Android7.0 AudioRecord

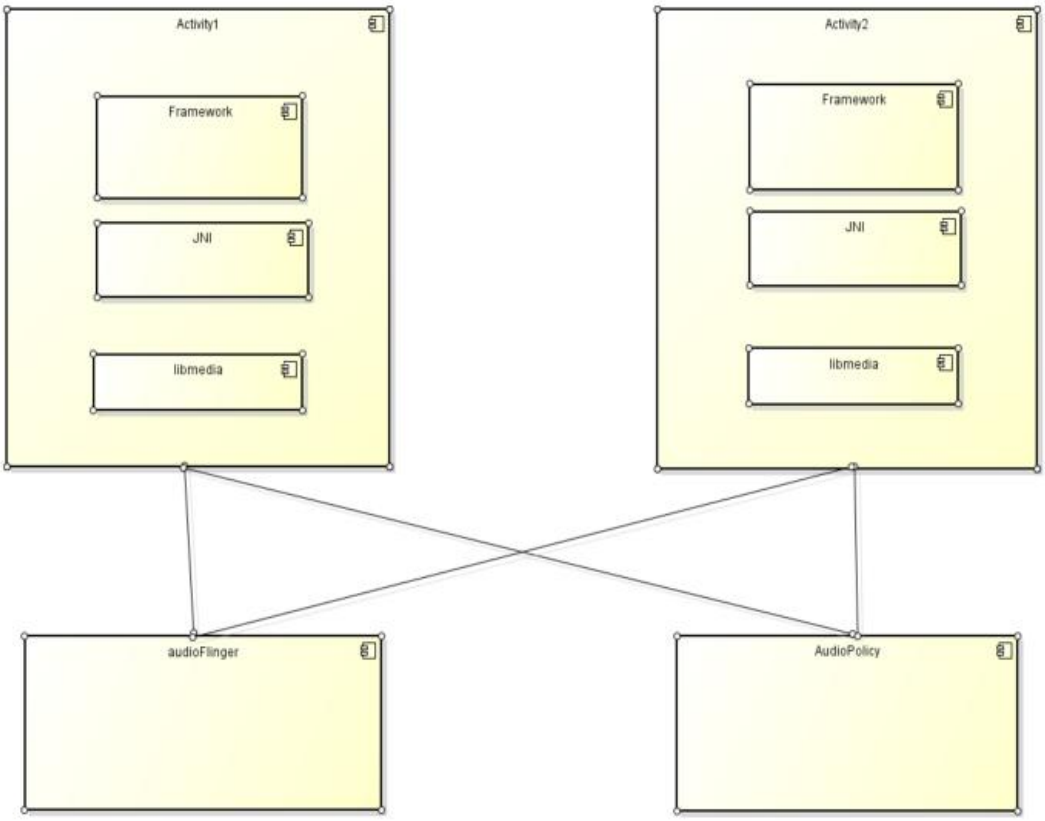
部分相关流程学习

—赵春春
—2021 年 2 月 4 日星期四

目录

Audio 架构

下图是基于 Android 的 Audio APP 的架构图：



从图中可以看出，每个 APP 都有自己独立的 Java 部分的 Audio Code，JNI 部分的 code 以及 Native 部分的 code，但是对于 AudioFlinger 和 AudioPolicy 则是在整个系统中是独一份的，所有的 APP 的共用 AudioFlinger 和 AudioPolicy 的 service。

1. AudioRecord 相关的名词介绍
   (1) 音频源：一般可以使用麦克风作为采集音频的数据源。（包括 MediaRecorder.AudioSource.MIC 和 MediaRecorder.AudioSource.VOICE_CALL 等等，详见 MediaRecorder.AudioSource 类）
   (2) 采样率：一秒内对声音数据的采样次数，采样率越高，音质越好。（包括 48KHz 和 96KHz 等等）
   (3) 通道：单声道，双声道等（包括 CHANNEL_IN_MONO 和 CHANNEL_IN_STEREO 等，详见 AudioFormat.java）。
   (4) 音频格式：一般选用 PCM 格式，即原始的音频样本（包括 ENCODING_PCM_8BIT 和 ENCODING_PCM_16BIT 等，详见 AudioFormat.java）。
   (5) 缓冲区大小：音频数据写入缓冲区的总数，通过 AudioRecord.getMinBufferSize 获取最小的缓冲区。
   (6) PCM：脉冲编码调制，通用的低级别的数字音频编码.音频信号被采样，然后根据位深在合适的范围内被量化成离散值。比如，对于 16 位的 PCM, 采样值是介于 -32768 到+32767 之间。

2. Android 中 Audio 各模块的代码位置

(1) Java Framework 部分：

\frameworks\base\media\java\android\media

```
zhaochch@lnxsrv5:~/code/g200/frameworks/base/media/java/android/media$ ls  -al *AudioRe*
-rw-rw-r-- 1 zhaochch zhaochch   677 1月  20 10:43 AudioRecordingConfiguration.aidl
-rw-rw-r-- 1 zhaochch zhaochch  8696 1月  20 10:43 AudioRecordingConfiguration.java
-rw-rw-r-- 1 zhaochch zhaochch 77693 1月  20 10:43 AudioRecord.java
-rw-rw-r-- 1 zhaochch zhaochch  1180 1月  20 10:43 AudioRecordRoutingProxy.java
zhaochch@lnxsrv5:~/code/g200/frameworks/base/media/java/android/media$ ls  -al *AudioT*
-rw-rw-r-- 1 zhaochch zhaochch   3481 1月  20 10:43 AudioTimestamp.java
-rw-rw-r-- 1 zhaochch zhaochch 130761 1月  20 10:43 AudioTrack.java
-rw-rw-r-- 1 zhaochch zhaochch   1168 1月  20 10:43 AudioTrackRoutingProxy.java
```

(2) JNI 部分:

\frameworks\base\core\jni

```
-rw-rw-r-- 1 zhaochch zhaochch  1905 1月  20 10:43 android_media_AudioErrors.h
-rw-rw-r-- 1 zhaochch zhaochch  5279 1月  20 10:43 android_media_AudioFormat.h
-rw-rw-r-- 1 zhaochch zhaochch 33974 1月  20 10:43 android_media_AudioRecord.cpp
-rw-rw-r-- 1 zhaochch zhaochch 77420 1月  20 10:43 android_media_AudioSystem.cpp
-rw-rw-r-- 1 zhaochch zhaochch 52459 1月  20 10:43 android_media_AudioTrack.cpp
-rw-rw-r-- 1 zhaochch zhaochch  1041 1月  20 10:43 android_media_AudioTrack.h
-rw-rw-r-- 1 zhaochch zhaochch  2560 1月  20 10:43 android_media_DeviceCallback.cpp
-rw-rw-r-- 1 zhaochch zhaochch  1553 1月  20 10:43 android_media_DeviceCallback.h
-rw-rw-r-- 1 zhaochch zhaochch 20189 1月  20 10:43 android_media_JetPlayer.cpp
-rw-rw-r-- 1 zhaochch zhaochch  6772 1月  20 10:43 android_media_RemoteDisplay.cpp
-rw-rw-r-- 1 zhaochch zhaochch  4921 1月  20 10:43 android_media_ToneGenerator.cpp
```

(3) Native Framework(libmedia)部分：

\frameworks\av\media\libmedia

```
-rw-rw-r-- 1 zhaochch zhaochch  14441 1月  20 10:43 AudioEffect.cpp
-rw-rw-r-- 1 zhaochch zhaochch   4959 1月  20 10:43 AudioParameter.cpp
-rw-rw-r-- 1 zhaochch zhaochch   3676 1月  20 10:43 AudioPolicy.cpp
-rw-rw-r-- 1 zhaochch zhaochch  44792 1月  20 14:36 AudioRecord.cpp
-rw-rw-r-- 1 zhaochch zhaochch  44396 1月  20 10:43 AudioSystem.cpp
-rw-rw-r-- 1 zhaochch zhaochch 105732 1月  20 14:37 AudioTrack.cpp
-rw-rw-r-- 1 zhaochch zhaochch  41883 1月  20 10:43 AudioTrackShared.cpp
-rw-rw-r-- 1 zhaochch zhaochch   3142 1月  20 10:43 IAudioFlingerClient.cpp
-rw-rw-r-- 1 zhaochch zhaochch  56645 1月  20 10:43 IAudioFlinger.cpp
-rw-rw-r-- 1 zhaochch zhaochch   5369 1月  20 10:43 IAudioPolicyServiceClient.cpp
-rw-rw-r-- 1 zhaochch zhaochch  55728 1月  20 10:43 IAudioPolicyService.cpp
-rw-rw-r-- 1 zhaochch zhaochch   2674 1月  20 10:43 IAudioRecord.cpp
-rw-rw-r-- 1 zhaochch zhaochch   6519 1月  20 10:43 IAudioTrack.cpp
```

(4) AudioFlinger 部分：

\frameworks\av\services\audioflinger

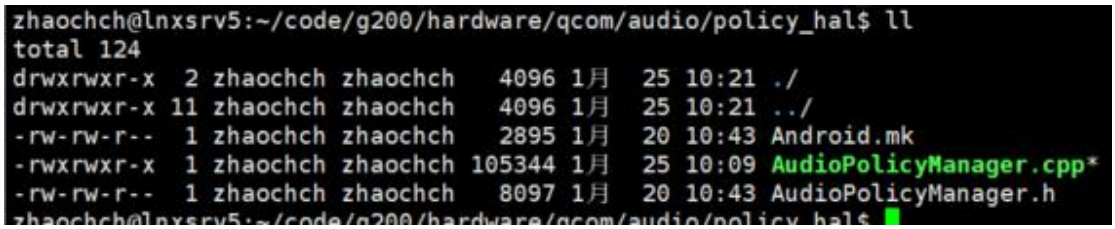
```
zhaochch@lnxsrv5:~/code/g200/frameworks/av/services/audioflinger$ ls -al *Audio*
-rw-rw-r-- 1 zhaochch zhaochch 116345 1月  20 12:29 AudioFlinger.cpp
-rw-rw-r-- 1 zhaochch zhaochch  34605 1月  20 10:43 AudioFlinger.h
-rw-rw-r-- 1 zhaochch zhaochch   3291 1月  20 10:43 AudioHwDevice.cpp
-rw-rw-r-- 1 zhaochch zhaochch   2798 1月  20 10:43 AudioHwDevice.h
-rw-rw-r-- 1 zhaochch zhaochch  80527 1月  20 10:43 AudioMixer.cpp
-rw-rw-r-- 1 zhaochch zhaochch  16393 1月  20 10:43 AudioMixer.h
-rw-rw-r-- 1 zhaochch zhaochch  14967 1月  20 10:43 AudioMixerOps.h
-rw-rw-r-- 1 zhaochch zhaochch  29137 1月  20 10:43 AudioResampler.cpp
-rw-rw-r-- 1 zhaochch zhaochch   6049 1月  20 10:43 AudioResamplerCubic.cpp
-rw-rw-r-- 1 zhaochch zhaochch   2410 1月  20 10:43 AudioResamplerCubic.h
```

(5) AudioPolicy 部分：

\frameworks\av\services\audiopolicy

```
zhaochch@lnxsrv5:~/code/g200/frameworks/av/services/audiopolicy$ ls -al
total 80
drwxrwxr-x 11 zhaochch zhaochch  4096 1月  20 10:43 .
drwxrwxr-x 12 zhaochch zhaochch  4096 1月  20 10:43 ..
-rw-rw-r--  1 zhaochch zhaochch  4356 1月  20 10:43 Android.mk
-rw-rw-r--  1 zhaochch zhaochch  5491 1月  20 10:43 audio_policy.conf
-rw-rw-r--  1 zhaochch zhaochch 18188 1月  20 10:43 AudioPolicyInterface.h
drwxrwxr-x  4 zhaochch zhaochch  4096 1月  20 10:43 common
drwxrwxr-x  2 zhaochch zhaochch  4096 1月  20 10:43 config
drwxrwxr-x  3 zhaochch zhaochch  4096 1月  20 10:43 engine
drwxrwxr-x  7 zhaochch zhaochch  4096 1月  20 10:43 engineconfigurable
drwxrwxr-x  4 zhaochch zhaochch  4096 1月  20 10:43 enginedefault
drwxrwxr-x  2 zhaochch zhaochch  4096 1月  20 10:43 manager
drwxrwxr-x  2 zhaochch zhaochch  4096 1月  25 10:05 managerdefault
drwxrwxr-x  2 zhaochch zhaochch  4096 1月  23 14:45 service
drwxrwxr-x  3 zhaochch zhaochch  4096 1月  20 10:43 utilities
```

对于 Qcom 平台会在 Hardware 部分重构 AudioPolicyManager 的实现，具体 code 位置：\hardware\qcom\audio\policy_hal



AudioFlinger 的介绍

1. AudioFlinger（AF）的启动

```
media/audioserver/audioserver.rc
service audioserver /system/bin/audioserver
    class main
    user audioserver
    # media gid needed for /dev/fm (radio) and for /data/misc/media (tee)
    group audio camera drmrpc inet media mediadrm net_bt net_bt_admin net_bw_acct qcom_diag
    ioprio rt 4
    writepid /dev/cpuset/foreground/tasks /dev/stune/foreground/tasks
```

```
LOCAL_SRC_FILES := \
    main_audioserver.cpp
LOCAL_SHARED_LIBRARIES := \
    libaudioflinger \
    libaudiopolicyservice \
    libbinder \
    libcutils \
    liblog \
    libmedia \
    libmedialogservice \
    libnbaio \
    libradioservice \
    libsoundtriggerservice \
    Libutils
LOCAL_MODULE := audioserver
LOCAL_INIT_RC := audioserver.rc
```

(1) AudioFlinger 是在在 AudioServer 中启动的，先分析 audioserver 是怎么启动的。首先分析 frameworks/av/media/audioserver 下面的 Android.mk:

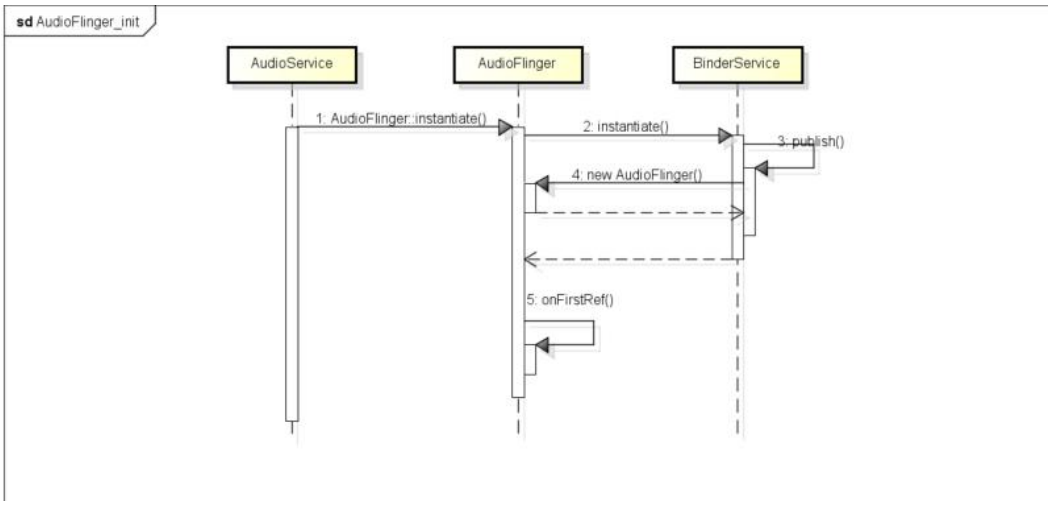
可以看到 audioserver 的入口类是在 main_audioserver.cpp 中，audioserver 的 rc 文件是 audioserver.rc

(2) 根据 audioserver.rc 可以知道 audioserver 是在开机 init 的过程中被 start 的。

(3) 根据 main_audioserver.cpp 中 main 函数可以看到在 audioserver 启动的时候同时调用了 AudioFlinger 和 AudioPolicy 的初始化函数去初始化这两个 server。

```
147:            sp<ProcessState> proc(ProcessState::self());
148:            sp<IServiceManager> sm = defaultServiceManager();
149:            ALOGI("ServiceManager: %p", sm.get());
150:            AudioFlinger::instantiate();
151:            AudioPolicyService::instantiate();
152:            RadioService::instantiate();
153: #ifdef DOLBY_ENABLE
154:            DolbyMemoryService::instantiate();
155: #endif
156:            SoundTriggerHwService::instantiate();
157:            ProcessState::self()->startThreadPool();
158:            IPCThreadState::self()->joinThreadPool();
```

2. AudioFlinger 的初始化



上图是 AudioFlinger 的初始化的流程图，下图是 AudioFlinger 的类。

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

```
template<typename SERVICE>
class BinderService
{
public:
    static status_t publish(bool allowIsolated = false) {
        sp<IServiceManager> sm(defaultServiceManager());
        return sm->addService(
                String16(SERVICE::getServiceName()),
                new SERVICE(), allowIsolated);
    }
```

对于 AudioFlinger，因为它继承了 BinderService，所有在调用 instantiate()时候实际上调用到了 publish，这里会把 AudioFlinger 添加到 servicemanager 中；而且 AudioFlinger 继承了 BnAudioFlinger，所以它是一个 Binder 通信的 Server 端，同时它在被第一次创建的时候会走到 onFirstRef()。（具体可以看 RefBase 接口的实现。）

AudioPolicyService 的介绍

1. AudioPolicyService（APS）的启动

   AudioPolicyService 的启动可以参考 AudioFlinger（AF）的启动部分。

```
sp<ProcessState> proc(ProcessState::self());
sp<IServiceManager> sm = defaultServiceManager();
ALOGI("ServiceManager: %p", sm.get());
AudioFlinger::instantiate();
AudioPolicyService::instantiate();
RadioService::instantiate();
```

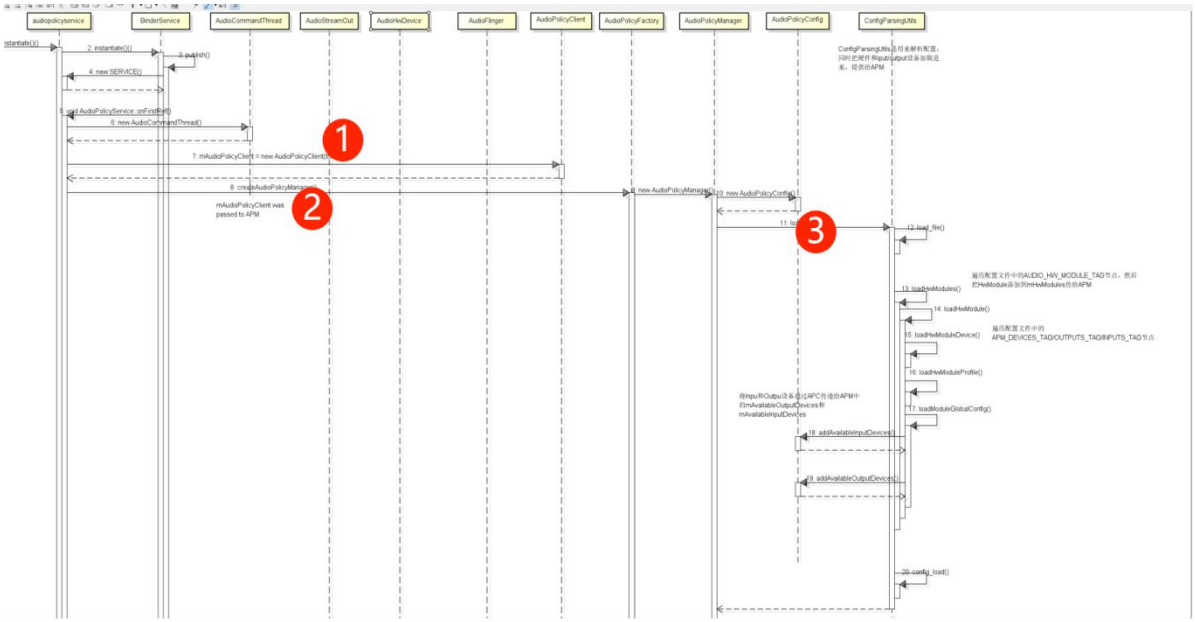
2. AudioPolicyService 的初始化

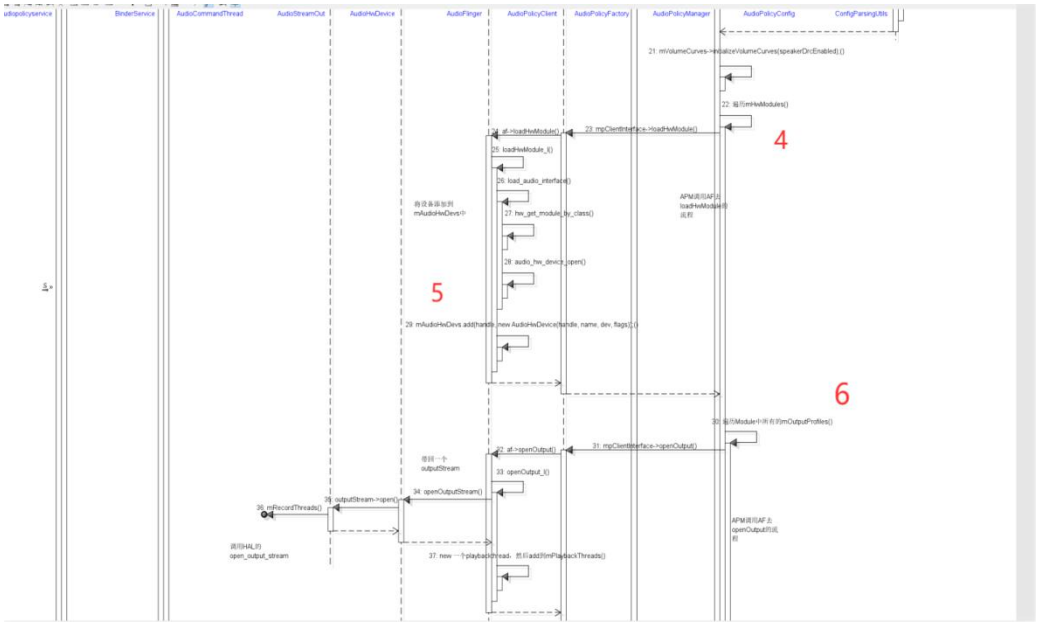
```
class AudioPolicyService :
    public BinderService<AudioPolicyService>,
    public BnAudioPolicyService,
    public IBinder::DeathRecipient
{
```

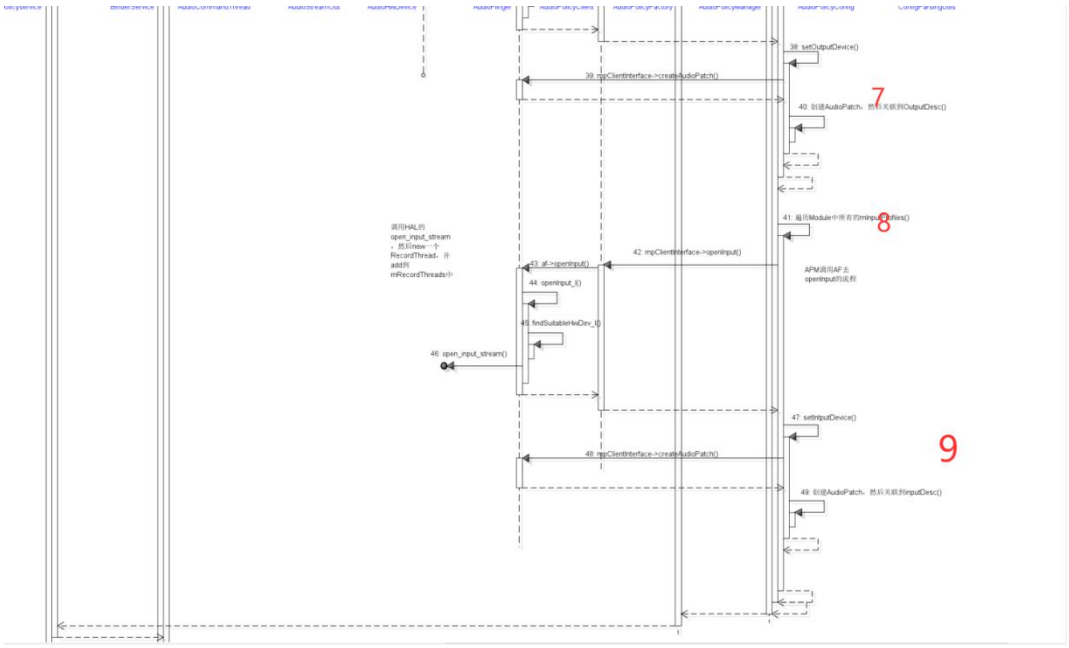
```
void AudioPolicyService::onFirstRef()
{
    {
        Mutex::Autolock _l(mLock);

        // start tone playback thread
        mTonePlaybackThread = new AudioCommandThread(String8("ApmTone"), this);
        // start audio commands thread
        mAudioCommandThread = new AudioCommandThread(String8("ApmAudio"), this);
        // start output activity command thread
        mOutputCommandThread = new AudioCommandThread(String8("ApmOutput"), this);
```

因为 AudioPolicyService 也是继承了 BinderService，所有在调用 instantiate()也会 new 一个 AudioPolicyService 对象并添加到 ServiceManager 中，同时会走到 OnFirstRef()中。下图是 AudioPolicyManager（APM）的初始化流程，涉及到 AudioPolciyManger，AudioFlinger，HAL 等模块，

其中几个关键的函数调用已经在流程图中标出来，后续会单独详细分析这几个函数调用。








```
mAudioPolicyClient = new AudioPolicyClient(this);
mAudioPolicyManager = createAudioPolicyManager(mAudioPolicyClient);
```

(1) 新建一个 AudioPolicyClient 对象，并作为参数传递给 APM，方便后续从 APM 回调到 APS；

(2) createAudioPolicyManager 通过 AudioPolicyFactory 去创建 APM，同时去真正的对 APS 做各种的 init。

```
#ifdef USE_XML_AUDIO_POLICY_CONF
    mVolumeCurves = new VolumeCurvesCollection();
    AudioPolicyConfig config(mHwModules, mAvailableOutputDevices, mAvailableInputDevices,
                             mDefaultOutputDevice, speakerDrcEnabled,
                             static_cast<VolumeCurvesCollection *>(mVolumeCurves));
    PolicySerializer serializer;
    if (serializer.deserialize(AUDIO_POLICY_XML_CONFIG_FILE, config) != NO_ERROR) {
#else
    mVolumeCurves = new StreamDescriptorCollection();
    AudioPolicyConfig config(mHwModules, mAvailableOutputDevices, mAvailableInputDevices,
                             mDefaultOutputDevice, speakerDrcEnabled);
    if ((ConfigParsingUtils::loadConfig(AUDIO_POLICY_VENDOR_CONFIG_FILE, config) != NO_ERROR) &&
            (ConfigParsingUtils::loadConfig(AUDIO_POLICY_CONFIG_FILE, config) != NO_ERROR)) {
#endif
        ALOGE("could not load audio policy configuration file, setting defaults");
        config.setDefault();
    }
```

(3) APM 根据 USE_XML_AUDIO_POLICY_CONF 是否被定义决定是通过 PolicySerializer 还是 ConfigParsingUtils 去 load audio 的 policy config。在 load 的过程中会解析 config 的文件，把所有 module 的 TAG 添加到 mHwModules 中，所有的 OutputDevice 的 TAG 添加到 mAvailableOutputDevices，所有的 InputDevice 的 TAG 添加到 mAvailableInputDevices。

```
for (size_t i = 0; i < mHwModules.size(); i++) {
    mHwModules[i]->mHandle = mpClientInterface->loadHwModule(mHwModules[i]->getName());
    if (mHwModules[i]->mHandle == 0) {
        ALOGW("could not open HW module %s", mHwModules[i]->getName());
        continue;
    }
```

(4) APM 遍历所有的 Module，然后调用 AF 去 load 这些 module；

```
audio_hw_device_t *dev;

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

mHardwareStatus = AUDIO_HW_INIT;
rc = dev->init_check(dev);
```

```
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 from %s (%s) handle %d",
      name, dev->common.module->name, dev->common.module->id, handle);
```

(5) 在 AF 中的 loadHwModule_l 中，AF 先是根据传入的 name 去 HAL 层 load 对应的 audio_hw_device_t *dev,然后和自动生成的 handle 成对加入到 mAudioHwDevs 中。

```
for (size_t j = 0; j < mHwModules[i]->mOutputProfiles.size(); j++)
{
    const sp<IOProfile> outProfile = mHwModules[i]->mOutputProfiles[j];
```

```
status_t status = mpClientInterface->openOutput(outProfile->getModuleHandle(),
                                                &output,
                                                &config,
                                                &outputDesc->mDevice,
                                                address,
                                                &outputDesc->mLatency,
                                                outputDesc->mFlags);
```

(6) 遍历每个 module 中的每一个 Outputfile，然后再通过 AF 去 open 对应的 output device；

```
status_t status = mpClientInterface->createAudioPatch(&patch,
                                                      &afPatchHandle,
                                                      delayMs);
ALOGV("setOutputDevice() createAudioPatch returned %d patchHandle %d"
        "num_sources %d num_sinks %d",
                              status, afPatchHandle, patch.num_sources, patch.num_sinks);
if (status == NO_ERROR) {
    if (index < 0) {
        patchDesc = new AudioPatch(&patch, mUidCached);
        addAudioPatch(patchDesc->mHandle, patchDesc);
    } else {
        patchDesc->mPatch = patch;
    }
    patchDesc->mAfPatchHandle = afPatchHandle;
    if (patchHandle) {
        *patchHandle = patchDesc->mHandle;
    }
    outputDesc->setPatchHandle(patchDesc->mHandle);
    nextAudioPortGeneration();
    mpClientInterface->onAudioPatchListUpdate();
}
```

(7) 通过 AF 创建对应的 patch（前端到硬件的路由）并设置到该 outputfile 中。

```
// open input streams needed to access attached devices to validate
// mAvailableInputDevices list
for (size_t j = 0; j < mHwModules[i]->mInputProfiles.size(); j++)
{
    const sp<IOProfile> inProfile = mHwModules[i]->mInputProfiles[j];

    if (!inProfile->hasSupportedDevices()) {
        ALOGW("Input profile contains no device on module %s", mHwModules[i]->getName());
        continue;
    }
```

```
status_t status = mpClientInterface->openInput(inProfile->getModuleHandle(),
                                               &input,
                                               &config,
                                               &inputDesc->mDevice,
                                               address,
                                               AUDIO_SOURCE_MIC,
                                               AUDIO_INPUT_FLAG_NONE);
```

(8) 遍历每个 module 中的每一个 Inputfile，然后再通过 AF 去 open 对应的 input device；

```
    if (status == NO_ERROR) {
        const DeviceVector &supportedDevices = inProfile->getSupportedDevices();
        for (size_t k = 0; k  < supportedDevices.size(); k++) {
            ssize_t index =  mAvailableInputDevices.indexOf(supportedDevices[k]);
            // give a valid ID to an attached device once confirmed it is reachable
            if (index >= 0) {
                sp<DeviceDescriptor> devDesc = mAvailableInputDevices[index];
                if (!devDesc->isAttached()) {
                    devDesc->attach(mHwModules[i]);
                    devDesc->importAudioPort(inProfile);
                }
            }
```

(9) 查找到 mAvailableInputDevices 中对应的 device，关联上对应的 module 和 inputfile。

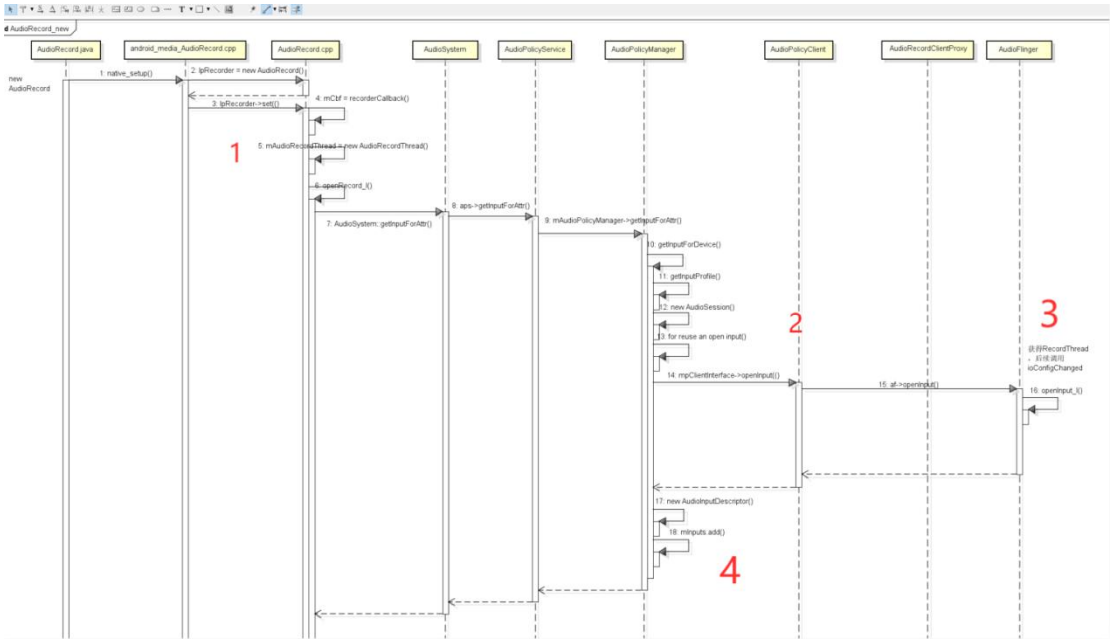
AudioRecord 的初始化流程

AudioRecord 再 Java 端的初始化实现：

```
mMinBufferSize = AudioRecord.getMinBufferSize( sampleRateInHz: 96000, AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_PCM_16BIT);
mAudioRecord = new AudioRecord(MediaRecorder.AudioSource.MIC, sampleRateInHz: 96000, AudioFormat.CHANNEL_IN_MONO, AudioFormat.ENCODING_PCM_16BIT, bufferSizeInBytes: mMinBufferSize * 2);
```

从对应的初始化 code 中可以看到，new 一个 AudioRecord 对象需要传入各种参数，包括采样率，信号源，最小 buffer 值等；

下图是 AudioRecord 对象初始化过程中涉及到各个模块和函数调用关系，再 AudioReocrd 初始化的过程中主要是获得 AF 模块创建的 Sharememory 的指针，以及获得 AudioRecord 再 AF 中的代理对象 RecordHandler 和 RecordTrack。




```
// create an uninitialized AudioRecord object
lpRecorder = new AudioRecord(String16(opPackageNameStr.c_str()));
```

```
const status_t status = lpRecorder->set(paa->source,
    sampleRateInHertz,
    format,        // word length, PCM
```

```
// create the IAudioRecord
status_t status = openRecord_l(0 /*epoch*/, mOpPackageName);
```

```
status = AudioSystem::getInputForAttr(&mAttributes, &input,
                                      mSessionId,
                                      // FIXME compare to AudioTrack
                                      mClientPid,
```

(1) 这里是从 Java 的 AudioRecord 调用到 JNI 层，在这里 new 一个 native 的 AudioRecord，然后调用他的 Set()函数，随后逐层调用到 APM 的 getInputForAttr()中。

```
    sp<AudioSession> audioSession = new AudioSession(session,
                                                     inputSource,
                                                     format,
                                                     samplingRate,
                                                     channelMask,
                                                     flags,
                                                     uid,
                                                     isSoundTrigger,
                                                     policyMix, mpClientInterface);

// TODO enable input reuse
#if 0
    // reuse an open input if possible
    for (size_t i = 0; i < mInputs.size(); i++) {
        sp<AudioInputDescriptor> desc = mInputs.valueAt(i);
        // reuse input if it shares the same profile and same sound trigger attribute
        if (profile == desc->mProfile &&
            isSoundTrigger == desc->isSoundTrigger()) {

            sp<AudioSession> as = desc->getAudioSession(session);
```

```
status_t status = mpClientInterface->openInput(profile->getModuleHandle(),
                                               &input,
                                               &config,
                                               &device,
                                               address,
                                               halInputSource,
                                               profileFlags);
```

(2) 在 APM 中的 getInputForDevice（）会先 new 一个 AudioSession，然后通过 AF 的代理 client 调用到 AF 的 openInput_l 函数，同时这里也有 APM 对同一个应用多路录音的支持，只是被 disable，如果需要单个 app 多路录音，可以在这里打开；

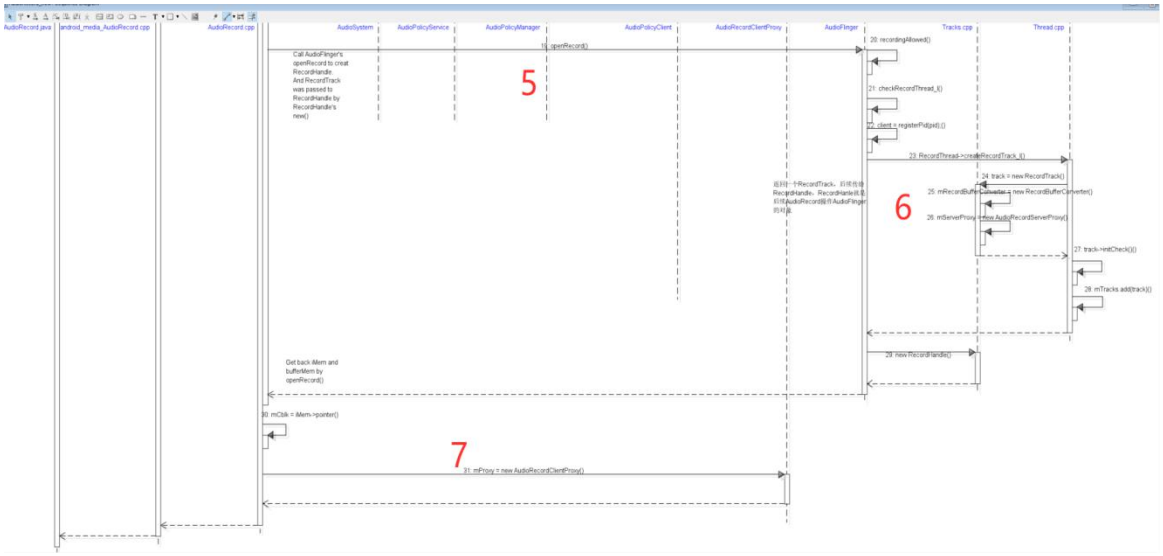
```
audio_config_t halconfig = *config;
audio_hw_device_t *inHwHal = inHwDev->hwDevice();
audio_stream_in_t *inStream = NULL;
status_t status = inHwHal->open_input_stream(inHwHal, *input, devices, &halconfig,
                                             &inStream, flags, address.string(), source);
```

```
        AudioStreamIn *inputStream = new AudioStreamIn(inHwDev, inStream, flags);

        // Start record thread
        // RecordThread requires both input and output device indication to forward to audio
        // pre processing modules
        sp<RecordThread> thread = new RecordThread(this,
                                  inputStream,
                                  *input,
                                  primaryOutputDevice_l(),
                                  devices,
                                  mSystemReady
#ifdef TEE_SINK
                                  , teeSink
#endif
                                  );
        mRecordThreads.add(*input, thread);
```

(3) 然后在 AF 中会调用 HAL 的接口 open 对应的 input device；最后会 new 一个 AudioStreamIn 的对象传递给新建的 RecordThread 对象，在把整个 RecordThread 对象 add 到 mRecordThreads 中；

```
sp<AudioInputDescriptor> inputDesc = new AudioInputDescriptor(profile);
inputDesc->mSamplingRate = profileSamplingRate;
inputDesc->mFormat = profileFormat;
inputDesc->mChannelMask = profileChannelMask;
inputDesc->mDevice = device;
inputDesc->mPolicyMix = policyMix;
inputDesc->addAudioSession(session, audioSession);

addInput(input, inputDesc);
mpClientInterface->onAudioPortListUpdate();
```

(4) 在 APM 中 getInputForAttr()的最后，会根据 profile 新建一个 AudioInputDescriptor 对象，包含了前面 new 出来的 AudioSession 对象。最后把这个 Descriptor 和前面的 input 按照键值对的方式 add 到 mInputs 中，为了后面使用；




```
sp<IMemory> iMem;            // for cblk
sp<IMemory> bufferMem;
sp<IAudioRecord> record = audioFlinger->openRecord(input,
                                                   mSampleRate,
                                                   mFormat,
                                                   mChannelMask,
                                                   opPackageName,
                                                   &temp,
                                                   &flags,
                                                   mClientPid,
                                                   tid,
                                                   mClientUid,
                                                   &mSessionId,
                                                   &notificationFrames,
                                                   iMem,
                                                   bufferMem,
                                                   &status);
```

(5) 对于 Native AudioRecord 来说，在 set 函数中通过调用 AF 的 openRecord 函数创建一个 IAudioRecord 对象，这个对象代理对象也是后续访问 RecordHandler 的 proxy（基于 Binder 机制）。

```
Mutex::Autolock _l(mLock);
RecordThread *thread = checkRecordThread_l(input);
```

```
recordTrack = thread->createRecordTrack_l(client, sampleRate, format, channelMask,
                                          frameCount, lSessionId, notificationFrames,
                                          clientUid, flags, tid, &lStatus);
```

```
track = new RecordTrack(this, client, sampleRate,
              format, channelMask, frameCount, NULL, sessionId, uid,
              *flags, TrackBase::TYPE_DEFAULT);

lStatus = track->initCheck();
if (lStatus != NO_ERROR) {
    ALOGE("createRecordTrack_l() initCheck failed %d; no control block?", lStatus);
    // track must be cleared from the caller as the caller has the AF lock
    goto Exit;
}
mTracks.add(track);
```

```
cblk = recordTrack->getCblk();
buffers = recordTrack->getBuffers();

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

(6) 在 AF 的 openRecord 函数中，首先获得前面创建的 RecordThread，然后利用这个 Thread 去 Create RecordTrack，同时把这个 track 添加到 mTracks 中。最后 AF 利用返回的 RecordTrack 获得生成的 cblk 和 buffer 指针，在利用 recordTrack 生成一个 RecordHandle 对象传递给 Native AudioRecord.

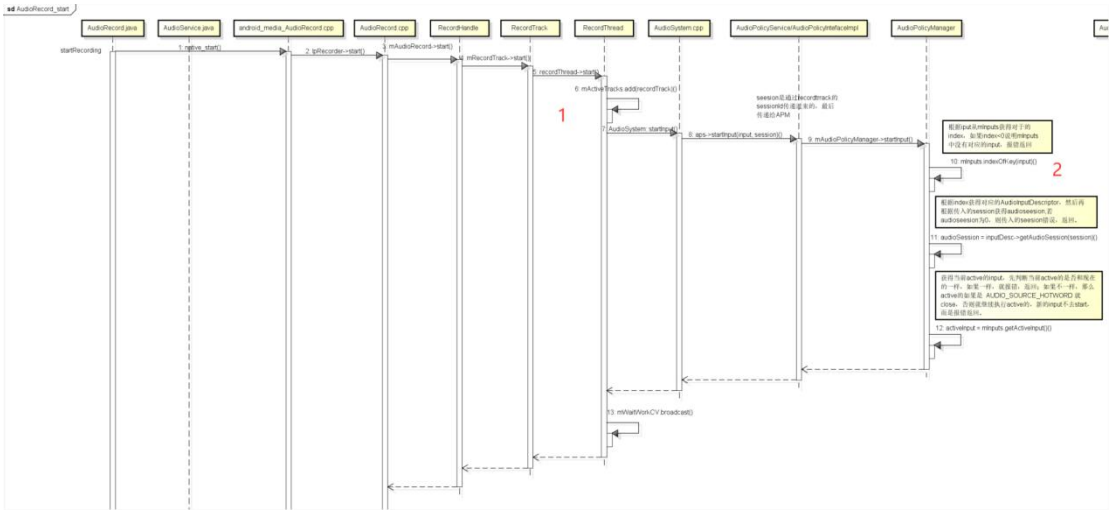
```
mAudioRecord = record;
mCblkMemory = iMem;
mBufferMemory = bufferMem;
IPCThreadState::self()->flushCommands();

mCblk = cblk;
```

```
// update proxy
mProxy = new AudioRecordClientProxy(cblk, buffers, mFrameCount, mFrameSize);
mProxy->setEpoch(epoch);
mProxy->setMinimum(mNotificationFramesAct);

mDeathNotifier = new DeathNotifier(this);
IInterface::asBinder(mAudioRecord)->linkToDeath(mDeathNotifier, this);
```

(7) 在 Native AudioRecord 利用 AF 创建一个 RecordHandle 的 proxy 后，会从 AF 带回 Sharememory 相关的指针 mCblkMemory/mBufferMemory/mCblk,同时利用这些指针 new 出 AudioRecordClientProxy 的对象 mProxy，这个 mProxy 就是后续 Native AudioRecord 从 ShareMemory 中读取数据的代理类。

AudioRecord 的 Start 流程

下面是 AudioRecord Start 的相关流程图



```
recordTrack->mState = TrackBase::STARTING_1;
mActiveTracks.add(recordTrack);
mActiveTracksGen++;
status_t status = NO_ERROR;
if (recordTrack->isExternalTrack()) {
    mLock.unlock();
    status = AudioSystem::startInput(mId, recordTrack->sessionId());
    mLock.lock();
```

```
// Catch up with current buffer indices if thread is already running.
// This is what makes a new client discard all buffered data.  If the track's mRsmpInFront
// was initialized to some value closer to the thread's mRsmpInFront, then the track could
// see previously buffered data before it called start(), but with greater risk of overrun.

recordTrack->mResamplerBufferProvider->reset();
// clear any converter state as new data will be discontinuous
recordTrack->mRecordBufferConverter->reset();
recordTrack->mState = TrackBase::STARTING_2;
// signal thread to start
mWaitWorkCV.broadcast();
```

(1) 从 JAVA 的 AudioRecord start 一直到 Native 的 AudioRecord start，然后通过前面获得 RecordHandler 的 proxy 去 start RecordHander，再调用到 AudioTrack 和 AudioThread 的 start 中；最后再 AudioThread 中把当前的 track 添加到 mActiveTracks 中，再通过 AudioSystem 调用到 AFM 的 startInput 函数；当 APM 那边 start 完成后，AudioThread 会发送 mWaitWorkCV 的 singal 通知 RecordThread 可以去从 HAL 那边读取数据到 ShareMemory 中。

```
ALOGV("startInput() input %d", input);
ssize_t index = mInputs.indexOfKey(input);
if (index < 0) {
    ALOGW("startInput() unknown input %d", input);
    return BAD_VALUE;
}
sp<AudioInputDescriptor> inputDesc = mInputs.valueAt(index);

sp<AudioSession> audioSession = inputDesc->getAudioSession(session);
```

```
// indicate active capture to sound trigger service if starting capture from a mic on
// primary HW module
audio_devices_t device = getNewInputDevice(input);
audio_devices_t primaryInputDevices = availablePrimaryInputDevices();
if (((device & primaryInputDevices & ~AUDIO_DEVICE_BIT_IN) != 0) &&
        mInputs.activeInputsCountOnDevices(primaryInputDevices) == 0) {
    SoundTrigger::setCaptureState(true);
}
setInputDevice(input, device, true /* force */);
```

```
// for a non-virtual input device, check if there is another (non-virtual) active input
audio_io_handle_t activeInput = mInputs.getActiveInput();
if (activeInput != 0 && activeInput != input) {

    // If the already active input uses AUDIO_SOURCE_HOTWORD then it is closed,
    // otherwise the active input continues and the new input cannot be started.
    sp<AudioInputDescriptor> activeDesc = mInputs.valueFor(activeInput);
    if ((activeDesc->inputSource() == AUDIO_SOURCE_HOTWORD) &&
            !activeDesc->hasPreemptedSession(session)) {
        ALOGW("startInput(%d) preempting low-priority input %d", input, activeInput);
        //FIXME: consider all active sessions
        AudioSessionCollection activeSessions = activeDesc->getActiveAudioSessions();
        audio_session_t activeSession = activeSessions.keyAt(0);
        SortedVector<audio_session_t> sessions =
                                    activeDesc->getPreemptedSessions();
        sessions.add(activeSession);
        inputDesc->setPreemptedSessions(sessions);
        stopInput(activeInput, activeSession);
        releaseInput(activeInput, activeSession);
    } else {
        ALOGE("startInput(%d) failed: other input %d already started", input, activeInput);
        return INVALID_OPERATION;
    }
} « end if activeInput!=0&&activ... »
```

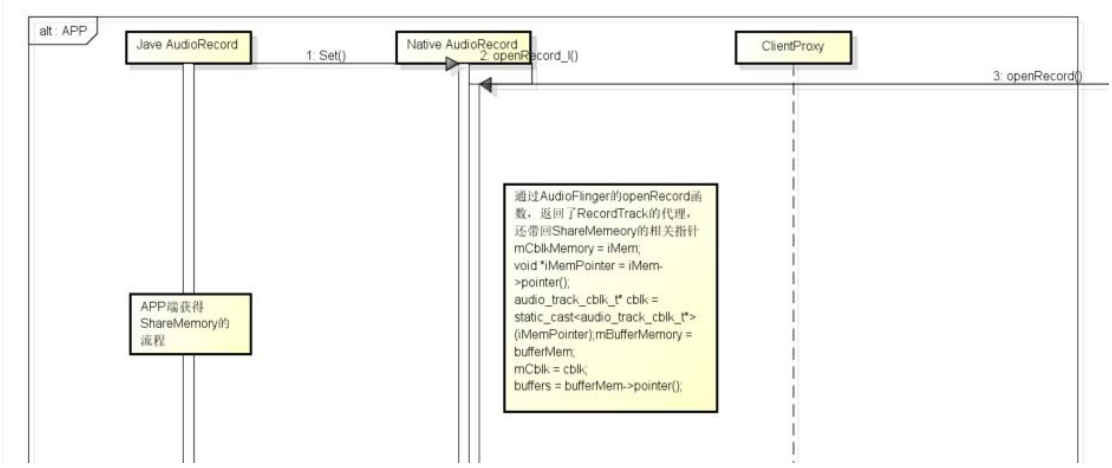
(2) 在 APM 这边，首先通过 input 获得当前的 inputDesc 和 audiosession，然后判断现在需要 start 的 Desc 是否和当前 active 的一致，如果不一致就需要考虑多应用同时录音的情况，这里会根据各种情况做出 stop 当前的，还是返回错误。

(3) 最后通过 getNewInputDevice(input)和 setInputDevice(input, device, true /* force */); 两个函数通过 AF 去 start HAL 中的 device。

AudioReocrd 数据流的分析

对于整个 AudioRecord 在 new 到 start，最后到 read 的过程中涉及到 APP 进程和 AudioFlinger 进程两个不同的进程，而且这两个进程间进行数据的传输对于 Audio 来说采用的是 ShareMemory 的机制。下面主要介绍 AudioRecord 的 ShareMemory 在各个过程中是如何生成和传递的。

1. APP 端在 AudioRecord new 的过程中的流程：

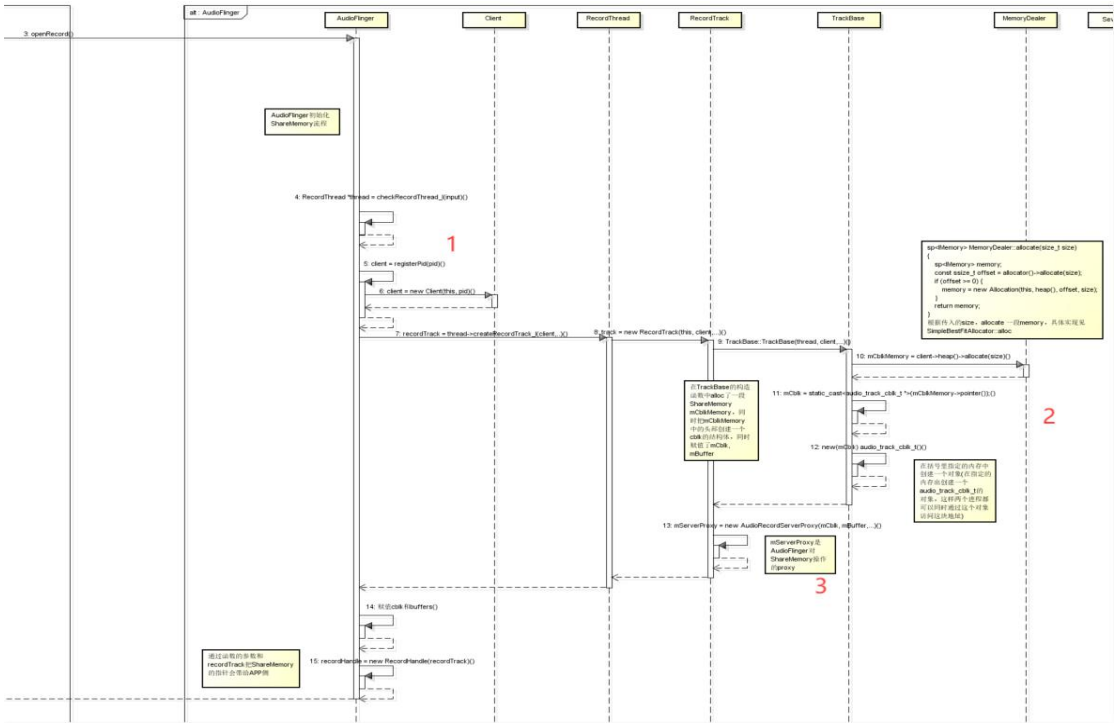



```
audio_session_t originalSessionId = mSessionId;

sp<IMemory> iMem;           // for cblk
sp<IMemory> bufferMem;
sp<IAudioRecord> record = audioFlinger->openRecord(input,
                                                   mSampleRate,
                                                   mFormat,
                                                   mChannelMask,
                                                   opPackageName,
                                                   &temp,
                                                   &flags,
                                                   mClientPid,
                                                   tid,
                                                   mClientUid,
                                                   &mSessionId,
                                                   &notificationFrames,
                                                   iMem,
                                                   bufferMem,
                                                   &status);
```

```
mAudioRecord = record;
mCblkMemory = iMem;
mBufferMemory = bufferMem;
IPCThreadState::self()->flushCommands();
```

APP 端通过调用 AudioFlinger 的 openRecord 函数在 AudioFlinger 的进程创建了对应的 ShareMemory，同时也把该 ShareMemory 的指针通过参数传递的方式带了回来。

2. AudioFlinger 端在 new 的过程的流程：




```
RecordThread *thread = checkRecordThread_l(input);
if (thread == NULL) { ... }

client = registerPid(pid);

if (sessionId != NULL && *sessionId != AUDIO_SESSION_ALLOCATE) { ... } else { ... }
ALOGV("openRecord() lSessionId: %d input %d", lSessionId, input);

recordTrack = thread->createRecordTrack_l(client, sampleRate, format, channelMask,
                                          frameCount, lSessionId, notificationFrames,
                                          clientUid, flags, tid, &lStatus);
```

```
track = new RecordTrack(this, client, sampleRate,
        format, channelMask, frameCount, NULL, sessionId, uid,
        *flags, TrackBase::TYPE_DEFAULT);

lStatus = track->initCheck();
```

```
// RecordTrack constructor must be called with AudioFlinger::mLock and ThreadBase::mLock held
AudioFlinger::RecordThread::RecordTrack::RecordTrack(
            RecordThread *thread,
            const sp<Client>& client,
            uint32_t sampleRate,
            audio_format_t format,
            audio_channel_mask_t channelMask,
            size_t frameCount,
            void *buffer,
            audio_session_t sessionId,
            int uid,
            audio_input_flags_t flags,
            track_type type)
    :   TrackBase(thread, client, sampleRate, format,
                  channelMask, frameCount, buffer, sessionId, uid, false /*isOut*/,
                  (type == TYPE_DEFAULT) ?
                          ((flags & AUDIO_INPUT_FLAG_FAST) ? ALLOC_PIPE : ALLOC_CBLK) :
                          ((buffer == NULL) ? ALLOC_LOCAL : ALLOC_NONE),
                  type),
        mOverflow(false),
        mFramesToDrop(0),
        mResamplerBufferProvider(NULL), // initialize in case of early constructor exit
        mRecordBufferConverter(NULL),
        mFlags(flags)
{
    if (mCblk == NULL) { ... }

    mRecordBufferConverter = new RecordBufferConverter(
            thread->mChannelMask, thread->mFormat, thread->mSampleRate,
            channelMask, format, sampleRate);
    // ...
    if (mRecordBufferConverter->initCheck() != NO_ERROR) { ... }

    mServerProxy = new AudioRecordServerProxy(mCblk, mBuffer, frameCount,
            mFrameSize, !isExternalTrack());

    mResamplerBufferProvider = new ResamplerBufferProvider(this);
```

```
mCblkMemory = client->heap()->allocate(size);
if (mCblkMemory == 0 ||
        (mCblk = static_cast<audio_track_cblk_t *>(mCblkMemory->pointer())) == NULL) {
```

```
sp<MemoryDealer> AudioFlinger::Client::heap() const
{
    return mMemoryDealer;
}
```

```
sp<IMemory> MemoryDealer::allocate(size_t size)
{
    sp<IMemory> memory;
    const ssize_t offset = allocator()->allocate(size);
    if (offset >= 0) {
        memory = new Allocation(this, heap(), offset, size);
    }
    return memory;
}
```

```
// construct the shared structure in-place.
if (mCblk != NULL) {
    new(mCblk) audio_track_cblk_t();
```

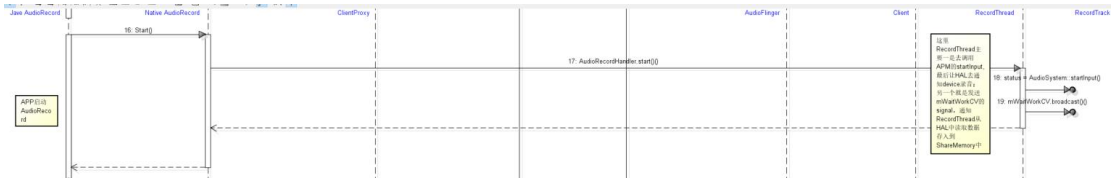
```
ssize_t SimpleBestFitAllocator::alloc(size_t size, uint32_t flags)
{
    if (size == 0) {
        return 0;
    }
    size = (size + kMemoryAlign-1) / kMemoryAlign;
    chunk_t* free_chunk = 0;
    chunk_t* cur = mList.head();

    size_t pagesize = getpagesize();
    while (cur) {
```

(1) 跟 pid 新建一个 Client 的对象，并传递给 RecordTrack,为了后面真正去 allocate memory；

(2) 通过 Client 调用 MemoryDealer 真正的去 allocate 一段 sharememory，并且在 memory 的头部 new 出一个 cblk 结构的对象；

(3) 根据 allocated 出来的 memory 新建一个 AudioRecordServerProxy 的对象，这个类和 RecordBufferConverter，以及 ResamplerBufferProvider 是后面把 HAL 中的数据 copy 到 sharememory 中的关键类。

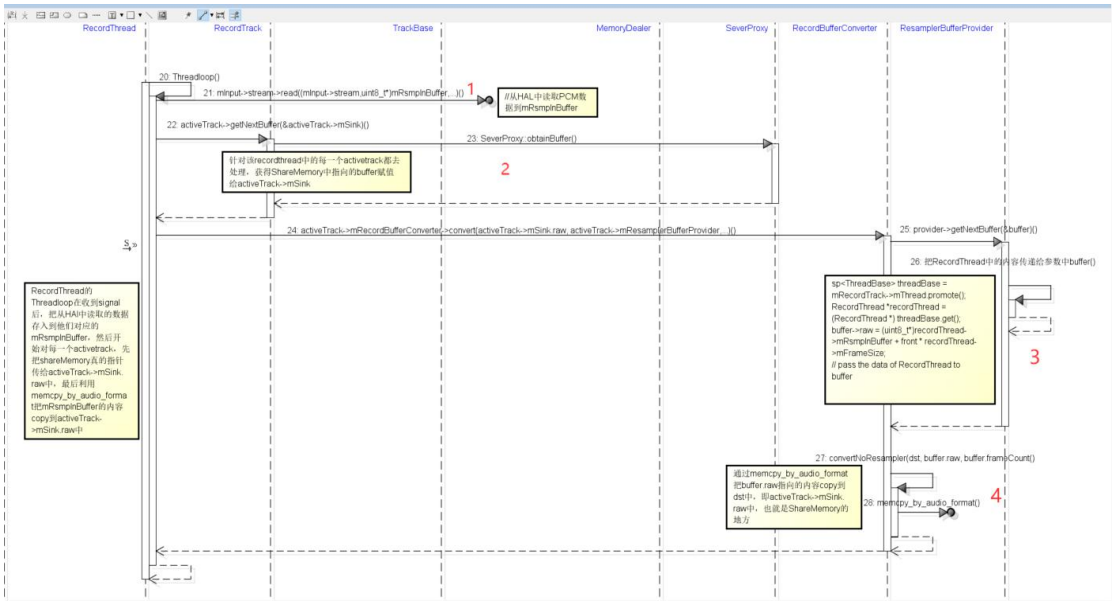
3. APP 端和 AudioFlinger 端的 Start 过程：



```
    if (recordTrack->isExternalTrack()) {
        mLock.unlock();
        status = AudioSystem::startInput(mId, recordTrack->sessionId());
        mLock.lock();
        // FIXME should verify that recordTrack is still in mActiveTracks
        if (status != NO_ERROR) { ... }
    }
    // Catch up with current buffer indices if thread is already running.
    // This is what makes a new client discard all buffered data.  If the track's mRsmpInFront
    // was initialized to some value closer to the thread's mRsmpInFront, then the track could
    // see previously buffered data before it called start(), but with greater risk of overrun.

    recordTrack->mResamplerBufferProvider->reset();
    // clear any converter state as new data will be discontinuous
    recordTrack->mRecordBufferConverter->reset();
    recordTrack->mState = TrackBase::STARTING_2;
    // signal thread to start
    mWaitWorkCV.broadcast();
```

这个部分主要是通过发送 mWaitWorkCV 的 signal 来通知 RecordThread 去读取数据。

4. AudioFlinger 端读取 HAL 数据的过程：




```
// If an NBAIO source is present, use it to read the normal capture's data
if (mPipeSource != 0) {
    size_t framesToRead = mBufferSize / mFrameSize;
    framesRead = mPipeSource->read((uint8_t*)mRsmpInBuffer + rear * mFrameSize,
            framesToRead);
    if (framesRead == 0) {
        // since pipe is non-blocking, simulate blocking input
        sleepUs = (framesToRead * 1000000LL) / mSampleRate;
    }
// otherwise use the HAL / AudioStreamIn directly
} else {
    ATRACE_BEGIN("read");
    ssize_t bytesRead = mInput->stream->read(mInput->stream,
            (uint8_t*)mRsmpInBuffer + rear * mFrameSize, mBufferSize);
    ATRACE_END();
    if (bytesRead < 0) { ... } else { ... }
}
```

```
activeTrack->mSink.frameCount = 0;
status_t status = activeTrack->getNextBuffer(&activeTrack->mSink);
size_t framesOut = activeTrack->mSink.frameCount;
```

```
ServerProxy::Buffer buf;
buf.mFrameCount = buffer->frameCount;
status_t status = mServerProxy->obtainBuffer(&buf);
buffer->frameCount = buf.mFrameCount;
buffer->raw = buf.mRaw;
```

```
    // is assignment redundant in some cases?
    buffer->mFrameCount = part1;
    buffer->mRaw = part1 > 0 ?
            &((char *) mBuffers)[(mIsOut ? front : rear) * mFrameSize] : NULL;
    buffer->mNonContig = availToServer - part1;
```

```
size_t AudioFlinger::RecordThread::RecordBufferConverter::convert(void *dst,
        AudioBufferProvider *provider, size_t frames)
{
    if (mInputConverterProvider != NULL) {
        mInputConverterProvider->setBufferProvider(provider);
        provider = mInputConverterProvider;
    }

    if (mResampler == NULL) {
        ALOGVV("NO RESAMPLING sampleRate:%u mSrcFormat:%#x mDstFormat:%#x",
                mSrcSampleRate, mSrcFormat, mDstFormat);

        AudioBufferProvider::Buffer buffer;
        for (size_t i = frames; i > 0; ) {
            buffer.frameCount = i;
            status_t status = provider->getNextBuffer(&buffer);
            if (status != OK || buffer.frameCount == 0) {
                frames -= i; // cannot fill request.
                break;
            }
            // format convert to destination buffer
            convertNoResampler(dst, buffer.raw, buffer.frameCount);

            dst = (int8_t*)dst + buffer.frameCount * mDstFrameSize;
            i -= buffer.frameCount;
            provider->releaseBuffer(&buffer);
        }
```

```
    buffer->raw = (uint8_t*)recordThread->mRsmpInBuffer + front * recordThread->mFrameSize;
    buffer->frameCount = part1;
    mRsmpInUnrel = part1;
    return NO_ERROR;
```

```
void AudioFlinger::RecordThread::RecordBufferConverter::convertNoResampler(
        void *dst, const void *src, size_t frames)
{
    // src is native type unless there is legacy upmix or downmix, whereupon it is float.
    if (mBufFrameSize != 0 && mBufFrames < frames) {
        free(mBuf);
        mBufFrames = frames;
        (void)posix_memalign(&mBuf, 32, mBufFrames * mBufFrameSize);
    }
    // do we need to do legacy upmix and downmix?
    if (mIsLegacyUpmix || mIsLegacyDownmix) { ... }
    // do we need to do channel mask conversion?
    if (mSrcChannelMask != mDstChannelMask) { ... }
    // convert to destination buffer
    const void *convertBuf = mBuf != NULL ? mBuf : src;
    memcpy_by_audio_format(dst, mDstFormat, convertBuf, mSrcFormat,
            frames * mDstChannelCount);
}
```

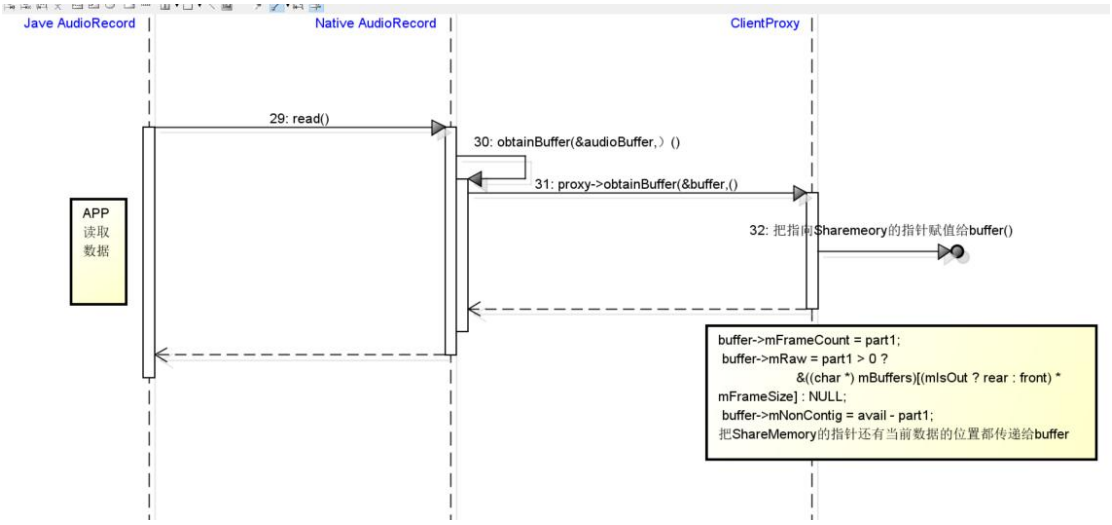
(1) 通过 HAL 的 read 接口，把 HAL 中的数据读到 mRsmpInBuffer 缓存中；

(2) 针对该 recordthread 中的每一个 activetrack 都去处理，获得 ShareMemory 中指向的 buffer 赋值给 activeTrack->mSink；

(3) 利用 mResamplerBufferProvider 把 RecordThread 中的 mRsmpInBuffer 指向的数据地址赋值给给到一个 AudioBufferProvider 的 buffer；

(4) 在 convertNoResampler 函数中利用 memcpy_by_audio_format 将 buffer 中的数据 copy 到 activeTrack->mSink 指向的地址中（即 sharemeory 的地址）。

5. APP 端读取数据的过程：



```
ssize_t AudioRecord::read(void* buffer, size_t userSize, bool blocking)
{
    if (mTransfer != TRANSFER_SYNC) {
        return INVALID_OPERATION;
    }

    if (ssize_t(userSize) < 0 || (buffer == NULL && userSize != 0)) {
        // sanity-check. user is most-likely passing an error code, and it would
        // make the return value ambiguous (actualSize vs error).
        ALOGE("AudioRecord::read(buffer=%p, size=%zu (%zu)", buffer, userSize, userSize);
        return BAD_VALUE;
    }

    ssize_t read = 0;
    Buffer audioBuffer;

    while (userSize >= mFrameSize) {
        audioBuffer.frameCount = userSize / mFrameSize;

        status_t err = obtainBuffer(&audioBuffer,
                blocking ? &ClientProxy::kForever : &ClientProxy::kNonBlocking);
        if (err < 0) {
            if (read > 0) {
                break;
            }
            if (err == TIMED_OUT || err == -EINTR) {
                err = WOULD_BLOCK;
            }
            return ssize_t(err);
        }

        size_t bytesRead = audioBuffer.size;
        memcpy(buffer, audioBuffer.i8, bytesRead);
        buffer = ((char *) buffer) + bytesRead;
        userSize -= bytesRead;
```

```
    buffer.mFrameCount = audioBuffer->frameCount;
    // FIXME starts the requested timeout and elapsed over from scratch
    status = proxy->obtainBuffer(&buffer, requested, elapsed);

} while ((statu
audioBuffer->fr
audioBuffer->si
audioBuffer->raw = buffer.mRaw;
```

```
    buffer->mFrameCount = part1;
    buffer->mRaw = part1 > 0 ?
            &((char *) mBuffers)[(mIsOut ? rear : front) * mFrameSize] : NULL;
    buffer->mNonContig = avail - part1;
    mUnreleased = part1;
    status = NO_ERROR;
    break;
```

该过程中，Native AudioRecord 先是利用 obtainBuffer 函数从 ClientProxy 中获得 ShareMemory 的地址，然后用 memcpy 把该地址的数据 copy 到传入参数对应的 buffer 中。

自此 AudioRecord 所有的数据通路都已经完成。