华南理工大学

South China University of Technology

硕士学位论文

FFmpeg 在 Android 多媒体平台下的编码优化研究

作者姓名　华耀波

学科专业　电路与系统

指导教师　吴宗泽　副教授

所在学院　电子与信息学院

论文提交日期　2014 年 5 月

Research on The Optimization of FFmpeg Encoder in Android-based Multimedia

A Dissertation Submitted for the Degree of Master

Candidate：Hua yaobo

Supervisor：Prof. Wu Zongze

South China University of Technology

Guangzhou, China

分类号： TP37　　**学校代号：**10561

学　号：201120107543

华南理工大学硕士学位论文

FFmpeg在Android多媒体平台下的编码优化研究

作者姓名：华耀波　　指导教师姓名、职称：吴宗泽 副教授

申请学位级别：全日制工学硕士　　学科专业名称：电路与系统

研究方向：　图像技术与智能系统

论文提交日期：2014 年 5 月 3 日　　论文答辩日期：　2014 年 6 月 7 日

学位授予单位：华南理工大学　　学位授予日期：　　年　月　日

答辩委员会成员：

主席：　周智恒副教授

委员：　傅予力教授，　吴宗泽副教授，　李波副教授，　向友君副教授

华南理工大学
学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：华耀波　　日期：2014 年 6 月 7 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属华南理工大学。学校有权保存并向国家有关部门或机构送交论文的复印件和电子版，允许学位论文被查阅（除在保密期内的保密论文外）；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。本人电子文档的内容和纸质论文的内容相一致。

本学位论文属于：

□保密，在年解密后适用本授权书。

☑不保密，同意在校园网上发布，供校内师生和与学校有共享协议的单位浏览；同意将本人学位论文提交中国学术期刊(光盘版)电子杂志社全文出版和编入CNKI《中国知识资源总库》，传播学位论文的全部或部分内容。

（请在以上相应方框内打“√”）

作者签名：华耀波　　日期：2014.6.7

指导教师签名：　　日期：2014.6.7

摘　　要

在移动互联网飞速发展的现今，人们对移动终端的需求不再满足于发短信和通电话了，越来越多的人都将手机作为了日常生活的消遣和娱乐工具。伴随着智能手机的普及以及性能的提高，多媒体应用也层出不穷，可以说在未来的移动互联网领域中，多媒体应用将占据着举足轻重的地位。Android 智能手机作为移动互联网的载体，是目前最为流行的智能手机之一，其在全球的市场份额已经达到了 78.1%，因此吸引了很多的开发者投向 Android 阵型。然而，Android 系统上层的多媒体接口并没有像它的市场份额一样发展迅速，很多涉及到多媒体流式传输和可视化通信的应用都是基于 Android 底层开发，开发者首先需要熟悉 Android 的底层代码，而不同开发者又有不同的实现方式。这样的结果导致了 Android 多媒体应用的开发难度大，开发周期长。

为了实现一个通用的 Android 多媒体开发方案，本文将目前最为流行的音视频编解码库 FFmpeg 整合到 Android 多媒体中，通过使用 FFmpeg 的接口来达到开发 Android 多媒体应用的目的。针对于 FFmpeg 只能软件编码从而导致编码效率过低的问题，本文还提出了优化 FFmpeg 编码器的方案设计。在深入理解 Android 多媒体编码机制的基础上，从中提取出硬件编码器，并将其设计成插件的形式融合到 FFmpeg 中，实现高效的编码。

为了能够体现 FFmpeg 优化后的编码性能，本文设定了在三种视频分辨率的环境下，对优化前后的视频编码进行比较。分别记录不同分辨率下，它们的 PSNR 值、编码平均效率、CPU 平均使用率和内存平均使用率。结果表明：从 PSNR 值可以发现优化后的视频编码质量要比优化前的稍微好一些，但从视觉效果上是完全相同的。由于高负荷工作导致 CPU 平均使用率相差并不大；而在 144x176 分辨率下，优化后的 FFmpeg 比优化前具有高出将近 8 倍的编码效率，而且分辨率越高，编码效率相差的倍数也越高。对于内存的平均使用率，优化后比优化前降低了约 2 倍。因此，优化后的 FFmpeg 编码器更能适用于 Android 平台。

关键词： Android 多媒体应用； FFmpeg 整合；编码优化

Abstract

With the trend of rapid development of mobile Internet nowadays, the demand for the mobile terminals are no longer constrained to sending SMS and making phone calls. More and more people have regarded mobile phones as an entertainment tool in the daily life. With the popularity of smart phones as well as the improvement of its performance, multimedia applications are just emerging in the endless stream. So it is said that the multimedia applications will occupy a pivotal position in the field of mobile Internet in the future. As a mobile Internet carrier, Android smartphone is currently one of the most popular smart phones, which has taken about 78.1 percent of the global market share and attracts a lot of developers to come around. However, the upper multimedia Java interface of Android system is not expanding so rapidly as that of its market share. As many applications dealing with multimedia streaming transmission and visualisation communication are based on the infrastructure development of Andriod, it will not be possible to for developers to get started on the work before they can get familiar with the underlying code of Andriod. And what's even worse is that different developers will have different implementations. All these phenomenons will it more difficult to develop new Andriod multimedia applications and the development cycle will be relatively longer than before.

In order to achieve a common Android multimedia development solution and reduce the development cycle for developers, this paper trys to integrate the most popular audio and video codec library FFmpeg into the Android multimedia, and make use of the FFmpeg interface to achieve the purpose of the development of Android multimedia applications.On the other hand, this paper also proposes another solution to optimize the FFmpeg encoder,in order to deal with the problem that FFmpeg can only be targeted at software coding, which results in low coding efficiency. What this paper does is to extract the hardware encoder with the in-depth understanding of the mechanism of Andriod multimedia encoding and try to integrate the encoder into FFmpeg as the format of a plug to achieve the efficient coding.

In order to fully reflect the coding performance after the optimization with FFmpeg, this paper trys to compare the results of video encoding before and after the optimization in three different video resolution environment. The average coding efficiency, the average usage of CPU and memory are recorded respectively at different resolutions. It showed that:

It can be found from the perspective of the PSNR value that the video encoding quality is slightly better after optimization than before, but the video effect is exactly the same. The difference of the average CPU usage is not big due to the high load work of our environment. While in the resolution of 144X176, the coding efficiency after optimization of FFmpeg is nearly eight times higher than that before optimisation of FFmpeg. And the higher the resolution is, the higher the mutiples of coding efficiency will be. When it comes to the matters of average usage of memory, it shows that the average usage after optimization is reduced by about 2-fold than that before optimization. Therefore, we can get the conclusion that the optimized FFmpeg encoder is even sutiable for Android platform comparing with no optimized encoder.

Keyword: Android multimedia application; FFmpeg integration; code optimization

目录

第一章 绪论

1.1 研究背景及意义

FFmpeg 作为一个开源免费跨平台的视频和音频流方案，不仅包含了丰富和先进的音视频编解码器，同时还提供了非常多对音视频的处理，比如添加水印、视频切割、比例缩放、颜色转换、视频录制等，甚至还能够实现视频的流式传输，支持 HTTP、RTSP、RTMP 等网络实时传输协议。正因为 FFmpeg 的强大用途以及开源，使得大量的项目都使用了它，或者在它基础上进行修改，其中一些有名的项目包括有 MPlayer、QQ 影音、暴风影音、VLC、GStreamer 等等，另外像 Ubuntu 这样的 Linux 系统也添加了对它的支持。尽管 FFmpeg 应用非常广泛并且跨平台，但在嵌入式开发中，却极少使用到它的编码器。因为 FFmpeg 提供的是软件编码，如果要实现软件编码，就需要处理器有强大的运算能力来支撑，而这恰恰是嵌入式所欠缺的。所以 FFmpeg 应用在嵌入式时，更多的是利用它的解码功能来开发视频播放器等应用。

然而，随着移动互联网的蓬勃发展，嵌入式的应用也越来越广泛，加上移动网络技术的不断升级，人们在日常生活中变得不再仅仅是想获取咨询了，越来越多的人希望能够创作和分享属于自己的作品，这些作品更多的是指视频。因此视频编码技术在嵌入式的使用日渐重要。

在移动互联网的发展中，以 Android 智能手机的发展最为突出。Android 作为一种基于 Linux 的自由及开源的嵌入式操作系统，于 2008 年 9 月正式发布了 1.0 版本。在之后的几年时间里，迅速发展，市场占有率不断上涨。2011 年第一季度，Android 在全球的市场份额首次超过 Symbian 系统，跃居全球第一。截至 2013 年第四季度，Android 手机的全球市场份额已经达到 78.1%，用户数也接近 10 亿，较 2012 年的 69.7%还增长了 8.4%，这样的增长率对已经饱和的智能手机市场而言，实属不易[1]。Android 的流行，加上 Google 对自家系统的大力支持，使得 Android 在短短的几年里更新迭代了多个重大版本。因此，在这样一个具有传奇色彩的嵌入式操作系统上研究其多媒体技术显得更加之有意义。

随着 Android 系统的不断更新，Android 默认的多媒体框架也伴随着由 OpenCore 改

成了 StageFright。这样转变的原因也很明显：OpenCore 框架对于嵌入式应用来说仍然过于庞大，改用 StageFright 也是出于精简系统的考虑。虽然如此，StageFright 却保留了 OpenCore 框架中使用到的 OpenMAX IL 引擎，这是因为 OpenMAX IL 引擎是与硬件编解码的结合。在嵌入式开发中，硬件编码显得异常重要，它能够使得编码的效率比软件编码高出许多，从而满足人们对高清视频的创作需求。然而，StageFright 仍然存在一些自身的问题：

（1） 应用不广泛。StageFright 主要是应用于 Android 底层，很少有人将其用在其他项目中。如果在 StageFright 框架下开发，那么就需要首先熟悉这个框架，这无疑是增加了开发的时间成本。相比之下，人们更愿意使用已掌握的 FFmpeg 技术进行开发。

（2） Android 版本的频繁迭代更新，也导致了 StageFright 接口的不稳定。如果使用 StageFright 进行开发，那么开发出来的应用也要根据 Android 的版本来使用不同接口的 StageFright。这导致了代码的重用性很低。

（3） StageFright 支持的格式很少，特别是对流媒体的支持力度不够。虽然 StageFright 自身也带有 SIP，RTSP 等流式传输协议，但这些协议目前只能用于音频，或者用于视频的接收端，而缺乏对视频流式上传的支持。然而，面对如今进入移动互联多媒体的繁盛时期，多媒体的流式传输已日益受到重视，未来这个领域必将会成为世界的潮流。

经过上述对 FFmpeg 和 Android 多媒体框架 StageFright 的优劣分析可以得出以下结论：FFmpeg 应用非常广泛，支持丰富的视频处理技术，但其软件编码不适用于嵌入式；StageFright 变化大，应用不广泛并且支持的格式少，但结合了硬件编码，具有非常高效的编码效率。在这里我们发现了虽然两者都有缺点，但却是互补的关系。如果将它们整合起来，那么这些缺点就不复存在了，这也正是本论文研究 FFmpeg 在 Android 多媒体平台下的编码优化的意义所在。FFmpeg 将 Android 多媒体框架中的硬件编码以 Codec 库的方式加载进来，然后结合自身丰富的多媒体格式，可以实现多媒体的各种需求，比如流式传输，高清视频录制，视频通话等。另外，它可以使得 FFmpeg 开发人员无需了解 Android 多媒体框架的实现，只要像原来一样使用 FFmpeg 的接口就可以利用上 Android 的多媒体硬件资源。对于已有的 FFmpeg 项目，只要支持跨平台，都可以在不

做大修改的前提下移植到 Android，并能实现编码加速。

1.2 研究现状

在 Android 版本的频繁迭代更新和人们对娱乐多媒体的日渐依赖下，Android 系统下的多媒体服务却没有相应的得到提高。相反的，Android 多媒体框架由 OpenCore 转向 StageFright 之后，支持的多媒体格式变得更加少了。也因此，引发了一些人对 Android 多媒体框架的扩展研究。

2012 年中国科学院的童方圆等人在 Android 平台上扩展了基于点对点的实时视频流传输系统[2]；电子科技大学的一篇硕士论文中讲述了在 Android 平台下移植第三方的多媒体框架 xCore，以此来兼容更多的多媒体格式[3]；而在《Android 系统多媒体功能增强的研究与实现》一文中，作者将 Android 多媒体框架 StageFright 与 V1 解码芯片做了融合，并从 FFmpeg 中提出 WMA 格式代码，植入到 Android 多媒体框架中，从而实现了编解码的扩展[4]；在 2013 年北京航空航天大学的温伟等人更是将 FFmpeg 中的 Muxer/Demuxer 和 Codec 拆分开来，并封装成插件的形式供 StageFright 调用，从而也实现了多媒体格式的扩展[5]。

对于这些扩展性的研究，虽然一定程度上是丰富了 Android 的多媒体框架，但却都不能成为通用的解决方法。比如开发实时流传输功能，只是针对这种特定的需求而开发的，并没有什么可扩展性；移植第三方的多媒体框架 xCore，却不能使用移动终端自身的硬件资源，根本就不适合视频的编码。又比如在已经定型的 Android 设备上扩展编解码芯片，除非是专门定制，否则用户很难接受在自己手机上再增加这样的一个芯片；将 FFmpeg 进行拆分，从而做成 StageFright 的一个插件，这样的话，要开发应用就必须先要熟悉 StageFright 的接口，并且要将那么多的 Muxer/Demuxer 和 Codec 做成插件也是一件非常复杂和繁琐的事情。

而在本论文中，提出的 FFmpeg 编码优化方案，是指将 StageFright 框架下的硬件编码库做为 FFmpeg 的一个插件，开发者利用 FFmpeg 的接口进行开发。这样做能够很好地解决了上述方案所提的缺点，为别人提供一个通用的多媒体应用开发方案。

1.3 文章组织结构

全文共分为6章，各章节的内容概括如下：

第一章是文章的绪论。主要介绍FFmpeg在Android多媒体平台下进行编码优化的研究背景和意义，阐述了Android平台下多媒体的扩展研究现状，最后给出论文的组织结构。

第二章在Android源码的基础上对Android多媒体框架进行了研究。分析了OpenCore和StageFright框架的差异，重点讲述了StageFright的实现流程。同时对OpenMAX IL的结构以及工作原理进行了详细分析。最后通过视频解码的流程来展示OpenMAX IL在Android多媒体框架的应用。

第三章描述了FFmpeg编码库的研究与实现。本章主要对FFmpeg的相关数据结构和函数进行了分析，设计并实现了视频转码的例子，并详细说明了扩展编码器的流程。

第四章重点讲述了FFmpeg在Android多媒体平台下的编码优化设计与实现。包括整体架构的设计和实现，代码的移植等工作。

第五章从PNR值、单帧编码平均时间、CPU平均使用率、内存平均使用率三方面对优化前后的FFmpeg视频编码进行了测试和对比，结果表明了优化后的FFmpeg在Android平台下能够实现高效的编码工作。

第六章对本论文的相关技术研究进行了总结以及展望。

第二章 Android 多媒体框架研究

2.1 多媒体框架介绍

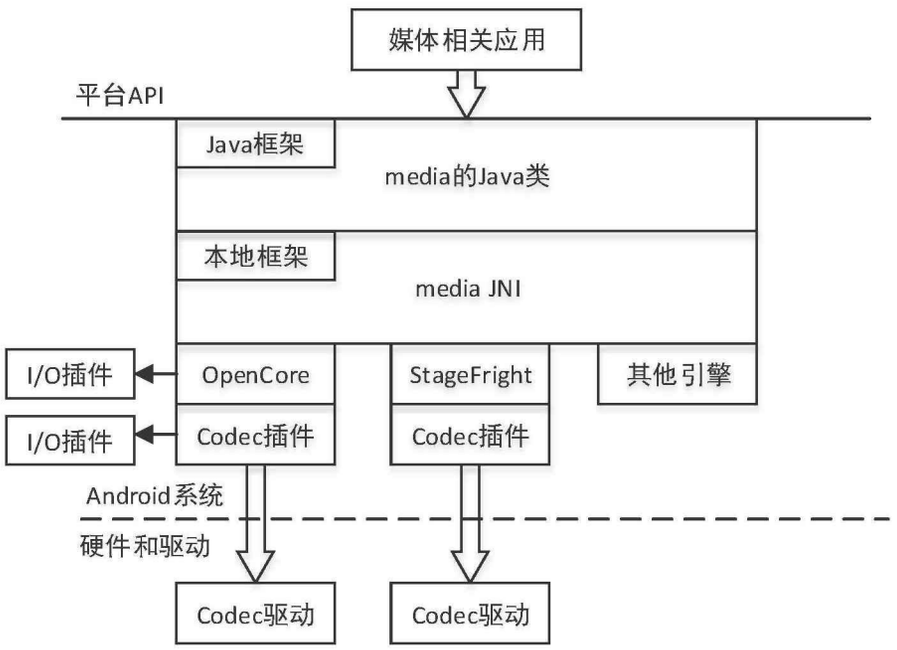

图 2-1 Android 多媒体系统

图 2-1 描述了 Android 的多媒体系统。在 Android 上层通过调用 Java 接口进行多媒体的录制、编解码和播放等操作。在 Android 2.0 之前，Android 只使用了 OpenCore 作为系统的多媒体框架解决方案。到 Android 2.0 之后，加入了 Stagefight 多媒体框架，并且有取代 OpenCore 的趋势。从 Android 2.3 的版本开始，系统预设的多媒体框架正式改为了 StageFright。

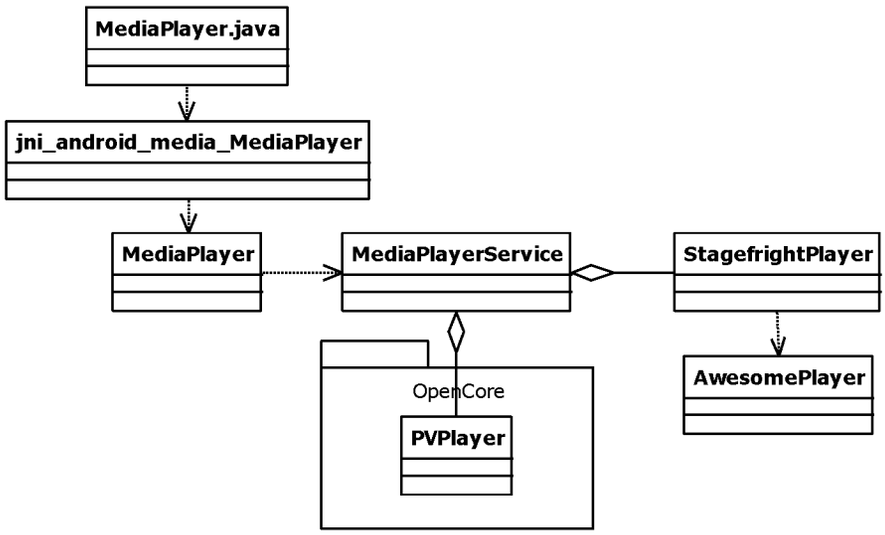

图 2-2 MediaPlayer.java 的内部实现

以多媒体的播放机制为例，从图 2-2 可知，OpenCore 和 StageFright 并列地提供视频播放组件，并且 MediaPlayerService 做了兼容性的整合，因此能够非常方便地在 OpenCore 和 StageFright 之间进行切换使用。

但这两种多媒体框架在内部设计上却有非常大的差异，而底层都是使用了 OpenMAX 多媒体引擎标准（OpenMAX 在 Android 中简称为 OMX）。

2.2 OpenCore 和 StageFright 框架比较

2.2.1 所支持的文件格式差异

OpenCore 和 StageFright 支持的格式

Name	File Format		Parser_node		Codec			OMX_component	
aac	parser	○	parser	○	dec		○◎	dec	○
	composer		composer		enc			enc	
amr	parser	○◎	parser	○◎	dec	nb	○◎	dec	○
						wb	○◎		
	composer	◎	composer	◎	enc	nb	○◎	enc	○
						wb			

wav	parser	○◎	parser	○◎				
	composer		composer					
mp3	parser	○◎	parser	○◎	dec	○◎	dec	○
	composer		composer		enc		enc	
avi	parser	○	parser					
	composer		composer					
mp4	parser	○◎	parser	○◎	dec	○◎	dec	○◎
	composer	○◎	composer	○◎	enc	○	enc	○
pvx	parser		parser					
	composer		composer					
sbv					dec		dec	
					enc	○	enc	
h264					dec	○	dec	○
					enc	○	enc	○

○ 表示 OpenCore 支持　　◎ 表示 StageFright 支持

表 2-1 OpenCore 和 StageFright 支持的文件格式对比

由表 2-1 可知，OpenCore 比 StageFright 支持更多的编解码格式，但也非常有限[2]。

2.2.2 扩展文件格式的实现机制差异

OpenCore 和 StageFright 具有比较明确的模块划分，这为文件格式的扩展提供了便利，文件格式扩展主要体现在 parser 和 codec 部分。对于 OpenCore，需要实现相应的 parser-node，而由于 OpenCore 使用 OMX IL 层作为编解码插件，所以 codec 的设计需要按照 OMX 的规范实现相应的 component。对于 StageFright，主要是实现相应的 extractor/writer 和 decoder/encoder。由于 StageFright 只在 OMX 接口之上简单地封装一层，所以 OpenCore 中实现的 codec 可以方便的应用到 StageFright 中[6]。

2.2.3 数据处理机制差异

以下通过文件播放的工作过程了说明 OpenCore 和 StageFright 的数据处理机制。

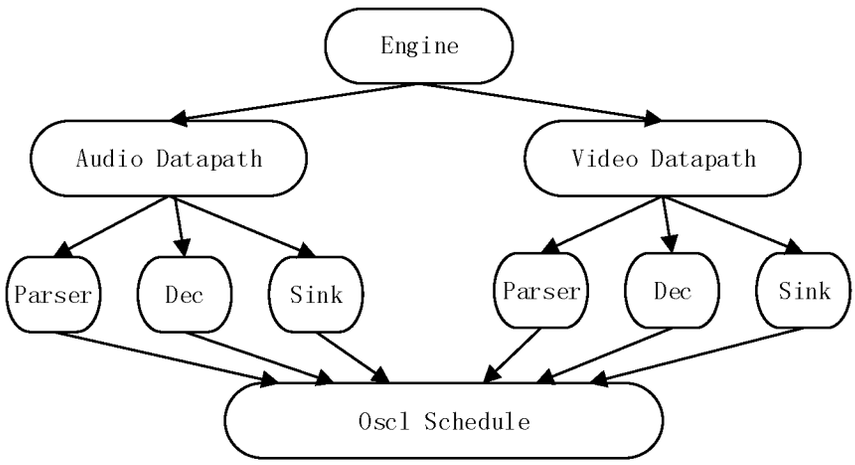

图 2-3 OpenCore 数据处理流程

图 2-3 中，OpenCore 多媒体引擎将文件数据分离出音频和视频，创建了 Audio 和 Video Datapath，这些 Datapath 会将各自相对应的 Paser、Decoder、Sink 节点连接起来，然后由 Oscl 统一调度。

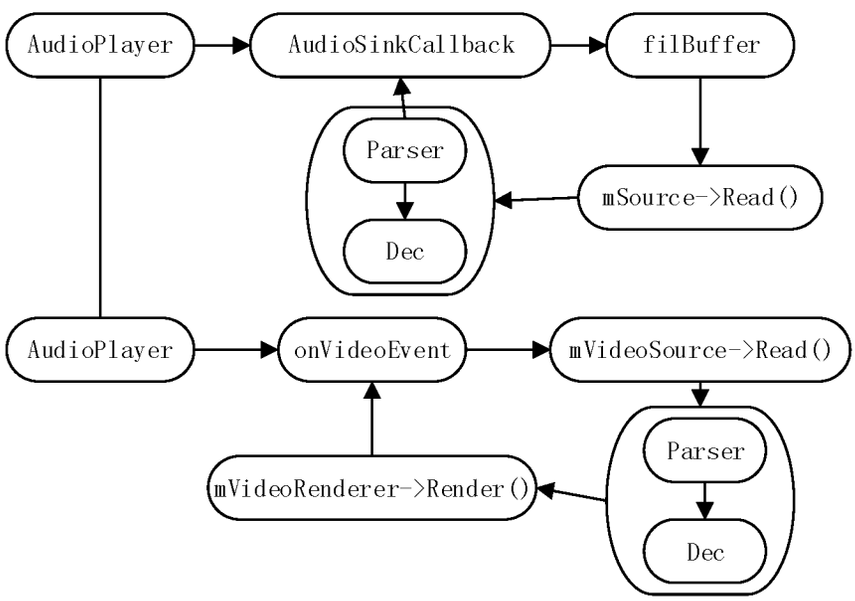

图 2-4 StageFright 数据处理流程

图 2-4 中，AudioPlayer 为 AwesomePlayer 的内部成员，分管音频数据的处理，主要是通过 AudioSinkCallback 来驱动。而视频主要是通过 AwesomePlayer 中的 onVideoEvent 来驱动。至于数据的处理过程，音频和视频都是一致的，都是通过 source->read 来获取数据，经过 paser 和 decode 后进行播放，最后会执行回调或事件通知，获取下一帧数据，

以此循环。

根据以上分析，OpenCore 和 StageFright 在数据处理方面的差异总结如下：

（1） OpenCore 的各个功能模块以节点形式存在；StageFright 将 paser 和 decoder 封装在了一起。

（2） OpenCore 使用了 Oscl 统一调度，并行处理；StageFright 则是串行处理。

（3） OpenCore 是通过节点控制输出；StageFright 是通过回调和事件通知来驱动数据输出。

2.2.4 音视频同步机制差异

OpenCore 在播放视频时，会启动一个独立的计时器，音频和视屏都是以该计时器作为基准进行播放。StageFright 会以音频时间作为基准，视频帧同步音频[7][8]。

2.3 多媒体框架详细分析

StageFright 由于具有比 OpenCore 更简洁的架构和更简单的扩展机制，从而使得 StageFright 受到了 Android 的青睐，将其设定为默认的多媒体框架。接下来，本节将主要对 StageFright 框架进行分析。

2.3.1 多媒体目录结构及其功能

在 Android 中，关于多媒体的相关源码存放在 frameworks/base/media 目录下，该目录下包含了几个文件夹，每个文件夹作为一个模块实现了不同的功能，其中主要的模块及其功能如下所述[9]：

（1） java：Java 层代码，包含了与多媒体相关的系统 Service 和应用层开发接口。

（2） jni：Java 与 C/C++之间交互的接口层。

（3） libmedia：作为多媒体 Server 的代理，实现与 Service 通信。由 Client 调用。

（4） libmediaplayerservice：多媒体 Service 的具体实现。

（5） libstagefright：stagefright 框架源码，多媒体 Service 的功能实现。

（6） mediaserver：MediaServer 进程的入口函数（main 函数）。

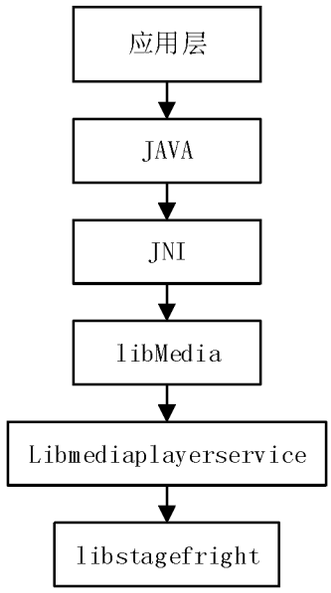

图 2-5 主要模块的层次结构

由图 2-5 所示，这几个模块共同完成了与多媒体相关的各种操作。应用层要实现某个功能时（比如媒体播放功能），首先是调用 java 模块中的相应接口，该接口实际上是通过 jni 模块调用了 libmedia 中的函数。libmedia 模块不会具体实现功能，而是利用了 Android 的 Binder 机制将需求提交给多媒体 Service，再由多媒体 Service 直接调用 libstagefright 模块中的相关功能函数，最终实现功能。由此可见，Android 多媒体使用了典型的 C/S 架构，用户通过 Client 和 Server 进行通信[10]。

上述中，提到了几个名称：多媒体 Server、多媒体 Service 和 Client[11]。他们之间的关系如下：

（1） 多媒体 Server 作为一个独立进程存在，包含有多个多媒体 Services。

（2） Client 用于与 Server 通信，获取相应的 Service 服务。

多媒体操作主要包括多媒体的录制和播放，其中录制涉及到音视频的源数据获取、编码和混合，播放涉及到音视频的分离、解码和显示。关于 StageFright 更详细的内部架构，如图 2-6 所示。

2.3.2 总体工作流程

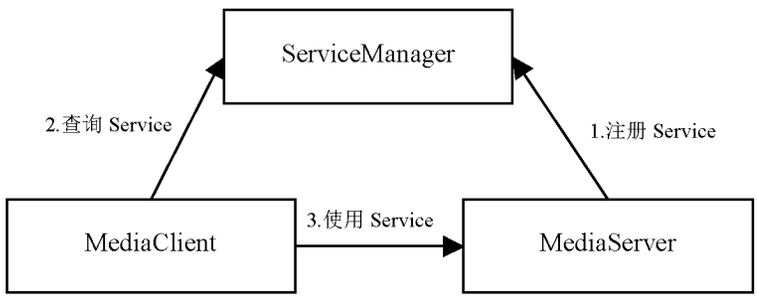

图 2-6 多媒体框架的总体工作流程

如图 2-6 所示，多媒体框架包含了三个部分的交互。这三个部分都是作为独立进程存在，其中 ServiceManager 和 MediaServer 属于系统进程，MediaClient 属于用户进程。在 2.3.1 节中介绍的目录结构，主要是包含了 MediaServer 进程的实现代码和提供给用户开发 MediaClient 的相应接口，ServiceManager 进程作为 Android 所有的 Service 管理者，并不只属于多媒体框架。如果只是为了实现多媒体功能的话，实现 MediaClient 和 MediaServer 就足够了，那么为什么需要用到 ServiceManager 呢？这是因为在 Android 中，ServiceManager 具有非常重要的意义：

（1） Android 中运行着各种各样的 Service，多媒体 Services 只是其中很小的一部分。为了能够统一管理这些服务，必须需要有一个管理者，那就是 ServiceManager。他能够施加权限控制，并不是所有的进程都可以注册服务，也并不是所有用户都能任意使用服务。

（2） ServiceManager 就像是一个 DNS 服务器。ServiceManager 根据接收到的 Service 名称（字符串类型）查找对应的 Service 信息，然后返回给用户。用户以此与 Server 建立连接通道。

（3） 因为系统各种原因的影响，Server 进程可能会生死无常。现在有了一个统一的管理机构，Client 只需要查询 ServiceManager，就能把握动向，得到最新消息。这样也不用为了查询 Server 的状态而给每个 Server 设计一套查询机制，并且如果多个 Client 连接同一个 Server 会导致频繁收到查询请求，无疑增加了 Server 的工作压力。

由此可见，ServiceManager 在多媒体框架下也是必不可少的。这三者的工作流程总

结如下：

（1） MediaServer 进程需要在运行之初，将 Services 注册到 ServiceManager 进程。

（2） MediaClient 进程需要使用某个多媒体 Service 时，首先向 ServiceManager 获取该 Service 的相关信息。

（3） MediaClient 进程根据获取到的 Service 信息与 MediaServer 建立通信的通道，之后就可以直接交互了。

（4） 三者之间的通信都是通过 Binder 机制来实现的。

2.3.3 MediaServer 进程下的 Binder 通信机制

上节提到，MediaClient、MediaServer 和 ServiceManager 两两之间都是通过 Binder 机制来进行通信的。Binder 机制是 Android 底层一种非常重要的进程间通信方式。在了解 Binder 机制的基础上，要学习 Android 多媒体框架或者是其它的框架都会事半功倍。本节为了更好的说明多媒体框架下的 Binder 通信，将以 MediaServer 进程为例进行分析。其入口函数位于 frameworks/base/media/mediaserver/Main_mediaserver.cpp[12]中，

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

从代码中可以看到，MediaServer 进程包括了几个 Service[13]：

（1） AudioFlinger：音频系统中的核心服务。

（2） MediaPlayerService：多媒体框架中的核心服务。这里是本节中研究的重点。

（3） CameraService：与摄像/照相相关的服务。

（4） AudioPolicyService：与音频策略相关的服务。

1. 打开/dev/binder 设备

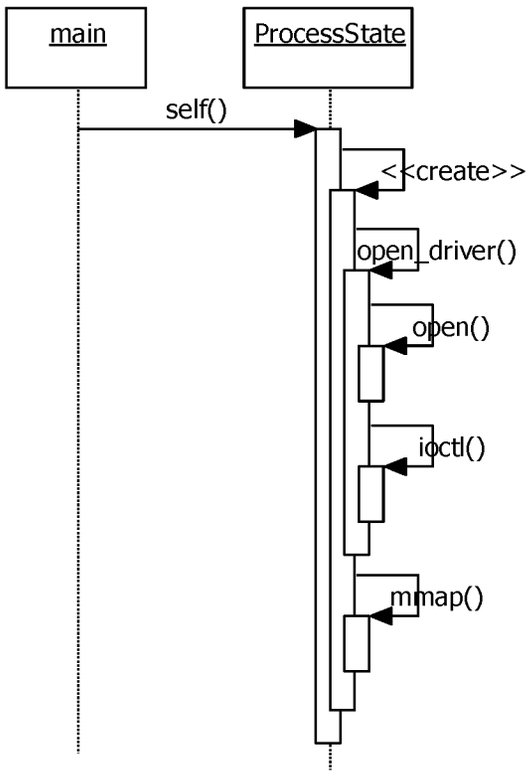

图 2-7 ProcessState 对象创建过程

图 2-7 描述了 MediaServer 进程设置 Binder 的过程。ProcessState 使用了单例模式，在 self 函数中创建对象。在 ProcessState 的构造函数中，会打开/dev/binder 设备，并且会将返回的文件描述符和内存做映射，这样就能够通过内存来处理数据，从而实现了与内核 Binder 驱动交互的通道。

2. 创建 IServiceManager 对象

在获取 IServiceManger 对象之前，先要调用 ProcessState::getContextObject 函数。该函数会根据 Handle 值（int32_t 类型，用于标识不同的 Binder，这里使用了固定值 0）去查找是否已经创建了对应的 BpBinder 对象（ProcessState::lookupHandleLocked 函数），如果没有找到，那么就会创建一个。最终会返回这个 BpBinder 对象。那么 BpBinder 是什么，有什么作用呢？

事实上，BpBinder 和 BBinder 一样，都是 Android 中与 Binder 通信相关的类，而且都是从 IBinder 类派生而来。

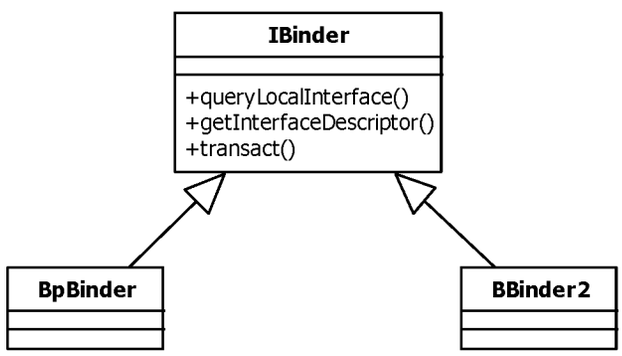

图 2-9 Binder 相关类图

BpBinder 和 BBinder 之间的关系如下：

（1） BpBinder 是客户端用来与服务器交互的代理类。BpBinder 中的 B 和 p 分别是指 Binder 和 Proxy。由于 MediaServer 与 ServiceManager 的交互只是用于注册 Service，就此而言，MediaServer 就相当于一个客户端，ServiceManger 相当于服务器。

（2） BBinder 是与 BpBinder 相对的类，工作于服务器端。BBinder 是 BpBinder 交互的目的端，对于 BpBinder 发送过来的请求，都由 BBinder 来处理，BBinder 实现了具体的业务逻辑。任何一个 BpBinder，都会有一个 BBinder 与之对应，并且是通过前面提到的 Handle 值来对应。Handle 值由/dev/binder 设备来创建和维护，值为 0 代表的就是 ServiceManager 所对应的 BBinder。

（3） BBinder 和 BpBinder 是成对出现，每个 Service 都会实现这两个类的子类。

到目前为止，我们只是获取到了 BpBinder 对象，这个对象就是用来和 ServiceManager 进程通信的，但 MediaServer 进程不会直接持有 BpBinder 对象，而是将其放进 IServiceManager 对象中，通过 IServiceManager 对象来使用。所以接下来，interface_cast<IServiceManager>函数返回的就是 IServiceManager 对象，该函数的参数就是 BpBinder 对象。由图 2-8 可以看到，在函数 interface_cast<IServiceManager>中创建了一个 BpServiceManager 对象，而最终返回的也就是这个对象，那么它和 IServiceManager 是什么关系呢？而他头两个字符是 Bp，是否和 BpBinder 也有关系？

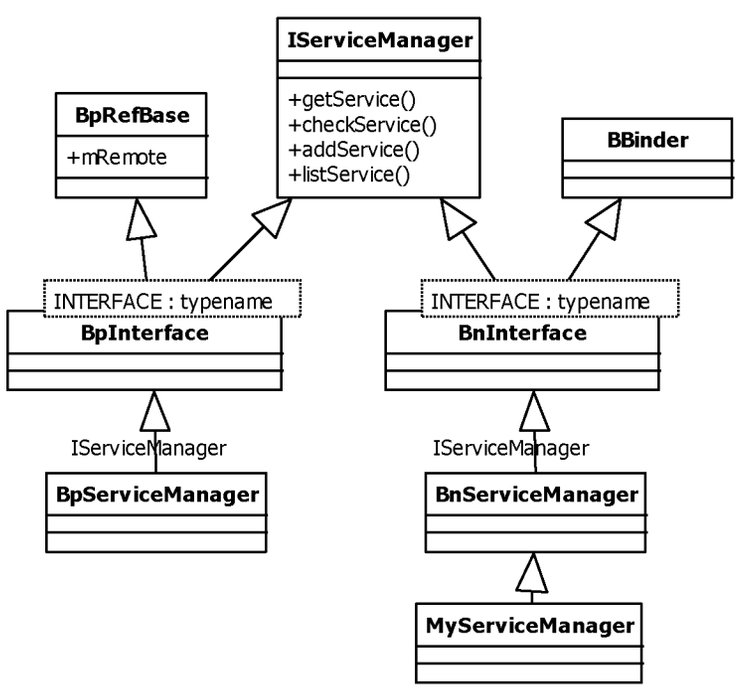

图 2-10 IServiceManager 相关类图

从图 2-10 中可以得出以下几个结论：

（1） BpServiceManager 虽然不是继承 BpBinder，但父类中的成员变量 mRemote 实际指向的是 BpBinder 对象。因此 BpServiceManager 同样具有了 Proxy 的功能，用于客户端。

（2） BnServiceManager 继承了 BBinder，因此很明显地，它是工作在服务器端，也就是 ServiceManager 进程。BnServiceManager 的头两个字符 Bn 是指 Binder 和 Native 的意思。又因为 BnServiceManager 是虚类，所以其业务需要子类 BServiceManager 来实现。

通过上述的分析，MediaServer 进程获取到的 IServiceManager 对象就是 BpServiceManager 对象，其目的是用来和 ServiceManager 进程通信，接下来就是注册 Service 的过程了。

3. 注册 MediaPlayerServie

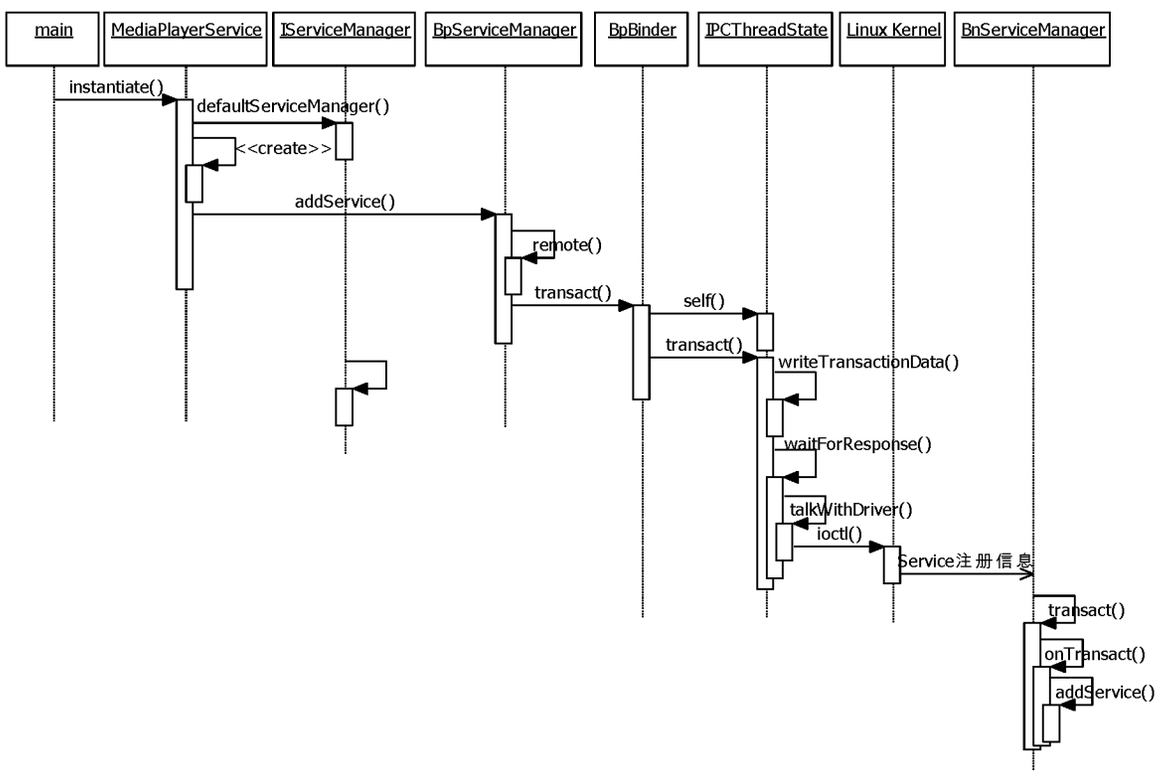

图 2-11 MediaPlayerService 注册流程

图中，通过调用 defaultServiceManager 函数实际返回了 BpServiceManager 对象。再由该对象通过调用 addService 函数将 MediaPlayerService 注册到 ServiceManager 进程中。addService 函数的实现如下所示：

```
    virtual status_t addService(const String16& name, const sp<IBinder>&
service)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor())
;
        data.writeString16(name);
        data.writeStrongBinder(service);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
&reply);
        return err == NO_ERROR ? reply.readInt32() : err;
    }
```

函数参数 name 代表 Service 的名称，所有的 Client 进程都是通过 Service 名称来查询、获取相应的 Service 的，这里的值为“media.player”。参数 service 代表要注册的 Service，这里是指 MediaPlayerService 对象。从代码中可以看出，Parcel 类是用于数据的打包，remote 函数返回的实际上是在之前创建的 BpBinder 对象，通过 transact 函数实现了 Binder 通信。

BpBinder::transact 函数内部其实是通过 IPCThreadState 类真正地实现通信。IPCThreadState::writeTransactionData 函数会将上述经过 Parcel 封装后的数据和 Handle 等信息进行 binder_transaction_data 封装，接着结合 cmd 命令（这里值为 BC_TRANSACTION）再次进行 Parcel 封装。binder_transaction_data 是和 BC_TRANSACTION 命令相对应的数据结构，ServiceManager 服务器进程根据 BR_TRANSACTION 命令就能知道是用 binder_transaction_data 结构体来承接数据（发送端向接收端发送 BC_XXX 命令时，由于 binder 设备处理机制，接收端接收到的是 BR_XXX 命令，关于 cmd 命令的定义及其与数据结构的对应关系可以查看 kernel/common/linux/binder.h）。最后将 Parcel 封装的数据放到 binder_write_read 对象中，利用 ioctl 函数将其发送到 ServiceManager 服务器进程。binder_write_read 是负责与 /dev/binder 设备交互的数据结构。关于数据的打包过程，从图 2-12 中可以更清晰地说明。

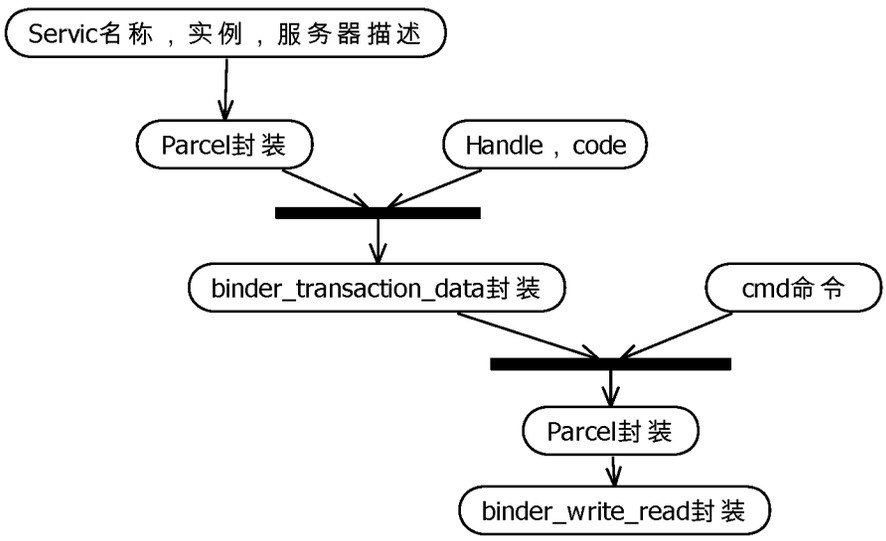

图 2-12 数据打包过程

4. 等待客户端请求

通过以上的过程，MediaServer 进程已经完成了注册 Service 的过程，但自身作为服务器，同样需要接收和处理 Client 发送过来的请求。这一过程在 IPCThreadState::joinThreadPool()函数中，具体如图 2-13 所示。

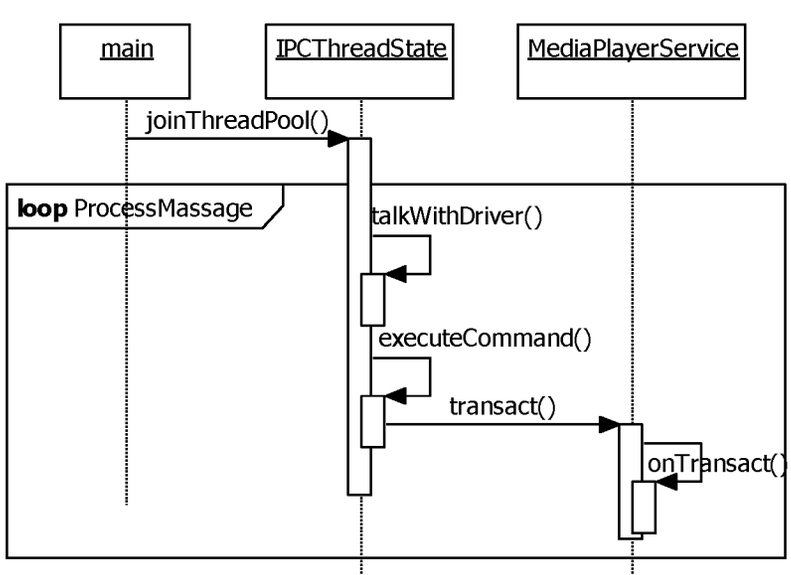

图 2-13 等待请求流程

其中 IPCThreadState::executeCommand 函数实现具体的 MediaServer 业务逻辑。

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    ……
    case BR_TRANSACTION:
        ……
        sp<BBinder> b((BBinder*)tr.cookie);
        const status_t error = b->transact(tr.code, buffer, &reply, 0);
        if (error < NO_ERROR) reply.setError(error);
    ……
}
```

在代码中，获取到了一个 BBinder 对象 b，这是/dev/binder 设备根据客户端 BpBinder 对象查找到相应 Service 的 BBinder 对象。

在介绍 BpBinder 和 BBinder 之间关系的时候，就已经知道了 BBinder 是具体业务逻辑的实现类。那么其具体是在哪里实现的呢？由上述代码可以看到，是调用了 BBinder::transact 函数，再结合图 2-11 的函数调用流程可以发现，最终的业务处理逻辑在 BnServiceManager::onTransact 函数中。

由此，我们也可以总结出 IBinder::transact 函数的重要意义：对于 BpBinder 家族类，transact 函数是用于实现与服务器的通信；对于 BBinder 家族类，transact 函数是用于实现具体的业务逻辑。

5. 总结

通过上述分析，可以将多媒体的 Binder 通信过程总结为以下几个步骤：

（1） 打开/dev/binder 设备。

（2） 获取与远端进程 BBinder 对象相对应的 BpBinder 对象。

（3） 通过 BpBinder 对象，最终利用系统函数 ioctl 进行数据收发。

其中获取 BpBinder 对象是 Binder 通信的关键。对于与 ServiceManager 进程通信，可以直接创建 BpBinder 对象（Handle 值为 0，因为这是/dev/binder 设备为 ServiceManager 配置的默认值）；对于其他 Service 的 BpBinder 对象，是通过将 Service 名称作为函数参数，调用 BpServiceManager::getService 函数来获取。该函数内部会和 ServiceManager 进程通信，获取到相关的 Service 信息，再由/dev/binder 设备驱动根据 Service 信息创建对应的 BpBinder 对象，然后返回。

/dev/binder 设备内部的工作原理为：

（1） 服务器进程向/dev/binder 设备传入 BBinder 子类对象，/dev/binder 设备会为其分配一个 Handle 值，并以此值为参数创建 BpBinder 对象，返回给客户端进程，从而将 BBinder 和 BpBinder 对应起来。/dev/binder 设备内部会维持 Handle 值和 BBinder 子类对象的对应关系。

（2） 客户端进程向/dev/binder 设备传入 BpBinder 对象，/dev/binder 设备会通过其中的 Handle 值查找相应的 BBinder 子类对象，然后返回给服务器进程。

通过 starUML 软件的反向工程功能，得到了 MeidaServer 的整体类图，如图 2-14 和图 2-15 所示。

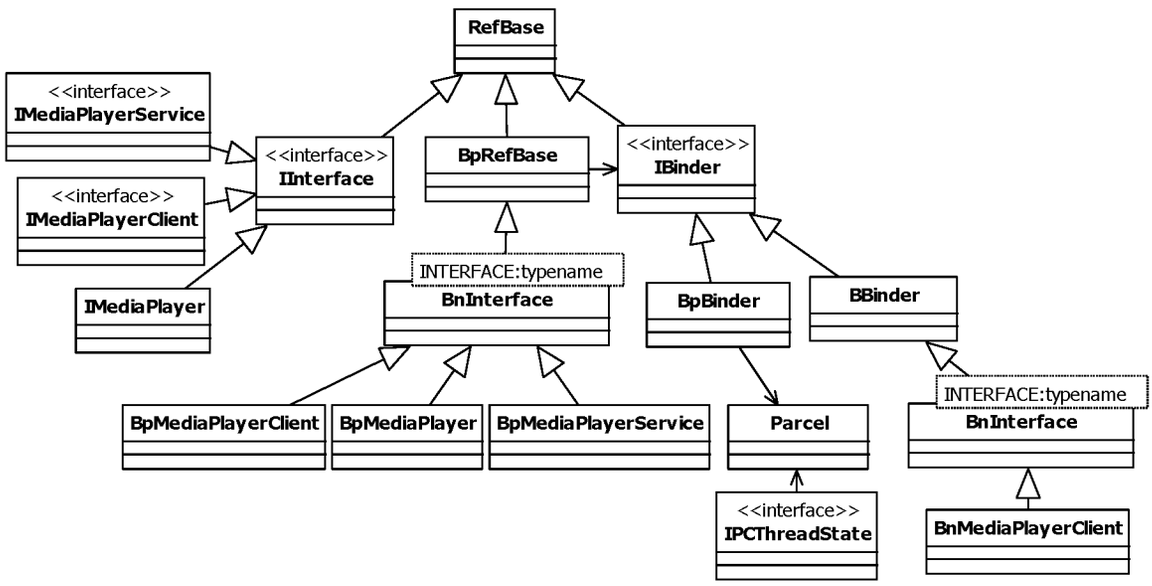

图 2-14 Client 进程使用的相关类图

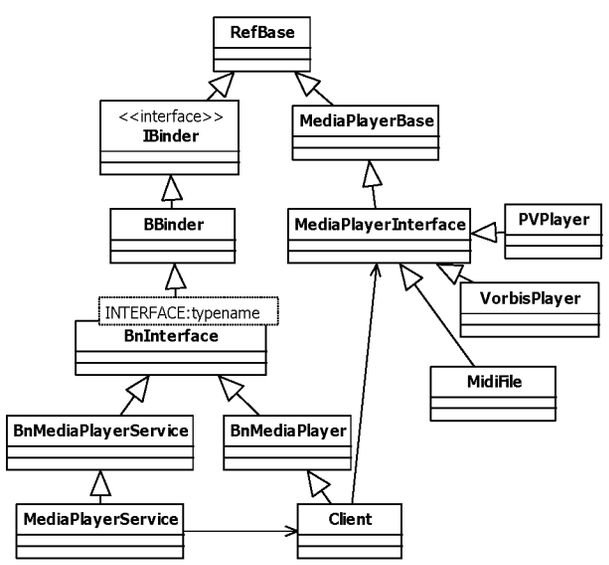

图 2-15 MediaServer 进程使用的相关类图

图中发现，MediaPlayer 类继承 BnMediaPlayerClient，而 BnMediaPlayerClient 最终继承了 BBinder，因此它也和 Service 一样向外提供了业务处理。但在下节中会讲到 MediaPlayer 是工作于 Client 进程，也就是说 MediaPlayer 为其它进程与 Client 进程通信提供了途径，其它进程可以通过 BpMediaPlayerClient 来和 MediaPlayer 通信。

2.3.4 本地视频的播放原理分析

在上节，我们已经介绍了与服务器通信是通过 Binder 机制实现的，接下来以本地视频播放为例，分析视频播放的底层实现。这里先给出总体的视频播放交互图[14][15]。

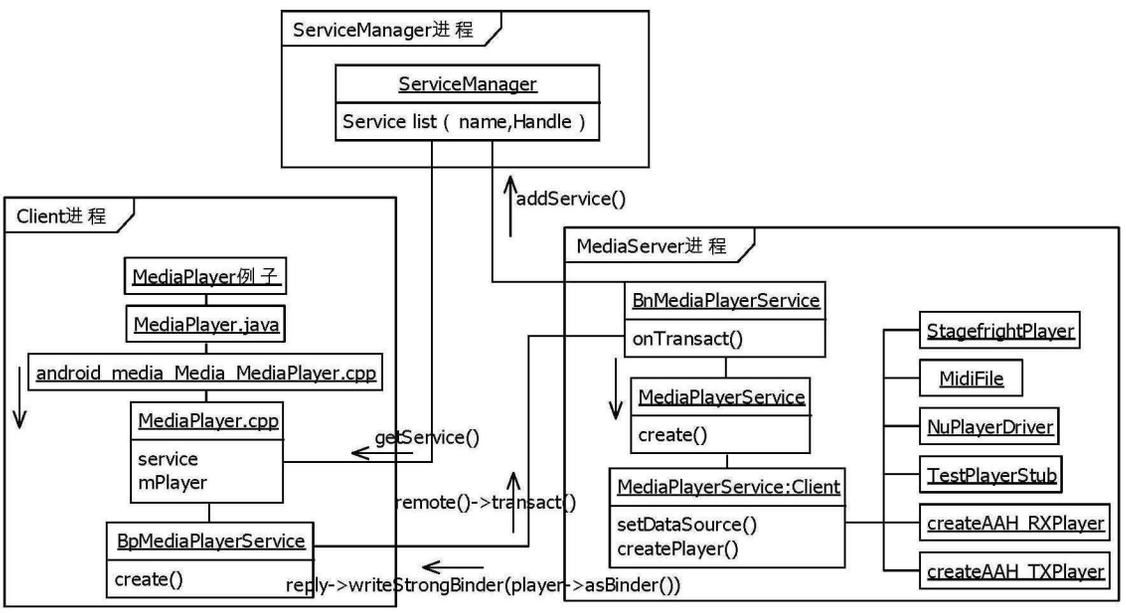

图 2-16 视频播放总体交互图

首先从 Client 进程的 Java 层接口说起，使用 MediaPlayer 进行播放的主要步骤是：

```
MediaPlayer mp = new MediaPlayer();
mp.setDataSource(filename);
mp.prepare();
mp.start();
```

1. setDataSource 底层实现

（1） Client 进程的 setDataSource 流程

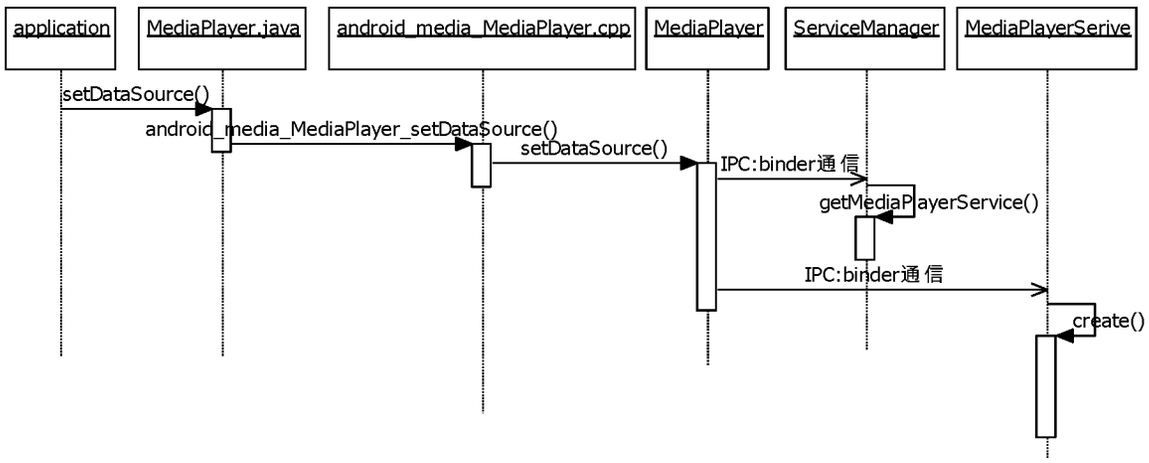

图 2-17 Client 的 setDataSource 流程

对 Client 进程 setDataSource 流程的一些解析：

1） 应用层调用的 setDataSource 实际上是调用了 MediaPlayer 类中的 setDataSource 函数。

2） getMediaPlayerService 函数内部会通过 binder 与 ServiceManager 进程通信，获取并返回 BpMediaPlayerService 对象。

（2） MediaServer 进程的工作流程

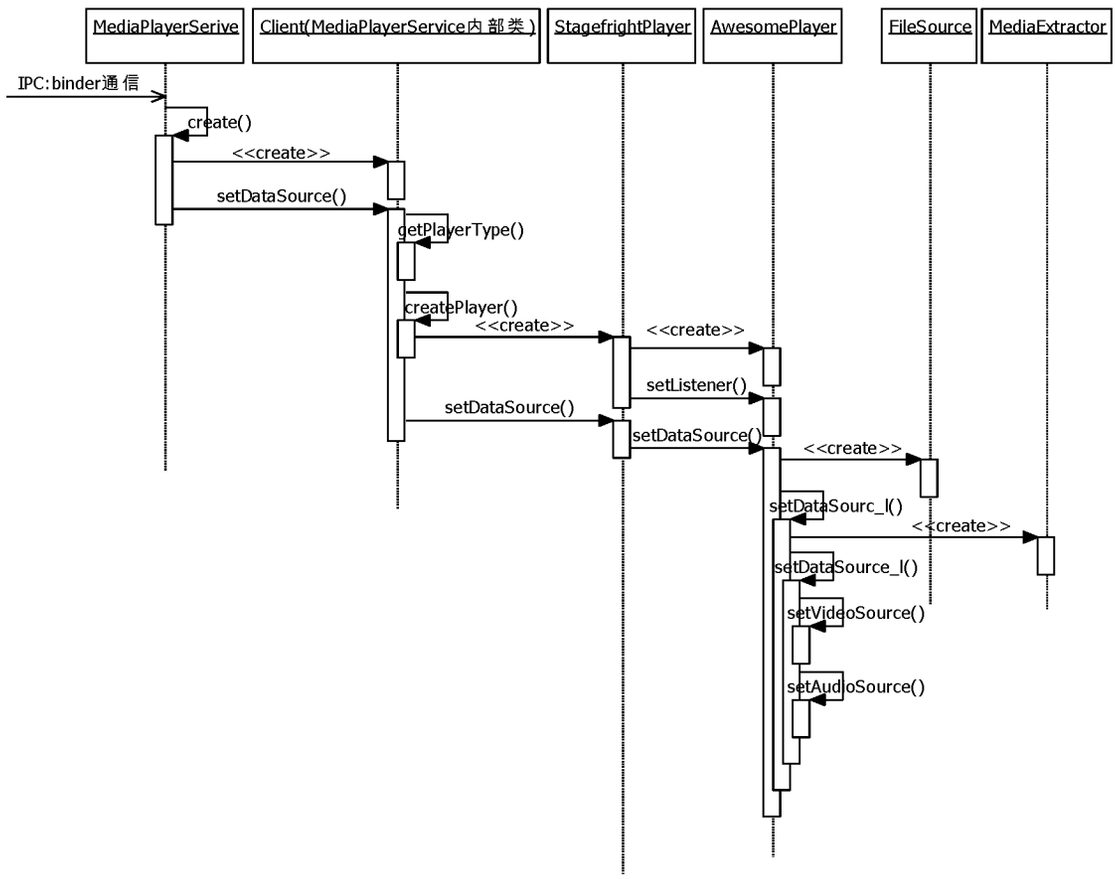

图 2-18 Server 的 setDataSource 流程

对 MediaServer 进程 setDataSource 流程的一些解析：

1) 图 2-18 假设使用 StagefrightPlayer 播放器，所以只显示了 StagefrightPlayer 的创建和设置流程。但其他的播放器的创建流程和此类似。

2) 调用 setDataSource 函数的目的一是为了创建播放器，二是为了得到能够分析视频文件的 MediaExtractor 对象，并从中获取到 VideoTrack 和 AudioTrack。需要特别注意的是，如果播放的是网络视频（如 http://开头的 url），那么 setDataSource 函数中并不会获取 MediaExtractor 对象，而是将其放到了 AwesomePlayer:: onPrepareAsyncEvent 函数中，这样做是为了避免和资源服务器交互所造成的阻塞。

3) MediaPlayerService::Client 类继承了 BnMediaPlayer，再结合图 2-14 和图 2-15 分析可以发现，BnMediaPlayer 和 BpMediaPlayer 的关系以及通信过程与 BnServiceManager 和 BpServiceManager 是相同的，所以显然 MediaPlayerService::Client 类可以让 Client 进程通过 BpMediaPlayer 来与之通信。而事实上，在 setDataSource 函数运行完成时，MediaServer 会将 BpMediaPlayer 对象返回给 Client 进程，之后 Client 进程就使用

BpMediaPlayer 对象来与 MediaServer 进程上的 MediaPlayerService::Client 通信了。

2. prepare 的底层实现

（1） MediaServer 进程的工作流程

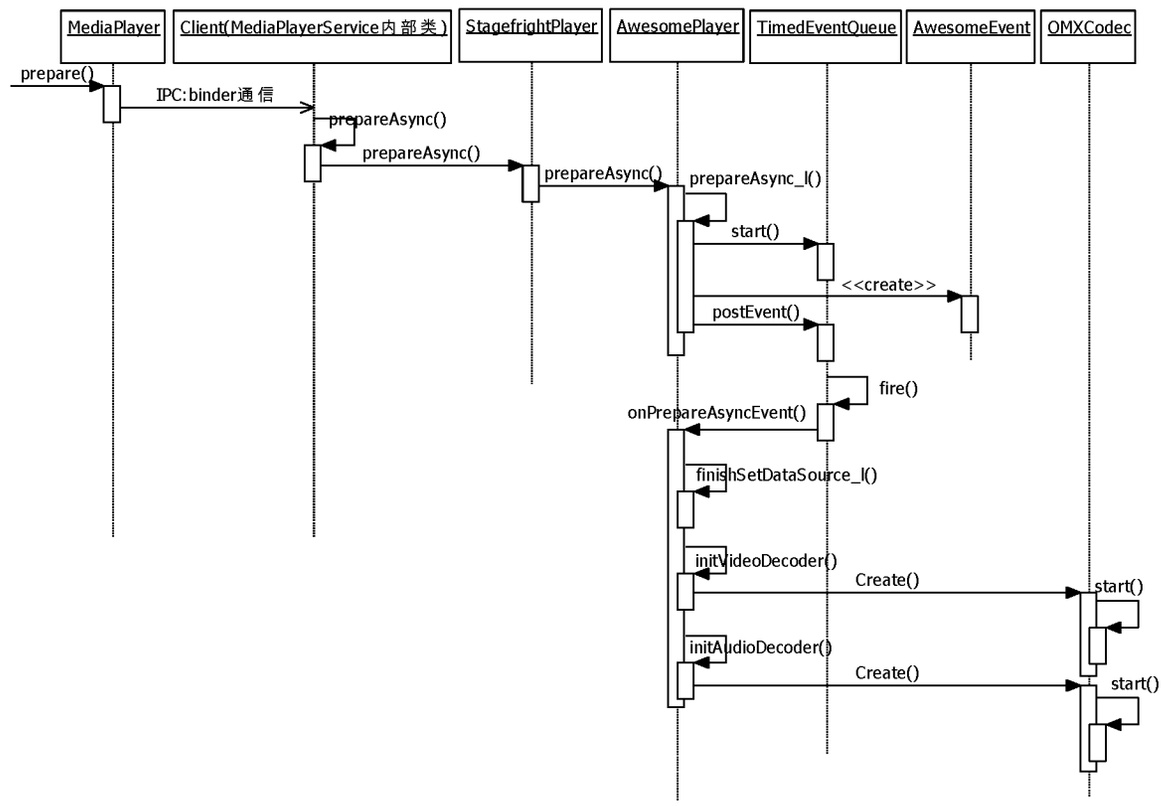

图 2-19 prepare 流程

由于 Client 进程只是将业务请求发送给 MediaPlayer 进程，所以这里不再叙述 Client 的工作流程，而是重点分析 MediaPlayer 上的业务处理流程。对 prepare 流程的一些解析：

1） TimedEventQueue::start 函数会创建一个基于时间调度的事件处理线程，主要用于异步的事件处理。

2） 在 prepare 函数调用的最后，会 post 一个 AsyncPrepare 的事件。事件调度线程接收到该事件后，在到达指定的时间执行 AwesomePlayer::onPrepareAsyncEvent 函数，该函数内部主要是完成 setDataSource 未完成的工作（如创建 http:// 资源的 MediaExtractor）和创建音视频解码器（实际是获得 VideoSource 和 AudioSource，并开启解码器）。

3. start 的底层实现

（1） MediaServer 进程的工作流程

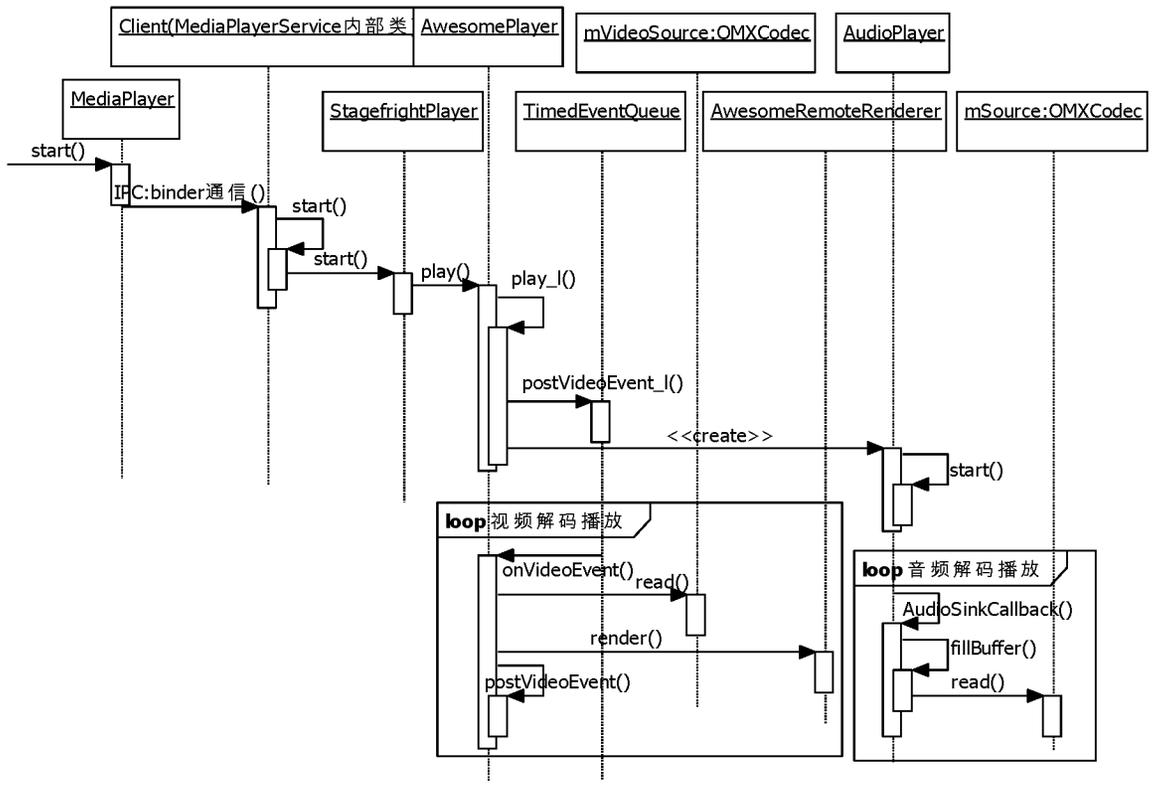

图 2-20 start 流程

对 start 流程的一些解析：

1) 视频解码播放是使用 VideoEvent 事件来驱动的，音频播放是通过 AudioPlayer 自身的 AudioSinkCallback 来驱动。

2) OMXCodec::read 函数执行数据的解码操作，并返回解码后的数据。Render 函数用于显示视频帧。

3) AwesomePlayer 根据音频的播放进度来确定视频的播放延迟时间。然后通过 postVideoEvent_l 函数告知事件调度线程，在这个延时之后执行 onVideoEvent 函数，再次处理视频帧。

2.4 OpenMAX 多媒体引擎

2.4.1 OpenMAX 综述

OpenMAX（全称：Open Media Acceleration， 开放多媒体加速层）是一个免费并且跨平台的多媒体应用标准。在遵循 OpenMAX 标准的操作系统和硬件平台下，能够快速、

轻松地在实现各种流媒体编解码器和应用的开发、整合和编程。同时，使得该库和编解码器的实现者能够加速有效地利用新硬件的加速功能，而无需关心底层的硬件架构[16]。

OpenMAX 被分为三个层次，自上而下分别为：OpenMAX AL（应用层），OpenMAX IL（整合层）和 OpenMAX DL（开发层）。然而在实际应用中，OpenMAX IL 被广泛地使用。而 OpenMAX AL 和 OpenMAX DL 由于操作系统和底层硬件的差异很大，所以很少被使用。在 Android 多媒体框架中也只是使用了 OpenMAX IL 标准，所以后面主要是对 OpenMAX IL 进行研究和分析[17]。

2.4.2 OpenMAX IL 介绍

OpenMAX IL 作为一个较 OpenMAX AL 更底层的接口，主要为嵌入式或者移动设备提供音频、视频和图像的编解码器。它以统一的方式使得应用程序和媒体框架能够接合这些多媒体编解码器，并支持各种组件（如 sources 和 sinks）。

多媒体编解码器本身可以是任意软件和硬件的结合，对于用户是完全透明的。如果没有这样一个标准化的接口，那么编解码器厂商就必须实现专有的或者封闭的接口集成到移动设备中。而一旦如此，要想在不同媒体架构的系统实现相同的功能应用，将会是一件非常繁杂的事情。OpenMAX IL 就是为了解决这个问题而被提出的。它将编解码器进行抽象，形成一个标准的接口，只要编解码器厂商按照该标准实现编解码功能，那么用户就可以利用这套接口方便地进行上层开发。

OpenMAX IL 标准只提供了数据处理接口及其相互之间的关系，定义了音频、视频和图像的处理过程，但具体的实现是由使用它的厂商确定。OpenMAX IL 接口由 10 个头文件组成，关于这些头文件的功能总结为[18]：

（1） OMX_Core.h

涵盖了 OpenMAX Core 接口函数，包括 Component 的查找，参数设置等功能。

（2） OMX_Types.h

定义了 OpenMAX IL 使用到的相关数据类型。

（3） OMX_Component.h

定义了操作 Component 的相关接口。

（4） OMX_Audio.h

定义了与音频相关的数据结构。

（5） OMX_Video.h

定义了与视频相关的数据结构。

（6） OMX_Image.h

定义了与图像相关的数据结构。

（7） OMX_IVCommon.h

定义了图像和视频共同使用的数据结构。

（8） OMX_Other.h

其它数据结构，如音视频同步用的结构。

（9） OMX_Index.h

定义 OpenMAX IL 所有数据结构的索引值。

2.4.3 OpenMAX IL 的系统位置

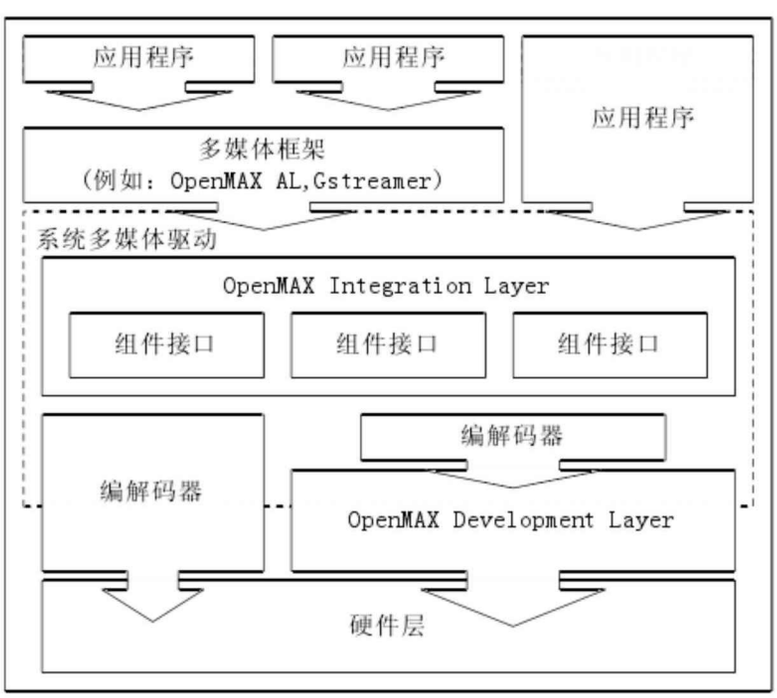

图 2-21 OpenMAX IL 系统位置

如图 2-21 所示，OpenMAX IL 为上层提供了统一的接口，IL client 可以是多媒体框架，也可以直接是顶层应用。一般而言，在系统开发中，经常都会在顶层应用和 OpenMAX IL 之间增加一层多媒体框架。比如在 Android 中，就使用了 OpenCore 和

StageFright（android2.0 之后 StageFright 代替了 OpenCore）多媒体框架。OpenMAX IL 下层对编解码器进行了封装，使得编码器开发者既可以以传统的方式调用底层驱动直接封装编解码器，也可以通过 OpenMAX DL 封装。

2.4.4 OpenMAX IL 工作过程

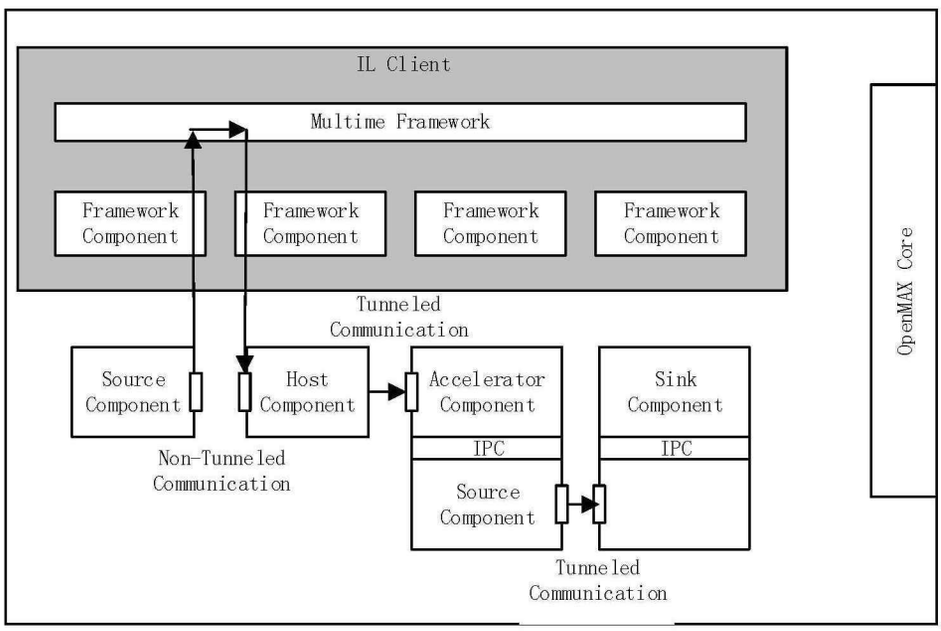

图 2-22 OpenMAX IL 工作过程

图 2-22 显示了 OpenMAX IL 完成一个功能所经历的流程。涉及到的功能模块主要包括：

（1） IL Clint（集成层客户端）

调用 OpenMAX Core 或者 OpenMAX Component 接口的软件层，例如 GStreamer。IL Client 可以在 GUI 层的一层或几层之下。

（2） OpenMAX Core（OpenMAX 核心）

与平台相关的代码。主要是指示 OpenMAX Component 在主存中的位置，或者将 OpenMAX Component 载入到主存中。当不再需要使用 OpenMAX Component 时，将其从主存中卸载。一般情况下，Core 将 Component 载入到内存后，不再参与 Application 和 Component 间的通信。

（3） OpenMAX Component（OpenMAX 组件）

是用于封装目标系统所需功能的其中一个独立部分。OpenMAX 将这种组件封装为

一个标准的接口。组件包括有：sources（输入组件），sinks（输出组件），codecs（编解码器组件），filters（过滤器组件），splitters（分离器组件），mixers（混合器组件）和一些其他的 data operator（数据运算组件）。一个组件通过一系列相关的数据结构、枚举类型和接口来设置或者查询参数。这些参数包括组件运作参数（如编解码器组件的参数配置）和组件的实际运行状态。

（4） Port（端口）

OpenMAX Component 的数据输入输出接口。

（5） Tunnels/Tunneling（隧道）

两个 OpenMAX Component 之间通过 Port 建立起的连接通道。用于管理两个组件间的数据建立和通信。

其中 IL Clint 位于 OpenMAX IL 之上，主要是调用 OpenMAX IL 的接口实现相应功能，其他功能模块属于 OpenMAX IL 范畴。

如图 2-22 所示，IL Client 通过 OpenMAX Core 加载了 Source，Host，Accelerator 和 Sink 四个组件。Source 组件提供数据的来源，有一个输出端口。Host 组件有一个输入端口和一个输出端口。Accelerator 组件有一个输入端口，调用了硬件编解码器，并通过 IPC（进程间通信)实现了与 Sink 组件的数据交互，加速体现在这个环节上。Sink 组件提供了数据的去处，有一个输入端口。由此可见，在 OpenMAX IL 中，组件处理的核心内容是数据的填充和消耗。

OpenMAX IL 允许两种不同的数据流处理方式，即经过 IL Client 和不经过 IL Client。图 2-3 中，从 sourc 组件获得的数据上传给了 IL Client，再由 IL Client 下传到 Host 组件，而 Host 组件和 Accelerator 组件之间不经过 IL Client，而是之间通过隧道进行数据通信。Accelerator 组件和 Sink 组件之间数据交互是通过 IPC 进行，也没有流经 IL Client。

2.4.4.1 OpenMAX Core 工作机制

1. Core 的初始化过程

OpenMAX Core 首先调用 OMX_Init()函数来获取已经注册的 Component 列表。这些 Component 信息存储在一个数组中。OpenMAX IL 开发者可以在这个数组中方便地加入自己设计的 Component。因为 Component 是根据名称来装载，因此在系统注册时，必须确保其名称的唯一性。

2. 装载 Component

初始化完成之后，OpenMAX Core 开始动态地装载 Component。具体由 OMX_GetHandle 函数完成。该函数按照 IL Client 提供的名称来查找 Component。如果找到，则将其加载进内存，并设置 Component 的运行参数和回调函数等信息。如果没找到，则返回失败。当 OMX_GetHandle 函数执行成功后，会返回一个 Component 实例之后 IL Client 就能够直接对 Component 进行操作了。

3. 设置 Component 之间的 Tunneled 连接方式

Component 之间的数据交换有两种方式。一种是非 Tunneled 方式，一种是 Tunneled 的方式。工作于非 Tunneled 方式下的 Component 需要 IL Client 提供数据；工作于 Tunneled 方式下的 Component 由前一个与它相连的 Component 提供数据。Tunneled 连接方式需要额外的函数完成，该函数为 OMX_SetupTunnel。具体处理过程如下：

1) 判断两个需要 Tunneled 的 Component 端口是否属于同一个 Domain 内（指是否都为 Audio 或 Video 等）和数据的格式是否相同。
2) 协商由哪个 Component 端口作为 Buffer 的提供者。

2.4.4.2 OpenMAX Component

1. Component 架构

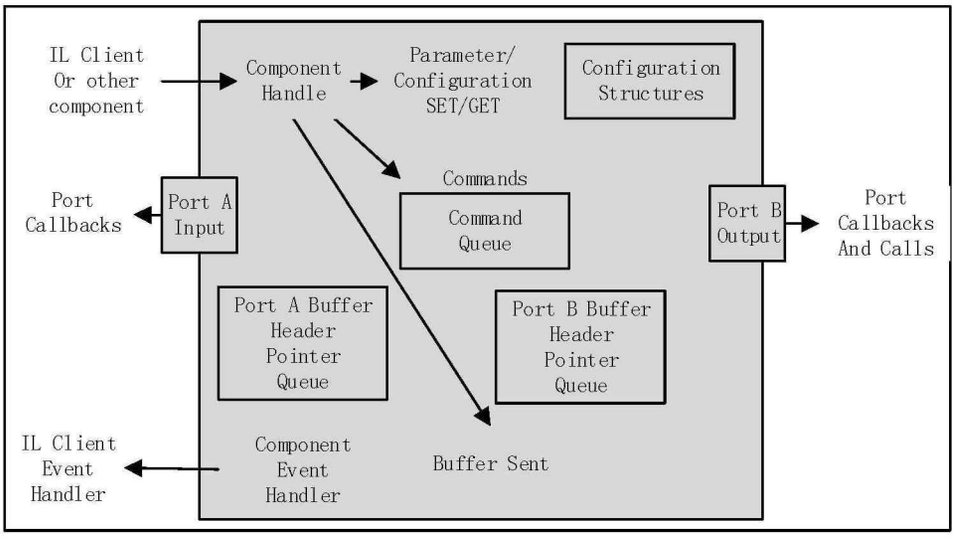

图 2-23 Component 架构

图 2-23 显示了一个 Component 的内部结构，它有一个入口为 Handle 指针，用户通过该 Handle 能够调用一系列标准函数，包括设置或查询参数，发送命令和数据缓冲区。

组件的运行参数存储于结构体中，命令和数据缓冲区分别存储于内部的命令队列和缓冲区队列中。Component 根据端口个数的多少而可以有多种可能的输出调用，每个端口都有外部调用或者回调函数与之对应起来。当组件内部有事件发生时，也会反映给指定的 IL Client event handler[19]。

2. Component 状态

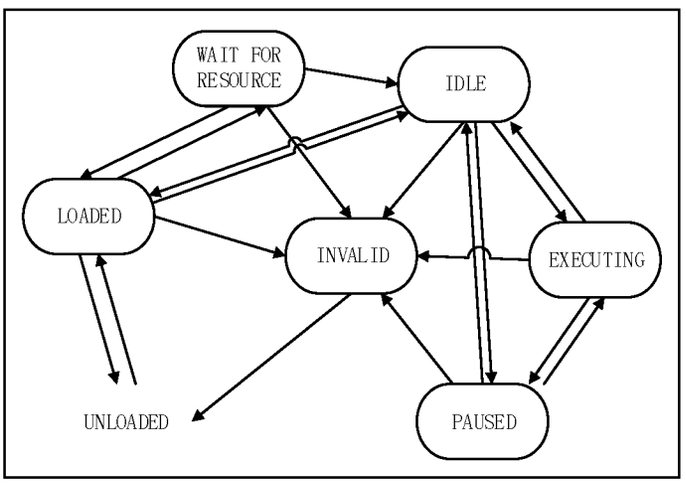

图 2-24 Component 状态

OpenMAX 组件的工作过程被划分成了不同的状态。在最初时为 unloaded 状态。通过 OpenMAX Core 将组件载入内存后变为 loaded 状态，之后随着组件的运作而在不同的状态之间切换。当遇到无效数据时，组件就会进入 invalid 状态，比如回调函数指针设置不正确。IL Client 在检测到组件的状态为 invalid 时，应该停止，去初始化并卸载组件，然后才重新装载。如图 2-24 所示，除了在 unloaded 之外的其他状态下都有可能进入 invalid 状态，此时需要将 invalid 状态切换回 unloaded 状态[20]。

在切换到 idle 状态时，有可能会失败，因为 idle 状态需要对所有的业务资源进行分配。当状态由 loaded 切换到 idle 失败时，IL Client 应该重试或者先将状态由 loaded 切换到 wait for resources。进入 wait for resources 状态后，一旦资源变为可用，那么组件就会向资源管理器注册来发出警告，资源管理器随后将组件切换到 idle 状态。IL Client 通过发送命令来控制所有状态的切换，但不能控制 invalid 状态。

Idle 状态只是表明组件需要的所有资源已经准备好了，而会开始处理数据。executing 状态表明组件正在接收和处理数据，并准备执行所需的回调函数。paused 状态下将停止数据的处理和缓冲区的交换，保留了缓冲区的上下文信息。从 paused 切换到 executing 状态会恢复组件的运行。而从 executing 或 paused 切换到 idle 状态会使运行的上下文信

息丢失，必须从流的起端重新开始。idle 状态切换到 loaded 状态会使得像通信缓冲区这样的资源丢失。

2.5 OpenMAX IL 在 Android 多媒体框架中的应用

在本章开始时，也提到了 Android 多媒体框架对的 OpenMAX 标准的支持。更确切地说，Android 多媒体框架主要是使用了 OpenMAX IL 中的 Codecs（编解码器）组件。Android 系统将硬件实现的 OpenMAX IL 库命名为 libstagefrighthw.so，该库实现了第二章所讲述的相关接口。不同硬件厂商都会有自己的 libstagefrighthw.so。本节将会详细分析 Android 是如何使用这些 Codecs 组件接口的。这里先给出与 OMX 相关的类图，从而对这些类关系有个总体的认识。

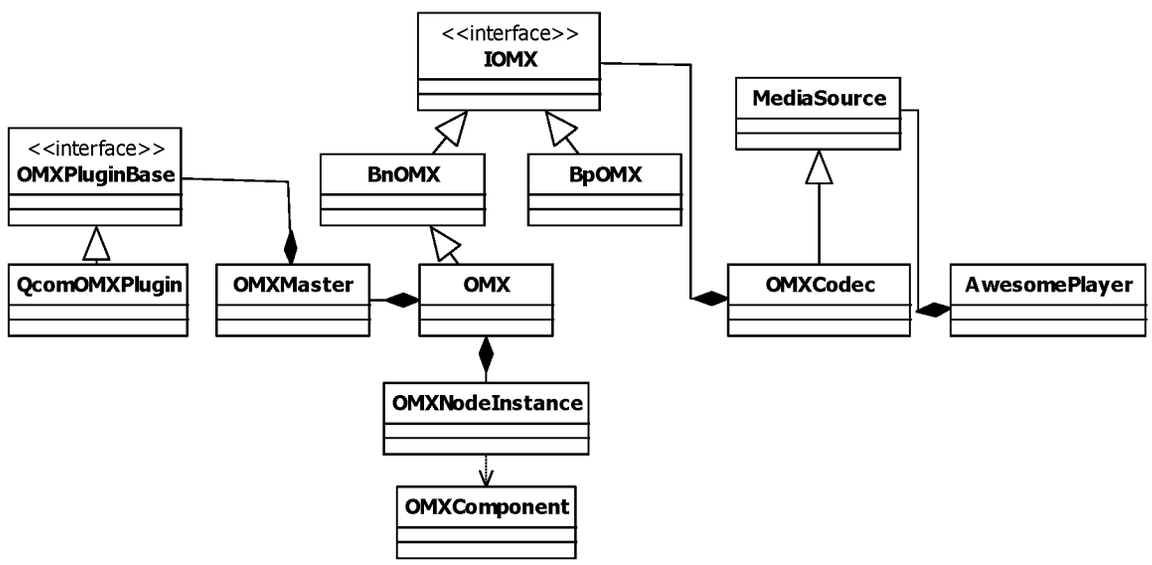

图 2-25 OMX 相关类图

从图 2-25 中，我们发现了在上一节分析视频播放原理时出现过的 OMXCodec 和 AwesomePlayer 类[21]。没错，OMXCodec 类就是 Android 为了调用 OpenMAX IL 而设计的一个封装类。AwesomePlayer 类就是其中的一个调用者。另外，还可以从图中看出，OMX 类是 OpenMAX IL 实现的核心类，由它来管理和完成所有的 OpenMAX IL 功能。OMX 类继承了 BBinder，也就是说 OpenMAX IL 在 Android 平台中是以独立进程存在并工作的，通过 BpOMX 来远程调用。这里将其定义为 OMX 进程。

接下来以 MP4 视频播放中的视频解码为例，详细分析 OMX 的工作过程。

2.5.1 初始化视频解码器

视频解码器的初始化过程，是在 prepare 流程的 AwesomePlayer::initVideoDecoder 函数中，如图 2-19 所示。下面给出该函数更详细的内部实现序列图：

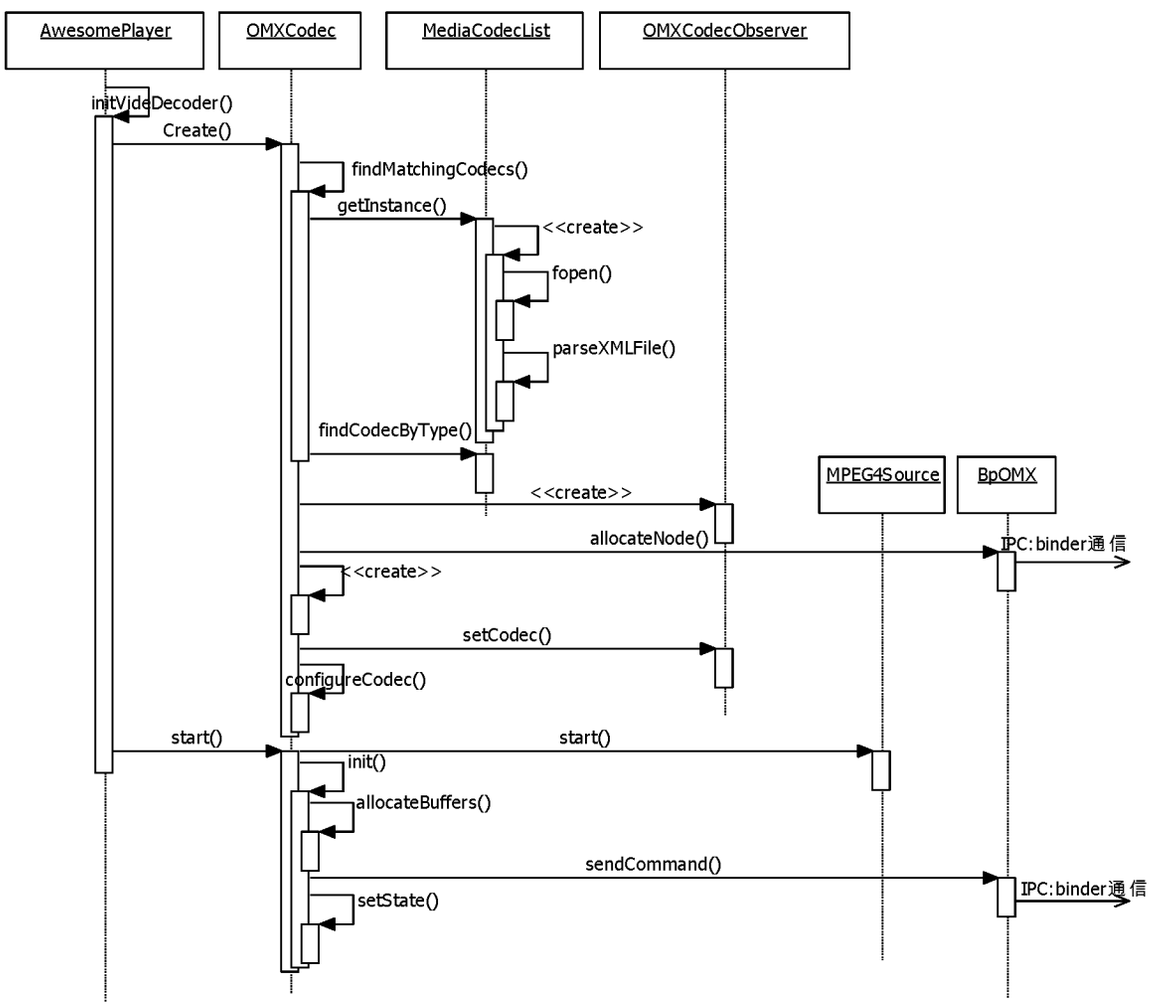

图 2-26 MediaServer 进程的视频解码初始化流程

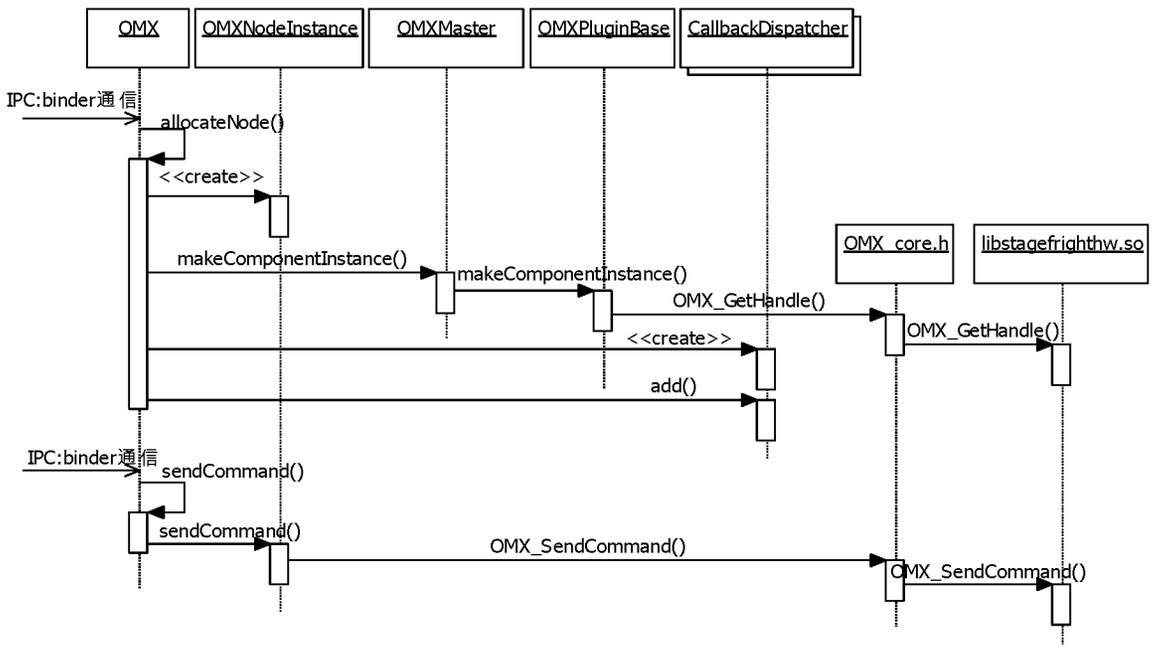

图 2-27 OMX 进程的视频解码初始化流程

关于初始化视频解码器流程的解析：

（1） 图中 BpOMX 及其左侧的部分工作于 MediaPlayerServie（MediaServer 进程中）。BnOMX 及其右侧的部分工作于 OMX 进程中。两者之间通过 Binder 机制通信。

（2） MediaCodecList::findMatchingCodecs 函数主要是根据 mime 来查找合适的 Codecs，返回这些 Codecs 的名称。查找过程在 Android4.1 之后做了重大修改：将编码器信息放到了“/etc/media_codecs.xml”中，通过分析该 XML 文件从而获取到所有的 Codecs 信息列表，然后根据 mime 筛选出合适的 Codecs。上图给出的就是修改后的查找流程。而在 Android4.1 之前，Codecs 信息直接是定义在 OMXCodec::kEncoderInfo 和 OMXCodec::kDecoderInfor 变量中。将 Codecs 信息放到文件中的优点是可以在不改代码的前提下，方便地增加 Codecs。

（3） 从 MediaCodecList::findMatchingCodecs 函数返回一个 Codec 名称列表，也就是说合适的 Codec 可能不止一个（因为可能有很多相同 mime 类型的编码器）。这时根据列表的排序，选取第一个能够使用的 Codec。因此“/etc/media_codecs.xml”中 Codecs 的排序会影响到最终 Codec 的使用。

（4） 在创建 OMXMaster 对象的时候，会将支持的插件通过 OMXMaster::addPlugin 函数加载进来，由 OMXMaster 管理（例如 libstagefrighthw.so）。上图在执行 OMXMaster::makeComponentInstance 函数时，首先根据先前获取到的 Codec 名称来查找相应的插件，得到相应的 OMXPluginBase 对象，然后再执行 OMXPluginBase::makeComponentInstance 函数创建 Component 实例（这里的 Component 和上节中所说的 Component 意思相同，Component 实例化实际上是调用了 OpenMAX IL 标准中的 OMX_GetHandle 函数）。OMXPluginBase 类是 C++接口类，任何要放到 OMX 中使用的插件都需要继承并实现该接口。所以这里的 OMXPluginBase 对象实际上是指向其子类。Plugin 和 Component 的关系：任何的 Component 都是以 Plugin 的形式提供，一个 Plugin 可以包含多个 Components。

（5） 在创建 Component 实例时，需要注册该 Component 的回调函数，其定义为：OMX_CALLBACKTYPE OMXNodeInstance::kCallbacks = {&OnEvent, &OnEmptyBufferDone, &OnFillBufferDone}。OMX 在 CallbackDispatcher 线程中通过 BpOMXObserver 将消息传回给 OMXCodec 中的 OMXCodecObserver，再由 OMXCodec 最终处理回调。关于回调函数的工作流程会在下节中介绍。

（6） OMX 类中有一个主 CallbackDispatcher 线程，通过持有该线程对象而可与之通信。另外，针对每个 OMXNodeInstance 实例都会创建一个副 CallbackDispatcher 线程，并由主 CallbackDispatcher 线程调配。副 CallbackDispatcher 线程主要是为了完成回调任务。

（7） OMXCodec:: configureCodec 函数主要是设置 Codec 的相关参数，比如输出的宽高等。这些参数由 OMXCodec::Create 函数传入。

（8） OMXCodec::start 主要是做了四件事：开启 MPEG4Source（setDataSource 流程中获取到的 VideoTrack），实际是创建其输入的最大缓冲区；创建 VideoSource 输入输出缓冲区；将远端 OMX 的状态设置成 IDLE 状态；将自身也设置成 IDLE 状态。这个状态与 OpenMAX IL 标准中定义的状态相一致，详细的解析可以参见第二章相关内容，OMXNodeInstance:: sendCommand 函

数中调用了 OpenMAX IL 标准中的 OMX_SendCommand 函数。OMXCodec 在实例化的时候为 LOADED 状态。

（9） 在上述说明中，仍然存在一些疑问：OMXCodec 是如何获得 BpOMX 对象来和远端 OMX 通信的？OMX 和 OMXMaster 对象在何时创建的？实际上，这些事情在创建 AwesomePlayer 对象的时候就已经完成了，而 AwesomePlayer 实例化是在视频播放的 setDataSource 流程中（图 2-18 所示）。下图给出了 AwesomePlayer 实例化过程。通过调用 BpMediaPlayerService::getOMX 函数，返回了 BpOMX 对象，之后通过该对象来与 OMX 进程通信。OMXMaster::addVendorPlugin 函数用于将厂商的 libstagefrighthw.so 插件加载进来。

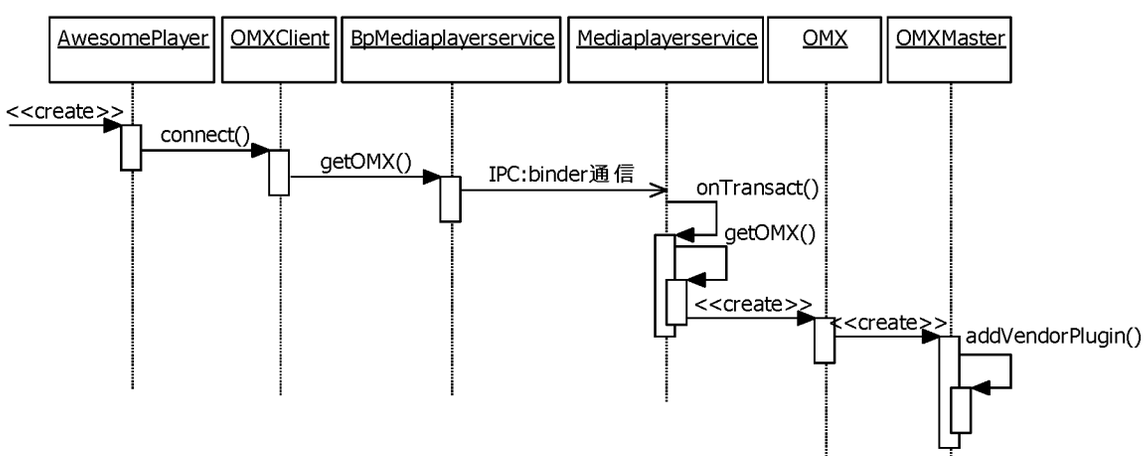

图 2-28 AwesomePlayer 实例化过程

2.5.2 视频解码流程

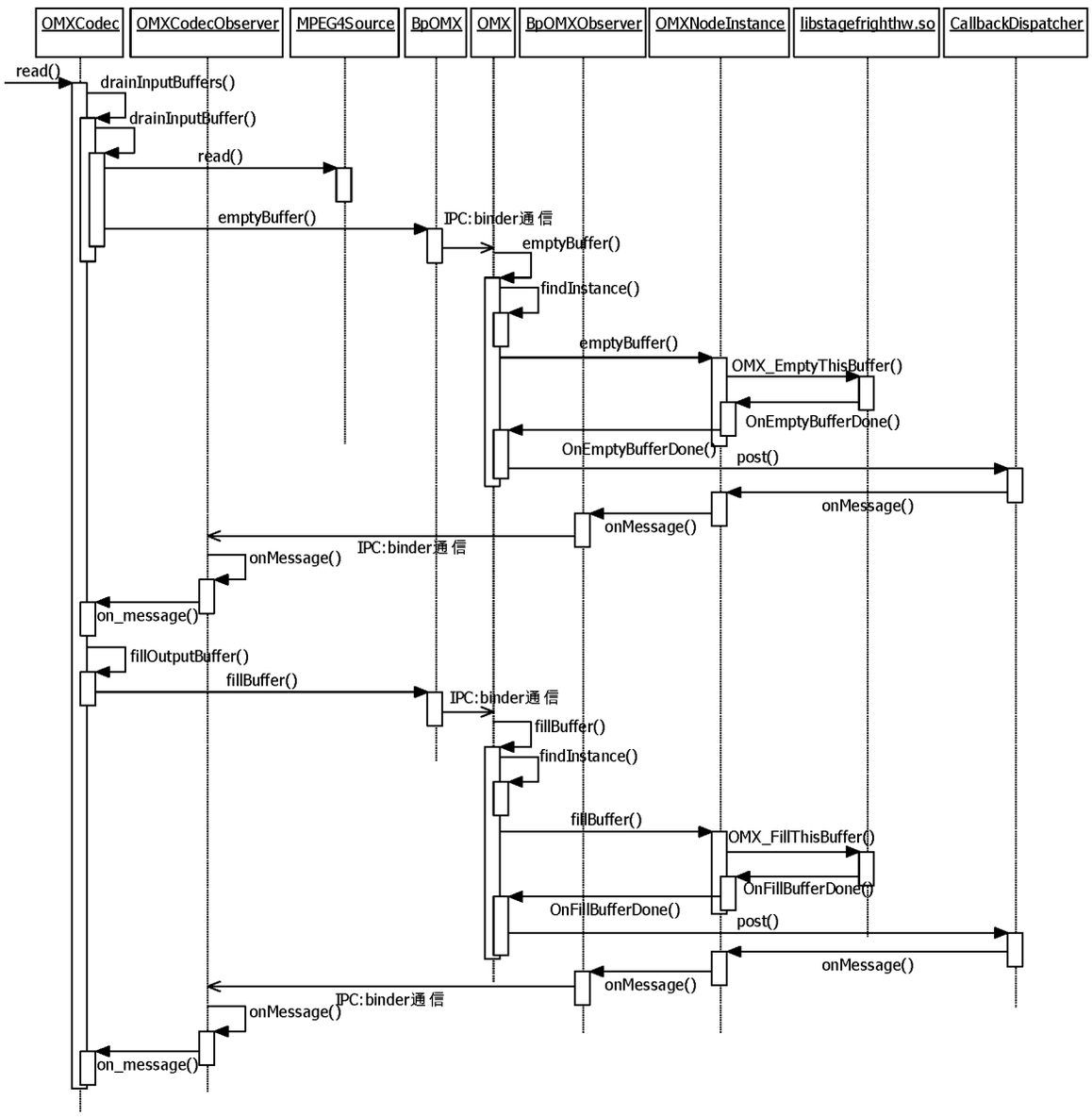

图 2-29 OMX 视频解码流程

如上图所示，视频解码主要是在 OMXCodec::read 函数中实现，在前面分析视频播放的 start 流程时，给出了调用该函数的位置。下面是对视频解码流程的解析：

（1） OMXCodec::read 函数实现主要分为两大步骤：drainInputBuffers 是为 OMX 进程解码视频提供数据；fillOutputBuffer 是从 OMX 进程获取解码后的数据。OMXCodec::read 函数是一个循环调用，知道所有的数据都被处理完。

（2） drainInputBuffers 函数内部首先是通过 MPEG4Source::read 来分析 MP4 文件，

并从中获取视频帧数据，然后利用 binder 通信将数据传给 OMX 进程。OMX 进程最后调用了 libstagefrighthw.so 中的 OMX_EmptyThisBuffer 函数来消耗这些数据，也就是解码视频帧。在完成解码后，会执行 OMXNodeInstance::OnEmptyBufferDone 回调函数。该函数最终会通过 BpOMXObserver 将信息传回给 OMXCodec，由 OMXCodec 处理回调事件。

（3） fillOutputBuffer 函数内部最终是通过调用 libstagefrighthw.so 中的 OMX_FillThisBuffer 函数来获取数据的。其回调过程与 drainInputBuffers 函数的一样，只是回调事件不同。

2.5.3 OMX 相关类的功能总结

到目前为止，已经将 OpenMAX IL 在 Android 多媒体框架中的应用全部讲述完成了，其中最关键的地方在于了解 OMX 相关类的功能及其协同工作过程。下面对这些类进行总结：

（1） OMXCodec：最外层的封装类，提供对外的调用接口。工作于客户端。

（2） OMX 和 BpOMX：OMX 类是业务的接收和逻辑处理类，工作于服务器端，即 OMX 进程。BpOMX 类负责与 OMX 通信，工作于客户端。

（3） OMXClient：用于获取 BpOMX 实例。工作于客户端。

（4） OMXCodecObserver 和 BpOMXObserver：OMXCodecObserver 类接收回调消息，工作于客户端。相应地，BpOMXObserver 类用于发送回调消息，工作于 OMX 进程。

（5） OMXNodeInstance：节点实例，对于每一个 Codec 都会创建一个节点实例，所有节点都由 OMX 类维护。工作于 OMX 进程。

（6） CallbackDispatcher：为每个节点实例创建一个回调调度线程，专门负责该节点回调事件的调度工作。工作于 OMX 进程。

（7） MediaCodecList：维护一个系统支持的 Codecs 名称列表，通过指定的 mime 类型来获取对应的 Codec 名称。工作于客户端。

（8） OMXMaster：负责所有插件的维护和管理。工作于 OMX 进程。

（9） OMXPluginBase：插件接口类，每个插件都要继承并实现该类。工作于 OMX 进

程。

2.6 本章小结

本章首先介绍了 Android 多媒体框架的发展过程，并将 StageFright 和 OpenCore 框架进行了对比。结果表明 StageFright 相比于 OpenCore 具有更简单的架构和更方便的插件开发，以至于在 Android2.x 之后 StageFright 逐渐取代了 OpenCore，成为了 Android 默认的多媒体框架。接着在源代码的基础上对 Android 多媒体框架进行了分析，并以视频播放为例，详细地讲述了多媒体处理的过程以及 OpenMAX IL 在 Android 多媒体框架下的应用，为后面整合 FFmpeg 提供了依据。

第三章 FFmpeg 编码库的研究与设计

3.1 FFmpeg 概述

3.1.1 简介

FFmpeg 是一个开源的音视频编解码解决方案，任何人都可以自由使用，但必须严格遵守 LGPL、GPL 协议。开发者如果使用了以 GPL 协议发布的模块（如 libx264），那么开发出来的应用也必需使用 GPL 协议[22]。

FFmpeg 最初由 Fabrice Bellard 发起的，而现在是由 Michael Niedermayer 在进行维护。许多 FFmpeg 的开发者同时也是 MPlayer 项目的成员，FFmpeg 在 MPlayer 项目中是被设计为服务器版本进行开发。2011 年初，FFmpeg 内部发生政变，Fabrice Bellard 等人不满 FFmpeg 现有的项目管理方式，于是从中跳出来成立了新项目，称作 Libav。而 Debian 系统也宣布 2012 年开始将 ffmpeg 指令改成 avconv。Libav 与 FFmpeg 相比，Libav 在原有资源的基础上，更注重于开发。而 FFmpeg 项目维护者更倾向于接口的稳定，并且会定期从 Libav 同步修改，所以可以认为每一个提交到 Libav 的修改，都会被 FFmpeg 维护者复审一遍，然后再合并进 FFmpeg 项目中[23]。

FFmpeg 本身不仅实现了非常多的音视频编解码器，还支持许多第三方的编解码器，比如 libx264,、libfaac、libmp3lame 等。也因此使得很多视频播放器都使用了 FFmpeg 库。常见的有 VLC，MPlayer，暴风影音，QQ 影音，KMP 等等。FFmpeg 本身是基于 Linux 系统开发的，但是因为使用了纯 C 语言编程，使得它很容易就能编译到不同的平台中去，具有很好的跨平台特性。

3.1.2 关键词解析

Container：存储音视频和字幕数据的容器。比如我们常见的 mp4、avi 文件[24]。

Muxer/Demuxer：复用/解复用器。实现了将音频、视频和字幕数据封装进一个容器或从容器中分离出来的过程。

Codec：编解码器。实现了将原始数据进行压缩编码或者解码的过程。

Stream：指音视频或者字幕流数据。

Frame：Stream 中的一个数据单元。例如表示视频的一帧数据。

Packet：一个原始数据单元。

3.1.3 目录结构

在 FFmpeg 源码的根目录中提供了几个 main 文件：ffmpeg.c、ffserver.c、ffplay.c、ffprobe.c。功能如下所述：

（1） ffmpeg.c：强大的转码工具。提供了视频采集，格式转换，给视频添加水印，流媒体传输，视频抓图等功能。

（2） ffserver.c：HTTP/RTSP 流媒体服务器。它作为单独的服务器可以实现点播服务，与 ffmpeg 结合可以实现实时视频直播服务。

（3） ffplay.c：基于 SDL 显示的音视频播放器。既可以播放本地视频，也可以播放网络流媒体。

（4） ffprobe.c：简单的音视频信息分析器。能够检测出音视频文件的码率、帧率、编码格式等。

FFmpeg 源码中还包含了几个文件夹，每个文件夹都会在编译后生成一个库，对应不同的功能。几个文件夹和对应的功能如下：

（1） libavformat：存储各种 muxer/demuxer 模块源码，并向外提供 muxer/demuxer 的接口。开发者如果使用该文件夹或者第三方的 muxer/demuxer 模块，则需要在代码中包含 avformat.h 头文件。

（2） libavcodec：存储各种 codec 模块源码，并向外提供 codec 的接口。开发者如果使用了该文件夹或者第三方的 codec 模块，则需要在代码中包含 avcodec.h 头文件。

（3） libswscale：用于图像的缩放和颜色空间转换。

（4） libavutil：提供了一些公用的功能函数，比如 CRC 校验，MD5 加密解密，Base64 编解码等。

（5） libavfilter：音视频过滤器。可以对视频进行裁剪，添加 logo、水印等。

（6） libpostproc：用于视频的后期处理。

（7） libswresample：用于音频的重采样。

（8） libavdevice：提供了许多通用输入/输出设备的数据抓取和绘制功能。这些输入/输出设备包括有 Video4Linux2，VfW，DShow，和 ALSA。

3.2 FFmpeg 关键接口分析

3.2.1 主要数据结构

1. AVFormatContext

AVFormatContext 是 FFmpeg 中最顶层的结构体，它包含了音视频处理的所有信息。只要涉及到 Container 或者网络流操作，就都会使用到该结构体，它在 libavformat/avformat.h 中定义。AVFormatContext 结构体中包含的主要成员有：AVInputFormat（或者 AVOutputFormat，同一时间 AVFormatContext 内只能存在其中一个）、AVStream、AVPacket。另外还包含了一些其他的相关信息，比如 title、author、copyright 等。当然如果只是做一些比如 codec 的操作，那么就可以不使用该结构体。

2. AVInputFormat /AVOutputFormat

AVInputFormat/AVOutputFormat 分别是在音视频的 demuxer/muxer 时使用。他们记录了音视频数据使用的解码/编码器。AVInputFormat 提供了一组读入函数指针，用于数据的获取，这些数据可能来自于文件、网络流、摄像头、MIC 等。AVOutputFormat 提供了一组写出函数指针，用于数据的消耗，比如写出到文件中或者通过网络上传到服务器。

AVInputFormat 的初始化设置会在初始化 AVFormatContext 的时候完成，具体实现是在 avformat_open_input 函数中。AVOutputFormat 会根据输出的文件名称来设置默认的编码器。

3. AVStream

AVStream 记录了音频或视频的相关流信息，比如时长，平均帧率，pts（播放时间戳），dts（解码时间戳），time_base（基本时间计量单位）等。对于既包含了音频，也包含了视频的文件，会生成两个 AVStream，一个与音频对应，一个与视频对应。在 AVStream 结构体中还包含了一个非常重要的结构体 AVCodecContext。

4. **AVCodecContext**

AVCodecContext 记录了音频或视频的相关编码参数信息，是在编解码时最重要的一个结构体。这些参数包括编码器/解码器的 id，比特率，视频尺寸，输出的数据格式等等[25]。在编解码的时候，该结构体对象作为函数参数传入。AVCodecContext 中有两个重要的结构体 AVCodec 和 AVFrame。

5. **AVCodec**

AVCodec 用于指示一个编码器或者一个解码器。开发者在使用 FFmpeg 库之前，先要将编解码器注册进来，所谓的注册实际上是将编解码器对应的 AVCodec 结构体对象放进到一个全局的链表中。当开发者需要使用某个编解码器的时候，就会根据编码器/解码器的 id 从该链表中获取到 AVCodec 结构体对象。在 AVCodec 中有包含了编/解码相关的函数指针，就是通过这些函数指针来实现具体的编解码操作。开发者不能直接设置 AVCodec 结构体中的成员值。

6. **AVFrame**

AVFrame 结构体中存储的是原始数据（对于视频为 yuv 或者 rgb 数据，对于音频为 pcm 数据）。编码时作为输入，解码时作为输出。该结构体同时也指示了该帧数据的相关信息，如是否为关键帧，是否作为参考帧等。

7. **AVPicture**

AVPicture 结构体可以通过 AVFrame 结构体对象强制类型转换得到。他只保存了原始数据。一般在对 AVFrame 进行内存分配的时候会强制转换为 AVPicture。

8. **AVPacket**

AVPacket 结构体存储的是音视频的编码数据及相关信息。从媒体文件中获取数据，或者向媒体文件写入时都使用了该结构体。

3.2.2 主要函数

1. av_register_all

注册所有的 muxer 和 demuxer，实际是指将 muxer/demuxer 对应的 AVOutputFormat 对象放进全局链表中。该函数内部同时调用了 avcodec_register_all 函数，用于注册所有的 codec。该函数在使用其他 FFmpeg API 之前首先被调用[26]。

2. avformat_alloc_output_context2

为 AVFormatContext 分配内存。其函数内部会根据函数参数提供的文件名和指定的格式调用 av_guess_format 函数来找到对应 muxer 的 AVOutputFormat 对象。当 AVFormatContext 不再需要使用时，开发者应该调用 avformat_free_context 来释放内存。

3. avcodec_find_encoder

根据函数参数提供的编码 id 查找到对应的 AVCodec[27]。该编码 id 通过 AVOutputFormat->video_codec 或者 AVOutputFormat-> audio_codec 赋值得到。AVOutputFormat->video_codec 或者 AVOutputFormat-> audio_codec 既可以使用默认值，也可以由开发者自己设定。

4. avformat_new_stream

创建 AVStream 结构体对象，并设置进 AVFormatContext。该函数内部同时会创建 AVCodecContext 对象。

5. avcodec_open2

利用给定的 AVCodec 来初始化 AVCodecContext 内部成员变量。当编解码完成之后，开发者应该调用 avcodec_close 来关闭编解码器，释放内存。

6. avcodec_encode_audio2/ avcodec_encode_video2

实现音频/视频的编码。其函数内部实际上是通过调用 AVCodec->encode2 函数指针来实现。

7. avformat_write_header

分配内存给私有数据流，并向媒体文件中写入文件头。该函数内部实际上是通过调用 AVOutputFormat-> write_header 函数指针来实现。

8. av_write_frame

将一帧音频或视频数据写入到媒体文件中。每一帧的数据都必须按照 Container 的规范交错写入，否则应该使用 av_interleaved_write_frame 函数代替（使用 av_interleaved_write_frame 会增加 muxer 的时间，并且需要更多的内存空间了进行帧交错处理，因此尽量避免使用此函数）。该函数内部实际上是通过调用 AVOutputFormat->write_packet 函数指针来实现。

9. av_write_trailer

向媒体文件写入文件尾，并且释放文件私有数据内存。该函数内部实际上是通过调用 AVOutputFormat-> write_trailer 函数指针。

10. avformat_open_input

打开输入流，创建 AVFormatContext 对象，从媒体文件中读取头部数据来探测其封装信息，从而获取对应的 AVInputFormat 对象，然后通过 AVInputFormat 中的 read_header 函数指针来创建 AVStream 对象和 AVCodecContext 对象，设置 AVCodecContext->codec_id。在结束时，应该调用 avformat_close_input 函数来关闭流。

11. avformat_find_stream_info

获取流信息，设置 AVStream 中的成员变量值。

12. avcodec_find_decoder

根据解码器 id 获取对应的 AVCodec。该解码器 id 值等于 AVCodecContext->codec_id。

13. av_read_frame

获取一帧的音频/视频数据。内部实际上是通过调用 AVInputFormat->read_header 函数指针来实现。

14. avcodec_decode_video2/avcodec_decode_audio4

实现视频/音频解码。其函数内部实际上是通过调用 AVCodec->decode 函数指针来实现。

15. sws_getContext

创建 SwsContext 结构体对象。用于图像空间颜色转换。

16. sws_scale

图像空间颜色转换的实现函数。例如将 rgb 图像数据转换成 yuv 图像数据。

3.3 FFmpeg 转码的设计与实现

本节为了更好地说明 FFmpeg 的工作流程，也为了在最终整合 Android 多媒体框架下的编码器时提供思路，这里设计并实现了一个视频转码的例子。该例子中，输入的媒体文件名称为 in.mp4，音频采用 amrnb 编码，视频采用 h263 编码，640x480 的分辨率。要求输出文件为 out.flv，音视编码为 aac，视频编码为 h264，分辨率为 352x288。

该例子中因为既涉及到了 muxer/demuxer，也需要使用到 codec，另外还包含了视频分辨率的转换，因此这里需要将 FFmpeg 中的 libavformat、libavcodec、libswscale、libavutil 四个库添加到项目的依赖库中，并且在代码中包含相应的头文件[28][29]。

1. 注册 muxer/demuxer 和 codec

```
av_register_all();
```

av_register_all 函数通过调用 REGISTER_MUXER(X, x)、REGISTER_DEMUXER(X, x)、REGISTER_ENCODER(X, x) 、REGISTER_DECODER(X, x)，将所有 FFmpeg 支持的 muxer/demuxer 和 codec 放进链表中。其中存放所有 muxer 的链表为 AVOutputFormat *first_oformat；存放所有 demuxer 的链表为 AVInputFormat *first_iformat；存放所有 codec 的链表为 AVCodec *first_avcodec。

2. 初始化输入

（1） 打开源视频文件

```
if (avformat_open_input(&fmt_ic, src_filename, NULL, NULL) < 0) {
    fprintf(stderr, "Could not open source file %s\n", src_filename);
    exit(1);
}
```

其中 src_filename=”in.mp4”，avformat_open_input 函数通过侦测文件内容来获取相应的 demuxer。

（2） 获取视频信息

```
if (avformat_find_stream_info(fmt_ic, NULL) < 0) {
    fprintf(stderr, "Could not find stream information\n");
    exit(1);
}
```

avformat_find_stream_info 函数会获取每个 stream 的编码数据参数。

（3） 查找并打开音视频解码器

```
AVCodecContext *dec_ctx = NULL;
AVCodec *dec = NULL;
……
for(int i=0; i< fmt_ic->nb_streams; ++i) {
```

```
    dec_ctx = fmt_ic->streams[i]->codec;
    if((dec = avcodec_find_decoder(dec_ctx->codec_id)) == NULL) {
       exit(1);
    }
    if ((avcodec_open2(dec_ctx, dec, NULL)) < 0) {
       exit(1);
    }
}
```

由于源文件 in.mp4 包含了音频和视频，因此有两个 stream，需要分别找到音频和视频的解码器并打开。avcodec_open2 函数调用了 AVCoodec->init 函数指针。

3. 初始化输出

（1） 创建 AVFormatContext 对象

```
avformat_alloc_output_context2(&fmt_oc, NULL, NULL, dst_filename);
if (!oc) {
    printf("Could not deduce output format from file extension.\n");
    exit(1);
}
```

向 AVFormatContext *fmt_oc 分配内存，并通过目标文件 dst_filename 来获取默认的 AVOutputFormat 对象，从而获得默认的音视频编码器。

（2） 创建音视频 stream，并打开编码器

```
AVOutputFormat *fmt;
AVCodec *audio_codec, *video_codec;
AVCodecContext *video_ctx, *audio_ctx;
……
fmt = fmt_oc->oformat;
if (fmt->video_codec != AV_CODEC_ID_NONE) {
    fmt->video_codec = AV_CODEC_ID_H264;
    video_codec = avcodec_find_encoder(fmt->video_codec);
    avformat_new_stream(fmt_oc, video_codec);
    video_ctx = fmt_oc->streams[fmt_oc->nb_streams-1]->codec;
    video_ctx ->width = 352;
    video_ctx ->height = 288;
    avcodec_open2(video_ctx, video_codec, NULL);
    ……
}
```

```
if (fmt->audio_codec != AV_CODEC_ID_NONE) {
    fmt->audio_codec = AV_CODEC_ID_AAC;
    audio_codec = avcodec_find_encoder(fmt->audio_codec);
    avformat_new_stream(fmt_oc, audio_codec);
    audio_ctx = fmt_oc->streams[fmt_oc->nb_streams-1]->codec;
    avcodec_open2(audio_ctx, audio_codec, NULL);
    ……

}
ret = avformat_write_header(fmt_oc, NULL);
```

上述代码中，人为地指定了音视频的编码器 id。利用编码器来创建音视频流，并设置 AVStream 对象中的成员参数。这里除了设置 AVOutputFormat->audio_codec 外，还可以设置 AVStream->AVCodecContext 中的编码参数，例如视频的分辨率，比特率，帧率等。最后打开编码器并向目标文件写入文件头。

4. 获取一帧的音（视）频数据

```
while (av_read_frame(fmt_ic, &pkt) >= 0) {
    //解码，颜色空间转换，编码，封装
    ……
}
```

av_read_frame 会从源文件中获取一帧的音频或视频数据。该函数返回的数据不一定是有效数据，但为了给解码器提供完整的解码信息，该函数不会忽略无效的帧数据。由上述代码可以看到，程序在循环的读取一帧的数据，直到所有的数据都被读完才停止。在循环体中，需要实现的是一帧数据的解码，再编码，然后封装到目标文件中（这里为 out.flv）。

5. 音（视）频解码

```
if(pkt.stream_index == video_stream_idx) {
    ret = avcodec_decode_video2(video_dec_ctx, frame, got_frame, &pkt);
    if(*got_frame) {
        //视频颜色空间转换，编码，封装
    }
} else if(pkt.stream_index == audio_stream_idx) {
    ret = avcodec_decode_audio4(audio_dec_ctx, frame, got_frame, &pkt);
    if(*got_frame) {
```

```
            //音频格式转换，编码，封装
        }
    }
```

当解码出一帧视频或者音频后，got_frame 值会置 1，此时说明已经完整地解码出了一帧数据，接下来才可以进行编码等操作。

6. 视频颜色空间转换

```
    AVCodecContext *src_ctx;
    ……
    src_ctx = fmt_ic->streams[pkt. stream_index]->codec;
    img_convert_ctx = sws_getContext(src_ctx->width, src_ctx->height,
src_ctx-> pix_fmt, video_ctx->width,   video_ctx->height, video_ctx->
pix_fmt, SWS_POINT, NULL, NULL, NULL);//SWS_BICUBIC
    if(img_convert_ctx == NULL){
        return -1;
    }
    sws_scale(img_convert_ctx, m_tmpPicture->data, m_tmpPicture->linesize, 0,
srcHeight, m_picture->data, m_picture->linesize);
    sws_freeContext(img_convert_ctx);
```

这里视频的颜色空间转换主要是将 640x480 的分辨率转换成 352x288 的分辨率。

7. 音（视）频的编码和封装

```
    //视频
    AVFrame *frame = (AVFrame *) m_picture;
    ret = avcodec_encode_video2(fmt_oc, &out_pkt, frame, &got_packet);
    if (!ret && got_packet && pkt.size) {
        pkt.stream_index = video_stream->index;
        av_write_frame(fmt_oc, &out_pkt);
    }
    ……
    //音频
    ret = avcodec_encode_audio2(c, &pkt, frame, &got_packet);
    if (!ret && got_packet && pkt.size) {
        pkt.stream_index = audio_stream->index;
        av_write_frame(fmt_oc, &out_pkt);
    }
```

当编码出一帧数据之后，got_packet 会置 1，这时就可以把编码出来的数据按照 flv

格式封装进目标文件中。

由于h264编码具有延时性，即是说在将第i帧的数据都放进avcodec_encode_video2编码之后，出来的并不是第i帧的编码数据，而是之前某一帧的编码数据，所以为了能够编码出所有的数据，在最后结束编码时，还需要以NULL作为源数据进行编码，直到avcodec_encode_video2返回错误码。

8. 写文件尾，释放所有资源

```
av_write_trailer(fmt_oc);
avcodec_close (video_stream->codec);
avcodec_close (audio_stream->codec);
avcodec_close(video_dec_ctx);
avcodec_close(audio_dec_ctx);
avformat_free_context(fmt_ic);
avformat_free_context(fmt_oc);
```

3.4 FFmpeg扩展编码库的设计与实现

FFmpeg作为一个完善的多媒体编解码解决方案，不仅仅是因为他本身就集合了极多的muxer/demuxer和codec，还因为他提供了非常方便的接口来加载第三方的muxer/demuxer和codec，这些第三方的muxer/demuxer和codec被称为是FFmpeg的插件。由于在实际应用中FFmpeg自带的muxer/demuxer已经完全满足需求，而我们常常需要加载第三方的codec，例如libx264库，FFmpeg就是用它来进行H264编码的。所以本节中我们主要研究FFmpeg是如何设计和加载codec的。在FFmpeg的编码优化设计中，主要也是使用了这种加载方式将Android多媒体中的硬件编码器。

前面在介绍FFmpeg的主要函数时，也曾提到编码函数内部实际上是调用了AVCodec->encode2函数指针，而解码函数则是调用了AVCodec->decode函数指针。所以如果我们要加载第三方的编解码库就必须要实现AVCodec结构体中的相关成员变量。为了更好说明FFmpeg codec的设计和加载过程，我们将通过一个yuv4编码器的实现例子进行解析。

以下列出了AVCodec结构体中重要的成员变量及其说明：

```
typedef struct AVCodec {
```

```
        ……
        //编(解)码器名称
        const char *name;
        //编(解)码器类型，enum AVMediaType 中的一种
        enum CodecType type;
        //编(解)码器的 id，enum AVCodecID 中的一种
        enum CodecID id;
        //只与该编(解)码器相关的私有数据大小，与 AVCodecContext->priv_data 对应
        int priv_data_size;
        //编(解)码器初始化函数指针
        int (*init)(AVCodecContext *);
        //编码函数指针
        int (*encode2)(AVCodecContext *, uint8_t *buf, int buf_size, void
*data);
        //关闭编(解)码器函数指针
        int (*close)(AVCodecContext *);
        //解码函数指针
        int (*decode)(AVCodecContext *, void *outdata, int *outdata_size,
uint8_t *buf, int  buf_size);
        //为统一编解码行为而设定的一些兼容选项，见 CODEC_CAP_*宏
        int capabilities;
        //支持的像素点格式。编码时表示支持的输入图像格式；解码时表示支持的输出图像格式
        const enum AVPixelFormat *pix_fmts;
        //支持的采样格式
        const enum AVSampleFormat *sample_fmts;
        ……
    } AVCodec;
```

首先我们需要做的是实现这个 AVCodec 结构体：

```
    AVCodec ff_yuv4_encoder = {
        .name         = "yuv4",
        .long_name    = NULL_IF_CONFIG_SMALL("Uncompressed packed 4:2:0"),
        .type         = AVMEDIA_TYPE_VIDEO,
        .id           = AV_CODEC_ID_YUV4,
        .init         = yuv4_encode_init,
        .encode2      = yuv4_encode_frame,
        .close        = yuv4_encode_close,
        .pix_fmts     = (const enum AVPixelFormat[]){ AV_PIX_FMT_YUV420P,
AV_PIX_FMT_NONE },
    };
```

这个 AVCodec 对象就是我们在使用 yuv4 编码时用到的。在调用 avcodec_register_all 函数时（会在 av_register_all 中调用），会将该对象注册到链表中。

```
void avcodec_register_all()
{
    ……
    REGISTER_ENCDEC (YUV4,            yuv4);
    ……
}
```

宏 REGISTER_ENCDEC 扩展开之后，发现是调用了 register_avcodec 函数，并且需要定义了 CONFIG_YUV4_ENCODER 宏才会去调用 register_avcodec 函数。通过判断是否定义宏的方式可以避免将所有的编码器都编译到库中，这也有利于做编码器的裁剪。那么这个 CONFIG_YUV4_ENCODER 具体是在哪里生成的呢？他是在项目的 Configure 文件中提取出来的，提取命令为：

```
find_things(){
    thing=$1
    pattern=$2
    file=$source_path/$3
    sed -n "s/^[^#]*$pattern.*([^,]*, *\([^,]*\)\(,.*\)*).*/\1_$thing/p"
"$file"
}
ENCODER_LIST=$(find_things  encoder  ENC     libavcodec/allcodecs.c)
```

从 AVCodec 对象还可以看出，编码器的名称为 yuv4，类型为 AVMEDIA_TYPE_VIDEO，表明这是一个视频编码器，编码器的 id 为 AV_CODEC_ID_YUV4，在枚举类型 enum AVCodecID 中定义：

```
enum AVCodecID {
    ……
    AV_CODEC_ID_YUV4      = MKBETAG('Y','U','V','4'),
    ……
}
```

enum AVCodecID 定义了所有 FFmpeg 支持的编（解）码器 id，开发者在不打乱原有 id 值的情况下，可以添加自己的编（解）码器 id。

另外，还需要实现 AVCodec 中的几个函数指针，这也是能够使得编（解）码器正常

工作的重要前提。例子中，将 AVCodec->init 函数指针指向了 yuv4_encode_init 函数，AVCodec->encode2 函数指针指向了 yuv4_encode_frame 函数，AVCodec->close 函数指针指向了 yuv4_encode_close 函数。

最后为了能够编译通过，需要在 libavcodec/Makefile 文件中加入依赖关系

```
……
OBJS-$(CONFIG_YUV4_ENCODER)           += Yuv4enc.o
……
```

3.5 本章小结

本章首先介绍了 FFmpeg 项目的架构，以及对其关键接口进行了分析。接着设计并编写了一个视频转码的应用程序来说明 FFmpeg 的基本工作流程：demuxer -> decode -> scale -> encode -> muxer。最后，为了能够让 FFmpeg 和 Android 平台的硬件编码器结合起来，我们还研究并实现了 FFmpeg 编码器插件的扩展，这在 FFmpeg 的优化中起到了至关重要的作用。

第四章 FFmpeg 在 Android 多媒体中的编码优化

将 FFmpeg 应用到 Android 多媒体平台中，能够利用 FFmpeg 丰富的多媒体处理能力，实现各种功能应用，比如音视频录制，音视频实时通话，添加水印等等。然而 FFmpeg 在 Android 中有着严重的缺点：编码性能低下。为了解决这一问题，本章依据上一章中的编码库扩展技术将 Android 多媒体中的硬件编码器设计成了插件形式，提供给 FFmpeg 使用，从而极大地优化了 FFmpeg 的编码效率。

4.1 框架结构设计

4.1.1 整体框架

目前，FFmpeg 在 Android 平台中的应用都是通过调用 FFmpeg 库的形式来完成的。然而这样直接调用 FFmpeg 库的方式只能用来设计一些关于解码或者颜色空间转换的项目，例如设计视频播放器。而对于视频编码的性能需求，由于智能手机 CPU、内存等硬件的限制，FFmpeg 即使拥有非常优秀的代码设计架构，也无法达到令人满意的编码效率。为此，本章将对 FFmpeg 进行优化，优化过程中利用了 Android 平台自有的硬件编码器，从而实现 FFmpeg 在 Android 中的高效编码。其优化后整体的框架图如下所示：

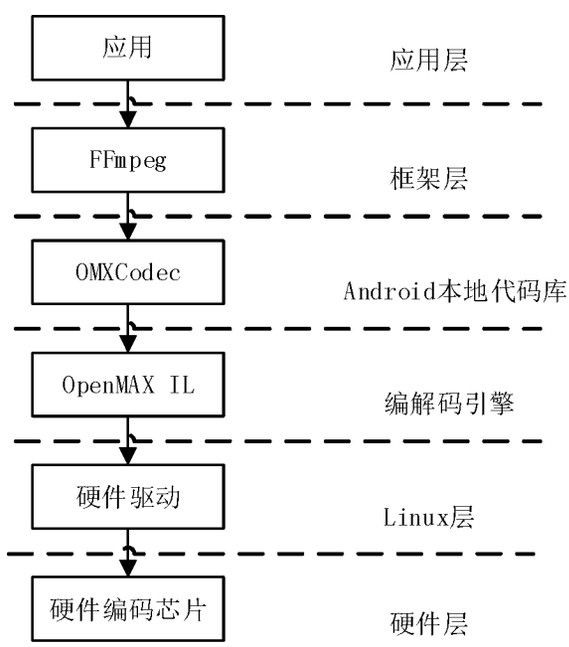

图 4-1 FFmpeg 在 Android 多媒体平台下的编码优化设计框架

图 4-1 给出了 FFmpeg 在 Android 多媒体平台下优化后的整体框架图。从图中可以看到，FFmpeg 库衔接了应用层和 Android 底层，为顶层应用提供了编码器的调用接口，而在底层则整合了 Android 系统下的 OpenMAX IL 接口。OMXCodec 类是为 OpenMAX IL 设计的封装类，FFmpeg 通过直接使用该类来达到使用 OpenMAX IL 的目的。在前面章节中已经介绍过，OpenMAX IL 的数据通信方式有两种：一钟是隧道通信，另一种是非隧道通信。在实际应用中可以同时存在这两种通信方式。然而，在与 FFmpeg 整合过程中，由于 FFmpeg 本身具有一套非常清晰明了的接口来使用数据，而且这些数据的流通是由上层调用者来决定的，所以 OpenMAX IL 的隧道通信方式并不适用于 FFmpeg。在具体的优化设计中，为了避免修改 FFmpeg 的标准接口，选择了非隧道数据通信的工作方式。以视频转码为例，在非隧道通信方式下，FFmpeg 和 OpenMAX IL 的数据流通过程如图 4-2 所示，

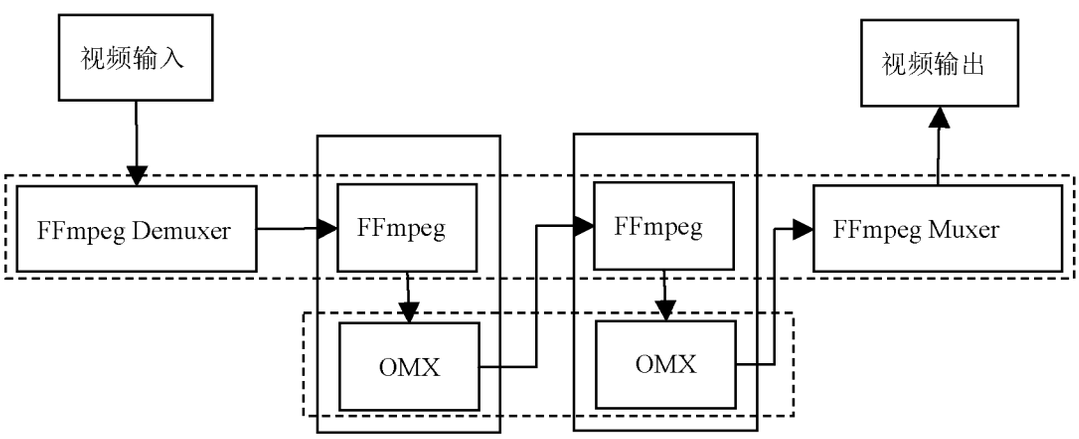

图 4-2 视频转码的数据流通过程

4.1.2 OpenMAX IL 非隧道通信工作过程

4.1.2.1 组件初始化

（1） IL Client 调用 OMX_GetHandle 函数，激活实际应用到的组件。同时将该组件的所有配置资源都加载到内存中。IL Core 会通过调用 SetCallbacks 函数将 IL Client 的回调函数传递给组件。当上述的步骤执行成功后，返回一个与组件对应的 Handle，并且该组件进入 OMX_StateLoaded 状态。

（2） IL Client 通过调用 OMX_SetParameter 函数来配置组件和它的端口，该函数可以被多次执行。当完成配置之后，IL Client 调用 OMX_SendComand 函数将组件设

置成 OMX_StateIdle 状态。在这之后，IL Client 才能为组件建立缓冲区和使用组件的所有端口。

（3） IL Client 通过调用 OMX_AllocateBuffer 或者 OMX_UseBuffer 函数来创建缓冲区，并且根据组件的总端口数和每个端口所需的缓冲区数目会多次调用该函数。如果使用的是 OMX_UseBuffer 函数，那么需要预先创建缓冲区，然后将其传递给组件。又或者，IL Client 通过使用 OMX_AllocateBuffer 函数让组件来创建缓冲区。

（4） 当上述的初始化配置完成之后，组件也会完成状态的转换，并向 IL Client 返回状态转换完成事件。至此，组件初始化的工作完成，IL Client 将通过返回的 Handle 来使用该组件。

4.1.2.2 数据流处理

从整体上来看，OpenMAX IL 组件完成的工作就是对数据进行处理。例如，源视频数据经过编码组件后输出的是编码数据，而解码组件能够将编码数据解码出源数据。组件在初始化完成后就可以进行数据的处理，其处理过程如下:

（1） IL Client 调用 OMX_FillThisBuffer 函数为组件的输出端口提供一个或多个空缓冲区。当组件有数据输出时，就会填充这些缓冲区，并且这些缓冲区可以被重复使用。

（2） IL Client 调用 OMX_EmptyThisBuffer 函数，将存有数据的缓冲区传递给组件的输入端口，组件开始对数据进行处理。当组件有数据输出时会执行 OMX_FillBufferDone 回调函数。该回调函数在初始化的时候由 IL Client 设定。当数据处理完成后，组件还会回调 OMX_EmptyBufferDone 函数。

4.1.2.3 释放资源

（1） 当 IL Client 决定要停止执行组件时，首先需要将组件的状态设置成 OMX_StateIdle 状态，这样组件才会将所有的缓冲区返回给它们的提供者。

（2） 在组件完成了 OMX_StateIdle 状态的转换之后，IL Client 就可以继续请求组件转换成 OMX_StateLoaded 状态。IL Client 应该调用 OMX_FreeBuffer 函数来释放所有的组件缓冲区。OMX_FreeBuffer 函数需要依赖组件来移除指定端口下的缓冲区。如果组件是通过 OMX_AllocateBuffer 函数来创建缓冲区，那么 OMX_FreeBuffer 函数内部会利用组件来释放缓冲区内存；如果是通过 IL Client

调用 OMX_UseBuffer 函数来创建缓冲区并传递给组件，那么需要 IL Client 在调用了 OMX_FreeBuffer 函数后，还要自己释放缓冲区内存。

（3） 当释放完所有的缓冲区后，IL Client 还要调用 OMX_FreeHandle 函数来释放 Handle 资源。

4.1.3 FFmpeg 在 Android 多媒体平台下的编码优化设计

由于无论是 FFmpeg 还是 Android 多媒体框架中使用到的 OpenMAX IL，其编码流程都是相同的，包含三个步骤：初始化（编码器），数据处理（编码音视频帧），释放资源（停止编码器，释放内存）。因此这为 FFmpeg 融合 Android 多媒体下的硬件编码器提供了非常有力的前提条件。在 FFmpeg 中实现这三个步骤的函数分别为：avcodec_open2，avcodec_encode_audio2/ avcodec_encode_video2，avcodec_close。因此，将 Android 多媒体中的硬编码器作为一个插件整合到 FFmpeg 中，必须将 Android 多媒体中相关的这三个流程分别整合到这些函数中。FFmpeg 中通过函数指针来指向不同编码器的具体编码操作，从而可以使用相同的函数接口就能得到不同的编码输出。这些函数指针位于 AVCodec 结构体中，所以优化设计的重点就是如何实现该结构体。

然而在优化 FFmpeg 编码器的过程中，除了利用好 FFmpeg 和 Android 多媒体相同的部分之外，还需要兼容它们不同的工作机制。能够充分考虑到这这些因素才能完成最终的优化目的。对于它们的工作机制，描述如下：

（1） FFmpeg 在编码的时候，数据源需要由上层提供。即只有使用者提供了数据后，FFmpeg 才会进行编码，否则会暂停工作，这是一个被动接受数据然后处理的过程。

（2） Android 多媒体将 OpenMAX IL 进行了封装，对外提供的接口类为 OMXCodec。在使用该接口前，首先需要为数据源创建 MediaSource 子类对象并传递进 OpenMAX IL 组件中，之后调用 OMXCodec::start 函数能够让 OpenMAX IL 开始自动化工作，所谓的自动化是指 OpenMAX IL 自身能够循环地从 MediaSource 子类对象中获取源数据，然后进行编码。使用者只需要调用 OMXCodec::read 就能够获取到编码后的数据。因此这是一个主动获取数据然后处理的过程。

相比较之后可以发现，Android 多媒体下的编码自动化程度要比 FFmpeg 高，但其实

现的复杂度也要比 FFmpeg 高。因此，在考虑 FFmpeg 与 Android 多媒体编码器整合时，尽量将 FFmpeg 兼容到 Android 多媒体的工作机制中。

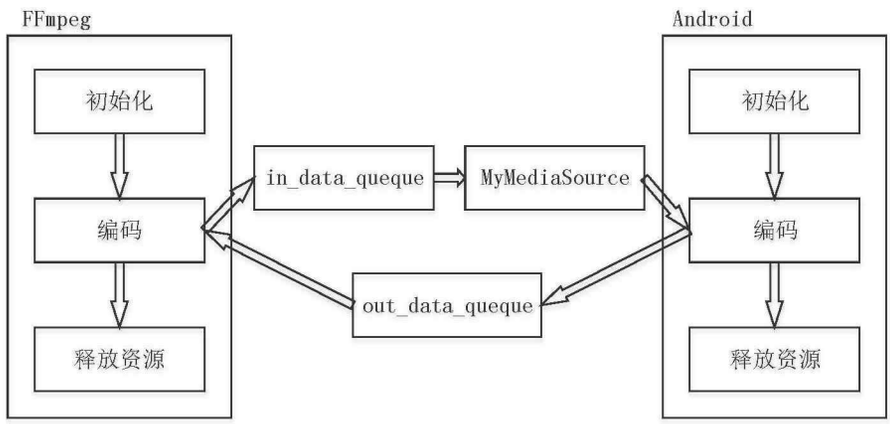

图 4-3 FFmpeg 和 Android 编码器的协作示意图

FFmpeg 和 Android 多媒体的编码协作过程如图 4-3 所示，由于它们的工作机制不同导致了在两者之间需要使用队列来传输数据。这里设计了两个队列：一个是输入队列，用于存储未编码的源数据；另一个是输出队列，用于存储编码完之后的数据。FFmpeg 和 Android 多媒体通过共同存取这两个数据队列的方式进行通信，具体过程如下：FFmpeg 将源数据放到输入队列中，然后检测输出队列是否存有数据，如果有就从中取出并返回给上层使用者；Android 多媒体下的 OMX 则循环的检测输入队列中是否有有效数据，如果有则取出并送到编码器组件进行编码，将编码后的数据放进输出队列中。这样做的目的是能够使得 FFmpeg 和 Android 多媒体在无需考虑彼此的工作流程的情况下，也能够整合到一起，相互利用。

在设计中，因为 Android 多媒体中的编码是一个自动化过程，无需加以人工的干涉，所以将其放进一个独立的子线程中。利用 MyMediaSource 来获取输入队列中的源数据，然后进行处理。

4.2 FFmpeg 编码优化的实现

FFmpeg 的编码优化是指将 Android 中的硬编码器设计成插件的形式加入到 FFmpeg 中，从而利用硬件编码达到高效编码的目的。插件设计的重点是要实现 AVCodec 结构体中的相关成员变量。由于 Android 移动终端很多时候都是使用了 H264 编码，下面就以 H264 硬编码器为例说明优化的实现过程。

```
AVCodec ff_android_encoder = {
    .name        = "android_h264",
    .long_name    = NULL_IF_CONFIG_SMALL("Android H.264"),
    .type        = AVMEDIA_TYPE_VIDEO,
    .id          = AV_CODEC_ID_H264,
    .priv_data_size = sizeof(AndroidH264Context),
    .init        = android_encode_init,
    .encode2     = android_encode_frame,
    .close       = android_encode_close,
    .pix_fmts    = (const enum AVPixelFormat[]){ AV_PIX_FMT_YUV420P,
AV_PIX_FMT_NONE },
};
```

上述代码实例化了 AVCodec 结构体，它是作为 FFmpeg 和 Android 多媒体结合的接口。其中 AVCodec::init，AVCodec::encode2，AVCodec::close 三个函数指针分别指向了 android_encode_init，android _encode_frame，android _encode_close 三个函数。而这三个函数分别需要完成编码器的初始化，数据处理和释放资源的工作。

4.2.1 初始化

（1） 私有数据

关于编码的一些通用参数信息，会存储在 FFmpeg 的 AVCodecContext 结构体中。然而，不同的编码器会有特定的私有数据信息，这些信息则存在 AVCodecContext::priv_data 中。它属于 void 类型，所以可以将其转换成任意的结构体类型。在这里我们将整合需要用到的私有数据信息定义在 AndroidH264Context 结构体：

```
struct AndroidH264Context {
    sp<MediaSource> *source;
    List<Frame*> *in_data_queue, *out_data_queue;
    pthread_mutex_t in_data_mutex, out_data_mutex;
    pthread_cond_t condition;
    volatile sig_atomic_t thread_started, thread_exited, stop_encode;
    OMXClient *client;
    sp<MediaSource> *encoder;
    ……
};
```

上述给出了编码中最主要的私有数据，它们的作用分别为：

source：指向 MyMediaSource 类对象，提供源数据。

in_data_queque，out_data_queque：分别存取输入、输出数据。

in_data_mutex, out_data_mutex：防止多线程同时对输入、输出队列进行存取。

condition：信号量，用于等待有效的源数据输入。

thread_started, thread_exited, stop_encode：记录 Android 多媒体编码线程的工作状态。

client：用于连接 Android 多媒体中的 OMX。

encoder：Android 多媒体编码的接口类 OMXCodec 对象。

（2） MyMediaSource 类实现

MyMediaSource 类中最重要的是实现 read 函数。Android 中的编码器通过调用该函数来获取未编码的数据。

```
    status_t MyMediaSource::read(MediaBuffer **buffer, const
MediaSource::ReadOptions *options) {
        //如果编码子线程已经停止，则返回数据流结束错误
        if (s->thread_exited)
            return ERROR_END_OF_STREAM;
        ……
        //等待，直到输入队列中存入有效数据
        while (s->in_data_queue->empty())
             pthread_cond_wait(&s->condition, &s->in_data_mutex);
        ……
        //将源数据放进 buffer 中，作为函数参数返回
        frame = *s->in_data_queue->begin();
        buf_group.acquire_buffer(buffer);
        memcpy((*buffer)->data(), frame->buffer, frame->size);
        (*buffer)->set_range(0, frame->size);
        (*buffer)->meta_data()->clear();
    }
```

（3） 初始化函数实现

```
    static int android_encode_init(AVCodecContext *avctx) {
        ……
        //设置 Android 多媒体编码器的参数
        meta->setCString(kKeyMIMEType, MEDIA_MIMETYPE_VIDEO_AVC);
```

```
        meta->setInt32(kKeyWidth, avctx->width);
        meta->setInt32(kKeyHeight, avctx->height);
        //开启新线程，处理 OMX 返回的事件
        android::ProcessState::self()->startThreadPool();
        //初始化私有数据
        s->source    = new sp<MediaSource>();
        *s->source   = new MyMediaSource(avctx, meta);
        s->in_data_queue  = new List< Frame*>;
        s->out_data_queue = new List< Frame*>;
        s->client    = new OMXClient;
        //连接 Android 多媒体中的 OMX 进程
        s->client->connect();
        //创建与 OMX 编码器通信的接口类
        s->encoder  = new sp<MediaSource>();
        *s->encoder = OMXCodec::Create(s->client->interface(), meta,
            true, *s->source, NULL, OMXCodec::kClientNeedsFramebuffer);
        ……
        //让 OMX 编码器开始工作
        (*s-> encoder)->start();
        //初始化队列的互斥变量和信号量
        pthread_mutex_init(&s->in_data_mutex, NULL);
        pthread_mutex_init(&s->out_data_mutex, NULL);
        pthread_cond_init(&s->condition, NULL);
        ……
    }
```

该函数会在 avcodec_open2 中执行。OMXCodec::Create 函数内部通过函数参数完成编码器组件的获取，以及组件的初始化操作。

4.2.2 数据处理

（1） 数据流处理函数

```
 static int android_encode_frame (AVCodecContext *avctx, AVPacket *avpkt,
const AVFrame *frame, int *got_packet_ptr) {
        ……
        //创建 OMX 编码子线程
        if (!s->thread_started) {
           pthread_create(&s->encode_thread_id, NULL, &encode_thread, avctx);
           s->thread_started = true;
```

```
    }
    ……
    //将未编码数据存入到 in_data_queque 队列中
    pthread_mutex_lock(&s->in_data_mutex);
    s->in_data_queue->push_back(raw_frame);
    pthread_cond_signal(&s->condition);
    pthread_mutex_unlock(&s->in_data_mutex);
    ……
    while (true) {
        //等待编码数据输出
        pthread_mutex_lock(&s->out_data_mutex);
        if (!s->out_data_queue->empty()) break;
        pthread_mutex_unlock(&s->out_data_mutex);
        ……
    }
    //获取最先的编码数据
    Encoded_frame = *s->out_data_queue->begin();
    s->out_data_queue->erase(s->out_data_queue->begin());
    pthread_mutex_unlock(&s->out_data_mutex);
    ……
}
```

该函数会在 avcodec_encode_video2 函数中执行。从上述代码可以看到，这里其实并没有执行真正的编码操作，而是负责存取数据。真正的编码操作是在 encode_thread 子线程中。

```
void* encode_thread(void *arg) {
    do {
        //编码，返回编码后的数据
        status = (*s->encoder)->read(&buffer);
        ……
        //将编码数据存入 out_data_mutex 队列中
        pthread_mutex_lock(&s->out_data_mutex);
        s->out_data_queue->push_back(frame);
        pthread_mutex_unlock(&s->out_data_mutex);
        ……
    }while(!encode_done && !s->stop_encode)
}
```

(*s->encoder)->read 函数实际是调用了 OMXCodec::read 函数。该函数会从

MyMediaSource::read 中获得源数据，然后通过 OMX_EmptyThisBuffer 来通知 OMX 编码器组件进行处理，最后将编码数据返回。由于在编码子线程运行之初，还没有数据存入到输入队列 in_data_queque 中，所以它会阻塞在 MyMediaSource::read 函数中，直到上层使用者调用 avcodec_encode_video2 函数将数据存进输入队列。编码子线程相当于是运行在后台的自动化工作过程，与 Android 多媒体中的编码工作机制相适应。

4.2.3 资源释放

```
static int android_encode_close(AVCodecContext *avctx) {
    ……
    //停止编码子线程
    s->stop_encode = 1;
    pthread_join(s->encode_thread_id, NULL);
    ……
    while (!s->in_data_queue->empty()) {
        //清空输入队列，并释放内存
    }
    ……
    while (!s->out_data_queue->empty()) {
        //清空输出队列，并释放内存
    }
    //停止 OMX 编码器组件工作
    (*s->encoder)->stop();
    s->client->disconnect();
    //释放私有数据内存资源
    delete s->in_data_queue;
    delete s->out_data_queue;
    delete s->client;
    delete s->encoder;
    delete s->source;
    pthread_mutex_destroy(&s->in_data_mutex);
    pthread_mutex_destroy(&s->out_data_mutex);
    pthread_cond_destroy(&s->condition);
}
```

该函数会在 avcodec_close 函数中执行。

4.3 代码移植

编译平台：Ubuntu 10.04

目标平台：Android 2.33

4.3.1 Linux 下的编译环境搭建

（1） 安装 Java 开发环境 JDK

```
$ sudo apt-get install openjdk-7-jdk
```

（2） 安装相关软件包

```
$ sudo apt-get install gnupg git-core flex bison gperf mingw32 xsltproc \
  zip libx11-dev zlib1g-dev curl lib32z-dev lib32ncurses5-dev tofrodos \
  x11proto-core-dev libc6-dev python-markdown build-essential ia32-libs \
  libgl1-mesa-dev libxml2-utils lib32readline5-dev g++-multilib
```

4.3.2 Android 源码下载和编译

（1） 安装 Repo

```
$ curl http://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

repo 是在 linux 环境中开发 android 主机环境需要的工具包名称，主要用来下载和管理 android 项目。[30]

（2） 设置身份验证

利用 gmail 账号登陆网页 https://android.googlesource.com/new-password，将获取到的身份信息追加到~/.netrc 文件结尾。如果不存在此文件则自己新建一个[31]。

（3） 建立 Repo 客户端

```
$ mkdir ANDROID-4.0.4
$ cd ANDROID-4.0.4
$ repo init -u https://android.googlesource.com/a/platform/manifest -b
android-4.0.4_r1
```

repo 客户端用于与 Git 服务器通信。

（4） 自动下载

由于需要长时间下载 Android 源码，但因为网络的原因，可能会中断。为了能够在网络恢复后在断开的位置重新下载，将相关的命令写进 repo_sync.sh 文件中，文件内容如下：

```
#!/bin/bash
repo sync
while [ $? -ne 0 ];do
sleep 3
repo sync
done
```

开启下载：

```
$ ./ repo_sync.sh
```

（5） 编译源码

进入 Android 源码根目录，执行以下命令：

```
$ source build/envsetup.sh
$ lunch full-eng
$ make -j4
```

当编译完成后，会生成三个镜像文件、apk 应用程序和共享库。其中本论文使用到的主要是 libstagefright.a 以及/frameworks/base/include/media/stagefright/文件夹中的头文件。将 libstagefright.a 放到~/ffmpeg/lib/文件夹中，将头文件放到~/ffmpeg/include/libstagefright/文件夹中。

4.3.3 获取独立交叉编译链

每个 Android 源码版本中都会提供 NDK（Native Development Kit），用于编译 Android 源码。如果直接使用 NDK 编译 FFmpeg 库，就需要为 FFmpeg 编写.mk 文件，增加了编译的复杂度。为了尽可能地减少对编译文件的增减，这里会从 NDK 中提取出独立的交叉编译链，这样就可以使用 FFmpeg 原有的 Makefile 文件来进行编译。

（1） 提取编译链

进入 Android 源码根目录，执行命令：

```
$ ndk/build/tools/make-standalone-toolchain.sh --platform=android-10
--install-dir=~/android-toolchain
```

（2） 设置交叉编译环境

```
$ export PATH=~/android-toolchain/bin:$PATH
$ export CC=arm-linux-androideabi-gcc
$ export CXX=arm-linux-androideabi-g++
$ export CXXFLAGS="-lstdc++"
```

4.3.4 编译 FFmpeg 源码

```
$ ./configure --prefix=~/ffmpeg --enable-static --enable-small
--enable-memalign-hack --enable-gpl --enable-nonfree --enable-pthreads
--extra-ldflags=-L~/ffmpeg/lib \
--extra-cflags=-I~/ffmpeg/include/libstagefright --enable-android_h264
--disable-yasm
$ make
$ make install
```

编译完成后，将会得到 FFmpeg 编码优化后的静态库和相关头文件，存放在~/ffmpeg 中。开发者只要使用这些库就能够编写出在 Android 中高效运行的多媒体程序。

4.4 本章小结

本章详细地说明了 FFmpeg 在 Android 多媒体平台下的编码过程。首先分析了 FFmpeg 和 Android 多媒体下的编码流程，并由此提炼出两者相同的三个编码步骤：资源初始化、数据处理、资源释放。根据这三个步骤将 Android 中的编码过程分别嵌入到 FFmpeg 的三个函数中，从而实现整合的目的。最后介绍了如何从 NDK 中提取出交叉编译链，利用它来编译 FFmpeg，最终生成能够在 Android 中运行的代码库。

第五章 结果测试与分析

5.1 测试环境

FFmpeg 在 Android 多媒体平台下的编码优化设计是为了能够让 FFmpeg 适合在 Android 中应用，提高其编码效率。本文使用的测试机型号为三星 I9100 galaxy S2，主要参数如下表所示：

部件	参数
CPU/主频	三星 Exynos 4210/ 1228MHz
CPU 核心数	双核
手机内存 ROM	1GB
系统内存 RAM	16GB
操作系统	Android OS v4.0.3
屏幕分辨率	480x800

表 5-1 测试机参数

5.2 测试方案

本文的测试主要是对比 FFmpeg 在优化前后的编码性能。为此会设计一个屏幕截屏并转成视频流的测试代码。未优化时，FFmpeg 使用 libx264 中的 H264 进行软编码；优化后，FFmpeg 使用 Android 中的 H264 硬件编码。由于优化前后并没有影响到 FFmpeg 的调用接口，所以这两种情况都可以使用相同的一套测试代码。唯一的区别是在编译时需要链接不同的编码库。对于优化后的编码库生成方式在上一章已经详述。对于没优化的编码库，交叉编译链的生成相同，而在最后一步编译 FFmpeg 时不同。不同点在于需要先编译出 libx264 库，然后再依据它来编译 FFmpeg。编译 libx264 的命令如下：

```
$ ./configure --prefix=~/orig_ffmpeg --enable-static --disable-asm
$ make
$ make install
```

编译 FFmpeg 命令：

```
    $ ./configure --prefix=~/orig_ffmpeg --enable-static --enable-small
--enable-memalign-hack --enable-gpl --enable-nonfree --enable-pthreads
--extra-ldflags=-L~/orig_ffmpeg/lib --extra-cflags=-I~/orig_ffmpeg/include
--enable-libx264 --disable-yasm
    $ make
    $ make install
```

关于手机屏幕截屏的测试代码，其工作流程如下图所示：

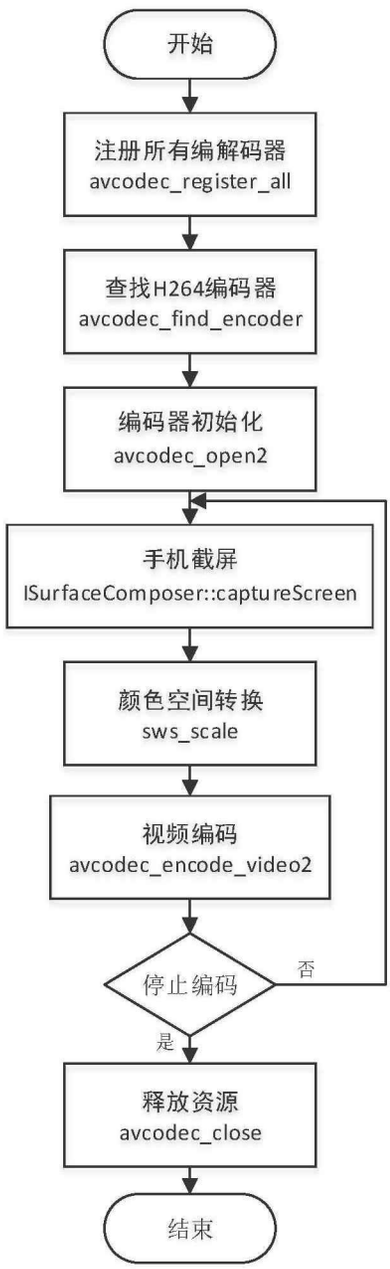

图 5.1 手机截屏编码流程

从图中可以看到，其编码的过程同时也体现了 FFmpeg 接口的调用过程。查找编码

器的过程实际是查找对应的 AVCodec 结构体对象，这一步是获得正确编码器的关键步骤。在使用 Android 硬件编码的情况下，avcodec_find_encoder 返回的 AVCodec 结构体对象就是在编码优化实现中出现的 ff_android_encoder。手机截屏使用了 Android 底层自带的接口 ISurfaceComposer，从该接口中截取到的屏幕数据格式为 rgba8888，需要调用 sws_scale 函数将其转换成 yuv420 格式，然后才能送进编码器编码。整个过程会不断地循环截屏，格式转换和编码，直到用户停止。

5.3 参数测试及结果分析

5.3.1 测试参数

为了比较 FFmpeg 编码优化前后的运行性能，提出了几个重要的性能评判参数，其中包括：单帧编码的平均时间，CPU 平均使用率，内存平均占用率，视频帧的 PSNR 值。对于这些参数的获取过程，如下所述：

（1） 单帧编码的平均时间

在编码一帧视频的开始和结束位置通过 gettimeofday 函数分别获取到本地时间，两个时间相减得到每一帧的编码时间，再多次测试求平均值，从而最终获取到单帧编码的平均时间

（2） CPU 平均使用率和内存平均占用率

通过 adb 命令进入 Android 手机的命令行终端，在编码进行过程中执行以下命令：

```
$ su
# top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
```

top 是一个动态监控系统运行状态的命令，能够将总体的和每一个进程的 CPU 使用时间和内存使用率等信息显示出来。“-d”表示信息刷新的间隔，这里设置为 1 秒；“-n”表示刷新次数，这里设置为 10 次。grep 为查找命令，这里用于查找编码进程 mediastream 的运行参数信息，“-e PID”是用来保留标识这些参数名称的那一行。sort 命令和 uniq 命令联合使用，可以将相同的信息删除，只保留一份，这里用来将多余的参数名称行去掉。

执行完上述命令后，会得到 10 组关于 mediastream 程序运行状态信息，包括 CPU 使用率和内存使用率。将这 10 组的数据求平均值，得到最后的 CPU 平均使用率和内存

平均使用率。

（3） 视频帧的 PSNR 值

为比较编码优化前后的视频编码质量，本文从源 YUV 格式视频和编码后再解码的视频中各提取 5 帧视频帧，计算 PSNR 值，然后再取平均值。解码时，使用了 FFmpeg 命令：

```
ffmpeg.exe -i encoded.h264 out.yuv
```

PSNR 的计算使用了 ComputePSNR 工具。该过程在 Windows 环境中完成。

5.3.2 测试结果

为得到测试结果的正确性和有效性，本论文使用了三种不同的分辨率进行测试，分别为 144x176,288x362,480x800。测试结果如下所示：

1. 144x176 分辨率

（1） 优化前的编码参数测试

```
.>>>>encode frame 1 cost 10159(us)
.>>>>encode frame 2 cost 12193(us)
.>>>>encode frame 3 cost 9112(us)
.>>>>encode frame 4 cost 6231(us)
.>>>>encode frame 5 cost 8950(us)
.>>>>encode frame 6 cost 9171(us)
.>>>>encode frame 7 cost 7956(us)
.>>>>encode frame 8 cost 6938(us)
.>>>>encode frame 9 cost 13864(us)
.>>>>encode frame 10 cost 13978(us)
```

图 5-2 144x176 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID   PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
28743 17830 0         R    36388  4.2   0 32.0 ./mediastream 1600
28743 17830 0         R    36964  4.3   1 17.3 ./mediastream 1600
28743 17830 0         R    36968  4.3   0 14.3 ./mediastream 1600
28743 17830 0         R    36980  4.3   0 13.0 ./mediastream 1600
28743 17830 0         R    36980  4.3   0 15.9 ./mediastream 1600
28743 17830 0         R    36988  4.3   0 17.3 ./mediastream 1600
28743 17830 0         S    36368  4.2   0 19.5 ./mediastream 1600
28743 17830 0         S    36372  4.2   0 13.1 ./mediastream 1600
28743 17830 0         S    36372  4.2   0 17.2 ./mediastream 1600
28743 17830 0         S    36388  4.2   0 10.0 ./mediastream 1600
```

图 5-3 144x176 分辨率的 CPU 和内存使用率

图 5-4 编码前后的视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	52.424316	52.421391	52.431639
1	52.424316	52.421391	52.431639
2	52.474354	52.421391	52.44338
3	53.327268	53.097592	53.069718
4	54.972928	54.737594	54.592559
Average	53.698622	53.5794925	53.512099

表 5-2 PSNR 值

（2） 优化后的编码参数测试

```
.>>>>encode frame 1 cost 2106(us)
.>>>>encode frame 2 cost 1007(us)
.>>>>encode frame 3 cost 1160(us)
.>>>>encode frame 4 cost 1282(us)
.>>>>encode frame 5 cost 1404(us)
.>>>>encode frame 6 cost 1618(us)
.>>>>encode frame 7 cost 702(us)
.>>>>encode frame 8 cost 854(us)
.>>>>encode frame 9 cost 1434(us)
.>>>>encode frame 10 cost 915(us)
```

图 5-6 144x176 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID   PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
30270 17830 0         R    17213  2.0   0 16.2 ./mediastream 1600
30270 17830 0         R    17504  2.0   0 17.5 ./mediastream 1600
30270 17830 0         R    17504  2.0   1 16.9 ./mediastream 1600
30270 17830 0         S    17213  2.0   1 16.3 ./mediastream 1600
30270 17830 0         S    17504  2.0   0 16.8 ./mediastream 1600
30270 17830 0         S    17512  2.0   0 16.1 ./mediastream 1600
30270 17830 0         S    17512  2.0   1 16.5 ./mediastream 1600
30270 17830 0         S    17700  2.0   0 17.1 ./mediastream 1600
30270 17830 0         S    18051  2.1   0 16.4 ./mediastream 1600
30270 17830 0         S    18051  2.1   0 17.1 ./mediastream 1600
```

图 5-7 144x176 分辨率的 CPU 和内存使用率

图 5-8 编码前后的视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	55.220872	54.852056	54.848836
1	55.312138	54.949801	54.903913
2	55.352621	55.006182	54.972928
3	55.361669	55.009522	54.972928
4	55.361669	55.009522	54.972928
Average	55.2912705	54.930789	54.910882

表 5-3 PSNR 值

2. 288x352 分辨率

（1） 优化前的编码参数测试

```
.>>>>encode frame 1 cost 31209(us)
.>>>>encode frame 2 cost 35950(us)
.>>>>encode frame 3 cost 34464(us)
.>>>>encode frame 4 cost 32528(us)
.>>>>encode frame 5 cost 57141(us)
.>>>>encode frame 6 cost 38196(us)
.>>>>encode frame 7 cost 37767(us)
.>>>>encode frame 8 cost 27336(us)
.>>>>encode frame 9 cost 36247(us)
.>>>>encode frame 10 cost 56733(us)
```

图 5-10 288x352 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID   PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
29447 17830 0         R     37712  4.4   0 32.2 ./mediastream 1600
29447 17830 0         R     38488  4.5   0 19.7 ./mediastream 1600
29447 17830 0         R     38488  4.5   1 24.0 ./mediastream 1600
29447 17830 0         R     38508  4.5   0 24.7 ./mediastream 1600
29447 17830 0         S     37688  4.4   0 26.6 ./mediastream 1600
29447 17830 0         S     37712  4.4   0 16.5 ./mediastream 1600
29447 17830 0         S     37712  4.4   1 18.7 ./mediastream 1600
29447 17830 0         S     37736  4.4   1 25.6 ./mediastream 1600
29447 17830 0         S     37760  4.4   1 20.5 ./mediastream 1600
29447 17830 0         S     37760  4.4   1 22.6 ./mediastream 1600
```

图 5-11 288x352 分辨率的 CPU 和内存使用率

图 5-12 编码前后视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	47.885777	48.462944	48.244872
1	47.914091	48.462944	48.244872

2	47.914091	48.466645	48.248744
3	48.491106	48.699575	48.495623
4	49.82287	49.837731	49.766697
Average	48.8543235	49.1503375	49.0057845

表 5-4 PSNR 值

（2） 优化后的编码参数测试

```
.>>>>encode frame 1 cost 3326(us)
.>>>>encode frame 2 cost 1801(us)
.>>>>encode frame 3 cost 3601(us)
.>>>>encode frame 4 cost 2930(us)
.>>>>encode frame 5 cost 2259(us)
.>>>>encode frame 6 cost 1709(us)
.>>>>encode frame 7 cost 1679(us)
.>>>>encode frame 8 cost 1587(us)
.>>>>encode frame 9 cost 1617(us)
.>>>>encode frame 10 cost 1586(us)
```

图 5-14 288x352 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID   PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
30455 17830 0         R    17985  2.1   1 19.5 ./mediastream 1600
30455 17830 0         R    18151  2.1   1 22.9 ./mediastream 1600
30455 17830 0         R    18151  2.1   1 24.3 ./mediastream 1600
30455 17830 0         R    18444  2.1   0 17.1 ./mediastream 1600
30455 17830 0         R    18444  2.1   0 20.5 ./mediastream 1600
30455 17830 0         S    18444  2.1   0 18.5 ./mediastream 1600
30455 17830 0         S    18444  2.1   0 19.3 ./mediastream 1600
30455 17830 0         S    18444  2.1   0 20.8 ./mediastream 1600
30455 17830 0         S    18444  2.1   0 25.3 ./mediastream 1600
30455 17830 0         S    18513  2.2   1 22.8 ./mediastream 1600
```

图 5-15 288x352 分辨率的 CPU 和内存使用率

图 5-16 编码前后的视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	50.081311	50.947331	50.900654
1	50.245289	50.953638	50.011106
2	50.65658	51.359166	51.338702
3	50.763758	51.478158	51.408446
4	50.824852	51.537118	51.480501
Average	50.4530815	51.2422245	51.1905775

表 5-5 PSNR 值

3. 480x800 分辨率

（1） 优化前的编码参数测试

```
.>>>>encode frame 1 cost 118498(us)
.>>>>encode frame 2 cost 69056(us)
.>>>>encode frame 3 cost 72113(us)
.>>>>encode frame 4 cost 85363(us)
.>>>>encode frame 5 cost 100625(us)
.>>>>encode frame 6 cost 102929(us)
.>>>>encode frame 7 cost 101038(us)
.>>>>encode frame 8 cost 104116(us)
.>>>>encode frame 9 cost 146568(us)
.>>>>encode frame 10 cost 94169(us)
```

图 5-18 480x800 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID   PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
29760 17830 0         R     46112  5.4   1 49.9 ./mediastream 1600
29760 17830 0         R     46176  5.4   1 18.9 ./mediastream 1600
29760 17830 0         R     48244  5.6   0 28.8 ./mediastream 1600
29760 17830 0         R     48244  5.6   1 18.6 ./mediastream 1600
29760 17830 0         R     48244  5.6   1 25.8 ./mediastream 1600
29760 17830 0         S     46112  5.4   0 22.0 ./mediastream 1600
29760 17830 0         S     46112  5.4   1 24.1 ./mediastream 1600
29760 17830 0         S     46176  5.4   1 11.0 ./mediastream 1600
29760 17830 0         S     46180  5.4   1 13.5 ./mediastream 1600
29760 17830 0         S     46244  5.4   0 21.1 ./mediastream 1600
```

图 5-19 480x800 分辨率的 CPU 和内存使用率

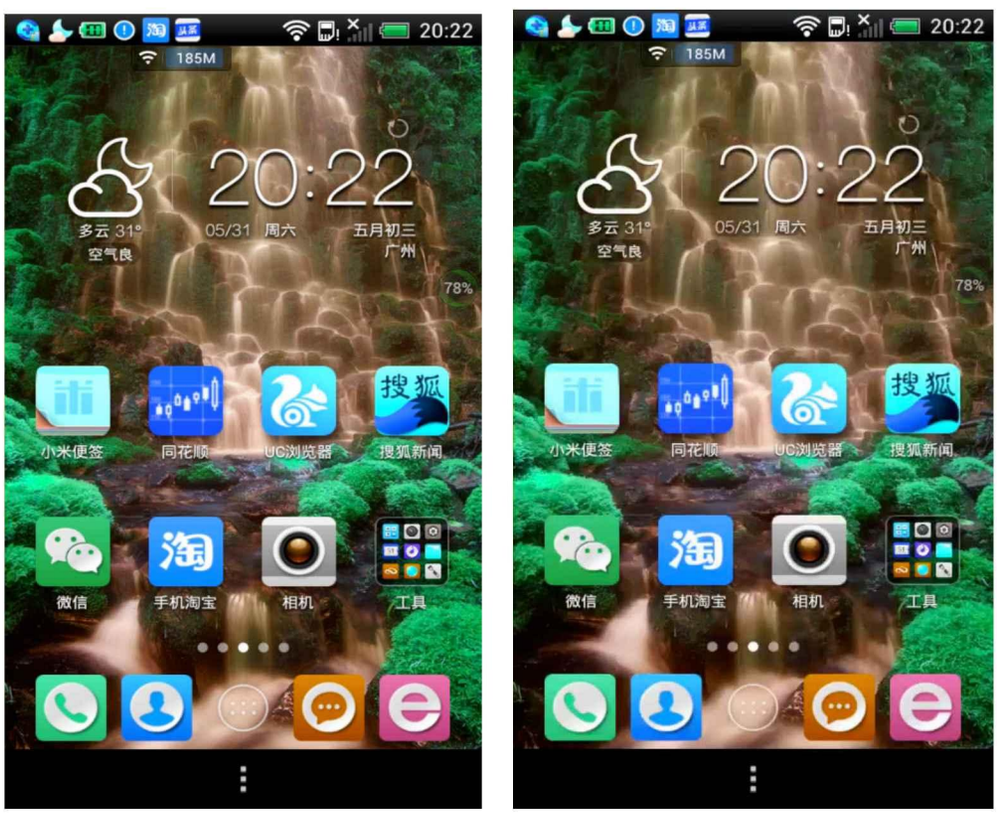

图 5-20 编码前后视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	35.735602	38.301113	39.782386
1	35.735602	38.301113	39.782386
2	35.737347	38.302525	39.783921
3	35.800647	38.363527	39.85701
4	38.051856	39.242886	40.532473
Average	36.893729	38.7719995	40.1574295

表 5-6 PSNR 值

（2） 优化后的编码参数测试

```
.>>>>encode frame 1 cost 8545(us)
.>>>>encode frame 2 cost 3875(us)
.>>>>encode frame 3 cost 4486(us)
.>>>>encode frame 4 cost 3998(us)
.>>>>encode frame 5 cost 4303(us)
.>>>>encode frame 6 cost 3997(us)
.>>>>encode frame 7 cost 4303(us)
.>>>>encode frame 8 cost 4577(us)
.>>>>encode frame 9 cost 4334(us)
.>>>>encode frame 10 cost 3967(us)
```

图 5-22 480x800 分辨率的单帧编码时间

```
root@android:/ # top -b -d 1 -n 10 | grep -e mediastream -e PID | sort | uniq
  PID  PPID USER      STAT   VSZ %MEM CPU %CPU COMMAND
30721 17830 0         R    25720  3.0   0 15.4 ./mediastream 1600
30721 17830 0         R    25720  3.0   0 23.1 ./mediastream 1600
30721 17830 0         R    25720  3.0   0 23.3 ./mediastream 1600
30721 17830 0         R    26050  3.0   0 20.1 ./mediastream 1600
30721 17830 0         S    24877  2.9   0 20.1 ./mediastream 1600
30721 17830 0         S    25662  3.0   0 22.3 ./mediastream 1600
30721 17830 0         S    25720  3.0   0 21.4 ./mediastream 1600
30721 17830 0         S    25720  3.0   1 22.2 ./mediastream 1600
30721 17830 0         S    25720  3.0   1 24.1 ./mediastream 1600
30721 17830 0         S    26050  3.0   0 20.5 ./mediastream 1600
```

图 5-23 480x800 分辨率的 CPU 和内存使用率

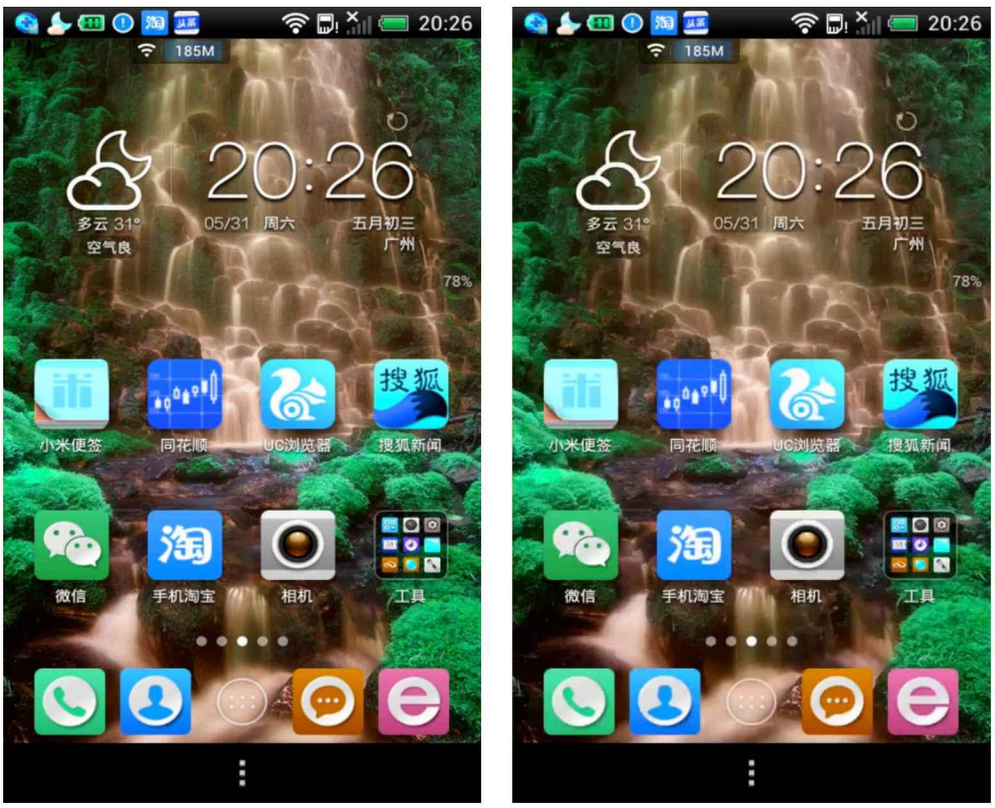

图 5-24 编码前后视频帧对比

FrameNUM	PSNR_Y	PSNR_U	PSNR_V
0	38.033717	39.283334	40.568794
1	39.488739	40.377392	41.419799
2	39.501565	40.38698	41.425166
3	39.972003	40.873133	41.73569
4	40.251521	41.091176	41.854468
Average	39.142619	40.187255	41.211631

5-7 PSNR 值

5.3.3 测试分析

将上述数据整理成表格如下所示：

分辨率	方案	单帧编码平均时间	编码效率（帧/秒）	平均占用率（%）	内存平均占用率（%）	平均PSNR-Y（dB）	平均PSNR-U（dB）	平均PSNR-V（dB）
		(us)						
144x176	优化前	9855	101.5	17.0	4.3	53.698622	53.5794925	53.512099
	优化后	1248	801.3	16.7	2.0	55.2912705	54.930789	54.910882
288x352	优化前	38757	25.8	23.1	4.4	48.8543235	49.1503375	49.0057845
	优化后	2210	452.5	21.1	2.1	50.4530815	51.2422245	51.1905775
480x800	优化前	99448	10.1	23.4	5.5	36.893729	38.7719995	40.1574295
	优化后	4639	215.6	21.3	3.0	39.142619	40.187255	41.211631

图 5-8 测试数据整合表

对测试数据的分析如下：

（1） 优化前后的视频质量，从人体视觉上来看，几乎没有差别。从计算的 PSNR 值来看，优化后的视频质量要比优化前的稍微好一些，但也可以说这样的差别是微乎其微的。所以总体看来，优化前后的视频质量是一致的。

（2） 优化后的编码效率远高于优化前的编码效率，消耗的内存也同样比未优化时的低。造成该结果的原因是优化后的编码运用了 Android 中的 H264 硬件编码器，而未优化时使用的是 libx264 软件编码器，软件编码要比硬件编码消耗更多的系统资源。

（3） 另外，我们看到 CPU 的平均占用率无论是横向还是纵向比较，都相差不大（横向比较是指相同分辨率下不同测试方案的比较，纵向比较是指相同测试方案下不同分辨率的比较）。这是因为测试程序在运行期间循环地进行编码操作，中间没有任何挂起进程的操作，使得 CPU 一直处于高负荷工作状态中。优化后的编码效率虽然很高，但却在一秒钟内要比未优化时编码更多的视频帧，因此导致了 CPU 占用率一直居高不下。但在实际应用中，必然会控制编码的速率，因此，对于优化后的编码，CPU 占用率并不会达到表中的数值。

通过以上结果分析，我们可以得出以下结论：通过对 FFmpeg 的编码优化设计，在不影响其编码质量的前提下，实现了 FFmpeg 在 Android 平台下的高效编码，事实表明优化后的 FFmpeg 更能适用于 Android 平台上。

5.4 本章小结

本章为了验证 FFmpeg 在 Android 多媒体平台下进行优化后的编码性能，特意将其与优化前进行了比较，设计并实现了手机截取屏幕并编码成 H264 视频流的测试程序。通过单帧编码的平均时间，CPU 使用率和内存占用率三个方面来说明优化后的编码性能要优于未优化的编码性能，说明了整合后的 FFmpeg 编码库可以很好地应用于 Android 平台。

第六章 总结与展望

6.1 总结

FFmpeg 是目前应用最为广泛的开源音视频编解码解决方案，市场上非常多的视频播放器都是基于 FFmpeg 进行设计和编写的。FFmpeg 的成功不仅得益于其优秀的代码设计架构和丰富的编解码格式，同时也因为它采用了纯 C 语言，能够应用于各种各样的平台上。Android 是当今最为流行的移动操作系统之一，其开源特性使得越来越多的开发者投向了 Android 阵型。而随着移动互联网的急速发展，人们对移动互联网的载体--手机要求越来越高，日常生活中使用它来看各种视频资讯的人们也逐渐增多，娱乐应用也都在往人与人之间的可视化通信上如火如荼地发展着。可以毫不夸张地说，掌握了 Android 多媒体的开发技术，就等于掌握了时代的发展脉络。

本文主要从源码级别上研究了 FFmpeg 和 Android 多媒体的框架，并对各自的优缺点进行了分析。FFmpeg 虽然应用广泛，但由于是软件编码导致在 Android 平台上的运行效率低下；Android 多媒体能够利用硬件高效编码，但却无法跟上多媒体应用发展的需求，例如目前还不具备视频通话的功能。正是由于这些优缺点，才有了将 FFmpeg 和 Android 多媒体整合在一起优化编码的想法。在研究和优化的过程中，本论文做了以下几个方面的工作：

（1） 对 Android 底层源码进行了深入学习，掌握了 Binder 通信机制的原理。它是 Android 多媒体得以正常工作的关键所在。Binder 是 Android 中广泛使用的一种远程调用接口，专门应用于 Android 内部的进程间通信。它的设计采用了 C/S 结构，包含了 Binder Client，Binder Server 和 Binder 驱动三个部分。

（2） Android 多媒体中使用了 OpenMAX IL 标准，硬件编码器也都是按照该标准进行设计和实现的。在 Android 中，OMX 作为 Binder Server 运作着，任何需要与之通信的请求都需要通过一个 Binder Client 来进行。为了方便上层使用，Android 特意将所有与 OMX 通信的请求都封装到了 OMXCodec 类中，只暴露了非常简单的几个接口。OMX 在工作时最重要的一个步骤就是数据源的获取，为此使用者需要提供一个 MediaSource 子类对象，该对象可以是从已有的容器封装类中提取出来，也可以由

使用者自己创建。OMX 会循环地从该对象中获取数据，然后处理，并将结果数据返回给使用者。

（3） FFmpeg 具有非常明确的功能划分，这从它的目录结构就可以看出来。FFmpeg 的工作过程也非常地清晰明了，包含了以下几个方面：初始化，容器解封装，解码，编码，容器封装，释放资源。

（4） FFmpeg 和 Android 多媒体整合的关键是要在深刻理解两者的工作机制基础上，把握两者相同的部分，从中提取出优化后的整体架构。接着还需要兼容不同的部分，为此考虑了使用队列来存储数据，创建子线程来与 OMX 通信。

6.2 展望

在编码优化的设计中，都遵循了设计尽可能简单方便，外部调用接口保持不变的原则。但仍然还有可以优化的地方，具体可以从以下几个方面进行考虑：

（1） 减少 FFmpeg 和 Android 多媒体之间的数据传递。原有的优化过程中，使用的是 OMX 非隧道工作方式。在这种方式下，数据源需要上层来提供，这里的上层就是指 FFmpeg，因此数据就会在两者之间相互拷贝，增加了内存的损耗。如果采用 OMX 的隧道工作方式将工作流程中相邻的两个组件串起来，使得数据无需经过 FFmpeg。例如在视频转码的例子中将解码组件得到的数据以隧道的方式传递给编码组件，这样可以大大节省内存的使用。这样做需要考虑的重点是如何设计才能够保证 FFmpeg 的调用接口不变。

（2） 从目前优化的设计来看，Android 多媒体下的每一个编解码器都需要单独做成 FFmpeg 中的一个插件。但这些插件的设计过程都是相同的，如果可以设计出一个通用的插件，通过传入不同的参数信息来获得不同的编解码器，那么将会大大提高开发的效率，节约研发的时间成本。

（3） 可以考虑为 FFmpeg 的通用接口设计 JNI（Java Native Interface），这样除了 C 语言开发人员可以用之来开发之外，同时也方便 Java 开发人员使用。

参考文献

[1] 报告称 Android 中国市场份额增至 78.1%. http://tech.sina.com.cn/t/2013-12-02/19028968408.shtml

[2] 童方圆,于强. 基于 Android 的实时视频流传输系统[J]. 计算机工程与设计, 2012,33(12):4639-4642.

[3] 孙俊辉. 基于 Android 平台的多媒体框架研究与实现[D]. 成都:电子科技大学, 2012.

[4] 白璐. Android 系统多媒体功能增强的研究与实现[D]. 西安:西安科技大学，2012.

[5] 温伟, 刘荣科. Android 多媒体框架下 StageFright 的功能扩展[J]. 太赫兹科学与电子信息学报,2013,11(5):718-723.

[6] 苗忠良. Android 多媒体编程从初学到精通[M]. 北京:电子工业出版社,2010.

[7] 韩超. Android 系统原理及开发要点详解[M]. 北京:电子工业出版社,2010.

[8] 胡成, 任平安, 李文莉. 基于 Android 系统的 FFmpeg 多媒体同步传输算法研究. 西安:计算机技术与发展,2011(10):98-101.

[9] 叶炳发. Android 操作系统移植与关键技术研究[D]. 广州:暨南大学,2010.

[10] 公磊攻擂, 周聪. 基于 Android 的移动终端应用程序开发与研究[J]. 计算机系统应用, 2008(11).

[11] 张仕成. 基于 Google Android 平台的应用程序开发与研究[J]. 电脑知识与技术,2009,5(28):7959-7962.

[12] Android Binder 通讯机制. http://blog.sina.com.cn/s/blog_a43898d001018v0f.html

[13] 邓凡平. 深入理解 Android[C]. 北京:机械工业出版社,2011:131-158.

[14] Maoqiang Song, Wenkuo Xiong, Xiangling Fu. Research on Architecture of Multi media and Its Design Based on Android[C]. International Conference on Internet T echnology and Applications, ITAP 2010 - Proceedings,2010. United States：IEEE Computer Societ,2010.

[15] Maoqiang Song, Jie Sun,, Xiangling Fu, Wenkuo Xiong. Design and Implementati on of Media Player Based on Android[C].2010 6th International Conference on Wi reless Communications, Networking and Mobile Computing, 2010. United States：I

EEE Computer Societ,2010.

[16] 零崇伟. 基于 OpenCore 多媒体框架的应用扩展[M]. 西安:西安电子科技大学,2011.

[17] openmax_il_spec_1_0.pdf. http://www.khronos.org/registr/omxil.

[18] 李晓凤. Android 下视频解码组件的研究与设计. 浙江:浙江工业大学,2012.

[19] OpenMax 在 Android 上的实现. http://blog.csdn.net/yuyin86/article/details/7107704.

[20] 郭亮. OpenMAX IL 的研究与应用[D]. 青岛:中国海洋大学,2012.

[21] 和 OpenMAX 的運作. http://blog.csdn.net/zjc0888/article/details/6279657.

[22] 胡成, 任平安, 李文莉. 基于 Android 系统的 FFmpeg 多媒体同步传输算法研究. 西安:计算机技术与发展,2011(10):98-101.

[23] FFmpeg. http://zh.wikipedia.org/zh-cn/FFmpeg.

[24] 马洪堂. 基于 FFMPEG 的视频转换系统[D]. 浙江:浙江工业大学,2009

[25] 李刚. 基于 Android 平台的只能手机流媒体播放器的研究及实现[M]. 南京:南京邮电大学,2012.

[26] 和 OpenMAX 的運作. http://blog.csdn.net/zjc0888/article/details/6279657.

[27] FFmpeg 开发工程组. FFmpeg 开发手册. http://www.ffmpeg.com.cn.

[28] 刘建敏, 杨斌. 嵌入式 Linux 下基于 FFmpeg 的视频硬件编解码[J]. 单片机与嵌入式系统应用,2011,(6):28-31.

[29] 覃艳. 基于 FFMPEG 的视频格式转换技术研究. 电脑知识与技术,2011,(12):111-112.

[30] Downloading the Source. http://source.android.com/source/downloading.html.

[31] 韩超, 梁泉. Android 系统原理及开发要点详解[M]. 北京:电子工业出版社,2010.

攻读硕士学位期间取得的研究成果

一、已发表（包括已接受待发表）的论文，以及已投稿、或已成文打算投稿、或拟成文投稿的论文情况**（只填写与学位论文内容相关的部分）：**

序号	作者（全体作者，按顺序排列）	题　　目	发表或投稿刊物名称、级别	发表的卷期、年月、页码	相当于学位论文的哪一部分（章、节）	被索引收录情况

注：在“发表的卷期、年月、页码”栏：
1 如果论文已发表，请填写发表的卷期、年月、页码；
2 如果论文已被接受，填写将要发表的卷期、年月；
3 以上都不是，请据实填写“已投稿”，“拟投稿”。
不够请另加页。

二、与学位内容相关的其它成果（包括专利、著作、获奖项目等）

致 谢

弹指之间，研究生生涯即将画上了句号，这一切来得太突然，去得也太突然了。回想过往生活和工作的点点滴滴，都历历在目，仿佛是昨天才发生的事情。3 年的求学生涯，我付出了汗水，却收获了满满的学识和经验，这一切都要感谢我的授业恩师吴宗泽老师。他用自身的奋斗经历鼓舞着我，让我变得更加有勇气去面对困难，克服困难。知识渊博的他常常在学术上给与我很大的指导，甚至于有时候他的一句话就能让我看清问题的本质，有一种豁然开朗的感觉。而在生活上他却是那么的平易近人，以至于我们能够互相玩笑取乐。吴宗泽老师，您是一位毫无架子却能令人信服的人，是我们的良师益友。在此，我向您致以最衷心的感谢。

此外，我也要感谢傅予力老师，周智恒老师，李波老师和向有君老师，他们的敬业精神和学术造诣让我深感敬佩，一直是我学习的榜样。

接着，我还要感谢实验室的所有兄弟姐妹们，吴容、耀城、曾星、李爽、段伊竹、啟成、罗涛、石清、国坚、赖文华、东凯、妍蓉等等，因为有你们的努力才组建起了一个这么优秀的团队。是你们让我感觉到了即使在外头也能拥有家的温暖，谢谢你们 3 年来的陪伴。我将会永记这份真挚情谊，并以此作为我不断前行的动力。

最后，我要感谢我的爸爸妈妈和两位姐姐，谢谢你们无私的付出。因为有你们的宽容和支持，我才能取得今天的成绩，谢谢你们一路上的陪伴。

在这里，我再一次感谢所有关心我的人，祝你们万事顺心，芝麻开花--节节高。

华耀波

二零一四年五月

Ⅳ - 2答辩委员会对论文的评定意见

华耀波同学的硕士学位论文《FFmpeg 在 Android 多媒体平台下的编码优化研究》以时下较为热门的多媒体应用为背景，对 Android 多媒体的工作原理和 FFmpeg 视频编解码器进行了深入研究，结合 Android 底层的硬件编码器对 FFmpeg 进行了优化，实现了高效编码。论文选题适当，论文工作具有广泛的应用价值。

论文的主要工作包括：

（1） 从源码级别上深入分析了 Android 多媒体的框架设计以及使用 Binder 机制通信的原理，重点阐述了多媒体框架下使用 OpenMAX 标准进行编码的工作过程。

（2） 在深入研究了 FFmpeg 和 Android 多媒体工作原理的基础上，分析比较了 FFmpeg 和 Android 多媒体工作过程的异同点，从中总结出优化后的代码设计框架，并将 Android 多媒体中的硬件编码器设计成插件的形式供 FFmpeg 使用。

（3） 设计了一个手机屏幕截屏编码的测试方案来比较优化前后的编码效率，结果验证了优化后的 FFmpeg 能在 Android 平台下高效编码。

论文工作反映作者具有较扎实的专业技术基础，具有较强的科研工作能力。论文结构清晰，论述严谨，逻辑性强，已达到硕士学位论文的要求。在答辩过程中表述清晰，回答问题基本正确，经答辩委员会无记名投票表决，一致同意通过华耀波的硕士学位论文答辩，建议授予其工学硕士学位。

论文答辩日期：2014 年 6 月 7 日

答辩委员会委员共 5 人，到会委员 5 人

表决票数：优秀（0）票；良好（5）票；及格（0）票；不及格（0）票

表决结果（打“√”）：优秀（ ）；良好（√）；及格（ ）；不及格（ ）

决议：同意授予硕士学位（√） 不同意授予硕士学位（ ）

答辩委员会成员签名			
	（主席）		