**Android 应用程序构建原理**

Android是Google基于Linux平台开发的开源手机操作系统，每个应用程序在底层都对于着一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行。

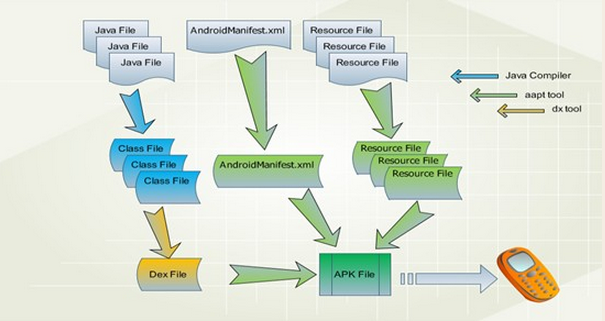
每一个Android应用程序都运行在一个Dalvik虚拟机实例中，每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制、内存分配和管理、mutex等等都是依赖底层操作系统而实现的。所有Android应用的线程对应一个Linux线程，虚拟机因而可以更多的依赖操作系统的线程调度和管理机制。

不同的应用在不同的进程空间里运行，加之对不同来源的一个用都使用不同的Linux用户来运行，可以最大程度的保护应用的安全和独立运行。

Zygote是一个虚拟机进程，同时也是一个虚拟机实例的孵化器，每当系统要求执行一个Android应用程序，Zygote就是FORK出一个子进程来执行该应用系统。

这样的好处显而易见：Zygote进程是在系统启动时产生的，它会完成虚拟机的初始化，库的加载，预置类库和初始化等等操作，而在系统需要一个新的虚拟机实例时。Zygote通过复制自身，最快速的提供一个系统。另外，对于一些只读的系统库，所有虚拟机实例都和Zygote共享一块内存区域，大大节省了内存开销。

应用程序包(APK)被发布到手机上后，运行前会对其中的DEX文件进行优化，优化后的文件被保持到缓存区域(优化后的格式称为DEY)，虚拟机会直接执行该文件。如果应用包文件不发生变化，DEY文件不会重新生成。



Android应用程序所使用的编程语言是Java语言，和JavaSE一样，编译时使用SunJDK将Java源代码编译成标准的Java字节码文件(.class文件)。而后通过工具软件DX把所有的字节码转换成DEX文件(classes.dex)。

最后使用Android打包工具(aapt)讲DEX文件、资源文件以及AndroidManifest.xml文件(二进制格式)组合成一个应用程序包(APK)。应用程序包可以被发布到手机上运行。而后Dalvik虚拟机会从其中读取指令和数据。

**Andorid系统架构**

Android系统从底层上一共分了4层，每一层都把底层实现封装，并暴露调用接口上一层：



1 Linux内核(Linux Kernel)

. Android运行在Linux Kernel2.6之上，但是把Linux内受GNU协议约束的部分做了取代，这样在Android的程序可以用于商业目的。

. Linux内核是硬件和软件层的抽象层。

2 中间件

. 中间件包括两部分：核心库和运行时(Libraries&Android runtime)

. 核心库包括：SurfaceManager显示系统管理库，负责把2D或3D内容显示到屏幕；Media Framework媒体库，负责支持图像，支持多种视频和音频的录制和回放；Sqlite数据库，一个功能强大的轻量级嵌入式关系数据库；Webkit浏览器引擎等。

. Dalvik虚拟机：区别于Java虚拟机的是，每一个Android应用程序都在它自己的进程中运行，都有一个属于自己的Dalvik虚拟机，这一点可以让系统在运行时可以达到优化，程序间的影响大大降低。Dalvik虚拟机并非运行Java字节码，而是运行自己的字节码。

3 应用程序框架(Application Framework)

. 丰富而又可以扩展性的视图(Views)，可以用来构建应用程序，包括列表(Lists)，网格(Grids)，文本框(Text Boxes)，按钮(Buttons) ，可嵌入的web浏览器。

. 内容提供者(Content Providers)使得应用程序可访问另一个应用程序的数据(如联系人数据库)，或者共享它们自己的数据。

. 资源管理器(Resource Manager)提供飞代码资源的访问，如本地字符串，图形和布局文件(Layout Files)。

. 通知管理器(Notification Manager)使得应用程序可以再状态栏中显示自定义的提示信息。

. 活动管理器(Activity Manager)用来管理应用程序生命周期并提供常用的导航回退功能。

4 应用程序(Applications)

. Android系统会内置一些应用程序包包括email客户端，SMS短消息，日历，地图，浏览器，联系人管理程序等。所有的应用程序都是使用Java语言编写的。

**Android基础之Process**

默认情况下，同一个应用程序中的所有组件运行在同一个进程中，而且绝大多数的应用程序也都是这样的。但是，如果我们想要控制让某个特定的组件属于某个进程，我们可以在manifest文件中进行配置。

在每种组件元素(Activity、Service、Receiver、Provider)的Manifest条目中，都支持一个“android:process”的属性，通过这个属性，我们可以指定某个组件运行的进程。我们可以通过设置这个属性，让每个组件运行在它自己的进程中，也可以只让某些组件共享一个进程。我们要可以通过设置“android:process”属性，让不同应用程序中的组件运行在相同的进程中，这些应用程序共享相同的Linux用户ID，拥有相同的证书。

<application>元素也有一个“android:process”属性，可以设置一个应用的全部组件的默认值。

当可用内存数量低，而一些与用户即时交互的进程又需要内存时，Android随时可能会终止某个进程。运行在被终止的进程中的组件会因此被销毁，但是，当再次需要这些组件工作时，就会再启动一个进程。

在决定要终止哪个进程时，Android系统会权衡它们对于用户的重要性。例如，相较于运行可见activities的进程，终止一个运行不可见activities的进程会更加合理。是否终止一个进程，依赖于运行在这个进程中的组件的状态。

**RIL(Radio Interface Layer)**

目前的智能机在硬件上多采用双cpu的架构，一个是基带处理器，主要处理数字信号、语音信号的编码解码以及GSM通信协议，另一个是应用处理器，运行[操作系统](http://baike.baidu.com/view/880.htm)和各种[应用程序](http://baike.baidu.com/view/330120.htm)。基带处理器、射频和其它外围芯片作为一个模块，成为GSM/GPRS modem，提供AT命令接口。网络的应用已经是一个需求的热点，而目前的GSM模块大多都能支持GPRS功能。应用处理器通过AT命令集与带GPRS功能的无线通讯模块通信，为实现网络的应用提供了一个最底层的支持。

从[软件](http://baike.baidu.com/view/37.htm)的角度来看，RIL(Radio Interface Layer)工作在PPP、[TCP/IP协议](http://baike.baidu.com/view/7649.htm)之下，负责数据的[可靠传输](http://baike.baidu.com/view/1832003.htm)、AT命令的发送以及response的解析。当然，除了对网络的支持，RIL也支持SMS、Voice Call等功能。从这一点来看，RIL的性能好坏影响着所有无线通信应用相关的[软件](http://baike.baidu.com/view/37.htm)，而软件设计的合理性又影响着RIL的性能。

**Windows Mobile上的RIL**

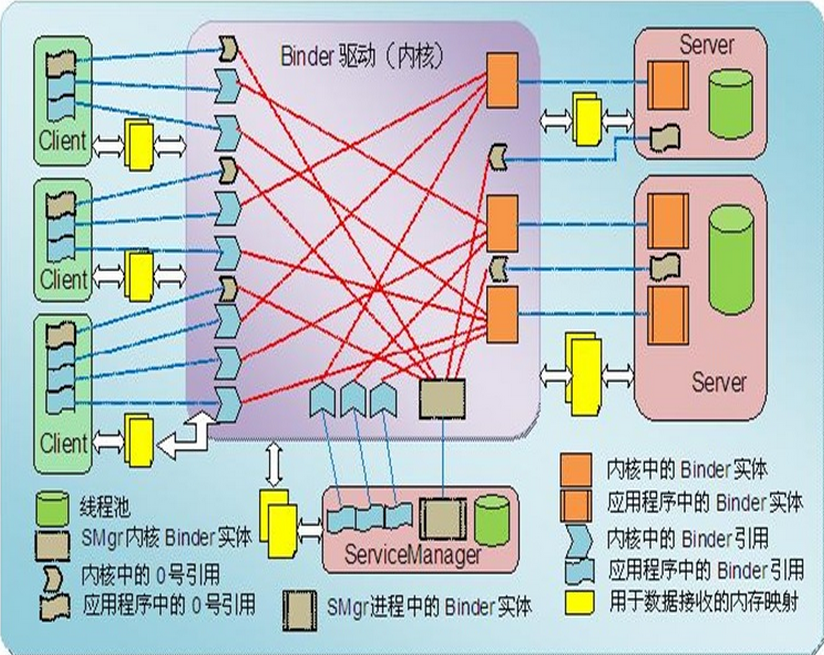
Windows Mobile是用在智能手机上的主流[操作系统](http://baike.baidu.com/view/880.htm)之一。[微软](http://baike.baidu.com/view/2353.htm)是这样介绍RIL的：作为Windows Mobile的一个非常重要的组件，RIL使各种无线语音和数据应用成为可能。运行在Windows Mobile上的[软件](http://baike.baidu.com/view/37.htm)可以通过RIL无缝地与GSM/GPRS或者CDMA2000 1X modem通信。RIL的位置处于无线基带系统的协议栈之上，Windows Mobile的CellCore层之下。由于RIL隐藏了硬件上的一些细节，OEM[厂商](http://baike.baidu.com/view/2369144.htm)可以根据自己的需要将不同型号的无线modem集成到它们的产品之中。

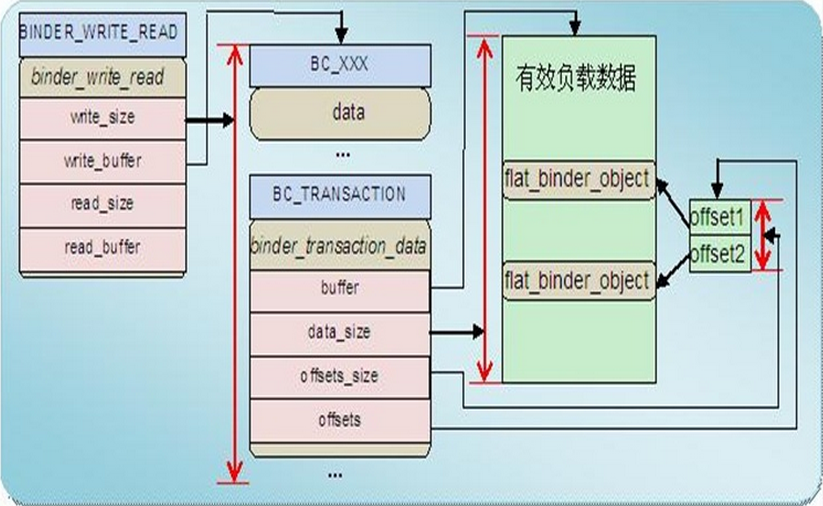
**Android上的RIL**

[Android](http://baike.baidu.com/view/1241829.htm)是目前最流行[智能手机操作系统](http://baike.baidu.com/view/1047304.htm)之一，Android的RIL位于[应用程序](http://baike.baidu.com/view/330120.htm)[框架](http://baike.baidu.com/view/66971.htm)与[内核](http://baike.baidu.com/view/1366.htm)之间，分成了两个部分，一个部分是rild,它负责socket与应用程序框架进行通信。另外一个部分是Vendor RIL，这个部分负责向下是通过两种方式与radio进行通信，它们是直接与radio通信的AT指令通道和用于传输包数据的通道，数据通道用于手机的[上网](http://baike.baidu.com/view/7944.htm)功能。

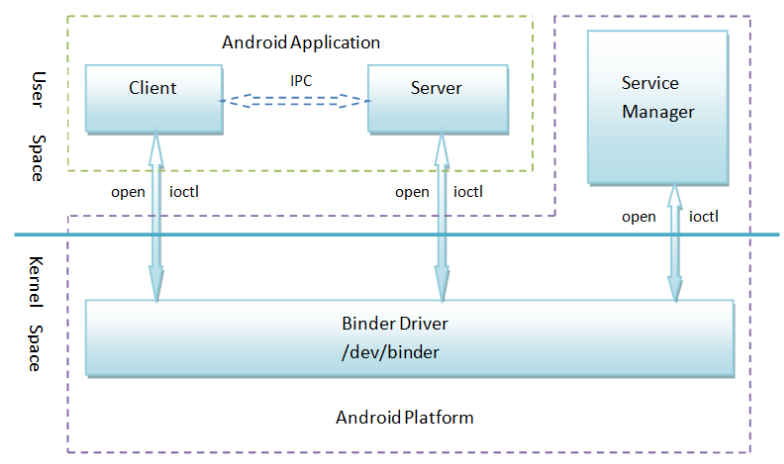
对于RIL的java框架部分，也被分成了两个部分，一个是RIL模块，这个模块主要用于与下层的rild进行通信，另外一个是Phone模块，这个模块直接暴露电话功能接口给应用开发用户，供他们调用以进行电话功能的实现。

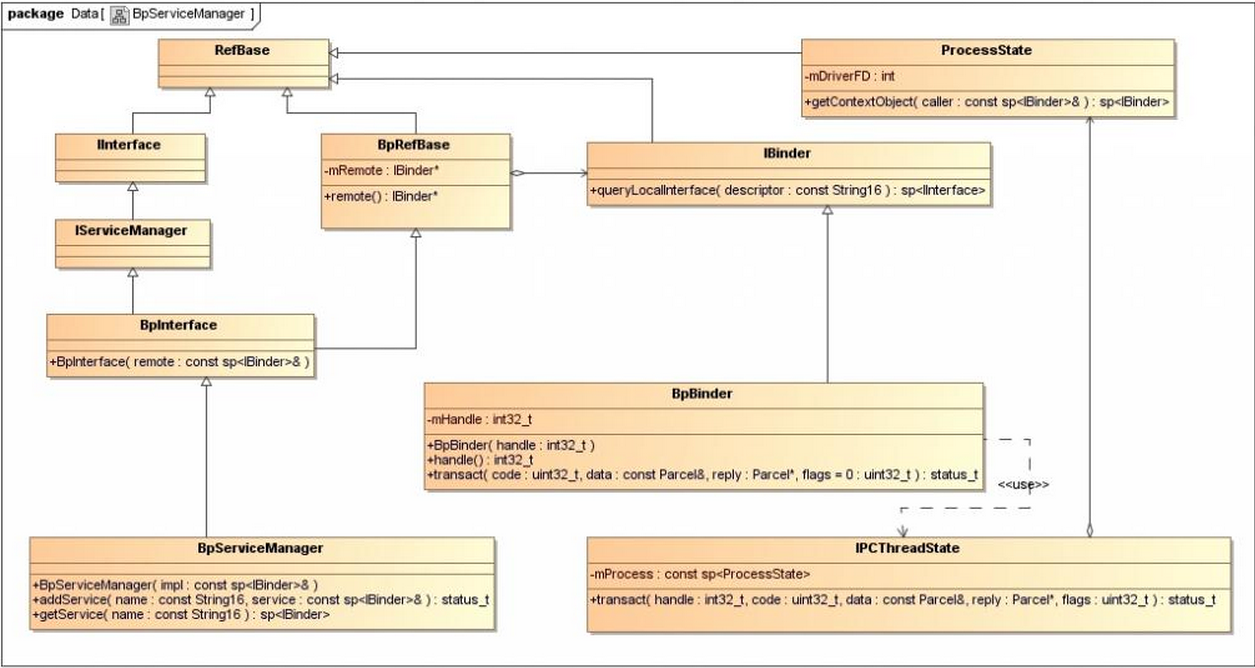
**Android Binder设计与实现**

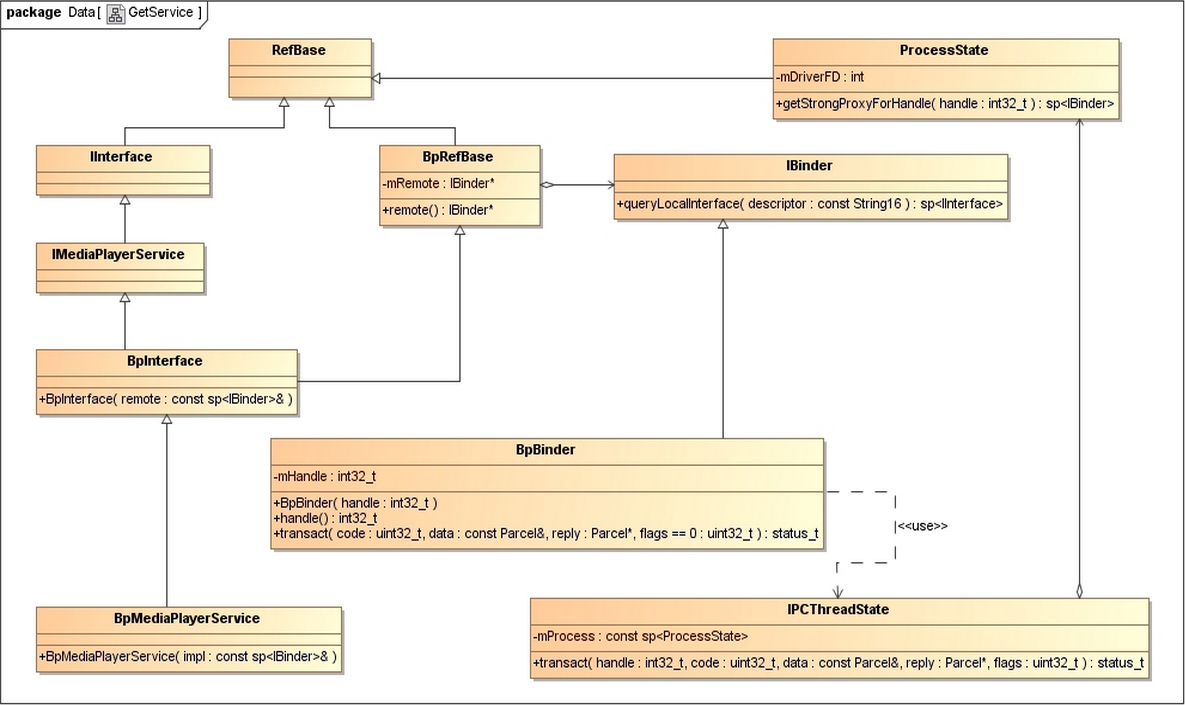


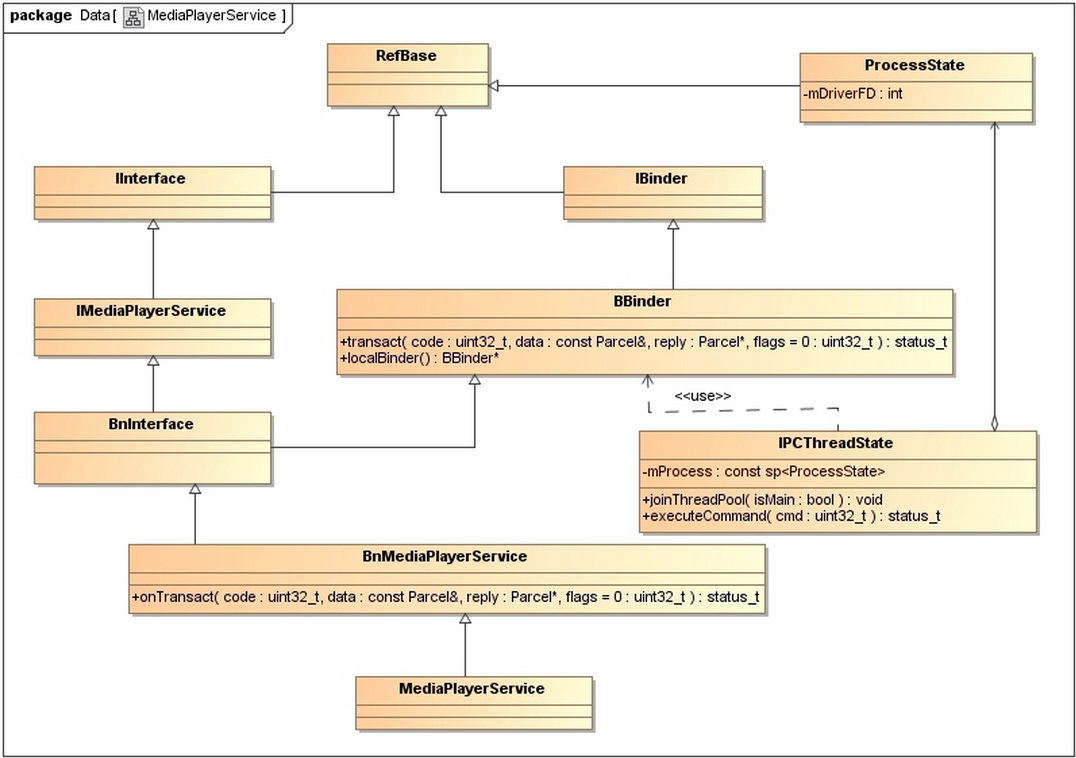


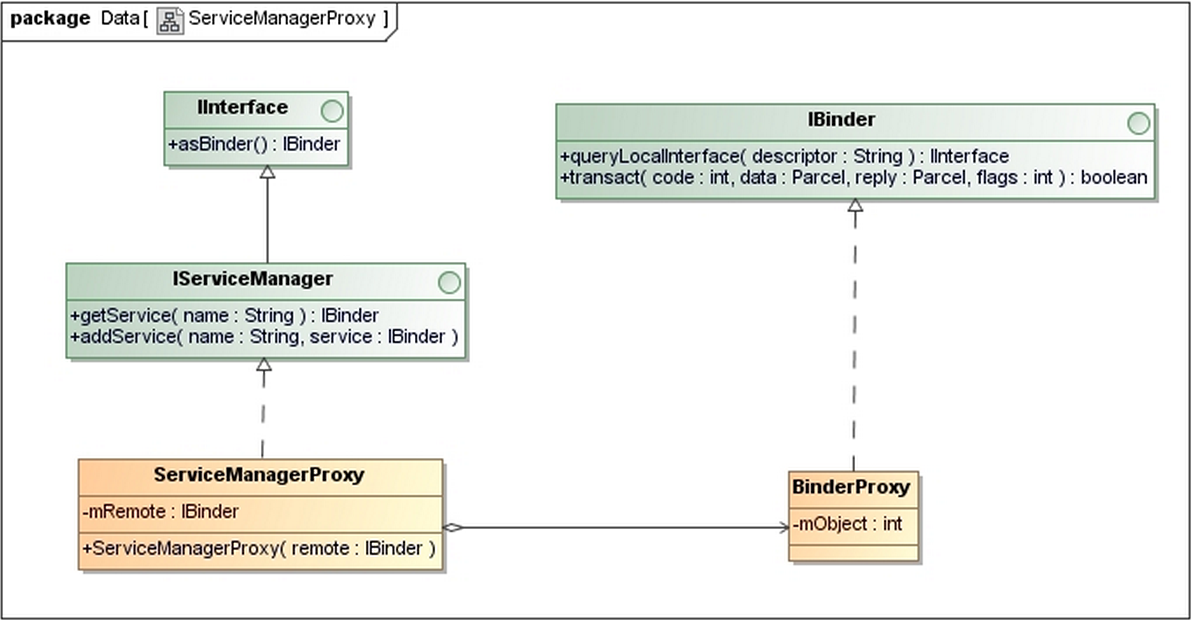
<http://disanji.net/2011/02/28/android-bnder-design/>

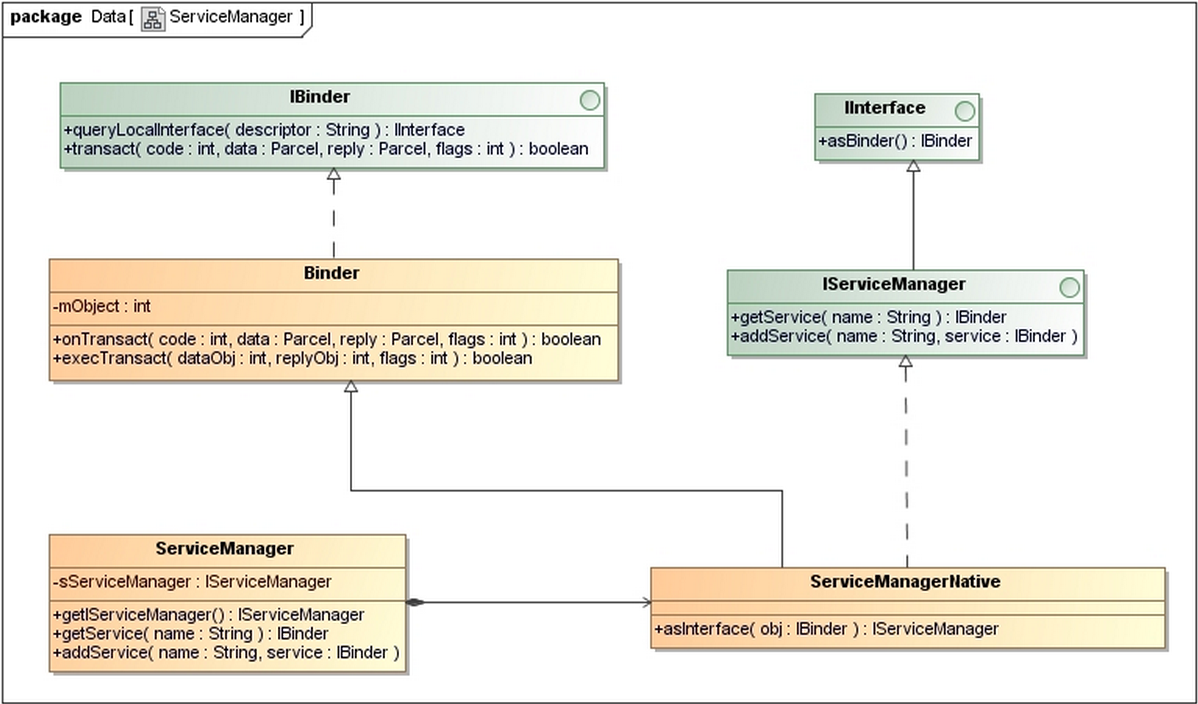












**1 获取Service Manager的Java远程接口的过程**

**ServiceManager.java**: getIServiceManager()=>

**.** **BinderInternal.java**: getContextObject()=> android\_util\_Binder.cpp: android\_os\_BinderInternal\_getContextObject()=> ProcessState.cpp: getContextObject()->getStrongProxyForHandle()->return **new** BpBinder(0)=> BinderInternal.java: javaObjectForIBinder()->**new**  BinderProxy()

**. ServiceManagerNative.java**: asInterface()->return **new** ServiceManagerProxy(**new**  BinderProxy())

**sServiceManager = new ServiceManagerProxy(new BinderProxy());**

在Java层，拥有一个Service Manager远程接口ServiceManagerProxy，而这个ServiceManagerProxy对象在JNI层有一个句柄值为0的BpBinder对象与之通过gBinderProxyOffsets关联起来。

**2 服务端(Server)：HelloService接口的定义**

**IHelloService.aidl**

1. **package** android.os;
3. **interface** IHelloService
4. {
5. **void** setVal(**int** val);
6. **int** getVal();
7. }

**IHelloService.java**

1. /\*
2. \* This file is auto-generated.  DO NOT MODIFY.
3. \* Original file: frameworks/base/core/java/android/os/IHelloService.aidl
4. \*/
5. **package** android.os;
6. **public** **interface** IHelloService **extends** android.os.IInterface
7. {
8. /\*\* Local-side IPC implementation stub class. \*/
9. **public** **static** **abstract** **class** Stub **extends** android.os.Binder **implements** android.os.IHelloService
10. {
11. **private** **static** **final** java.lang.String DESCRIPTOR = "android.os.IHelloService";
12. /\*\* Construct the stub at attach it to the interface. \*/
13. **public** Stub()
14. {
15. **this**.attachInterface(**this**, DESCRIPTOR);
16. }
18. /\*\*
19. \* Cast an IBinder object into an android.os.IHelloService interface,
20. \* generating a proxy if needed.
21. \*/
22. **public** **static** android.os.IHelloService asInterface(android.os.IBinder obj)
23. {
24. **if** ((obj==**null**)) {
25. **return** **null**;
26. }
27. android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
28. **if** (((iin!=**null**)&&(iin **instanceof** android.os.IHelloService))) {
29. **return** ((android.os.IHelloService)iin);
30. }
31. **return** **new** android.os.IHelloService.Stub.Proxy(obj);
32. }
34. **public** android.os.IBinder asBinder()
35. {
36. **return** **this**;
37. }
39. @Override
40. **public** **boolean** onTransact(**int** code, android.os.Parcel data, android.os.Parcel reply, **int** flags) **throws** android.os.RemoteException
41. {
42. **switch** (code)
43. {
44. **case** INTERFACE\_TRANSACTION:
45. {
46. reply.writeString(DESCRIPTOR);
47. **return** **true**;
48. }
49. **case** TRANSACTION\_setVal:
50. {
51. data.enforceInterface(DESCRIPTOR);
52. **int** \_arg0;
53. \_arg0 = data.readInt();
54. **this**.setVal(\_arg0);
55. reply.writeNoException();
56. **return** **true**;
57. }
58. **case** TRANSACTION\_getVal:
59. {
60. data.enforceInterface(DESCRIPTOR);
61. **int** \_result = **this**.getVal();
62. reply.writeNoException();
63. reply.writeInt(\_result);
64. **return** **true**;
65. }
66. }
67. **return** **super**.onTransact(code, data, reply, flags);
68. }
70. **private** **static** **class** Proxy **implements** android.os.IHelloService
71. {
72. **private** android.os.IBinder mRemote;
74. Proxy(android.os.IBinder remote)
75. {
76. mRemote = remote;
77. }
79. **public** android.os.IBinder asBinder()
80. {
81. **return** mRemote;
82. }
84. **public** java.lang.String getInterfaceDescriptor()
85. {
86. **return** DESCRIPTOR;
87. }
89. **public** **void** setVal(**int** val) **throws** android.os.RemoteException
90. {
91. android.os.Parcel \_data = android.os.Parcel.obtain();
92. android.os.Parcel \_reply = android.os.Parcel.obtain();
93. **try** {
94. \_data.writeInterfaceToken(DESCRIPTOR);
95. \_data.writeInt(val);
96. mRemote.transact(Stub.TRANSACTION\_setVal, \_data, \_reply, 0);
97. \_reply.readException();
98. }
99. **finally** {
100. \_reply.recycle();
101. \_data.recycle();
102. }
103. }
105. **public** **int** getVal() **throws** android.os.RemoteException
106. {
107. android.os.Parcel \_data = android.os.Parcel.obtain();
108. android.os.Parcel \_reply = android.os.Parcel.obtain();
109. **int** \_result;
110. **try** {
111. \_data.writeInterfaceToken(DESCRIPTOR);
112. mRemote.transact(Stub.TRANSACTION\_getVal, \_data, \_reply, 0);
113. \_reply.readException();
114. \_result = \_reply.readInt();
115. }
116. **finally** {
117. \_reply.recycle();
118. \_data.recycle();
119. }
120. **return** \_result;
121. }
122. }
124. **static** **final** **int** TRANSACTION\_setVal = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);
125. **static** **final** **int** TRANSACTION\_getVal = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 1);
126. }
128. **public** **void** setVal(**int** val) **throws** android.os.RemoteException;
129. **public** **int** getVal() **throws** android.os.RemoteException;

根据IHelloService接口的定义生成相应的Stub和Proxy类，实现这个HelloService的Server必须继承于这里的IHelloService.Stub类，而这个HelloService的远程接口就是这里的IHelloService.Stub.Proxy对象获得的IHelloService接口。

**3 HelloService的启动过程Client**

**4获取HelloService的Java远程接口的过程**

1. **public** **class** Hello **extends** Activity **implements** OnClickListener {
2. ......
4. **private** IHelloService helloService = **null**;
6. ......
8. @Override
9. **public** **void** onCreate(Bundle savedInstanceState) {
11. helloService = IHelloService.Stub.asInterface(
12. ServiceManager.getService("hello"));
13. }
15. ......
16. }

**. ServiceManager.java:** getService() => **ServiceManagerProxy.java:** getService() => **BinderProxy.java:** transact() =>**android\_util\_Binder.cpp:** android\_os\_BinderProxy\_transact() -> android\_os\_Parcel\_readStrongBinder() -> **return** javaObjectForIBinder(env, **new** BpBinder(handle)); -> **new** BinderProxy()

**. IHelloService.java: return** **new** android.os.IHelloService.Stub.Proxy(**new** BinderProxy());

获得了HelloService的远程接口了，它实质上是一个实现了IHelloService接口的IHelloService.Stub.Proxy对象

Android平台输入法

<http://wenku.baidu.com/view/80de2573a417866fb84a8efc.html>

# Android中的Audio播放：音量和远程播放控制

|  |  |  |
| --- | --- | --- |
| [日期：2012-04-03] | 来源：Linux社区  作者：田海立 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

本文从Audio控制使用的角度，讲解了AudioStream的分类，硬件音量控制键调整对相应AudioStream音量大小的影响，以及如何响应远程MediaButton控制Audio播放。

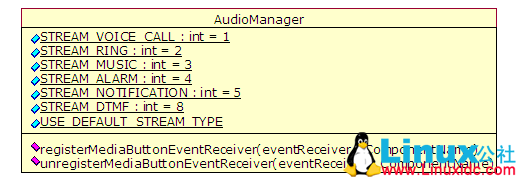
相关阅读：

[Android](http://www.linuxidc.com/topicnews.aspx?tid=11)中的Audio播放：控制Audio输出通道切换 <http://www.linuxidc.com/Linux/2012-04/57901.htm>

Android中的Audio播放：竞争Audio之Audio Focus的应用 <http://www.linuxidc.com/Linux/2012-04/57902.htm>

**1. Audio Stream**

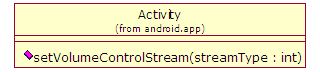
Android为不同的应用场合定义了不同的**Audio Stream**: Voice Call, Ring, Music,Alarm, Notification, DTMF。 这些AudioStream是相互独立的，所以也有各自的音量。AudioStream的定义在android.media.AudioManager中，如下图所示：

****

**2. 硬件音量控制键Vol+/-控制Audio Stream的音量**

用户按下音量控制的HardKey，希望能调出音量调整的界面。

缺省情况下，按下音量控制的硬件控制键Vol+/-，调节的是当前被激活的（Active）AudioStream的音量，如果你的程序当前没有正在播放任何声音，按下Vol+/-调节的是来电铃声的音量。【笔者注：基本是翻译的原话，需要明确！By default, pressing the volume controls modifythe volume of the active audio stream. If your app isn't currently playing anything, hitting the volume keys adjusts the ringer volume.】



在某一个程序运行时，希望按下Vol+/-调节的是当前所使用的AudioStream的音量，Android在Activity中提供了setVolumeControlStream()方法用来指定你的应用程序使用的Audio Stream类型。所以，如果你的程序用到Audio的播放，你首先要知道你的程序所用的Audio Stream类型，并在**onCreate()中调用setVolumeControlStream()来设定Audio Stream的类型**。

Q：**setVolumeControlStream()**之后就起效，还是可见之后才有用？后台播放呢？

要明确这些标红的地方，需要看Android的内部实现！//TODO：明确化；内部实现，另外专题写…

**3. 遥控Audio Playback**

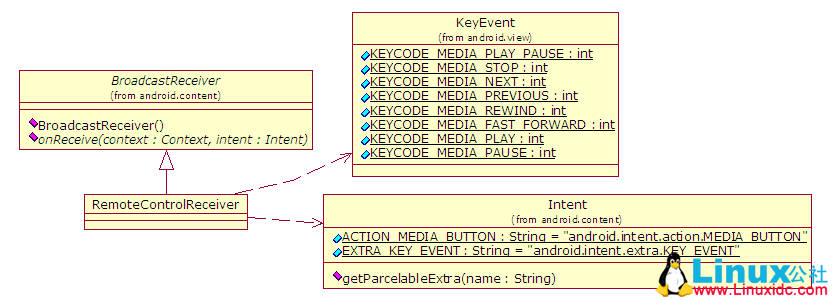
有些耳机上有诸如音量控制、切换前一首下一首歌、播放/暂停等控制键，Bluetooth的AVRCP Profile也能远程控制的。这些键被按下后，Android是通过broadcast [ACTION\_MEDIA\_BUTTON](http://developer.android.com/reference/android/content/Intent.html#ACTION_MEDIA_BUTTON)这个Intent发出去的。

所以，要在你的应用中处理这些按键，只要侦听这个广播，并处理即可。

如果知道什么时候开始侦听广播，程序中动态注册/注销侦听都是很好的选择（与，写在AndroidManifest中，apk加载时就注册相比）。而Audio控制在什么时候才处理这些键值是比较明确的，一般获得AudioFocus的情况下，响应[ACTION\_MEDIA\_BUTTON](http://developer.android.com/reference/android/content/Intent.html#ACTION_MEDIA_BUTTON)广播；失去Audio Focus的情况下，不响应[ACTION\_MEDIA\_BUTTON](http://developer.android.com/reference/android/content/Intent.html#ACTION_MEDIA_BUTTON)广播，这也就是相应registerMediaButtonEventReceiver()/unregisterMediaButtonEventReceiver()的最佳时机。

[ACTION\_MEDIA\_BUTTON](http://developer.android.com/reference/android/content/Intent.html#ACTION_MEDIA_BUTTON)广播的处理，只要在override onReceive()，并在其中通过判断是否Intent.ACTION\_MEDIA\_BUTTON确保是这个广播，从Intent.EXTA\_KEY\_EVENT中获得KeyEvent，做相应的处理即可。

涉及的几个类的关系图如下：



相应处理的代码片段如下：

1. **public** **class** RemoteControlReceiver **extends** BroadcastReceiver {
2. @Override
3. **public** **void** onReceive(Context context, Intent intent) {
4. **if** (Intent.ACTION\_MEDIA\_BUTTON.equals(intent.getAction())) {
5. KeyEvent event = (KeyEvent)intent.getParcelableExtra(Intent.EXTRA\_KEY\_EVENT);
6. **if** (KeyEvent.KEYCODE\_MEDIA\_PLAY == event.getKeyCode()) {
7. // Handle key press.
8. }
9. }
10. }
11. }

**总结一下要点：**

1.      Android的Audio控制是按照Audio Stream划分的；

2.      各个Audio Stream的音量是独立的。推荐在onCreate()中通过Activity.setVolumeControlStream()方法设置所使用AudioStream的类型，实现按下硬键Vol+/-调整的是相对应AudioStream的音量；

3.      侦听Intent.[ACTION\_MEDIA\_BUTTON](http://developer.android.com/reference/android/content/Intent.html#ACTION_MEDIA_BUTTON)广播实现对远程Media控制的处理。

# Android中的Audio播放：竞争Audio之Audio Focus的应用

|  |  |  |
| --- | --- | --- |
| [日期：2012-04-03] | 来源：Linux社区  作者：田海立 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

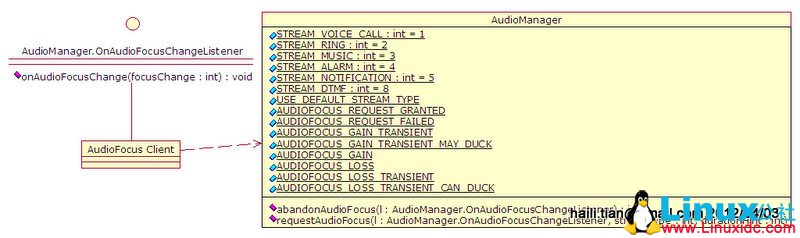
[Android](http://www.linuxidc.com/topicnews.aspx?tid=11)是多任务系统，Audio系统是竞争资源。Android2.2之前，没有内建的机制来解决多个程序竞争Audio的问题，2.2引入了称作AudioFocus的机制来管理对Audio资源的竞争的管理与协调。本文主要讲解AudioFocus的使用。

按照AudioFocus的机制，在使用AudioStream之前，需要申请AudioFocus，在获得AudioFocus之后才可以使用相应的AudioStream；如果有别的程序竞争你正在使用的AudioStream，你的程序需要在收到通知之后做停止播放或者降低声音的处理。值得指出的是，这种机制是需要合作完成的，需要所有使用Audio资源的程序都按照这种机制来做，而如果有程序在它失去AudioFocus的时候仍然在使用Audio，AudioFocus拿它也没办法。而这一点对于开放系统的Android来说很致命的：用户可能安装没遵守这种机制的程序，或者版本太老还没引入这种机制的程序，这最终会导致很差的用户体验。

对于手机方案公司来说，要做的能做的事情就是教育和培训团队成员以保证自己内建的程序遵守机制没问题，这包括了Android原生的程序、自己开发的程序，以及适配第三方的程序。

**一、AudioFocus的申请与释放**

下面看与AudioFocus的相关的类：



获取/放弃AudioFocus的方法都在android.media.AudioManager中，获取AudioFocus用[requestAudioFocus()](http://developer.android.com/reference/android/media/AudioManager.html#requestAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener, int, int))；用完之后，放弃AudioFocus，用[abandonAudioFocus()](http://developer.android.com/reference/android/media/AudioManager.html#abandonAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener))。

其中，**参数**：

* **streamType**是《[Android中的Audio播放：音量和远程播放控制](http://www.linuxidc.com/Linux/2012-04/57903.htm)》中说明的AudioStream，其值取决于AudioManager中的STREAM\_xxx；
* **durationHint**是持续性的指示：
  + [AUDIOFOCUS\_GAIN](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_GAIN)指示申请得到的Audio Focus不知道会持续多久，一般是长期占有；
  + [AUDIOFOCUS\_GAIN\_TRANSIENT](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_GAIN_TRANSIENT)指示要申请的AudioFocus是暂时性的，会很快用完释放的；
  + [AUDIOFOCUS\_GAIN\_TRANSIENT\_MAY\_DUCK](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK)不但说要申请的AudioFocus是暂时性的，还指示当前正在使用AudioFocus的可以继续播放，只是要“duck”一下（降低音量）。
* [**AudioManager.OnAudioFocusChangeListener**](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html)是申请成功之后监听AudioFocus使用情况的Listener，后续如果有别的程序要竞争AudioFocus，都是通过这个Listener的[onAudioFocusChange()](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int))方法来通知这个Audio Focus的使用者的。

**返回值**，可能是：

* [AUDIOFOCUS\_REQUEST\_GRANTED](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_REQUEST_GRANTED)：申请成功；
* [AUDIOFOCUS\_REQUEST\_FAILED](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_REQUEST_FAILED)：申请失败。

**二、AudioFocus被抢占与重新获得**

由上节中知道，申请/释放AudioFocus时传入了[**AudioManager.OnAudioFocusChangeListener**](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html)这个参数，其[onAudioFocusChange()](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int))方法是Audio Focus被抢占与再次获得通知的地方。所以，每个要使用AudioFocus的程序都要小心实现这个函数，保证AudioFocus实现的一致性。

[onAudioFocusChange()](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int))方法的focusChange参数指示了该AudioFocus的竞争者对AudioFocus的拥有情况，取值如下：

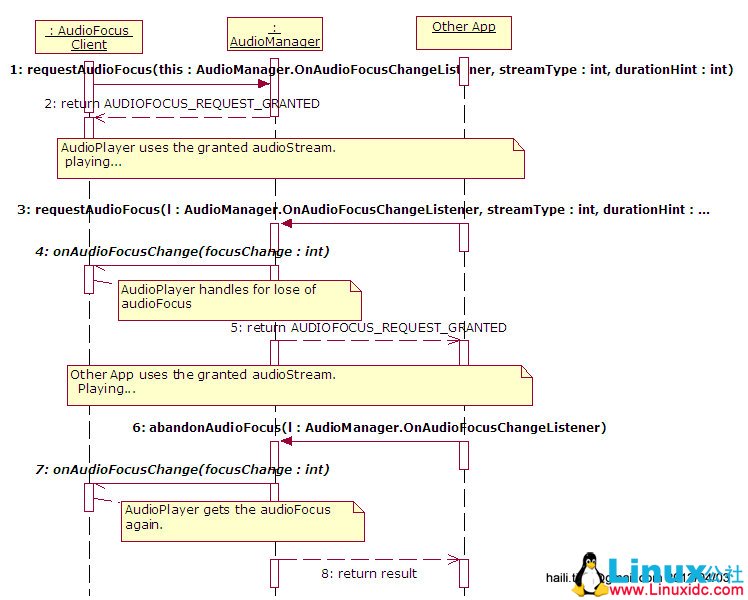
* [AUDIOFOCUS\_GAIN](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_GAIN)：获得了Audio Focus；
* [AUDIOFOCUS\_LOSS](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_LOSS)：失去了Audio Focus，并将会持续很长的时间。这里因为可能会停掉很长时间，所以不仅仅要停止Audio的播放，最好直接释放掉Media资源。而因为停止播放Audio的时间会很长，如果程序因为这个原因而失去AudioFocus，最好不要让它再次自动获得AudioFocus而继续播放，不然突然冒出来的声音会让用户感觉莫名其妙，感受很不好。这里直接放弃AudioFocus，当然也不用再侦听远程播放控制【如下面代码的处理】。要再次播放，除非用户再在界面上点击开始播放，才重新初始化Media，进行播放。
* [AUDIOFOCUS\_LOSS\_TRANSIENT](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_LOSS_TRANSIENT)：暂时失去Audio Focus，并会很快再次获得。必须停止Audio的播放，但是因为可能会很快再次获得AudioFocus，这里可以不释放Media资源；
* [AUDIOFOCUS\_LOSS\_TRANSIENT\_CAN\_DUCK](http://developer.android.com/reference/android/media/AudioManager.html#AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK)：暂时失去AudioFocus，但是可以继续播放，不过要在降低音量。

下面是[onAudioFocusChange()](http://developer.android.com/reference/android/media/AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange(int))方法处理的代码片段：

1. OnAudioFocusChangeListener afChangeListener = **new** OnAudioFocusChangeListener() {
2. **public** **void** onAudioFocusChange(**int** focusChange) {
3. **if** (focusChange == AudioManager.AUDIOFOCUS\_LOSS\_TRANSIENT
4. // Pause playback
5. } **else** **if** (focusChange == AudioManager.AUDIOFOCUS\_LOSS) {
6. am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
7. am.abandonAudioFocus(afChangeListener);
8. // Stop playback
9. } **else** **if** (focusChange == AUDIOFOCUS\_LOSS\_TRANSIENT\_CAN\_DUCK) {
10. // Lower the volume
11. } **else** **if** (focusChange == AudioManager.AUDIOFOCUS\_GAIN) {
12. // Resume playback or Raise it back to normal
13. }
14. }
15. };

**三、典型的应用AudioFocus的场景**

下面的时序图描述了AudioFocus被抢占与再次获取的典型场景：



**Audio Focus被抢占与再次获取的时序图**

**注意**：为了描述简单，此图中除了两个竞争Audio Focus的App之外，只用AudioManager表征了Android的AudioFocus机制中内部参与的对象，实际AudioManager只是外部的表象，内部参与的对象很多，回调函数也并非简单的直接由AudioManager调用，其中还包含了复杂的IPC机制。

图中：

* AudioFocus Client通过requestAudioFocus()获取AudioFocus，在获得AudioFocus之后，开始播放Audio[Step#1 ~ #2]；
* 其它程序（Other App）也通过requestAudioFocus()获取同类AudioStream的AudioFocus [Step#3]
* AudioFocus Client失去了Audio Focus，在onAudioFocusChanged()中，根据focusChange的值，做第二节中的处理[Step#4]；
* 其它程序（Other App）获取Audio Focus之后，开始播放Audio[Step#5]；
* 其它程序（Other App）使用Audio之后，通过[abandonAudioFocus()](http://developer.android.com/reference/android/media/AudioManager.html#abandonAudioFocus(android.media.AudioManager.OnAudioFocusChangeListener))归还AudioFocus [Step#6]；
* AudioFocus Client重新获得了Audio Focus，可做进一步的处理 [Step#7]

**小结**

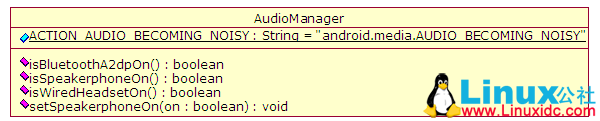
Audio Focus机制要参与各方充分理解并统一遵照施行，有没有遵照者或者实现有误的程序存在就可能打破这一机制，带来糟糕的用户体验。在保证Built-in程序没问题的前提下，如果进入AndroidMarket之前的程序都严格执行了AudioFocus相关的测试，应该也没问题。

# Android中的Audio播放：控制Audio输出通道切换

|  |  |  |
| --- | --- | --- |
| [日期：2012-04-03] | 来源：Linux社区  作者：田海立 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

Audio 输出通道有很多，Speaker、headset、bluetooth A2DP等。通话或播放音乐等使用Audio输出过程中，可能发生Audio输出通道的切换。比如，插入有线耳机播放音乐时，声音是从耳机发出的；而此时拔出耳机，Audio输出通道会发生切换。如果音乐播放器不做处理，Audio输出是被切换到扬声器的，声音直接从Speaker发出。我们在编写程序时，要捕获并按照需求来处理这样的事，本文就是讲解如何处理的。

[Android](http://www.linuxidc.com/topicnews.aspx?tid=11)中可以通过android.media.AudioManager查询当前Audio输出的情况，并且在Audio输出发生变化时，捕获并处理这种变化。



**一、Audio输出状态查询与控制**

android.media.AudioManager提供的下列方法可以用来查询当前Audio输出的状态：

* **isBluetoothA2dpOn()**：检查A2DPAudio是否通过蓝牙耳机；
* **isSpeakerphoneOn()**：检查扬声器是否打开；
* **isWiredHeadsetOn()**：检查线控耳机是否连着；注意这个方法只是用来判断耳机是否是插入状态，并不能用它的结果来判定当前的Audio是通过耳机输出的，这还依赖于其他条件。

另外还有一些设置这些Audio输出的setXYZ()方法，这些方法在一般使用Audio输出的应用程序不要直接调用，他们由系统来管理，实现Audio输出通道的自动切换。除非，界面提供给用户切换的菜单或按钮，而用户选择了却换，比如要直接选择扬声器发声，可直接调用setSpeakerphoneOn()。

**二、Audio输出通道切换的事件的捕获与处理**

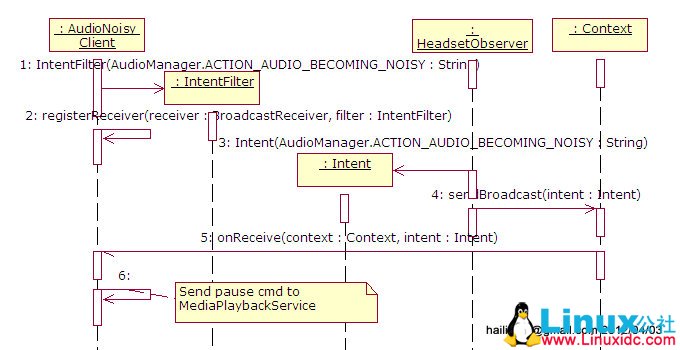
因为耳机插拔、蓝牙耳机的断开，Audio输出通路会自动切换。此时正在播放Audio的程序要获得通知，知道这一事件的发生。Android中是通过广播[ACTION\_AUDIO\_BECOMING\_NOISY](http://developer.android.com/reference/android/media/AudioManager.html#ACTION_AUDIO_BECOMING_NOISY)这个Intent通知的。

处理广播的较好的方式，是动态注册/注销自己所关心的广播。下面代码演示了，开始播放时注册广播的Receiver；停止播放时注销广播的Receiver。对Audio输出通道切换的处理是暂停当前的播放，不直接从新的通道里发出声来。

1. **private** **class** NoisyAudioStreamReceiver **extends** BroadcastReceiver {
2. @Override
3. **public** **void** onReceive(Context context, Intent intent) {
4. **if** (AudioManager.ACTION\_AUDIO\_BECOMING\_NOISY.equals(intent.getAction())) {
5. // Pause the playback
6. }
7. }
8. }
10. **private** IntentFilter intentFilter = **new** IntentFilter(AudioManager.ACTION\_AUDIO\_BECOMING\_NOISY);
12. **private** **void** startPlayback() {
13. registerReceiver(myNoisyAudioStreamReceiver(), intentFilter);
14. }
16. **private** **void** stopPlayback() {
17. unregisterReceiver(myNoisyAudioStreamReceiver);
18. }

**三、Audio输出通道切换的典型场景—— 用耳机听音乐时，拔出耳机**

听耳机听音乐时，耳机别拔出的时序图如下：



图中：

* AudioNoisy Client注册了侦听广播AudioManager.ACTION\_AUDIO\_BECOMING\_NOISY[Step#1 ~ #2]；
* 用耳机一直在听音乐；
* HeadsetObserver一直在监视耳机状态的变化。检测到耳机被拔出之后，发出广播AudioManager.ACTION\_AUDIO\_BECOMING\_NOISY[Step#3~4]；
* AudioNoisy Client收到了广播，发送暂停命令给MediaPaybackService去暂停当前的播放 [Step#5~6]。

**小结**

        Audio 输出通道切换时，要根据具体需求来做相应的处理。