《ANDROID 多媒体编程从初学到精通》

作者：华清远见

第 4 章　多媒体框架

君子务本，本立而道生。　--　《论语》

在目前业界主流的多媒体处理框架中，Windows 通常采用的是 DirectShow，而桌面 Linux 上的多媒体处理框架则较多，其中最常见的为 GStearmer、xine 等。在嵌入式 Linux 领域，Qtopia 平台采用的多媒体处理框架为 GStearmer。Nokia 开发的 Maemo Linux 系统采用的多媒体处理框架也是 GStearmer。在 Android 中，采用的多媒体处理框架为 OpenCORE。

在 Android 2.2 后，Android 对多媒体框架进行了很大的调整，弃用了之前的 openCORE 框架，默认改用 stagefright 框架，仅仅对 opencore 中的 omx-component 部分做了引用。主要是为了录像和视频电话功能，另外在混音和多摄像头支持方面也做了增强。stagefright 框架相对也比 openCORE 框架更加易懂，封装也相对简单。但 stagefright 框架推出时间不长，支持的文件格式也不如 openCORE 框架丰富。

在 Android 2.2 及以前，OpenCORE 位于 external 目录下，在 Android 2.3 以后，多媒体的功能被放置到 frameworks/base/media 中，OpenCore 的概念被弱化。在本书中，多媒体框架部分将主要依据 OpenCORE 进行讲解。

在 OpenCORE 的 pvmf_format_type.h 文件中，给出了 OpenCORE 目前所支持的图像、音频、视频编码类型。

在 Android2.2 中，目前内置支持的解码媒体格式包括 AAC LC/LTP、HE-AACv1 (AAC+)、HE-AACv2 (enhanced AAC+)、AMR-NB、AMR-WB、MP3、MIDI、Ogg Vorbis、PCM/WAVE、JPEG、GIF、PNG、BMP、H.263、H.264 AVC、MPEG-4 SP 等，部分厂商在开发的 Android 智能终端中已经提供了对 RM 的支持。

内置支持的编码格式包括 AAC LC/LTP、AMR-NB、AMR-WB、PNG、JPEG、H.263 等，如果期望在产品中支持更多的媒体格式，只需增加相应的编解码器即可。

4.1 框架概述

在 Android 中，与桌面 Linux 和 Qtopia 通常采用 GStearmer、xine、MPlayer 不同，Android 采用的是基于 PacketVideo 的 OpenCORE 的多媒体框架方案。由于 OpenCORE 高度的模块化，采用 OpenCORE 作为多媒体框架有利于加速将产品推向市场、减小操作成本和资源投入、扩展编解码器、增强用户体验等。

在对数据源的支持上，除了本地文件，OpenCORE 还对内容目录、流媒体、OTA（Over-the-Air）下载、DRM 等提供了支持。目前 OpenCORE 已经应用于 170 多款移动终端中。

2009 年 2 月发布的 OpenCORE 2.0 已经提供了对基于 3G-324M 协议的视频电话的支持，同时在多媒体硬件加速方面进行了优化。

需要说明的是，OpenCORE 遵循 OpenMAX 的接口规范，本质上是 OpenMAX 的一种实现。关于 OpenCORE 在 Android 中的编译，以及如何编译测试项请参考 android\ external\opencore\ Android.mk 和 android\external\opencore\quick_start.txt 文件。

4.2 OpenMAX 接口规范

OpenMAX 是 NVIDIA 和 Khronos 在 2006 年制定的多媒体处理框架规范，同时 Khronos 制定的标准/规范还有 OpenGL、OpenGL ES、OpenVG、OpenEL 等。OpenGL ES 在 Android 中作为 3D 渲染引擎使用。

OpenMAX 是一个无须授权费的、跨平台的应用程序接口规范，该规范针对嵌入式设备、移动设备的多媒体软件架构。在 OpenMAX 架构中，为多媒体的编解码器和数据处理定义了一套统一的集成接口（OpenMAX IL），通过对底层硬件的多媒体数据的处理功能进行系统级抽象，为用户屏蔽了底层的细节。因此，多媒体应用程序和多媒体框架通过 OpenMAX IL 可以以一种统一的方式来使用编解码器和其他多媒体数据处理功能，这使得 OpenMAX 拥有跨平台的能力。OpenMAX 的框架如图 4-1 所示。

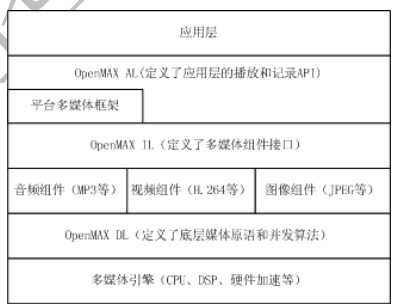

图 4-1 OpenMAX 框架

下面简要介绍 OpenMAX 的 OpenMAX AL、OpenMAX IL、OpenMAX DL 实现。关于更详细的说明请参见 android\external\opencore\doc 目录下的文档。

4.2.1 OpenMAX AL 应用层

OpenMAX AL（Appliction Layer）API 在应用程序和 OpenMAX IL 之间提供了一个标准化接口，OpenMAX IL 提供服务以实现被期待的 API 功能。使得应用在多媒体接口上具有了可移植性。

OpenMAX AL 包括引擎对象（Engine Object）、媒体对象（Media Object）、元数据提取器对象（Metadata Extractor Object）、音频输出混音器对象（Audio Output Mix Objects）、照相机对象（Camera Objects）、LED 阵列对象（LED Array Objects）、FM 对象（Radio Objects）、振动控制对象（Vibration Control Objects）等。

在 Android 中，并没有提供多少关于 OpenMAX AL 的内容，这里就不再详述了，OpenMAX AL 的头文件位于 external\opencore\extern_libs_v2\khronos\openmax\include 目录下。关于 OpenMAX AL 的更多内容，请参考文献[35]。

为了实现封装的编解码器给上层提供一个标准化的接口，在 Android 中，提供了 AuthorDriver 作为记录引擎和上层应用的接口，PlayerDriver 作为播放引擎与上层应用的接口。

4.2.2 OpenMAX IL 集成层

OpenMAX IL（Integration Layer）作为在嵌入式和移动设备中使用的音频、视频、图像等编解码器的底层接口。使得应用和多媒体框架可以以统一的方式访问多媒体编解码器和支持组件。编解码器可以是硬件和软件的任意组合，对用户透明。

为了把一个编解码器集成到 OpenCORE 多媒体框架中，有多种途径，可以将编解码器封装成一个媒体 I/O 的组件，也可以封装成一个 OpenCORE 的节点，或者作为 OpenMAX 的组件被集成到 OpenMAX 的编解码器节点中。

对于包含了硬件加速的编解码器而言，通常会被封装为 OpenMAX 的组件出现。这就要求其必须遵守 OpenMAX IL 的接口规范。OpenMAX IL 的接口 OpenMax Core 在 OMX_Core.h 中定义。OpenMAX 的组件在 OMX_Component.h 中定义。OpenMAX 的组件框架如图 4-2 所示。

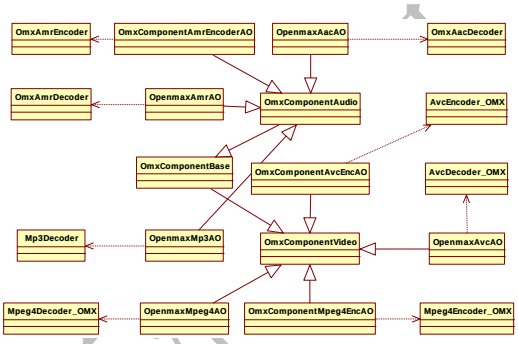

图 4-2 多媒体组件框架

在 OpenMAX IL 的接口规范中，有些接口是厂商必须提供的，必须提供的方法如下：

```
1）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Init(void);
2）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Deinit(void);
3）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetHandle(…);
4）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_FreeHandle(…);
5）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_ComponentNameEnum(...);
6）OMX_API OMX_ERRORTYPE OMX_GetRolesOfComponent (…);
7）OMX_API OMX_ERRORTYPE OMX_GetComponentsOfRole (…);
8）OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_SetupTunnel(…);
9）OMX_API OMX_ERRORTYPE OMX_GetContentPipe(…);
```

除以上方法必须实现外，为了使 OpenCORE 能够获悉 OpenMAX Core 的配置信息，OMXConfigParser()被强烈推荐实现。OMXConfigParser()的定义位于 pv_omx_config_parser.h 文件中。

在 OpenCORE 中，关于 OpenMAX Core 的实现位于 pv_omxcore.cpp 文件中。考虑在实际开发中，可能存在由多个厂商开发的 OpenMAX Core 实现的可能，为了避免造成静态编译时的链接问题，厂商在实现 OpenMAX Core 时，应考虑增加一个简单的封装层。以封装 OpenMAX Core 的标准接口。如在 OpenCORE 中，OMX_Init()接口的实现为：

```
OSCL_EXPORT_REF OMX_ERRORTYPE OMX_MasterInit()
{
   return OMX_Init(); // OMX_Init()的具体实现位于同文件中
}
```

在 OpenCORE 中，为上层提供的封装接口为 OMXInterface。由于采用的是动态加载的方法，考虑到可能存在多家厂商的 OpenMAX Core 情况，OpenMAX Core 必须以动态库的方式出现。在 OpenCORE 中，

提供了两种编译模式。

一种编译模式是封装器和 OpenMAX Core 共享库分别编译。在 OpenMAX Core 动态库中并不包含封装器。该编译模式的实现位于 OpenCORE\codecs_v2\omx\ omx_core_ plugins\template\src\pv_omx_interface.cpp 中。在 PVOMXInterface 类的构造函数中，利用 dlopen()函数以 RTLD_NOW 模式打开 OpenMAX Core 共享库。为 OpenMAX IL 接口利用 dlsym()函数查找对应的符号进行赋值。具体实现为：

代码 4-1 封装器和共享库分别编译

```
#ifndef OMX_CORE_LIBRARY
#define OMX_CORE_LIBRARY "libOmxCore.so"
#endif
class PVOMXInterface : public OMXInterface
{
    public:
    ……
    private:
       PVOMXInterface()
      {
          ipHandle=dlopen(OMX_CORE_LIBRARY, RTLD_NOW);//打开共享库
           if (NULL==ipHandle)
           {
               pOMX_Init=NULL;
               pOMX_Deinit=NULL;
               pOMX_ComponentNameEnum=NULL;
               pOMX_GetHandle=NULL;
               pOMX_FreeHandle=NULL;
               pOMX_GetComponentsOfRole=NULL;
               pOMX_GetRolesOfComponent=NULL;
               pOMX_SetupTunnel=NULL;
               pOMX_GetContentPipe=NULL;
               pOMXConfigParser=NULL;
               const char* pErr=dlerror();
               if (NULL==pErr)
               {
                       ……
               }
               else
               {
                           ……
               }
           }
           else
           {
              //加载 OMX core 符号
               pOMX_Init=(tpOMX_Init)dlsym(ipHandle, "OMX_Init");
               pOMX_Deinit=(tpOMX_Deinit)dlsym(ipHandle, "OMX_Deinit");
               pOMX_ComponentNameEnum=(tpOMX_ComponentNameEnum)dlsym(ipHandle,
    "OMX_ComponentNameEnum");
               pOMX_GetHandle=(tpOMX_GetHandle)dlsym(ipHandle, "OMX_GetHandle");
               pOMX_FreeHandle=(tpOMX_FreeHandle)dlsym(ipHandle, "OMX_FreeHandle");
               pOMX_GetComponentsOfRole=(tpOMX_GetComponentsOfRole)dlsym(ipHandle,
    "OMX_GetComponentsOfRole");
               pOMX_GetRolesOfComponent=(tpOMX_GetRolesOfComponent)dlsym(ipHandle,
    "OMX_GetRolesOfComponent");
              pOMX_SetupTunnel=(tpOMX_SetupTunnel)dlsym(ipHandle, "OMX_SetupTunnel");
               pOMX_GetContentPipe=(tpOMX_GetContentPipe)dlsym(ipHandle,
    "OMX_GetContentPipe");
             pOMXConfigParser=(tpOMXConfigParser)dlsym(ipHandle, "OMXConfigParser");
           }
      };
    }
```

另一种编译模式是封装器和OpenMAX Core共享库同时编译。在OpenMAX Core动态库中包含封装器。该编译模式的实现位于 OpenCORE\codecs_v2\omx\ omx_ sharedlibrary\interface\src\ pv_omx_interface.cpp 中。在该编译模式下，只能同时编译一个 OpenMAX Core 动态库。具体实现为：

代码 4-2　封装器和共享库同时编译

```
class PVOMXInterface : public OMXInterface
{
   public:
        ……
            private:
      PVOMXInterface()
      {
         //直接赋值
         pOMX_Init=OMX_Init;
         pOMX_Deinit=OMX_Deinit;
         pOMX_ComponentNameEnum=OMX_ComponentNameEnum;
         pOMX_GetHandle=OMX_GetHandle;
         pOMX_FreeHandle=OMX_FreeHandle;
         pOMX_GetComponentsOfRole=OMX_GetComponentsOfRole;
         pOMX_GetRolesOfComponent=OMX_GetRolesOfComponent;
         pOMX_SetupTunnel=OMX_SetupTunnel;
         pOMX_GetContentPipe=OMX_GetContentPipe;
         pOMXConfigParser=OMXConfigParser;
      };
};
```

需要注意的是，不管是何种编译模式，实现上均采用了单子模式（Singleton Pattern）的设计方法。在运行期间，PVOMXInterface 仅有唯一对象出现。

为了使集成的编解码器能够在系统中运行，必须对编解码器进行注册，在 OpenCORE 中，已经提供了 AVC、M4V（Apple 公司开发）、H.263、WMA、AAC、AMR、MP3、WMA、RV、RA 等格式的解码器，提供了 AVC、M4V、H.263、AMR、AAC 等格式的编码器。在 OpenMAX IL 层中，编解码器均是在 Open Core 的 OMX_Init() 函数中进行注册的，该函数的实现位于 external\opencore\codecs_v2\omx\omx_common\src\pv_omxcore.cpp 文件中，而编解码器的注册函数则位于 external\opencore\codecs_v2\omx\omx_common\src\ pv_omxregistry.cpp 文件中。编解码器注册的信息位于 ComponentRegistrationType 的对象中，ComponentRegistrationType 的定义如下：

代码 4-3　ComponentRegistrationType

```
class ComponentRegistrationType
{
   public:
      OMX_STRING ComponentName; //组件名
      OMX_STRING RoleString[MAX_ROLES_SUPPORTED];
      OMX_U32 NumberOfRolesSupported;      //角色数量
      OMX_ERRORTYPE(*FunctionPtrCreateComponent)(OMX_OUT OMX_HANDLETYPE* pHandle,
OMX_IN  OMX_PTR pAppData,
      OMX_PTR pProxy,OMX_STRING aOmxLibName,OMX_PTR &aOmxLib, OMX_PTR aOsclUuid,
OMX_U32 &aRefCount);//创建组件
       OMX_ERRORTYPE(*FunctionPtrDestroyComponent)(OMX_IN OMX_HANDLETYPE pHandle,
OMX_PTR &aOmxLib, OMX_PTR aOsclUuid, OMX_U32 &aRefCount);//销毁组件
      void GetRolesOfComponent(OMX_STRING* aRole_string)
      {
         for (OMX_U32 ii=0; ii<NumberOfRolesSupported; ii++)
         {
            aRole_string[ii]=RoleString[ii];
         }
      }
        //用于动态加载
      OMX_STRING          SharedLibraryName;          //共享库名
```

```
        OMX_PTR             SharedLibraryPtr;           //共享库指针
        OMX_PTR             SharedLibraryOsclUuid;      //共享库 UUID
        OMX_U32             SharedLibraryRefCounter;    //共享库引用计数
    };
```

为了进行注册，首先需要创建一个 ComponentRegistrationType 对象，对其成员进行赋值，然后将其添加到 OMXGlobalDatad 的 ipRegTemplateList[MAX_SUPPORTED_COMPONENTS]成员中。需要说明的是，在组件库中的每个组件均有唯一的一个 UUID 来标识，以 MP3 解码器为例的注册过程如下：

代码 4-4　MP3Register

```
    OMX_ERRORTYPE Mp3Register(OMXGlobalData *data)
    {
        OMX_S32 ii;
        ComponentRegistrationType     *pCRT=(ComponentRegistrationType     *)     oscl_malloc(sizeof
(ComponentRegistrationType));
        if (pCRT)
        {
            pCRT->ComponentName=(OMX_STRING)"OMX.PV.mp3dec";       //组件名
            pCRT->RoleString[0]=(OMX_STRING)"audio_decoder.mp3";   //角色名
            pCRT->NumberOfRolesSupported=1;                        //角色数量
            pCRT->SharedLibraryOsclUuid=NULL;
    #if USE_DYNAMIC_LOAD_OMX_COMPONENTS
              //构造组件
            pCRT->FunctionPtrCreateComponent=&OmxComponentFactoryDynamicCreate;
              //析构组件
            pCRT->FunctionPtrDestroyComponent=&OmxComponentFactoryDynamicDestructor;
              //共享库名
            pCRT->SharedLibraryName=(OMX_STRING)"libomx_mp3dec_sharedlibrary";
            pCRT->SharedLibraryPtr=NULL;
              //UUID
            OsclUuid *temp=(OsclUuid *) oscl_malloc(sizeof(OsclUuid));
            if (temp==NULL)
            {
              //释放已分配内存
               oscl_free(pCRT);
               return OMX_ErrorInsufficientResources;
            }
            OSCL_PLACEMENT_NEW(temp, PV_OMX_MP3DEC_UUID);
            pCRT->SharedLibraryOsclUuid=(OMX_PTR) temp;
            pCRT->SharedLibraryRefCounter=0;
    #endif
    #if REGISTER_OMX_MP3_COMPONENT
    #if (DYNAMIC_LOAD_OMX_MP3_COMPONENT==0)
            pCRT->FunctionPtrCreateComponent=&Mp3OmxComponentFactory;
            pCRT->FunctionPtrDestroyComponent=&Mp3OmxComponentDestructor;
            pCRT->SharedLibraryName=NULL;
            pCRT->SharedLibraryPtr=NULL;
            if (pCRT->SharedLibraryOsclUuid)
               oscl_free(pCRT->SharedLibraryOsclUuid);
            pCRT->SharedLibraryOsclUuid=NULL;
            pCRT->SharedLibraryRefCounter=0;
    #endif
    #endif
        }
        else
        {
            return OMX_ErrorInsufficientResources;
        }
        for (ii=0; ii<MAX_SUPPORTED_COMPONENTS; ii++)    //注册组件
        {
            if (NULL==data->ipRegTemplateList[ii])
```

```
            {
                data->ipRegTemplateList[ii]=pCRT;
                break;
            }
        }
        if (MAX_SUPPORTED_COMPONENTS==ii)
        {
            return OMX_ErrorInsufficientResources;
        }
        return OMX_ErrorNone;
    }
```

每一个编解码器在 OpenMAX 中都对应唯一的一个 UUID 标识符。这些 UUID 标识符定义位于 pv_omxcore.h 文件中。

4.2.3 OpenMAX DL 开发层

OpenMAX DL（Development Layer）定义了一套 API，包含了音频、视频和图像功能的函数集合，这些函数需要由芯片厂商针对处理器特性进行实现和优化，然后被编解码器厂商在各种编解码器上使用。

OpenMAX DL 涵盖了音频信号的处理功能，如 FFT、过滤器等；图像处理功能，如颜色空间转换等；视频处理功能，如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编解码器的优化。OpenMAX 通过 iDL 和 aDL 来支持加速， iDL 使用 OpenMAX IL 结构， aDL 向 Open MAX DL API 增加了异步接口。

由于 OpenMAX DL 涉及太多的硬件细节，这里就不再介绍了。

4.3 OpenCORE 框架

OpenCORE 的实现是基于 C++语言的，要求平台必须支持 C++模板，但并非所有的 C++标准（如运行时类型识别（RTTI，Run Time Type Indication））都要求平台支持。

根据层次划分，OpenCORE 主要分为内容策略管理（Content Pollcy Manager）、多媒体引擎（MultiMedia Engines）、数据格式解析器（Data Formats Parser）、视频编解码器（Video Codecs）、音频编解码器（Audio Codecs）、操作系统兼容库（OSCL, Operating System Compatibility Library）等几个部分。OpenCORE 的框架如图 4-3 所示。

图 4-3 OpenCORE 框架

其中，内容策略管理允许移动终端支持多种商业模型和商业规则。

多媒体引擎分为两个部分：PVPlayer 和 PVAuthor。其中，PVPlayer 提供媒体播放器的功能，完成各种音频、视频流的回放（Playback）功能；PVAuthor 提供媒体流记录的功能，完成各种音频（Audio）、视频（Video）流的功能及静态图像捕获功能。

数据格式解析器则负责文件格式的解析。

视频编解码器、音频编解码器则完成压缩流和元数据流之间的转换。目前 OpenCORE 已经能够支持全部的主流音、视频格式。音频格式有 AAC、AMR、MP3、WAV 等，视频格式有 3GP、MP4、JPG 等。

为了更好地在不同操作系统提供可移植性。OSCL 包含了基本数据类型、配置、字符串工具、输入/输出、错误处理、线程等内容，类似一个基础的 C++库。

相对其他模块而言，OpenCORE 的代码量非常庞大，OpenCORE 基于 C++实现，定义了全功能的操作系统移植层，各种基本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。

4.3.1 内容策略管理

内容策略管理（CPM，Content Policy Manager）作为公共框架的子系统，一般用于确立控制和实施内容的若干规则。最常见的策略管理即数字版权管理（DRM，Digital Rights Management），由于 DRM 规则的多样性，在 CPM 中，并不侧重于具体算法或协议的直接实现，而是提供了一个灵活的框架，为实现这些算法或协议的插件提供开发接口和注册接口。利用 CPM 提供的鉴权、授权、接入接口，CPM 提供了接入被 CPM 保护内容的方法。

除了 DRM 以外，CPM 还支持其他策略，如根据等级或者用户配置限制内容的接入等。在 Android 中，目前提供了对 DRM 1.0 和 DRM 2.0 的原生实现，但仅对 DR,M 1.0 提供了 Java 层的接口，而且 DRM 2.0 的代码在 Android 2.3 中被移除，DRM 的代码主要集中在 frameworks/base/drm 和 frameworks/base/media，在本书中主要介绍 DRM 的内容。

DRM 是保护音频、视频、文档等数字内容版权的一种加密技术。利用 DRM 保护数字内容版权，首先要建立数字内容授权中心，编码压缩后的数字内容，同时利用密钥的公钥对数字内容进行加密保护，加密的数字内容头部存有密钥 ID 和数字内容授权中心的 URL。用户浏览或者点播时，根据数字内容头部的密钥 ID 和 URL 信息，通过数字内容授权中心验证授权后送出的相关私钥进行解密，用户就可以浏览或播放了。

DRM1.0 的制定工作开始于 2002 年。2004 年 6 月，OMA 正式发布了 DRM 1.0 版本，除了 OMA DRM 外，业界比较主流的 DRM 架构还有 Windows Media DRM。但 DRM 架构基本相同。DRM 1.0 没有涉及很强的保护，主要制定了 4 种分发方式：转发锁定（Forward Lock）、组合分发（Combined Delivery）、分组分发（Separate Delivery）、超级分发（Super distribution）。在具体的实现上，分发类型定义位于 frameworks/base/media/java/android/drm/mobile1/ DrmRawContent.java 中。包括四种：DRM_FORWARD_LOCK、DRM_COMBINED_
DELIVERY、DRM_SEPARATE_DELIVERY、DRM_SEPARATE_DELIVERY_DM。其中 DRM_SEPARATE_DELIVERY_DM 是指在 DRM 消息中分组分发。

DRM 的 MIME 类型在 Android 中，主要包括四种：DRM_MIMETYPE_RIGHTS_XML（application/vnd.oma.drm.rights+xml）、DRM_MIMETYPE_RIGHTS_WBXML（application/vnd.
oma.drm.rights+wbxml）、DRM_MIMETYPE_MESSAGE（application/vnd.oma.drm.message）。

在 Android 中，在 DrmStore.java 中对转发锁定提供了一个内容提供器，其受"android.permission.ACCESS_DRM"权限保护。

每打开一个 DRM 对象，就将创建一个会话（Session），原生层的接口位于 frameworks/base/media/libdrm/mobile1/include/objmng/Svc_drm.h 中，在 Java 层中，对版权对象和 DRM 消息做了封装，主要 Java 类的作用如下：

DrmConstraintInfo 类提供了 DRM 约束的接口，如开始日期、结束日期、使用次数等。

DrmRights.java 提供了接入 DRM 版权对象的接口。

DrmRightsManager 类提供了接入 DRM 版权管理器的接口。允许安装、查询、删除版权对象。

DrmRawContent 类提供了接入 DRM 原始内容的接口。能够获取分发的类型和内容的类型等。

而 DrmInputStream 类则提供了经解密后的媒体对象内容的接口。

在 DRM 服务器方面，除了商业解决方案外，开源的 openIPMP 也是个不错的选择，其基于 JBoss 应用服务器和 Mysql 数据库，能够在 Windows 和 Linux 下实现。遗憾的是 openIPMP 在 2006 年后已经不再有人维护。但考虑到 DRM 的重要性，在本节中将稍加介绍。

另外，在 Android 3.0 中，Google 还引入了新的 DRM 框架。

在转发锁定方式中，移动终端禁止转发 DRM 消息（DRM 消息是将媒体对象打包后生成的文件，但未加密，明文存储），但必须支持 DRM 消息文件格式解析。如果移动终端接收到一个包含版权对象（Drm Rights）的 DRM 消息（在组合分发方式中，处理的对象是包含版权对象的 DRM 消息），则需要在提示用户后，将该 DRM 消息抛弃。移动终端可以播放媒体对象，但不能对其修改。

支持组合分发方式的移动终端必须支持转发锁定方式。在该方式中，移动终端根据版权对象来播放媒体对象，版权对象和媒体对象被封装在同一个 DRM 消息中，在拆包后，允许用户丢弃媒体对象，但必须永久保存版权对象。移动终端不得将组合分发方式中的媒体对象转发。

支持分组分发方式的移动终端必须支持组合分发和转发锁定。分组分发方式将媒体对象打包成 DCF（DRM Content Format）格式，使用对称密钥加密，DCF 文件通过 OMA 下载方式下载到移动终端上，版权对象则通过其他方式（如 WAP 推送等）发送。在分组发送中，允许 DCF 文件转发，但版权对象不允许转发。接收到 DCF 文件的其他终端需要从版权引发器（Right Issuer）上获取版权对象。如图 4-4 所示为通过短信推送的方式，将版权对象发送给移动终端。

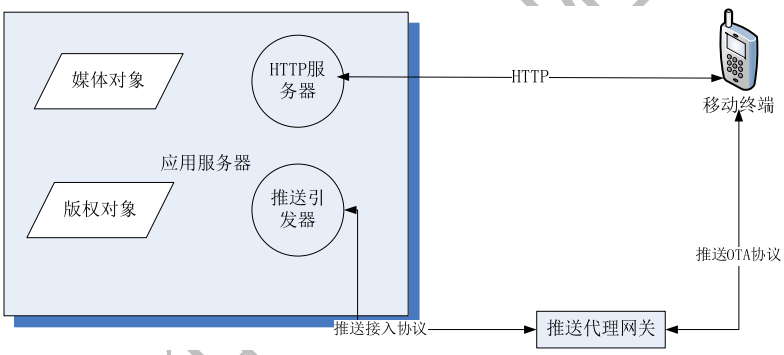

图 4-4　OMA DRM 1.0 分组分发

根据分组分发，OMA DRM1.0 还提出了超级分发的概念，允许在多个移动终端之间传递 DCF 文件，但不能传递版权对象。当未包含版权对象的移动终端接收到 DCF 文件后，会根据文件中的定义，访问对应的版权对象服务器，提示用户购买相应的版权对象并下载。图 4-5 显示了 OMA DRM 1.0 进行超级分发过程。

2005 年 6 月 14 日，OMA 发布了最新的 OMA DRM V2.0，制定了基于 PKI 的安全信任模型，给出了移动 DRM 的功能体系结构、权利描述语言标准、DRM 数字内容格式（DCF）和权利获取协议（ROAP）等。OMA DRM V2.0 包括终端 DRM 代理、内容中心（Content Issuer）、授权中心（Rights Issuer）、用户和移动存储设备等外置存储设备（Off-device Storage）。

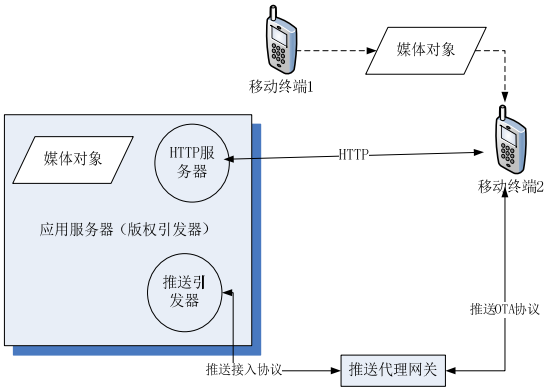

图 4-5　OMA DRM 1.0 超级分发

在 OMA DRM V2.0 中，用户能够通过超级分发等各种方式获得受保护的数字内容，数字内容使用权利需要通过 ROAP 协议获取，使用权利与一个或者一组 DRM 代理绑定，数字内容的使用受到严格的控制。

但是由于 DRM 的局限性，存在着操作麻烦和保护漏洞等问题，DRM 日益受到业界和消费者的质疑，全球四大唱片公司的百代唱片（EMI，The Electric and Musical Industries Ltd）、维旺迪环球唱片公司（UMG，Universal Music Group）、华纳音乐集团（WMG，Warner Music Group）、索尼贝塔斯曼（SONY&BMG Music Entertainment）先后在 2007 年初到 2008 年初宣布开始提供不带数字版权保护的音乐唱片。Apple 公司在 2009 年早些时候就宣布，所有通过 iTunes 商店售出的音乐都将无 DRM 限制，也许在不久的将来 DRM 将会在数字出版领域消亡。

1．DRM 管理

在 CPM 中，按照 DRM 加密策略的不同，目前支持两种类型的 DRM：封装式 DRM（Wrapped DRM）和嵌入式 DRM（Embedded DRM）。 其中封装式 DRM 适用于 OMA DRM 1.0，嵌入式 DRM 适用于 OMA DRM 2.0。

在封装式 DRM 中，媒体对象的所有数据被封装在一个统一的加密层中，这有助于 DRM 在更高层次（文件解析器）上去处理数据。但缺点也很明显，封装式 DRM 不适用于流媒体对象，对媒体对象的所有数据进行加密层封装是不现实的。

而嵌入式 DRM 适应了流媒体数据的特点，能够支持 OMA DRM 2.0 规范。OMA DRM 2.0 提供的保护机制包括本地回放、累进下载、流播放等。

OMA DRM 2.0 基于 ISO BaseMedia 文件格式来保护媒体内容，该文件格式定义了一个本身为明文的容器来存储 DRM 信息和相应的加密数据。这使上层应用可以不依赖 DRM 代理就能解析该文件的部分内容，但如果期望解密数据则必须有文件格式解析器的参与。

封装式 DRM 和嵌入式 DRM 在 DRM 代理和上层应用间的交互过程有所不同。在封装式 DRM 中，上层应用不需要从 DRM 容器文件中为 DRM 代理提取 DRM 信息，封装式 DRM 的控制交互过程如图 4-6 所示。

图 4-6　封装式 DRM 的控制交互过程

嵌入式 DRM 需要从 DRM 容器文件中为 DRM 代理提取 DRM 信息，嵌入式 DRM 的控制交互过程如图 4-7 所示。

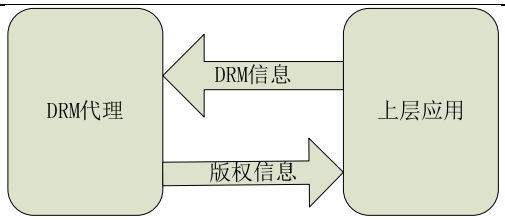

图 4-7　嵌入式 DRM 的控制交互过程

在封装式 DRM 中当对媒体对象的回放所涉及的内容进行读取或者查找时，媒体内容被封装在一个统一的加密层上，明文数据的接入依赖常集成于底层 I/O 文件接口的 DRM 代理，但数据对上层是透明的。封装式 DRM 的数据交互过程如图 4-8 所示。

图 4-8　封装式 DRM 的数据交互过程

对于嵌入式 DRM，文件被部分加密，这就要求上层应用必须自己了解文件格式，并必须直接和 DRM 代理通信以获得明文内容。嵌入式 DRM 的数据交互过程如图 4-9 所示。

图 4-9　嵌入式 DRM 的数据交互过程

为了完成媒体内容的接入，在 CPM 中，大致需要经过如下过程：请求 DRM 资源、打开会话、注册内容句柄、获取内容类型、执行操作（播放、暂停等）、执行内容接入、操作结束、关闭会话、释放 DRM 资源。

如图 4-10 所示为播放器引擎和 CPM 基于封装式 DRM 的 MP4 操作的交互过程。在播放器引擎收到 NodeCommandCompleted()消息后，MP4 文件开始执行播放等操作，当播放停止或者结束时，CPM 会收到上层发来的 UsageComplete()消息。

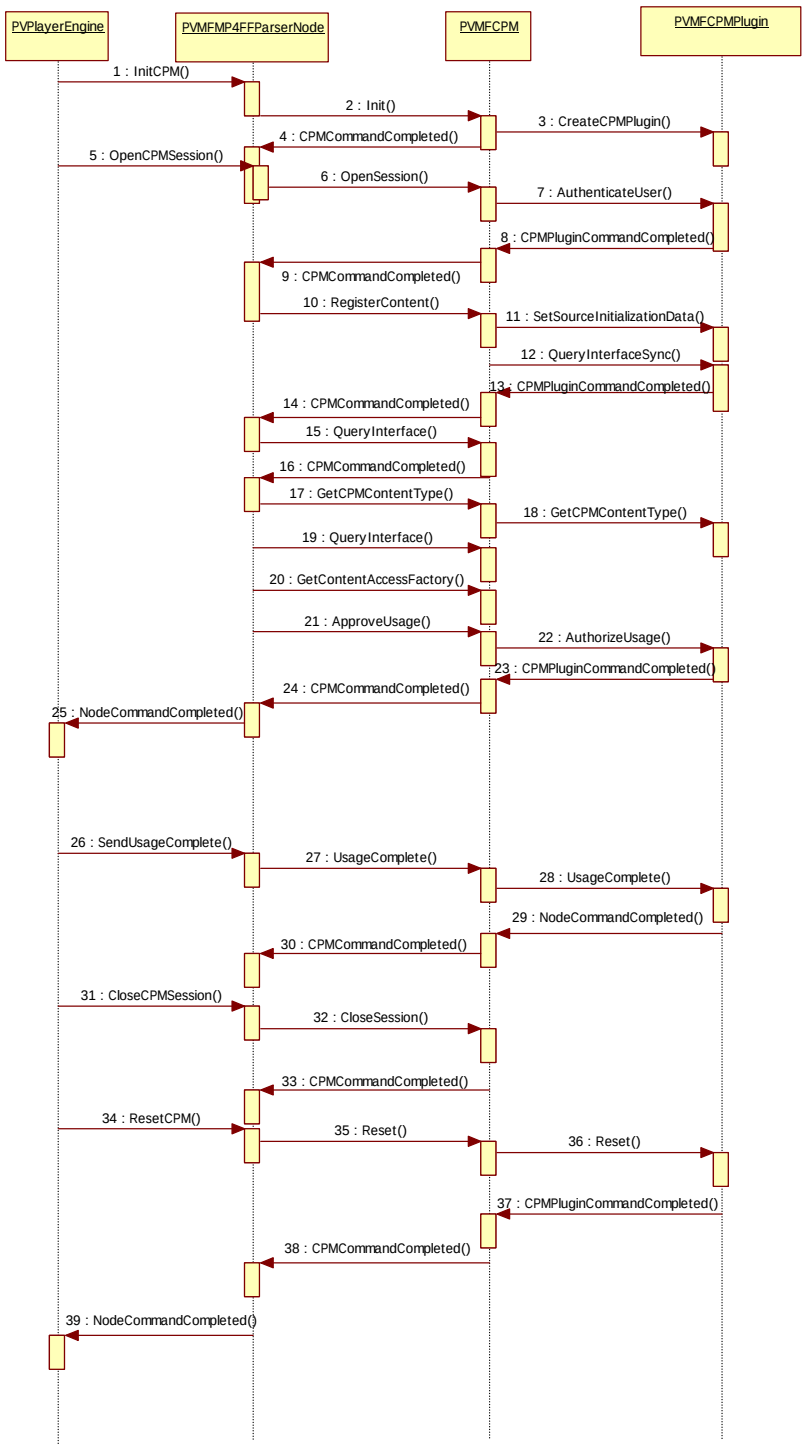

图 4-10　封装式 DRM 的操作交互过程

对于嵌入式 DRM 而言，PVMFCPM::ApproveUsage()可能会被反复调用，用于 DRM 授权。

2．CPM 插件机制

内容策略管理器的作用就是整合一系列与内容管理相关的服务，并提供统一的接口供上层应用调用，这些服务如 DRM 内容接入、起始控制（Parental Control）等均是作为 CPM 插件出现的。根据多媒体的特性，CPM 为 CPM 插件设计了一系列的规则，这些规则主要分为 3 类：

用户鉴权。

操作授权（对播放、暂停、快进、快退、复制、保存等进行授权）。

内容接入（提供打开、阅读、查找等媒体对象的操作）。

在实现完 CPM 插件后，为了使 CPM 插件能够在 OpenCORE 框架内可用，必须首先完成 CPM 插件在 OSCL 组件工厂（OsclComponentFactory）的注册。

在进行 CPM 插件注册时，其 MIME 类型标识符必须以“X-CPM-PLUGIN”作为起始字符，CPM 插件的注册过程如下：

代码 4-5　RegisterPlugin

```
    OSCL_EXPORT_REF   PVMFStatus   PVMFCPMPluginFactoryRegistryClient::RegisterPlugin   (OSCL_String&
aMimeType,
           PVMFCPMPluginFactory& aFactory){
       if (!iClient)
           return PVMFErrInvalidState;
    //确保它是一个有效的 CPM-plugin 的 MIME 字符串
       OSCL_HeapString<OsclMemAllocator> cpmregid(PVMF_MIME_CPM_PLUGIN);
       if (aMimeType.get_size()>=cpmregid.get_size()
             && oscl_CIstrncmp(cpmregid.get_cstr(), aMimeType.get_cstr(), cpmregid.get_size())==0
              && (aMimeType.get_cstr()[cpmregid.get_size()] ==’/’
                  || aMimeType.get_cstr()[cpmregid.get_size()] ==’\0’)){
           switch (iClient->Register(aMimeType, (OsclComponentFactory)&aFactory)){
              case OsclErrNone:
                 return PVMFSuccess;
              case OsclErrAlreadyExists:
                 return PVMFErrAlreadyExists;
              case OsclErrNoMemory:
                 return PVMFErrNoMemory;
              default:
                 return PVMFFailure;
           }
       } else{
          return PVMFErrArgument;
       }
    }
```

在 CPM 中，内容是无法直接接入的，必须通过相应的接口来进行，如为了获得数据流的内容接入，必须获得一个相应的 PVMIDataStreamSyncInterface 接口。为了获得内容的描述符信息，必须获得一个相应的 PVMFCPMPluginAccessUnitDecryptionInterface 接口等。图 4-11 显示了内容的接入过程。

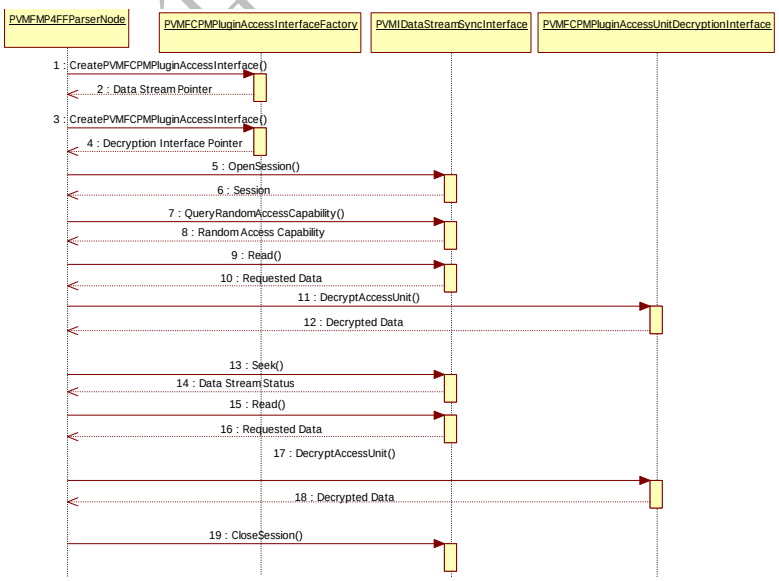

图 4-11　CPM 内容的接入过程

在 OpenCORE 中，给出了一个 OMA DRM 1.0 规格的 CPM 插件的参考实现。目录为 external\opencore\pvmi\content_policy_manager\plugins\oma1。

在目前的实现中，CPM 支持的解析器节点包括 PVMFAMRFFParserNode、PVMFAACFFParserNode、

PVMFMP3FFParserNode 、PVMFWAVFFParserNode 、PVMFMP4 FFParserNode 、PVMFDownloadManagerNode 等。图 4-12 显示了 CPM 的主要类图。

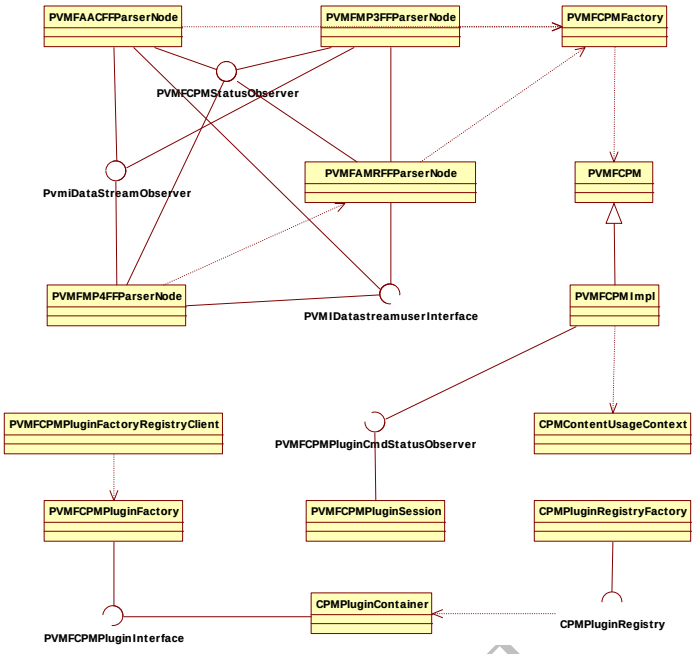

图 4-12　CPM 主要类图

3．OpenIPMP 服务器搭建

OpenIPMP 服务器基于 Java 实现，能够支持 Windows 和 Linux 平台，其和 DRM 客户端的通信机制基于消息系统。在 OpenIPMP 服务器上，事实上定义了两种消息系统：OpenIPMP 消息系统和 OMA 消息系统，对于通用的 DRM 客户端而言，基于 OMA 消息系统和 OpenIPMP 服务器进行通信是个现实的选择。

需要注意的是，基于 OMA 消息系统尚无法基于 Web 进行，目前提供的通信方式为 Web 服务。

由于 OpenIPMP 在 2006 年后已经多年没有更新，和当前的操作系统和编译器会存在着兼容性问题，如果开发者搭建的是单一的 DRM 服务器，建议考虑较老版本的操作系统以减少移植的工作量。而通常情况下，DRM 服务器需要和其他服务器协调工作，在本书中，将基于的操作系统为 Ubuntu 10.04 LTS 进行简要介绍。

OpenIPMP 服务器的搭建需要 My SQL Server 5.1、JBoss 4.2.3 GA 和 Open JDK 6（推荐的 JDK 1.4 事实上存在兼容性问题）等开发工具。下面开始介绍基于 localhost 的 OpenIPMP 服务器的搭建过程，如果希望更改 OpenIPMP 的配置，修改 OMADRMWS/server_config.xml 和 osms/serverConfigData.xml 配置文件即可。

通过 Ubuntu 10.04 LTS 的新立德软件管理器即可顺利安装 My SQL Server 5.1 和 Open JDK 6，然后从 http://sourceforge.net/projects/openipmp/上下载 openipmp_v202.zip，从 JBoss 官网上下载 JBoss 4.2.3 GA。将 OpenIPMP 和 JBoss 解压到/usr 目录下。

然后在/etc/environment 中添加 JBOSS_HOME 环境变量设置为/usr/jboss-4.2.3.GA。接着开始 OpenIPMP 服务器的编译：

```
#cd /usr/openipmp2/src/server
#chmod a+x install.sh
#./install.sh
```

在编译过程中，会出现一些简单的语法错误，根据日志提示，修改错误，即可顺利完成服务器的编译。在服务器编译完成后，OpenIPMP 会根据系统配置的 JBOSS_HOME 环境变量，将 OpenIPMP 的输出文件安装到 JBoss 的/usr/jboss-4.2.3.GA /server/default 目录下。接下来需要启动 JBoss 服务器：

```
#cd /usr/jboss-4.2.3.GA/bin
#chmod a+x run.sh
```

```
#./run.sh
```

完成 JBoss 服务器的启动后，即可在浏览器中登录 OpenIPMP 服务器了，地址为 http://localhost:8080/openipmp/jsp/login.jsp。上述工作一切正确的话，在浏览器中会出现的界面如图 4-13 所示。

图 4-13 OpenIPMP 的注册界面

接着要开始的是进行用户的注册过程。在进行注册前，需要将 OpenIPMP 中的 server.p12 文件拷贝一份到/usr/jboss-4.2.3.GA /conf 中。否则无法完成注册过程。在注册完成后，OpenIPMP 服务器会为相应的用户名如 miaozl 生成一份 P12 文件如 miaozl.p12。需注意保存哟。

为了对不同的多媒体格式进行 DRM 保护，需要为 OpenIPMP 添加相应的格式插件。在 OpenIPMP 中默认携带了 MPEG2 和 MPEG4 的插件，下面以 MPEG4 插件为例介绍编译过程：

首先进入/usr/openipmp2/src/Demo/mpeg4ip 下找到 mpeg4ip-1.5.rar 文件并解压。然后开始编译：

```
    # cd /usr/openipmp2/src/Demo/mpeg4ip/mpeg4ip-1.5/SDL
    #chmod a+x configure
    #./configure
    #make
    #make install
    #cd ..
    #chmod a+x cvs_bootstrap
    #./ mpeg4ip-1.5 –disable-mp4live
#make
#make install
```

当然由于编译器已经是 gcc 4.4 的缘故，在整个编译过程会遇到非常多的语法错误问题，需要开发者耐心的一步步修改。这是件十分头疼的事。

在完成编译后，即可生成 mp4creator 和 mp4player 两个工具，mp4creator 用来问多媒体文件增加保护，mp4player 用来播放经 DRM 加密后的文件。MPEG2 的插件的生成也有类似的编译过程。

4.3.2 多媒体引擎

在 OpenCORE 中，媒体引擎从编解码的类型上可以分为 3 种：播放引擎（PVPlayer Engine）、记录引擎（PVAuthor Engine）、双向引擎（PV2way Engine）。另外适配器引擎（Adapter Engine）和元数据引擎（PV Metadata Engine）也是不可缺少的一部分，在本书中将着重介绍播放引擎、记录引擎和双向引擎。

数据源（Data Source）和数据槽（Data Sink）等通常被抽象成媒体 I/O（Media I/O），或被直接封装在 PVMF 节点中。如果直接封装在 PVMF 节点中，媒体设备可以直接和多媒体引擎进行通信，多媒体引擎和媒体设备间的代码实现也最少，但这要求创建一个新的 PVMF 节点。如果将媒体设备抽象成媒体 I/O 设备，多媒体引擎可以通过与该媒体 I/O 接口通信的方式操作媒体设备，这有利于减小实现的复杂性，但会导致代码规模和层次的增加。

在目前的实现中，数据源类包括 PvmfMediaInputNode、PVMFDummyFileInputNode、PVMFFileDataSource 等；数据槽类包括 PVMFDummyFileOutputNode、PVMFFileOutputNode、PVMFFileDataSink 等。

1．PVPlayerInterface 播放引擎

在 OpenCORE 中，目前支持的播放媒体格式包括 MP4、3GPP、RSTP 流会话和 SMIL 等。播放引擎依赖于 OSCL、PVMF、OMX IL 等组件。

图 4-14 中描述了播放引擎的主要接口类图，客户端通过 PVPlayerInterface 获得播放引擎进行相关的操作，而播放引擎则利用 3 个观察器 PVMFNodeCmdStatusObserver、PVMFNodeInfoEventObserver、PVMFNodeErrorEventObserver 来为客户端提供异步的命令完成状况和异常信息等消息。

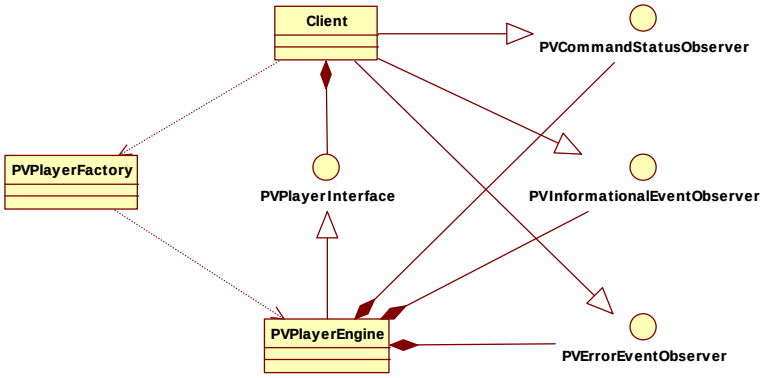

图 4-14　播放接口类图

需要说明的是，在一般情况下，在上层应用和播放引擎之间不需要适配器，但如果 PVPlayerInterface 提供的接口无法满足上层应用的需要，就需要在 OSCL 和上层应用之间构建一个基于新 OSCL 接口的适配层。在为特定平台或者操作系统做 OpenCORE 移植时，这种设计策略常常需要。

播放引擎的实现主要位于 external\opencore\engines\player 目录中，图 4-15 显示的是播放引擎的类图。

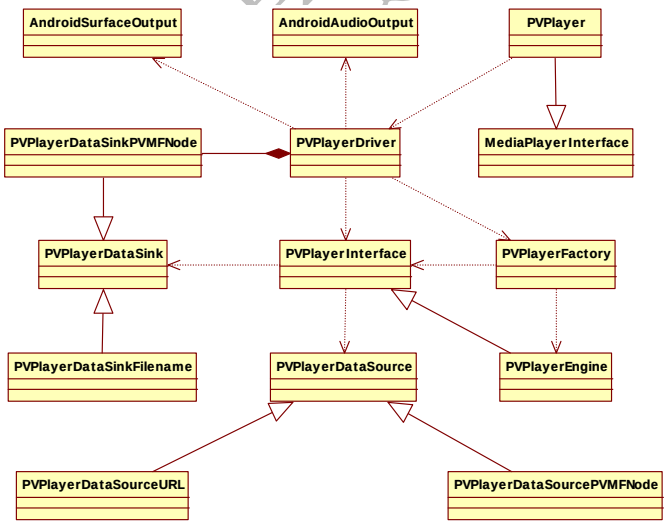

图 4-15　播放引擎主要类图

播放引擎具有较复杂的运行环境，为了便于播放引擎的运行，播放引擎引入了状态机的设计框架。在 OpenCORE 中，播放引擎的状态切换是基于 PVPlayerInterface 接口的，相互的状态触发事件由下层的 PVMF 组件发出。相关的状态在 pv_player_types.h 文件中定义。

在完成引擎的初始化后，播放引擎将处于 PVP_STATE_IDLE 状态，在该状态下，通过 PVPlayerEngine::AddDataSource() 方法可以将数据源添加到引擎中，在添加数据源成功后，通过 PVPlayerEngine::Init() 方法发送 PVP_ENGINE_COMMAND_INIT 命令，引擎将会转换为 PVP_STATE_INITIALIZED 状态。

在 PVP_STATE_INITIALIZED 状态下，客户端可以查询数据源的各种信息，如媒体轨迹、元数据等。

同时通过 PVPlayerEngine::AddDataSink()方法可以添加播放引擎的数据槽。在完成添加数据槽的操作后，通过 PVPlayerEngine::Prepare()方法发送 PVP_ENGINE_ COMMAND_PREPARE 命令，引擎将会转换为 PVP_ENGINE_STATE_PREPARING 状态。

在 PVP_ENGINE_STATE_PREPARING 状态，数据源将会为接下来的媒体播放进行一个排序方面的准备，然后通过 PVPlayerEngine::Start()方法发送 PVP_ENGINE_ COMMAND_START 命令，引擎将会转换为 PVP_STATE_STARTED 状态，开始播放媒体。

在 PVP_STATE_STARTED 状态，用户可以执行暂停、停止等操作，如果执行了 PVPlayerEngine::Stop()方法操作，引擎将会转换为 PVP_STATE_INITIALIZED 状态；如果执行 PVPlayerEngine::Pause()方法操作，引擎将会转换为 PVP_STATE_PAUSED 状态。

在 PVP_STATE_PAUSED 状态，用户可以执行恢复、停止等操作，如果执行了 PVPlayerEngine::Stop()方法操作，引擎将会转换为 PVP_STATE_INITIALIZED 状态；如果执行 PVPlayerEngine:: Resume()方法操作，引擎将会转换为 PVP_STATE_STARTED 状态。

在执行操作的过程中，如果发生了错误或者异常，系统将会转为 PVP_STATE_ERROR 状态，并尝试从错误中恢复。如果发生的是无法恢复的错误，引擎将会清除所有痕迹，转为 PVP_STATE_IDLE 状态。如果错误可以恢复，在出错前状态为 PVP_STATE_INITIALIZED、PVP_STATE_PREPARED、PVP_STATE_STARTED、PVP_STATE_PAUSED 等时，引擎将会转为 PVP_STATE_INITIALIZED 状态，否则引擎转为 PVP_STATE_IDLE 状态。当出错恢复完成时，引擎将会发送相关的异步消息通知。图 4-16 显示了播放引擎的状态图。

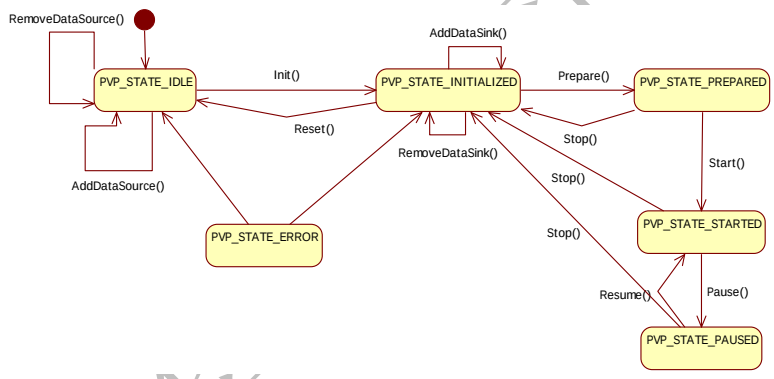

图 4-16　播放引擎状态图

需要说明的是，在默认情况下，播放引擎并不是多线程安全的，为了支持多线程，可以有两种方法，一种方法是利用 OSCL 代理接口组件提供多线程支持，另一种方法是在适配层中为特定平台添加多线程支持。

2．PVAuthorEngineInterface 记录引擎

在记录引擎中，对音频、视频文件均提供了记录支持，其客户端可以分为两种类型：上层应用和适配器。其中适配器客户端作用在于将记录引擎的接口映射为其他框架或者应用所需的特定接口。

记录引擎支持的数据源包括摄像头、麦克风，甚至未经编码的元数据流等。经过记录引擎处理，源数据流会被编码为客户端指定的数据格式。

客户端通过 PVAuthorEngineInterface、PVErrorEventObserver、PVInformationalEvent Observer、PVCommandStatusObserver 等观察器来接受命令、状态信息和错误信息等。记录引擎的实现位于 external\opencore\engines\author 目录下，图 4-17 显示的是记录引擎的类图。

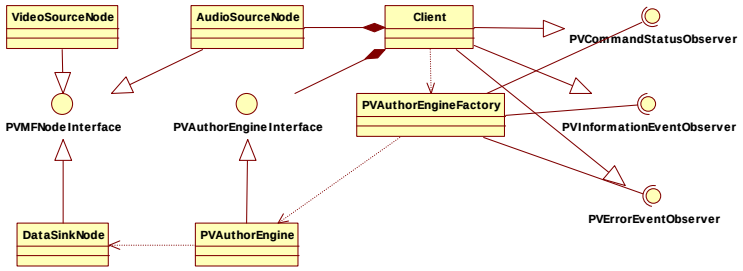

图 4-17　PVAuthor 类继承关系图

在记录引擎中，状态之间的变迁同样是通过状态机控制的，其状态包括 PVAE_STATE_IDLE、PVAE_STATE_OPENED 、 PVAE_STATE_INITIALIZED 、 PVAE_STATE_ RECORDING 、PVAE_STATE_PAUSED、PVAE_STATE_ERROR 等，图 4-18 显示了记录引擎中的状态跃迁。

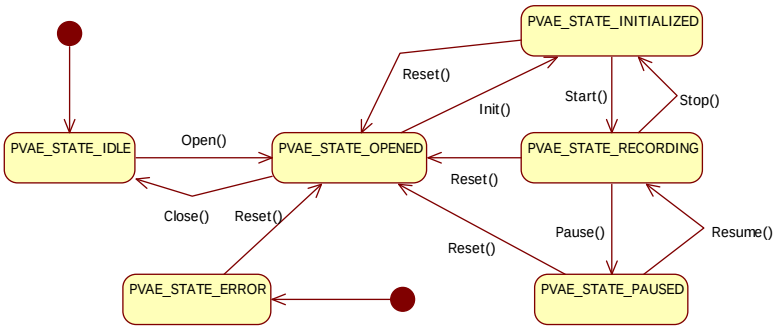

图 4-18　PVAuthor 引擎状态图

当执行多媒体记录时，首先要创建一个记录引擎，然后添加数据源。具体的执行过程如图 4-19 所示。

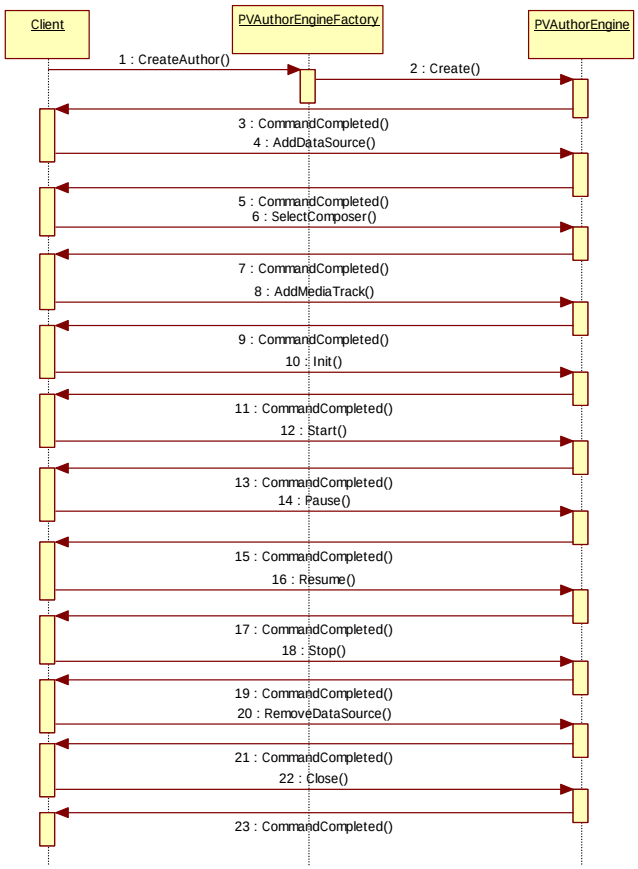

图 4-19　PVAutoor 执行过程图

在添加完数据源后，还需要选择组合器和添加 MediaTrack，才方能开始媒体记录。

3．CPV2WayInterface 双向引擎

随着芯片处理能力和无线带宽的提高，3G 技术得到不断推广，视频电话逐渐走进普通用户的视野。为了支持视频电话业务，在 OpenCORE 中，提供了依赖于平台的多媒体双向引擎，对 H.324M（3G-324M 在 H.324M 基础上指定了 H.263 作为强制基本标准，而把 MPEG-4 作为视频编码推荐标准，AMR 作为音频编码强制标准，主要用于无线网络）、H.323（主要用于有线互联网，无 QoS 保障）、SIP（侧重 NGN 网络）等主流视频电话协议栈都提供了支持。

在 3G 移动终端中，通常采用的是 3G-324M 协议栈，目前拥有 3G-324M 协议栈核心技术的厂商主要有达丽星（Dilithium）、锐迪讯（Radvision）等。

随着无线移动网络进一步朝着 IP 化的方向演进和基于 3G-324M 协议栈控制复杂等缺陷方面的考虑，在 3GPP R5 版本中，正式引入了基于 SIP 协议族的多媒体子系统（IMS, IP Multimedia Subsystem），使基于 SIP 的移动视频通话业务成为可能。

双向引擎的输入数据可以有麦克风、摄像头等，输出数据源有显示屏、扬声器等。数据源和数据槽均由以应用或者适配层的形式存在的客户端添加。

客户端利用工厂类 CPV2WayEngineFactory 或者 CPV2WayProxyFactory 来获得一个继承 CPV2WayInterface 接口的双向引擎的引用（在目前的版本中，双向引擎为 CPV324m2Way）。另外，客户端还必须实现双向引擎的观察者接口（PVCommandStatusObserver、PVInformationalEventObserver、PVErrorEventObserver），以获得收到命令完成情况、状态信息和错误信息的途径，并将这些接口传递给双向引擎。

客户端还应基于 PV2WayMIO 类或 PVMFNodeInterface 接口实现双向引擎所需的数据源和数据槽，数据源和数据槽在实现上都是基于 Oscl_Vector 向量的。客户端会通过 CPV2WayInterface 接口执行视频会话过程中的相关操作，如初始化、添加数据源、添加数据槽、连接、暂停、恢复、断连等。双向引擎的实现位于 external\opencore\engines\2way 目录下，如图 4-20 所示为双向引擎的主要类的继承关系图。

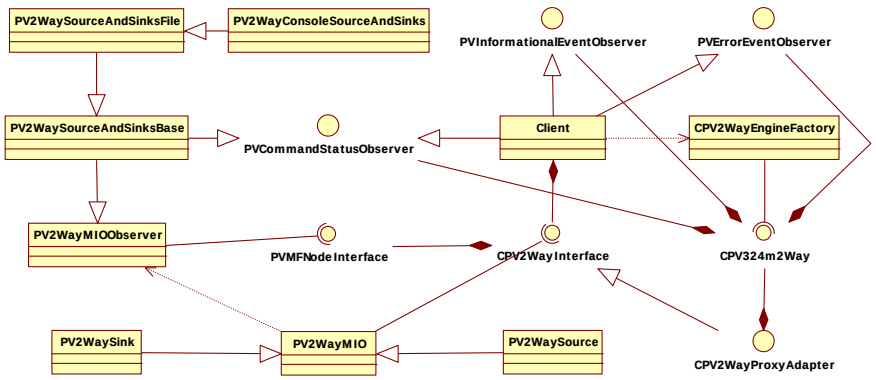

图 4-20　PV2Way 类继承关系图

在目前的设计中，OpenCORE 的双向引擎内置了 7 种状态：空闲（EIdle）、初始化（EInitializing）、建立（ESetup）、连接中（EConnecting）、已连接（EConnected）、断连中（EDisconnecting）、重设（EResetting）。

其中“EIdle”状态模式为双向引擎对象的初创状态，尚没有任何资源被分配。

在“EInitializing”状态下，引擎尝试获得可用的设备资源（如编解码、内存等），准备接受建立参数和视频连接。如果成功，则引擎状态转换为“ESetup”状态，如果失败，则释放已请求成功的资源，转换为“EIdle”状态。

在收到编码、复用、捕获能力、渲染能力等建立参数的过程中，引擎处于“ESetup”状态，在引擎状态转换为“EConnecting”状态之前，有效的数据源和数据槽应被添加到双向引擎中。

当引擎对象收到一个请求连接的呼叫时，引擎转换为“ESetup”状态。在该模式下，移动终端将和远

程终端交换媒体支持能力和信道配置信息，为接下来的媒体信道连接做好准备。

在协议栈的控制面信令完成通信后，终端开始尝试基于通信双方的支持能力建立音频、视频轨迹，数据源和数据槽将在这一过程中转入运行，引擎转换为“EConnecting”状态。

当因通信完成或者资源不足等原因，引擎决定断开信道和复用设备时，引擎的状态转换为“EDisconnecting”状态。

如果终端开始释放所有双向引擎占用的通信资源，则引擎转为“EResetting”状态。

图 4-21 显示了双向引擎的状态跃迁。

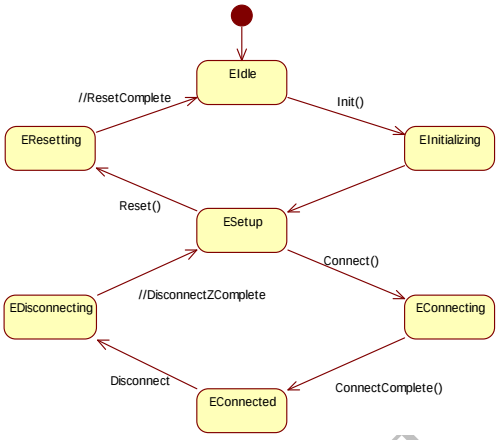

图 4-21　PV2Way 状态图

在 Android 中，目前支持的基于 CPV2WayInterface 接口的双向引擎为 CPV324m2Way。数据源和数据槽均是基于 PV2WayMIO 类派生的。

为了进行视频通话，首先需要创建一个双向引擎，然后添加数据源和数据槽，具体的交互过程如图 4-22 所示。

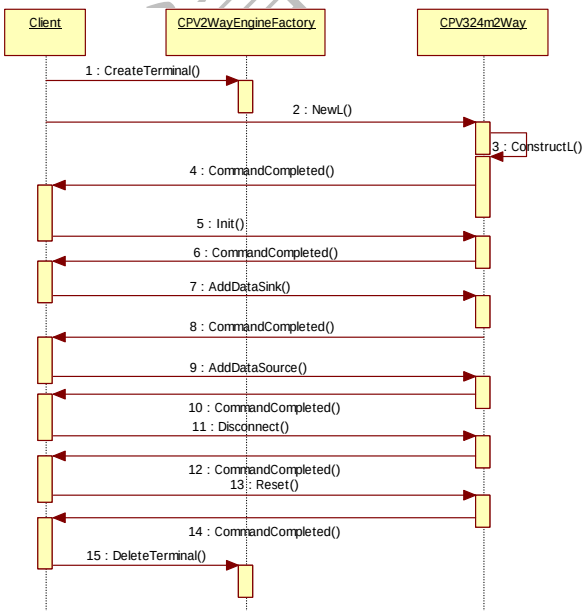

图 4-22　PV2Way 交互过程图

OpenCORE 所能支持的媒体格式参看 pvmf_format_type.h。客户端和服务器进行通信协商编码的过程如下：

代码 4-6　FormatInList 的实现

```
CodecSpecifier* PV2WayMIO::FormatInList(PVMFFormatType& type)
{
    Oscl_Map<PVMFFormatType, CodecSpecifier*,
```

```
        OsclMemAllocator, pvmf_format_type_key_compare_class >::iterator it =
            iFormatsMap.begin();
        it=iFormatsMap.find(type);      //查找匹配格式
        if (!(it==iFormatsMap.end()))
            return (*it).second;
        return NULL;
    }
```

4.3.3 文件解析和组合

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。一种是解析器（Parser），用于解析文件；另一种是组合器（Composer），用于记录文件。

在媒体文件的播放过程中，一个视频文件会包含音频流和视频流的组合，如在一个 MP4 文件中，可能包含了 AMR 或 AAC 的音频压缩流，以及 H263、MPEG 4 或者 H264 AVC 的视频压缩流。这些流文件被封装在一个包文件中，媒体播放器首先要做的就是将这些流文件从包文件中解析出来，然后根据文件类型的不同调用不同的解码器进行解码，最后将解码后的元数据流输送到相应的硬件设备进行播放。

在媒体文件的记录过程中，涉及视频、音频、图像的捕获功能。如在录像过程中，首先需要从硬件设备分别获得视频元数据流和音频元数据流，然后根据相关的配置信息，将元数据流压缩成相应格式的压缩流，并写入文件中，最终组合成相应的包文件。

需要说明的是，目前中、高端的移动终端上都配备了专门用于多媒体加速的 DSP，相关的音频、视频编解码工作都会在专用 DSP 上进行。OpenCORE 关于文件格式解析的内容位于 external\opencore\fileformats 和 external\opencore\protocols\sdp\parser 目录下。

下面分别以 MP3 和 MPEG-4 为例来分析解析器、组合器的实现。

1．解析器

在 Android 中，目前内置了 AVI、MPEG-4、WAV、MP3、AMR、AAC 等格式的解析器。下面以 MP3 为例介绍解析器的实现，相关的文件包括 imp3ff.cpp、mp3fileio.cpp、mp3parser.cpp、mp3utils.cpp 等。具体实现位于 external\opencore\fileformats\ mp3\parser 目录下。

在 Android 中，MP3 解析器节点的注册是在 PVMERegistryPopulator::RegisterAllNodes()方法中完成的。

为了进行 MP3 文件的播放，需要将 MP3 的数据源添加到播放引擎中，在添加数据源的过程中会进行文件解析。MP3 播放时进行文件解析的过程如图 4-23 所示。

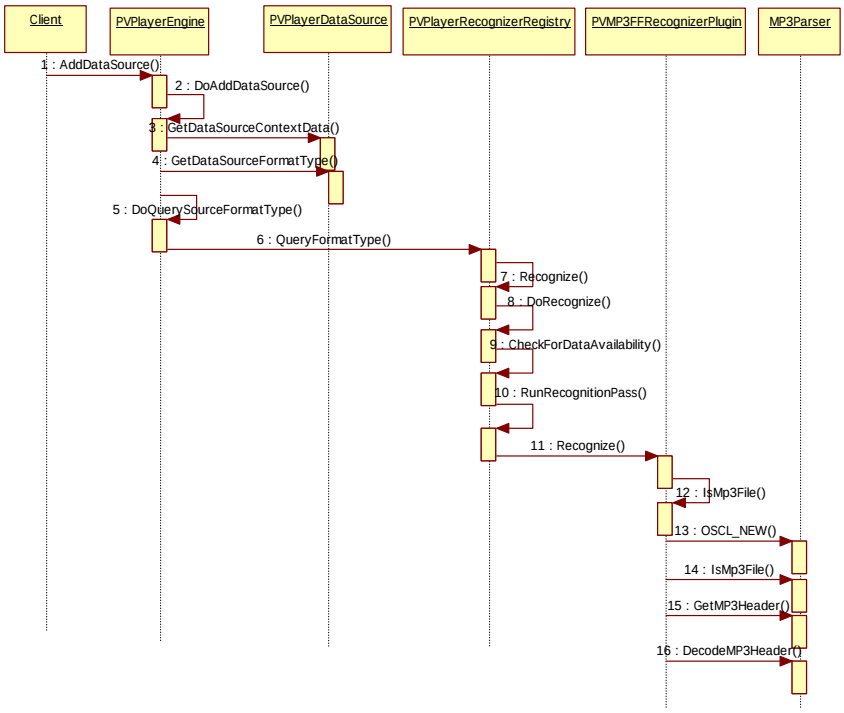

图 4-23　MP3 播放时的文件解析过程

流程说明：

在客户端通过 PlayerDriver 添加数据源的过程中，PVPlayerEngine 引擎会向一个线程安全的事件队列 iThreadSafeQueue 中添加 PVP_ENGINE_COMMAND_ADD_ DATA_SOURCE 命令。当 PVPlayerEngine 引擎收到该命令后，调用 DoAddDataSource()方法进行数据源的添加。

根据传过来的命令 ID，PVPlayerEngine 引擎获得当前的 PVPlayerDataSource 对象，然后调用 PVPlayerDataSource::GetDataSourceFormatType()方法查看当前的文件格式。如果文件格式未知，就需要根据传递的命令 ID 和命令上下文调用 DoQuerySourceFormatType()方法进行文件格式的解析了。

在 DoQuerySourceFormatType()方法中，会经过一系列的操作获得一个 PVMFCPMPluginAccessInterfaceFactory 对象，然后根据该对象调用 PVPlayer RecognizerRegistry:: QueryFormatType()方法打开识别器（Recognizer）的会话并进行识别。

PVMFRecognizerRegistry 对象通过调用 PVMFRecognizerRegistryImpl::Recognize()方法，向自身发送一个 PVMFRECREG_COMMAND_RECOGNIZE 命令，将工作交给 PVMF RecognizerRegistryImpl::DoRecognize()方法进行接下来的处理。

在 PVMFRecognizerRegistryImpl::DoRecognize()方法中，首先会通过 CheckFor DataAvailability()方法验证数据的有效性，然后通过 RunRecognitionPass()方法调用解析器插件。直到这时，PVMP3FFRecognizerPlugin 才有了用武之地。

在 PVMP3FFRecognizerPlugin::Recognize()方法中，会首先通过 OSCL_NEW()方法创建一个 IMpeg3File 对象。然后调用 IMpeg3File::IsMp3File()方法判断该文件是不是 MP3 文件。

在 IMpeg3File::IsMp3File()方法中，会通过 OSCL_NEW()方法创建一个 MP3Parser 对象，然后在 MP3Utils 和 MP3FileIO 的配合下，通过 MP3Parser::GetMP3Header()尝试获得 MP3 的头文件。如果成功，再通过 MP3Parser::DecodeMP3Header()来解析该头文件，通过同样成功，则确认该文件就是 MP3 文件了。

在完成了文件的解析后，即开始解码工作，关于 MP3 的解码工作请参考 6.2.3 节 MP3 的解码过程。

2．组合器

在 Android 中，目前内置了 MPEG-4 等格式的组合器，组合器的实现要比解析器复杂的多。下面以 MPEG-4 为例介绍组合器的实现，相关的文件位于 external\了 opencore\fileformats\mp4\composer 目录下。如图 4-24 所示为 MP4 记录时的文件解析过程。

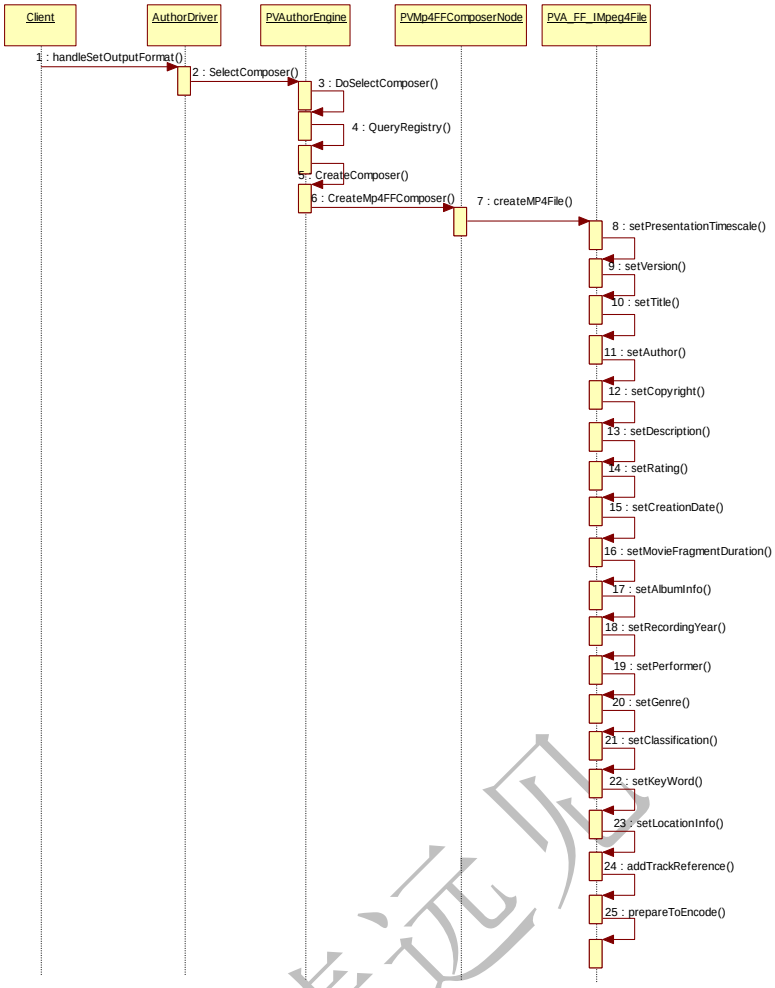

图 4-24　MP4 记录时的文件解析过程

流程说明：

在进行 MP4 记录时，客户端会通过 PVMediaRecorder::setOutputFormat()方法向记录引擎 PVAuthorEngine 发送 AUTHOR_SET_OUTPUT_FORMAT 消息，设置输出文件的格式。

在 AuthorDriver 的监听线程收到 AUTHOR_SET_OUTPUT_FORMAT 消息后，会调用 handleSetOutputFormat()方法进行输出格式 MIME 类型的设置。然后调用 PVAuthorEngine 引擎的 SelectComposer()方法进行组合器的选择。

PVAuthorEngine 引擎首先向自身发送 PVAE_CMD_SELECT_COMPOSER 消息到事件队列中，一旦收到该消息，即调用 DoSelectComposer()方法，根据消息携带的 MIME 类型获得相应的 UUID，然后调用 PVAuthorEngineNodeFactoryUtility::CreateComposer()方法创建组合器。接下来系统根据 UUID 判定是否创建 MP4 文件，如果是，则调用 PVMp4FFComposerNodeFactory::CreateMp4FFComposer()方法创建 MP4 的组合器。

当开始媒体记录时，MP4 组合器会被调用 DoStart()方法，在该方法中，组合器会创建 MP4 文件，设置记录模式、缓冲等，然后设置时间戳、版本号、标题、作者、版权、备注等信息，最后调用 PVA_FF_IMpeg4File::prepareToEncode()方法开始编码工作。

需要说明的是，AuthorDriver 为记录引擎和上层应用的接口，PlayerDriver 为播放引擎与上层应用的接口。

4.3.4　编解码器

所有的 OpenCORE 编解码器均被封装在由 Khronos 定义的开放标准 OpenMAX IL 接口中，上层的

PVMF 框架在与编解码器进行通信时，必须通过 OpenMAX IL 接口进行。这样的设计便于集成第三方的编解码器到 OpenCORE 中，保证了 PVMF 框架的可扩展性和灵活性。

如果希望为 OpenCORE 框架增加一种文件格式的播放支持，则包含文件解析器、解码器等方面的工作。如果希望增加一种文件格式的记录支持，则包含文件组合器、编码器等方面的工作。

1．编码器

就目前而言，主流的视频编码包括 MPEG-4 等，音频编码包括 AMR、AAC 等。在 Android 中，目前提供了 AMR-NB、SBC 等音频编码器的实现和 AVC H264 和 M4V H263 等视频编码器的实现。下面针对 AAC 涉及的 MPEG-2 和 MP4、3GP 等涉及的 MPEG-4 进行简单的背景介绍。

1）MPEG-2

MPEG-2 是 MPEG 工作组于 1994 年发布的一个视频和音频压缩国际标准。MPEG-2 通常用来为广播信号（如卫星电视、有线电视等）提供视频和音频编码，MPEG-2 经少量修改后，成为 DVD 产品的核心编码技术。

MPEG-2 的系统描述部分（第一部分）定义了传输流，它定义了一套在非可靠介质上传输数字视频信号和音频信号的机制，主要用在广播电视领域。

MPEG-2 的视频部分（第二部分）和 MPEG-1 类似，但是它提供了对隔行扫描视频显示模式的支持（隔行扫描广泛应用在广播电视领域）。MPEG-2 视频并没有对低位速率（小于 1Mbps）进行优化，在 3Mb/s 及以上位速率情况下，MPEG-2 明显优于 MPEG-1。另外 MPEG-2 向后兼容，也就是说，所有符合标准的 MPEG-2 解码器也能够正常播放 MPEG-1 视频流。

MPEG-2 技术也应用在了 HDTV 传输系统中。

MPEG-2 的第三部分定义了音频压缩标准。该部分改进了 MPEG-1 的音频压缩，支持两通道以上的音频。MPEG-2 音频压缩部分也保持了向后兼容的特点。

MPEG-2 的第七部分定义了不能向后兼容的音频压缩，该部分提供了更强的音频功能。通常我们所说的 MPEG-2 AAC 指的就是这一部分。

MPEG-2 视频通常包含多个 GOP（GOP=Group Of Pictures），每一个 GOP 包含多个帧。帧的类型有 I-帧、P-帧和 B-帧等。其中 I-帧用于帧内编码，P-帧用于前向估计，B-帧用于双向估计。

一般来说输入视频速率为 25 帧/秒（CCIR 标准）或者 29.97 帧/秒（FCC 标准）。

MPEG-2 支持隔行扫描和逐行扫描。在逐行扫描模式下，编码的基本单元是帧；在隔行扫描模式下，基本编码可以是帧，也可以是场（field）。

原始输入图像首先被转换到 YCbCr 颜色空间。其中 Y 是亮度，Cb 和 Cr 是两个色度通道。对于每一通道，首先采用块分割，然后形成“宏块”（macroblocks），宏块构成了编码的基本单元。每一个宏块再分割成 8×8 的小块。色度通道分割成小块的数目取决于初始参数设置。例如，在常用的 YCbCr 420 格式下，每个色度宏块只采样出一个小块，所以 3 个通道宏块能够分割成的小块数目是 4+1+1=6 个。

对于 I-帧，整幅图像直接进入编码过程。对于 P-帧和 B-帧，首先做运动补偿。通常来说，由于相邻帧之间的相关性很强，宏块可以在前帧和后帧中对应相近的位置找到相似的区域匹配。偏差即偏移量作为运动向量被记录下来，运动估计重构区域的误差被送到编码器中编码。

对于每一个 8×8 小块，需要通过离散余弦变换把图像从空间域转换到频域，将得到的变换系数被量化并重新组织排列顺序，从而增加长零的可能性。然后做游程编码（run-length code），最后进行哈夫曼编码（Huffman Encoding）。

I-帧编码有助于减少空间域冗余，P-帧和 B-帧有助于减少时间域冗余。

GOP 是由固定模式的一系列 I-帧、P-帧、B-帧组成的。常用的结构由 15 个帧组成，具有“IBBPBBPBBPBBPBB” 形式。GOP 中各个帧的比例的选取和带宽、图像的质量要求有一定关系。例如 B-帧的压缩时间几乎是 I-帧的 3 倍，所以对于计算能力不强的某些实时系统，可能需要减少 B-帧的比例。

MPEG-2 输出的比特流可以是匀速也可以是变速的。在 DVD 应用上，最大比特率可达 10.4 Mb/s。如果要使用固定比特率，就需要不断地调节量化尺度以产生匀速的比特流。但是，提高量化尺度可能带来可

视的失真效果，如马赛克现象。

MPEG-2 由 ISO/IEC 13818 定义，其各部分的定义如下。

ISO/IEC 13818-1：系统-描述视频和音频的同步和多路技术。

ISO/IEC 13818-2：视频-视频压缩。

ISO/IEC 13818-3：音频-音频压缩，包括多通道的 MP3 扩展。

ISO/IEC 13818-4：测试规范。

ISO/IEC 13818-5：仿真软件。

ISO/IEC 13818-6：DSM-CC（Digital Storage Media Command and Control）扩展。

ISO/IEC 13818-7：Advanced Audio Coding（AAC）。

ISO/IEC 13818-9：实时接口扩展。

ISO/IEC 13818-10：DSM-CC 规范。

2）MPEG-4

MPEG-4 是一套用于音频、视频信息的压缩编码标准，由 ISO 和 IEC 下属的活动图像专家组（MPEG，Moving Picture Experts Group）负责制定。MPEG-4 V1.0 在 1998 年 10 月通过，MPEG-4 V2.0 在 1999 年 12 月通过。MPEG-4 主要用途在于流媒体、光盘、视频电话、电视广播等。

MPEG-4 包含了 MPEG-1 及 MPEG-2 的绝大部分功能及其他格式的长处，并加入及补充了对虚拟现实模型语言（VRML，Virtual Reality Modeling Language）的支持、面向对象的合成文件（包括音效、视频及 VRML 对象），以及对 DRM 及其他交互功能。

MPEG-4 具有灵活的框架，用户可以根据自身的情况采用某些功能的集合。MPEG-4 由 ISO/IEC 14496 定义，其各部分的定义如下。

ISO/IEC 14496-1，系统部分：描述视讯和音频的同步，以及混合方式。

ISO/IEC 14496-2，视讯部分：定义了一个对各种视觉信息（包括视讯、静态纹理、计算机合成图形等）的编解码器 XviD。

ISO/IEC 14496-3，音频部分：定义了一个对各种音频信号进行编码的编解码器的集合。包括 AAC 的若干变形和其他一些音频、语音编码工具。

ISO/IEC 14496-4，一致性部分：定义了对本标准其他的部分进行一致性测试的程序。

ISO/IEC 14496-5，参考软件部分：提供了用于演示功能和说明本标准其他部分功能的软件。

ISO/IEC 14496-6，多媒体传输集成框架（DMIF，Delivery Multimedia Integration Framework）。

ISO/IEC 14496-7，优化的参考软件：提供了对实现进行优化的例子。

ISO/IEC 14496-8，在 IP 网络上传输：定义了在 IP 网络上传输 MPEG-4 内容的方式。

ISO/IEC 14496-9，参考硬件：提供了用于演示怎样在硬件上实现本标准其他部分功能的硬件设计方案。

ISO/IEC 14496-10，高级视频编码（AVC，Advanced Video Coding）：定义了一个视频编解码器。AVC 和 XviD 都属于 MPEG-4 编码，但由于 AVC 属于 MPEG-4 Part 10，在技术特性上比属于 MPEG-4 Part2 的 XviD 要先进，而从技术上讲，它和 ITU-T H.264 标准是一致的，故全称为“MPEG-4 AVC/H.264”。

ISO/IEC 14496-12，基于 ISO 的媒体文件格式：定义了一个存储媒体内容的文件格式。

ISO/IEC 14496-13，知识产权管理和保护（IPMP，Intellectual Property Management and Protection）拓展。

ISO/IEC 14496-14，MPEG-4 文件格式：定义了基于 ISO/IEC 14496-12 的用于存储 MPEG-4 内容的容器文件格式。

ISO/IEC 14496-15，AVC 文件格式：定义了基于 ISO/IEC 14496-12 的用于存储 ISO/IEC 14496-10 的视频内容的文件格式。

ISO/IEC 14496-16，动画框架扩展（AFX，Animation Framework eXtension）。

ISO/IEC 14496-17，同步文本字幕格式。

ISO/IEC 14496-18，字体压缩和流式传输（针对公开字体格式）。

ISO/IEC 14496-19，综合用材质流（Synthesized Texture Stream）。

ISO/IEC 14496-20，简单场景表示（LASeR，Lightweight Scene Representation）。

ISO/IEC 14496-21，用于渲染的 MPEG-J 拓展。

在 OpenCORE 中，目前已经提供了 AVC、M4V、H.263、AMR、AAC 等格式的编码器。

M4V 是由 Apple 公司开发的一个文件格式，它是基于 MPEG-4 的第二部分压缩的。M4V 与 MP4 的不同在于，M4V 文件中的音频、视频轨道是相互独立的，M4V 文件的音频采用的是 AAC 的编码方式，而视频采用的则是 H.264 的编码方式。

离散余弦变换（DCT, Discrete Cosine Transform）是对语音和图像信号进行变换的优秀方法，其变换特性接近 KLT（Karhunen-Loeve Transform）。1986 年由 Prencen 和 Bradly 提出了一种改进的离散余弦变换[32]（MDCT，Modified Discrete Cosine Transform），利用 50%的样点叠加和时域混叠消除（TDAC）滤波器组，在不降低变换编码性能的情况下，有效地克服了 DCT 块处理运算中的边缘效应[33]（块边缘噪声）。在相同编码效率的情况下，MDCT 的性能优于 DCT。目前在语音、宽带音频及图像信号的变换编码中，都普遍采用 MDCT。

下面是 Android 在设置视频数据源后，调用视频编码器的过程，如图 4-25 所示。

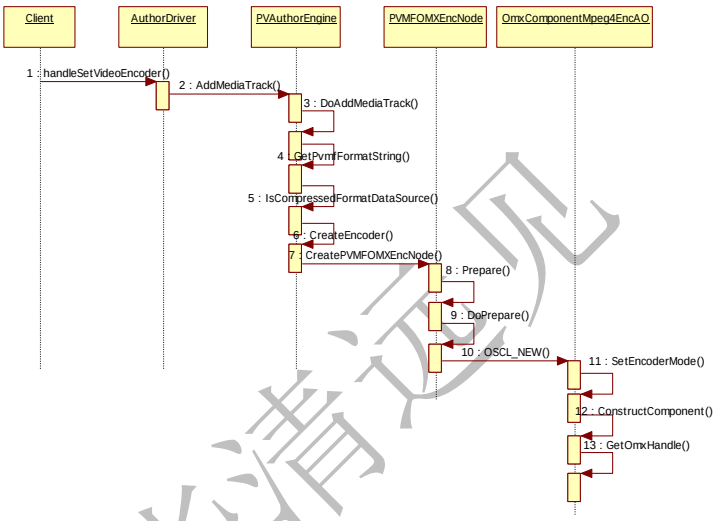

图 4-25　录制 MP4 时调用编码器的过程

流程说明：

当进行录像时，开发者需要通过 PVMediaRecorder::setVideoEncoder()方法向记录引擎发送 AUTHOR_SET_VIDEO_ENCODER 消息来设置视频的解码器。

当记录引擎中的 AuthorDriver 收到 AUTHOR_SET_VIDEO_ENCODER 消息后，通过 handleSetVideoEncoder()方法设置媒体的 MIME 类型，并通过 mVideoInputMIO:: SetFrameRate()方法和 mVideoInputMIO:: SetFrameSize()方法来设置帧速率和帧大小，接着通过调用 PVAuthorEngine::AddMediaTrack()方法开始添加媒体轨迹。

PVAuthorEngine::AddMediaTrack()方法会通过向自身发送 PVAE_CMD_ADD_ MEDIA_TRACK 命令，调用 PVAuthorEngine::DoAddMediaTrack()方法来执行添加媒体轨迹的请求。在这一过程中，首先根据命令携带的 MIME 类型，通过 PVAuthorEngine NodeFactoryUtility::QueryRegistry()方法来查询相应的 UUID，然后调用 PVAuthorEngine NodeFactoryUtility::CreateEncoder()进行编码器节点的创建。

当录制视频格式为 MP4 时，相应的 UUID 为 KPVMFOMXVideoEncNodeUuid，通过 PVMFOMXEncNodeFactory 创建编码节点 PVMFOMXEncNode。

当上层调用 PVMediaRecorder::prepare()方法时，最终会调用到 PVMFOMX EncNode::Prepare()方法来执行准备工作。在执行准备工作的 PVMFOMXEncNode::DoPrepare()方法中，会调用 OpenMAX Core 的 OMX_MasterGetHandle()方法，根据组件名来具体执行编码器的检索和创建，最终创建 Mpeg4EncOmxComponentFactory 对象。

在 Mpeg4EncOmxComponentFactory 对象创建的过程中，会首先通过 OSCL_NEW()方法为一个 OmxComponentMpeg4EncAO 对象分配内存，然后调用 OmxComponentMpeg4EncAO 的 SetEncoderMode()方法和 ConstructComponent()方法构建出具体的编码器对象，并返回给上层相应的句柄。

编码器的具体实现位于 external\opencore\codecs_v2 目录中，具体的编码器实现在这里就不再多做介绍了，音频编码请参考 6.2.1 节 AMR 的编码过程，视频编码请参考 7.3.1 节视频的编码过程。

2．解码器

在 OpenCORE 中，目前已经提供了 AVC、M4V、H.263、WMA、AAC、AMR、MP3、WMA、RV、RA 等格式的解码器。

为了提高渲染速度，在解码器的输出缓冲（Output Buffer）和渲染组件的缓冲之间应尽量避免复制和转换等操作的发生。出于这样的目的，在设计时，解码器的输出缓冲的宽高、位深等约束应与渲染组件的特性保持一致。图 4-26 显示了相关的解码类图。

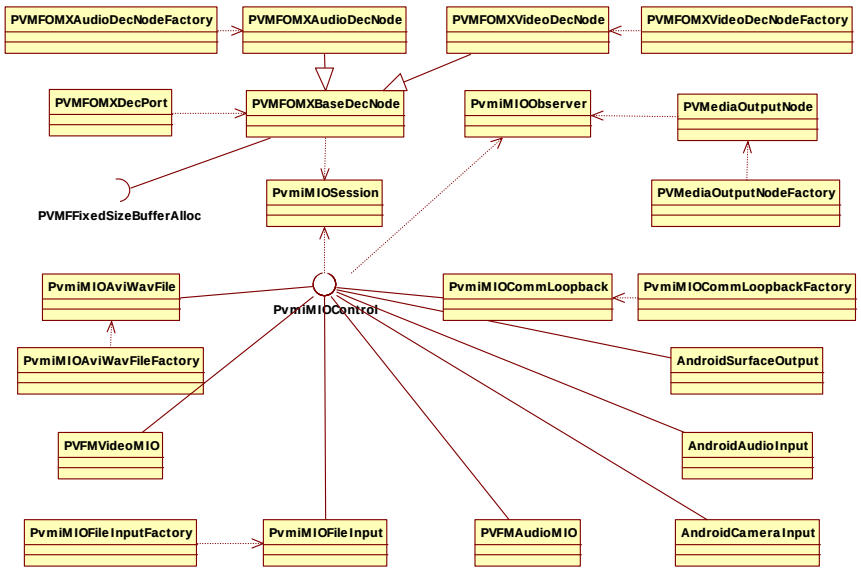

图 4-26　解码相关类图

下面以 MP4 为例进行解码器调用的介绍，需要注意的是，对于视频的播放，需要同时调用两个解码器：音频解码器和视频解码器，分别送到相应的设备进行解码。其中视频解码器的调用过程如图 4-27 所示。音频解码器的调用过程和视频解码器相似，这里就不再介绍了。

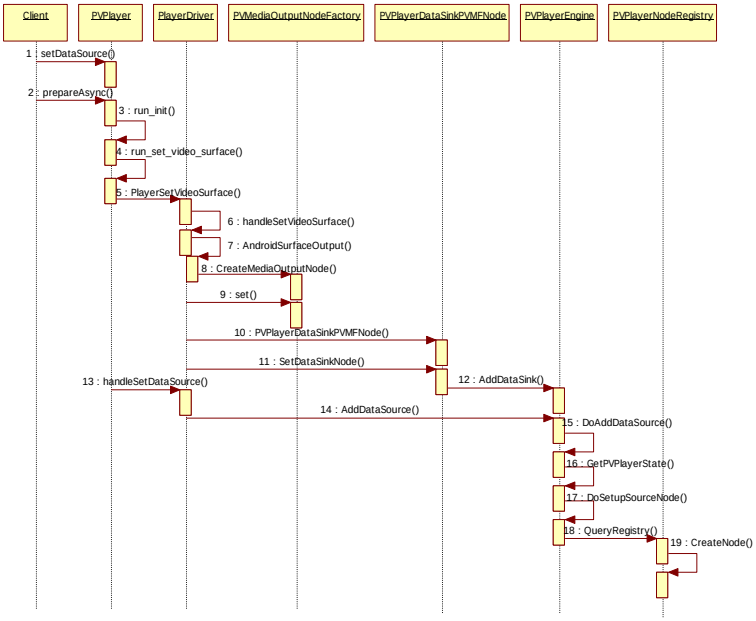

图 4-27　播放 MP4 时调用视频解码器的过程

流程说明：

当欲进行视频的播放时，首先通过 MediaPlayer 调用原生层的媒体播放服务，如果播放类型为 PV_PLAYER，则提供服务的对象为 PVPlayer。

当通过 MediaPlayer 添加数据源时，最终将调用 PVPlayer::setDataSource()方法来执行这一操作，但仅保存数据源的路径。

当调用 MediaPlayer::prepareAsync()方法时，通过 PVPlayer 会创建一个 Player SetDataSource 对象，并向自身发送一个 PLAYER_SET_DATA_SOURCE 命令。当处理完成时，调用回调函数 run_init()。

在 run_init()方法中，会调用 run_set_video_surface()方法进行视频 Surface 的设置。同时通过 run_set_audio_output()方法来设置音频输出。

在 run_set_audio_output()方法中，为了设置音频输出，需要通过向自身发送 PLAYER_SET_AUDIO_SINK 命令，并在 handleSetAudioSink()方法中处理该命令。

在 handleSetAudioSink()方法中，如果对实时性有要求，则创建一个 AndroidAudioOutput 对象，否则创建 AndroidAudioStream 对象。将音频路由到 Audio Flinger。

然后通过 PVMediaOutputNodeFactory::CreateMediaOutputNode()方法创建一个输出节点，并将输出节点添加到音频槽中。

当播放引擎收到自身发送的 PLAYER_SET_DATA_SOURCE 命令时，会调用 handleSetDataSource()方法进行处理，播放引擎首先从命令中提取出 URL，然后对 URL 类型进行判断。如果是 RTSP，则调用 PVPlayerDataSourceURL::SetDataSourceFormatType() 方法设置数据源类型为 PVMF_MIME_DATA_SOURCE_RTSP_URL；如果是 HTTP，则调用 PlayerDriver::setupHttpStreamPre()方法设置数据源类型为 PVMF_MIME_DATA_SOURCE_ HTTP_URL 并设置上下文。

当数据源是文件时，如果后缀为 sdp，则设置数据源类型为 PVMF_MIME_DATA_ SOURCE_SDP_FILE，然后设置上下文。

接着通过 PVPlayerEngine::AddDataSource() 方法向播放引擎发送 PVP_ENGINE_COMMAND_ADD_DATA_SOURCE 命令，PVPlayerEngine 引擎会在 DoAddDataSource()方法中进行处理。

为了添加数据源，首先要判断播放引擎状态是不是 PVP_STATE_IDLE，如果是，则进行 DoSetupSourceNode()方法的调用，通过 PVPlayerNodeRegistry:: QueryRegistry()方法进行解码器节点的查询，并创建节点。

如果是进行 MP4 的播放，创建的节点为 Mpeg4Decoder_OMX。

解码器本身的实现比较复杂，在本书中提供了 AMR 和 MP3 的解码过程，可以参考第 6.2.2 节 AMR 的解码过程和第 6.2.3 节 MP3 的解码过程。视频的解码过程可以参考 7.3.2 节视频的解码过程。

4.3.5 OSCL 底层移植

OSCL 在 OpenCORE 中扮演了举足轻重的角色，提供了跨平台特性的支持，要求底层系统能够提供如动态内容管理、线程、文件 I/O、网络套接字、DNS（Domain Name Services）和时间服务等。

如果希望将 OpenCORE 向一个新平台移植，主要工作就是在 OSCL 层进行的。由于 OSCL 涉及的硬件细节较多，不是本书的重点，在此就不做过多介绍了。下面介绍 OSCL 中涉及的最常用的字符编码格式的转换情况。

Unicode 向 UTF8 转换的算法为：

代码 4-7　Unicode 向 UTF8 转换的过程

```
    OSCL_EXPORT_REF int32 oscl_UnicodeToUTF8(const oscl_wchar *szSrc, int32 nSrcLen, char *strDest, int32
nDestLen)
    {
       int32 i=0;
       int32 i_cur_output=0;
       char ch_tmp_byte;

       if (nDestLen<=0)
       {
          return 0; /* ERROR_INSUFFICIENT_BUFFER */
       }

       for (i=0; i<nSrcLen; i++)
       {
          if (BYTE_1_REP>szSrc[i]) /* 1 个字节 utf8 表示*/
          {
             if (i_cur_output+1<nDestLen)
             {
                strDest[i_cur_output++]=(char)szSrc[i];
             }
             else
             {
                    //设置结束符
                strDest[i_cur_output]='\0';
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
             }
          }
          else if (BYTE_2_REP>szSrc[i]) /* 2 个字节 utf8 表示*/
          {
             if (i_cur_output+2<nDestLen)
             {
                strDest[i_cur_output++]=(char)(szSrc[i]>>6 | 0xc0);
                strDest[i_cur_output++]=(char)((szSrc[i] & 0x3f) | 0x80);
             }
             else
             {
                strDest[i_cur_output]='\0'; /* Terminate string */
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
             }
          }
          else if (SURROGATE_MAX>szSrc[i] && SURROGATE_MIN<szSrc[i])
    {       /* 4 个字节代理对表示*/
             if (i_cur_output+4<nDestLen)
             {
                ch_tmp_byte=(char)(((szSrc[i] & 0x3c0)>>6)+1);
```

```
                strDest[i_cur_output++]=(char)(ch_tmp_byte>>2 | 0xf0);
                strDest[i_cur_output++]=(char)(((ch_tmp_byte & 0x03) | 0x80) | (szSrc[i]
        & 0x3e) >> 2);
            }
            else
            {
                    //设置结束符
                strDest[i_cur_output]='\0';
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
            }
        }
        else /* 3 个字节 utf8 表示*/
        {
            if (i_cur_output+3<nDestLen)
            {
                strDest[i_cur_output++]=(char)(szSrc[i]>>12 | 0xe0);
                strDest[i_cur_output++]=(char)(((szSrc[i]>>6)  & 0x3f) | 0x80);
                strDest[i_cur_output++]=(char)((szSrc[i] & 0x3f) | 0x80);
            }
            else
            {
                    //设置结束符
                strDest[i_cur_output]='\0';
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
            }
        }
    }
                //设置结束符
    strDest[i_cur_output]='\0';
     //以字节为单位返回
    return i_cur_output;
}
```

UTF8 向 Unicode 转换的算法为：

代码 4-8 UTF8 向 Unicode 转换的过程

```
OSCL_EXPORT_REF int32 oscl_UTF8ToUnicode(const char *szSrc, int32 nSrcLen, oscl_wchar *strDest, int32
nDestLen)
{
    int32 i=0;
    int32 i_cur_output=0;

    if (nDestLen<=0)
    {
        // We cannot append terminate 0 at this case.
        return 0; /* ERROR_INSUFFICIENT_BUFFER */
    }

    unsigned char *pszSrc=(unsigned char *)szSrc;
    while (i<nSrcLen)
    {
      //处理代理对
        if (SIGMASK_3_1 <= pszSrc[i])
        {
            if (i+2<nSrcLen && i_cur_output+1<nDestLen)
            {
                strDest[i_cur_output++]=(wchar_t)(((wchar_t)pszSrc[i]<<12) |
                                                  (((wchar_t)pszSrc[i+1] & 0x3f)<<6) |
                                                  ((wchar_t)pszSrc[i+2] & 0x3f));
                i += 3;
            }
            else
```

```
            {
                    //设置结束符
                strDest[i_cur_output]=0;
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
            }
        }
        else if (SIGMASK_2_1 <= pszSrc[i]) /*
        {
            if (i+1<nSrcLen && i_cur_output+1<nDestLen)
            {
                strDest[i_cur_output++]=(wchar_t)(((wchar_t)pszSrc[i] & ~0xc0)<<6 |
                                                  ((wchar_t)pszSrc[i+1] & ~0x80));
                i+=2;
            }
            else
            {
                    //设置结束符
                strDest[i_cur_output]=0; /
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
            }
        }
        else /*单个字节表示*/
        {
            if (i<nSrcLen && i_cur_output+1<nDestLen)
            {
                strDest[i_cur_output++]=(wchar_t)pszSrc[i];
                ++i;
            }
            else
            {
                    //设置结束符
                strDest[i_cur_output]=0;
                return 0; /* ERROR_INSUFFICIENT_BUFFER */
            }
        }
    }
                    //设置结束符
    strDest[i_cur_output]=0;
    return i_cur_output;
}
```

4.3.6 A/V 同步

播放引擎在渲染（Render）多媒体数据时需要保持一个暂时的同步，也就是通常所说的 A/V 同步。为了达到 A/V 同步，需要如下信息：媒体回放的时钟、媒体数据的时间戳、从 Sink 中获取的时间信息（比如从音频设备设定的特定的采样率来获取的播放速率）等。

1．媒体时钟

PVMFMediaClock，媒体时钟主要负责维持一个时间的引用，从而保持媒体回放的节奏，获取和实现媒体播放的同步。

1）时间源

媒体时钟可以作为一个时间源提供给多媒体，它本身可能来自于系统时钟或其他时间源（如音频设备时钟）。它可以给多媒体提供一个时间基准，同时来维护该时间基准。

2）时钟观察者

媒体时钟可以把自己作为一个观察者，来通知对象时钟状态的改变。以下接口实现了其作为观察者的

角色。

PVMFMediaClockObserver：用来通知时钟基值，时钟计数的更新，时钟的调整。

PVMFMediaClockStateObserver：用来通知时钟状态的改变。

PVMFMediaClockNotificationsObs：用来获取回调通知。

3）NPT 映射

媒体时钟是一个单调递增的时钟，而媒体在播放时却可能需要定位（Seek）到任意位置。为了控制媒体的播放，使其正确渲染 ，媒体时钟需要在媒体时钟时间和 NPT 之间维护一个 NPT （ Normal Play Time ）映射，任意对媒体播放位置的改变将会通知进行一次映射。

媒体时钟和 NPT 之间的映射公式为：NPT＝(media_time－5550+380) 。

4）时钟的回调

在媒体时钟上设置回调是组件采取动作的基础，这些回调可以减少在时钟发生改变时，组件自己需要设置它们的时钟的时间。媒体时钟采用输入特定时间窗口来取代绝对时间，这样可以使处于竞争状态的任务或线程尽可能早的得到响应。

5）延迟处理

当集成了多个不同媒体流的数据槽同时输出时，每个槽都可能会有不同程度的延迟，为了弥补不同媒体流之间的延迟从而同步播放，就需要进行延迟处理。每个 Sink 都向媒体时钟注册自己的延迟，最后由媒体时钟来调整最终调度的延迟。

6）NPT 时钟转换

当一个新的 NPT 开始时，用户可以给媒体时间设置一个绝对时间。用户还可以任意调整 NPT 的方向（如向前、向后）。

2．时间戳

为了及时，准确地输出媒体数据，就不得不考虑媒体数据中包含的时间戳信息，以及媒体回放时钟。如果时间戳值等于当前回放时间，则媒体数据是同步的，需要进行渲染 ；如果时间戳小于当前回放时间，则说明媒体数据到达时间晚了；反之，如果时间戳大于当前回放时间，则说明媒体数据到达时间早了。如何处理这些来早的或来晚的媒体数据则取决于 PVPlayer 引擎的配置。在通常情况下，来早的数据需要等待，直到播放时间到达；来晚的数据则会被丢弃而不被渲染，但有时候来晚的数据也会被渲染 。

3．同步音频

音频数据的渲染通常不需要外部时钟来进行同步，因为音频设备通常会被配置一定的采样率来消化音频数据。因此，音频设备被配置的这个采样率通常也作为媒体回放时钟的速率。

1）渲染开始时的同步

一旦媒体时钟开始后，就必须要求媒体数据尽可能快地被渲染，然而硬件在渲染时很可能需要额外的时间，或者硬件需要等到更多的媒体数据被缓存。因此会导致媒体时钟的开始时间与媒体数据真正被输出的时间不一致，从而导致播放引擎报告给应用程序的播放进度与真实播放进度产生误差。为了解决这一问题，在硬件没有开始输出媒体数据时，将媒体时钟设置暂停状态，当硬件开始输出媒体数据时，给媒体时钟发一个消息来通知媒体时钟也开始运行。这样就可以保证在媒体时钟开始的时候，媒体数据也开始进行输出。

作为一个主动态的 MIO 组件，无论时钟状态为暂停还是运行，播放引擎都向 MIO 组件传送数据，而 MIO 组件也应该继续接受和缓存数据，并决定何时把数据发送到硬件。

而作为一个被动态的 MIO 组件，只有当时钟状态为运行时，播放引擎才向 MIO 组件发送数据，而接收到数据的 MIO 组件也需要将收到的数据立即发送给硬件。

2）重新定位后的同步

当应用程序请求重新定位媒体播放位置时，MIO 组件和硬件缓存的数据需要被立即释放，并且在新的位置开始播放。

3）播放中的同步

尽管硬件消费数据的速率应该与媒体时钟的速率是一致的，但他们毕竟是单独运行的，因此不可避免会有一些不同。在通常情况下，这中间的差距会很小，然而随着播放的进行，差距将会被积累，最终导致不同步。

因此，为了控制短时间内播放时钟与音频输出进程差距在很小的范围内，需要不时地调整两者之间的差，使之小于一个特定的阈值。

4．同步视频

与音频输出相比，视频的输出需要参考一个时钟来决定何时输出一个特定的视频帧，视频帧的输出要尽可能与该帧的时间戳保持一致。 播放引擎维护了一个与音频播放同步的播放时钟，因此一旦视频输出与播放时钟同步，那么也就意味着视频输出与音频输出同步。

5．音视频同步

音视频同步是音频同步和视频同步的终极目标。在播放引擎架构中，媒体时钟需要调整以便与音频输出过程相一致，而对于视频输出来说，在输出一个视频帧时，应使该帧的时间戳与媒体时钟同步。因此音频输出设备、视频帧的时间戳与媒体时钟的结合便构成了 A/V 同步。

A/V 同步的具体实现，在本书中不是介绍的重点，这里就不再详述了。关于媒体时钟的内容请参考 external\opencore\pvmi\pvmf\src\Pvmf_media_clock.cpp 文件。

4.4 Stagefright 框架

Stagefright 框架是 Android 2.3 正式引入的多媒体框架，其在 Android 2.0 中就已经添加到 Android 的代码库中。Stagefright 框架的引入对上层应用没有影响，其接口在原生的多媒体服务中引入。

```
static sp<MediaPlayerBase> createPlayer(player_type playerType, void* cookie,
       notify_callback_f notifyFunc)
{
         ……
         case STAGEFRIGHT_PLAYER:
   LOGV(" create StagefrightPlayer");
   p = new StagefrightPlayer;
   break;
        ……
}
```

Stagefright 框架和 openCORE 框架切换非常简单，目前 Stagefright 已经成为默认的媒体播放器。

```
static player_type getDefaultPlayerType() {
   return STAGEFRIGHT_PLAYER;
}
```

Stagefright 框架并没有完全抛弃 openCORE 框架的内容，而是封装了一个 OMX 层用于引用 openCORE 的 OMX 组件部分。

Stagefright 框架本身同样比较复杂，下面分播放框架和记录框架两部分进行简要说明。

1．播放框架

Stagefright 的播放主要是围绕着 AwesomePlayer 进行的， Stagefright 播放框架的类图如图 4-28 所示。

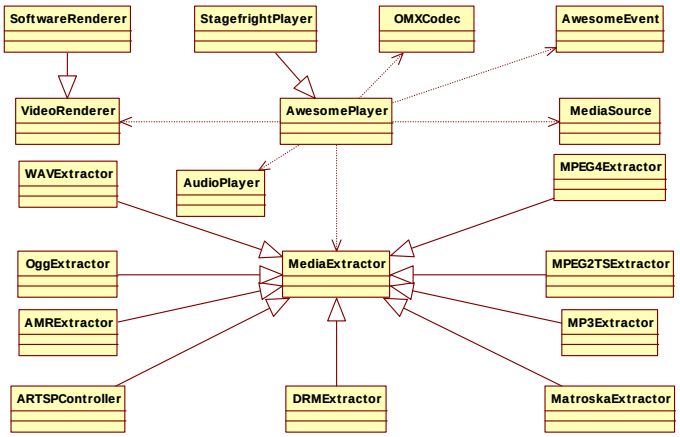

图 4-28　Stagefright 播放框架的类图

在 Stagefright 播放框架中，AwesomePlayer 是最重要的一个类，如果支持硬件视频渲染，则需要对 VideoRenderer 进行实现，在 Qualcomm 平台上，相应的类为 QComHardwareRenderer。如果不支持硬件视频渲染，系统将采取软件渲染的方式，相应的渲染类为 SoftwareRenderer。当然软件渲染会会造成系统性能的下降。也不支持高清视频。

解析器的封装是通过 MediaExtractor 类进行的，对于不同的多媒体格式，针对 MediaExtractor 进行实现即可，目前 Stagefright 支持的媒体播放格式包括 WAV、OGG、MP4、AMR、MP3 等。

AwesomePlayer 通过 AwesomeEvent 事件来驱动整个播放过程。

具体的解码过程通过 OMXCodec 根据文件格式调用不同的解码器进行解码，Stagefright 播放框架提供的解码器包括 AMRNBDecoder、AMRWBDecoder、AACDecoder、AVCDecoder、G711Decoder、M4vH263Decoder、VorbisDecoder、VPXDecoder 等。

2．记录框架

在 Stagefright 记录框架中，StagefrightRecorder 是最重要的一个类。Stagefright 记录框架的类图如图 4-29 所示。

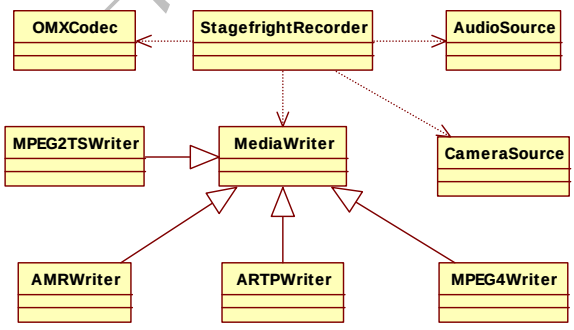

图 4-29　Stagefright 记录框架的类图

Stagefright 记录框架的音频数据源通过 AudioSource 配置，视频数据源通过 CameraSource 配置。针对不同多媒体格式的记录过程，Stagefright 记录框架提供了 AMRWriter、MPEG2TSWriter、MPEG4Writer 等类进行支持。

具体的编码过程通过 OMXCodec 根据文件格式调用不同的编码器进行编码，Stagefright 记录框架提供的编码器包括 AMRNBEncoder、AMRWBEncoder、AACEncoder、AVCEncoder、M4vH263Encoder 等。

4.5 元数据

对于实际的数据而言，为了在单位内存或者单位带宽上传递更多的信息，必须对数据进行压缩，减小冗余，在媒体播放时，又必须将数据还原为元数据。其中图像元数据包括 RGB 和 YUV，音频元数据包括 PCM。

通常视频文件是由压缩视频流和压缩音频流组成的包文件。

4.5.1 色彩模式

在常用的视频和图像编码中，色彩模式主要包括 RGB 和 YUV 两大类。当然在印刷业，采用的色彩模式为 CMYK，在本书中，所讲述的内容自然与印刷无关，重点介绍 RGB 和 YUV，RGB 和 YUV 根据各自子通道的不同，又有许多细分。

1．RGB

RGB 色彩模式是指通过对红（Red）、绿（Green）、蓝（Blue）3 个颜色通道的变化，以及它们相互之间的叠加来得到颜色的一种标准。

目前在显示器领域多采用这种颜色方案，在 CRT 显示器上，是通过电子枪打在屏幕的红、绿、蓝三色发光极来产生色彩的。而在 LED 显示器上，则是利用了三合一点阵全彩技术，即在一个发光单元里有 RGB 三色晶片组成全彩像素。

RGB 色彩模式在输出时需要 3 个独立的图像信号同时传输，带宽占用较高。

常见的 RGB 格式包括：RGB1、RGB4、RGB8、RGB565、RGB555、RGB24、RGB32、ARGB32 等。在 OpenCORE 中，目前支持的 RGB 格式包括：RGB8、RGB12 、RGB16、RGB24 等。RGB 模式通常用于最原始的视频数据和图像。

需要说明的是 ARGB 中的 A 表示的是 Alpha 通道，通常理解为透明度，这在软件开发者，是个比较重要的概念。

对于 UI 设计而言，还应注意安全色和透明色的概念，如何能够使自己的设计吸引人，能够将想要表达的信息完整、准确的传递给用户，是个很大的学问。

对于软件开发者而言，对 RGB 格式也应有基本的了解，特别是在进行多媒体相关的编程时，会时常需要这方面的知识，例如在开发 Android 照相机应用时，如果希望加入人脸检测的功能，目前 Android 仅对 RGB565 格式的图像提供了人脸检测支持。

2．YUV

YUV 色彩模式在早期主要是 PAL 和 SECAM 模拟彩电制式采用的颜色空间，其中 Y 代表亮度即灰阶值，U、V 代表色度即色调和饱和度，U 和 V 是构成彩色的两个分量。如果只有 Y 信号分量而没有 U、V 信号分量，则图像显示为黑白灰度图像，基于 YUV 色彩空间可以有效解决彩电和黑白电视的相容问题。

在实际的编码中，色调用 Cr 表示，即反映了 RGB 输入信号红色部分和 RGB 信号亮度值之间的差异；饱和度用 Cb 表示，即反映了 RGB 输入信号蓝色部分和 RGB 信号亮度值之间的差异。

常见的 YUV 格式包括：YUY2、YUYV、YVYU、UYVY、AYUV、Y41P、Y411、Y211、IF09、IYUV、YV12、YVU9、YUV411、YUV420 等。在 OpenCORE 中，目前支持的 YUV 格式包括：YUV420、YUV422 等。YUV 模式通常用于视频处理。

在多媒体开发中，尤其是驱动调试中，YUV 的概念需要了解。

4.5.2 脉冲调制

PCM 编码调制数字音频格式是 20 世纪 70 年代末发展起来的，在 80 年代初由飞利浦和索尼公司共同

推出。PCM 的音频格式也被 DVD-A 所采用，它支持立体声和 5.1 环绕声，1999 年由 DVD 讨论会发布和推出。

PCM 编码必须经过 3 个过程，即抽样、量化和编码。PCM 编码的主要过程是将话音、图像等模拟信号每隔一定时间进行取样，使其离散化，同时将抽样值按分层单位四舍五入取整量化，同时将抽样值按一组二进制码来表示抽样脉冲的幅值，以实现话音数字化。PCM 编码的最大的优点是音质好，最大的缺点是体积大。我们常见的 Audio CD 就采用了 PCM 编码，一张光盘的容量只能容纳 72 分钟的音乐信息。

PCM 的采样精度从 14bit 发展到 16bit、18bit、20bit 直到 24bit；采样频率从 44.1kHz 发展到 192kHz。到目前为止 PCM 这种单纯依赖提高采样规格的技术，其可改进的地方已经越来越来小。简单地增加 PCM 比特率和采样率，不能从底层改善它的根本问题。

在将其他音频格式转换为 PCM 并输出到硬件音频设备的过程中，采用的 PCM 采样精度通常是 16bit。

另外，常见的 WAV 格式的音频文件的音频编码即 PCM 编码。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218