Android源码学习笔记--AudioFinger

- 简介
- 构造方法

AduioFingler继承于 BinderService 和 BnAduioFinger 两个类。

```
class AudioFlinger :
    public BinderService<AudioFlinger>,
    public BnAudioFlinger
{
    friend class BinderService<AudioFlinger>;   // for AudioFlinger()

private:
    // FIXME The 400 is temporarily too high until a leak of writers in
media.log is fixed.
    static const size_t kLogMemorySize = 400 * 1024;
    sp<MemoryDealer>    mLogMemoryDealer;    // == 0 when NBLog is disabled
    // When a log writer is unregistered, it is done lazily so that
media.log can continue to see it
    // for as long as possible.  The memory is only freed when it is needed
for another log writer.
    Vector< sp<NBLog::Writer> > mUnregisteredWriters;
    Mutex               mUnregisteredWritersLock;
```

源码提供了最大 400*1024字节的缓存池作为日志存储。并且提供各种音量、EQ、PCM 写入等等接口方法。并且内部定义了一个 SyncEvent 的基础类，用于时间的传递和 会话的管理。这里还定义了一个 Client 类和一个 NotificationClient 类，作为客户端的通用定义，以及有一个 MediaLogNotifier 的输出线程。还有一个是 TrackHandle 是一个音频控制类，一个 RecordHandle 的录音控制类，MmapThreadHandle类用于数据流的接口。大致过了一遍 AudioFlinger 的大致内容如下：

```
class AudioFlinger : public BinderService<AudioFlinger>,
                     public BnAudioFlinger
{
    friend class BinderService<AudioFlinger>;

    class SyncEvent : public RefBase
    {
    }

// --- Client ---
    class Client : public RefBase
    {
    }
```

```
// --- Notification Client ---
    class NotificationClient : public IBinder::DeathRecipient
    {
    }

// --- MediaLogNotifier ---
// Thread in charge of notifying MediaLogService to start merging.
// Receives requests from AudioFlinger's binder activity. It is used to
reduce the amount of
// binder calls to MediaLogService in case of bursts of AudioFlinger binder
calls.
    class MediaLogNotifier : public Thread
    {
    }

    class TrackHandle;

    class RecordHandle;

    class RecordThread;

    class PlaybackThread;

    class MixerThread;

    class DirectOutputThread;

    class OffloadThread;

    class DuplicatingThread;

    class AsyncCallbackThread;

    class Track;

    class RecordTrack;

    class EffectModule;

    class EffectHandle;

    class EffectChain;

// server side of the client's IAudioTrack  音频播放处理单元
    class TrackHandle : public android::BnAudioTrack
    {
    }

// server side of the client's IAudioRecord 音频 Mic 处理单元
    class RecordHandle : public android::BnAudioRecord
```

```
    {
    }

// Mmap stream control interface implementation. Each MmapThreadHandle
controls one
// MmapPlaybackThread or MmapCaptureThread instance. 大文件存储处理类
    class MmapThreadHandle : public MmapStreamInterface
    {
    }

// AudioStreamIn is immutable, so their fields are const.
// For emphasis, we could also make all pointers to them be "const *",
// but that would clutter the code unnecessarily.

// AudioStreamIn 音频设备输入结构体定义
    struct AudioStreamIn
    {
        // 硬件设备映射
        AudioHwDevice* const audioHwDev;
        // 设备流 HAL 层接口
        sp <StreamInHalInterface> stream;
        // 输入标识位
        audio_input_flags_t       flags;
        // 获取 Device 的指针对象
        sp <DeviceHalInterface> hwDev()
        const
        { return audioHwDev->hwDevice(); }


        // 往设备中写入数据的接口函数方法
        AudioStreamIn(AudioHwDevice
                      * dev,
                      sp <StreamInHalInterface> in, audio_input_flags_t
                      flags) :


                audioHwDev(dev), stream(in), flags(flags)
        {}
    };

// for mAudioSessionRefs only
    struct AudioSessionRef
    {
        // AudioSession 定义，有 Pid 和 Cnt 进程号和信息
        AudioSessionRef(audio_session_t
                        sessionid, pid_t pid) :
                mSessionid(sessionid), mPid(pid), mCnt(1)
        {}


        const audio_session_t mSessionid;
```

```
        const pid_t             mPid;
        int                     mCnt;
    };

// 音频控制信号
// for dump, indicates which hardware operation is currently in progress
(but not stream ops)
    enum hardware_call_state
    {
        AUDIO_HW_IDLE = 0,               // no operation in progress
        AUDIO_HW_INIT,                   // init_check
        AUDIO_HW_OUTPUT_OPEN,            // open_output_stream
        AUDIO_HW_OUTPUT_CLOSE,           // unused
        AUDIO_HW_INPUT_OPEN,             // unused
        AUDIO_HW_INPUT_CLOSE,            // unused
        AUDIO_HW_STANDBY,                // unused
        AUDIO_HW_SET_MASTER_VOLUME,      // set_master_volume
        AUDIO_HW_GET_ROUTING,            // unused
        AUDIO_HW_SET_ROUTING,            // unused
        AUDIO_HW_GET_MODE,               // unused
        AUDIO_HW_SET_MODE,               // set_mode
        AUDIO_HW_GET_MIC_MUTE,           // get_mic_mute
        AUDIO_HW_SET_MIC_MUTE,           // set_mic_mute
        AUDIO_HW_SET_VOICE_VOLUME,       // set_voice_volume
        AUDIO_HW_SET_PARAMETER,          // set_parameters
        AUDIO_HW_GET_INPUT_BUFFER_SIZE,  // get_input_buffer_size
        AUDIO_HW_GET_MASTER_VOLUME,      // get_master_volume
        AUDIO_HW_GET_PARAMETER,          // get_parameters
        AUDIO_HW_SET_MASTER_MUTE,        // set_master_mute
        AUDIO_HW_GET_MASTER_MUTE,        // get_master_mute
    };
}
```

分段开始 AudioFinger 的学习，首先看到了构造方法，看他做了哪些事情：

```
AudioFlinger::AudioFlinger()
    : BnAudioFlinger(),
      mMediaLogNotifier(new AudioFlinger::MediaLogNotifier()),
      mPrimaryHardwareDev(NULL),
      mAudioHwDevs(NULL),
      mHardwareStatus(AUDIO_HW_IDLE),
      mMasterVolume(1.0f),
      mMasterMute(false),
      // mNextUniqueId(AUDIO_UNIQUE_ID_USE_MAX),
      mMode(AUDIO_MODE_INVALID),
      mBtNrecIsOff(false),
      mIsLowRamDevice(true),
      mIsDeviceTypeKnown(false),
      mGlobalEffectEnableTime(0),
      mSystemReady(false)
{
```

```
    // unsigned instead of audio_unique_id_use_t, because ++ operator is
unavailable for enum
    for (unsigned use = AUDIO_UNIQUE_ID_USE_UNSPECIFIED; use <
AUDIO_UNIQUE_ID_USE_MAX; use++) {
        // zero ID has a special meaning, so unavailable
        mNextUniqueIds[use] = AUDIO_UNIQUE_ID_USE_MAX;
    }
    //初始化 log
    getpid_cached = getpid();
    const bool doLog = property_get_bool("ro.test_harness", false);
    if (doLog) {
        mLogMemoryDealer = new MemoryDealer(kLogMemorySize, "LogWriters",
                MemoryHeapBase::READ_ONLY);
        (void) pthread_once(&sMediaLogOnce, sMediaLogInit);
    }

    // reset battery stats.
    // if the audio service has crashed, battery stats could be left
    // in bad state, reset the state upon service start.
    // 初始化 audio 硬件 上电
    BatteryNotifier::getInstance().noteResetAudio();
    // 创建 device 和 effects HAL 层接口
    mDevicesFactoryHal = DevicesFactoryHalInterface::create();
    mEffectsFactoryHal = EffectsFactoryHalInterface::create();

    mMediaLogNotifier->run("MediaLogNotifier");

#ifdef TEE_SINK
    char value[PROPERTY_VALUE_MAX];
    (void) property_get("ro.debuggable", value, "0");
    int debuggable = atoi(value);
    int teeEnabled = 0;
    if (debuggable) {
        (void) property_get("af.tee", value, "0");
        teeEnabled = atoi(value);
    }
    // FIXME symbolic constants here
    if (teeEnabled & 1) {
        mTeeSinkInputEnabled = true;
    }
    if (teeEnabled & 2) {
        mTeeSinkOutputEnabled = true;
    }
    if (teeEnabled & 4) {
        mTeeSinkTrackEnabled = true;
    }
#endif
}
```

- 功能接口

其次，可以看一下其余部分，如设置参数音量，静音等。

```
status_t AudioFlinger::setVoiceVolume(float value)
{
    status_t ret = initCheck();
    if (ret != NO_ERROR) {
        return ret;
    }

    // check calling permissions
    if (!settingsAllowed()) {
        return PERMISSION_DENIED;
    }

    AutoMutex lock(mHardwareLock);
    sp<DeviceHalInterface> dev = mPrimaryHardwareDev->hwDevice();
    mHardwareStatus = AUDIO_HW_SET_VOICE_VOLUME;
    ret = dev->setVoiceVolume(value);
    mHardwareStatus = AUDIO_HW_IDLE;

    return ret;
}
```

但凡需要控制 HAL 以下接口的，都需要初始化时序，并且判断权限，在从 mPrimaryHardwareDev 中取得一个 HwDevice 的接口对象。那么 HAL层 Device 是如何与上层建立关系的呢?

- 与 HAL 的功能结构

framework 源码在/framework/av/include/media/audiohal/DeviceHalInterface.h 中定义，这个类是 HAL 提供给 Audio 的接口功能类。audioFinger 是用这个 HAL 抽象出来的接口方法来控制更底层的 device 接口。

```
namespace android {

class StreamInHalInterface;
class StreamOutHalInterface;

class DeviceHalInterface : public RefBase
{
  public:
    // Sets the value of 'devices' to a bitmask of 1 or more values of audio_devices_t.
    virtual status_t getSupportedDevices(uint32_t *devices) = 0;

    // Check to see if the audio hardware interface has been initialized.
    virtual status_t initCheck() = 0;
```

```
    // Set the audio volume of a voice call. Range is between 0.0 and 1.0.
    virtual status_t setVoiceVolume(float volume) = 0;

    // Set the audio volume for all audio activities other than voice call.
    virtual status_t setMasterVolume(float volume) = 0;

    // Get the current master volume value for the HAL.
    virtual status_t getMasterVolume(float *volume) = 0;

    // Called when the audio mode changes.
    virtual status_t setMode(audio_mode_t mode) = 0;

    // Muting control.
    virtual status_t setMicMute(bool state) = 0;
    virtual status_t getMicMute(bool *state) = 0;
    virtual status_t setMasterMute(bool state) = 0;
    virtual status_t getMasterMute(bool *state) = 0;

    // Set global audio parameters.
    virtual status_t setParameters(const String8& kvPairs) = 0;

    // Get global audio parameters.
    virtual status_t getParameters(const String8& keys, String8 *values) = 0;

    // Returns audio input buffer size according to parameters passed.
    virtual status_t getInputBufferSize(const struct audio_config *config,
            size_t *size) = 0;

    // Creates and opens the audio hardware output stream. The stream is closed
    // by releasing all references to the returned object.
    virtual status_t openOutputStream(
            audio_io_handle_t handle,
            audio_devices_t devices,
            audio_output_flags_t flags,
            struct audio_config *config,
            const char *address,
            sp<StreamOutHalInterface> *outStream) = 0;

    // Creates and opens the audio hardware input stream. The stream is closed
    // by releasing all references to the returned object.
    virtual status_t openInputStream(
            audio_io_handle_t handle,
            audio_devices_t devices,
            struct audio_config *config,
```

```
            audio_input_flags_t flags,
            const char *address,
            audio_source_t source,
            sp<StreamInHalInterface> *inStream) = 0;

    // Returns whether createAudioPatch and releaseAudioPatch operations are supported.
    virtual status_t supportsAudioPatches(bool *supportsPatches) = 0;

    // Creates an audio patch between several source and sink ports.
    virtual status_t createAudioPatch(
            unsigned int num_sources,
            const struct audio_port_config *sources,
            unsigned int num_sinks,
            const struct audio_port_config *sinks,
            audio_patch_handle_t *patch) = 0;

    // Releases an audio patch.
    virtual status_t releaseAudioPatch(audio_patch_handle_t patch) = 0;

    // Fills the list of supported attributes for a given audio port.
    virtual status_t getAudioPort(struct audio_port *port) = 0;

    // Set audio port configuration.
    virtual status_t setAudioPortConfig(const struct audio_port_config *config) = 0;

    virtual status_t dump(int fd) = 0;

  protected:
    // Subclasses can not be constructed directly by clients.
    DeviceHalInterface() {}

    // The destructor automatically closes the device.
    virtual ~DeviceHalInterface() {}
};

} // namespace android
```

完整的调用体系大体上可以用图来表示：

那么 AudioFinger如何与 DeviceHalInterface 进行 bind 呢，这里主要看源码 AudioFinger 中的一段代码：

```
// ------------------------------------------------------------------------
```

```
audio_module_handle_t AudioFlinger::loadHwModule(const char *name)
{
    if (name == NULL) {
        return AUDIO_MODULE_HANDLE_NONE;
    }
    if (!settingsAllowed()) {
        return AUDIO_MODULE_HANDLE_NONE;
    }
    Mutex::Autolock _l(mLock);
    return loadHwModule_l(name);
}

// loadHwModule_l() must be called with AudioFlinger::mLock held
audio_module_handle_t AudioFlinger::loadHwModule_l(const char *name)
{
    for (size_t i = 0; i < mAudioHwDevs.size(); i++) {
        if (strncmp(mAudioHwDevs.valueAt(i)->moduleName(), name,
strlen(name)) == 0) {
            ALOGW("loadHwModule() module %s already loaded", name);
            return mAudioHwDevs.keyAt(i);
        }
    }

    sp<DeviceHalInterface> dev;

    int rc = mDevicesFactoryHal->openDevice(name, &dev);
    if (rc) {
        ALOGE("loadHwModule() error %d loading module %s", rc, name);
        return AUDIO_MODULE_HANDLE_NONE;
    }

    mHardwareStatus = AUDIO_HW_INIT;
    rc = dev->initCheck();
    mHardwareStatus = AUDIO_HW_IDLE;
    if (rc) {
        ALOGE("loadHwModule() init check error %d for module %s", rc,
name);
        return AUDIO_MODULE_HANDLE_NONE;
    }

    // Check and cache this HAL's level of support for master mute and
master
    // volume.  If this is the first HAL opened, and it supports the get
    // methods, use the initial values provided by the HAL as the current
    // master mute and volume settings.

    AudioHwDevice::Flags flags = static_cast<AudioHwDevice::Flags>(0);
    {  // scope for auto-lock pattern
        AutoMutex lock(mHardwareLock);
```

```
        if (0 == mAudioHwDevs.size()) {
            mHardwareStatus = AUDIO_HW_GET_MASTER_VOLUME;
            float mv;
            if (OK == dev->getMasterVolume(&mv)) {
                mMasterVolume = mv;
            }

            mHardwareStatus = AUDIO_HW_GET_MASTER_MUTE;
            bool mm;
            if (OK == dev->getMasterMute(&mm)) {
                mMasterMute = mm;
            }
        }

        mHardwareStatus = AUDIO_HW_SET_MASTER_VOLUME;
        if (OK == dev->setMasterVolume(mMasterVolume)) {
            flags = static_cast<AudioHwDevice::Flags>(flags |
                    AudioHwDevice::AHWD_CAN_SET_MASTER_VOLUME);
        }

        mHardwareStatus = AUDIO_HW_SET_MASTER_MUTE;
        if (OK == dev->setMasterMute(mMasterMute)) {
            flags = static_cast<AudioHwDevice::Flags>(flags |
                    AudioHwDevice::AHWD_CAN_SET_MASTER_MUTE);
        }

        mHardwareStatus = AUDIO_HW_IDLE;
    }

    audio_module_handle_t handle = (audio_module_handle_t)
nextUniqueId(AUDIO_UNIQUE_ID_USE_MODULE);
    mAudioHwDevs.add(handle, new AudioHwDevice(handle, name, dev, flags));

    ALOGI("loadHwModule() Loaded %s audio interface, handle %d", name,
handle);

    return handle;

}
```

在他们的上一级，会发现一个 findSuitableHwDev的方法，并且找到三个播放介质，更上一级是 openOutput 的方法，打开相应的 device。

```
static const char * const audio_interfaces[] = {
    AUDIO_HARDWARE_MODULE_ID_PRIMARY,
    AUDIO_HARDWARE_MODULE_ID_A2DP,
    AUDIO_HARDWARE_MODULE_ID_USB,
};

AudioHwDevice* AudioFlinger::findSuitableHwDev_l(
```

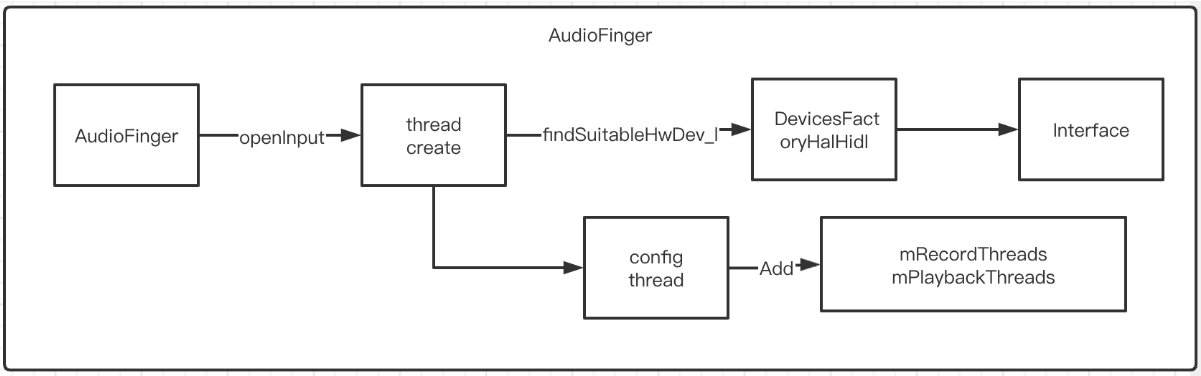
```
        audio_module_handle_t module,
        audio_devices_t devices)
{
    // if module is 0, the request comes from an old policy manager and we
should load
    // well known modules
    if (module == 0) {
        ALOGW("findSuitableHwDev_l() loading well know audio hw modules");
        for (size_t i = 0; i < arraysize(audio_interfaces); i++) {
            loadHwModule_l(audio_interfaces[i]);
        }
        // then try to find a module supporting the requested device.
        for (size_t i = 0; i < mAudioHwDevs.size(); i++) {
            AudioHwDevice *audioHwDevice = mAudioHwDevs.valueAt(i);
            sp<DeviceHalInterface> dev = audioHwDevice->hwDevice();
            uint32_t supportedDevices;
            if (dev->getSupportedDevices(&supportedDevices) == OK &&
                    (supportedDevices & devices) == devices) {
                return audioHwDevice;
            }
        }
    } else {
        // check a match for the requested module handle
        AudioHwDevice *audioHwDevice = mAudioHwDevs.valueFor(module);
        if (audioHwDevice != NULL) {
            return audioHwDevice;
        }
    }
    return NULL;
}
```

在追述代码到 HAL 层驱动部分，看到 framework/av/media/libaudiohal/DevicesFactoryHalHidl.cpp 下面，里面存在一个静态方法和工厂。用于HAL 接口 bind。

```
// static
status_t DevicesFactoryHalHidl::nameFromHal(const char *name,
IDevicesFactory::Device *device) {
    if (strcmp(name, AUDIO_HARDWARE_MODULE_ID_PRIMARY) == 0) {
        *device = IDevicesFactory::Device::PRIMARY;
        return OK;
    } else if(strcmp(name, AUDIO_HARDWARE_MODULE_ID_A2DP) == 0) {
        *device = IDevicesFactory::Device::A2DP;
        return OK;
    } else if(strcmp(name, AUDIO_HARDWARE_MODULE_ID_USB) == 0) {
        *device = IDevicesFactory::Device::USB;
        return OK;
    } else if(strcmp(name, AUDIO_HARDWARE_MODULE_ID_REMOTE_SUBMIX) == 0)
{
```

```
        *device = IDevicesFactory::Device::R_SUBMIX;
        return OK;
    } else if(strcmp(name, AUDIO_HARDWARE_MODULE_ID_STUB) == 0) {
        *device = IDevicesFactory::Device::STUB;
        return OK;
    }
    ALOGE("Invalid device name %s", name);
    return BAD_VALUE;
}

status_t DevicesFactoryHalHidl::openDevice(const char *name, sp<DeviceHalInterface> *device) {
    if (mDevicesFactory == 0) return NO_INIT;
    IDevicesFactory::Device hidlDevice;
    status_t status = nameFromHal(name, &hidlDevice);
    if (status != OK) return status;
    Result retval = Result::NOT_INITIALIZED;
    Return<void> ret = mDevicesFactory->openDevice(
            hidlDevice,
            [&](Result r, const sp<IDevice>& result) {
                retval = r;
                if (retval == Result::OK) {
                    *device = new DeviceHalHidl(result);
                }
            });
    if (ret.isOk()) {
        if (retval == Result::OK) return OK;
        else if (retval == Result::INVALID_ARGUMENTS) return BAD_VALUE;
        else return NO_INIT;
    }
    return FAILED_TRANSACTION;
}

```

大致的启动流程：



这里新来一个 HWInterface 会 create 一个 thread，并把它放到 record 线程容器或者是 Playback 线程容器中。

- AudioTrack 与 Client 关系

首先上层注册下来是一个 客户端，就是创建了一个 AudioTrack，这里会进行一个双向的接口 IPC 通信。这里看下回调接口av/include/media/IAudioFingerClient.h：

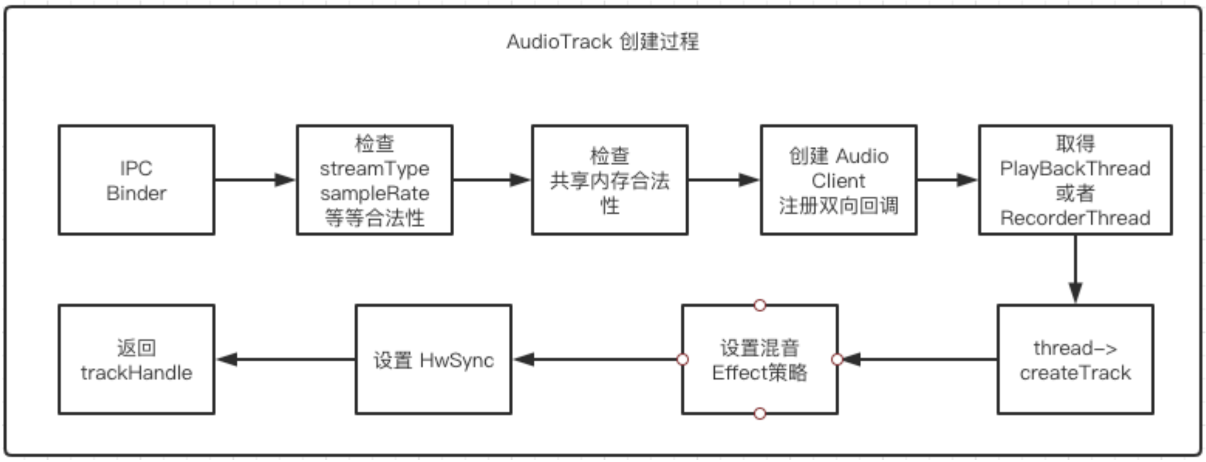
```
class IAudioFlingerClient : public IInterface
{
public:
    DECLARE_META_INTERFACE(AudioFlingerClient);

    // Notifies a change of audio input/output configuration.
    virtual void ioConfigChanged(audio_io_config_event event,
                                 const sp<AudioIoDescriptor>& ioDesc) = 0;

};


// ---------------------------------------------------------------------------
----

class BnAudioFlingerClient : public BnInterface<IAudioFlingerClient>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};

// ---------------------------------------------------------------------------
----

}; // namespace android
```

注册部分代码：在 AudioFinger 创建一个 Track，这里和 AIDL 一样，是跨进程通信。前面会确认权限、shareBuffer、参数是否匹配。可以看代码：

```
sp<IAudioTrack> AudioFlinger::createTrack(
        audio_stream_type_t streamType,
        uint32_t sampleRate,
        audio_format_t format,
        audio_channel_mask_t channelMask,
        size_t *frameCount,
        audio_output_flags_t *flags,
        const sp<IMemory>& sharedBuffer,
        audio_io_handle_t output,
        pid_t pid,
        pid_t tid,
        audio_session_t *sessionId,
        int clientUid,
```

```
        status_t *status,
        audio_port_handle_t portId)
{
    sp<PlaybackThread::Track> track;
    sp<TrackHandle> trackHandle;
    sp<Client> client;
    status_t lStatus;
    audio_session_t lSessionId;
    // 关键代码，从容器中取得 play back thread
    PlaybackThread *thread = checkPlaybackThread_l(output);
    ...
    // 注册客户端
    client = registerPid(pid);

    PlaybackThread *effectThread = NULL;
    if (sessionId != NULL && *sessionId != AUDIO_SESSION_ALLOCATE) {
        if (audio_unique_id_get_use(*sessionId) !=
AUDIO_UNIQUE_ID_USE_SESSION) {
            ALOGE("createTrack() invalid session ID %d", *sessionId);
            lStatus = BAD_VALUE;
            goto Exit;
        }

    …
    // 添加任务
    setAudioHwSyncForSession_l(thread, lSessionId);

  // return handle to client
    trackHandle = new TrackHandle(track);

Exit:
    *status = lStatus;
    return trackHandle;
}
```

这边的流程可以看到大致上是这样的一个逻辑调度。



若是底下内存发生变化，会使用 IAudioFingerClient 的回调接口返回事件，以及文件的

句柄，具体代码以及完整的关联：

```
class IAudioFlingerClient : public IInterface
{
public:
    DECLARE_META_INTERFACE(AudioFlingerClient);

    // Notifies a change of audio input/output configuration.
    virtual void ioConfigChanged(audio_io_config_event event,
                                 const sp<AudioIoDescriptor>& ioDesc) = 0;

};
```