密　　级：

文档编号：

版 本 号：V1.0.0

Android Audio 策略系统研究

北京京联云软件有限公司

编制：姜睿	生效日期：
审核：	批准：

修订历史

修订日期	修订版本	修订章节	修订内容	修订人
2013-05-23	V1.0.0	全部	初始编写	姜睿

注：版本历史中修改记录是指该版本正式成稿并下发后，又经过修改完善，再次下发时填写所作修改的内容。

目录

1. 概述

Android 的 Audio 策略系统是 Android 中一个复杂的模块，其范围涉及到框架层中 AudioService 和 AudioFlinger 两个服务程序以及 Audio 抽象层(HAL)。其功能主要包括：

1. 设置声音的输出设备

系统中有多种不同类型的声音，如来电铃声，短信提示音，闹钟，音乐等等。同时系统中有多种声音输出设备，如扬声器，有线耳机，听筒，蓝牙 SCO 设备，蓝牙 A2DP 设备等等。控制各种声音从不同设备中输出是 Audio 策略系统解决的重要问题。

2. 管理各种声音的音量

系统中不同类型声音的音量是单独控制的。如调整铃声的音量不会影响音乐的音量。策略系统负责调整，保存和管理所有类型声音的音量。

3. 声音冲突控制

如果系统中有多种声音需要同时播放出来，应该播放哪种声音。如媒体播放器播放音乐的过程中，来了一条短信，需要播放短信提示音。Audio 系统应该如何处理。

4. 设置和管理铃声模式

铃声模式包括普通，静音和震动三种模式。铃声模式的改变会影响其他类型声音的静音设置。

本文将在后续章节重点阐述问题 1，2，3。

2. 概念

Audio 策略系统管理各种声音设备，从硬件中抽取了许多逻辑概念。了解这些概念是理解策略系统的前提。

2.1. 硬件模块

Audio 抽象层将声音硬件设备抽象成一组硬件模块。一个硬件模块对应一

个声音硬件，基本的声音硬件有 Primary, A2DP, USB 等。框架层可通过不同硬件模块的接口访问不同的声音硬件。程序中可以看到如下类，多是从不同的方面描述硬件模块的，看代码时可以认为它们是硬件模块的同义词。

1. HwModule
2. Audio_HwDevice
3. audio_module_handle_t
4. audio_hw_device

2.2. 输入／输出

一个硬件模块包括一组声音输出和声音输入。如主硬件模块(Primary)，包括一个组输出（primary, hdmi）和一组输入（primary）。有的硬件模块仅仅包括一组输入，如 A2DP, USB，都没有输入设备。程序中可以看到如下类，多是从不同的方面描述硬件模块的，看代码时可以认为它们是输入/输出的同义词。

1. IOProfile
2. audio_io_handle_t
3. AudioOutputDescriptor
4. AudioStreamOut
5. AudioInputDescriptor
6. AudioStreamIn

2.3. 设备

一个输入或输出中包括一组设备。如主模块的主输出中包括了扬声器，耳机，耳麦，蓝牙 SCO 等等输出设备。主模块的主输入中包括了内置麦克风，蓝牙 SCO，耳麦等等输入设备。

2.4. 硬件配置

硬件配置是系统对于硬件模块，输入输出已经设备的全局描述。Android

默认的配置文件位于源代码目录中：

hardware/libhardware_legacy/audio/audio_policy.conf

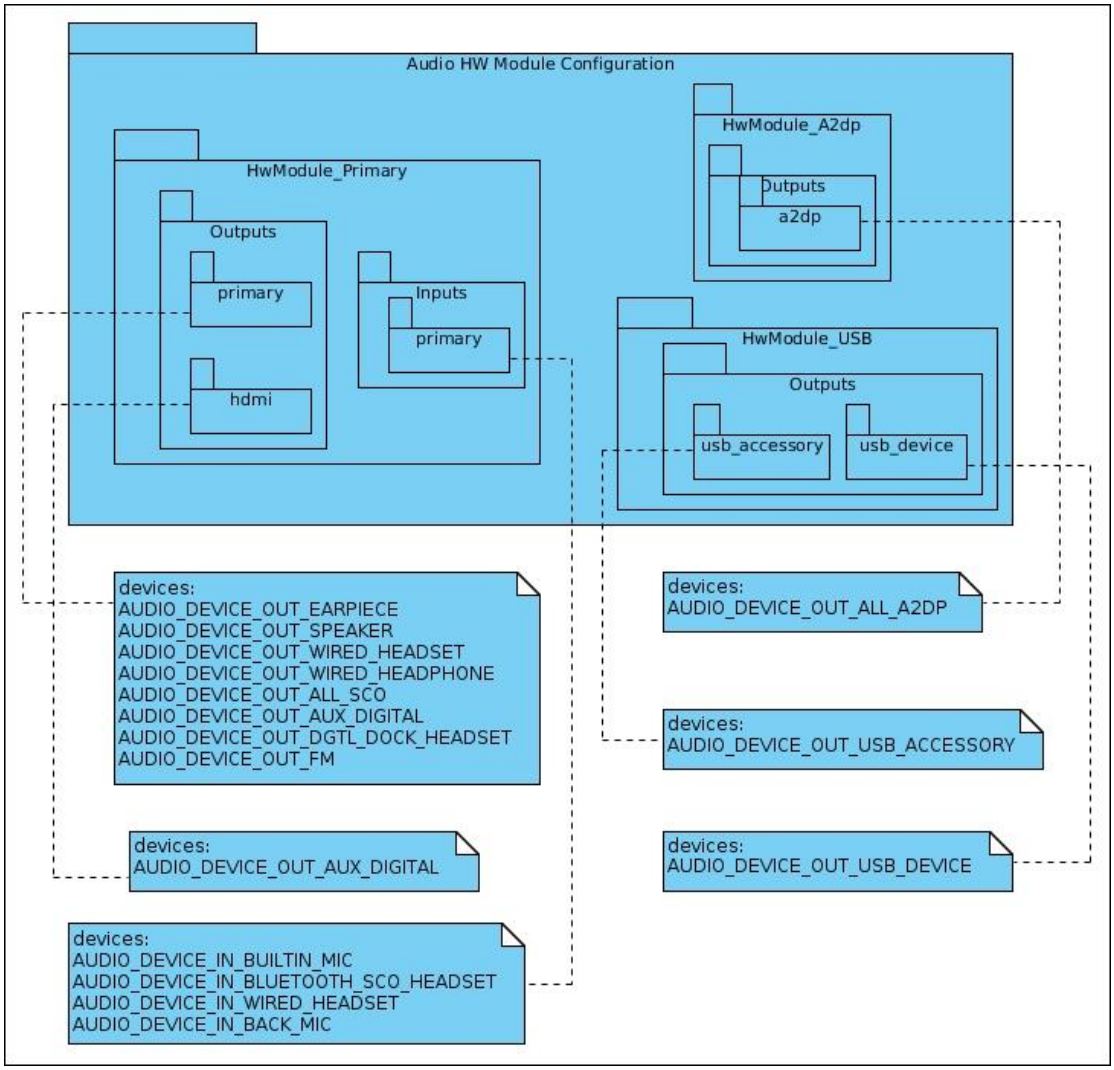
各个 vendor 的文件如下：

device/samsung/manta/audio_policy.conf

device/samsung/crespo/libaudio/audio_policy.conf

device/samsung/tuna/audio/audio_policy.conf

下图是 Samsung manta 系统中声音硬件的组成示意图：



从图中可以看到manta系统包括了三个硬件模块，primary, a2dp, usb。硬件模块 primary 包括了一组输出：primary 和 hdmi。输出 primary 包括了一组输出设备：AUDIO_DEVICE_OUT_SPEAKER（扬声器），AUDIO_DEVICE_OUT_WIRED_HEADSET（有线耳机+麦克），AUDIO_DEVICE_OUT_WIRED_HEADPHONE（有线耳机），

AUDIO_DEVICE_OUT_ALL_SCO（所有蓝牙 SCO 设备），AUDIO_DEVICE_OUT_AUX_DIGITAL, AUDIO_DEVICE_OUT_DGTL_DOCK_HEADSET。输出 hdmi 包括一个 AUDIO_DEVICE_OUT_AUX_DIGITAL 设备。

硬件模块 primary 包括了一个输入：primary。输入 primary 包括了一组输入设备：AUDIO_DEVICE_IN_BUILTIN_MIC（内置麦克），AUDIO_DEVICE_IN_BLUETOOTH_SCO_HEADSET（蓝牙 SCO 麦克），AUDIO_DEVICE_IN_WIRED_HEADSET（有线耳机麦克）。

硬件模块 a2dp 包括了一个输出 a2dp。输出 a2dp 包括个设备 AUDIO_DEVICE_OUT_ALL_A2DP（所有 a2dp 设备）。

硬件模块 usb 包括了一组输出 usb_accessory 和 usb_device。输出 usb_accessory 包括了设备 AUDIO_DEVICE_OUT_USB_ACCESSORY。输出 usb_device 包括了设备 AUDIO_DEVICE_OUT_USB_DEVICE。

对输出的配置包括采样率，声道数，支持声音格式，设备，标识位。硬件模块 primary 的 primary 输出配置如下：

```
primary {
  sampling_rates 44100
  channel_masks AUDIO_CHANNEL_OUT_STEREO
  formats AUDIO_FORMAT_PCM_16_BIT
  devices
AUDIO_DEVICE_OUT_SPEAKER|AUDIO_DEVICE_OUT_WIRED_HEADSET|AUDIO_DEVICE_OUT_WIRED_HEADPHONE|AUDIO_DEVICE_OUT_ALL_SCO|AUDIO_DEVICE_OUT_AUX_DIGITAL|AUDIO_DEVICE_OUT_DGTL_DOCK_HEADSET
  flags AUDIO_OUTPUT_FLAG_PRIMARY
}
```

对输入的配置包括采样率，声道数，支持声音格式，设备，标识位。硬件模块 primary 的 primary 输如配置如下：

```
primary {
  sampling_rates 8000|11025|16000|22050|32000|44100|48000
```

channel_masks

AUDIO_CHANNEL_IN_MONO|AUDIO_CHANNEL_IN_STEREO

formats AUDIO_FORMAT_PCM_16_BIT

devices

AUDIO_DEVICE_IN_BUILTIN_MIC|AUDIO_DEVICE_IN_BLUETOOTH_SCO_HEADSET|AUDIO_DEVICE_IN_WIRED_HEADSET

}

同时配置文件指定了系统的默认打开的输入输出设备。Audio 策略系统初始化时会打开默认设备的输入/输出，并将输入/输出的设备切换到默认设备。

global_configuration {

attached_output_devices AUDIO_DEVICE_OUT_SPEAKER

default_output_device AUDIO_DEVICE_OUT_SPEAKER

attached_input_devices AUDIO_DEVICE_IN_BUILTIN_MIC

}

2.5. Stream

Audio 系统中定义了各种声音的类型，每个类型称为一种 stream。如播放铃音时，声音类型为 STREM_RING，播放音乐时，类型为 STREAM_MUSIC，播放闹钟时，类型为 STREAM_ALARM。不同的 stream，系统选择输出设备的策略是不一样的。

Audio 系统选择声音通道的时候，会根据当前系统中连接的设备和播放的声音的 stream 类型（确切的说应该是 strategy 类型）来决定输出设备。例如，用户插入耳机到系统，系统中可用的设备就包括了耳机，扬声器和听筒。如果用户打开媒体播放器播放音乐，这时应用会设置声音类型为 STREAM_MUSIC。Audio 策略系统为 STREAM_MUSIC 设置的策略是优先选择A2DP设备，然后选择耳机，再后选择USB输出设备，最后是扬声器。这时由于耳机连接上了，策略系统就会通过耳机播放音乐。如果用户停止音乐后，闹钟响起了。Audio 策略系统为 STREAM_ALARM 设置的策略是首先选择扬声器为第一个输出设备，然后选择第二个输出设备，优先级顺序是 A2DP

设备，有线耳机，USB 输出设备。

由于耳机已经连接到了系统，所以闹钟声音同时通过扬声器和耳机播放出来。

所有声音类型定义如下：

```
    /* These values must be kept in sync with AudioSystem.h */
    /*
     * If these are modified, please also update Settings.System.VOLUME_SETTINGS
     * and attrs.xml and AudioManager.java.
     */
    /* The audio stream for phone calls */
    public static final int STREAM_VOICE_CALL = 0;
    /* The audio stream for system sounds */
    public static final int STREAM_SYSTEM = 1;
    /* The audio stream for the phone ring and message alerts */
    public static final int STREAM_RING = 2;
    /* The audio stream for music playback */
    public static final int STREAM_MUSIC = 3;
    /* The audio stream for alarms */
    public static final int STREAM_ALARM = 4;
    /* The audio stream for notifications */
    public static final int STREAM_NOTIFICATION = 5;
    /* @hide The audio stream for phone calls when connected on bluetooth */
    public static final int STREAM_BLUETOOTH_SCO = 6;
    /* @hide The audio stream for enforced system sounds in certain countries (e.g camera in Japan) */
    public static final int STREAM_SYSTEM_ENFORCED = 7;
    /* @hide The audio stream for DTMF tones */
    public static final int STREAM_DTMF = 8;
    /* @hide The audio stream for text to speech (TTS) */
    public static final int STREAM_TTS = 9;
```

2.6. Routing Strategy

从 AudioSystem.h 中声音 stream 类型看，系统中共有 10 中 stream 类型。这 10 种类型中有一些设备选择策略是相同的。如果为每一种 stream 类型都写一套选择策略，就显得比较冗余。于是 Audio 系统抽象出 Routing Strategy 的概念。一个 Routing Strategy 包括了多个 stream 类型，同一个 Routing Strategy 使用相同的设备选择策略。函数

AudioPolicyManagerBase::getStrategy 用于将 stream 转化 strategy。从该函数发现，strategy phone 包括 voice call, bluetooth sco 两种 stream，也就是说 voice call 和 bluetooth sco 选择设备的策略相同。strategy media 包括 system, tts 和 music 三种 stream。strategy sonification 包括 ring, alarm 种 stream，等等。

Strategy 的定义和选择函数列举如下：

```
        enum routing_strategy {
            STRATEGY_MEDIA,
            STRATEGY_PHONE,
            STRATEGY_SONIFICATION,
            STRATEGY_SONIFICATION_RESPECTFUL,
            STRATEGY_DTMF,
            STRATEGY_ENFORCED_AUDIBLE,
            NUM_STRATEGIES
        };

AudioPolicyManagerBase::routing_strategy AudioPolicyManagerBase::getStrategy(
        AudioSystem::stream_type stream) {
    // stream to strategy mapping
    switch (stream) {
    case AudioSystem::VOICE_CALL:
    case AudioSystem::BLUETOOTH_SCO:
        return STRATEGY_PHONE;
    case AudioSystem::RING:
    case AudioSystem::ALARM:
        return STRATEGY_SONIFICATION;
    case AudioSystem::NOTIFICATION:
        return STRATEGY_SONIFICATION_RESPECTFUL;
    case AudioSystem::DTMF:
        return STRATEGY_DTMF;
    default:
        ALOGE("unknown stream type");
    case AudioSystem::SYSTEM:
        // NOTE: SYSTEM stream uses MEDIA strategy because muting music and
switching outputs
        // while key clicks are played produces a poor result
    case AudioSystem::TTS:
    case AudioSystem::MUSIC:
        return STRATEGY_MEDIA;
    case AudioSystem::ENFORCED_AUDIBLE:
        return STRATEGY_ENFORCED_AUDIBLE;
```

```
    }
}
```

3. Audio 系统全局结构

3.1. 框架层

框架层提供 AudioManager 接口和 AudioSystem 接口给应用程序,用于应用程序调节音量，设置静音，设置声音模式，设置振动，设置声音输入输出设备，播放音效等等。框架层主要时通过 AudioSystem 的 JNI 调用 native 层的 AudioSystem 来实现底层硬件相关功能。总体上来说，AudioManager 和 AudioSystem 仅仅提供接口功能，很少有功能逻辑的实现。

AudioService 主要的作用如下：

1. 提供系统音量调节，静音相关 UI，以及相关设置的保存和读取。
2. 提供铃音模式相关 UI 以及相关设置的保存和读取。
3. 提供 Audio Focus 机制，用于系统中应用程序解决声音冲突。也就是概述中描述的第二个问题（如果系统中有多种声音需要同时播放出来，应该播放哪种声音）。音量调节，静音的功能是通过调用 AudioSystem 到 Native 层，在 AudioFlinger 中实现的。

3.2. 原生层

原生层主要包括 libmedia 和 libaudioflinger 这两个动态库。libmeida 中的 AudioSystem 主要提供接口，调用 AudioPoliceService（简称 APS）和 AudioFlinger（简称 AFL）中的方法来实现具体功能。AudioSystem 几乎没有提供任何具体功能的实现。

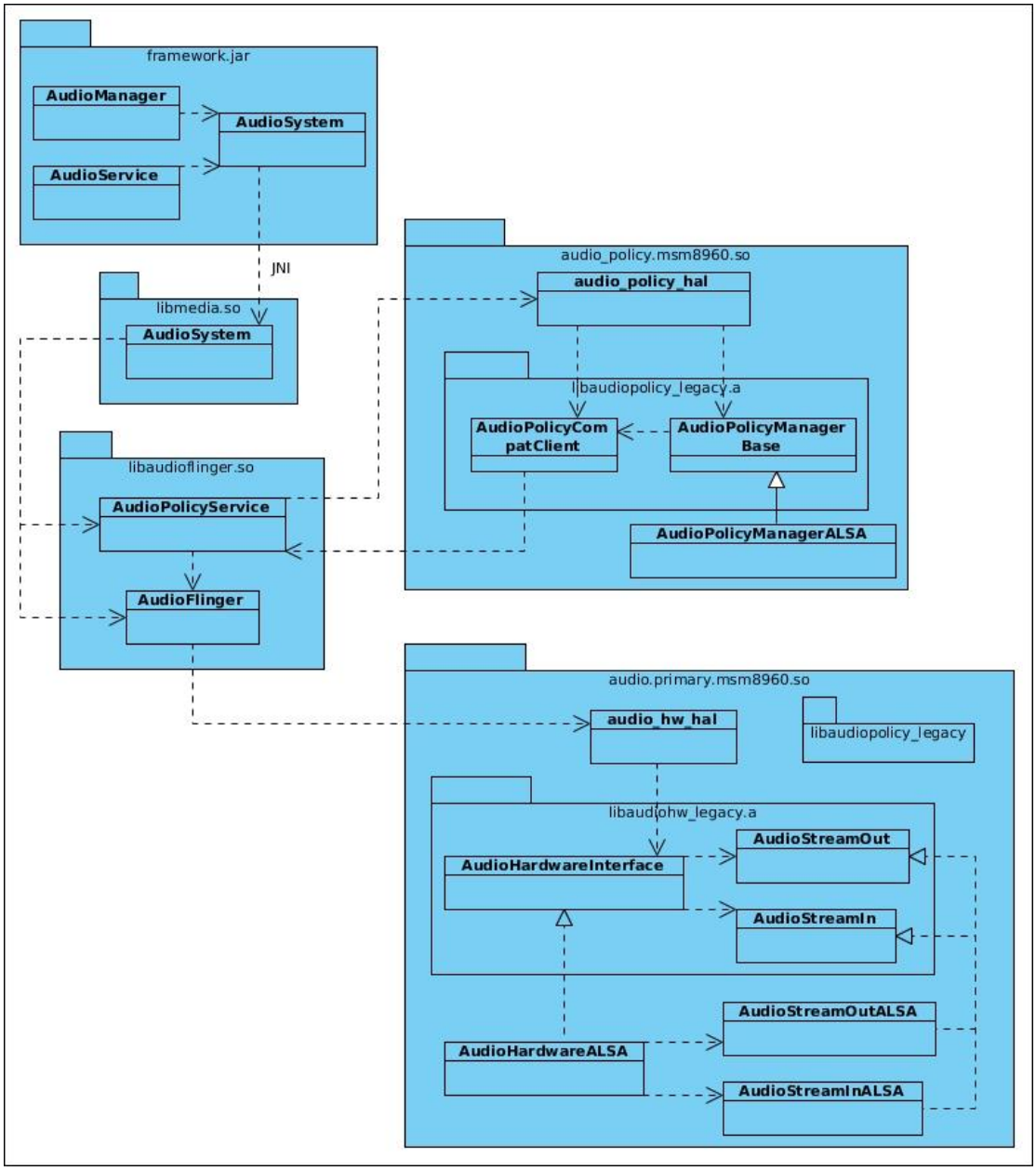
Libaudioflinger 包括了 Audio 系统中两个重要的服务程序：AudioPolicyService 和 AudioFlinger。AudioPolicyService 提供声音策略的接口，包括声音通道的选择，设置声音模式（normal, ringtone, incall），音量控制，设置硬件参数等等功能。概述中第一个问题涉及到的接口就是由 APS 提供的。APS 仅仅提供接口，具体的策略实现几乎都是由抽象层中

AudioPolicyManagerBase 实现的。具体的硬件相关功能是由 AudioFlinger 实现的。AudioFlinger 主要功能是接受上层模块如 AudioPolicyService 的调用，并调用 Audio 抽象层的接口实现具体的硬件操作。

3.3. 抽象层

Audio 抽象层封装了和硬件相关的接口和特性。硬件厂商提供自己的动态库来实现抽象层。抽象层包括了两类动态库，策略库和硬件接口库。 策略库一般命名如下，audio_policy.vendor.so。策略库是实现和硬件相关的声音策略。如下图中 AudioPolicyManagerALSA 扩展了 AudioPolicyManagerBase（简称 APMB）。不过从代码看 AudioPolicyManagerALSA 几乎没有修改 APMB，99%的策略功能都是 APMB 完成的。硬件接口库一般命名如下：audio.primary.vendor.so。实现了硬件相关的具体操作。如下图中 AudioHardwareALAS, AudioStreamOutALSA, AudioStreamInALSA。AudioHardwareALAS 提供硬件全局控制功能，并负责创建 AudioStreamOutALSA 和 AudioStreamInALSA。AudioStreamOutALSA 实现输出相关功能。AudioStreamInALSA 实现输入相关功能。

Audio 系统的全局结构图如下：



3.4 重要接口

Audio 策略系统中功能的具体实现几乎都在 AudioPolicyManagerBase 和 AudioFlinger 中。下面将介绍其中的一些重要的接口。

3.3.1. 硬件类接口

抽象层提供给 AudioFlinger 的硬件相关接口主要包括三个类 AudioHardwareInterface，AudioStreamOut，AudioStreamIn，位于

如下文件中：

hardware/libhardware_legacy/include/hardware_legacy/AudioHardwareInterface.h

硬件类接口包括了，设置电话音量，设置设备主音量，设置设备模式（普通，铃声，电话中），设置麦克静音，设置声音系统参数，创建输入/输出，设置输入/输出的采样率，声道数，声音格式等等接口。上述三个类的声明如下，接口的含义在其注释中已经清晰给出。

```
class AudioHardwareInterface
{
public:
    virtual ~AudioHardwareInterface() {}

    /**
     * check to see if the audio hardware interface has been initialized.
     * return status based on values defined in include/utils/Errors.h
     */
    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.
     * Range between 0.0 and 1.0. If any value other than NO_ERROR is returned,
     * the software mixer will emulate this capability.
     */
    virtual status_t      setMasterVolume(float volume) = 0;

    /**
     * Get the current master volume value for the HAL, if the HAL supports
     * master volume control.  AudioFlinger will query this value from the
     * primary audio HAL when the service starts and use the value for setting
     * the initial master volume across all HALs.
     */
    virtual status_t      getMasterVolume(float *volume) = 0;

    /**
     * setMode is called when the audio mode changes. NORMAL mode is for
     * standard audio playback, RINGTONE when a ringtone is playing, and IN_CALL
     * when a call is in progress.
     */
```

```
    virtual status_t      setMode(int mode) = 0;

    // mic mute
    virtual status_t      setMicMute(bool state) = 0;
    virtual status_t      getMicMute(bool* state) = 0;

    // set/get global audio parameters
    virtual status_t      setParameters(const String8& keyValuePairs) = 0;
    virtual String8       getParameters(const String8& keys) = 0;

    // Returns audio input buffer size according to parameters passed or 0 if one of the
    // parameters is not supported
    virtual size_t      getInputBufferSize(uint32_t sampleRate, int format, int channelCount)
= 0;

    /** This method creates and opens the audio hardware output stream */
    virtual AudioStreamOut* openOutputStream(
                                uint32_t devices,
                                int *format=0,
                                uint32_t *channels=0,
                                uint32_t *sampleRate=0,
                                status_t *status=0) = 0;
    virtual     void          closeOutputStream(AudioStreamOut* out) = 0;

    /** This method creates and opens the audio hardware input stream */
    virtual AudioStreamIn* openInputStream(
                                uint32_t devices,
                                int *format,
                                uint32_t *channels,
                                uint32_t *sampleRate,
                                status_t *status,
                                AudioSystem::audio_in_acoustics acoustics) = 0;
    virtual     void          closeInputStream(AudioStreamIn* in) = 0;

    /**This method dumps the state of the audio hardware */
    virtual status_t dumpState(int fd, const Vector<String16>& args) = 0;

    static AudioHardwareInterface* create();
};
/**
 * AudioStreamOut is the abstraction interface for the audio output hardware.
 *
 * It provides information about various properties of the audio output hardware driver.
 */
```

```
class AudioStreamOut {
public:
    virtual                 ~AudioStreamOut() = 0;

    /** return audio sampling rate in hz - eg. 44100 */
    virtual uint32_t      sampleRate() const = 0;

    /** returns size of output buffer - eg. 4800 */
    virtual size_t        bufferSize() const = 0;

    /**
     * returns the output channel mask
     */
    virtual uint32_t      channels() const = 0;

    /**
     * return audio format in 8bit or 16bit PCM format -
     * eg. AudioSystem:PCM_16_BIT
     */
    virtual int           format() const = 0;

    /**
     * return the frame size (number of bytes per sample).
     */
    uint32_t      frameSize() const { return
popcount(channels())*((format()==AUDIO_FORMAT_PCM_16_BIT)?sizeof(int16_t):sizeof(i
nt8_t)); }

    /**
     * return the audio hardware driver latency in milli seconds.
     */
    virtual uint32_t      latency() const = 0;

    /**
     * Use this method in situations where audio mixing is done in the
     * hardware. This method serves as a direct interface with hardware,
     * allowing you to directly set the volume as apposed to via the framework.
     * This method might produce multiple PCM outputs or hardware accelerated
     * codecs, such as MP3 or AAC.
     */
    virtual status_t      setVolume(float left, float right) = 0;

    /** write audio buffer to driver. Returns number of bytes written */
    virtual ssize_t       write(const void* buffer, size_t bytes) = 0;
```

```
    /**
     * Put the audio hardware output into standby mode. Returns
     * status based on include/utils/Errors.h
     */
    virtual status_t      standby() = 0;

    /** dump the state of the audio output device */
    virtual status_t dump(int fd, const Vector<String16>& args) = 0;

    // set/get audio output parameters. The function accepts a list of parameters
    // key value pairs in the form: key1=value1;key2=value2;...
    // Some keys are reserved for standard parameters (See AudioParameter class).
    // If the implementation does not accept a parameter change while the output is
    // active but the parameter is acceptable otherwise, it must return
INVALID_OPERATION.
    // The audio flinger will put the output in standby and then change the parameter value.
    virtual status_t      setParameters(const String8& keyValuePairs) = 0;
    virtual String8       getParameters(const String8& keys) = 0;

    // return the number of audio frames written by the audio dsp to DAC since
    // the output has exited standby
    virtual status_t      getRenderPosition(uint32_t *dspFrames) = 0;

    /**
     * get the local time at which the next write to the audio driver will be
     * presented
     */
    virtual status_t      getNextWriteTimestamp(int64_t *timestamp);

};
/**
 * AudioStreamIn is the abstraction interface for the audio input hardware.
 *
 * It defines the various properties of the audio hardware input driver.
 */
class AudioStreamIn {
public:
    virtual                 ~AudioStreamIn() = 0;

    /** return audio sampling rate in hz - eg. 44100 */
    virtual uint32_t      sampleRate() const = 0;

    /** return the input buffer size allowed by audio driver */
```

```
    virtual size_t         bufferSize() const = 0;

    /** return input channel mask */
    virtual uint32_t       channels() const = 0;

    /**
     * return audio format in 8bit or 16bit PCM format -
     * eg. AudioSystem:PCM_16_BIT
     */
    virtual int            format() const = 0;

    /**
     * return the frame size (number of bytes per sample).
     */
    uint32_t       frameSize() const { return
AudioSystem::popCount(channels())*((format()==AudioSystem::PCM_16_BIT)?sizeof(int1
6_t):sizeof(int8_t)); }

    /** set the input gain for the audio driver. This method is for
     *   for future use */
    virtual status_t       setGain(float gain) = 0;

    /** read audio buffer in from audio driver */
    virtual ssize_t        read(void* buffer, ssize_t bytes) = 0;

    /** dump the state of the audio input device */
    virtual status_t dump(int fd, const Vector<String16>& args) = 0;

    /**
     * Put the audio hardware input into standby mode. Returns
     * status based on include/utils/Errors.h
     */
    virtual status_t       standby() = 0;

    // set/get audio input parameters. The function accepts a list of parameters
    // key value pairs in the form: key1=value1;key2=value2;...
    // Some keys are reserved for standard parameters (See AudioParameter class).
    // If the implementation does not accept a parameter change while the output is
    // active but the parameter is acceptable otherwise, it must return
INVALID_OPERATION.
    // The audio flinger will put the input in standby and then change the parameter value.
    virtual status_t       setParameters(const String8& keyValuePairs) = 0;
    virtual String8        getParameters(const String8& keys) = 0;
```

```
    // Return the number of input frames lost in the audio driver since the last call of this function.
    // Audio driver is expected to reset the value to 0 and restart counting upon returning the current value by this function call.
    // Such loss typically occurs when the user space process is blocked longer than the capacity of audio driver buffers.
    // Unit: the number of input audio frames
    virtual unsigned int  getInputFramesLost() const = 0;

    virtual status_t addAudioEffect(effect_handle_t effect) = 0;
    virtual status_t removeAudioEffect(effect_handle_t effect) = 0;
};
```

3.3.2. 策略类接口

策略类接口主要包括 AudioPolicyInterface。AudioPolicyManagerBase 实现了这个接口。该类实现了声音设备设备切换和音量控制的功能。

该类的主要接口如下：

1. setDeviceConnectionState

当有外部输出设备连接到系统时，该接口会被调用到。APMB 会将这个设备加入到可用输出设备变量中。比如用户将有线耳机插入手机，这时可用输出设备就包括了耳机，扬声器和听筒。当用户播放音乐时，APMB 就会根据为音乐设置的设备选择策略选择耳机作为输出设备。

```
virtual status_t setDeviceConnectionState(audio_devices_t device,
                                          AudioSystem::device_connection_state state,
                                          const char *device_address) = 0;
```

2. setPhoneState

设置声音系统的模式，normal, in call 和 ringtone。

```
virtual void setPhoneState(int state) = 0;
```

3. setForceUse

强制设置声音的通道。如打电话时，按照默认的策略，声音会从听筒输出。当用户使用免提时，系统会调用 setForceUse 函数，将声音通道强制切换

到扬声器。

virtual void setForceUse(AudioSystem::force_use usage, AudioSystem::forced_config config) = 0;

4. setStreamVolumeIndex

设置 stream 的音量。系统中每种 stream 的音量都是单独控制的。

virtual status_t setStreamVolumeIndex(AudioSystem::stream_type stream, int index,audio_devices_t device) = 0;

4. Audio 系统的初始化

Audio 系统主要包括 AudioFlinger 和 AudioPolicyService 两个服务。这两个服务在下面函数中初始化：

```
frameworks/av/media/mediaserver/main_mediaserver.cpp
int main(int argc, char** argv)
{
    ...
        AudioFlinger::instantiate();
    ...
        AudioPolicyService::instantiate();
    ...
}
```

4.1. AudioFlinger 初始化

AudioFlinger 的构造函数中的初始化比较简单，仅仅设置了一些内部状态。虽然对于硬件系统的初始化也是通过 AudioFlinger 完成的，但是这个过程是由 AudioPolicyService 发起的，将在后面章节描述。

```
AudioFlinger::AudioFlinger()
: BnAudioFlinger(),
    mPrimaryHardwareDev(NULL),
    mHardwareStatus(AUDIO_HW_IDLE),
    mMasterVolume(1.0f),
    mMasterMute(false),
    mNextUniqueId(1),
    mMode(AUDIO_MODE_INVALID),
```

```
    mBtNrecIsOff(false)
{
    ALOGV("AudioFlinger()");
}
```

4.2. AudioPolicyService 初始化

Android 的硬件接口一般都是封装在 HAL 层中。每个 HAL 会编译成一个动态库 so 文件。Audio 系统包括两种类型的 HAL, Policy HAL 和 Hardware HAL。Policy HAL 实现声音的控制策略，Android 提供一个默认实现 audio_policy.default.so。厂商也可以自己对默认实现的扩展，提供一个 audio_policy.vendor_name.so。Audio 系统会优先加载厂商的动态库。Hardware HAL 实现声音的硬件操作，Android 4.2 上一般会有如下几个动态库：audio.primary.default.so，audio.a2dp.default.so，audio.usb.default.so。每个动态库提供一个硬件模块的操作接口。

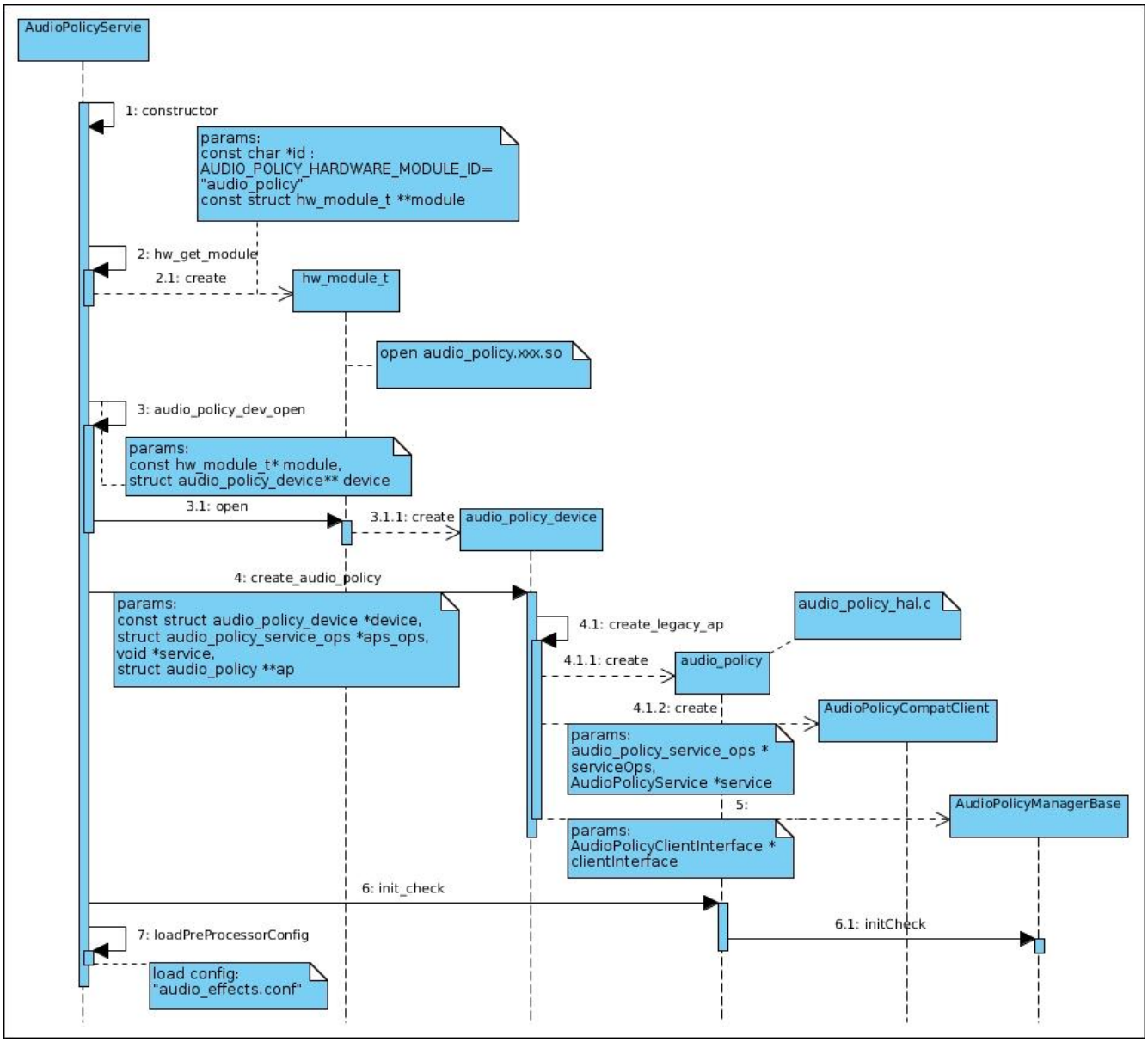
APS 初始化的主要工作就是加载这两类抽象层动态库。APS 首先加载 Policy HAL， 在 Policy HAL 中加载声音系统配置文件。然后通过配置文件加载 Hardware HAL。

4.2.1. 加载 Audio Policy 抽象层

```
AudioPolicyService::AudioPolicyService()
: BnAudioPolicyService() , mpAudioPolicyDev(NULL) , mpAudioPolicy(NULL)
{
    //  加载下面几个 so 文件之一
    // /system/lib/hw/audio_policy.default.so,
    // /vendor/lib/hw/audio_policy.vendor_name(like sc8825).so,
    // /system/lib/hw/audio_policy.vendor_name(like sc8825).so,
    const struct hw_module_t *module;
    rc = hw_get_module(AUDIO_POLICY_HARDWARE_MODULE_ID, &module);
    // 打开 audio_policy.default.so 动态库的 open 函数
    // 并注册了一个 create_audio_policy 的钩子函数到 APS
    rc = audio_policy_dev_open(module, &mpAudioPolicyDev);

    // 调用 create_audio_policy 的钩子函数
    // 为硬件相关的 audio policy 钩子函数赋值
    rc = mpAudioPolicyDev->create_audio_policy(mpAudioPolicyDev,
            &aps_ops, this, &mpAudioPolicy);
```

加载 Policy HAL 的大体流程如下面序列图：



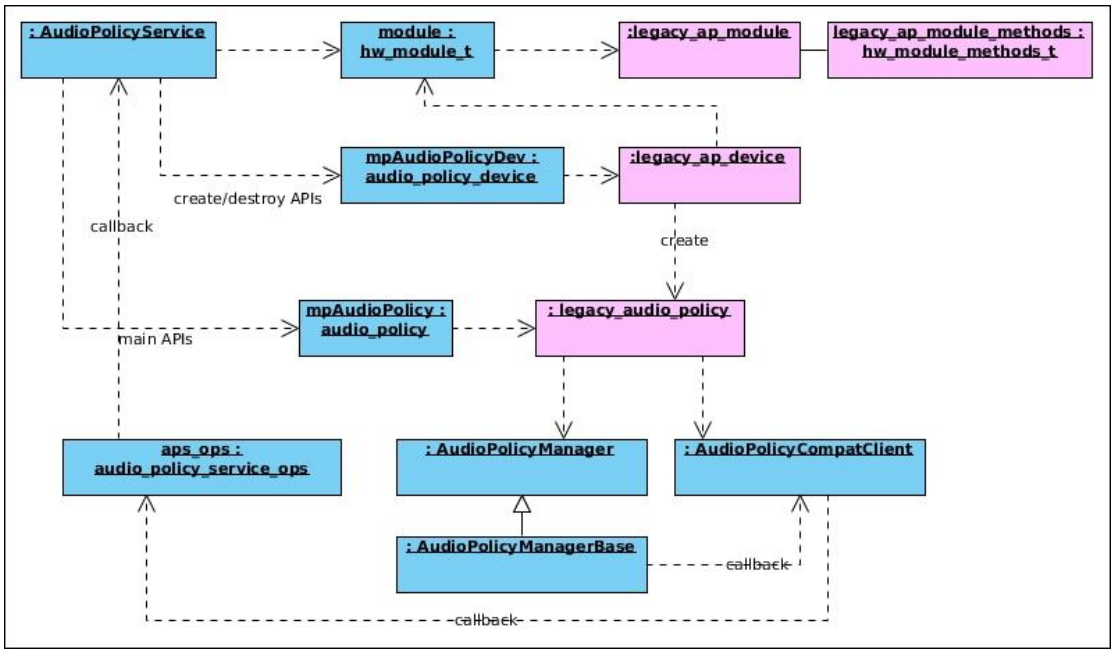
1. 通过 Android 的 HAL 通用机制找到 Audio Policy 的动态库。例如：/system/lib/hw/audio_policy.default.so

2. 通过 dlopen，并调用动态库的 open 方法创建 audio_policy_device

3. 通过 audio_policy_device 的 create 方法创建 audio_policy，AudioPolicyManager，AudioPolicyCompactClient。audio_policy 提供 APS 访问 Policy 抽象层的所有接口。

4. 将 AudioPolicyCompactClient 连接到 aps_ops 的回调钩子函数上。

Policy 抽象层加载完成后，APS 和抽象层建立了如下面对象图的关系：



APS 通过 audio_policy 接口访问抽象层的接口。audio_policy 接口提供设备选择，声音状态，音量控制等等接口。这些接口的实现大部分都在 APMB 中。APMB 通过 AudioPolicyCompactClient 调用 audio_policy_service_ops 回调方法访问 APS 中的方法。

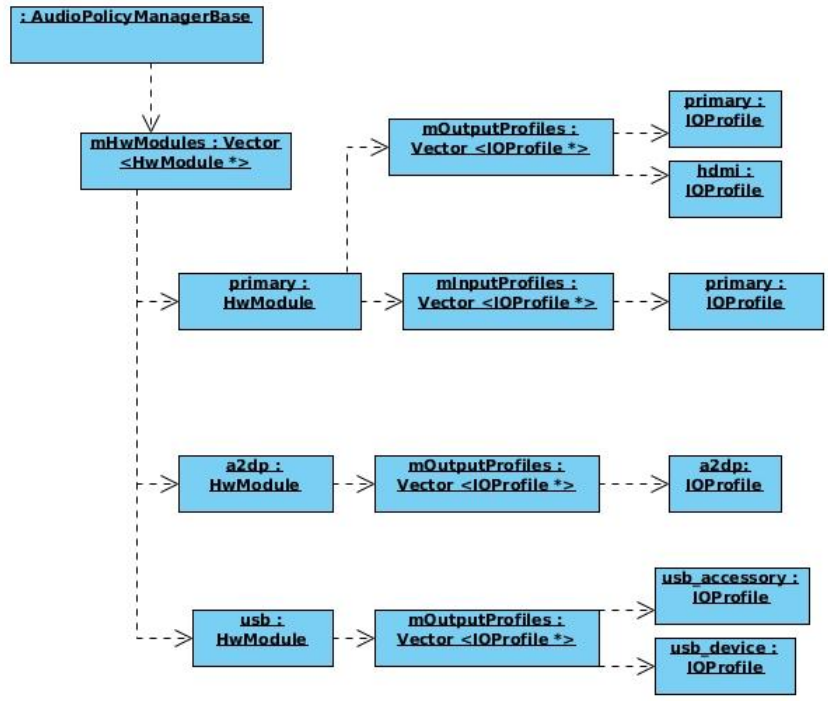
4.2.2. 加载硬件配置文件

在 Policy 抽象层的加载的最后，抽象层会调用 createAudioPolicyManager 方法创建 AudioPolicyManager。在 AudioPolicyManager 基类 AudioPolicyManagerBase(简称 APMB)的初始化中，会加载 2.4 节描述的硬件配置文件。然后通过配置文件，加载相关的 hardware 抽象层。加载配置文件的代码如下：

```
AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface
*clientInterface)
    // 根据配置文件创建 HwModule 层次结构
    if (loadAudioPolicyConfig(AUDIO_POLICY_VENDOR_CONFIG_FILE) != NO_ERROR) {
        #define AUDIO_POLICY_VENDOR_CONFIG_FILE "/vendor/etc/audio_policy.conf"
        {
            loadGlobalConfig(root);
```

```
        loadHwModules(root);
    }
    if (loadAudioPolicyConfig(AUDIO_POLICY_CONFIG_FILE) != NO_ERROR) {
    #define AUDIO_POLICY_CONFIG_FILE "/system/etc/audio_policy.conf"
        defaultAudioPolicyConfig();
    }
}
```

加载配置文件完成后，APMB 会解析配置文件，在内存中的会创建如下数据结构：



4.2.3. 加载 Audio Hardware 抽象层

配置文件加载并解析完成后，APMB 会根据配置文件加载相关的 hardware 抽象层。APMB 加载 Hardware 抽象层的代码如下：

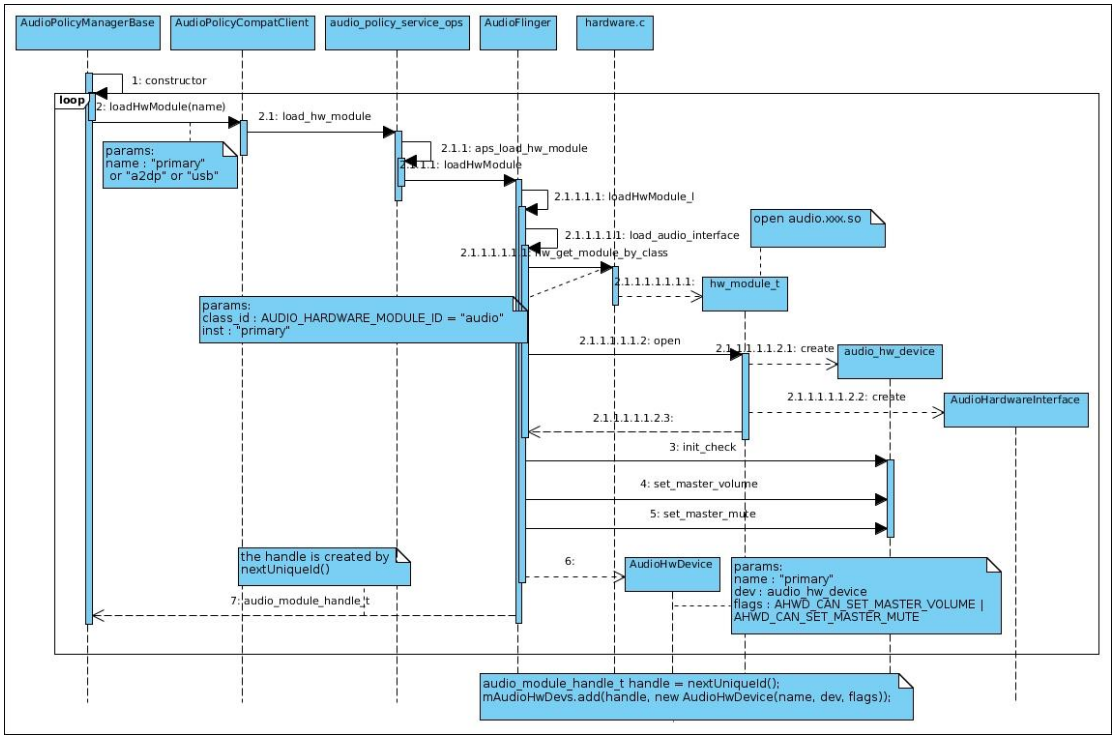
```
    // open all output streams needed to access attached devices
    for (size_t i = 0; i < mHwModules.size(); i++) {
        mHwModules[i]->mHandle =
mpClientInterface->loadHwModule(mHwModules[i]->mName);
        if (mHwModules[i]->mHandle == 0) {
            ALOGW("could not open HW module %s", mHwModules[i]->mName);
            continue;
        }
        // open all output streams needed to access attached devices
        for (size_t j = 0; j < mHwModules[i]->mOutputProfiles.size(); j++)
```

```
        {
            const IOProfile *outProfile = mHwModules[i]->mOutputProfiles[j];

            if (outProfile->mSupportedDevices & mAttachedOutputDevices) {
                AudioOutputDescriptor *outputDesc = new
AudioOutputDescriptor(outProfile);
                outputDesc->mDevice = (audio_devices_t)(mDefaultOutputDevice &

outProfile->mSupportedDevices);
                audio_io_handle_t output = mpClientInterface->openOutput(
                                                outProfile->mModule->mHandle,
                                                &outputDesc->mDevice,
                                                &outputDesc->mSamplingRate,
                                                &outputDesc->mFormat,
                                                &outputDesc->mChannelMask,
                                                &outputDesc->mLatency,
                                                outputDesc->mFlags);
                if (output == 0) {
                    delete outputDesc;
                } else {
                    mAvailableOutputDevices =
(audio_devices_t)(mAvailableOutputDevices |
                                            (outProfile->mSupportedDevices &
mAttachedOutputDevices));
                    if (mPrimaryOutput == 0 &&
                            outProfile->mFlags & AUDIO_OUTPUT_FLAG_PRIMARY) {
                        mPrimaryOutput = output;
                    }
                    addOutput(output, outputDesc);
                    setOutputDevice(output,
                                    (audio_devices_t)(mDefaultOutputDevice &
                                                        outProfile->mSupportedDevices),
                                    true);
                }
            }
        }
    }
```

加载的大体流程如下:



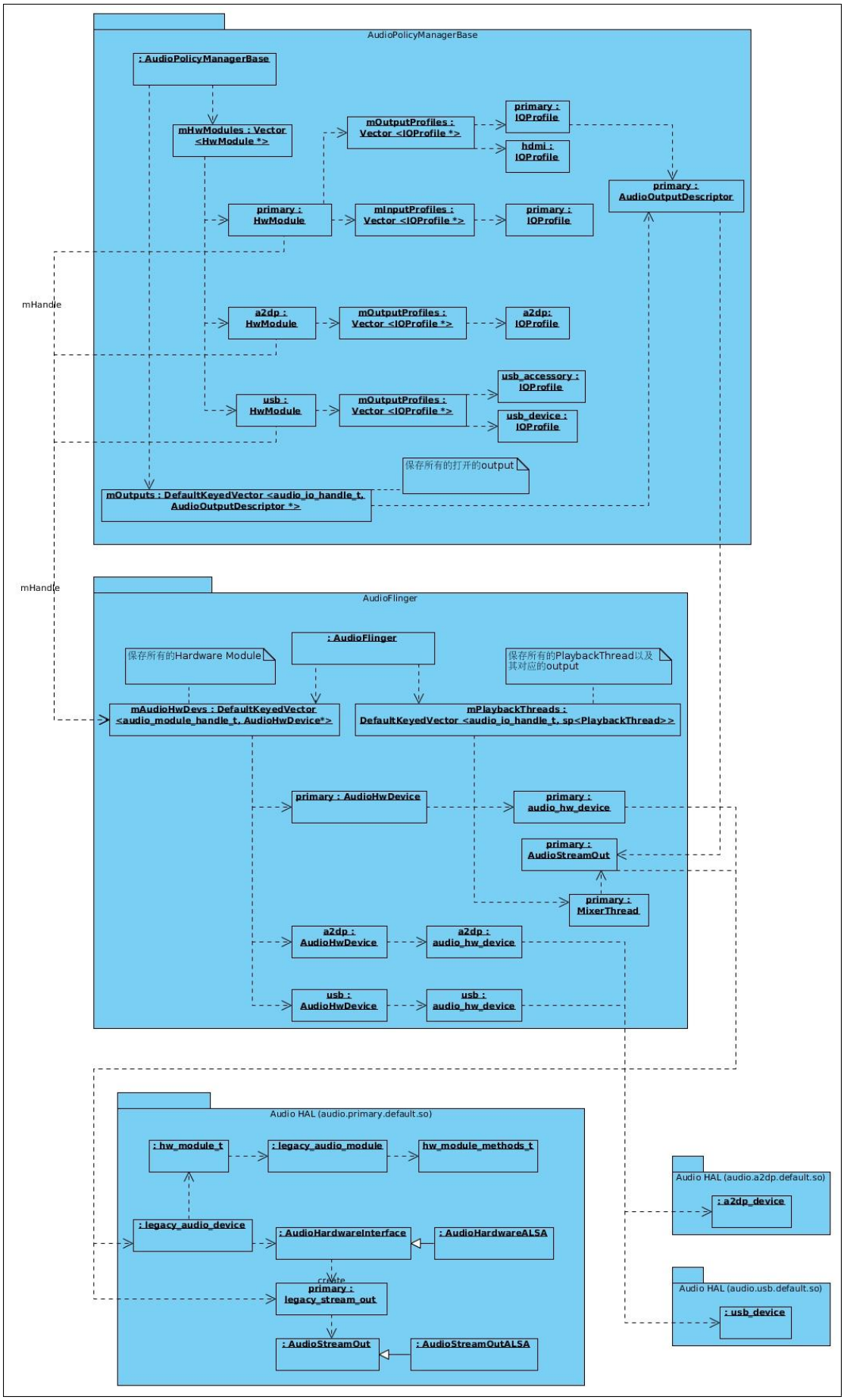
1. APMB 根据配置文件的描述，循序加载所有硬件模块。对应 2.4 节中的配置，会加载 primary, a2dp 和 usb 这三个模块。

2. APMB 通过 AudioPolicyClientInterface::loadHwModule 接口加载硬件模块，这个接口功能的实现在 AF 中的 loadHwModule_l 中。具体调用流程可参考上面序列图。

3. 在 loadHwModule_l 中，AF 通过 Android 的 HAL 接口打开对应硬件模块的动态库，如 audio.primary.vendor_name.so，audio.a2dp.vendor_name.so，audio.usb.vendor_name.so 等等。

4. AF 调用动态库的 open 方法创建 audio_hw_device，在 audio_hw_device 初始化中会创建 AudioHardwareInterface（3.3.1 节）。

5. AF 将 audio_hw_device 封装到 AudioHwDevice 中，并将该对象加入到全局数组 mAudioHwDevs 中。当 primary, a2dp 和 usb 三个模块加载完成后，数组 mAudioHwDevs 就包括了这三个模块的接口。AF 可通过数组的 id（audio_module_handle_t）来查找对应硬件模块的接口。


AudioPolicyManagerBase
: AudioPolicyManagerBase
mHwModules : Vector <HwModule *>
mOutputProfiles : Vector <IOProfile *>
primary : IOProfile
hdmi : IOProfile
primary : AudioOutputDescriptor
primary : HwModule
mInputProfiles : Vector <IOProfile *>
primary : IOProfile
mHandle
a2dp : HwModule
mOutputProfiles : Vector <IOProfile *>
a2dp: IOProfile
usb_accessory : IOProfile
usb : HwModule
mOutputProfiles : Vector <IOProfile *>
usb_device : IOProfile
保存所有的打开的output
mOutputs : DefaultKeyedVector <audio_io_handle_t, AudioOutputDescriptor *>
mHandle
AudioFlinger
保存所有的Hardware Module
: AudioFlinger
保存所有的PlaybackThread以及其对应的output
mAudioHwDevs : DefaultKeyedVector <audio_module_handle_t, AudioHwDevice*>
mPlaybackThreads : DefaultKeyedVector <audio_io_handle_t, sp<PlaybackThread>>
primary : AudioHwDevice
primary : audio_hw_device
primary : AudioStreamOut
primary : MixerThread
a2dp : AudioHwDevice
a2dp : audio_hw_device
usb : AudioHwDevice
usb : audio_hw_device
Audio HAL (audio.primary.default.so)
: hw_module_t
: legacy_audio_module
hw_module_methods_t
: legacy_audio_device
: AudioHardwareInterface
: AudioHardwareALSA
create
primary : legacy_stream_out
: AudioStreamOut
: AudioStreamOutALSA
Audio HAL (audio.a2dp.default.so)
: a2dp_device
Audio HAL (audio.usb.default.so)
: usb_device

6. loadHwModule_l 加载完成后，会将 AF 硬件接口的索引 audio_module_handle_t 返回给 APMB。

7. APMB 初始化过程中，会加载所有的 HwModule。不过根据硬件配置文件（2.4 节），仅仅会打开 primary module 的 primary output。并将 primary output 的 device 切到默认的输出设备扬声器上。

APMB 加载完硬件抽象层后，内存中生成如上图的数据结构。通过这些数据结构建立 APMB，AF 和硬件之间的接口。至此整个 Audio 系统的初始化就完成了。

5. Audio 策略系统主要场景

5.1. 打开声音输出

在 4.2.3 节中，我们知道 APMB 初始化过程中，会打开 primary module 的 primary output。下面将详细阐述这个过程：

1. APMB 创建一个 AudioOutputDescriptor 对象。该对象对应一个声音输出，保存 APMB 关注的声音输出的所有信息，如 audio_io_handle_t, 硬件配置信息（2.4 节），音量，静音等等。其中 audio_io_handle_t 是该输出在 AF 中的索引，用于获取该输出的硬件接口。audio_io_handle_t 在 AF 通过抽象层打开了硬件输入设备之后返回给 APMB。

2. APMB 通过 AudioPolicyCompatClient， audio_policy_service_ops 调用到 AF 的 openOutput 函数。主要参数包括 audio_module_handle_t 和 audio_devices_t。audio_module_handle_t 是硬件模块的 handle, AF 通过这个索引找到对应硬件模块的接口。audio_devices_t 表示打开输出后切换到哪个设备。

3. AF 在 openOutput 调用 findSuitableHwDev_l（参数是 audio_module_handle_t）找到对应硬件模块的接口。

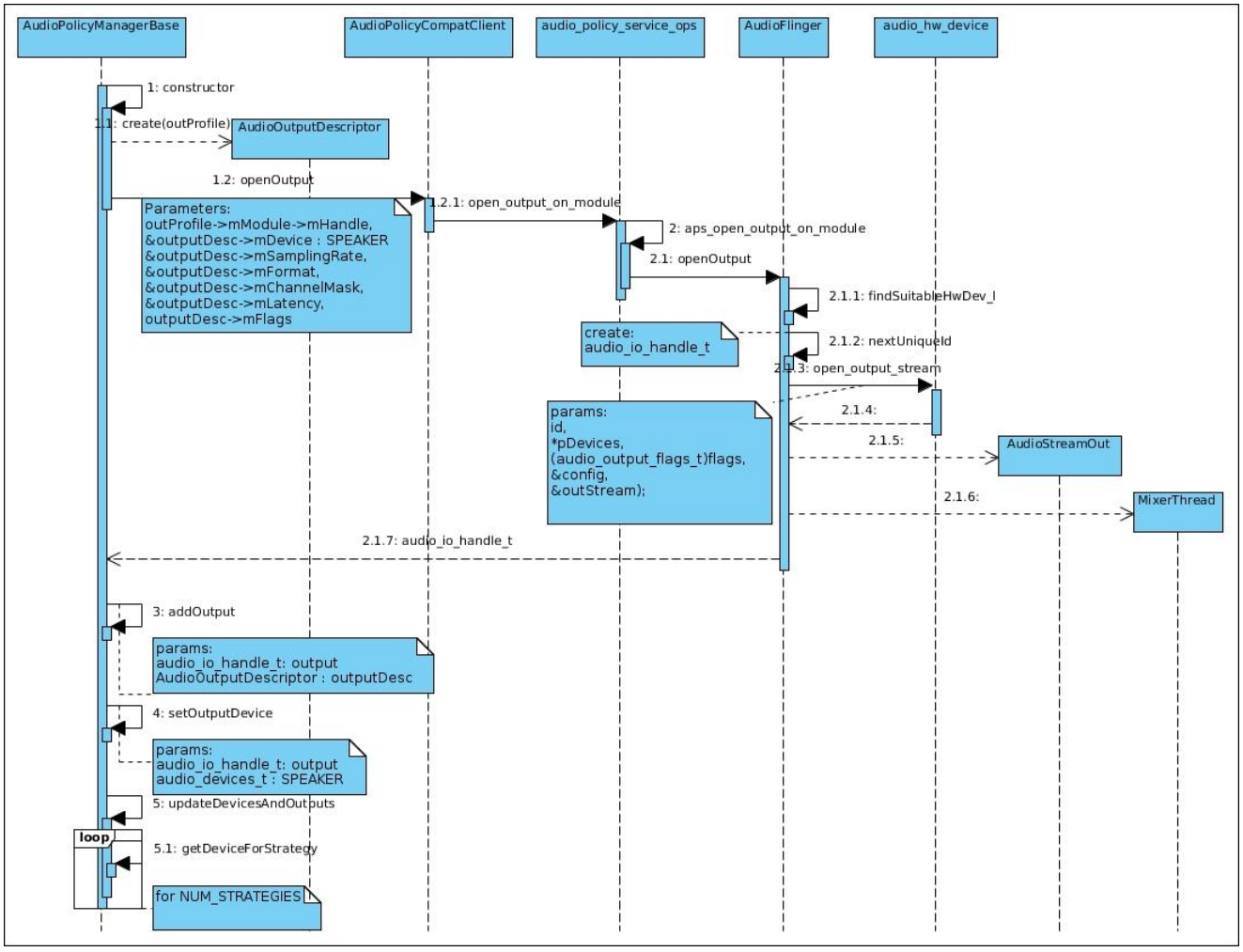
4. AF 调用 nextUniqueId 方法创建一个 audio_io_handle_t，就是步骤 1 中描述的输出接口的索引。

5. AF 调用硬件模块接口的方法 open_output_stream 打开声音输出。该接口

返回一个 audio_stream_out_t 的接口，这个接口封装了声音输出的所有操作。AF 将 audio_stream_out_t 封装到 AudioStreamOut 中。

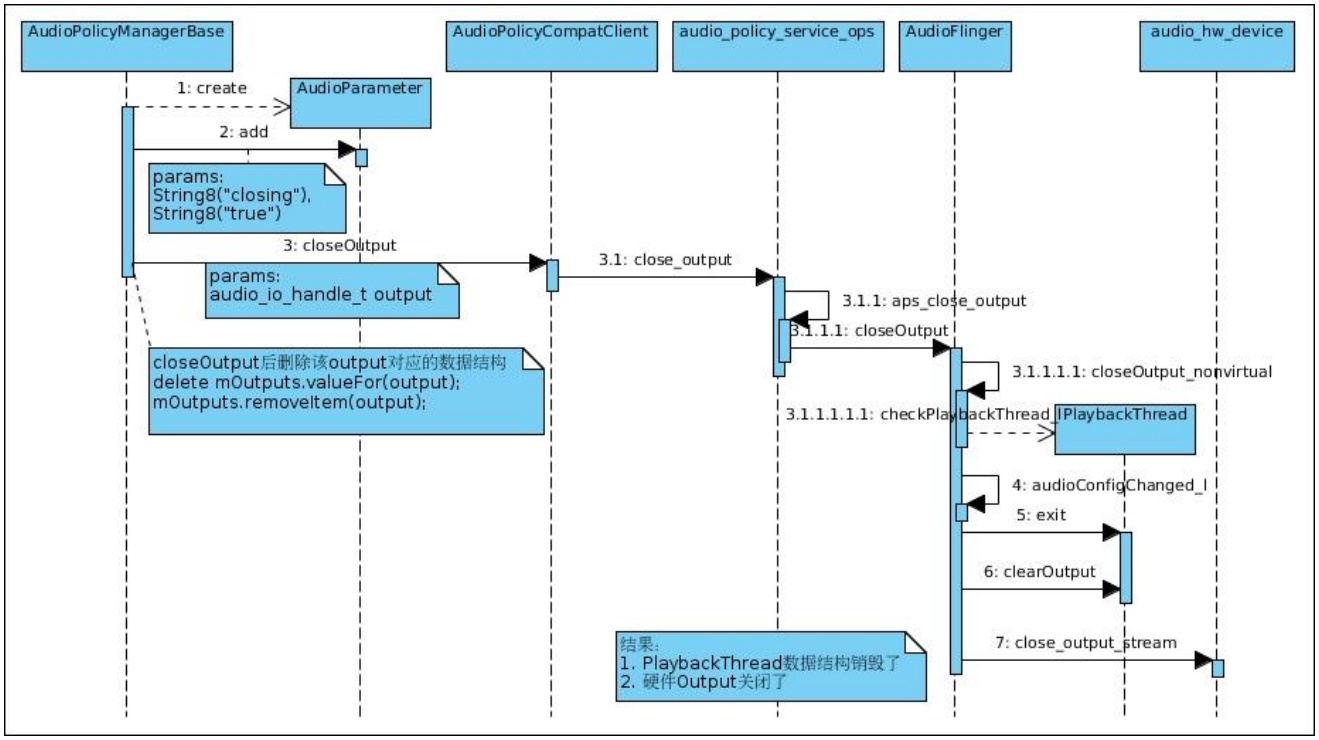
6. AF 创建一个 MixerThread，该对象封装了 AudioStreamOut。并将这个对象添加到 mPlaybackThreads vector 中。MixerThread 在 vector 中的索引就是步骤 4 创建的 audio_io_handle_t。

7. AF 将 audio_io_handle_t 返回给 APMB。APMB 将步骤 1 创建的 AudioOutputDescriptor 加入到全局 vector mOutputs 中，其索引就是 audio_io_handle_t。这样，通过这个 handle, APMB 中的 AudioOutputDescriptor 就和 AF 中 MixerThread，AudioStreamOut 一一对应起来了。



5.2. 关闭声音输出

关闭声音输出的流程和打开输出的流程类似，序列图如下：



5.3. 设置输出的设备

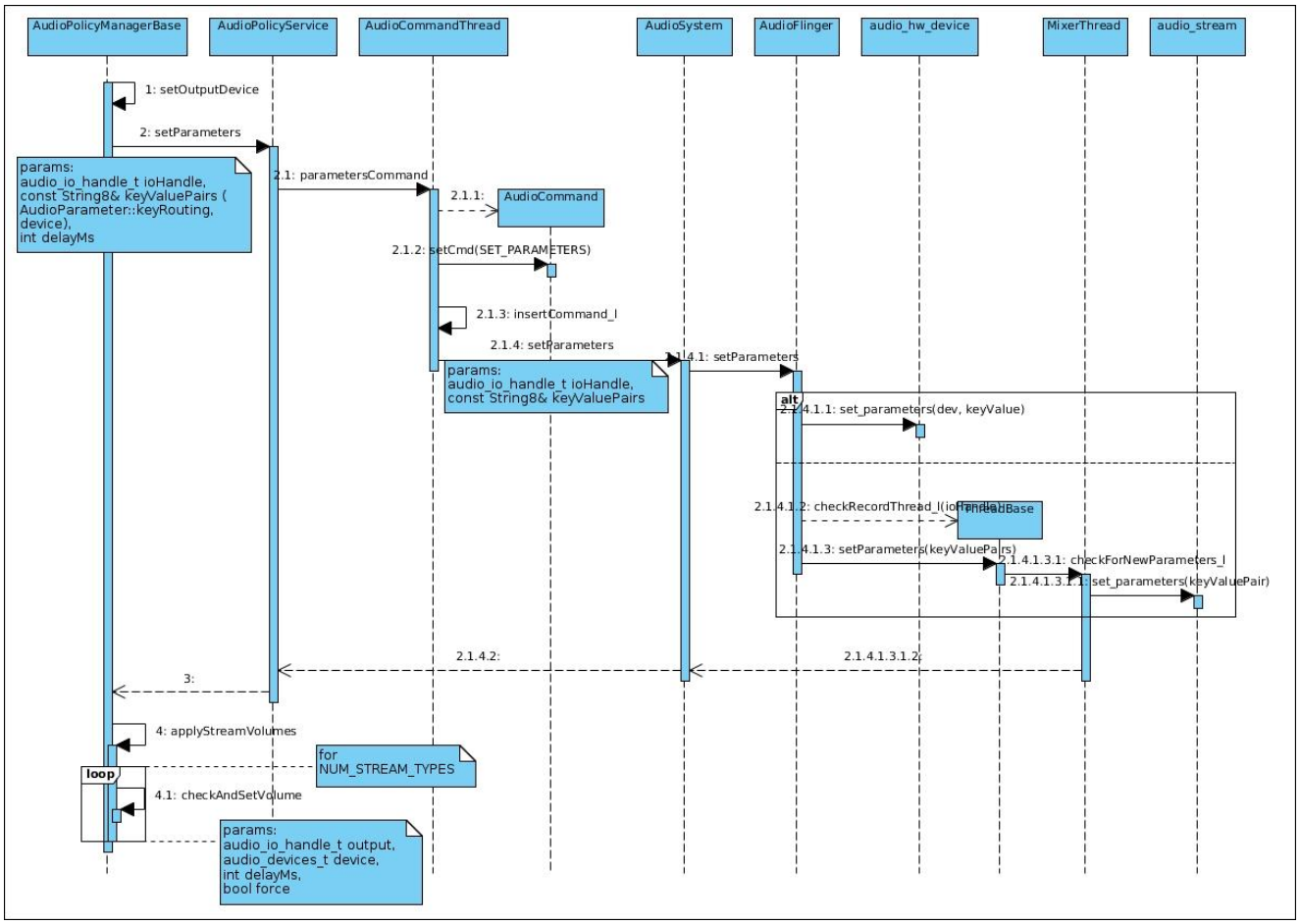
打开了硬件输出后，APMB 会调用 setOutputDevice 方法来设置输出中的设备。该方法的主要参数包括 audio_io_handle_t 和 audio_devices_t。audio_io_handle_t 表示需要操作的输出。audio_devices_t 表示需要切换到哪个设备上。具体流程如下：

1. APMB 通过 AudioPolicyCompatClient，audio_policy_service_ops 调用到 APS 的 setParameters 方法。该方法的参数中包括了一个 keyValuePairs(AudioParameter::keyRouting, device)的参数对。AudioParameter::keyRouting 表示需要输出切换设备， device 表示切换到的设备。

2. 在 APS 中有一个 AudioCommandThread，用于处理设置声音参数，音量和电话音量。APS 调用 parametersCommand 方法将声音参数传递到 AudioCommandThread。

3. AudioCommandThread 通过 AudioSystem 调用到 AF 的

setParameters。



4. 如果 audio_io_handle_t 是 0，AF 调用硬件模块接口 audio_hw_device 的 set_parameters，设置硬件模块全局的参数。切换设备时，audio_io_handle_t 一般都不为 0。

5. 如果 audio_io_handle_t 不为 0 时，AF 会找到 handle 对应的 MixerThread。MixerThread 会调用其对应 output 的接口 audio_stream 的 set_parameters 方法，设置该输出的设备。

5.4. 音量调节

Android Audio 系统为所有的 stream 定义了各自的音量。如来电铃声的音量和音乐的音量是分别控制的。

在 AudioManager 中定义了各个 stream 音量的默认值：

```
/** @hide Default volume index values for audio streams */
```

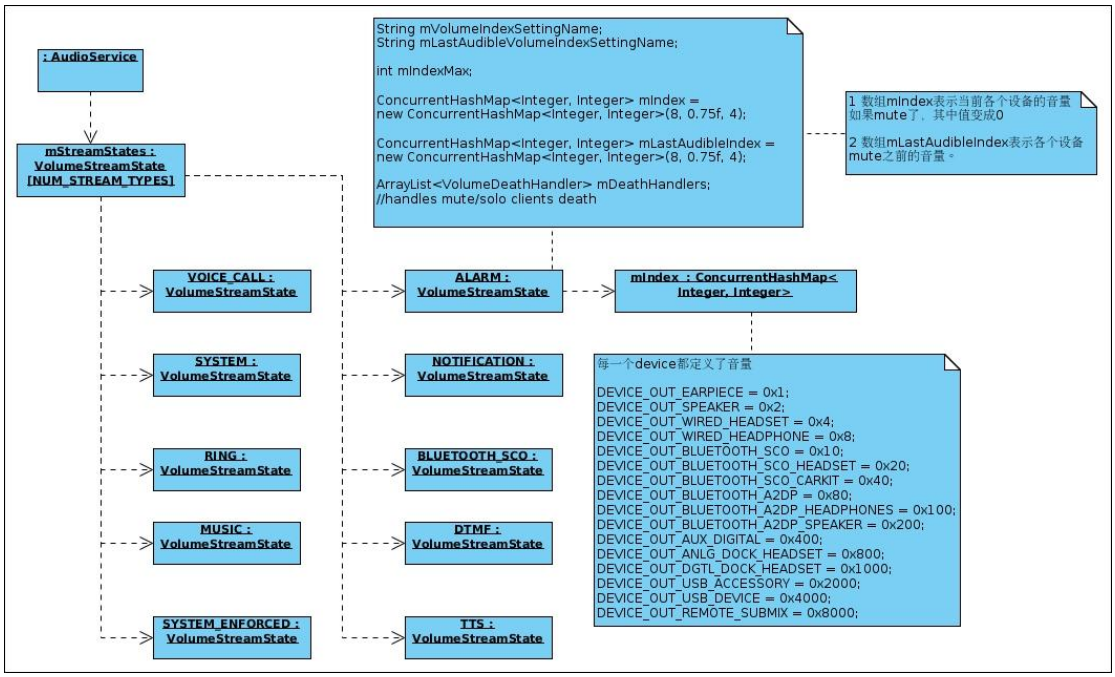
```
public static final int[] DEFAULT_STREAM_VOLUME = new int[] {
    4,  // STREAM_VOICE_CALL
    7,  // STREAM_SYSTEM
    5,  // STREAM_RING
    11, // STREAM_MUSIC
    6,  // STREAM_ALARM
    5,  // STREAM_NOTIFICATION
    7,  // STREAM_BLUETOOTH_SCO
    7,  // STREAM_SYSTEM_ENFORCED
    11, // STREAM_DTMF
    11,  // STREAM_TTS
    2   // STREAM_FM
};
```

在 AudioService 中定义了各个 stream 的最大值：

```
/** @hide Maximum volume index values for audio streams */
private final int[] MAX_STREAM_VOLUME = new int[] {
    5,  // STREAM_VOICE_CALL
    7,  // STREAM_SYSTEM
    7,  // STREAM_RING
    15, // STREAM_MUSIC
    7,  // STREAM_ALARM
    7,  // STREAM_NOTIFICATION
    15, // STREAM_BLUETOOTH_SCO
    7,  // STREAM_SYSTEM_ENFORCED
    15, // STREAM_DTMF
    15,  // STREAM_TTS
};
```

在 AudioService 中定义了全局数组 VolumeStreamState[] mStreamStates 来管理所有 stream 的音量。每个 VolumeStreamState 对象包括了一个 HashMap 来管理不同 device 的音量。如对于 stream music 有一个 VolumeStreamState 对象，该对象的 HashMap 中记录了不同设备播放音乐时的音量。使用耳机播放音乐和使用扬声器播放音乐时的音量是分开控制的。如用户开始使用耳机播放音乐，音量是 3。然后用户拔出耳机，使用扬声器开始播放。这时用户觉得声音小了，他将声音调节到 12。当用户再插入耳机时，由于耳机的音量和扬声器的音量是分开记录的，所以播放音乐的声音还是 3。如果耳机和扬声器的音量在一起控制，在上面情况就会导致耳机音量变得非常大，用户体验较差。下面对象图中描述了 AudioService 中管理音量的

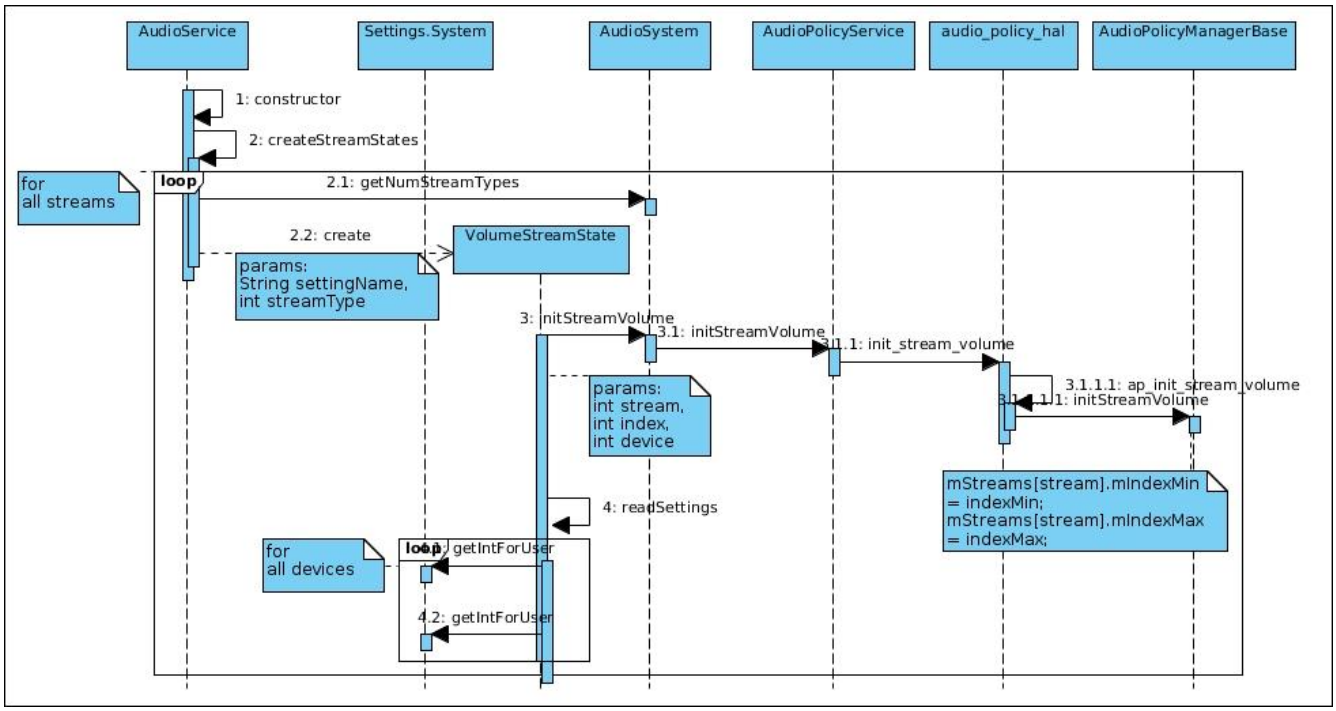
数据结构：



同时 Audio 系统定义了一个 volume alias 数组，该数组表示哪些 stream 的音量是一起进行调节的。从下面数组定义中看出，system, ring, notification, system enforced, dtmf 的是和 ring 的 alias。也就是说调节铃声音量时，system, notification, system enforced, dtmf 的音量都会相应发生变化。

```
private final int[] STREAM_VOLUME_ALIAS = new int[] {
    AudioSystem.STREAM_VOICE_CALL,        // STREAM_VOICE_CALL
    AudioSystem.STREAM_RING,              // STREAM_SYSTEM
    AudioSystem.STREAM_RING,              // STREAM_RING
    AudioSystem.STREAM_MUSIC,             // STREAM_MUSIC
    AudioSystem.STREAM_ALARM,             // STREAM_ALARM
    AudioSystem.STREAM_RING,              // STREAM_NOTIFICATION
    AudioSystem.STREAM_BLUETOOTH_SCO,     // STREAM_BLUETOOTH_SCO
    AudioSystem.STREAM_RING,              // STREAM_SYSTEM_ENFORCED
    AudioSystem.STREAM_RING,              // STREAM_DTMF
    AudioSystem.STREAM_MUSIC,             // STREAM_TTS
};
```

下面序列图描述了 AudioService 初始化系统音量的流程：



1. AudioStream 的 createStreamStates 中创建 VolumeStreamState 数组。并为每个 stream 创建 VolumeStreamState 对象。

2. 在VolumeStreamState的函数中设置stream的名字，类型，最大音量。

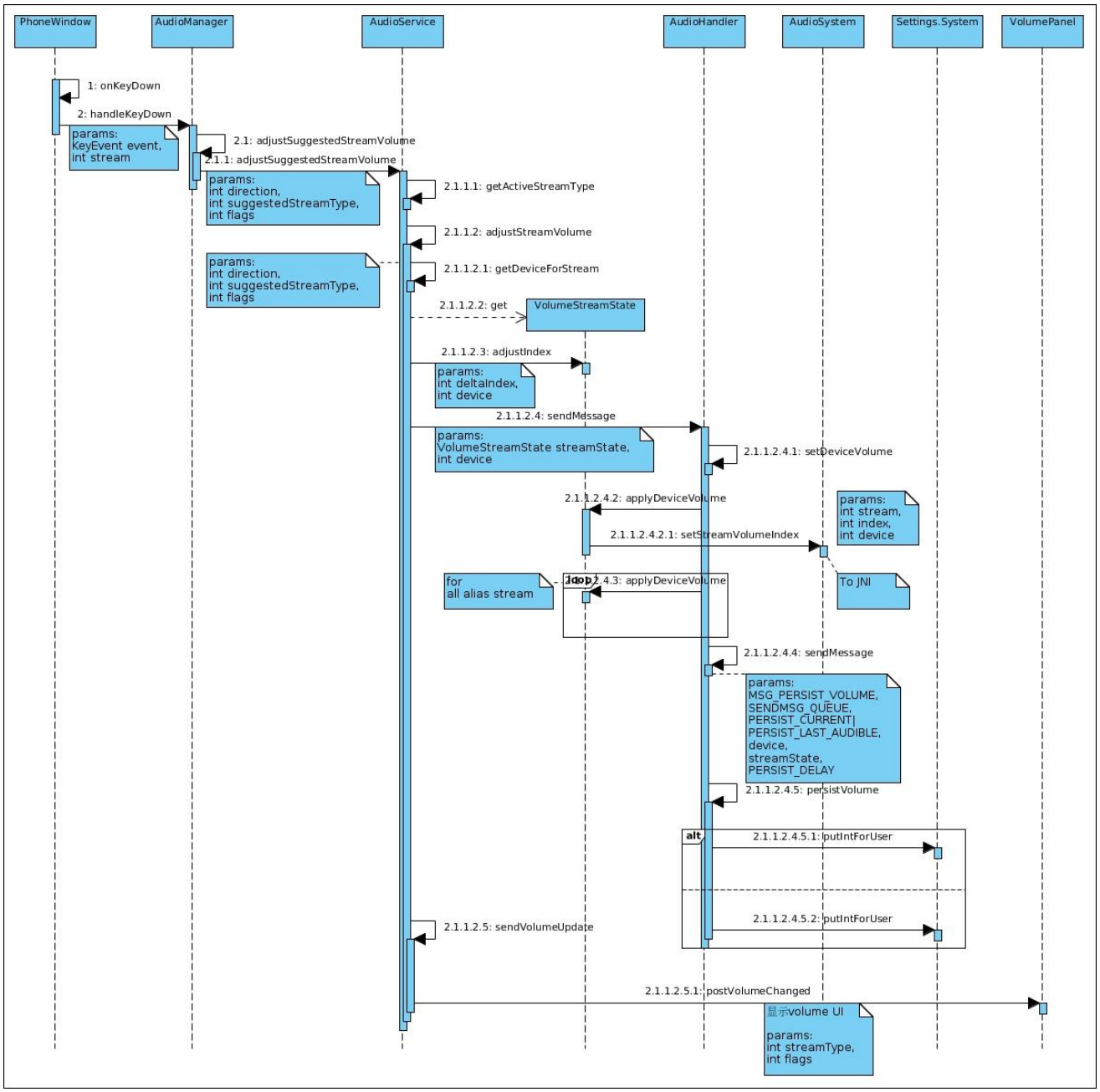
3. 在 VolumeStreamState 的函数中调用 AudioSystem 的 initStreamVolume 函数初始化 Native 层的音量相关数据结构。

4. 调用 readSettings 从系统数据库中获取每个 device 的音量。一般系统第一次启动时，数据库中没有 device 的音量。readSettings 会根据数组 DEFAULT_STREAM_VOLUME 设置一个默认的音量。

5. 调用 checkAllAliasStreamVolumes，将 alias 相同的 stream 的音量调整成一致的。该函数会调用 applyAllVolumes 将数据库中的音量设置到 native 层。

下面序列图描述用户通过音量键调节音量的过程：

1. PhoneWindow 收到 key down 事件后，调用 AudioManager 的 handleKeyDown 函数。



2. AudioManager 调用 AudioService 的 adjustSuggestedStreamVolume 函数。该函数主要参数包括 direction 和 suggestedStreamType。direction 是音量调节的方向，suggestedStreamType 是应用程序设置的 stream 的类型。一般应用程序都不会设置这个值。

3. AudioService 调用 getActiveStreamType 检查当前系统中正在播放的 stream 的类型。如果系统没有播放任何声音，返回 stream ring。

4. AudioService 调用 adjustStreamVolume 调节当前 stream 的音量。

5. 在函数 adjustStreamVolume 中获取该 stream 对应的 VolumeStreamState 对象。并通过 getDeviceForStream 获取该 stream 对应的设备。

6. 调用 VolumeStreamState 的 adjustIndex 方法调节音量，并向 AudioHandler 发送一条消息。AudioHandler 收到该消息后会将音量保存到系统数据库。

7. 调用 sendVolumeUpdate。该方法会触发显示调节音量的对话框，并发送音量变化的 Broadcast。

5.5. 声音冲突控制

声音冲突控制主要解决的问题是，如果系统中有多种声音需要同时播放出来，应该播放哪种声音。如媒体播放器播放音乐的过程中，闹钟到时，需要播放闹钟，Audio 系统应该如何处理。Audio 系统提供了 AudioFocus 机制来解决该问题。大致流程如下：

1. 程序 A 播放音乐前通过调用 requestAudioFocus 方法向 AudioService 申请 AudioFocus，并注册一个回调类 OnAudioFocusChangeListener 给 AudioService。该方法返回后，AudioService 就认为程序 A 获取到了 audio focus。

2. 程序 A 开始播放音乐。

3. 程序 B 准备播放音乐前，也向 AudioService 申请 AudioFocus。AudioService 收到 B 的申请后，发现 A 正在播放声音，则通过回调对象 OnAudioFocusChangeListener 的 onAudioFocusChange 方法通知程序 A，有其他程序准备播放声音了。

4. 程序 A 可在 onAudioFocusChange 方法中暂停，或降低自己的声音。

5. 程序 B 获取到了 audio focus 后开始播放自己的声音。

6. 程序 B 播放完毕后调用 abandonAudioFocus。AudioService 收到 B 的请求后，通知 A 重新获取到了 audio focus。

7. 程序 A 可在 onAudioFocusChange 方法中恢复自己的声音播放。

具体流程如下序列图：