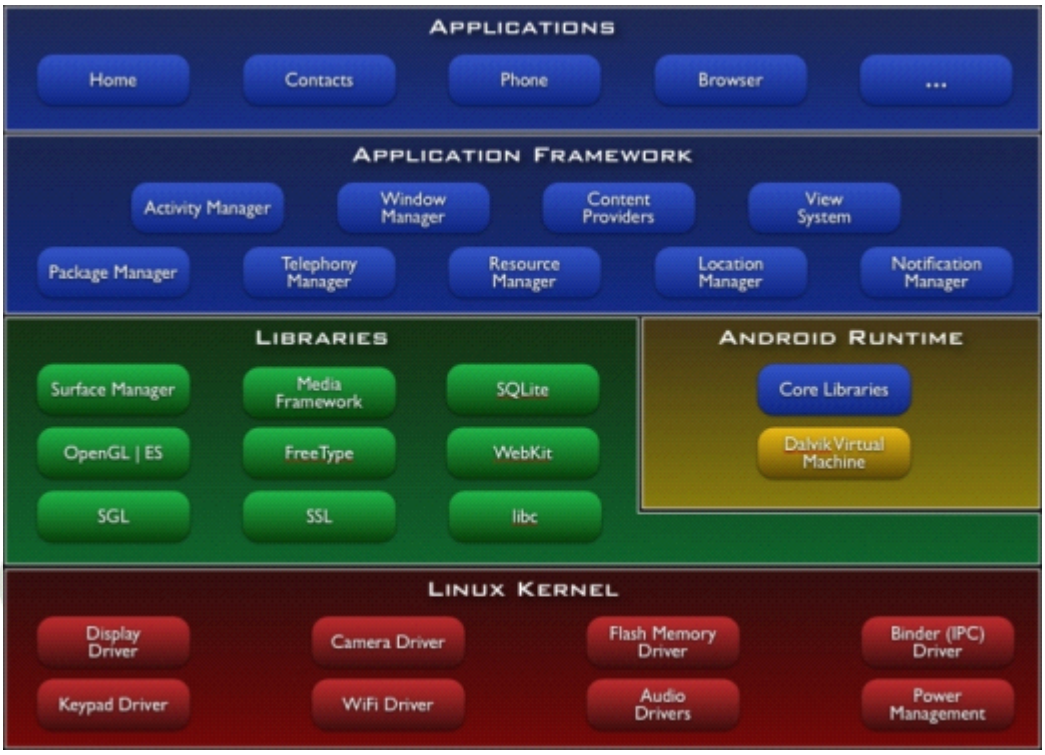
目录

【Android 多媒体框架】

1.1 Android 系统整体架构：

我们先看一下多媒体框架在整个 Android 系统所处的位置：



从框架图可以看出 Media Framework 处于 Libraries 这一层，这层的 Library 不是用 Java 实现，一般是 C/C++实现，它们通过 Java 的 JNI 方式调用。

1.2 多媒体架构：

基于第三方 PacketVideo 公司的 OpenCORE platform 来实现，支持所有通用的音频，视频，静态图像格式。CODEC(编解码器)使用 OpenMAX 1L interface 接口进行扩展，可以方便得支持 hardware / software codec plug-ins。支持的格式包括：MPEG4、H.264、MP3、AAC、AMR、JPG、PNG、GIF 等。

- Open Core 多媒体框架有一套通用可扩展的接口，针对第三方的多媒体遍解码器、输入、输出设备等等。
- 多媒体文件的播放，下载，包括 3GPP, MPEG-4,AAC and MP3 containers
- 流媒体文件的下载，实时播放，包括：3GPP, HTTP and RTSP/RTP
- 动态视频和静态图像的编码，解码，例如：MPEG-4, H.263 and AVC (H.264), JPEG
- 语音编码格式： AMR-NB and AMR-WB
- 音乐编码格式： MP3, AAC, AAC+
- 视频和图像格式： 3GPP, MPEG-4 and JPEG
- 视频会议：基于 H324-M standard

Anroid2.3 系统在多媒体方面则会提供全新的音效 API，并且还支持 VP8、WebM 格式多媒体文件，并且还可以用 AAC、AMR wideband 来录下更高质素的声音文件。

1.3 OpenCore介绍：

Open Core是Android 多媒体框架的核心，所有Android平台的音视频采集，播放的操作都是通过它来实现。

OpenCore的另外一个常用的称呼是PacketVideo。事实上，PacketVideo是一家公司的名称，而OpenCore是这套多媒体框架的软件层的名称。在Android的开发者中间，二者的含义基本相同。

通过Open Core程序员可以方便快速的开发出想要的多媒体应用程序，例如：音视频的采集，回放，视频会议，实时的流媒体播放等等应用。

对比Android的其它程序库，OpenCore的代码非常庞大，它是一个基于C++的实现，定义了全功能的操作系统移植层，各种基本的功能均被封装成类的形式，各层次之间的接口多使用继承等方式。

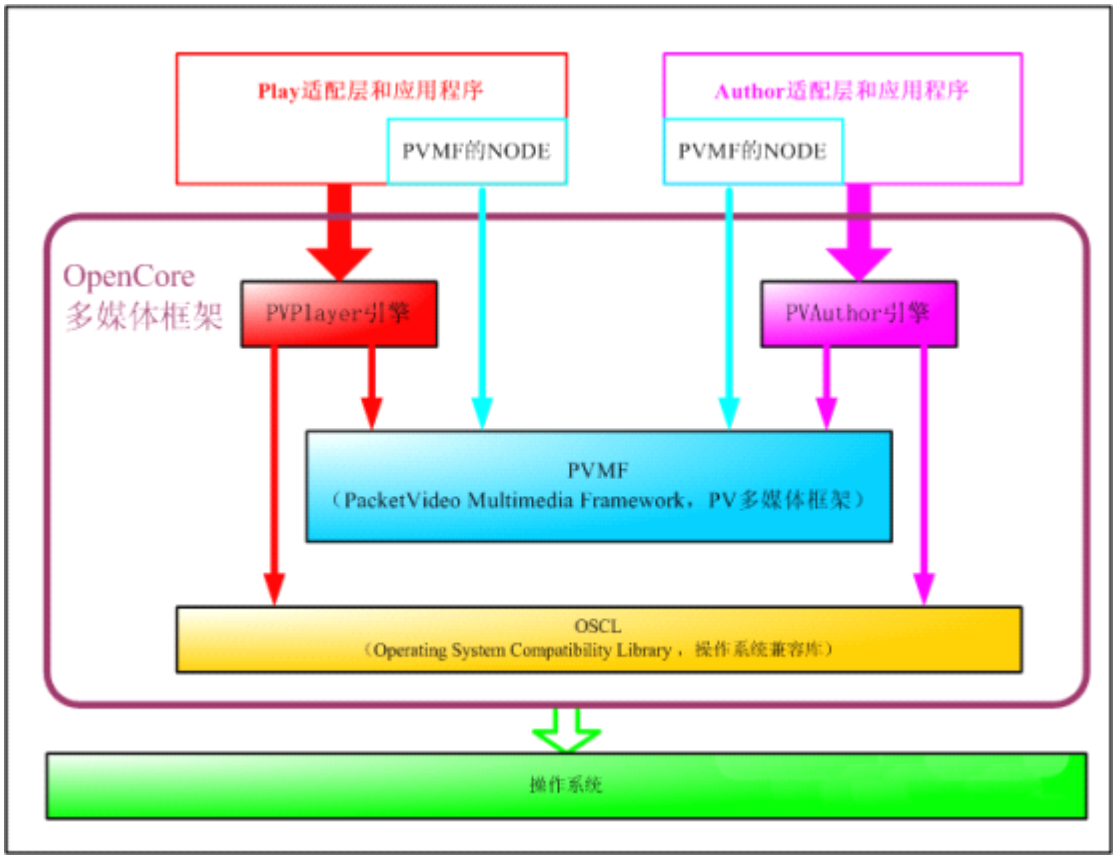
OpenCore是一个多媒体的框架，从宏观上来看，它主要包含了两大方面的内容：

PVPlayer： 提供媒体播放器的功能，完成各种音频(Audio)、视频(Video)流的回放(Playback)功能.

PVAuthor： 提供媒体流记录的功能，完成各种音频(Audio)、视频(Video)流的以及静态图像捕获功能.

PVPlayer和PVAuthor以SDK的形式提供给开发者，可以在这个SDK之上构建多种应用程序和服务。在移动终端中常常使用的多媒体应用程序，例如媒体播放器、照相机、录像机、录音机等等。

为了更好的组织整体的架构，OpenCore在软件层次在宏观上分成几个层次：



OSCL： Operating System Compatibility Library (操作系统兼容库)，包含了一些操作系统底层的操作，为了更好地在不同操作进行系统移植。包含了基本数据类型、配置、字符串工具、IO、错误处理、线程等内容，类似一个基础的C++库。

PVMF： PacketVideo Multimedia Framework(PV 多媒体框架)，在框架内实现一个文件解析(parser)和组成(composer)、编解码的NODE，也可以继承其通用的接口，在用户层实现一些NODE。

PVPlayer Engine： PVPlayer 引擎。
PVAuthor Engine： PVAuthor 引擎。

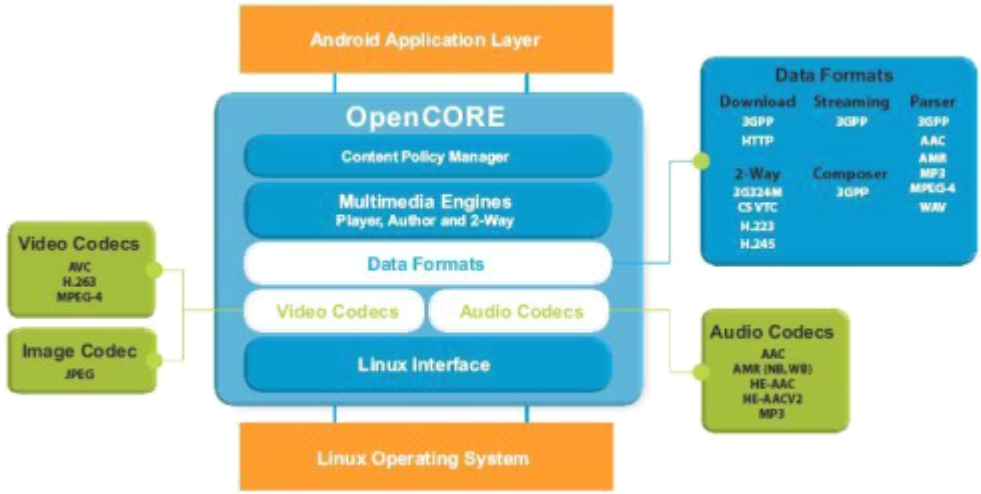
事实上，OpenCore 中包含的内容非常多：从播放的角度，PVPlayer 的输入的(Source)是文件或者网络媒体流，输出(Sink)是音频视频的输出设备，其基本功能包含了媒体流控制、文件解析、音频视频流的解码(Decode)等方面的内容。

除了从文件中播放媒体文件之外，还包含了与网络相关的RTSP流(Real Time Stream Protocol，实时流协议)。在媒体流记录的方面，PVAuthor 输入的(Source)是照相机、麦克风等设备，输出的(Sink)是各种文件，包含了流的同步、音频视频流的编码(Encode)以及文件的写入等功能。

在使用 OpenCore 的 SDK 的时候，有可能需要在应用程序层实现一个适配器(Adaptor)，然后在适配器之上实现具体的功能，对于 PVMF 的 NODE 也可以基于通用的接口，在上层实现，以插件的形式使用。

【OpenCore 的代码结构】

2.1 OpenCore 代码结构



以开源 Android 的代码为例，Open Core 的代码在 Android 代码的 External/Opencore 目录中。这个目录是 OpenCore 的根目录，其中包含的子目录如下所示：

android： 这里面是一个上层的库，它基于 PVPlayer 和 PVAuthor 的 SDK 实现了一个为 Android 使用的 Player 和 Author。
baselibs： 包含数据结构和线程安全等内容的底层库。
codecs_v2： 这是一个内容较多的库，主要包含编解码的实现，以及一个 OpenMAX 的实现。
engines： 包含 PVPlayer 和 PVAuthor 引擎的实现。
extern_libs_v2： 包含了 khronos 的 OpenMAX 的头文件。
fileformats： 文件格式的解析(parser)工具。
nodes： 提供一些 PVMF 的 NODE，主要是编解码和文件解析方面的。
oscl： 操作系统兼容库。
pvmi： 输入输出控制的抽象接口。
protocols： 主要是与网络相关的 RTSP、RTP、HTTP 等协议的相关内容。
pvcommon： pvcommon 库文件的 Android.mk 文件，没有源文件。
pvplayer： pvplayer 库文件的 Android.mk 文件，没有源文件。
pvauthor： pvauthor 库文件的 Android.mk 文件，没有源文件。
tools_v2： 编译工具以及一些可注册的模块。

在 external/opencore/目录中还有 2 个文件，如下所示：

Android.mk： 全局的编译文件
pvplayer.conf： 配置文件

在 external/opencore/的各个子文件夹中包含了众多的 Android.mk 文件，它们之间还存在着“递归”的关系。例如根目录下的 Android.mk，就包含了如下的内容片断：

```
include $(PV_TOP)/pvcommon/Android.mk
include $(PV_TOP)/pvplayer/Android.mk
include $(PV_TOP)/pvauthor/Android.mk
```

这表示了要引用 pvcommon，pvplayer 和 pvauthor 等文件夹下面的 Android.mk 文件。

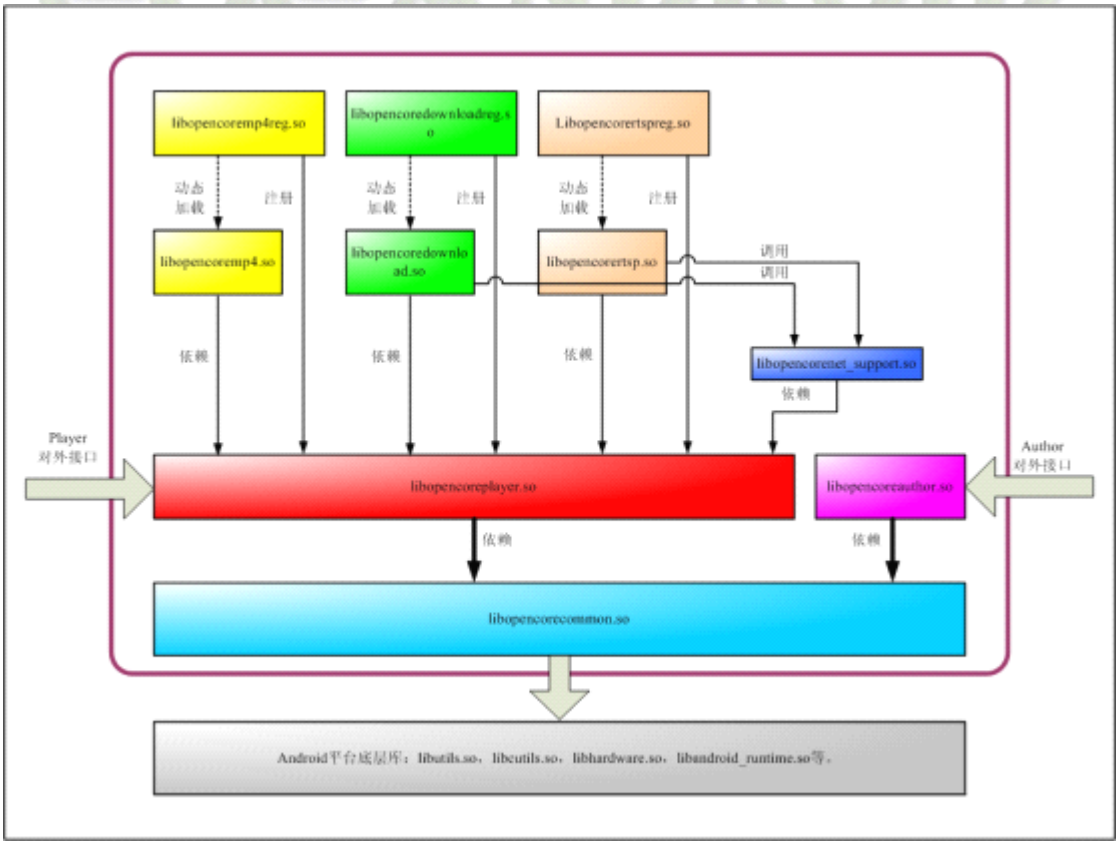
external/opencore/的各个 Android.mk 文件可以按照排列组合进行使用，将几个 Android.mk 内容合并在一个库当中。

2.2 OpenCore 的编译结构

1、库的层次关系：在 Android 的开源版本中编译出来的内容，OpenCore 编译出来的各个库如下所示：

libopencoreauthor.so： OpenCore 的 Author 库
libopencorecommon.so： OpenCore 底层的公共库
libopencoredownloadreg.so： 下载注册库
libopencoredownload.so： 下载功能实现库
libopencoremp4reg.so： MP4 注册库
libopencoremp4.so： MP4 功能实现库
libopencorenet_support.so： 网络支持库
libopencoreplayer.so： OpenCore 的 Player 库
libopencorertspreg.so： RTSP 注册库
libopencorertsp.so： RTSP 功能实现库

这些库的层次关系如下图所示：



OpenCore 的各个库之间具有如下的关系：

1) libopencorecommon.so 是所有的库的依赖库，提供了公共的功能；
2) libopencoreplayer.so 和 libopencoreauthor.so 是两个并立的库，分别用于回放和记录，而且这两个库是 OpenCore 对外的接口库；

3)libopencorenet_support.so 提供网络支持的功能;

一些功能以插件(Plug-In)的方式放入 Player 中使用，每个功能使用两个库，一个实现具体功能，一个用于注册。

2、libopencorecommon.so 库的结构

libopencorecommon.so 是整个 OpenCore 的核心库，它的编译控制的文件的路径为 pvcommon/Android.mk，这个文件使用递归的方式寻找子文件：include $(BUILD_SHARED_LIBRARY)

```
include $(PV_TOP)//oscl/oscl/osclbase/Android.mk
include $(PV_TOP)//oscl/oscl/osclerror/Android.mk
include $(PV_TOP)//oscl/oscl/osclmemory/Android.mk
include $(PV_TOP)//oscl/oscl/osclutil/Android.mk
include $(PV_TOP)//oscl/pvlogger/Android.mk
include $(PV_TOP)//oscl/oscl/osclproc/Android.mk
include $(PV_TOP)//oscl/oscl/osclio/Android.mk
include $(PV_TOP)//oscl/oscl/osclregcli/Android.mk
include $(PV_TOP)//oscl/oscl/osclregserv/Android.mk
include $(PV_TOP)//oscl/unit_test/Android.mk
include $(PV_TOP)//oscl/oscl/osclib/Android.mk
include $(PV_TOP)//pvmi/pvmf/Android.mk
include $(PV_TOP)//baselibs/pv_mime_utils/Android.mk
include $(PV_TOP)//nodes/pvfileoutputnode/Android.mk
include $(PV_TOP)//baselibs/media_data_structures/Android.mk
include $(PV_TOP)//baselibs/threadsafe_callback_ao/Android.mk
include $(PV_TOP)//codecs_v2/utilities/colorconvert/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/common/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/common/Android.mk
```

这些被包含的 Android.mk 文件真正指定需要编译的文件，这些文件在 Android.mk 的目录及其子目录中。事实上，在 libopencorecommon.so 库中包含了以下内容：

1）OSCL 的所有内容
2）Pvmf 框架部分的内容（pvmi/pvmf/Android.mk）
3）基础库中的一些内容（baselibs）
4）编解码的一些内容
5）文件输出的 node（nodes/pvfileoutputnode/Android.mk）

从库的结构中可以看出，最终生成库的结构与 OpenCore 的层次关系并非完全重合。libopencorecommon.so 库中就包含了底层的 OSCL 的内容、PVMF 的框架以及 Node 和编解码的工具。

3、libopencoreplayer.so 库的结构

libopencoreplayer.so 是用于播放的功能库，它的编译控制的文件的路径为 pvplayer/Android.mk，它包含了以下的内容：

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/player/Android.mk
```

```
include $(PV_TOP)//codecs_v2/audio/aac/dec/util/getactualaacconfig/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_wb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/common/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/mp3/dec/Android.mk
include $(PV_TOP)//codecs_v2/utilities/m4v_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/utilities/pv_video_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_common/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_queue/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_h264/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_amr/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_mp3/Android.mk
include $(PV_TOP)//codecs_v2/omx/factories/omx_m4v_factory/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_proxy/Android.mk
include $(PV_TOP)//nodes/common/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/oma1/passthru/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/common/Android.mk
include $(PV_TOP)//pvmi/media_io/pvmiofileoutput/Android.mk
include $(PV_TOP)//fileformats/common/parser/Android.mk
include $(PV_TOP)//fileformats/id3parcom/Android.mk
include $(PV_TOP)//fileformats/rawgsmamr/parser/Android.mk
include $(PV_TOP)//fileformats/mp3/parser/Android.mk
include $(PV_TOP)//fileformats/mp4/parser/Android.mk
include $(PV_TOP)//fileformats/rawaac/parser/Android.mk
include $(PV_TOP)//fileformats/wav/parser/Android.mk
include $(PV_TOP)//nodes/pvaacffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmp3ffparsernode/Android.mk
include $(PV_TOP)//nodes/pvamrffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmediaoutputnode/Android.mk
include $(PV_TOP)//nodes/pvomxvideodecnode/Android.mk
include $(PV_TOP)//nodes/pvomxaudiodecnode/Android.mk
include $(PV_TOP)//nodes/pvwavffparsernode/Android.mk
include $(PV_TOP)//pvmi/recognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvamrffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvmp3ffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvwavffrecognizer/Android.mk
include $(PV_TOP)//engines/common/Android.mk
include $(PV_TOP)//engines/adapters/player/framemetadatautility/Android.mk
include $(PV_TOP)//protocols/rtp_payload_parser/util/Android.mk
include $(PV_TOP)//android/Android.mk
include $(PV_TOP)//android/drm/oma1/Android.mk
```

```
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_net_support/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

libopencoreplayer.so 中包含了以下内容：

1）一些解码工具
2）文件的解析器（mp4）
3）解码工具对应的 Node
4）player 的引擎部分（engines/player/Android.mk）
5）为 Android 的 player 适配器（android/Android.mk）
6）识别工具（pvmi/recognizer）
7）编解码工具中的 OpenMax 部分(codecs_v2/omx)
8）对应几个插件 Node 的注册

libopencoreplayer.so 中的内容较多，其中主要为各个文件解析器和解码器，PVPlayer 的核心功能在 engines/player/Android.mk 当中，而 android/Android.mk 的内容比较特殊，它是在 PVPlayer 之上构建的一个为 Android 使用的播放器。

4、libopencoreauthor.so 库的结构

libopencoreauthor.so 是用于媒体流记录的功能库，它的编译控制的文件的路径为 pvauthor/Android.mk，它包含了以下的内容：

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/author/Android.mk
include $(PV_TOP)//codecs_v2/video/m4v_h263/enc/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/enc/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/enc/Android.mk
include $(PV_TOP)//fileformats/mp4/composer/Android.mk
include $(PV_TOP)//nodes/pvamrencnode/Android.mk
include $(PV_TOP)//nodes/pvmp4ffcomposernode/Android.mk
include $(PV_TOP)//nodes/pvvideoencnode/Android.mk
include $(PV_TOP)//nodes/pvavcencnode/Android.mk
include $(PV_TOP)//nodes/pvmediainputnode/Android.mk
include $(PV_TOP)//android/author/Android.mk
```

libopencoreauthor.so 中包含了以下内容：

1）一些编码工具（视频流 H263、H264，音频流 Amr）
2）文件的组成器（mp4）
3）编码工具对应的 Node
4）表示媒体输入的 Node（nodes/pvmediainputnode/Android.m）
5）author 的引擎部分（engines/author/Android.mk）
6）为 Android 的 author 适配器（android/author/Android.mk）

libopencoreauthor.so 中主要为各个文件编码器和文件组成器，PVAuthor 的核心功能在 engines/author/Android.mk 当中，而 android/author/Android.mk 是在 PVAuthor 之上构建的一个为 Android 使用的媒体记录器。

5、其他库

另外的几个库的 Android.mk 文件的路径如下所示：

网络支持库 libopencorenet_support.so：

tools_v2/build/modules/linux_net_support/core/Android.mk

MP4 功能实现库 libopencoremp4.so 和注册库 libopencoremp4reg.so：

tools_v2/build/modules/linux_mp4/core/Android.mk

tools_v2/build/modules/linux_mp4/node_registry/Android.mk

RTSP 功能实现库 libopencorertsp.so 和注册库 libopencorertspreg.so：

tools_v2/build/modules/linux_rtsp/core/Android.mk

tools_v2/build/modules/linux_rtsp/node_registry/Android.mk

下载功能实现库 libopencoredownload.so 和注册库 libopencoredownloadreg.so：

tools_v2/build/modules/linux_download/core/Android.mk

tools_v2/build/modules/linux_download/node_registry/Android.mk

2.3 OpenCore OSCL 简介

OSCL，全称为 Operating System Compatibility Library（操作系统兼容库），它包含了一些在不同操作系统中移植层的功能，其代码结构如下所示：

```
oscl/oscl
|-- config          : 配置的宏
|-- makefile
|-- makefile.pv
|-- osclbase        : 包含基本类型、宏以及一些 STL 容器类似的功能
|-- osclerror       : 错误处理的功能
|-- osclio          : 文件 IO 和 Socket 等功能
|-- oscllib         : 动态库接口等功能
|-- osclmemory      : 内存管理、自动指针等功能
|-- osclproc        : 线程、多任务通讯等功能
|-- osclregcli      : 注册客户端的功能
|-- osclregserv     : 注册服务器的功能
`-- osclutil        : 字符串等基本功能
```

在 oscl 的目录中，一般每一个目录表示一个模块。OSCL 对应的功能是非常细致的，几乎对 C 语言中每一个细节的功能都进行封装，并使用了 C++的接口提供给上层使用。事实上，OperCore 中的 PVMF、Engine 部分都在使用 OSCL，而整个 OperCore 的调用者也需要使用 OSCL。
在 OSCL 的实现中，很多典型的 C 语言函数被进行了简单的封装，例如：osclutil 中与数学相关的功能在 oscl_math.inl 中被定义成为了内嵌（inline）的函数：

```
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log(double value)
{
```

```
    return (double) log(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log10(double value)
{
        return (double) log10(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_sqrt(double value)
{
        return (double) sqrt(value);
}
```

oscl_math.inl 文件又被 oscl_math.h 所包含，因此其结果是 oscl_log()等功能的使用等价于原始的 log()等函数。

很多 C 语言标准库的句柄都被定义成为了 C++类的形式，实现由一些繁琐，但是复杂性都不是很高。以 oscllib 为例，其代码结构如下所示：

```
oscl/oscl/oscllib/
|-- Android.mk
|-- build
|   `-- make
|       `-- makefile
`-- src
    |-- oscl_library_common.h
    |-- oscl_library_list.cpp
    |-- oscl_library_list.h
    |-- oscl_shared_lib_interface.h
    |-- oscl_shared_library.cpp
    `-- oscl_shared_library.h
```

oscl_shared_library.h 是提供给上层使用的动态库的接口功能，它定义的接口如下所示：

```
class OsclSharedLibrary
{
public:
        OSCL_IMPORT_REF OsclSharedLibrary();
        OSCL_IMPORT_REF OsclSharedLibrary(const OSCL_String& aPath);
        OSCL_IMPORT_REF ~OsclSharedLibrary();
        OSCL_IMPORT_REF OsclLibStatus LoadLib(const OSCL_String& aPath);
        OSCL_IMPORT_REF OsclLibStatus LoadLib();
        OSCL_IMPORT_REF void SetLibPath(const OSCL_String& aPath);
        OSCL_IMPORT_REF    OsclLibStatus    QueryInterface(const    OsclUuid&
aInterfaceId, OsclAny*& aInterfacePtr);
        OSCL_IMPORT_REF OsclLibStatus Close();
        OSCL_IMPORT_REF void AddRef();
        OSCL_IMPORT_REF void RemoveRef();
}
```

这些接口显然都是与库的加载有关系的，而在 oscl_shared_library.cpp 中其具体的功能是使用 dlopen()等函数来实现的。

2.4 Open Core 上层代码结构

在实际开发中我们并不会过多的研究 Open Core 的实现，Android 提供了上层的 Media API 给开发人员使用，MediaPlayer 和 MediaRecorder。

Android Media APIs

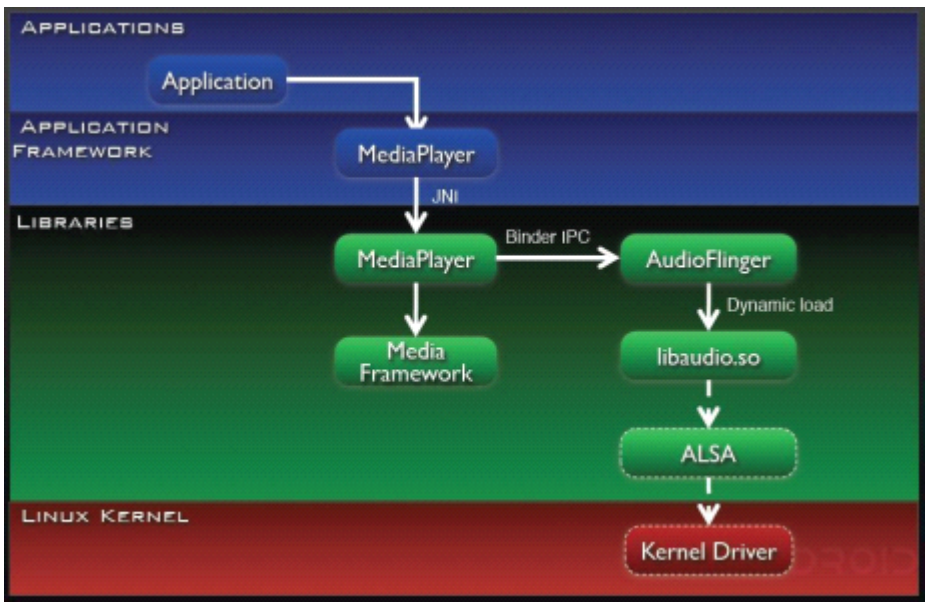
The Android platform is capable of playing both audio and video media. It is also capable of playing media contained in the resources for an application, or a standalone file in the filesystem, or even streaming media over a data connection. Playback is achieved through the android.media.MediaPlayer class.

The Android platform can also record audio. Video recording capabilities are coming in the future. This is achieved through the android.media.MediaRecorder class.

2.4.1 简单的例子教你怎么用 MediaPlayer。

```
Example:
MediaRecorder recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
recorder.setOutputFile(PATH_NAME);
recorder.prepare();
recorder.start(); // Recording is now started ... recorder.stop();
recorder.reset(); // You can reuse the object by going back to
setAudioSource() step
recorder.release(); // Now the object cannot be reused
```

2.4.2 整体的结构如下图所示：



MediaPlayer JNI

代码位置 /frameworks/base/media/jni

MediaPlayer (Native)

代码位置 /frameworks/base/media/libmedia

MediaPlayerService (Server)

代码位置 /frameworks/base/media/libmediaplayerservice

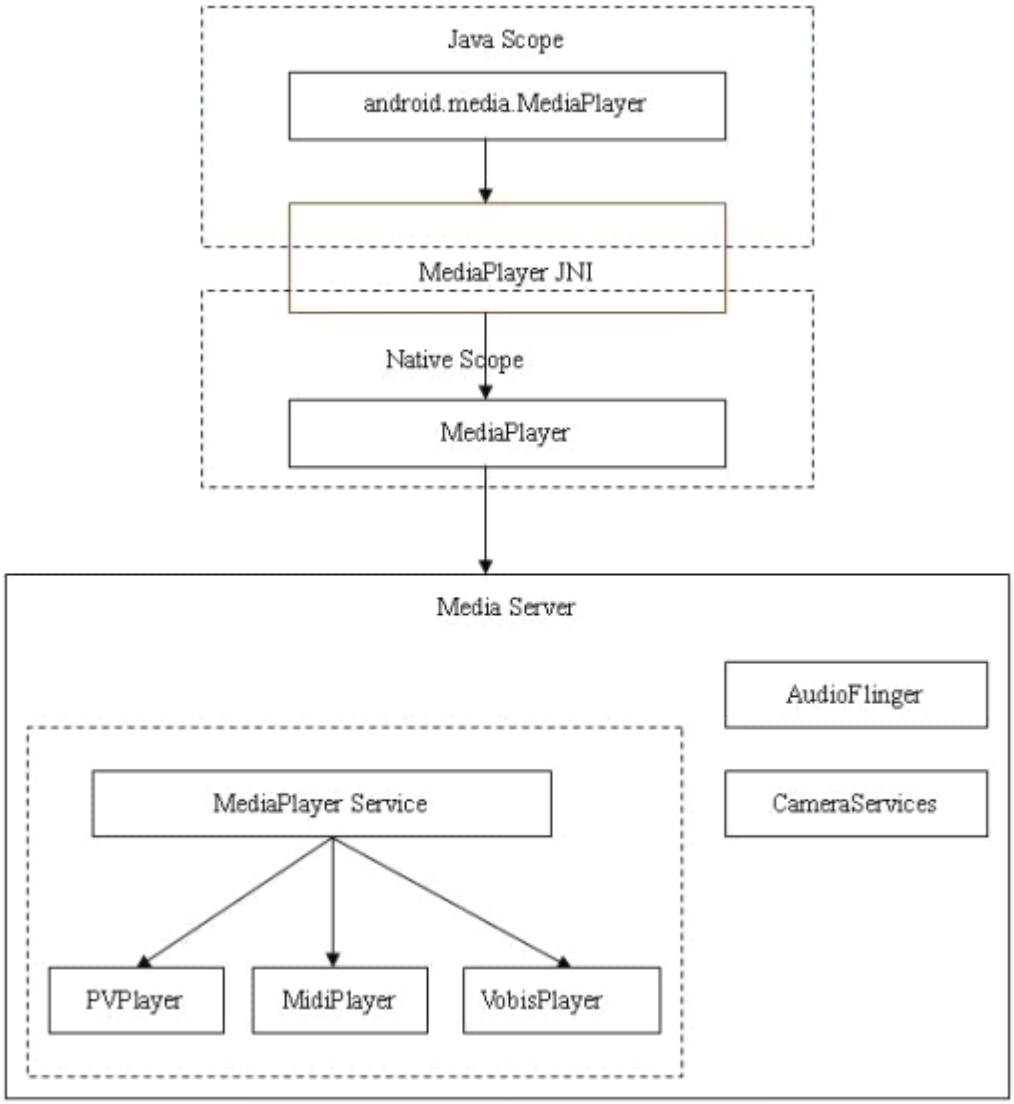
MediaPlayerService Host Process

代码位置 /frameworks/base/media/mediaserver/main_mediaserver.cpp

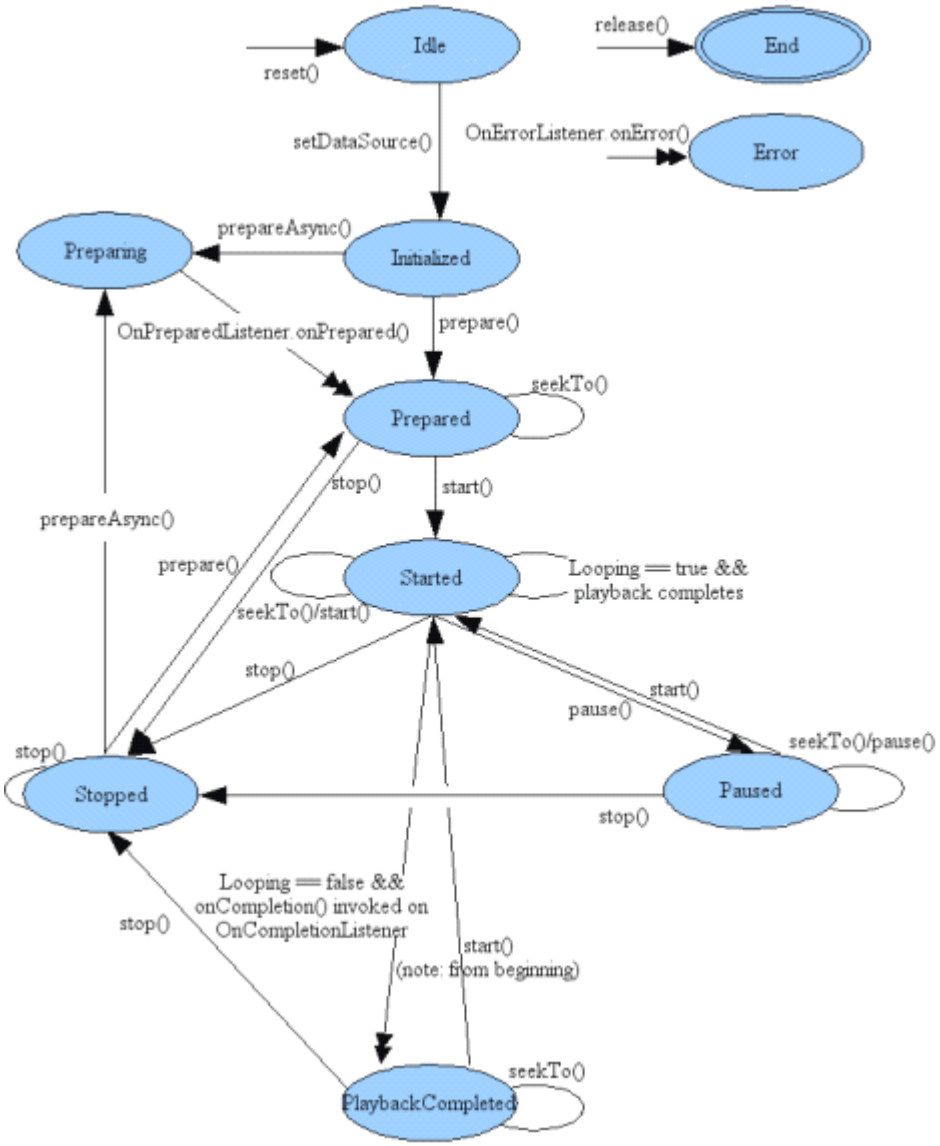
PVPlayer

代码位置 /external/opencore/android/

2.4.3 实际调用过程如下图所示：



2.4.4 MediaPlayer 的生命周期如下图：



2.4.5 MediaPlayer 常用方法介绍

1、**方法：** create(Context context, Uri uri)
解释： 静态方法，通过 Uri 创建一个多媒体播放器。

2、**方法：** create(Context context, int resid)
解释： 静态方法，通过资源 ID 创建一个多媒体播放器。

3、**方法：** create(Context context, Uri uri, SurfaceHolder holder)
解释： 静态方法，通过 Uri 和指定 SurfaceHolder 【抽象类】 创建一个多媒体播放器。

4、**方法：** getCurrentPosition()
解释： 返回 Int，得到当前播放位置。

5、**方法：** getDuration()
解释： 返回 Int，得到文件的时间。

6、**方法：** getVideoHeight()
解释： 返回 Int ，得到视频的高度。

7、**方法：** getVideoWidth()
解释： 返回 Int，得到视频的宽度。

8、**方法：** isLooping()
解释： 返回 boolean ，是否循环播放。

9、**方法：** isPlaying()
解释： 返回 boolean，是否正在播放。

10、**方法：** pause()
解释： 无返回值 ，暂停。

11、**方法：** prepare()
解释： 无返回值，准备同步。

12、**方法：** prepareAsync()
解释： 无返回值，准备异步。

13、**方法：** release()
解释： 无返回值，释放**MediaPlayer** 对象。

14、**方法：** reset()
解释： 无返回值，重置**MediaPlayer** 对象。

15、**方法：** seekTo(int msec)
解释： 无返回值，指定播放的位置（以毫秒为单位的时间）。

16、**方法：** setAudioStreamType(int streamtype)
解释： 无返回值，指定流媒体的类型

17、**方法：** setDataSource(String path)
解释： 无返回值，设置多媒体数据来源【根据 路径】

18、**方法：** setDataSource(FileDescriptor fd, long offset, long length)
解释： 无返回值，设置多媒体数据来源【根据 FileDescriptor】。

19、**方法：** setDataSource(FileDescriptor fd)
解释： 无返回值，设置多媒体数据来源【根据 FileDescriptor】。

20、**方法：** setDataSource(Context context, Uri uri)
解释： 无返回值，设置多媒体数据来源【根据 Uri】。

21、**方法：** setDisplay(SurfaceHolder sh)
解释： 无返回值，设置用 SurfaceHolder 来显示多媒体。

22、**方法：** setLooping(boolean looping)
解释： 无返回值，设置是否循环播放。

23、**事件：** setOnBufferingUpdateListener(MediaPlayer.OnBufferingUpdateListener listener)
解释： 监听事件，网络流媒体的缓冲监听。

24、**事件：** setOnCompletionListener(MediaPlayer.OnCompletionListener listener)
解释： 监听事件，网络流媒体播放结束。

25、**事件：** setOnErrorListener(MediaPlayer.OnErrorListener listener)
解释： 监听事件，设置错误信息监听。

26、**事件：** setOnVideoSizeChangedListener(MediaPlayer.OnVideoSizeChangedListener listener)
解释： 监听事件，视频尺寸监听。

26、**方法：** setScreenOnWhilePlaying(boolean screenOn)
解释： 无返回值，设置是否使用 SurfaceHolder 显示。

27、**方法：** setVolume(float leftVolume, float rightVolume)
解释： 无返回值，设置音量。

28、**方法：** start()
解释： 无返回值，开始播放

29、**方法：** stop()
解释： 无返回值，停止播放

另附： 本期附录中的 musicPlayer，是一个小播放器的源码，供 eoer 进行参考。其中，viewHolder 是一个空间的集合对象（实际上是按钮的集合）；MusicFilter.java 类是一个文件类型过滤器。

2.5 Open Core 中的核心概念：服务(Service)与插件(Plugin)的定义

OpenCore 纯插件体系结构中的核心概念包括：微内核、插件与服务。

1、微内核(MicroKernel)：

提供基础的插件与服务架构；负责插件的生命周期管理，包括插件的安装(Install)、启用(Activate)、停止(Deactivate)与卸载(Uninstall)；负责服务的生命周期管理，包括服务的注册(Register)、发现(Lookup)、启动(Start)、停止(Stop)，服务间的依赖绑定。

OpenCore 微内核分两层：

1）应用系统的核心层：以 OSGi 为基础
2）增强层：IoC 实现、扩展点机制与 RMI 插件

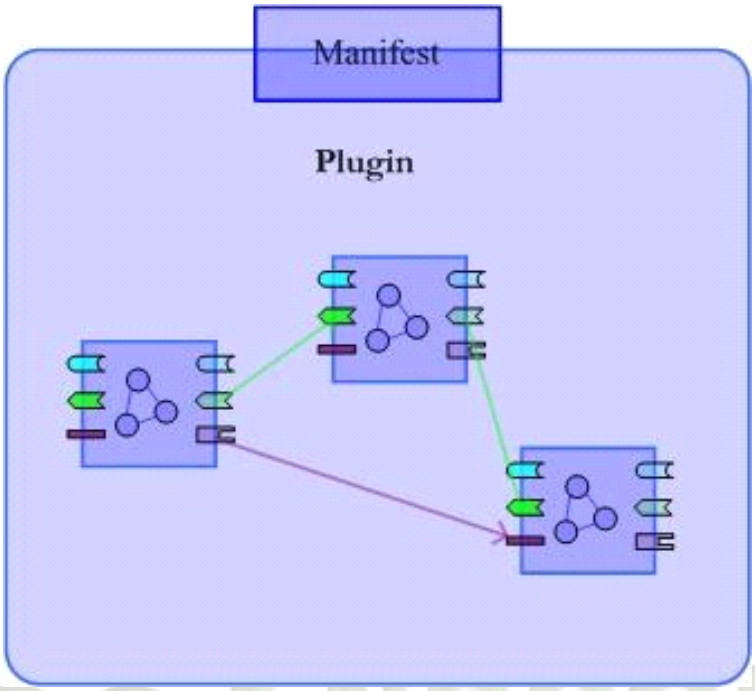
2、插件(Plugin)：

OpenCore 是一个纯插件结构的系统，包括内核在内的所有特性都由插件构成。插件是对系统中静态结构方面的抽象，满足某种约束并提供自描述的物理软件模块。OSGi 中插件叫 Bundle，物理上是一个提供自描述文件（MANIFEST.MF）的 Jar 文件。插件可以被动态的安装、激活、去激活与卸载。插件包含多个服务(Service)对象。

插件间的依赖关系包括两种：

1）物理依赖，即插件间的 Class 依赖，例如 A 插件 Import B 插件的 Class
2）逻辑依赖，插件 A 中的服务依赖插件 B 中的服务。

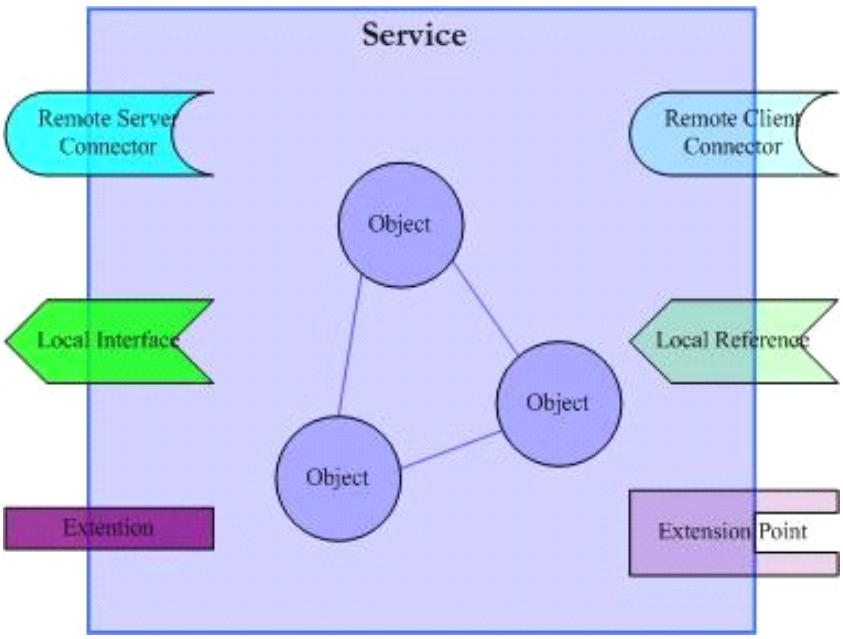
插件的概念描述图如下：



3、服务(Service)：

系统中动态结构方面的抽象，是运行时的概念，是具有良好接口与自我描述的业务功能提供者。Java 平台上是一个具有良好接口对象实例。服务可以被注册、发现、启动与停止。运行时，服务间在同一 JVM 内通过传引用（By-Reference）依赖协作，在不同 JVM 内通过 RMI、JMS、REST 等传值方式（By-Value）通信协作。

服务的概念描述图如下：



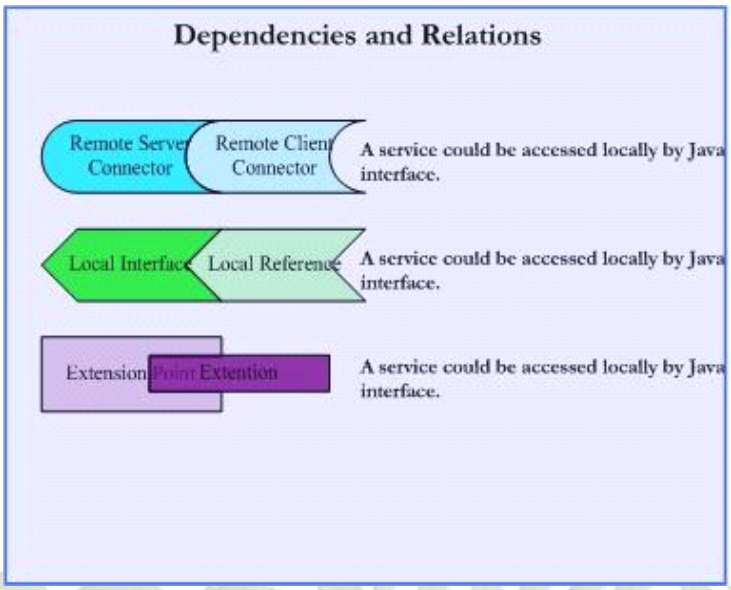
服务间依赖关系(Dependency and Relations)有三种：

1）进程间通过连接器(Connector)以传值(By-Value)方式通信和协作。

2）进程内通过 Java 接口调用以传引用(By-Reference)方式通信协作。

3）进程内一个服务可以作为对另一服务的扩展。

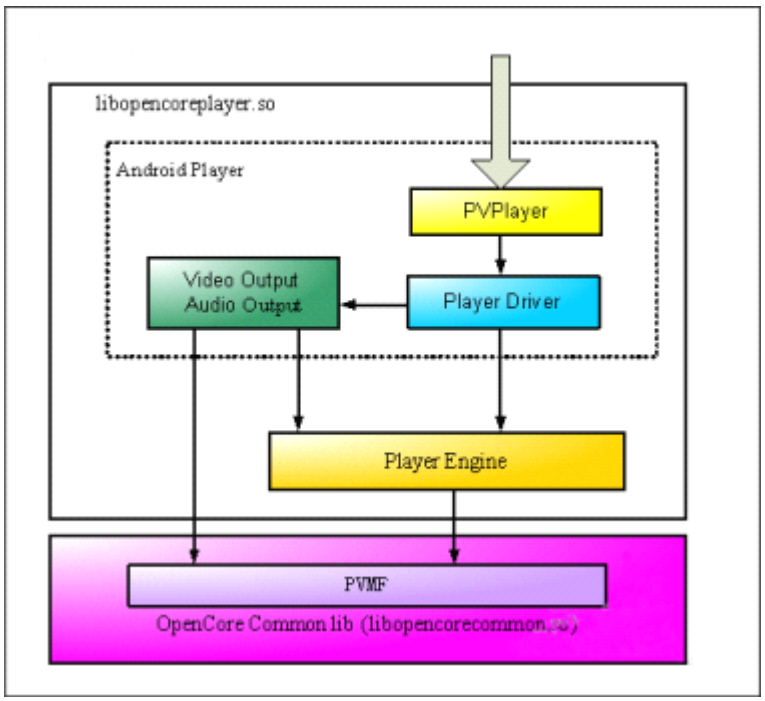
服务间关系的概念描述图：



2.6 OpenCore 的 PVPlayer 介绍。

2.6.1 Player 的组成

OpenCore 的 Player 的编译文件是 pvplayer/Android.mk，将生成动态库文件 libopencoreplayer.so。这个库包含了两方面的内容：一方是 Player 的 engine(引擎)，一方面是为 Android 构件的 Player，这实际上是一个适配器(adapter)。engine 的路径是 engine/player;adapter 的路径是 android。



2.6.2 Player Engine 部分

OpenCore 的 Player Engine 具有清晰明确的接口。在这个接口之上，不同的系统可以根据自己的情况实现不同 Player 。目录 engines 中的文件结构如下所示：

```
engines/player/
|-- Android.mk
|-- build
|   |-- linux_nj
|   |-- make
|   `-- makefile.conf
|-- config
|   `-- linux_nj
|-- include
|   |-- pv_player_datasink.h
|   |-- pv_player_datasinkfilename.h
|   |-- pv_player_datasinkpvmfnode.h
|   |-- pv_player_datasource.h
|   |-- pv_player_datasourcepvmfnode.h
|   |-- pv_player_datasourceurl.h
|   |-- pv_player_events.h
|   |-- pv_player_factory.h
|   |-- pv_player_interface.h
|   |-- pv_player_license_acquisition_interface.h
|   |-- pv_player_registry_interface.h
|   |-- pv_player_track_selection_interface.h
|   `-- pv_player_types.h
|-- sample_app
|   |-- Android.mk
|   |-- build
|   |-- sample_player_app_release.txt
|   `-- src
|-- src
|   |-- pv_player_datapath.cpp
|   |-- pv_player_datapath.h
|   |-- pv_player_engine.cpp
|   |-- pv_player_engine.h
|   |-- pv_player_factory.cpp
|   |-- pv_player_node_registry.h
|   `-- pv_player_sdkinfo.h
`-- test
    |-- Android.mk
    |-- build
    |-- config
    `-- src
```

其中，engines/player/include 目录中是接口头文件，engines/player/src 目录源文件和私有头文件，主要头文件的功能如下所示：

pv_player_types.h ： 定义一些数据结构和枚举值。

pv_player_events.h ： 定义 UUID 和一些错误值。

pv_player_datasink.h ： datasink 是媒体数据的输出，定义类 PVPlayerDataSink，这是媒体数据输出的基类， 作为接口使用。

pv_player_datasinkfilename.h ： 定义类 PVPlayerDataSinkFilename 继承 PVPlayerDataSink 。

pv_player_datasinkpvmfnode.h ： 定义类 PVPlayerDataSinkPVMFNode 继承 PVPlayerDataSink 。

pv_player_datasource.h ： datasource 是媒体数据的输入，定义类 PVPlayerDataSource，这是媒体数据输入的基类，作为接口使用。

pv_player_datasourcepvmfnode.h ： 定义类 PVPlayerDataSourcePVMFNode 继承 PVPlayerDataSource。

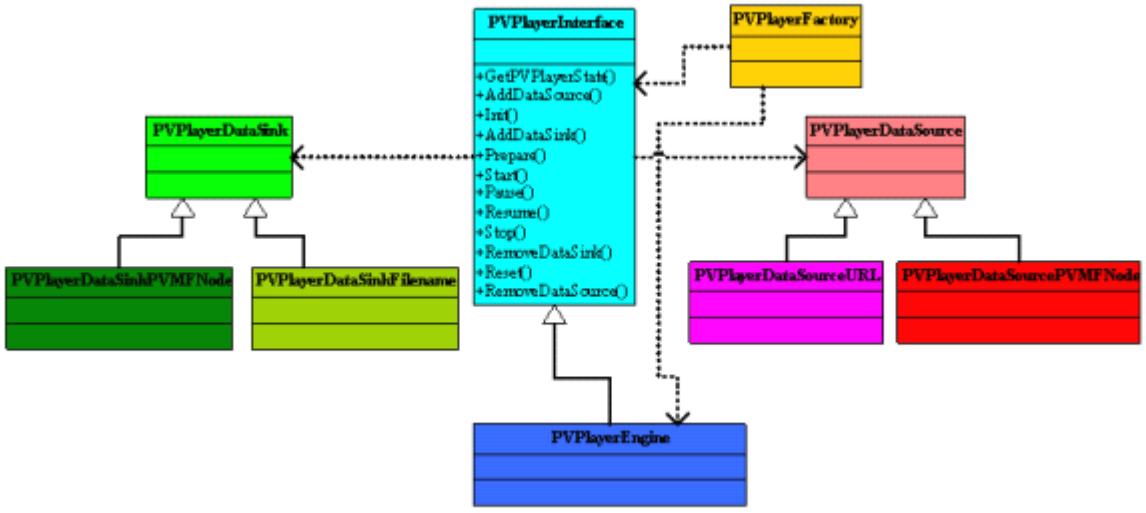
pv_player_datasourceurl.h ： 定义类 PVPlayerDataSourceURL 继承 PVPlayerDataSource。

pv_player_interface.h ： 定义 Player 的接口 PVPlayerInterface，这是一个接口类。

pv_player_factory.h ： 主要定义工厂类 PVPlayerFactory，用于创建和销毁 PVPlayerInterface。

事实上，在 engines/player/src 目录中，主要实现类为 pv_player_engine.cpp，其中定义了类 PVPlayerEngine ，PVPlayerEngine 继承了 PVPlayerInterface，这是一个实现类，在 PVPlayerFactory 创建 PVPlayerInterface 接口的时候，实际创建的是 PVPlayerEngine。

在 Player Engine 的实现中，包含了编解码和流控制等功能，而输出的介质需要从外部设置进来。 PVPlayerInterface 定义的接口基本是按照操作顺序的，主要的接口如下所示：



在 Player Engine 的实现中，包含了编解码和流控制等功能，而输出的介质需要从外部设置进来。PVPlayerInterface 定义的接口基本是按照操作顺序的，主要的接口如下所示：

```
PVCommandId AddDataSource(PVPlayerDataSource& aDataSource, const OsclAny* aContextData = NULL);
PVCommandId Init(const OsclAny* aContextData = NULL);
PVCommandId AddDataSink(PVPlayerDataSink& aDataSink, const OsclAny* aContextData = NULL);
PVCommandId Prepare(const OsclAny* aContextData = NULL);
PVCommandId Start(const OsclAny* aContextData = NULL);
PVCommandId Pause(const OsclAny* aContextData = NULL);
PVCommandId Resume(const OsclAny* aContextData = NULL);
```

```
PVCommandId Stop(const OsclAny* aContextData = NULL);
PVCommandId RemoveDataSink(PVPlayerDataSink& aDataSink, const OsclAny* aContextData = NULL);
PVCommandId Reset(const OsclAny* aContextData = NULL);
PVCommandId RemoveDataSource(PVPlayerDataSource& aDataSource, const OsclAny* aContextData = NULL);
```

这里面的DataSink可能包含Video的输出和Audio的输出两者部分。在pv_player_types.h文件中，定义了Player的状态机，以PVP_STATE_为开头，如下所示：

```
typedef enum
{
    PVP_STATE_IDLE        = 1,
    PVP_STATE_INITIALIZED = 2,
    PVP_STATE_PREPARED    = 3,
    PVP_STATE_STARTED     = 4,
    PVP_STATE_STARTED     = 5
    PVP_STATE_STARTED     = 6
} PVPlayerState
```

PVPlayerInterface 中的各个操作如果成功，可以更改 Player 的状态机：初始化的时候Player是PVP_STATE_IDLE状态，调用Init后，进入PVP_STATE_INITIALIZED状态；调AddDataSink，进入 PVP_STATE_PREPARED 状态；调用 Prepare 后，进入 PVP_STATE_PREPARED 状态；调用 start 后进入 PVP_STATE_STARTED 状态，之后可以调用 pause 进入 PVP_STATE_PAUSED 状态。

PVP_STATE_STARTED和PVP_STATE_PAUSED状态是播放情况下的状态，可以使用start和pause函数在这两个状态中切换。

在播放过程中，调用 stop 可以返回 PVP_STATE_INITIALIZED 状态，在调用 RemoveDataSource返回 PVP_STATE_IDLE 状态。

2.6.3 Android Player Adapter

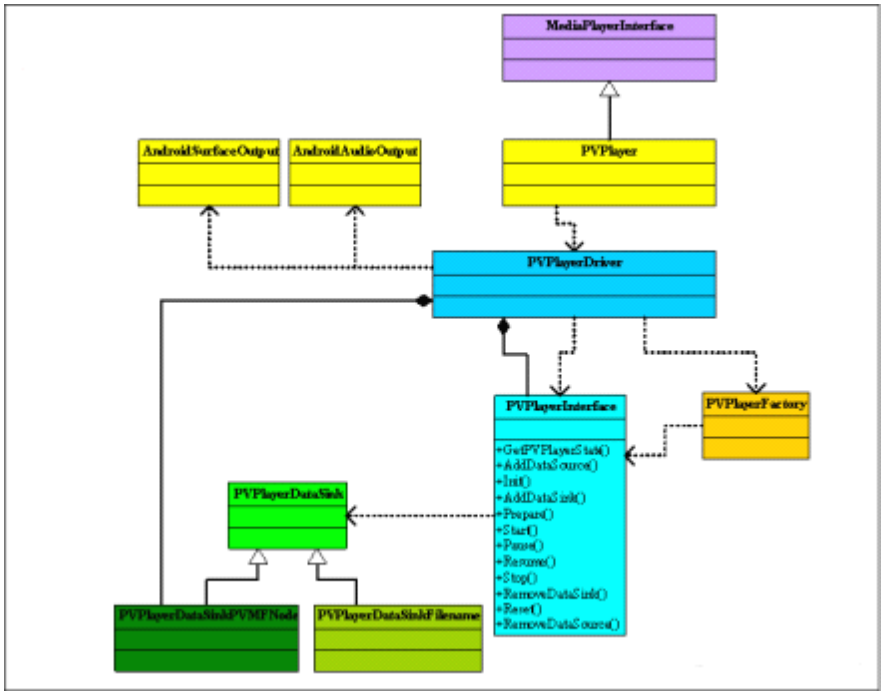
在Android目录中定义为Player的适配器，这个目录主要包含的文件如下所示：

```
android
|-- Android.mk
|-- android_audio_mio.cpp
|-- android_audio_mio.h
|-- android_audio_output.cpp
|-- android_audio_output.h
|-- android_audio_output_threadsafe_callbacks.cpp
|-- android_audio_output_threadsafe_callbacks.h
|-- android_audio_stream.cpp
|-- android_audio_stream.h
|-- android_log_appender.h
|-- android_surface_output.cpp
|-- android_surface_output.h
|-- mediascanner.cpp
|-- metadatadriver.cpp
|-- metadatadriver.h
```

```
|-- playerdriver.cpp
|-- playerdriver.h
`-- thread_init.cpp
```

这个Android的Player的“适配器”需要调用OpenCore的Player Engine的接口，实现Android的媒体播放器的服务所需要接口，即最终实现一个 PVPlayer，而 PVPlayer 实际上是继承了MediaPlayerInterface。

在实现过程中，首先实现了一个PlayerDriver，然后再使用PVPlayer，PVPlayer通过调用PlayerDriver来完成具体的功能。整个实现的结构图如图所示：



对PVPlayerDriver的各种操作使用各种命令来完成，这些命令在playerdriver.h中进行的定义。

```
enum player_command_type {
    PLAYER_QUIT                 = 1,
    PLAYER_SETUP                = 2,
    PLAYER_SET_DATA_SOURCE      = 3,
    PLAYER_SET_VIDEO_SURFACE    = 4,
PLAYER_SET_AUDIO_SINK         = 5,
PLAYER_INIT                   = 6,
PLAYER_PREPARE                = 7,
    PLAYER_START                = 8,
    PLAYER_STOP                 = 9,
    PLAYER_PAUSE                = 10,
    PLAYER_RESET                = 11,
    PLAYER_SET_LOOP             = 12,
    PLAYER_SEEK                 = 13,
    PLAYER_GET_POSITION         = 14,
    PLAYER_GET_DURATION         = 15,
```

```
    PLAYER_GET_STATUS               = 16,
    PLAYER_REMOVE_DATA_SOURCE       = 17,
    PLAYER_CANCEL_ALL_COMMANDS      = 18,
};
```

这些命令一般实现的是PVPlayerInterface各个接口的简单封装，例如对于较为简单的暂停播放这个操作，整个系统执行的过程如下所示：

1）在PVPlayer中的pause函数（在playerdriver.cpp文件中）

```
status_t PVPlayer::pause()
{
    LOGV("pause");
    return mPlayerDriver->enqueueCommand(new PlayerPause(0,0));
}
```

这时调用其成员mPlayerDriver（PlayerDriver类型）的函数，将一个PlayerPause命令加入了命令序列，具体的各种命令功能在playerdriver.h文件中。

2）PlayerDriver类的enqueueCommand将间接调用各个以handle为开头的函数，对于PlayerPause命令，调用的函数是handlePause。

```
void PlayerDriver::handlePause(PlayerPause* ec)
{
    LOGV("call pause");
    mPlayer->Pause(0);
    FinishSyncCommand(ec);
}
```

这里的mPlayer是一个PVPlayerInterface类型的指针，使用这个指针调用到了OpenCore的Player Engine中的PVPlayerEngine类。

在这个播放器适配器的实现中，一个主要工作是将Android框架中定义的媒体的输出（包括Audio的输出和Video的输出）转换成，OpenCore的Player Engine需要的形式。

在这里两个重要的类是android_surface_output.cpp实现的AndroidSurfaceOutput，android_audio_output.cpp实现的AndroidAudioOutput。

对于Video输出的设置过程，在类PlayerDriver中定义了3个成员：

```
PVPlayerDataSink        *mVideoSink;
PVMFNodeInterface       *mVideoNode;
PvmiMIOControl          *mVideoOutputMIO;
```

这里的mVideoSink的类型为PVPlayerDataSink，这是Player Engine中定义的类接口，mVideoNode的类型为VMFNodeInterface，在pvmi/pvmf/include的pvmf_node_interface.h中定义，这是所有的PVMF的NODE都需要继承的统一接口，mVideoOutputMIO的类型为PvmiMIOControl也在pvmi/pvmf/include中定义，这是媒体图形输出控制的接口类。

1）在PVPlayer的setVideoSurface用以设置一个Video输出的界面，这里使用的参数的类型是ISurface指针：

```
status_t PVPlayer::setVideoSurface(const sp<ISurface>& surface)
{
    LOGV("setVideoSurface(%p)", surface.get());
    mSurface = surface;
    return OK;
}
```

setVideoSurface 函数设置的是 PVPlayer 中的一个成员 mSurface，真正设置 Video 输出的界面的功能在 run_set_video_surface()函数中实现：

```
void PVPlayer::run_set_video_surface(status_t s, void *cookie)
{
    LOGV("run_set_video_surface s=%d", s);
    if (s == NO_ERROR) {
        PVPlayer *p = (PVPlayer*)cookie;
        if (p->mSurface == NULL) {
            run_set_audio_output(s, cookie);
        } else {
            p->mPlayerDriver->enqueueCommand(new
PlayerSetVideoSurface(p->mSurface, run_set_audio_output, cookie));
        }
    }
}
```

这时使用的命令是 PlayerSetVideoSurface，最终将调用到 PlayerDriver 中的 handleSetVideoSurface 函数。

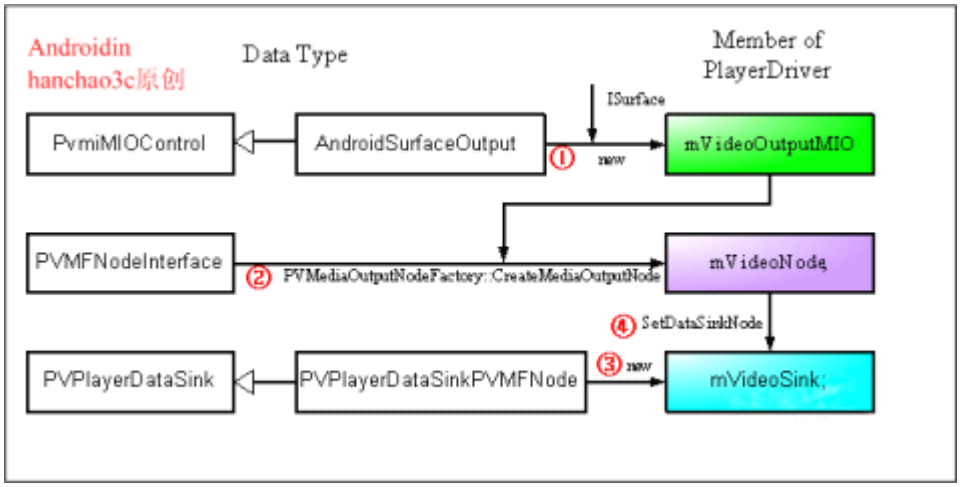
2）handleSetVideoSurface 函数的实现如下所示：

```
void PlayerDriver::handleSetVideoSurface(PlayerSetVideoSurface* ec)
{
    int error = 0;
    mVideoOutputMIO = new AndroidSurfaceOutput(ec->surface());
    mVideoNode = PVMediaOutputNodeFactory::CreateMediaOutputNode(mVideoOutputMIO);
    mVideoSink = new PVPlayerDataSinkPVMFNode;
    ((PVPlayerDataSinkPVMFNode *)mVideoSink)->SetDataSinkNode(mVideoNode);
    ((PVPlayerDataSinkPVMFNode *)mVideoSink)->SetDataSinkFormatType(PVMF_YUV420);
    OSCL_TRY(error, mPlayer->AddDataSink(*mVideoSink, ec));
    OSCL_FIRST_CATCH_ANY(error, commandFailed(ec));
}
```

在这里首先建立的创建成员 mVideoOutputMIO（类型为 PvmiMIOControl），这时建立的类是类 AndroidSurfaceOutput，这个类继承了 PvmiMIOControl，所以可以作为 PvmiMIOControl 使用。然后调用 PVMediaOutputNodeFactory::CreateMediaOutputNode 建立了 PVMFNodeInterface 类型的 mVideoNode。

随后创建 PVPlayerDataSinkPVMFNode 类型的 mVideoSink，PVPlayerDataSinkPVMFNode 本身继承了 PVPlayerDataSink，因此可以作为 PVPlayerDataSink 使用。调用 SetDataSinkNode 函数

将 mVideoNode 设置为 mVideoSink 的数据输出节点。



事实上，对于 Video 的输出，基本的功能都是在类 AndroidSurfaceOutput 中完成的，在这个类当中，主要的工作是将 Android 的 ISurface 输出作为 Player Engine 的输出。最后调用了 AddDataSink 将 mVideoSink 增加为了 PVPlayerInterface 的输出。

在 android_surface_output.cpp 文件中实现了类 AndroidSurfaceOutput，这个类相当于一个 OpenCore Player Engine 的 Video 输出和 Android 输出的“适配器”。AndroidSurfaceOutput 类本身继承了类 PvmiMIOControl，而其构造函数又以 ISurface 类型为参数。这个类的实现是使用 ISurface 实现 PvmiMIOControl 的各个接口。

【文件格式处理和编解码部分简介】

在多媒体方面，文件格式的处理和编解码(Codec)是很基础的两个方面的内容。多媒体应用的两个方面是媒体的播放(PlayBack)和媒体的记录(Recording)。

在媒体的播放过程中，通常情况是从对媒体文件的播放，必要的两个步骤为文件的解析和媒体流的解码。例如对于一个 mp4 的文件，其中可能包括 AMR 和 AAC 的音频流，H263、MPEG4 以及 AVC(H264)的视频流，这些流被封装在 3GP 的包当中，媒体播放器要做的就是从文件中将这些流解析出来，然后对媒体流进行解码，解码后的数据才可以播放。

在媒体的记录过程中，通过涉及到视频、音频、图像的捕获功能。对于将视频加音频录制成文件功能，其过程与播放刚好相反，首先从硬件设备得到视频和音频的媒体流，然后对其进行编码，编码号的流还需要被分层次写入到文件之中，最终得到组成好的文件。

OpenCore 有关文件格式处理和编解码部分两部分的内容，分别在目录 fileformats 和 codecs_v2 当中。这两部分都属于基础性的功能，不涉及具体的逻辑，因此它们被别的模块调用来使用，例如：构建各种 Node。

3.1 文件格式的处理

由于同时涉及播放文件和记录文件两种功能，因此 OpenCore 中的文件格式处理有两种类型，一种是 parser(解析器)，另一种是 composer(组成器)。

fileformats 的目录结构如下所示：

```
fileformats
|-- avi
|   `-- parser
|-- common
|   `-- parser
|-- id3parcom
|   |-- Android.mk
|   |-- build
|   |-- include
|   `-- src
|-- mp3
|   `-- parser
|-- mp4
|   |-- composer
|   `-- parser
|-- rawaac
|   `-- parser
|-- rawgsmamr
|   `-- parser
`-- wav
    `-- parser
```

目录包含各个子目录中，它们对应的是不同的文件格式，例如 mp3、mp4 和 wav 等。

3.2 编解码

编解码部分主要针对 Audio 和 Video，codecs_v2 的目录结构如下所示：

```
codecs_v2
|-- audio
|   |-- aac
|   |-- gsm_amr
|   |-- mp3
|   `-- sbc
|-- omx
|   |-- factories
|   |-- omx_aac
|   |-- omx_amr
|   |-- omx_common
|   |-- omx_h264
|   |-- omx_m4v
|   |-- omx_mp3
|   |-- omx_proxy
|   `-- omx_queue
|-- utilities
|   |-- colorconvert
|   |-- m4v_config_parser
|   `-- pv_video_config_parser
`-- video
    |-- avc_h264
    `-- m4v_h263
```


在 audio 和 video 目录中，对应了针对各种流的子目录，其中可能包含 dec 和 enc 两个目录，分别对应解码和编码。video 目录展开后的内容如下所示：

```
`-- video
    |-- avc_h264
    |   |-- common
    |   |-- dec
    |   |-- enc
    |   `-- patent_disclaimer.txt
    `-- m4v_h263
        |-- dec
        |-- enc
        `-- patent_disclaimer.txt
```

codecs_v2 目录的子目录 omx 实现了一个 khronos。

3.3 3OpenMAX 的功能

OpenMAX 是一个多媒体应用程序的框架标准，由 NVIDIA 公司和 Khronos 在 2006 年推出。OpenMAX IL 1.0（集成层）技术规格定义了媒体组件接口，以便在嵌入式器件的流媒体框架中快

速集成加速式编解码器。

OpenMAX 的设计实现可以让具有硬件编辑码功能的平台提供统一的接口和框架，在 OpenMAX 中可以直接使用硬件加速的进行编解码乃至输出的功能，对外保持统一的接口。但是在此处的 OpenMAX 则是一个纯软件的实现。

【Android 多媒体开发技巧】

4.1 Android 多媒体开发相关技巧一

4.1.1 ./packages/providers/MediaProvider/

编译后生成MediaProvider.apk。会在开机时扫描本机和sdcard上的媒体文件(图片、视频、音频），并在/data/data/com.android.providers.media/databases目录下生成internal.db(/system/meida)和external-?.db(/sdcard)两个数据库文件.此后，所有的多媒体信息都从这两个数据库中获取。

4.1.2 ./packages/apps/Music

编译后生成Music.apk，用来播放音频文件。播放列表及相关信息从external-?.db中获取。

4.1.3 ./packages/apps/Camera/

编译后生成Camera.apk,对应于Camera、Gallery、Camcorder三个应用。其中Gallery用来管理所有的视频和图片文件，具有播放、查看、删除等功能。

4.1.4 ./frameworks/base/core/java/android/widget/VideoView.java

Android 封装的一个视频播放控件，可一通过下面的方式使用：

```
import android.widget.VideoView;
...
final VideoView view = (VideoView)findViewById(R.id.video_view);
view.setVideoPath("/sdcard/test.mp4");
view.start();
...

在 XML 描述中加入：
< android:id="@+id/video_view">
android:layout_ width="fill_parent"
android:layout_border="1" height="300px"
/>
```

4.1.5 ./frameworks/base/core/java/android/provider/MediaStore.java

Android 提供的多媒体数据库，Android 的所有多媒体数据信息都可以从这里提取。数据库的操作通过利用ContentResolver调用相关的接口实现。

4.1.6 ./frameworks/base/media/java/android/media

提供了android上 多媒体应用层的操作接口。主要说明：

MediaPlayer.java： 提供了视频、音频、数据流的播放控制等操作的接口。

MediaRecorder.java： 提供了视频、音频录制的接口。

AudioManager.java： 提供了音频音量，以及播放模式(静音、震动等)的控制。

RingtoneManager.java、Ringtone.java： 提供了提醒、闹钟、事件等声音的播放控制。
MediaScanner*.java： 提供了媒体扫描接口的支持。
AudioTrack.java： SoundPool.java 播放 android application 的生音资源。
AudioRecord.java： 为 android applicatio 提供录音设置(sample、chanel 等)的接口;

4.1.7 一个简单的例子：

1、播放一个文件：

```
MediaPlayer mp = new MediaPlayer();
mp.setDataSource("/sdcard/test.mp3");
mp.prepare();
mp.start();
```

2、播放 raw resouce：

```
MediaPlayer mp = MediaPlayer.create(context, R.raw.sound_file_1);
mp.start();
```

3、录音：

```
    MediaRecorder recorder = new MediaRecorder();
    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);

recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setOutputFile("/sdcard/test.amr");
    recorder.prepare();
    recorder.start();
    ...
```

4、alarmclock 调用 Media 的基本过程：

```
 ...
mMediaPlayer = new MediaPlayer();
mMediaPlayer.setVolume(IN_CALL_VOLUME, IN_CALL_VOLUME);
// 检查是否是在通话中，若是，就换用较小的声音提示。
if (tm.getCallState() != TelephonyManager.CALL_STATE_IDLE) {
Log.v("Using the in-call alarm");
mMediaPlayer.setVolume(IN_CALL_VOLUME, IN_CALL_VOLUME);
setDataSourceFromResource(context.getResources(),
mMediaPlayer, R.raw.in_call_alarm);
}
else
{
mMediaPlayer.setDataSource(context, Uri.parse(mAlert));
```

```
}
mMediaPlayer.setAudioStreamType(AudioManager.STREAM_ALARM);
mMediaPlayer.setLooping(true);
mMediaPlayer.prepare();
mMediaPlayer.prepare();
...
```

4.2 Android 多媒体开发相关技巧二(FrameWork 相关)

4.2.1 ./frameworks/base/media/jni

JAVA 本地调用部分。编译后生成的目标是 libmedia_jni.so.

android_media_MediaPlayer.cpp: MediaPlayer 的 JAVA 本地调用部分。

它定义了一个 JNINativeMethod(JAVA 本地调用方法)类型的数据 gMethods,用来描述，接口的关联信息。

android_media_MediaRecorder.cpp: 定义了录音的本地调用实现。
android_media_AmrInputStream.cpp: amr 编码相关的本地调用实现。
android_media_MediaScanner.cpp: 媒体扫描相关的本地调用实现。
soundpool/android_media_SoundPool.cpp: 定义了音频系统的本地调用实现。

4.2.2 ./frameworks/base/media/libmedia
./frameworks/base/include/media/

这里为多媒体的的底层库，编译生成 libmedia.so.这个库处于 android 多媒体架构的核心位置，它对上层提供的接口主要有 MediaPlayer、MediaRecorder、MediaScanner 等类.

android.meida.* 就是通过 libmedia_jni.so 调用 libmedia.so 实现的接口实现的。还需要注意的是 MediaPlayerInterface.h 这个头文件，他定义了 MediaPlayer 的底层接口。它主要定义了下面的类:

MediaPlayerBase: MediaPlayerInterface 的抽象基础类，里面包含了音频输出、视频输出、播放控制等的基本接口。

MediaPlayerInterface、MediaPlayerHWInterface 继承自 MediaPlayerBase 针对不同输出作出的扩展。

4.2.3 ./frameworks/base/media/libmediaplayerservice

这是多媒体的服务部分，编译生成 libmediaplayerservice.so.

MediaPlayerService.cpp 实现了一个名字为 media.player 的服务，MediaPlayer 通过 IPC 同其实现通讯，以建立合适的播放器。

在 MediaPlayerService.cpp 会根据 playerType 的类型来决定创建不同的播放器。

现有的播放器类型有 PVPlayer、MidiFile、VorbiPlayer 三个播放器类，他们都是有继承 MediaPlayerInterface 得到，具有相同的播放接口。

我们这里可以通过继承 MediaPlayerInterface 的方法，实现增加新的播放器实现。

4.2.4 ./external/opencore/

opencore 多媒体播放器部分，编译主要生成 libopencoreplayer.so。他提供给上层的主要有两个部分：

* **PVPlayer：** 提供了媒体的播放功能。
* **PVAuthor：** 提供媒体的记录的功能。

opencore/android 目录下实现了 opencore 同 libmediaplayerservice.so 的交互接口。

【Android 多媒体实例教程】

5.1 音乐播放器功能的实现。(eoe社区 ID：clarck)

音乐播放器最主要的功能有：播放、暂停、上一首、下一首、歌曲拖拽播放、在线播放、后台播放、播放歌曲信息状态栏提醒、搜索歌曲、歌词同步、扫描歌曲、换肤、设置铃声、歌曲信息、下载歌曲、均衡器调节、来电时音乐播放器的音量调节等等。

音乐播放器的功能就是前台要实现什么功能通过接口语言告诉后台，具体怎么实现让service去实现，以达到前台和服务各自实现自己的功能，又能相结合通信的目的。

其中，网络搜索要用到xml解析，换肤主要是改变background，以下是一些名词的解释：

Service： 要实现后台播放就要结合到Android中的服务
Android Interface Definition Language： 跨进程通信机制AIDL（接口描述语言）
NotificationManager： 通知管理器
Handle： 消息机制
TelephonyManager： 手机来电接收器管理
RingtoneManager： 铃声管理器
WindowManager： 屏幕亮度调节

service中通过Binder来实现和IPlayback绑定

5.1.1 AIDL的使用：

以aidl为文件结尾，定义一些供前台调用的接口，接口全部在service中实现。以达到和service进行通信的目的，实现跨进程调用。

```
IPlayback.aidl
interface IPlayback {
     //开始播放
    void start();
     //暂停播放
    void pause();
     //停止播放
    void stop();
     //释放资源
    void release();
     //上一首
    void previous();
     //下一首
    void next() ;
     //获取正在播放的音乐的ID
    int getId() ;
     //获取正在播放的音乐的标题
    java.lang.String getTitle();
    //获取正在播放的音乐的艺术家
```

```
    java.lang.String getArtist() ;
    //获取正在播放的音乐的专辑
    java.lang.String getAlbum() ;
     //获取当前播放的位置
    int getDuration();
     //获取当前播放的时间
    int getTime();
     //拖放
    void seek(int time);
     //判断是否正在播放
    boolean isPlaying() ;
}
```

下面这中方法是查询系统数据库然后返回查询结果，但是有一个很大的弊端，那就是有时候的中文乱码无法解决，最好的办法是自己做解析(以下代码来自 0Phone 应用开发权威指南)

```
public class MusicService extends Service {

   private int state=IDLE;
    private static final int PLAYING=0;
    private static final int PAUSE=1;
   private static final int STOP=2;
   private static final int IDLE=3;
    private NotificationManager nMgr;
    private Cursor cursor;
    private MediaPlayer player;

    public static final String TRACK_UPDATE="com.tymx.player.TRACK_UPDATE";

    private Binder binder=new IPlayback.Stub() {

        public void stop() throws RemoteException {
            // TODO Auto-generated method stub
            //停止播放，播放器进入到STOP状态
           player.stop();
            state=STOP;
            //取消Notification
            nMgr.cancel(R.string.notification_titile);
        }

            public void start() throws RemoteException {
                // TODO Auto-generated method stub
            if(state == STOP)
            {
```

```
            //如果当前状态为STOP，调用prepare()重新播放歌曲
            try {
                player.prepare();
            } catch (IllegalStateException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        else if (state == PAUSE) {
            player.start();
            state=PLAYING;
        }

        public void seek(int time) throws RemoteException {
            // TODO Auto-generated method stub
            player.seekTo(time);
        }

        public void release() throws RemoteException {
            // TODO Auto-generated method stub
            //释放MediaPlayer对象
            player.release();
            player=null;
            state=IDLE;
        }

        public void previous() throws RemoteException {
             // TODO Auto-generated method stub
             //播放上一首歌曲
             if(!cursor.moveToPrevious())
                 cursor.moveToLast();
             _play();
        }

        public void pause() throws RemoteException {
            // TODO Auto-generated method stub
            //暂停播放
            if (player.isPlaying()) {
                player.pause();
                state = PAUSE;
```

```
            }
        }

        public void next() throws RemoteException {
            // TODO Auto-generated method stub
            //播放下一首歌曲
            _next();
            if(player.isPlaying())
            {
                player.pause();
                state=PAUSE;
            }
        }

        public boolean isPlaying() throws RemoteException {
            // TODO Auto-generated method stub
            return player.isPlaying();
        }

        public String getTitle() throws RemoteException {
            // TODO Auto-generated method stub
            //读取正在播放歌曲的标题
            return
cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.TITLE));
        }

        public int getTime() throws RemoteException {
            // TODO Auto-generated method stub
            //获得当前的媒体时间
            return player.getCurrentPosition();
        }

        public int getId() throws RemoteException {
            // TODO Auto-generated method stub
            if(cursor.getPosition() == -1)
            {
                return -1;
            }
            //获得当前正在播放歌曲的ID
            return
cursor.getInt(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media._ID));
         }
```

```
            public int getDuration() throws RemoteException {
                // TODO Auto-generated method stub
                //获得当前歌曲的时长
                return player.getDuration();
            }

            public String getArtist() throws RemoteException {
                // TODO Auto-generated method stub
                //读取正在播放歌曲的歌手信息
                return
cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.ARTIST))
;
            }

            public String getAlbum() throws RemoteException {
                // TODO Auto-generated method stub
                //读取正在播放歌曲的专辑

                return
cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.ALBUM));
            }
    };

    @Override
    public IBinder onBind(Intent intent) {
        // TODO Auto-generated method stub
        return binder;

    }

    @Override
    public void onCreate() {
        // TODO Auto-generated method stub
        super.onCreate();
        ContentResolver contentResolver=getContentResolver();

    cursor=contentResolver.query(MediaStore.Audio.Media.EXTERNAL_CONTEENT_URI,
              null, null, null, MediaStore.Audio.Media.DEFAULT_SORT_ORDER);
    }

    @Override
    public void onDestroy() {
        // TODO Auto-generated method stub
```

```
        super.onDestroy();
        if(nMgr!=null)
            nMgr.cancel(R.string.notification_titile);
        if(player!=null)
            player.release();
    }

    @Override
    public void onStart(Intent intent, int startId) {
        // TODO Auto-generated method stub
        super.onStart(intent, startId);
        if(position!=-1)
        {
          cursor.moveToPosition(position);
          _play();
        }
    }

    private void _play() {
        // TODO Auto-generated method stub
        String
path=cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.DATA));
        try {
            if(player==null)
            {
                player=new MediaPlayer();
                player.setOnCompletionListener(completionListener);
                player.setOnPreparedListener(preParedListener);
                player.setOnErrorListener(errorListener);
            }
            else
            {
                player.reset();
            }
            player.setDataSource(path);
            player.prepare();
        } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalStateException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

```
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    private void _next()
    {
        if(!cursor.moveToNext())
            cursor.moveToFirst();
        _play();
    }

    private OnErrorListener errorListener=new OnErrorListener() {
        public boolean onError(MediaPlayer mp, int what, int extra) {
            // TODO Auto-generated method stub
            _next();
            return true;
        }
    };

    private OnCompletionListener completionListener=new OnCompletionListener() {

        public void onCompletion(MediaPlayer mp) {
            // TODO Auto-generated method stub
            if(!cursor.moveToNext())
                cursor.moveToFirst();
            _play();
        }
    };
    private OnPreparedListener preParedListener= new OnPreparedListener() {

        public void onPrepared(MediaPlayer mp) {
            // TODO Auto-generated method stub
            player.start();
            state=PLAYING;
            Intent intent=new Intent(TRACK_UPDATE);
            sendBroadcast(intent);
            showNotification();
        }
    };
    private void showNotification() {
        CharSequence title = cursor.getString(cursor
                .getColumnIndexOrThrow(MediaStore.Audio.Media.TITLE));
```

```
        Notification notifi = new Notification(R.drawable.stat_sample, title,
                System.currentTimeMillis());
        //设置Intent，以便用户可以从通知列表中返回PlayingActivity
        PendingIntent pIntent = PendingIntent.getActivity(this, 0, new
Intent(this, Main.class), 0);
        String message =
cursor.getString(cursor.getColumnIndexOrThrow(MediaStore.Audio.Media.ARTIST))
;
        notifi.setLatestEventInfo(this, title, message, pIntent);
        if (nMgr == null)
            nMgr = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        //通知用户，歌曲已经开始播放
        nMgr.notify(R.string.notification_titile, notifi);
    }

}
```

绑定服务类：

```
private IPlayback service;
//获取连接
private ServiceConnection connection=new ServiceConnection() {

        public void onServiceConnected(ComponentName arg0, IBinder arg1) {
            // TODO Auto-generated method stub
            //绑定成功后，获得IPlayback接口
            service=IPlayback.Stub.asInterface(arg1);
            //更新播放屏
            handler.sendMessage(handler.obtainMessage(UPDATE));
        }

        public void onServiceDisconnected(ComponentName name) {
            // TODO Auto-generated method stub
            service=null;
        }
   };

if(service==null)
        {
        Intent intent=new Intent(this,MusicService.class);
        intent.putExtra("position", position);
        startService(intent);
         intent.removeExtra("position");
      //绑定到服务
```

```
            if(false==bindService(intent, connection,
            Context.BIND_AUTO_CREATE))
          {
             finish();
          }
         }
```

5.1.2 在项目开发中通过真机测试的一些内容，由于公司有保密协议，所以只能提供部分代码。

1、TelephonyManager：手机来电接收器管理

```
class IncomingSMSReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            // TODO Auto-generated method stub
            if (intent.getAction().equals("android.intent.action.ANSWER")) {
//手机来电接收器管理
                TelephonyManager telephonymanager = (TelephonyManager) context
                        .getSystemService(Context.TELEPHONY_SERVICE);
                int callMode = Integer.parseInt(sp.getString(
                        getString(R.string.callactionkey), "0"));
                //当前音量大小
                 current = am.getStreamVolume(AudioManager.STREAM_MUSIC);
                 switch (telephonymanager.getCallState()) {
                 case TelephonyManager.CALL_STATE_RINGING:
                    // 根据设置模式 完成不同的动作
                    switch (callMode) {
                    case 1:
                        // 暂停播放
                        smplayer.pause();
                        break;
                    case 2:
                        // 调低音量
                        am.setStreamVolume(AudioManager.STREAM_MUSIC, 1, 0);
                        break;
                    default:
                        break;
                    }
                    break;
                 case TelephonyManager.CALL_STATE_OFFHOOK: // 挂机后的处理
                    // 根据设置模式 完成不同的动作
                    switch (callMode) {
                    case 1:
                        // 重新开始播放
```

```
                    smplayer.resume();
                    break;
                case 2:
                    // 还原音量
                    am.setStreamVolume(AudioManager.STREAM_MUSIC, current,
                            0);
                    break;
                default:
                    break;
                }
                break;
            default:
                break;
            }
        }
    }
}
```

2、铃声管理器： RingtoneManager

```
//来电铃声:
RingtoneManager.setActualDefaultRingtoneUri(MainView.this,
RingtoneManager.TYPE_RINGTONE,Uri.parse("file:////sdcard/ringtone/music.mp3");

//通知铃声,
RingtoneManager.setActualDefaultRingtoneUri(MainView.this,
RingtoneManager.TYPE_NOTIFICATION,Uri.parse("file:////sdcard/ringtone/music.mp3");

//闹钟铃声
RingtoneManager.setActualDefaultRingtoneUri(MainView.this,
RingtoneManager.TYPE_ALARM,Uri.parse("file:////sdcard/ringtone/music.mp3")
;

//所有铃声
RingtoneManager.setActualDefaultRingtoneUri(MainView.this,
RingtoneManager.TYPE_ALL,Uri.parse("file:////sdcard/ringtone/music.mp3");
```

3、屏幕亮度管理： WindowManager

```
WindowManager.LayoutParams lp = getWindow().getAttributes();
lp.screenBrightness = 1.0f;  //1.0f 表示亮度为最亮、0.1f 表示最暗
getWindow().setAttributes(lp);
```

4、打开系统设置界面：

```
Intent intent=new Intent();
intent.setComponent(new ComponentName("com.android.settings",
"com.android.settings.Settings"));
 startActivity(intent);
```

5、进入无线网络配置界面：

```
startActivity(new Intent(Settings.ACTION_WIRELESS_SETTINGS));
```

5.2 自动下载歌词与歌词的解析。（eoe社区 ID：clarck）

研究了对于歌词的下载和歌词的解析，网上找了好久没有找到很好的方法，于是自己研究了一下，贡献上自己的方法，希望以后有更好的方法。

5.2.1 歌词下载代码：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class HttpDownloader {
    private static URL url = null;

    /**
     * 根据URL下载文件，前提是这个文件当中的内容是文本，函数的返回值就是文件当中的内容
     * 1.创建一个URL对象
     * 2.通过URL对象，创建一个HttpURLConnection对象
     * 3.得到InputStram
     * 4.从InputStream当中读取数据
     * @param urlStr
     * @return
     */
    public String download(String urlStr) {
        StringBuffer sb = new StringBuffer();
        String line = null;
        BufferedReader buffer = null;
        try {
            // 创建一个URL对象
            url = new URL(urlStr);
            // 创建一个Http连接
            HttpURLConnection urlConn =(HttpURLConnection) url.openConnection();
```

```
            // 使用IO流读取数据
            buffer = new BufferedReader(new
InputStreamReader(urlConn.getInputStream()));
            while ((line = buffer.readLine()) != null) {
                sb.append(line+"\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                buffer.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return sb.toString();
    }

    public static void main(String[] args) {
    //这里通过拼参数进去，得到URL地址
        String
Url=http://mp3.sogou.com/gecisearch.so?hl=zh-cn&query=”+MusicName+”&as=false&
st=&ac=1&pf=&class=5&gecisearch.so=";
        HttpDownloader downloader=new HttpDownloader();
        String ne=downloader.download(Url);
       //通过正则表达式来匹配到歌词的具体下载地址
        Pattern pattern = Pattern.compile("((downlrc.jsp?).*(LRC歌词下载))");
        Matcher matcher= pattern.matcher(ne);
        if(matcher.find())
        {
            String wangzhi=matcher.group(0);
            String down=wangzhi.substring(0, wangzhi.indexOf('"'));
            System.out.println(ne);
            //拼接上host
            String down_address = "http://mp3.sogou.com/"+down ;
            System.out.println(downloader.download(down_address));
        }
    }
```

5.2.2 实例讲述该如何自动下载

1、歌曲名为：后来

直接讲 MusicName 改为：“后来” 然后拼接进去。

以下是得到的结果：

```
[by:昨夜星辰昨夜风]
[00:03.21]刘若英:后来
[05:16.43][00:05.26]词:施人诚 曲:玉城千春 编曲:王继康
[04:03.05][03:37.50][02:33.36][00:12.18]后来,
[04:04.79][03:39.30][02:35.11][00:14.26]我总算学会了如何去爱。
[04:09.72][03:44.17][02:40.10][00:19.01]可惜你早已远去,
[04:12.56][03:46.92][02:42.88][00:21.87]消失在人海。
[04:15.82][03:50.10][02:46.12][00:25.14]后来,
[04:17.82][03:52.09][02:48.08][00:27.05]终于在眼泪中明白,
[04:22.53][03:56.77][02:52.83][00:31.93]有些人一旦错过就不再。
[00:39.67]槐子花,白花瓣,
[00:44.99]落在我蓝色百褶裙上。
[00:51.41]“爱你!”你轻声说。
[00:58.61]我低下头闻见一阵芬芳。
[01:04.92]那个永恒的夜晚,
[01:08.97]十七岁仲夏,
[01:11.88]你吻我的那个夜晚,
[01:17.98]让我往后的时光,
[01:21.31]每当有感叹,
[01:24.34]总想起当天的星光。
[01:30.84]那时候的爱情,
[01:36.46]为什么就能那样简单。
[01:42.48]而又是为什么人年少时,
[01:49.96]一定要让深爱的人受伤。
[01:56.06]在这相似的深夜里,
[02:00.32]你是否一样,
[02:03.32]也在静静追悔感伤。
[02:09.06]如果当时我们能不那么倔强,
[02:15.86]现在也不那么遗憾。
[03:25.13][02:20.91]你都如何回忆我,
[03:27.92][02:23.94]带著笑或是很沉默。
[03:31.14][02:27.19]这些年来有没有人能让你不寂寞。
[04:29.45]永远不会再重来,
[04:35.34]有一个男孩爱著那个女孩。
```

然后将该内容保存为.“后来.lrc”文件，以供解析。

2、解析歌词文件：

1）保存歌词的实体类：

```
public class Statement {
    private double time=0.0;                    //时间，单位为s，精确到0.01s
    private String lyric="";                    //歌词

    /*
     * 获取时间
     */
    public double getTime() {
        return time;
    }
    /*
     * 设置时间
     * time：被设置成的时间
     */
    public void setTime(double time) {
        this.time = time;
    }
    /*
     * 设置时间
     * time：被设置成的时间字符串，格式为mm:ss.ms
     */
    public void setTime(String time) {
        String str[] = time.split(":|\\.");
        this.time                                                    =
Integer.parseInt(str[0])*60+Integer.parseInt(str[1])+Integer.parseInt(str[2])
*0.01;
    }
    /*
     * 获取歌词
     */
    public String getLyric() {
        return lyric;
    }
    /*
     * 设置歌词
     */
    public void setLyric(String lyric) {
        this.lyric = lyric;
    }
    /*
```

```
     * 打印歌词
     */
    public void printLyric()
    {
        System.out.println(time+": "+lyric);
    }
}
```

2）具体解析类：

```
BufferedReader bufferReader = null;                                    //读取文件实例
    public String title = "";                                          //歌曲题目
    public String artist = "";                                         //演唱者
    public String album = "";                                          //专辑
    public String lrcMaker = "";                                       //歌词制作者
    Vector<Statement> statements = new Vector<Statement>();            //歌词

    /*
     * 实例化一个歌词数据. 歌词数据信息由指定的文件提供.
     * fileName: 指定的歌词文件.
     */

    public LrcData(String fileName) throws IOException{
         FileInputStream file = new FileInputStream(fileName);
        bufferReader = new BufferedReader(new InputStreamReader(file, "utf-8"));

        //将文件数据读入内存
        readData();
    }

    /*
     * 读取文件中数据至内存.
     */
    public void readData() throws IOException{
        statements.clear();
        String strLine;
        //循环读入所有行
        while(null != (strLine = bufferReader.readLine()))
        {
            //判断该行是否为空行
            if("".equals(strLine.trim()))
            {
                continue;
            }
```

```
            //判断该行数据是否表示歌名
            if(null == title || "".equals(title.trim()))
            {
                Pattern pattern = Pattern.compile("\\[ti:(.+?)\\]");
                Matcher matcher = pattern.matcher(strLine);
                if(matcher.find())
                {
                    title=matcher.group(1);
                    continue;
                }else
                {
                    title="未知歌名";
                    continue;
                }
            }

            //判断该行数据是否表示演唱者
            if(null == artist || "".equals(artist.trim()))
            {
                Pattern pattern = Pattern.compile("\\[ar:(.+?)\\]");
                Matcher matcher = pattern.matcher(strLine);
                if(matcher.find())
                {
                    artist=matcher.group(1);
                    continue;
                }else
                {
                    artist="未知演唱者";
                    continue;
                }
            }
            //判断该行数据是否表示专辑
            if(null == album || "".equals(album.trim()))
            {
                Pattern pattern = Pattern.compile("\\[al:(.+?)\\]");
                Matcher matcher = pattern.matcher(strLine);
                if(matcher.find())
                {
                    album=matcher.group(1);
                    continue;
                }else
                {
                    album="未知专辑";
```

```
                continue;
            }
        }
        //判断该行数据是否表示歌词制作者
        if(null == lrcMaker || "".equals(lrcMaker.trim()))
        {
            Pattern pattern = Pattern.compile("\\[by:(.+?)\\]");
            Matcher matcher = pattern.matcher(strLine);
            if(matcher.find())
            {
                lrcMaker=matcher.group(1);
                continue;
            }else
            {
                lrcMaker="未知歌词制作者";
                continue;
            }

        }
        //读取并分析歌词
        int timeNum=0;                                      //本行含时间个数
        String str[] = strLine.split("\\]");                //以]分隔
        for(int i=0; i<str.length; ++i)
        {
            String str2[] = str[i].split("\\[");            //以[分隔
            str[i] = str2[str2.length-1];
            if(isTime(str[i])){
                ++timeNum;
            }
        }
        for(int i=0; i<timeNum;++i)                         //处理歌词复用的情况
        {
            Statement sm = new Statement();
            sm.setTime(str[i]);
            if(timeNum<str.length)                          //如果有歌词
            {
                sm.setLyric(str[str.length-1]);
            }
            statements.add(sm);
        }
         if(1==str.length)                                  //处理没有歌词的情况
        {
             Statement sm = new Statement();
```

```
                sm.setTime(str[0]);
                sm.setLyric("没有歌词");
                statements.add(sm);
            }
        }

        //将读取的歌词按时间排序
        sortLyric();
    }
    /*
     * 判断给定的字符串是否表示时间.
     */
    public boolean isTime(String string)
    {
        String str[] = string.split(":|\\.");
        if(3!=str.length)
        {
            return false;
        }
        try{
            for(int i=0;i<str.length;++i)
            {
                Integer.parseInt(str[i]);
            }
        }
        catch(NumberFormatException e)
        {
            return false;
        }
        return true;
    }
    /*
     * 将读取的歌词按时间排序.
     */
    public void sortLyric()
    {
        for(int i=0;i<statements.size()-1;++i)
        {
            int index=i;
            double delta=Double.MAX_VALUE;
            boolean moveFlag = false;
            for(int j=i+1;j<statements.size();++j)
            {
```

```
            double sub;

    if(0>=(sub=statements.get(i).getTime()-statements.get(j).getTime()))
            {
                continue;
            }
            moveFlag=true;
            if(sub<delta)
            {
                delta=sub;
                index=j+1;
            }
        }
        if(moveFlag)
        {
            statements.add(index, statements.elementAt(i));
            statements.remove(i);
            --i;
        }
    }
}

/*
 * 打印整个歌词文件
 */
public void printLrcDate()
{
    System.out.println("歌曲名: "+title);
    System.out.println("演唱者: "+artist);
    System.out.println("专辑名: "+album);
    System.out.println("歌词制作: "+lrcMaker);
    for(int i=0;i<statements.size();++i)
    {
        statements.elementAt(i).printLyric();
    }
}

/*
 * 测试整个类
 */
public static void main(String[] args) throws IOException {
    {
        LrcData ld = new LrcData("F://歌词//后来.lrc"); //路径\\输入文件名
```

```
            ld.printLrcDate();
        }
    }
```

3）解析得到的结果：

歌曲名:
演唱者:
专辑名:
歌词制作：昨夜星辰昨夜风
3210 刘若英:后来
5260 词:施人诚 曲:玉城千春 编曲:王继康
12180 后来,
14260 我总算学会了如何去爱。
19010 可惜你早已远去,
21870 消失在人海。
25140 后来,
27050 终于在眼泪中明白,
31930 有些人一旦错过就不再。
39670 槐子花,白花瓣,
44990 落在我蓝色百褶裙上。
51410“爱你!”你轻声说。
58610 我低下头闻见一阵芬芳。
64920 那个永恒的夜晚,
68970 十七岁仲夏,
71880 你吻我的那个夜晚,
77980 让我往后的时光,
81310 每当有感叹,
84340 总想起当天的星光。
90840 那时候的爱情,
96460 为什么就能那样简单。
102480 而又是为什么人年少时,
109960 一定要让深爱的人受伤。
116060 在这相似的深夜里,
120320 你是否一样,
123320 也在静静追悔感伤。
129060 如果当时我们能不那么倔强,
135860 现在也不那么遗憾。
140910 你都如何回忆我,
143940 带著笑或是很沉默。
147190 这些年来有没有人能让你不寂寞。
153360 后来,
155110 我总算学会了如何去爱。
160100 可惜你早已远去,
162880 消失在人海。

166120 后来，
168080 终于在眼泪中明白，
172830 有些人一旦错过就不再。
205130 你都如何回忆我，
207920 带著笑或是很沉默。
211140 这些年来有没有人能让你不寂寞。
217500 后来，
219300 我总算学会了如何去爱。
224170 可惜你早已远去，
226920 消失在人海。
230100 后来，
232090 终于在眼泪中明白，
236770 有些人一旦错过就不再。
243050 后来，
244790 我总算学会了如何去爱。
249720 可惜你早已远去，
252560 消失在人海。
255820 后来，
257820 终于在眼泪中明白，
262530 有些人一旦错过就不再。
269450 永远不会再重来，
275340 有一个男孩爱著那个女孩。
316430 词:施人诚 曲:玉城千春 编曲:王继康

附录：多媒体例程可独立编译的 android 多媒体例程，源文件一份。
MusicPlayer，源文件一份。

【其他】

6.1 提交 BUG

如果你发现文档中有不妥的地方，请发邮件至 eoeandroid@eoemobile.com 进行反馈，我们会定期更新、发布更新后的版本。

6.2 关于 eoeAndroid

eoeAndroid 是国内成立最早，规模最大的 Android 开发者社区，拥有海量的 Android 学习资料。分享、互助的氛围，让 Android 开发者迅速成长，逐步从懵懂到了解，从入门到开发出属于自己的应用，eoeAndroid 为广大 Android 开发者奠定坚实的技术基础。从初级到高级，从环境搭建到底层架构，eoeAndroid 社区为开发者精挑细选了丰富的学习资料和实例代码。让 Android 开发者在社区中迅速的成长，并在此基础上开发出更多优秀的 Android 应用。

6.3 eoe Android 移动互联高峰论坛即将开始！

你一定想知道，投资者是怎样看待中国的移动互联网；移动互联网的前景是什么？让我们时下火热的 LBS 服务来看看；Android 平台真的不盈利吗？让移动广告商来告诉你……对于移动互联网的前路，似乎已经慢慢开始变得清晰。我们要做的是，立于目前，一起探讨和思考，用技术创新去推动它的发展。

在不久的未来，我们势必将见证一批伟大的移动互联网企业诞生，且不要急，我们还有很多的事要做，那么先从第一步开始。2011 年 4 月 9 日，相约 eoe android 移动互联高峰论坛，一起来深圳吧。

详情请点击：http://www.eoeandroid.com/thread-63736-1-1.html

6.4 eoe：Android 传教士

eoe 创始人靳岩早年在一家中间件公司，2007 年第一次体验了 iPhone，就被其强大的功能和体验深深吸引了。他认识到，智能手机将是未来的发展方向，但苹果的生态系统是封闭的，并且定位偏向时尚人群，而谷歌的 Android 系统由于自身的开源性，具有更大的市场机会。

2009 年 4 月，靳岩和另外一个创始人姚尚朗一起创办了 Android 论坛，即现在 Android 开发者社区的前身。他们立志做 Android 传教士，撰写了中国第一本 Android 技术教程《Android 开发入门与实战》。如今，这本书已经成为开发者的必备参考书之一。

详情请点击：http://www.eoeandroid.com/thread-63737-1-1.html

北京易联致远无限技术有限公司

责任编辑：莫言默语

美术支持：阿彦　　**技术支持：**gaotong86

中国最大的 Android 开发者社区　：www.eoeandroid.com

中国本土的 Android 软件下载平台：www.eoemarket.com