for this track

                // setup mixer needed information here
                AudioMixer& mixer(audioMixer());
                mixer.setBufferProvider(track);
                mixer.enable(AudioMixer::MIXING);

                enabledTracks++;
           } else {
```
最后进入真正的混音操作，再把混音过后的数据写到 AudioHardwareInterface 生成的 AudioOutputStream 中，由此整个音频输出完成。
```
    if (LIKELY(enabledTracks)) {
        // mix buffers...
        audioMixer().process(curBuf);

        // output audio to hardware
        mLastWriteTime = systemTime();
        mInWrite = true;
        mOutput->write(curBuf, mixBufferSize);
        mNumWrites++;
        mInWrite = false;
```

```
            mStandby = false;
            nsecs_t temp = systemTime();
            standbyTime = temp + kStandbyTimeInNsecs;
            nsecs_t delta = temp - mLastWriteTime;
            if (delta > maxPeriod) {
                LOGW("write blocked for %llu msecs", ns2ms(delta));
                mNumDelayedWrites++;
            }
            sleepTime = kBufferRecoveryInUsecs;
        } else {
```

当所有 Active 的 Track 都没有数据需要处理的时候，AudioFlinger 会 usleep 一段时间从而进入 standby。

```
            // There was nothing to mix this round, which means all
            // active tracks were late. Sleep a little bit to give
            // them another chance. If we're too late, the audio
            // hardware will zero-fill for us.
            LOGV("no buffers - usleep(%lu)", sleepTime);
            usleep(sleepTime);
            if (sleepTime < kMaxBufferRecoveryInUsecs) {
                sleepTime += kBufferRecoveryInUsecs;
            }
```

录制

音频系统对外的录制接口是 AudioRecord，它会通过 iBinder 来远程调用 AudioFlinger 的 openRecord 函数。

```
    // open record channel
    sp<IAudioRecord> record = audioFlinger->openRecord(getpid(), streamType,
            sampleRate, format, channelCount, bufferCount, flags);
    if (record == 0) {
        return NO_INIT;
    }
    sp<IMemory> cblk = record->getCblk();
    if (cblk == 0) {
        return NO_INIT;
    }
    if (cbf != 0) {
        mClientRecordThread = new ClientRecordThread(*this);
        if (mClientRecordThread == 0) {
            return NO_INIT;
        }
    }
```

而 AudioFlinger 的 openRecord 又会在内部先生成一个 AudioRecordThread 并且拿到 AudioStreamIn，

```
    // Create audio thread - take mutex to prevent race condition
    {
        Mutex::Autolock _l(mLock);
        if (mAudioRecordThread != 0) {
            LOGE("Record channel already open");
            goto Exit;
        }
        thread = new AudioRecordThread(this);
        mAudioRecordThread = thread;
    }
    // It's safe to release the mutex here since the client doesn't get a
    // handle until we return from this call

    // open driver, initialize h/w
    input = mAudioHardware->openInputStream(
            AudioSystem::PCM_16_BIT, channelCount, sampleRate);
```

再生成一个 RecordTrack 实例，将其包装成 RecordTrackHandle 返回给 AudioRecord。

```
    // create new record track and pass to record thread
    recordTrack = new RecordTrack(this, client, streamType, sampleRate,
            format, channelCount, bufferCount, input->bufferSize());

    // spin up record thread
    thread->open(recordTrack, input);
    thread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);

    // return to handle to client
    recordHandle = new RecordHandle(recordTrack);
```

所以 AudioRecord 和 AudioFlinger 实际操作的都是 RecordTrack 实例，AudioRecord 通过它来执行控制操作（start/stop）和读取操作（read），AudioFlinger 则负责从音频设备读取数据并放入 audio_track_cblk_t 结构中。两者之间的关系也可以用生产者/消费者来类比，AudioRecord 是消费者，AudioFlinger 则是生产者。

AudioRecord

AudioRecord 的 start/stop 操作可以理解成一个开关，控制的是 AudioRecordThread 的运行与否，下面仅以 start 操作为例。

```
status_t AudioRecord::start()
{
    status_t ret = NO_ERROR;

    // If using record thread, protect start sequence to make sure that
    // no stop command is processed before the thread is started
    if (mClientRecordThread != 0) {
        mRecordThreadLock.lock();
    }
```

```
        if (android_atomic_or(1, &mActive) == 0) {
                setpriority(PRIO_PROCESS, 0, THREAD_PRIORITY_AUDIO_CLIENT);
                ret = mAudioRecord->start();
                if (ret == NO_ERROR) {
                        if (mClientRecordThread != 0) {
                                mClientRecordThread->run("ClientRecordThread",
THREAD_PRIORITY_AUDIO_CLIENT);
                        }
                }
        }

        if (mClientRecordThread != 0) {
                mRecordThreadLock.unlock();
        }

        return ret;
}

status_t AudioFlinger::RecordHandle::start() {
        LOGV("RecordHandle::start()");
        return mRecordTrack->start();
}

status_t AudioFlinger::RecordTrack::start()
{
        return mAudioFlinger->startRecord();
}

status_t AudioFlinger::startRecord() {
        sp<AudioRecordThread> t = audioRecordThread();
        if (t == 0) return NO_INIT;
        return t->start();
}
```

AudioFlinger

从音频设备获取声音是实现在 AudioRecordThread::threadLoop 中的。

```
                        // promote strong ref so track isn't deleted while we access it
                        sp<RecordTrack> t = mRecordTrack.promote();

                        // if we lose the weak reference, client is gone.
                        if (t == 0) {
                                LOGV("AudioRecordThread: client deleted track");
                                break;
```

```
}

if (LIKELY(t->getNextBuffer(&mBuffer) == NO_ERROR)) {
        if (mInput->read(mBuffer.raw, t->mBufferSize) < 0) {
                LOGE("Error reading audio input");
                sleep(1);
        }
        t->releaseBuffer(&mBuffer);
}

// client isn't retrieving buffers fast enough
else {
        if (!t->setOverflow())
                LOGW("AudioRecordThread: buffer overflow");
}
```