Android音频系统之AudioPolicyService

http://blog.csdn.net/xuesen_lin/article/details/8805108

1.1 AudioPolicy Service

在AudioFlinger小节，我们反复强调它只是策略的执行者，而AudioPolicyService则是策略的制定者。这种分离方式有效地降低了整个系统的藕合性，而且为各个模块独立扩展功能提供了保障。

1.1.1 AudioPolicyService概述

汉语中有很多与策略有关联的俗语，比如"因地制宜"、"具体问题具体分析";战争中只遵照兵书制定战术的行为也被我们称为是"纸上谈兵"、死读书。这些都告诉我们，了解策略的执行环境是非常重要的，只有清晰地界定出"问题是什么"，才能有的放矢的制定出正确的Policy来解决问题。

Android系统中声音的种类有很多种，具体分类如下所示：

```
/*AudioManager.Java*/
public static final intSTREAM_VOICE_CALL = 0; /* 通话声音*/
public static final intSTREAM_SYSTEM = 1; /* 系统声音*/
public static final int STREAM_RING = 2; /* 电话铃声和短信提示 */
public static final intSTREAM_MUSIC = 3; /* 音乐播放 */
public static final intSTREAM_ALARM = 4; /* 闹铃 */
public static final intSTREAM_NOTIFICATION = 5; /* 通知声音 */

/*下面几个是隐藏类型，不对上层应用开放*/
public static final intSTREAM_BLUETOOTH_SCO = 6; /*当连接蓝牙时的通话*/
public static final intSTREAM_SYSTEM_ENFORCED = 7; /* 强制的系统声音，比如有的国家强制要求
                                                    摄像头拍照时有声音，以防止偷拍*/
public static final intSTREAM_DTMF = 8; /* DTMF声音 */
public static final intSTREAM_TTS = 9; /* 即text tospeech (TTS) */
```

针对这么多类型的音频，AudioPolicyService至少面临着如下几个问题：

- 上述类型的声音需要输出到哪些对应的硬件设备

比如一部典型的手机，它既有听筒、耳机接口，还有蓝牙设备。假设默认情况下播放音乐是通过听筒喇叭输出的，那么当用户插入耳机时，这个策略就会改变——从耳机输出，而不再是听筒;又比如在机器插着耳机时，播放音乐不应该从喇叭输出，但是当有来电铃声时，就需要同时从喇叭和耳机输出音频。这些"音频策略"的制定，主导者就是AudioPolicyService

- 声音的路由策略

如果把一个音乐播放实例(比如用MediaPlayer播放一首SD卡中的歌曲)比作源IP，那么上一步中找到的音频播放设备就是目标IP。在TCP/IP体系中，从源IP最终到达目标IP通常需要经过若干个路由器节点，由各路由器根据一定的算法来决定下一个匹配的节点是什么，从而制定出一条最佳的路由路径，如下图所示：

随笔档案

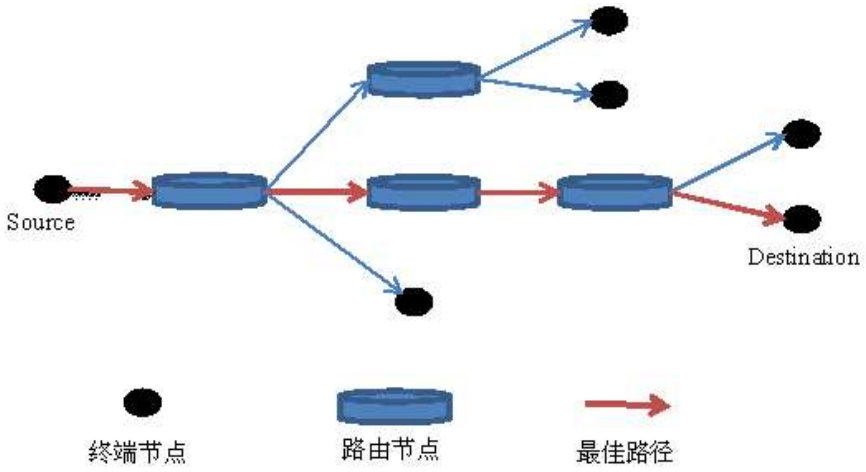

图 13-16 路由器示意图

AudioPolicyService所要解决的问题与路由器类似。因为系统中很可能存在多个audiointerface，每一个audio interface包含若干output，而每个output又同时支持若干种音频设备，这就意味着从播放实例到终端设备，需要经过audiointerface和output的选择，我们称之为AudioPolicyService的路由功能。

每种类型音频的音量调节

不同类型的音频，其音量的可调节范围是不一样的，比如有的是0-15，而有的则是1-20。而且它们的默认值也是有差别的，我们看AudioManager中的定义：

```
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
};
```

音量的调节部分后面我们有专门的小节来介绍。

为了让大家对AudioPolicyService有个感性的认识，我们以下图来形象地表示它与AudioTrack及AudioFlinger间的关系：

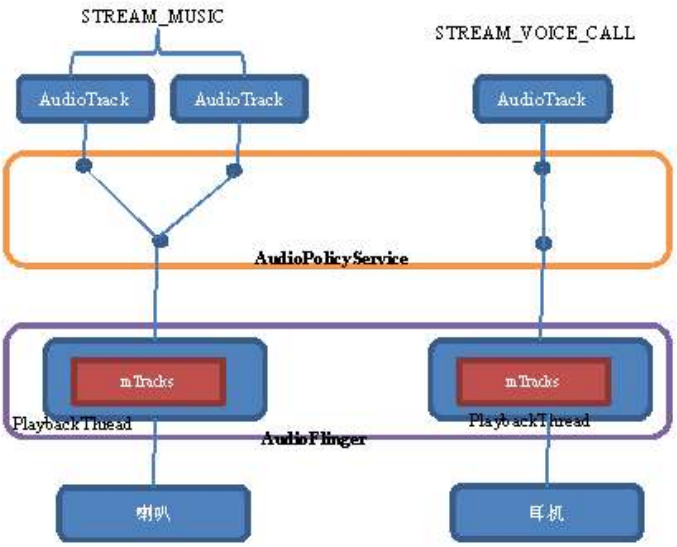

图 13-17 AudioPolicyService与AudioTrack和AudioFlinger的关系

这个图中的元素包括AudioPolicyService、AudioTrack、AudioFlinger、PlaybackThread以及两音频设备(喇叭、耳机)。它们之间的关系如下(特别注意，本例的目的只是说明这些元素的关系，并不代表图中的策略就是Android系统所采用的)：

l 一个PlaybackThread的输出对应了一种设备

比如图中有两个设备，就有两个PlaybackThread与之对应。左边的Thread最终混音后输出到喇叭，而右边的则输出到耳机

l 在特定的时间，同一类型音频对应的输出设备是统一的

也就是说，如果当前STREAM_MUSIC对应的是喇叭，那么所有该类型的音频都会输出到喇叭。结合上一点，我们还可以得出一个结论，即同一类型音频对应的PlaybackThread也是一样的

l AudioPolicyService起到了路由的作用

AudioPolicyService在整个选择过程中的作用有点类似于网络路由器，它有权决定某一个AudioTrack所产生的音频流最终会走向哪个设备，就像路由器可以根据一定的算法来决定发送者的包应该传递给哪个节点一样

接下来我们从三个方面来了解AudioPolicyService。

首先，从启动过程来看AudioPolicyService的工作方式。

其次，我们结合上面的关系图详细分析AudioPolicyService是如何完成“路由功能”的。

最后，我们来分析Android系统下默认的“路由策略”是怎样的。

1.1.2 AudioPolicyService的启动过程

还记得前面我们在分析AudioFlinger的启动时，曾经看到过AudioPolicyService的影子吗？没错，它和AudioFlinger是驻留在同一个程序中的，如下所示：

```
/*frameworks/av/media/mediaserver/Main_mediaserver.cpp*/

int main(int argc, char** argv)

{  …

  AudioFlinger::instantiate();

   …

  AudioPolicyService::instantiate();

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

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

}
```

因而从理论上来讲，AudioFlinger和AudioPolicyService是可以直接进行函数调用的。不过实际上它们仍然采用标准的Binder进行通信。

AudioPolicyService的启动方式和AudioFlinger也是类似的，我们这里就不赘述，直接来看它的构造函数：

```
AudioPolicyService::AudioPolicyService()
    : BnAudioPolicyService() ,mpAudioPolicyDev(NULL) , mpAudioPolicy(NULL)
{
    charvalue[PROPERTY_VALUE_MAX];
    const struct hw_module_t*module;
    int forced_val;
    int rc;
    …
    rc =hw_get_module(AUDIO_POLICY_HARDWARE_MODULE_ID, &module);//Step 1.
    …
    rc =audio_policy_dev_open(module, &mpAudioPolicyDev);//Step 2.
    …
    rc =mpAudioPolicyDev->create_audio_policy(mpAudioPolicyDev, &aps_ops, this,
                                              &mpAudioPolicy);//Step3.
    …
    rc =mpAudioPolicy->init_check(mpAudioPolicy); //Step 4.
…
//Step 5
   property_get("ro.camera.sound.forced", value, "0");
    forced_val = strtol(value,NULL, 0);
mpAudioPolicy->set_can_mute_enforced_audible(mpAudioPolicy,!forced_val);
//Step 6.
    if(access(AUDIO_EFFECT_VENDOR_CONFIG_FILE, R_OK) == 0) {
       loadPreProcessorConfig(AUDIO_EFFECT_VENDOR_CONFIG_FILE);
    } else if(access(AUDIO_EFFECT_DEFAULT_CONFIG_FILE, R_OK) == 0) {
        loadPreProcessorConfig(AUDIO_EFFECT_DEFAULT_CONFIG_FILE);
    }
}
```

我们将上述代码段分为6个步骤来讲解。

Step1@ AudioPolicyService::AudioPolicyService. 得到Audio Policy的hw_module_t，原生态系统中Policy的实现有两个地方，即Audio_policy.c和Audio_policy_hal.cpp，默认情况下系统选择的是后者(对应的库是libaudiopolicy_legacy)

Step2@ AudioPolicyService::AudioPolicyService. 通过上一步得到的hw_module_t打开Audio Policy设备(这并不是一个传统意义的硬件设备，而是把Policy虚拟成了一种设备。这样子的实现方式让音频硬件厂商在制定自己的音频策略时多了不少灵活性)。原生态代码中audio_policy_dev_open调用的是legacy_ap_dev_open@Audio_policy_hal.cpp，最终生成的Policy Device实现是legacy_ap_device

Step 3@ AudioPolicyService::AudioPolicyService. 通过上述的Audio Policy设备来产生一个策略，其对应的具体实现方法是create_legacy_ap@Audio_policy_hal.cpp。这个函数首先生成的一个legacy_audio_policy@Audio_policy_hal.cpp，而mpAudioPolicy对应的则是legacy_audio_policy::policy。除此之外，legacy_audio_policy还包含如下重要成员变量：

```
struct legacy_audio_policy {
    structaudio_policy policy;
    void *service;
    structaudio_policy_service_ops *aps_ops;
   AudioPolicyCompatClient *service_client;
```

```
    AudioPolicyInterface *apm;
};
```

其中aps_ops是由AudioPolicyService提供的函数指针(aps_ops)，这里面的函数是AudioPolicyService与外界沟通的接口，后面还会经常遇到。

最后一个apm是AudioPolicyManager的简写，AudioPolicyInterface是其基类，apm在原生态实现上是一个AudioPolicyManagerDefault对象，它是在create_legacy_ap中创建的：

```
static int create_legacy_ap(const struct audio_policy_device*device,
                            structaudio_policy_service_ops *aps_ops,
                           void *service,
                           struct audio_policy **ap)
{
    struct legacy_audio_policy*lap;
                ...
                lap->apm =createAudioPolicyManager(lap->service_client);
...}
```

函数createAudioPolicyManager默认情况下对应的是AudioPolicyManagerDefault.cpp中的实现，所以它将返回一个AudioPolicyManagerDefault。

是不是觉得Policy相关的类越来越多了？那为什么需要这么多类呢？我们先来看一下它们之间的关系：

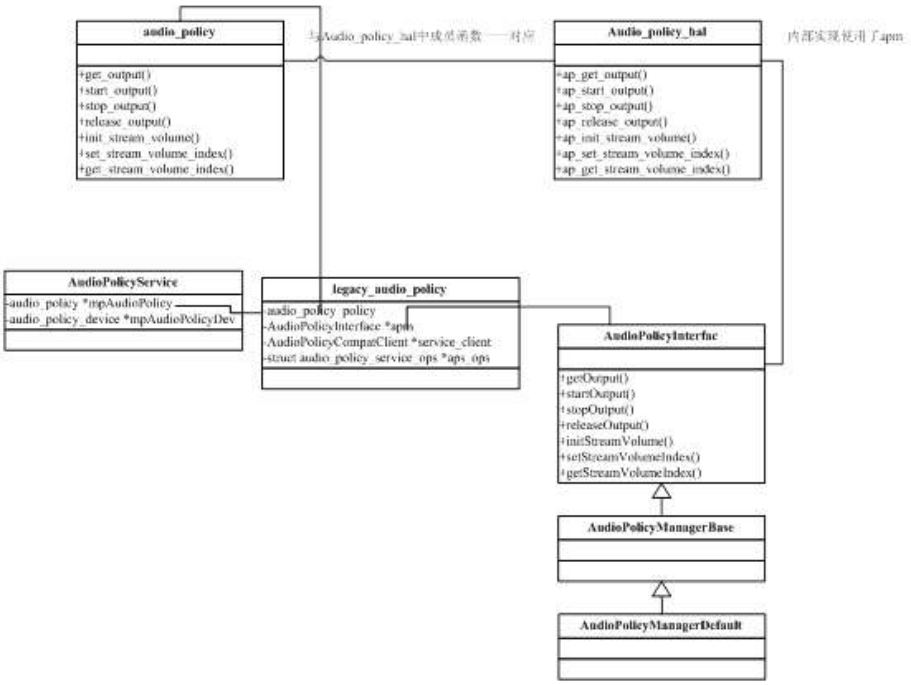

图 13-18 Audio Policy相关类的关系

看起来很复杂，其实概况起来就以下几点：

- AudioPolicyService持有的只是一个类似于接口类的对象，即audio_policy。换句话说，AudioPolicyService是一个“壳”，而audio_policy则是一个符合要求的插件。插件与壳之间的接口是固定不变的，而内部实现却可以根据厂商自己的需求来做

- 我们知道，audio_policy实际上是一个C语言中的struct类型，内部包含了各种函数指针，比如get_output、start_output等等。这些函数指针在初始化时，需要指向具体的函数实现，这就是Audio_policy_hal中的ap_get_output、ap_start_output等等

- 上面提到的各数据类型更多的只是一个“壳”，而真正的实现者是AudioPolicyManager。与此相关的又有三个类：AudioPolicyInterface是它们的基类，AudioPolicyManagerBase实现了一些基础的策略，而AudioPolicyManagerDefault则是最终的实现类。除了AudioPolicyService，后面这两个类也是我们研究Audio Policy的重点

Step 4@ AudioPolicyService::AudioPolicyService. 进行初始化检测，原生态的实现直接返回0

Step 5@ AudioPolicyService::AudioPolicyService. 判断是否强制执行相机拍照声音

Step 6@ AudioPolicyService::AudioPolicyService. 加载音频效果文件(如果存在的话)，文件路径如下:

AUDIO_EFFECT_DEFAULT_CONFIG_FILE"/system/etc/audio_effects.conf"

AUDIO_EFFECT_VENDOR_CONFIG_FILE"/vendor/etc/audio_effects.conf"

这样AudioPolicyService就完成了构造，它在ServiceManager中的注册名称为"media.audio_policy"。其中包含的mpAudioPolicy变量是实际的策略制定者，而它也是由HAL层创建的，换句话说是根据硬件厂商自己的"意愿"来执行策略的。

1.1.3 AudioPolicyService加载音频设备

在AudioFlinger的"设备管理"小节，我们曾简单提及AudioPolicyService将通过解析配置文件来加载当前系统中的音频设备。具体而言，当AudioPolicyService构造时创建了一个AudioPolicyDevice(mpAudioPolicyDev)并由此打开一个AudioPolicy(mpAudioPolicy)——这个Policy默认情况下的实现是legacy_audio_policy::policy(数据类型audio_policy)。同时legacy_audio_policy还包含了一个AudioPolicyInterface成员变量，它会被初始化为一个AudioPolicyManagerDefault，这些都是我们在前一个小节分析过的。

那么AudioPolicyService在什么时候去加载音频设备呢?

除了后期的动态添加外，另外一个重要途径是通过AudioPolicyManagerDefault的父类，即AudioPolicyManagerBase的构造函数。

```
AudioPolicyManagerBase::AudioPolicyManagerBase(AudioPolicyClientInterface*clientInterface
)...
{   mpClientInterface= clientInterface;
    ...
    if (loadAudioPolicyConfig(AUDIO_POLICY_VENDOR_CONFIG_FILE) != NO_ERROR){
        if (loadAudioPolicyConfig(AUDIO_POLICY_CONFIG_FILE) != NO_ERROR) {
            defaultAudioPolicyConfig();
        }
    }
    for (size_t i = 0; i < mHwModules.size();i++) {
      mHwModules[i]->mHandle = mpClientInterface->loadHwModule(mHwModules[i]-
>mName);
        if(mHwModules[i]->mHandle == 0) {
            continue;
        }
        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(...);
                        ...
}
```

不同的Android产品在音频的设计上通常是有差异的，利用配置文件的形式(audio_policy.conf)可以使厂商方便地描述其产品中所包含的音频设备，这个文件的存放路径有两处:

#define AUDIO_POLICY_VENDOR_CONFIG_FILE "/vendor/etc/audio_policy.conf"

#define AUDIO_POLICY_CONFIG_FILE"/system/etc/audio_policy.conf"

如果audio_policy.conf不存在的话，则系统将使用默认的配置，具体实现在defaultAudioPolicyConfig中。通过配置文件可以读取如下信息：

· 有哪些audiointerface，比如有没有“primary”、“a2dp”、“usb”

· 每个audiointerface的属性。比如支持的sampling_rates、formats、支持哪些device等等。这些属性是在loadOutput@AudioPolicyManagerBase中读取的，并存储到HwModule->mOutputProfiles中。每一个audiointerface下可能有若干个output和input，而每个output/input下又有若干具体的支持属性，关系如下图所示：

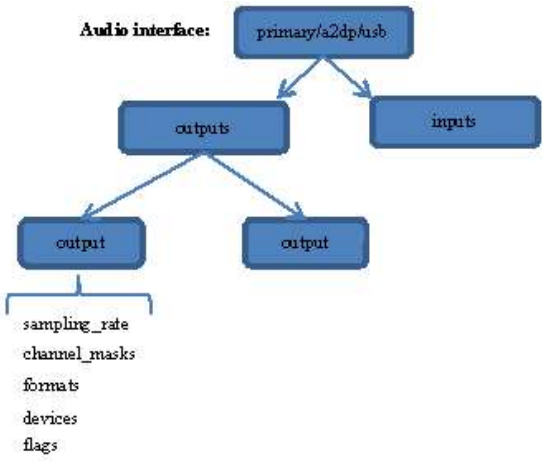

图 13-19 audio_policy.conf中各元素关系图

大家可以自己打开一个audio_policy.conf来具体了解这个文件的格式要求，我们这里就不做深入讲解了。

读取了相关配置后，接下来就要打开这些设备了。AudioPolicyService只是策略制定者，而非执行者，那么是由谁来完成这些具体的工作呢？没错，一定是AudioFlinger。我们可以看到上述函数段中有一个mpClientInterface变量，它是否和AudioFlinger有联系？可以先来分析下这个变量是如何来的。

很明显的mpClientInterface这个变量在AudioPolicyManagerBase构造函数的第一行进行了初始化，再回溯追踪，可以发现它的根源在AudioPolicyService的构造函数中，对应的代码语句如下：

```
rc =mpAudioPolicyDev->create_audio_policy(mpAudioPolicyDev, &aps_ops, this,
&mpAudioPolicy);
```

在这个场景下，函数create_audio_policy对应的是create_legacy_ap，并将传入的aps_ops组装到一个AudioPolicyCompatClient对象中，也就是mpClientInterface所指向的那个对象。

换句话说，mpClientInterface->loadHwModule实际上调用的就是aps_ops->loadHwModule，即：

```
static audio_module_handle_t  aps_load_hw_module(void*service,const char *name)
{
   sp<IAudioFlinger> af= AudioSystem::get_audio_flinger();
   …
   returnaf->loadHwModule(name);
}
```

AudioFlinger终于出现了，同样的情况也适用于mpClientInterface->openOutput，代码如下：

```
static audio_io_handle_t  aps_open_output(…)
{
   sp<IAudioFlinger> af= AudioSystem::get_audio_flinger();
   …
   return  af->openOutput((audio_module_handle_t)0,pDevices, pSamplingRate, pFormat,
pChannelMask,
                    pLatencyMs, flags);
}
```

再回到AudioPolicyManagerBase的构造函数中来，for循环的目标有两个：

Ø 利用loadHwModule来加载从audio_policy.conf中解析出的audio interface，即mHwModules数组中的元素

Ø 利用openOutput来打开各audio interface中包含的所有Output

关于AudioFlinger中这两个函数的实现，我们在前一个小节已经分析过了，这里终于把它们串起来了。通过AudioPolicyManagerBase，AudioPolicyService解析出了设置中的音频配置，并利用AudioFlinger提供的接口完成了整个音频系统的部署，从而为后面上层应用使用音频设备提供了底层支撑。下一小节我们就看下上层应用具体是如何使用这一框架来播放音频的。

分类: android 内核及系统

海王
关注 - 0
粉丝 - 253
+加关注

« 上一篇: 如何在Android平台上使用USB Audio设备
» 下一篇: Android音频系统之AudioFlinger(一)

posted on 2016-11-18 11:25 海王 阅读(1905) 评论(0) 编辑 收藏

相关博文:
- Android音频系统之AudioFlinger(二)
- Android音频系统之AudioFlinger(一)
- 声音、音频-Android音频系统之AudioPolicyService-by小雨
- Android AudioPolicyService服务启动过程
- Android音频系统之音频框架

最新新闻:
- 贺建奎最新回应：坚信伦理将站在我们一边
- 中国细胞生物学学会：科研伦理的高压线不容碰触
- 全球首例基因编辑婴儿 为什么该被黑?
- 报道称亚马逊上周曾请求西班牙警方镇压员工罢工
- 仅成功一例，医院拒认……首例基因编辑婴儿诞生的背后藏着多少谜?

» 更多新闻...

历史上的今天:
2013-11-18 Linux下timer延时的使用
2010-11-18 ubuntu root 登录
2010-11-18 Klimt 特点 与OpenGL和OpenGL|ES 对比及其关系
2010-11-18 在ARM Linux上使用OpenGL(转)