HZ BOOKS
华章IT

音视频开发进阶指南

基于Android与iOS平台的实践

展晓凯　魏晓红◎著

凝聚资深专家多年经验结晶和最佳实践，深入解析音视频开发技术
理论结合实战，全方面多角度指导读者快速深入开发，大幅提高学习和工作效率

机械工业出版社
China Machine Press

音视频开发进阶指南：基于Android与iOS平台的实践

展晓凯　魏晓红　著

ISBN：978-7-111-58582-4

本书纸版由机械工业出版社于2018年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

推荐序一
推荐序二
前言
第1章　音视频基础概念
1.1　声音的物理性质
1.1.1　声音是波
1.1.2　声波的三要素
1.1.3　声音的传播介质
1.1.4　回声
1.1.5　共鸣
1.2　数字音频
1.3　音频编码
1.4　图像的物理现象
1.5　图像的数值表示
1.5.1　RGB表示方式
1.5.2　YUV表示方式
1.5.3　YUV和RGB的转化
1.6　视频的编码方式
1.6.1　视频编码
1.6.2　编码概念
1.7　本章小结
第2章　移动端环境搭建
2.1　在iOS上如何搭建一个基础项目
2.2　在Android上如何搭建一个基础项目
2.3　交叉编译的原理与实践
2.3.1　交叉编译的原理
2.3.2　iOS平台交叉编译的实践
2.3.3　Android平台交叉编译的实践
2.3.4　使用LAME编码MP3文件
2.4　本章小结
第3章　FFmpeg的介绍与使用
3.1　FFmpeg的编译与命令行工具的使用
3.1.1　FFmpeg的编译
3.1.2　FFmpeg命令行工具的使用
3.2　FFmpeg　API的介绍与使用

3.3　FFmpeg源码结构
3.3.1　libavformat与libavcodec介绍
3.3.2　FFmpeg通用API分析
3.3.3　调用FFmpeg解码时用到的函数分析
3.3.4　调用FFmpeg编码时用到的函数分析
3.3.5　面向对象的C语言设计
3.4　本章小结
第4章　移动平台下的音视频渲染
4.1　AudioUnit介绍与实践
4.2　Android平台的音频渲染
4.2.1　AudioTrack的使用
4.2.2　OpenSL　ES的使用
4.3　视频渲染
4.3.1　OpenGL　ES介绍
4.3.2　OpenGL　ES的实践
4.3.3　上下文环境搭建
4.3.4　OpenGL　ES中的纹理
4.4　本章小结
第5章　实现一款视频播放器
5.1　架构设计
5.2　解码模块的实现
5.3　音频播放模块的实现
5.3.1　Android平台的音频渲染
5.3.2　iOS平台的音频渲染
5.4　画面播放模块的实现
5.4.1　Android平台的视频渲染
5.4.2　iOS平台的视频渲染
5.5　AVSync模块的实现
5.5.1　维护解码线程
5.5.2　音视频同步
5.6　中控系统串联起各个模块
5.6.1　初始化阶段
5.6.2　运行阶段
5.6.3　销毁阶段
5.7　本章小结
第6章　音视频的采集与编码
6.1　音频的采集

6.1.1　Android平台的音频采集
6.1.2　iOS平台的音频采集
6.2　视频画面的采集
6.2.1　Android平台的视频画面采集
6.2.2　iOS平台的视频画面采集
6.3　音频的编码
6.3.1　libfdk_aac编码AAC
6.3.2　Android平台的硬件编码器MediaCodec
6.3.3　iOS平台的硬件编码器AudioToolbox
6.4　视频画面的编码
6.4.1　libx264编码H264
6.4.2　Android平台的硬件编码器MediaCodec
6.4.3　iOS平台的硬件编码器
6.5　本章小结
第7章　实现一款视频录制应用
7.1　视频录制的架构设计
7.2　音频模块的实现
7.2.1　音频队列的实现
7.2.2　Android平台的实现
7.2.3　iOS平台的实现
7.3　音频编码模块的实现
7.3.1　改造编码器
7.3.2　编码器适配器
7.4　画面采集与编码模块的实现
7.4.1　视频队列的实现
7.4.2　Android平台画面编码后入队
7.4.3　iOS平台画面编码后入队
7.5　Mux模块
7.5.1　初始化
7.5.2　封装和输出
7.5.3　销毁资源
7.6　中控系统串联起各个模块
7.7　本章小结
第8章　音频效果器的介绍与实践
8.1　数字音频基础
8.1.1　波形图
8.1.2　频谱图

8.1.3　语谱图
8.1.4　深入理解时域与频域
8.2　数字音频处理：快速傅里叶变换
8.3　基本乐理知识
8.3.1　乐谱
8.3.2　音符的音高与十二平均律
8.3.3　音符的时值
8.3.4　节拍
8.3.5　MIDI格式
8.4　混音效果器
8.4.1　均衡效果器
8.4.2　压缩效果器
8.4.3　混响效果器
8.5　效果器实现
8.5.1　Android平台实现效果器
8.5.2　iOS平台实现效果器
8.6　本章小结
第9章　视频效果器的介绍与实践
9.1　图像处理的基本原理
9.1.1　亮度调节
9.1.2　对比度调节
9.1.3　饱和度调节
9.2　图像处理进阶
9.2.1　图像的卷积过程
9.2.2　锐化效果器
9.2.3　高斯模糊算法
9.2.4　双边滤波算法
9.2.5　图层混合介绍
9.3　使用FFmpeg内部的视频滤镜
9.3.1　FFmpeg视频滤镜介绍
9.3.2　滤镜图的构建
9.3.3　使用与销毁滤镜图
9.3.4　常用滤镜介绍
9.4　使用OpenGL　ES实现视频滤镜
9.4.1　加水印
9.4.2　添加自定义文字
9.4.3　美颜效果器

9.4.4　动图贴纸效果器
9.4.5　主题效果器
9.5　本章小结
第10章　专业的视频录制应用实践
10.1　视频硬件解码器的使用
10.1.1　初始化信息准备
10.1.2　VideoToolbox解码H264
10.1.3　MediaCodec解码H264
10.2　音频效果器的集成
10.2.1　Android音效处理系统的实现
10.2.2　iOS音效处理系统的实现
10.3　一套跨平台的视频效果器的设计与实现
10.4　将特效处理库集成到视频录制项目中
10.4.1　Android平台特效集成
10.4.2　iOS平台特效集成
10.5　本章小结
第11章　直播应用的构建
11.1　直播场景分析
11.2　拉流播放器的构建
11.2.1　Android平台播放器增加后处理过程
11.2.2　iOS平台播放器增加后处理过程
11.3　推流器的构建
11.4　第三方云服务介绍
11.5　礼物系统的实现
11.5.1　Cocos2dX项目的运行原理
11.5.2　关键API详解
11.5.3　实现一款动画
11.6　聊天系统的实现
11.6.1　Android客户端的WebSocket实现
11.6.2　iOS客户端的WebSocket实现
11.7　本章小结
第12章　直播应用中的关键处理
12.1　直播应用的细节分析
12.1.1　推流端细节分析
12.1.2　拉流端细节分析
12.2　推流端的关键处理
12.2.1　自适应码率的实践

12.2.2　统计数据保证后续的应对策略
12.3　拉流端的关键处理
12.3.1　重试机制的实践
12.3.2　首屏时间的保证
12.3.3　统计数据保证后续的应对策略
12.4　本章小结
第13章　工欲善其事，必先利其器
13.1　Android平台工具详解
13.1.1　ADB工具的熟练使用
13.1.2　MAT工具检测Java端的内存泄漏
13.1.3　NDK工具详解
13.1.4　Native层的内存泄漏检测
13.1.5　breakpad收集线上Crash
13.2　iOS使用Instruments诊断应用
13.2.1　Debug　Navigator
13.2.2　Time　Profiler
13.2.3　Allocations
13.2.4　Leaks
13.3　本章小结
附录A　通过Ne10的交叉编译输入理解ndk-build
附录B　编码器的使用细节
附录C　视频的表示与编码

推荐序一

我的第一部智能手机是多普达565，当时使用的是Windows Mobile操作系统，现在看来，不管是操作交互，还是系统的整体能力，都与今天的智能手机有着天壤之别。但是，即便是那样的操作系统，也已经足够让当时的我认识到一个真正的操作系统能给一部随身设备赋予的强大能力。智能手机后来这十几年的发展还是超出了很多人的预料，很难想象，如果没有现在的高速数据网络和每个人手头的这个小终端，我们的工作和生活会有多少不方便的地方。

生活在这个时代的程序员是足够幸运的，信息化的无限渗透也意味着有想法、有能力的程序员对人们生活范围的影响越来越大。我与很多资深的开发人员都有过交流，基本上能把这些人分成两类。一类是以对技术本身的钻研为目标的技术人员，他们所关注的是架构是不是足够先进，可扩展性如何，系统整体的负载能力，遇到错误时的鲁棒性等。总之，他们内心的成就感来自是否把技术做到了极致，同行（或者自己）看到的时候，会不会由衷地说这东西真棒。还有一类技术人员，他们的成就感来自自己的工作成果是否能够直接对使用者产生影响。相对技术本身的挑战，这类人更在乎自己所做的东西是否真正被身边的人使用，使用者用到自己作品时的感受，以及是否真正给使用者和社会带来了帮助。两类人没有高低之分，倒有点像理论研究和应用研究的关系，两个方向相辅相成，彼此成就，彼此推动。

音视频技术的发展正好处在理论和应用的十字路口。各种音视频技术天生就与老百姓的生活距离很近，拍照、唱歌、小视频、瘦脸、美颜、大混音，基本上算是大众手机里最常用的一些功能了。这些功能背后的技术，也会因用户的需求推动而快速发展。从软件到硬件，从各种人脸识别的算法到越来越强大的摄像头或是专用的DSP芯片，摩尔定律在这个细分领域的发挥可以算是淋漓尽致了，这也对有志于在这个领域发展的研发人员提出了更高的要求：一方面，要能沉得下去，音视频相关的底层技术可以说是CS领域里相当难啃的一块硬骨头，对算法、编码甚至是数学基础都有很高的要求；另一方面，还要能经常抬起头，不只是要跟上相关领域的快速发展，也要理解和挖掘用户的真实需求，这可以算是CS领域里挑战很大同时成就感也很大的困难模式了。

本书的作者展晓凯是音视频领域的权威专家。在几年间的持续研究

中，他总结出了一套在音视频领域比较系统的工程实践方法，希望这些总结能够帮助到对相关领域感兴趣的你。如果能进一步影响更多的人，将是对本书作者最大的鼓励和褒奖。

田然

2017年9月于北京

推荐序二

随着智能手机的出现，音视频传感器比以往任何时候都更加接近用户，可以说，移动互联网时代其实也是音视频内容爆炸的时代。几乎每个应用都希望能尽可能地开启用户所有的权限，让用户把自己的每时每刻都上传和分享出来，于是即时聊天IM几乎走进了每一个App。而这一两年的直播大战的背后，更是让直播这种富媒体几乎变成了各类App的标配功能，以至于市场对音视频研发人才求之若渴，音视频的学习材料也一时洛阳纸贵，由此催生出来的各种大小公司、大小产业也是层出不穷，业内对音视频处理的知识和经验的分享更是如火如荼。然而，音视频处理实在是一门艰深的学问，从傅里叶变换到差分编码等各种理论，再到充满差异化甚至Bug的安卓设备的现实应用环境，既要在国内复杂的网络环境下尽力满足用户视听观感的流畅感觉，又希望让小小的移动设备物尽其用来为用户提供最极致的感官体验，而这些灼烧无数程序员脑细胞的问题，实在不是一两篇长文就可以简单讲清楚的。若要让一个几无基础的开发者能够系统地掌握音视频的处理知识，确有必要撰写一本图书并通过具体实例来详细讲解才行。

我认识本书的作者展晓凯已经五年多了，他几乎是唱吧最勤奋的技术人员，在唱吧浩瀚的代码库里，到处都留有他的成果。唱吧是中国颇具影响力的K歌产品之一，从2012年雄踞苹果榜首之后，就再也没有离开过榜单，多年来其为用户创造了无数新奇的功能和体验，从各种振奋人心的混响，到节奏感逼人的自动说唱，所有的这些功能，都是出自展晓凯和他所在的三五个人的小团队之手。而以唱吧的用户体量，他们自然也遇到了许多问题，从各种花屏、黑屏、啸叫、白噪等通用技术问题，到用户手机上因形形色色系统或硬件差异而产生的稀奇古怪的问题，也无一不是展晓凯所在的团队逐个去解决的。可以说，发展到今天，唱吧几乎拥有业内最丰富的音视频处理经验，而这些经验中的精华，如今终于有机会整理付梓，实在是唱吧的一点骄傲，也是业内同行的一份福音。

虽然与展晓凯一起并肩战斗了四五年，但我本人并没有太多机会详细参与每一个音视频问题的处理，如今通读完这本书的原稿，我深感收获不小。本书是展晓凯花费了无数心血的作品，其中的每一行代码每一个实例都来自他日常工作中实际问题的总结。本书也许不是市面上唯一一本关于音视频处理的著作，但它的出现，足以为市场带来一个特有的

完整的视角，并令无数致力于打造移动设备上音视频处理完美体验的程序员受益！

黄全能

2017年9月于北京

前言

为什么要写这本书

整个音视频领域的架构以及开发已经演进了很长时间，从最开始的广电领域，到PC端的音视频领域，再到本书所介绍的移动端的音视频领域。尤其在这几年中，移动端音视频领域架构的变化是巨大的。在移动互联网的发展热潮中，我有幸从事了音视频领域的设计与开发，并且就职于最时尚的手机KTV——唱吧，这使得我开发出来的东西能够服务于几亿用户。对于音视频的移动端的应用，不论是开发还是使用，在近两年都达到了一个高峰，而作为一名工程师，如何高效地开发出一个音视频App，是一件非常困难的事情，特别是对于不太了解音视频概念的工程师。我从事软件开发已有7年多的时间，接触音视频领域也已经有5年多，在整个开发过程中，不同的时间段会遇到不同的挑战，尤其是在最开始涉足音视频领域的时候，真可谓举步维艰。首先，对于音视频的基础概念不是特别清楚，再者在工作中边学边做，很难对整个音视频领域有一个全面的了解，并且市面上没有相关成熟的资料从更高的层次来介绍音视频领域在移动端的演进与发展。这几年的设计实战与开发经验，以及带新人入门的众多感触，让我有了写这本书的动力，同时也形成了这本书的核心内容，我希望通过本书可以帮助更多想要在移动端音视频领域实现自己想法的工程师，让大家可以顺利地建立起自己的音视频App。我非常希望能为刚入门的读者或者遇到困难的读者提供帮助，希望大家可以享受整个开发的过程，享受自己开发的产品为人们的生活带来便利的成就感。另外，从整个音视频开发领域来讲，我也十分希望能够通过本书贡献出自己的绵薄之力。

读者对象

·产品经理，这部分读者可以从中了解在移动端进行音视频开发会遇到的很多问题以及对应的优化策略，例如：如何通过音视频的统计数据为产品提供更加流畅的策略（视频观看的秒开、直播推流的流畅度、视频上传的成功率等）。

·项目经理，这部分读者可以了解很多时下流行的名词与概念，不再会因为几个专业名词就让自己不知所措，并且有助于更好地评估音视频项目开发中的风险与进度。

·测试人员，这部分读者可以学习在音视频App中由于处理过程不同而导致的瓶颈问题，书中也提到了一些自动化测试相关的命令以及工具，可以对CPU的负载情况、内存的占用情况、内存泄漏问题等进行分析。

·架构师与工程师，这部分读者只需要一点移动开发经验就可以阅读本书了。当然如果你已经是一个高级移动开发工程师或者架构师，那么读起本书来将更加游刃有余。再进一步，如果你已经是移动领域的音视频开发工程师了，那么恭喜你，我们之间将会有一场关于技术领域内部的对话。

·开设相关课程的高等院校。

如何阅读本书

为了避免说教式的讲解带来枯燥乏味的阅读体验，本书给出了大量的实例及生产环境下的案例。本书可分为四个部分：第一部分是入门，从理论基础开始讲解，最终会产生两个实践项目；第二部分是提高，基于第一部分的项目添加特效，形成一个完整的多媒体项目；第三部分是扩展，结合当下比较流行的直播场景进行实际案例分析；第四部分是工具，介绍当下大部分可以提高开发以及测试效率的工具。下面是各个章节的基本介绍。

第1章，介绍音视频的基础概念，其中包括音视频的基础数据格式、编码后的数据格式以及不同格式之间的相互转换等。

第2章，从零开始讲解如何搭建一个iOS项目和一个Android项目，并且添加C++支持，因为在音视频领域的开发中，有相当一部分的代码需要用C++来编写，这样就可以做到两个平台（Android和iOS平台）共用一套代码仓库，以提升开发效率。然后讲解交叉编译，因为在音视频开发过程中会用到很多第三方开源库，如果将这些库编译到我们的项目中，势必要进行交叉编译，因此本章会重点讲解这些内容。

第3章，探讨FFmpeg开源库。对于音视频开发来讲，FFmpeg开源库是众所周知也是普遍使用的。本章首先从编译开始，接着是命令行使用，再到源码结构，最后是API调用，以层层递进的方式对FFmpeg开源库展开介绍。

第4章，讲解如何利用各自平台的API进行声音与画面的渲染以及解码，对于画面的渲染，推荐使用OpenGL ES，两个平台可以使用同一个代码仓库。

第5章，实现一款视频播放器。有了前四章的基础，我们已经完全可以构建起一个视频播放器了。本书最大的特点就是经过几章基础知识的学习立即开始一个项目的实践，通过本章的视频播放器项目，我们将会熟悉播放器是如何工作的。

第6章，重点介绍音视频的采集与编码器。特别是硬件编解码器在各个平台上的使用，使得应用能够更高效（耗电更少、发热更少、界面更流畅）地运行在用户的手机上。

第7章，继续开发一个视频录制的新项目，该项目可以使我们更加熟悉音视频应用在各个平台下的实现。

第8章，讲解如何处理音频流。毕竟让别人听采集出来的干声是很不礼貌的，本章将利用各种特效来美化采集的声音。

第9章，讲解如何处理视频流，使视频中的颜值变得更高，毕竟爱美之心人皆有之。

第10章，在第7章的项目基础之上，增加第8章的音频特效和第9章的视频特效，从而构建一个实际生产过程中的多媒体应用。

第11章，继续以项目作为驱动，详细讲解如何基于之前学习的内容构建一个直播的应用，重点介绍推流以及拉流端，同时还涉及礼物特效、聊天以及第三方云服务的内容。

第12章，由于直播应用很难用一章的篇幅讲完，所以本章针对一些核心的处理进行讲解。

第13章，介绍常用的工具和排错方法，说明在日常开发中如何更有

效率地解决问题，本章内容并不仅限于音视频的开发领域。

附录给出一些参考内容。

勘误和支持

由于作者水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，我特意创建了一个在线支持与应急方案的二级站点http://music-video.cn 。你可以将书中的错误发布在Bug勘误表页面中，同时如果你遇到问题，也可以访问Q&A页面，我将尽量在线上为你提供最满意的解答。书中的全部源代码文件都将发布在这个网站上，我也会及时地进行相应的功能更新。如果你有更多的宝贵意见，也欢迎发送邮件至我的邮箱zhanxiaokai2008@126.com，我很期待听到你们的真挚反馈。

致谢

感谢唱吧与唱吧的每一位同事，是这个公司让我的职业生涯发展到了今天，也是这个公司让我能在音视频领域达到今天的成就，可以说没有唱吧就不会有这本书的问世。

感谢唱吧的每一位用户，感谢你们对唱吧的长期支持和贡献，没有你们就不会有唱吧的今天，也就不会有这本书的问世。

感谢我的老婆，感谢你对我工作以及写作的支持，是你在我背后默默做了很多事情，才让我把更多的时间和精力放到工作以及写作中。

感谢机械工业出版社华章公司的编辑Lisa老师，感谢你的魄力和远见，在这一年多的时间中始终支持我的写作，正是你的鼓励和帮助引导我顺利完成全部书稿。

感谢互联网，我们在互联网上迈出的任何一步都是人类历史向前迈进的一步，感谢众多互联网人的辛苦工作，为我们创造了这么多机遇。

谨以此书献给我最亲爱的家人、同事，以及众多互联网从业者。

展晓凯

2017年9月于北京

第1章　音视频基础概念

为了避免枯燥的说教式讲解，本章将结合示意图来介绍音频与视频的基础概念，对于本章的学习，不需要使用任何开发环境。研究数字音频时，必须要对声学现象有一定的了解，若只研究数字音频而忽略了声学现象，那就本末倒置了，因为音频技术是为了记录、存储和回放声学现象才发明的，所以先了解声学现象对学习数字音频是有很大帮助的。声音产生于自然界，早在没有任何科学研究的时候，就已经存在声音了，并且各种声音还可以组成动听的音乐；当人类有了记录以及存储声音的能力之后，就迎来了模拟信号到数字信号的转换，所以本章首先会介绍如何记录以及存储声音。

相比声音，视频（画面）更易于观察，本章将从图像的物理现象开始讲解，然后讨论一帧帧的画面是如何描述的，以及视频是如何被记录和存储到设备中的。

学习完本章之后，相信大家对耳朵能够直接听到的声音，眼睛能够直接看到的图像会有更深入的认知。

1.1　声音的物理性质

1.1.1　声音是波

说到声音，爱好音乐的人首先可能会想到优美的音乐或者是劲爆十足的舞曲，这些音乐只是声音的一种。音乐是由乐器弹奏或者歌手演唱而产生的，那么声音是如何产生的呢？回想一下中学物理课本上的定义——声音是由物体振动而产生的（如图1-1所示）。

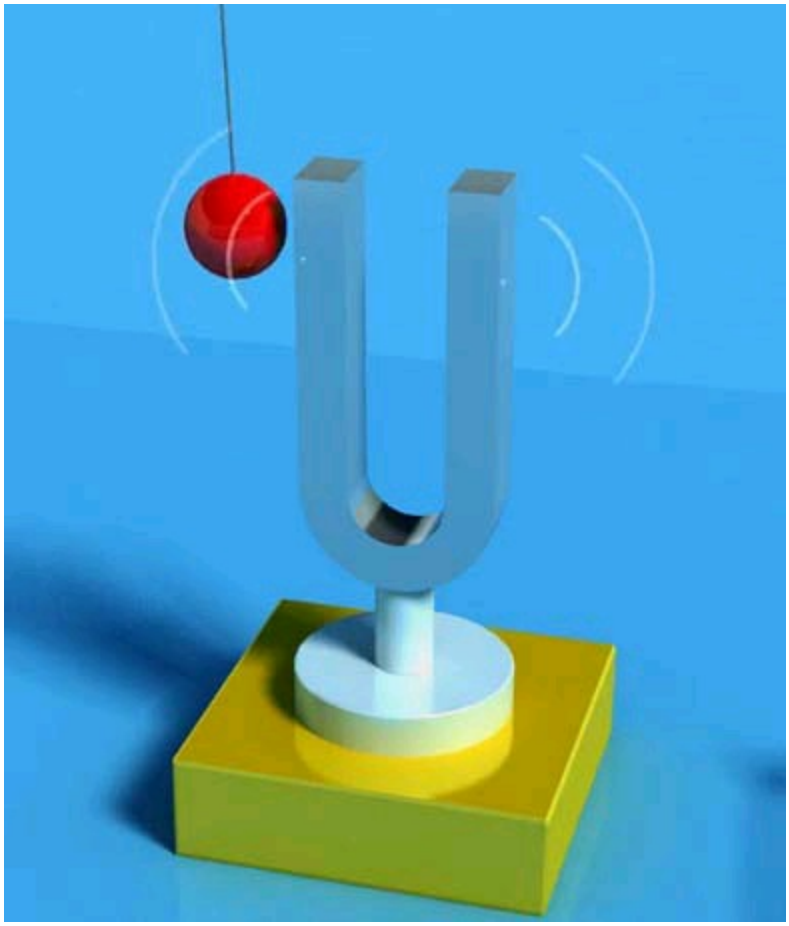


图　1-1

如图1-1所示，当小球撞击到音叉的时候，音叉会发生振动，对周围的空气产生挤压，从而产生声音。声音是一种压力波，当演奏乐器、拍打一扇门或者敲击桌面时，它们的振动都会引起空气有节奏的振动，使周围的空气产生疏密变化，形成疏密相间的纵波（可以理解为石头落

入水中激起的波纹），由此就产生了声波，这种现象会一直延续到振动消失为止。

1.1.2　声波的三要素

声波的三要素是频率、振幅和波形，频率代表音阶的高低，振幅代表响度，波形代表音色。

频率（过零率）越高，波长就越短。低频声响的波长则较长，所以其可以更容易地绕过障碍物，因此能量衰减就小，声音就会传得远，反之则会得到完全相反的结论。

响度其实就是能量大小的反映，用不同的力度敲击桌子，声音的大小势必也会不同。在生活中，分贝常用于描述响度的大小。声音超过一定的分贝，人类的耳朵就会受不了。

音色其实也不难理解，在同样的音调（频率）和响度（振幅）下，钢琴和小提琴的声音听起来是完全不相同的，因为它们的音色不同。波的形状决定了其所代表声音的音色，钢琴和小提琴的音色不同就是因为它们的介质所产生的波形不同。

人类耳朵的听力有一个频率范围，大约是20Hz～20kHz，不过，即使是在这个频率范围内，不同的频率，听力的感觉也会不一样，业界非常著名的等响曲线，就是用来描述等响条件下声压级与声波频率关系的，如图1-2所示。

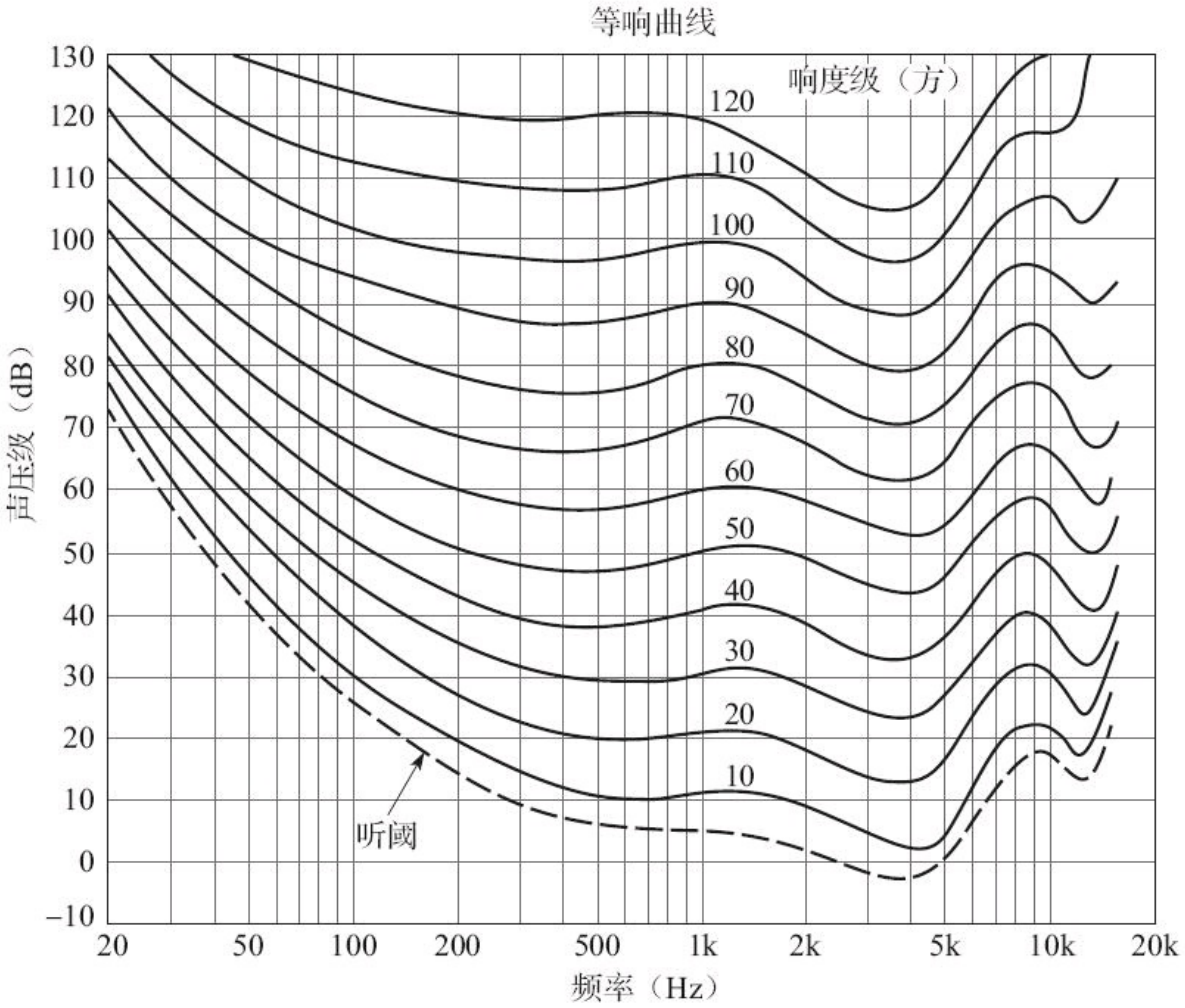


图　1-2

从图1-2中可以看出，人耳对3～4kHz频率范围内的声音比较敏感，而对于较低或较高频率的声音，敏感度就会有所减弱；在声压级较低时，听觉的频率特性会很不均匀；而在声压级较高时，听觉的频率特性会变得较为均匀。频率范围较宽的音乐，其声压以80～90dB为最佳，超过90dB将会损害人耳（105dB为人耳极限）。

1.1.3　声音的传播介质

吉他是通过演奏者拨动琴弦来发出声音的，鼓是通过鼓槌敲击鼓面发出声音的，这些声音的产生都离不开振动，就连我们说话也是因为声带振动而产生声音的。既然都是振动产生的声音，那为什么吉他、鼓和人声听起来相差这么大呢？这是因为介质不同。我们的声带振动发出声音之后，经过口腔、颅腔等局部区域的反射，再经过空气传播到别人的耳朵里，这就是我们说的话被别人听到的过程，其中包括了最初的发声介质与颅腔、口腔，还有中间的传播介质等。事实上，声音的传播介质很广，它可以通过空气、液体和固体进行传播；而且介质不同，传播的速度也不同，比如，声音在空气中的传播速度为340m/s，在蒸馏水中的传播速度为1497m/s，而在铁棒中的传播速度则可以高达5200m/s；不过，声音在真空中是无法传播的。

生活小贴士

在日常生活中，我们也会利用对声音的研究去做一些使我们更舒适的事情，比如吸音棉和隔音棉，这两种常见产品的发明就是通过研究声音在传播中的特性而研发出来的。

吸音主要是解决声音反射而产生的嘈杂感，吸音材料可以衰减入射音源的反射能量，从而达到对原有声源的保真效果，比如录音棚里面的墙壁上就会使用吸音棉材料。

隔音主要是解决声音的透射而降低主体空间内的吵闹感，隔音棉材料可以衰减入射音源的透射能量，从而达到主体空间的安静状态，比如KTV里面的墙壁上就会安装隔音棉材料。

1.1.4　回声

当我们在高山或空旷地带高声大喊的时候，经常会听到回声（echo）。之所以会有回声是因为声音在传播过程中遇到障碍物会反弹回来，再次被我们听到（如图1-3所示）。

但是，若两种声音传到我们的耳朵里的时差小于80毫秒，我们就无法区分开这两种声音了，其实在日常生活中，人耳也在收集回声，只不过由于嘈杂的外界环境以及回声的分贝（衡量声音能量值大小的单位）比较低，所以我们的耳朵分辨不出这样的声音，或者说是大脑能接收到但分辨不出。

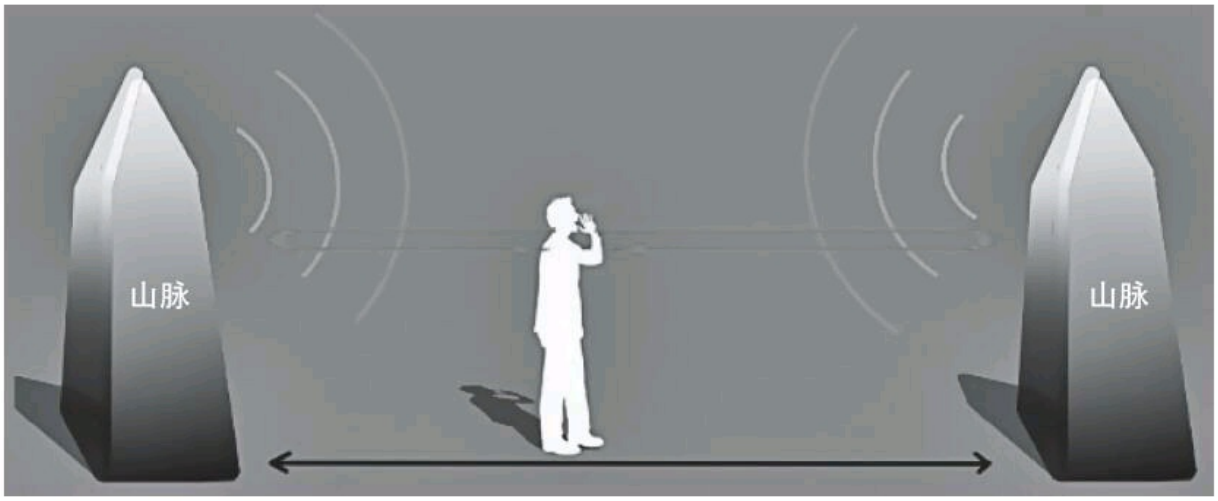


图　1-3

1.1.5　共鸣

自然界中有光能、水能，生活中有机械能、电能，其实声音也可以产生能量，例如两个频率相同的物体，敲击其中一个物体时另一个物体也会振动发声（如图1-4所示）。

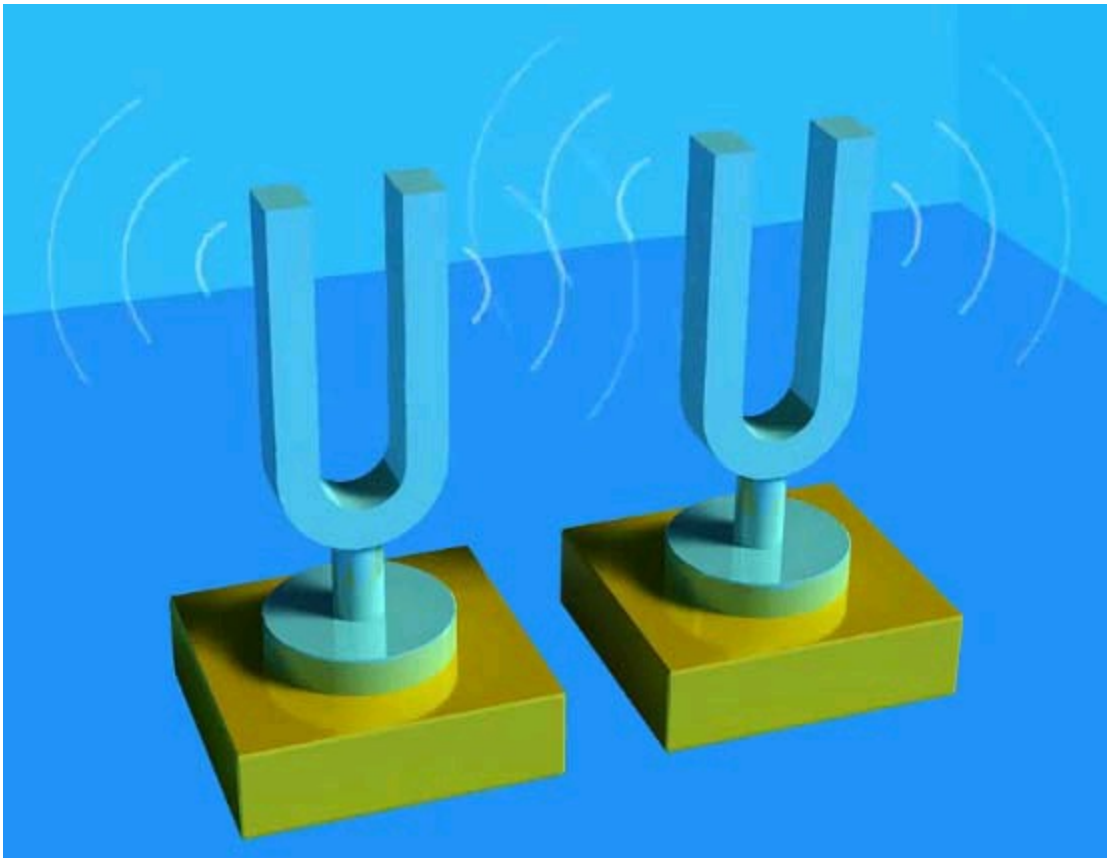


图　1-4

这种现象称为共鸣，共鸣证明了声音传播可以带动另一个物体振动，也就是说，声音的传播过程也是一种能量的传播过程。

1.2　数字音频

1.1节主要介绍了声音的物理现象以及声音中常见的概念，也为后续的讲解统一了术语，从本节开始，我们将进入数字音频概念的介绍。

为了将模拟信号数字化，本节将分3个概念对数字音频进行讲解，分别是采样、量化和编码。首先要对模拟信号进行采样，所谓采样就是在时间轴上对信号进行数字化。根据奈奎斯特定理（也称为采样定理），按比声音最高频率高2倍以上的频率对声音进行采样（也称为AD转换），1.1节中提到过，对于高质量的音频信号，其频率范围（人耳能够听到的频率范围）是20Hz～20kHz，所以采样频率一般为44.1kHz，这样就可以保证采样声音达到20kHz也能被数字化，从而使得经过数字化处理之后，人耳听到的声音质量不会被降低。而所谓的44.1kHz就是代表1秒会采样44100次（如图1-5所示）。

那么，具体的每个采样又该如何表示呢？这就涉及将要讲解的第二个概念：量化。量化是指在幅度轴上对信号进行数字化，比如用16比特的二进制信号来表示声音的一个采样，而16比特（一个short）所表示的范围是[-32768，32767]，共有65536个可能取值，因此最终模拟的音频信号在幅度上也分为了65536层（如图1-6所示）。

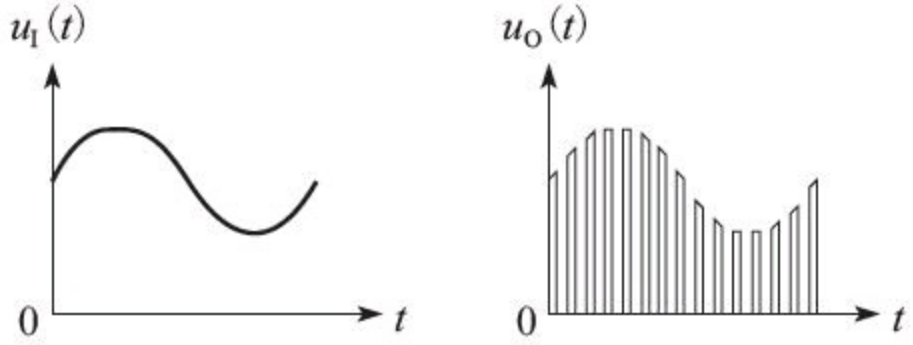


图　1-5

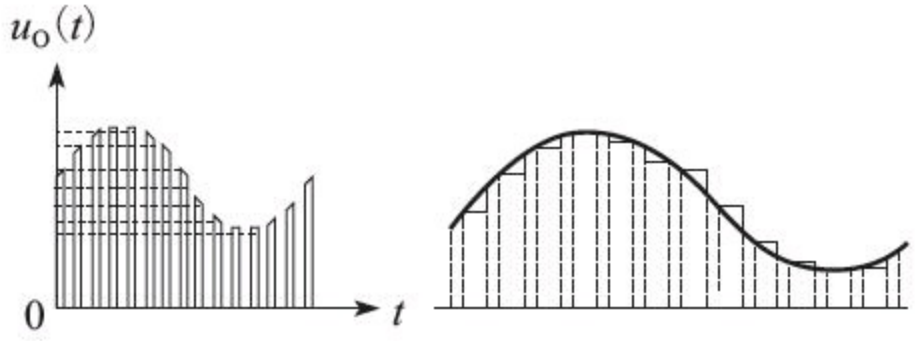


图　1-6

既然每一个量化都是一个采样，那么这么多的采样该如何进行存储呢？这就涉及将要讲解的第三个概念：编码。所谓编码，就是按照一定的格式记录采样和量化后的数字数据，比如顺序存储或压缩存储，等等。

这里面涉及了很多种格式，通常所说的音频的裸数据格式就是脉冲编码调制（Pulse Code Modulation，PCM）数据。描述一段PCM数据一般需要以下几个概念：量化格式（sampleFormat）、采样率（sampleRate）、声道数（channel）。以CD的音质为例：量化格式（有的地方描述为位深度）为16比特（2字节），采样率为44100，声道数为2，这些信息就描述了CD的音质。而对于声音格式，还有一个概念用来描述它的大小，称为数据比特率，即1秒时间内的比特数目，它用于衡量音频数据单位时间内的容量大小。而对于CD音质的数据，比特率为多少呢？计算如下：

```
44100 * 16 * 2 = 1378.125kbps
```

那么在1分钟里，这类CD音质的数据需要占据多大的存储空间呢？计算如下：

```
1378.125 * 60 / 8 / 1024 = 10.09MB
```

当然，如果sampleFormat更加精确（比如用4字节来描述一个采样），或者sampleRate更加密集（比如48kHz的采样率），那么所占的存储空间就会更大，同时能够描述的声音细节就会越精确。存储的这段

二进制数据即表示将模拟信号转换为数字信号了，以后就可以对这段二进制数据进行存储、播放、复制，或者进行其他任何操作。

麦克风是如何采集声音的

麦克风里面有一层碳膜，非常薄而且十分敏感。1.1节中介绍过，声音其实是一种纵波，会压缩空气也会压缩这层碳膜，碳膜在受到挤压时也会发出振动，在碳膜的下方就是一个电极，碳膜在振动的时候会接触电极，接触时间的长短和频率与声波的振动幅度和频率有关，这样就完成了声音信号到电信号的转换。之后再经过放大电路处理，就可以实施后面的采样量化处理了。

前面提到过分贝，那么什么是分贝呢？分贝是用来表示声音强度的单位。日常生活中听到的声音，若以声压值来表示，由于其变化范围非常大，可以达到六个数量级以上，同时由于我们的耳朵对声音信号强弱刺激的反应不是线性的（1.1节中提到过等响曲线），而是呈对数比例关系，所以引入分贝的概念来表达声学量值。所谓分贝是指两个相同的物理量（例如，A1和A0）之比取以10为底的对数并乘以10（或20），即：

```
N= 10 * lg（A1 / A0）
```

分贝符号为“dB”，它是无量纲的。式中A0是基准量（或参考量），A1是被量度量。

1.3　音频编码

1.2节中提到了CD音质的数据采样格式，曾计算出每分钟需要的存储空间约为10.1MB，如果仅仅是将其存放在存储设备（光盘、硬盘）中，可能是可以接受的，但是若要在网络中实时在线传播的话，那么这个数据量可能就太大了，所以必须对其进行压缩编码。压缩编码的基本指标之一就是压缩比，压缩比通常小于1（否则就没有必要去做压缩，因为压缩就是要减小数据容量）。压缩算法包括有损压缩和无损压缩。无损压缩是指解压后的数据可以完全复原。在常用的压缩格式中，用得较多的是有损压缩，有损压缩是指解压后的数据不能完全复原，会丢失一部分信息，压缩比越小，丢失的信息就越多，信号还原后的失真就会越大。根据不同的应用场景（包括存储设备、传输网络环境、播放设备等），可以选用不同的压缩编码算法，如PCM、WAV、AAC、MP3、Ogg等。

压缩编码的原理实际上是压缩掉冗余信号，冗余信号是指不能被人耳感知到的信号，包含人耳听觉范围之外的音频信号以及被掩蔽掉的音频信号等。人耳听觉范围之外的音频信号在1.2节中已经提到过，所以在此不再赘述。而被掩蔽掉的音频信号则主要是因为人耳的掩蔽效应，主要表现为频域掩蔽效应与时域掩蔽效应，无论是在时域还是频域上，被掩蔽掉的声音信号都被认为是冗余信息，不进行编码处理。

下面介绍几种常用的压缩编码格式。

（1）WAV编码

PCM（脉冲编码调制）是Pulse Code Modulation的缩写。前面已经介绍过PCM大致的工作流程，而WAV编码的一种实现（有多种实现方式，但是都不会进行压缩操作）就是在PCM数据格式的前面加上44字节，分别用来描述PCM的采样率、声道数、数据格式等信息。

特点：音质非常好，大量软件都支持。

适用场合：多媒体开发的中间文件、保存音乐和音效素材。

（2）MP3编码

MP3具有不错的压缩比，使用LAME编码（MP3编码格式的一种实现）的中高码率的MP3文件，听感上非常接近源WAV文件，当然在不同的应用场景下，应该调整合适的参数以达到最好的效果。

特点：音质在128Kbit/s以上表现还不错，压缩比比较高，大量软件和硬件都支持，兼容性好。

适用场合：高比特率下对兼容性有要求的音乐欣赏。

（3）AAC编码

AAC是新一代的音频有损压缩技术，它通过一些附加的编码技术（比如PS、SBR等），衍生出了LC-AAC、HE-AAC、HE-AAC v2三种主要的编码格式。LC-AAC是比较传统的AAC，相对而言，其主要应用于中高码率场景的编码（≥80Kbit/s）；HE-AAC（相当于AAC+SBR）主要应用于中低码率场景的编码（≤80Kbit/s）；而新近推出的HE-AAC v2（相当于AAC+SBR+PS）主要应用于低码率场景的编码（≤48Kbit/s）。事实上大部分编码器都设置为≤48Kbit/s自动启用PS技术，而>48Kbit/s则不加PS，相当于普通的HE-AAC。

特点：在小于128Kbit/s的码率下表现优异，并且多用于视频中的音频编码。

适用场合：128Kbit/s以下的音频编码，多用于视频中音频轨的编码。

（4）Ogg编码

Ogg是一种非常有潜力的编码，在各种码率下都有比较优秀的表现，尤其是在中低码率场景下。Ogg除了音质好之外，还是完全免费的，这为Ogg获得更多的支持打好了基础。Ogg有着非常出色的算法，可以用更小的码率达到更好的音质，128Kbit/s的Ogg比192Kbit/s甚至更高码率的MP3还要出色。但目前因为还没有媒体服务软件的支持，因此基于Ogg的数字广播还无法实现。Ogg目前受支持的情况还不够好，无论是软件上的还是硬件上的支持，都无法和MP3相提并论。

特点：可以用比MP3更小的码率实现比MP3更好的音质，高中低码率下均有良好的表现，兼容性不够好，流媒体特性不支持。

适用场合：语音聊天的音频消息场景。

1.4　图像的物理现象

在学习了音频的相关概念之后，现在开始讨论视频，视频是由一幅幅图像组成的，所以要学习视频还得从图像学习开始。

与音频的学习方法类似，视频的学习依然是从图像的物理现象开始回顾，这里需要回顾一下小学做过的三棱镜实验，还记得如何利用三棱镜将太阳光分解成彩色的光带吗？第一个做这个实验的人是牛顿，各色光因其所形成的折射角不同而彼此分离，就像彩虹一样，所以白光能够分解成多种色彩的光。后来人们通过实验证明，红绿蓝三种色光无法被分解，故称为三原色光，等量的三原色光相加会变为白光，即白光中含有等量的红光（R）、绿光（G）、蓝光（B）。

在日常生活中，由于光的反射，我们才能看到各类物体的轮廓及颜色。但是如果将这个理论应用到手机上，那么结论还是这个样子吗？答案是否定的，因为在黑暗中我们也可以看到手机屏幕上的内容，实际上人眼能看到手机屏幕上的内容的原理如下。

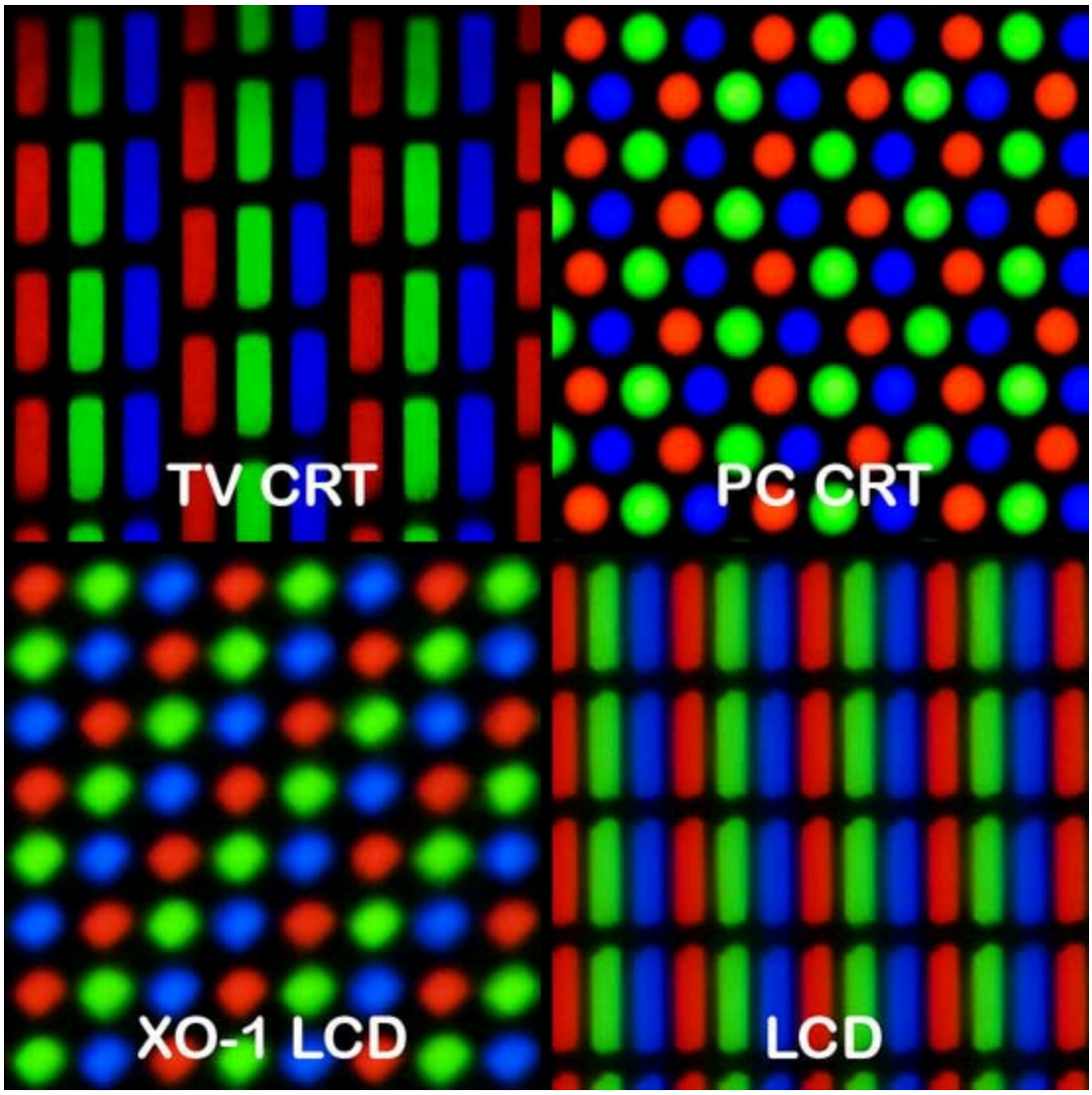


图　1-7

假设一部手机屏幕的分辨率是1280×720，说明水平方向有720个像素点，垂直方向有1280个像素点，所以整个手机屏幕就有1280×720个像素点（这也是分辨率的含义）。每个像素点都由三个子像素点组成（如图1-7所示），这些密密麻麻的子像素点在显微镜下可以看得一清二楚。当要显示某篇文字或者某幅图像时，就会把这幅图像的每一个像素点的RGB通道分别对应的屏幕位置上的子像素点绘制到屏幕上，从而显示整个图像。

所以在黑暗的环境下也能看到手机屏幕上的内容，是因为手机屏幕是自发光的，而不是通过光的反射才被人们看到的。

1.5　图像的数值表示

1.5.1　RGB表示方式

通过1.4节的讲解，我们已经知道任何一个图像都可以由RGB组成，那么一个像素点的RGB该如何表示呢？音频里面的每一个采样（sample）均使用16个比特来表示，那么像素里面的子像素又该如何表示呢？常用的表示方式有以下几种。

·浮点表示：取值范围为0.0～1.0，比如，在OpenGL ES中对每一个子像素点的表示使用的就是这种表达方式。

·整数表示：取值范围为0～255或者00～FF，8个比特表示一个子像素，32个比特表示一个像素，这就是类似于某些平台上表示图像格式的RGBA_8888数据格式。比如，Android平台上RGB_565的表示方法为16比特模式表示一个像素，R用5个比特来表示，G用6个比特来表示，B用5个比特来表示。

对于一幅图像，一般使用整数表示方法来进行描述，比如计算一张1280×720的RGBA_8888图像的大小，可采用如下方式：

```
1280 * 720 * 4 = 3.516MB
```

这也是位图（bitmap）在内存中所占用的大小，所以每一张图像的裸数据都是很大的。对于图像的裸数据来讲，直接在网络上进行传输也是不太可能的，所以就有了图像的压缩格式，比如JPEG压缩：JPEG是静态图像压缩标准，由ISO制定。JPEG图像压缩算法在提供良好的压缩性能的同时，具有较好的重建质量。这种算法被广泛应用于图像处理领域，当然其也是一种有损压缩。在很多网站如淘宝上使用的都是这种压缩之后的图片，但是，这种压缩不能直接应用于视频压缩，因为对于视频来讲，还有一个时域上的因素需要考虑，也就是说，不仅仅要考虑帧内编码，还要考虑帧间编码。视频采用的是更成熟的算法，关于视频压缩算法的相关内容将会在后续章节（1.6节）进行介绍。

1.5.2　YUV表示方式

对于视频帧的裸数据表示，其实更多的是YUV数据格式的表示，YUV主要应用于优化彩色视频信号的传输，使其向后兼容老式黑白电视。与RGB视频信号传输相比，它最大的优点在于只需要占用极少的频宽（RGB要求三个独立的视频信号同时传输）。其中“Y”表示明亮度（Luminance或Luma），也称灰阶值；而“U”和“V”表示的则是色度（Chrominance或Chroma），它们的作用是描述影像的色彩及饱和度，用于指定像素的颜色。“亮度”是透过RGB输入信号来建立的，方法是将RGB信号的特定部分叠加到一起。“色度”则定义了颜色的两个方面——色调与饱和度，分别用Cr和Cb来表示。其中，Cr反映了RGB输入信号红色部分与RGB信号亮度值之间的差异，而Cb反映的则是RGB输入信号蓝色部分与RGB信号亮度值之间的差异。

之所以采用YUV色彩空间，是因为它的亮度信号Y和色度信号U、V是分离的。如果只有Y信号分量而没有U、V分量，那么这样表示的图像就是黑白灰度图像。彩色电视采用YUV空间正是为了用亮度信号Y解决彩色电视机与黑白电视机的兼容问题，使黑白电视机也能接收彩色电视信号，最常用的表示形式是Y、U、V都使用8个字节来表示，所以取值范围就是0～255。在广播电视系统中不传输很低和很高的数值，实际上是为了防止信号变动造成过载，因而把这“两边”的数值作为“保护带”，不论是Rec.601还是BT.709的广播电视标准中，Y的取值范围都是16～235，UV的取值范围都是16～240。

YUV最常用的采样格式是4：2：0，4：2：0并不意味着只有Y、Cb而没有Cr分量。它指的是对每行扫描线来说，只有一种色度分量是以2：1的抽样率来存储的。相邻的扫描行存储着不同的色度分量，也就是说，如果某一行是4：2：0，那么其下一行就是4：0：2，再下一行是4：2：0，以此类推。对于每个色度分量来说，水平方向和竖直方向的抽样率都是2：1，所以可以说色度的抽样率是4：1。对非压缩的8比特量化的视频来说，8×4的一张图片需要占用48字节的内存（如图1-8所示）。

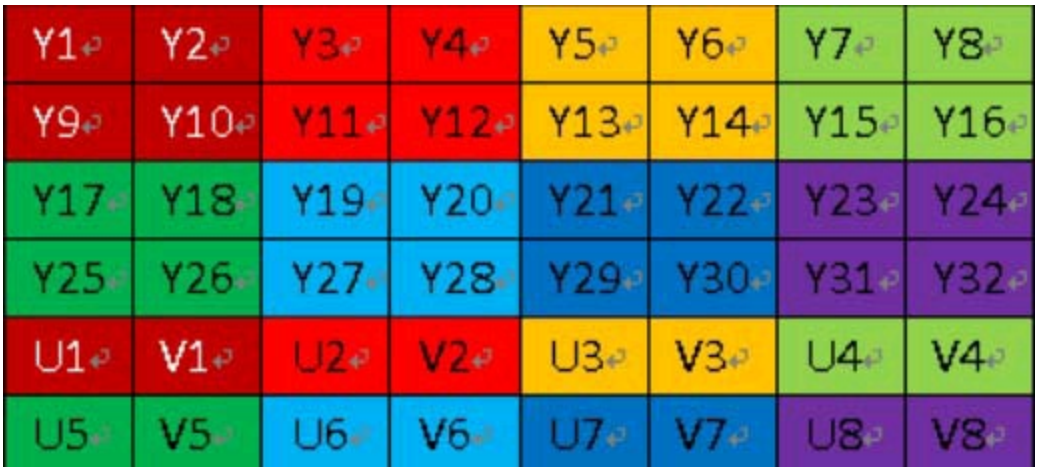


图　1-8

相较于RGB，我们可以计算一帧为1280×720的视频帧，用YUV420P的格式来表示，其数据量的大小如下：

```
1280 * 720 * 1 + 1280 * 720 * 0.5 = 1.318MB
```

如果fps（1秒的视频帧数目）是24，按照一般电影的长度90分钟来计算，那么这部电影用YUV420P的数据格式来表示的话，其数据量的大小就是：

```
1.318MB * 24fps * 90min * 60s = 166.8GB
```

所以仅用这种方式来存储电影肯定是不可行的，更别说在网络上进行流媒体播放了，那么如何对电影进行存储以及流媒体播放呢？答案是需要进行视频编码，下一节将会讨论视频的编码。

1.5.3　YUV和RGB的转化

前面已经讲过，凡是渲染到屏幕上的东西（文字、图片或者其他），都要转换为RGB的表示形式，那么YUV的表示形式和RGB的表示形式之间是如何进行转换的呢？对于标清电视601标准，它从YUV转换到RGB的公式与高清电视709的标准是不同的，通过如下的计算（如图1-9和图1-10）即可得知。

标清电视使用标准BT.601

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

图　1-9

高清电视使用标准BT.709

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.09991 & -0.33609 & 0.436 \\ 0.615 & -0.55861 & -0.05639 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.28033 \\ 1 & -0.21482 & -0.38059 \\ 1 & 2.12798 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

图　1-10

那么什么时候该用哪一种转换呢？比较典型的场景是在iOS平台中使用摄像头采集出YUV数据之后，上传显卡成为一个纹理ID，这个时候就需要做YUV到RGB的转换（具体的细节会在后面的章节中详细讲解）。在iOS的摄像头采集出一帧数据之后（CMSampleBufferRef），我们可以在其中调用CVBufferGetAttachment来获取YCbCrMatrix，用于决定使用哪一个矩阵进行转换，对于Android的摄像头，由于其是直接纹理ID的回调，所以不涉及这个问题。其他场景下需要大家自行寻找对应的文档，以找出适合的转换矩阵进行转换。

1.6　视频的编码方式

1.6.1　视频编码

还记得前面讨论的音频压缩方式吗？音频压缩主要是去除冗余信息，从而实现数据量的压缩。那么对于视频压缩，又该从哪几方面来对数据进行压缩呢？其实与前面提到的音频编码类似，视频压缩也是通过去除冗余信息来进行压缩的。相较于音频数据，视频数据有极强的相关性，也就是说有大量的冗余信息，包括空间上的冗余信息和时间上的冗余信息。

使用帧间编码技术可以去除时间上的冗余信息，具体包括以下几个部分。

·运动补偿：运动补偿是通过先前的局部图像来预测、补偿当前的局部图像，它是减少帧序列冗余信息的有效方法。

·运动表示：不同区域的图像需要使用不同的运动矢量来描述运动信息。

·运动估计：运动估计是从视频序列中抽取运动信息的一整套技术。

使用帧内编码技术可以去除空间上的冗余信息。

还记得前面提到过的图像编码标准JPEG吗？对于视频，ISO同样也制定了标准：Motion JPEG即MPEG，MPEG算法是适用于动态视频的压缩算法，它除了对单幅图像进行编码外，还利用图像序列中的相关原则去除冗余，这样可以大大提高视频的压缩比。截至目前，MPEG的版本一直在不断更新中，主要包括这样几个版本：Mpeg1（用于VCD）、Mpeg2（用于DVD）、Mpeg4 AVC（现在流媒体使用最多的就是它了）。

相比较于ISO制定的MPEG的视频压缩标准，ITU-T制定的H.261、H.262、H.263、H.264一系列视频编码标准是一套单独的体系。其中，H.264集中了以往标准的所有优点，并吸取了以往标准的经验，采用的

是简洁设计，这使得它比Mpeg4更容易推广。现在使用最多的就是H.264标准，H.264创造了多参考帧、多块类型、整数变换、帧内预测等新的压缩技术，使用了更精细的分像素运动矢量（1/4、1/8）和新一代的环路滤波器，这使得压缩性能得到大大提高，系统也变得更加完善。

1.6.2 编码概念

1.IPB帧

视频压缩中，每帧都代表着一幅静止的图像。而在进行实际压缩时，会采取各种算法以减少数据的容量，其中IPB帧就是最常见的一种。

·I帧：帧内编码帧（intra picture），I帧通常是每个GOP（MPEG所使用的一种视频压缩技术）的第一个帧，经过适度地压缩，作为随机访问的参考点，可以当成静态图像。I帧可以看作一个图像经过压缩后的产物，I帧压缩可以得到6：1的压缩比而不会产生任何可觉察的模糊现象。I帧压缩可去掉视频的空间冗余信息，下面即将介绍的P帧和B帧是为了去掉时间冗余信息。

·P帧：前向预测编码帧（predictive-frame），通过将图像序列中前面已编码帧的时间冗余信息充分去除来压缩传输数据量的编码图像，也称为预测帧。

·B帧：双向预测内插编码帧（bi-directional interpolated prediction frame），既考虑源图像序列前面的已编码帧，又顾及源图像序列后面的已编码帧之间的时间冗余信息，来压缩传输数据量的编码图像，也称为双向预测帧。

基于上面的定义，我们可以从解码的角度来理解IPB帧。

·I帧自身可以通过视频解压算法解压成一张单独的完整视频画面，所以I帧去掉的是视频帧在空间维度上的冗余信息。

·P帧需要参考其前面的一个I帧或者P帧来解码成一张完整的视频画面。

·B帧则需要参考其前一个I帧或者P帧及其后面的一个P帧来生成一张完整的视频画面，所以P帧与B帧去掉的是视频帧在时间维度上的冗余信息。

IDR帧与I帧的理解

在H264的概念中有一个帧称为IDR帧，那么IDR帧与I帧的区别是什么呢？首先来看一下IDR的英文全称instantaneous decoding refresh picture，因为H264采用了多帧预测，所以I帧之后的P帧有可能会参考I帧之前的帧，这就使得在随机访问的时候不能以找到I帧作为参考条件，因为即使找到I帧，I帧之后的帧还是有可能解析不出来，而IDR帧就是一种特殊的I帧，即这一帧之后的所有参考帧只会参考到这个IDR帧，而不会再参考前面的帧。在解码器中，一旦收到一个IDR帧，就会立即清理参考帧缓冲区，并将IDR帧作为被参考的帧。

2.PTS与DTS

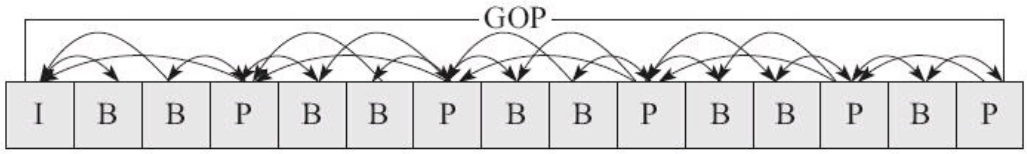
DTS主要用于视频的解码，英文全称是Decoding Time Stamp，PTS主要用于在解码阶段进行视频的同步和输出，全称是Presentation Time Stamp。在没有B帧的情况下，DTS和PTS的输出顺序是一样的。因为B帧打乱了解码和显示的顺序，所以一旦存在B帧，PTS与DTS势必就会不同，本书后边的章节里会详细讲解如何结合硬件编码器来重新设置PTS和DTS的值，以便将硬件编码器和FFmpeg结合起来使用。这里先简单介绍一下FFmpeg中使用的PTS和DTS的概念，FFmpeg中使用AVPacket结构体来描述解码前或编码后的压缩数据，用AVFrame结构体来描述解码后或编码前的原始数据。对于视频来说，AVFrame就是视频的一帧图像，这帧图像什么时候显示给用户，取决于它的PTS。DTS是AVPacket里的一个成员，表示该压缩包应该在什么时候被解码，如果视频里各帧的编码是按输入顺序（显示顺序）依次进行的，那么解码和显示时间应该是一致的，但是事实上，在大多数编解码标准（如H.264或HEVC）中，编码顺序和输入顺序并不一致，于是才会需要PTS和DTS这两种不同的时间戳。

3.GOP的概念

两个I帧之间形成的一组图片，就是GOP（Group Of Picture）的概念。通常在为编码器设置参数的时候，必须要设置gop_size的值，其代表的是两个I帧之间的帧数目。前面已经讲解过，一个GOP中容量最大的帧就是I帧，所以相对来讲，gop_size设置得越大，整个画面的质量就会越好，但是在解码端必须从接收到的第一个I帧开始才可以正确解码出原始图像，否则会无法正确解码（这也是前面提到的I帧可以作为随机访问的帧）。在提高视频质量的技巧中，还有个技巧是多使用B帧，一般来说，I的压缩率是7（与JPG差不多），P是20，B可以达到50，可见使用B帧能节省大量空间，节省出来的空间可以用来更多地保存I帧，

这样就能在相同的码率下提供更好的画质。所以我们要根据不同的业务场景，适当地设置gop_size的大小，以得到更高质量的视频。

结合IPB帧和图1-11，相信大家能够更好地理解PTS与DTS的概念。



解码顺序	1	3	4	2	6	7	5	9	10	8	12	13	11	15	14
显示顺序	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DTS	1	3	4	2	6	7	5	9	10	8	12	13	11	15	14
PTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

图　1-11

1.7　本章小结

本章的内容到此结束，这里简单复习一下要点。本章首先介绍了音频的物理现象，进而研究如何存储以及编码音频，然后又讨论了图像的物理现象，从RGB到YUV，最后讲解了视频的编码操作。本章的内容理论和概念比较多，难免会枯燥一些，但是了解这些概念是必需的。其实本章讲解的东西若想要继续深究，还是有很多可供研究的，由于本书的核心是应用层的开发，因此就不考虑展开来讲了。对于应用层的开发人员来说，掌握本章的内容就已经足够了，若想要进一步了解相关的知识，可自行参考其他资料。

第2章　移动端环境搭建

第1章学习了音视频的基础知识，本章将介绍如何在移动端（包括iOS和Android两个平台）搭建开发音视频的环境，以及如何在每一个平台的开发环境中添加C++支持。由于篇幅有限，在搭建开发环境时，本章将只以Mac OS操作系统为例进行讲解，对于Linux、Windows等操作系统的读者，需要大家自行将各种开发环境适配到对应的系统中。但是，书中所有项目的设计与实现是不区分开发系统环境的。

在音视频的开发过程中，不可能所有的编码、解码以及处理都由开发者从零开始编写，因此免不了会用到一些第三方库，所以本章还将讲解交叉编译，并尝试交叉编译几个音视频相关的开源库。在本章的最后，会使用LAME这个开源的MP3编码库在iOS平台和Android平台上将一个PCM文件编码为MP3文件，最终将编码后的MP3文件发送到电脑上即可进行播放，这也是本章要完成的最终目标。

2.1　在iOS上如何搭建一个基础项目

1.新建一个iOS项目

首先，在Xcode中选择新建一个项目，会弹出如图2-1所示的界面，然后我们选择一个新项目的模板，一般选择Single View Application模板。

之后点击Next，进入图2-2所示的界面，这里需要键入产品的名字及包名，注意产品的名字将作为安装到用户手机上的应用名称，包名则是该应用的唯一标识，键入完毕之后点击Next。

完成上述步骤之后，就完成了一个iOS项目的创建。查看该项目的工程文件，会看到Xcode默认是以Story board的形式构建的界面部分，由于我们不希望使用这种形式来构建界面，而是希望使用xib的形式来构建界面，所以要在Main Interface选项中删除其中的内容，如图2-3所示。

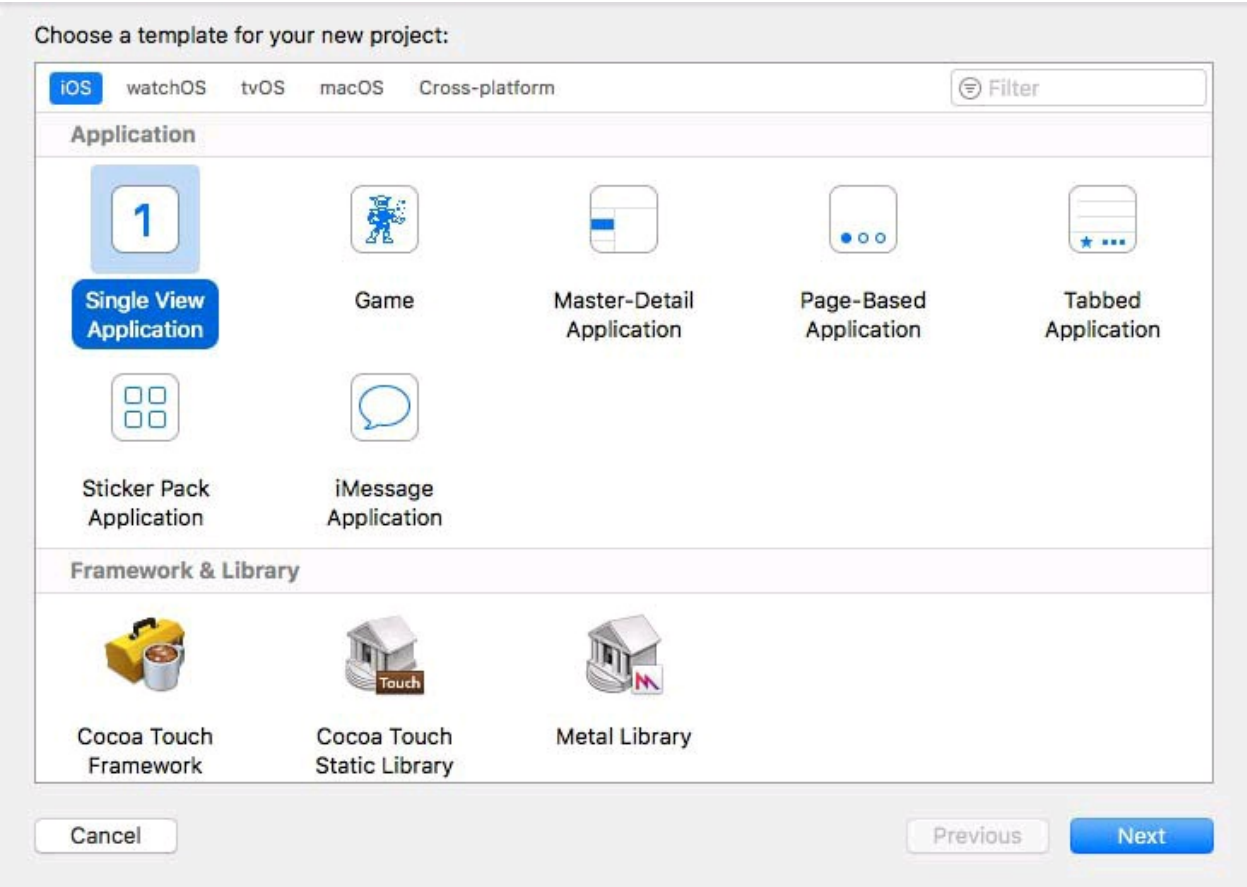

Choose a template for your new project:
iOS
watchOS
tvOS
macOS
Cross-platform
Filter
Application
Single View Application
Game
Master-Detail Application
Page-Based Application
Tabbed Application
Sticker Pack Application
iMessage Application
Framework & Library
Cocoa Touch Framework
Cocoa Touch Static Library
Metal Library
Cancel
Previous
Next

图　2-1

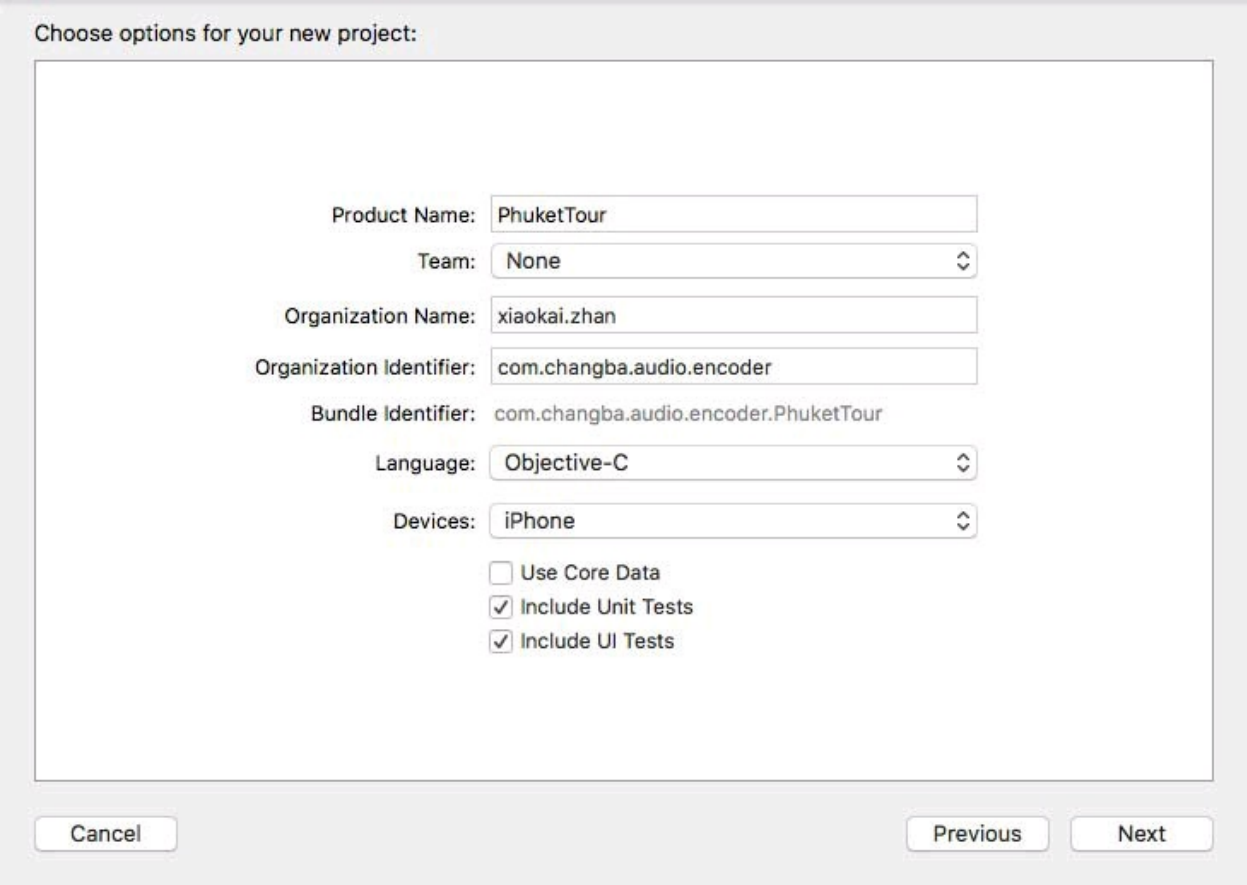


图　2-2

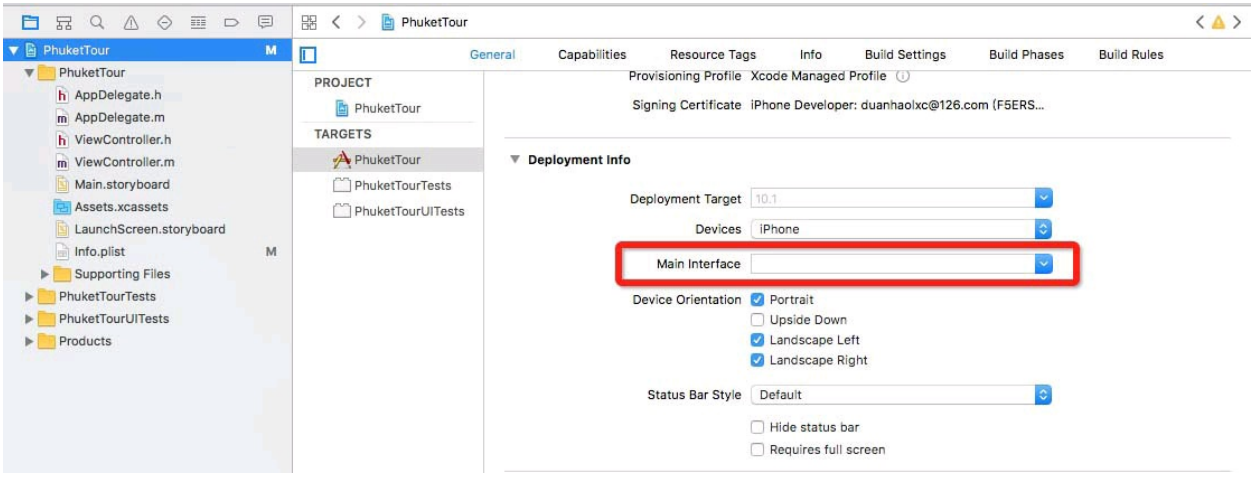


图　2-3

接下来，再建立一个界面文件作为应用的第一个界面，而在iOS中一个界面就是一个xib文件，如图2-4所示。

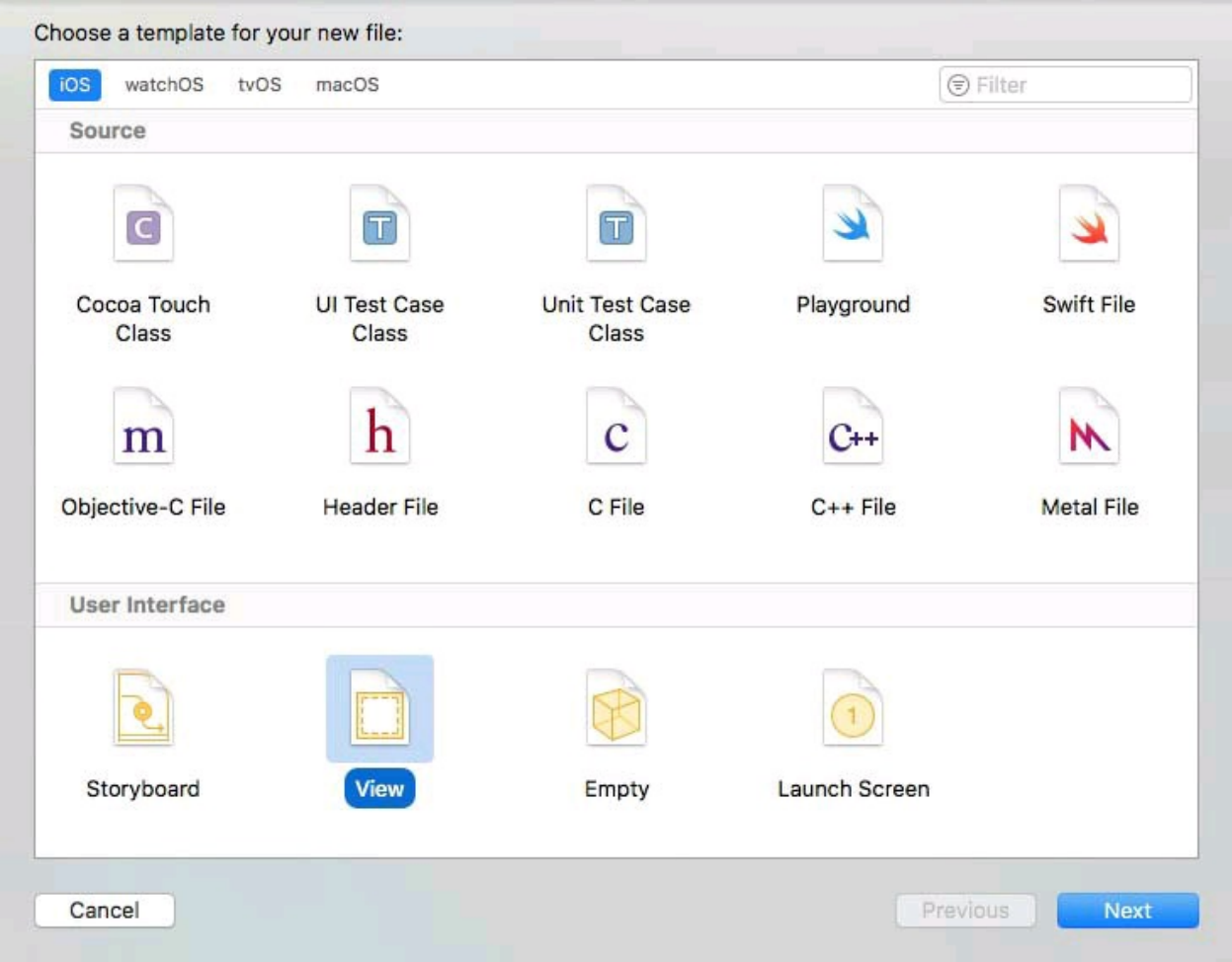


图　2-4

然后点击Next，进入图2-5所示的界面，键入该xib的名字，然后选择Target为主工程，点击Create。

现在，打开该xib文件，如图2-6所示，首先要在Placeholders选项下面的File’s Owner中连接视图中的view，然后设定Class的值为ViewController类。

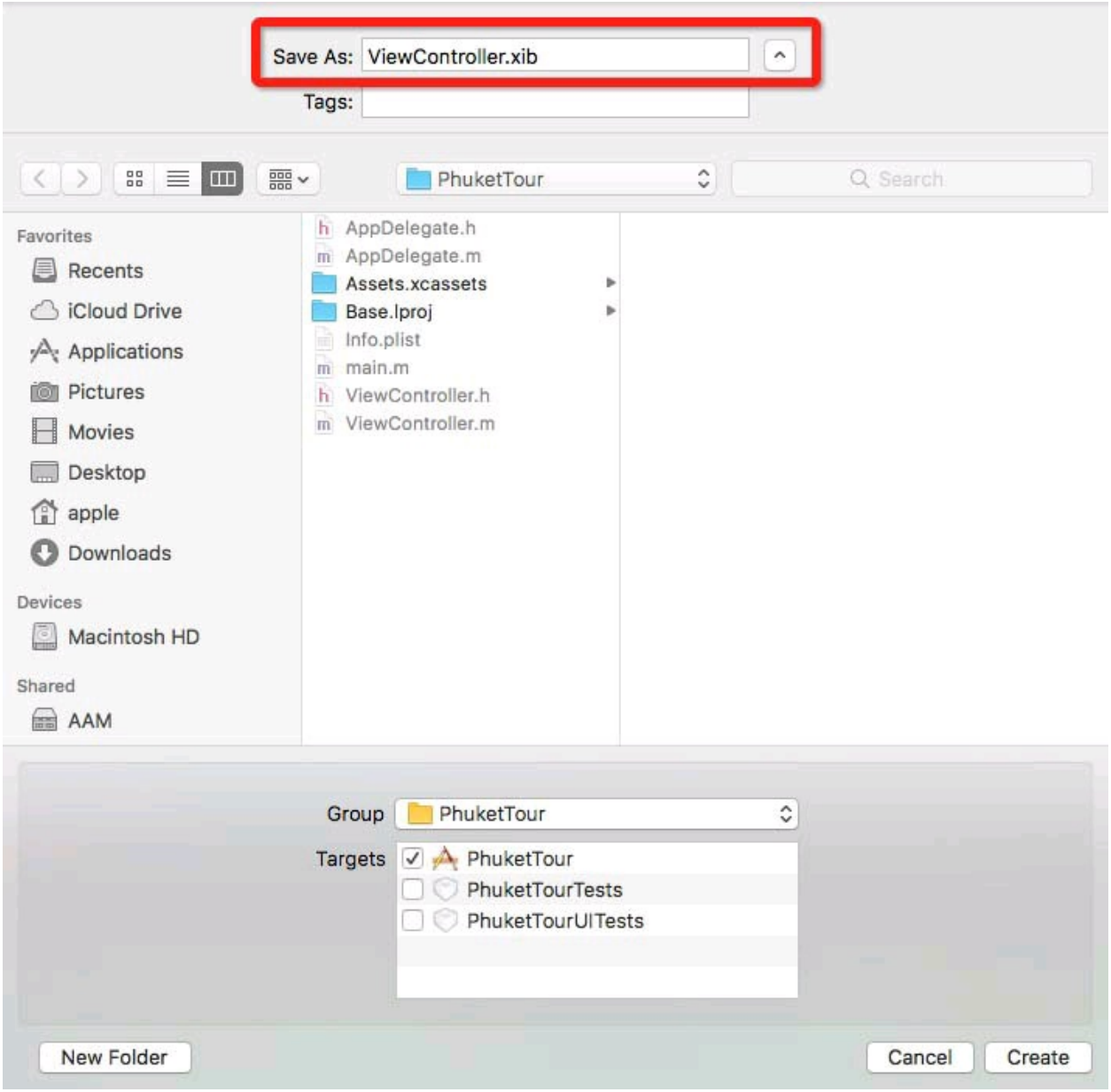


图　2-5

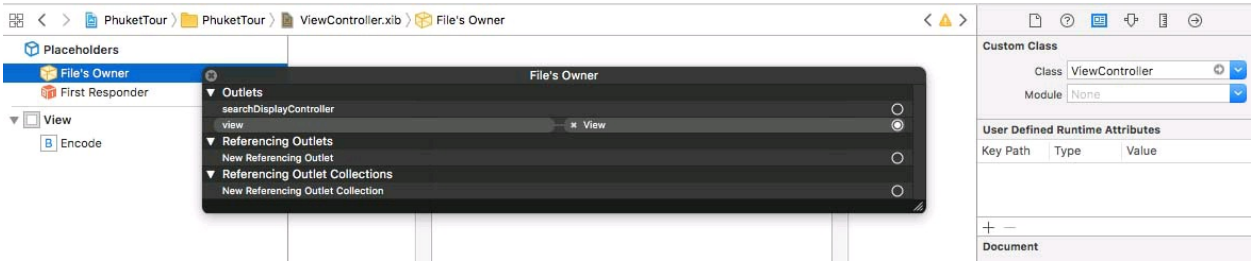


图　2-6

iOS程序运行的时候其入口是main.m，但是苹果公司不希望开发者

修改该文件，而是要求开发者去修改应用的入口——AppDelgate.m文件，该文件中提供了各种生命周期方法，应用启动的时候将会触发的方法是application：didFinishLaunchingWithOptions，所以修改该生命周期方法的代码如下：

```
self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
        bounds]];
UINavigationController *navigationController = [[UINavigationController
        alloc] initWithRootViewController:[[ViewController alloc]
        initWithNibName:@"ViewController" bundle:nil]];
self.window.rootViewController = navigationController;
[self.window makeKeyAndVisible];
```

接下来要为该xib拖入一个Encode按钮，并且将该按钮的点击事件委托给ViewController中的startEncode方法，如图2-7所示。

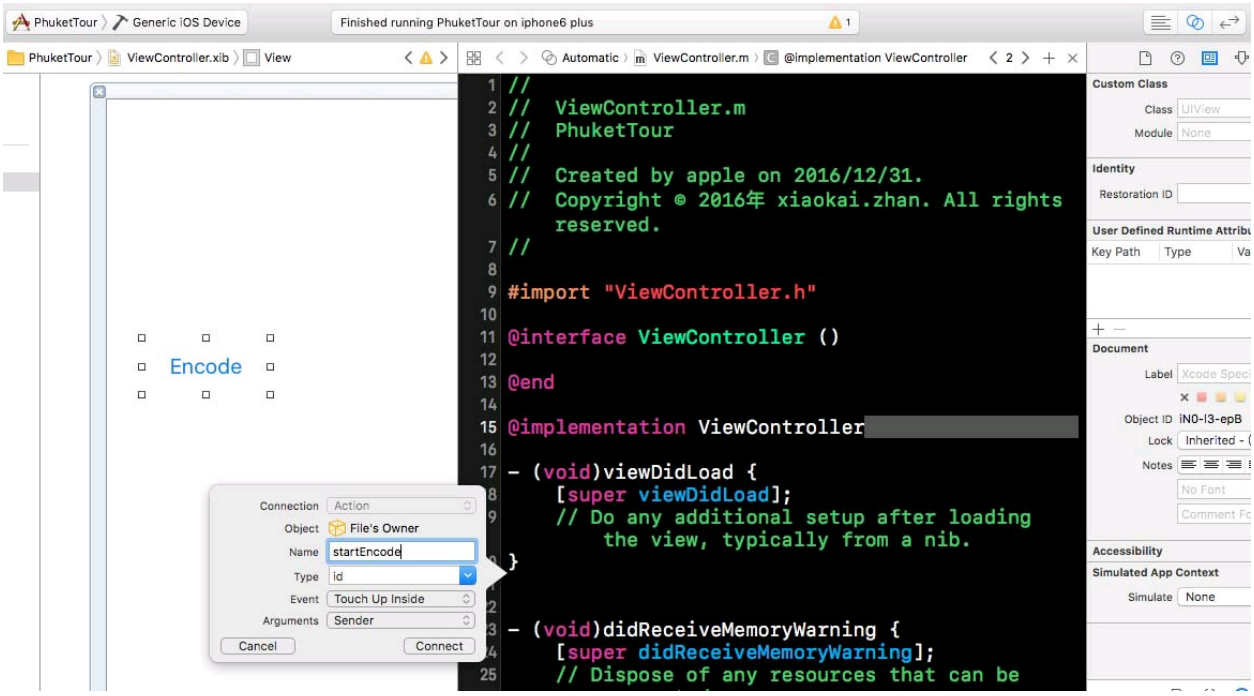


图 2-7

startEncode方法提供了NSLog来输出一条日志信息。然后运行整个项目，点击Encode按钮，如果可以看到正常的日志信息，则说明项目搭建成功了。

2.CocoaPods的介绍与使用

开发应用的时候，并不是所有的基础功能都由开发者从零开始开发，所以经常会用到第三方开源类库。一定要记住，不要重复造轮子！不过，在引用这些类库时，可能会出现几个令人头疼的问题：第一是所引用的第三方类库有可能需要依赖于其他的第三方类库；第二是需要使用已引用库的新功能，需要了解该类库对应的版本号，并且该类库对其所依赖的其他第三方类库可能也有版本的要求。如果我们仅仅是复制粘贴，那将会非常麻烦，尤其是对于大型项目来说，有时甚至可能会是一种灾难。所以在Java Web的开发中，大家通常使用Maven来构建项目。选择Maven，一方面是基于项目的依赖关系，另一个重要的原因就是为了方便引用第三方的类库。对于Android的应用开发，现在使用最多的就是Gradle，一方面是出于打包的需求，另一方面也是为了满足引用第三方库的需要。对于iOS应用呢？使用最多的则是CocoaPods，它可以为开发者提供引用众多第三方类库的功能，比如JSONKit、AFNetWorking等。

了解了CocoaPods的作用之后，接下来就是学习如何安装和使用CocoaPods来引用第三方库。

（1）安装CocoaPods

首先要安装好Ruby环境，Mac机器上已经自带了Ruby环境，如果是其他的开发系统，请读者自行安装，关于如何安装Ruby环境，互联网上提供了大量的资料。安装完Ruby环境之后，使用如下命令就可以安装CocoaPods了：

```
sudo gem install cocoapods
```

如果执行上述命令之后很长时间都没有反应，那么可能是gem所使用的默认源有问题，可以使用淘宝提供的源来安装CocoaPods，具体的做法是先删除掉默认的源，操作命令如下：

```
gem sources --remove https://rubygems.org/
```

然后替换上新的源，命令如下：

```
gem sources -a http://rubygems-china.oss.aliyuncs.com
```

此时，可以执行以下命令来查看现在的源到底是什么：

```
gem sources -l
```

如果出现下面的内容则说明源更改成功了：

```
*** CURRENT SOURCES ***
http://rubygems-china.oss.aliyuncs.com
```

此时，再一次在终端执行安装CocoaPods的命令：

```
sudo gem install cocoapods
```

稍等片刻，就可以将CocoaPods下载到本地并且安装成功了。

（2）使用CocoaPods

前面提到过CocoaPods是用来管理第三方库的工具，那么它是如何判断项目需要依赖哪一个第三方库的呢？答案是根据根目录下名为Podfile的配置文件来获知的。因此该配置文件的语法以及如何编写该配置文件自然就成了本章节学习的重点，下面一起来看一下。

首先输入平台系统的要求：

```
platform :ios, '7.0'
```

上述命令代表项目运行在iOS 7.0平台之上，配置文件从第二行开始就会出现一个从target'Project Name'do到end的代码块，该代码块将用于配置具体要引入的第三方库。代码块如下所示：

```
target :ktv do
    pod 'Mantle', '1.5'
```

```
    pod 'AFNetworking', '2.6.0'
end
```

上述配置代表项目需要引进版本号为1.5的Mantle与版本号为2.6.0的AFNetworking库。最后，在命令行中，进入项目的根目录下执行pod install命令，如果Podfile配置文件的语法没有问题，那么CocoaPods就会为项目生成两个文件和一个目录。其中，一个文件的名称是项目名称.xcworkspace，另一个文件的名称是Podfile.lock，而生成的目录则是Pods，其中放置了引用第三方库的源码。此时，双击后缀名为xcworkspace的文件即可打开此项目，查看Xcode目录里的结构，会发现项目中已经有了刚才新引入的第三方库了。

增加C++支持

在单独编写一个C或C++的项目时，如果该项目需要引用到第三方库，那么编译阶段需要配置参数“extra-cflags，-I”来指定引用头文件的位置，链接阶段需要配置参数“ld-flags，-L”来指定静态库的位置，并且使用-l来指定引用的是哪一个库。在C++的编译中，如果需要在程序的执行过程中带入一些宏（define的常量），那么就应该在“extra-cflags”的后面增加自己需要定义的宏，例如-DAUTO_TEST，这就相当于在程序中编写了如下一行代码：

```
define AUTO_TEST
```

那么在Xcode中，应该如何指定头文件、第三方库，以及预定义宏呢？其实，Xcode里面已经存在了对应的参数设置，只需要在Build Settings中设置这些参数即可。首先对应于-I来指定头文件的目录，Xcode使用Search Paths来设置头文件的搜索路径，通常会用到Header Search Paths选项来指定头文件的搜索路径。其中预定义变量$（SRCROOT）和$（PROJECT_DIR）都是项目的根目录，可以基于这两个预定义变量再加上相对路径来指定头文件所在的具体位置；预定义宏在Other C Flags选项中，可以写入-DAUTO_TEST，表示定义了AUTO_TEST宏；在指定第三方库时，-L对应到Xcode中就是other Link flags选项，其中可以写入需要链接的库文件；在Xcode项目中直接添加一个静态库文件，Xcode会默认在Build phases选项的Link Binary with

Library里加入该静态库，但是如果Xcode没有自动加入该静态库的话，就需要开发者手动加一下，这里其实也是-L的一种表示方式。

接下来，编写一个类Mp3Encoder，负责将PCM数据编码为MP3文件。首先建立Mp3Encoder.cpp和Mp3Encoder.h这两个文件，然后再编写一个encode方法，该方法将调用printf输出一行日志信息，以代表调用了这个方法。

现在就为前面搭建起来的iOS项目增加C++的支持，由于OC语法支持混编，所以开发者仅需要把引用C++的OC类的后缀名改为.mm（OC类的正常后缀名是.m），就可以和C++一块编译了。所以开发者仅需要把ViewController的后缀名改为.mm，再包含进Mp3Encoder.h头文件，实例化该类，最后调用该类的encode方法，代码如下：

```
Mp3Encoder* encoder = new Mp3Encoder();
encoder->encode ();
delete encoder;
```

如果大家可以看到printf输出的日志信息，则代表已经为该iOS项目成功添加了C++的支持。是不是很简单呢？其实这也是苹果公司为开发者提供的便利，下面还会讲解Android是如何添加C++支持的，相较于iOS平台而言，Android平台要麻烦得多。

2.2　在Android上如何搭建一个基础项目

首先要在开发机器上安装Eclipse（尽量选择已经安装好ADT的版本）与Android的SDK，然后在Eclipse中建立一个Android项目，命名为Mp3Encoder，如图2-8所示。

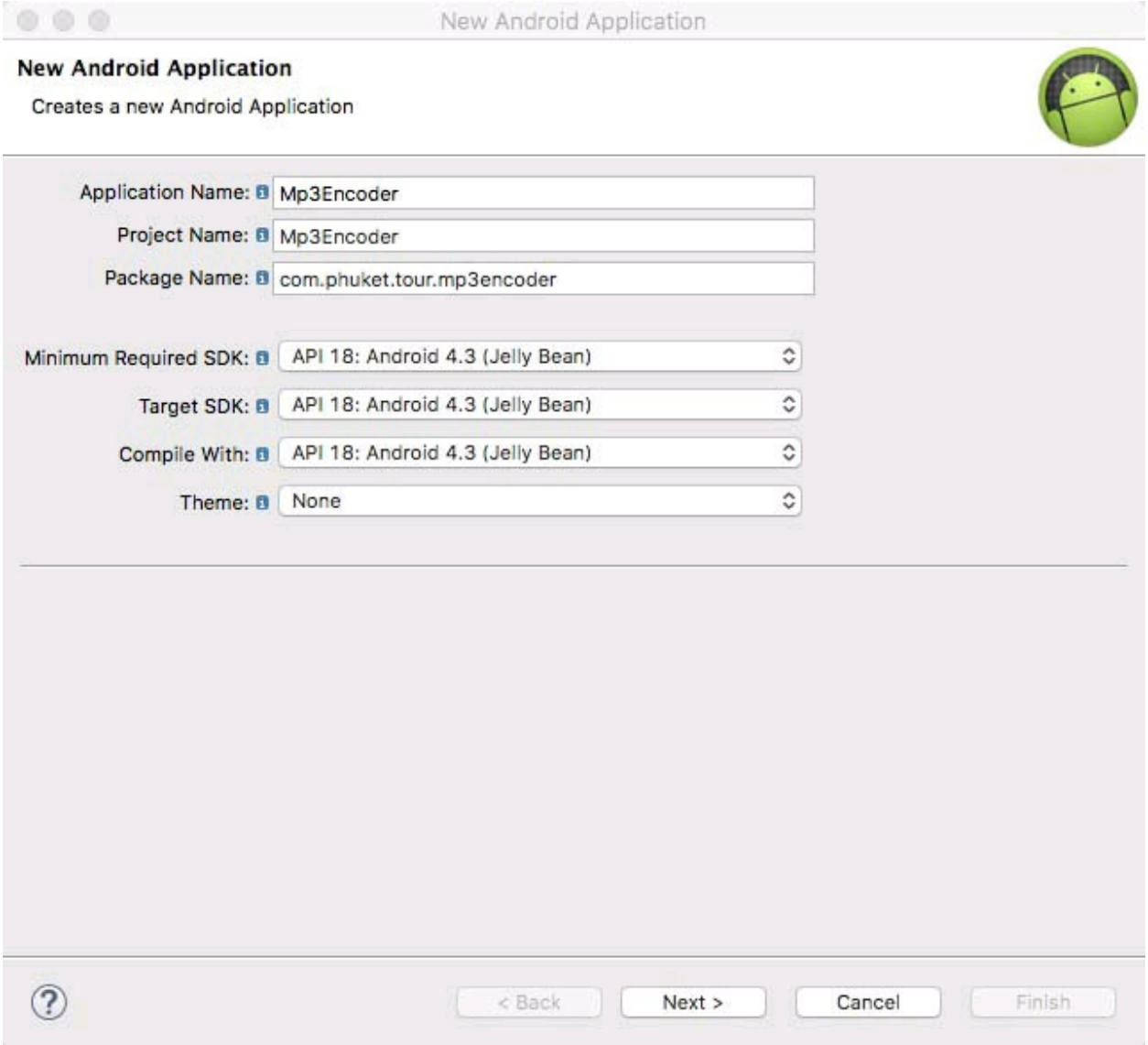


图　2-8

在图2-8所示的界面中，我们需要键入项目的名字、包的名字（应用程序的唯一标识），由于在决定写这本书的时候笔者正在普吉岛度假，所以就将其命名为com.phuket.tour.mp3encoder了。然后点击Next按

钮，进入图2-9所示的界面。

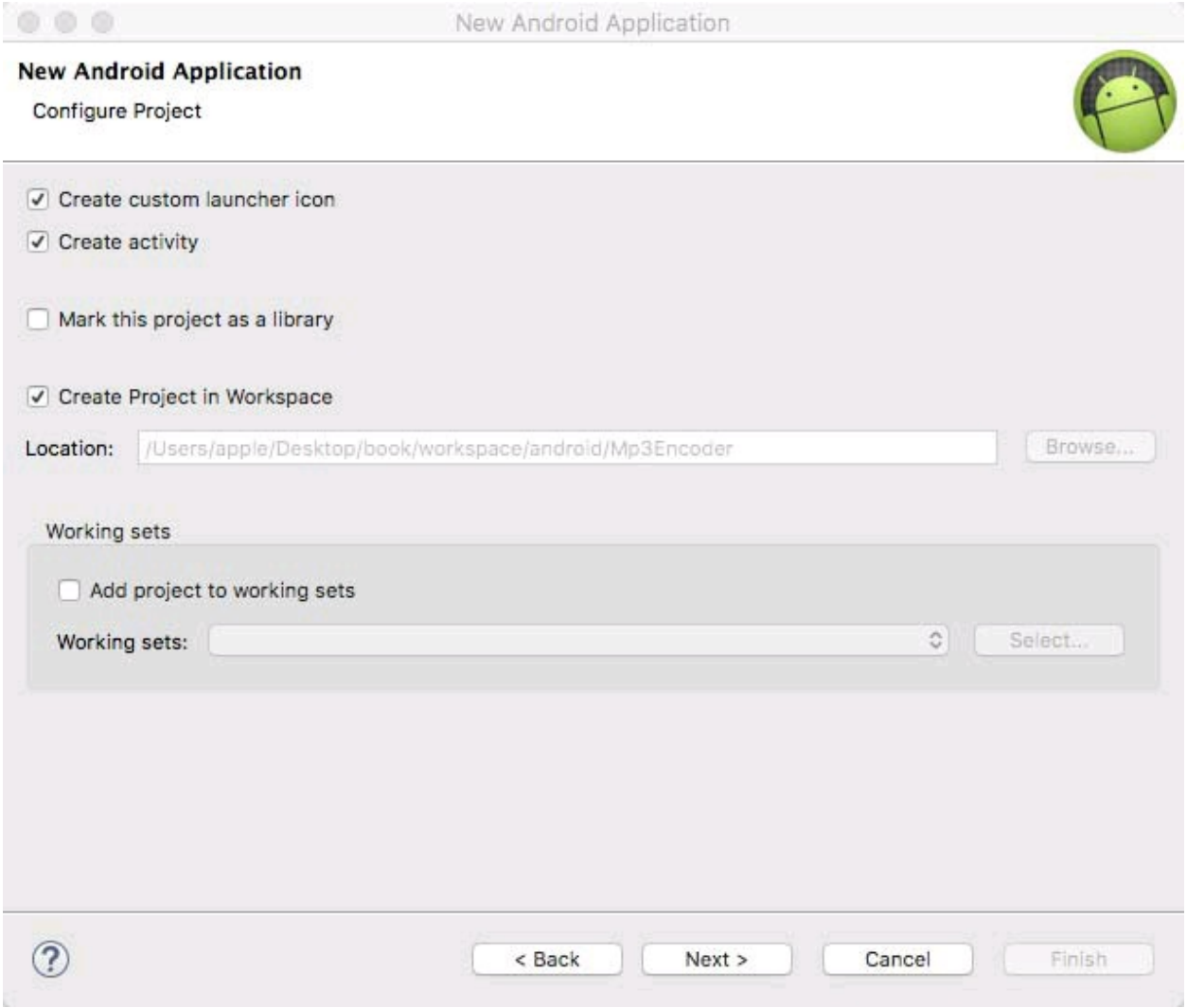


图　2-9

在这一步骤中，可以让IDE建立一个默认的Activity，选中图2-9中的Create activity复选框，点击Next按钮，进入图2-10所示的界面。

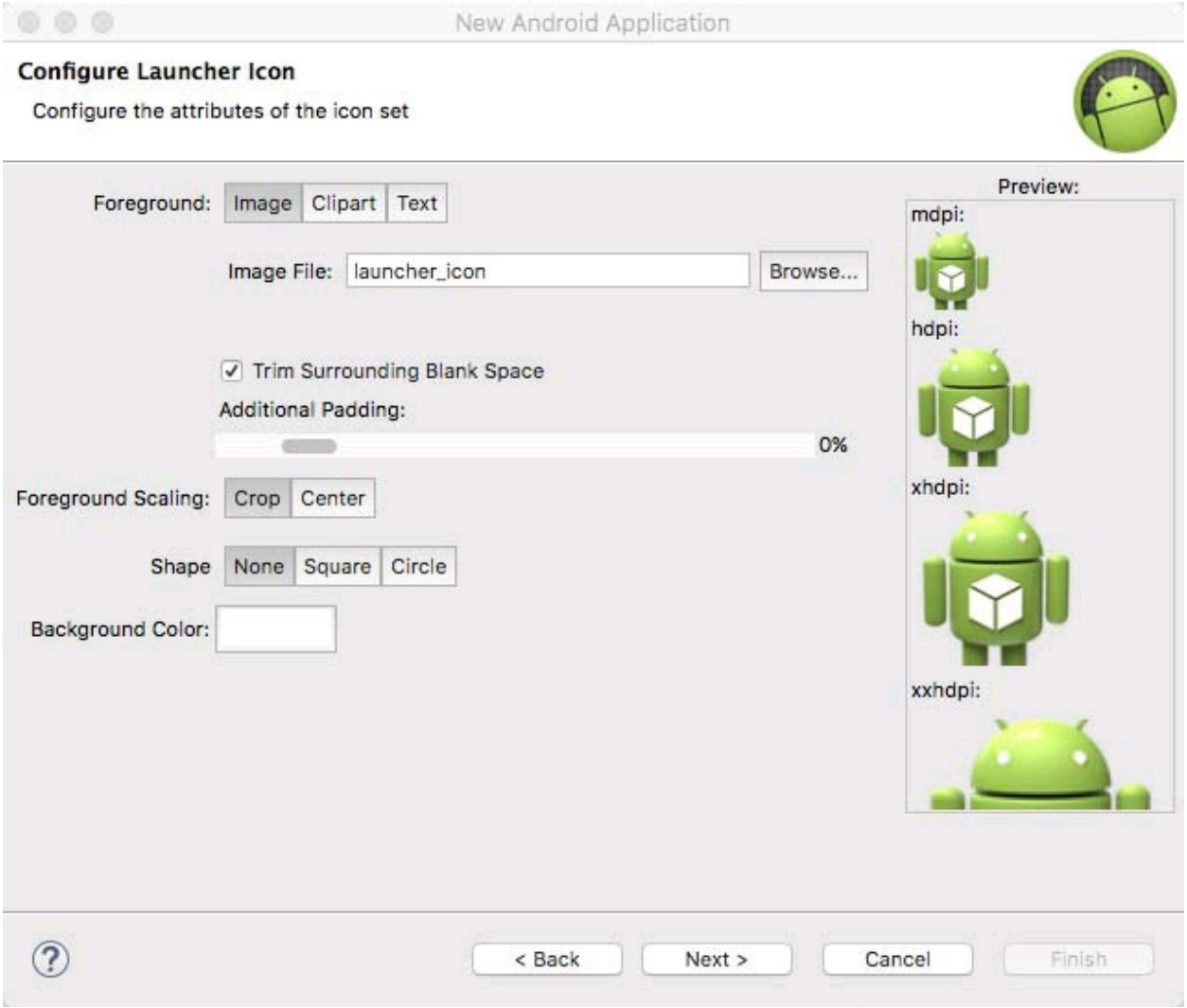


图　2-10

这一步不用做任何选择，全部使用默认配置，点击Next，进入图2-11所示的界面。

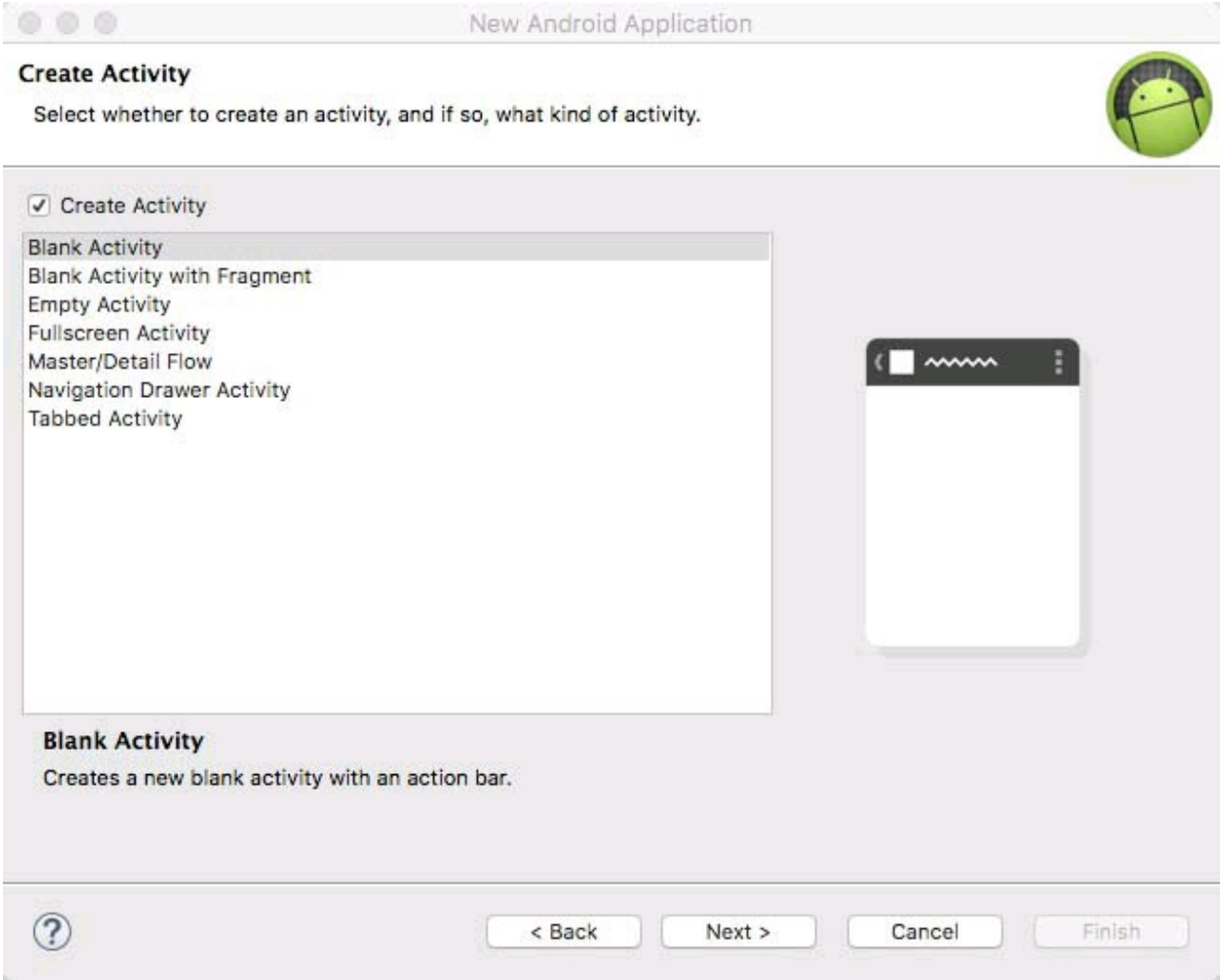


图　2-11

在图2-11中，选择Blank Activity，代表应用默认是一个空的页面（后面会加上自己的按钮），点击Next，进入图2-12所示的界面。

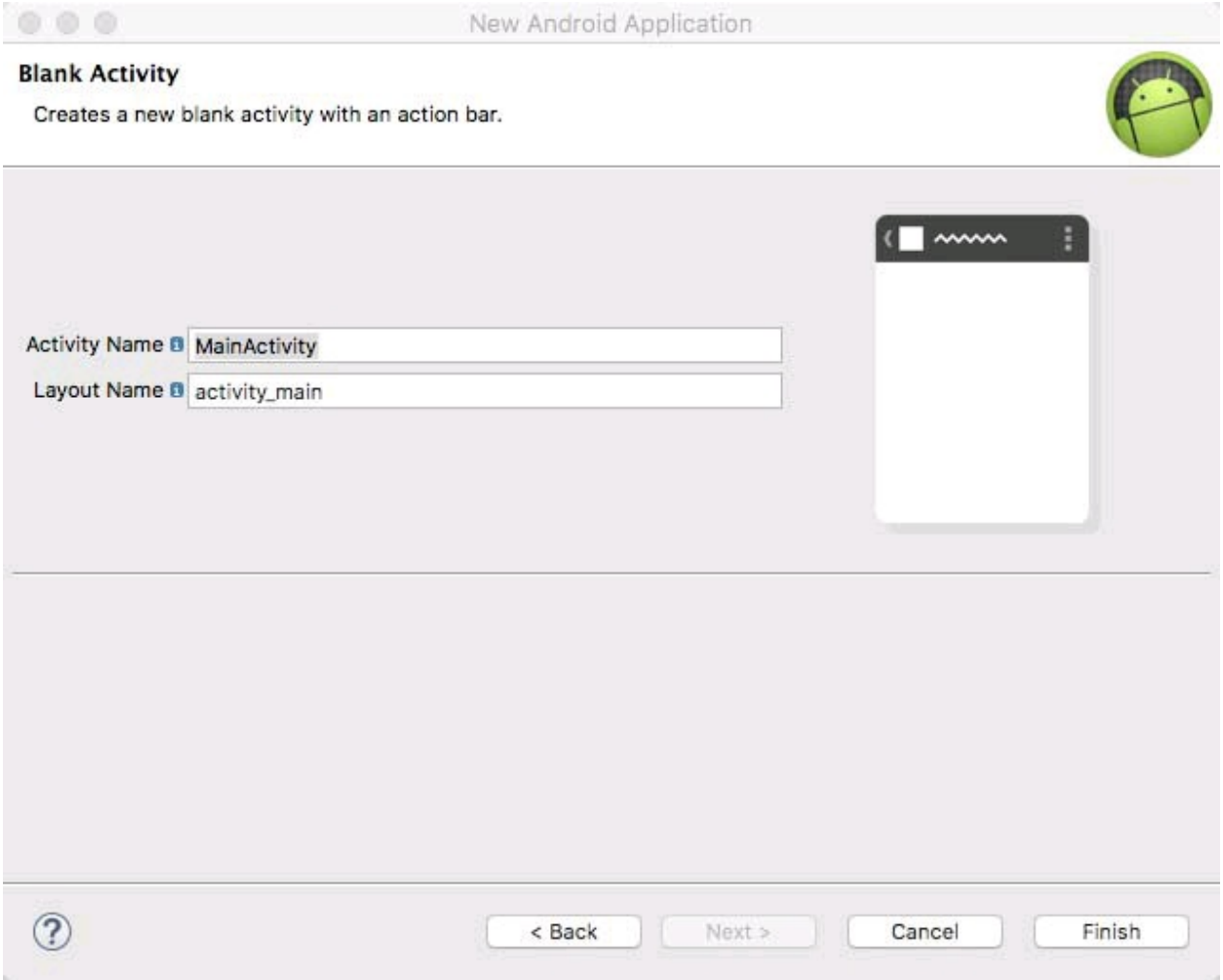


图　2-12

在图2-12中，键入主Activity的名字为MainActivity，然后点击Finish，即可成功建立该项目，其目录的结构如图2-13所示。

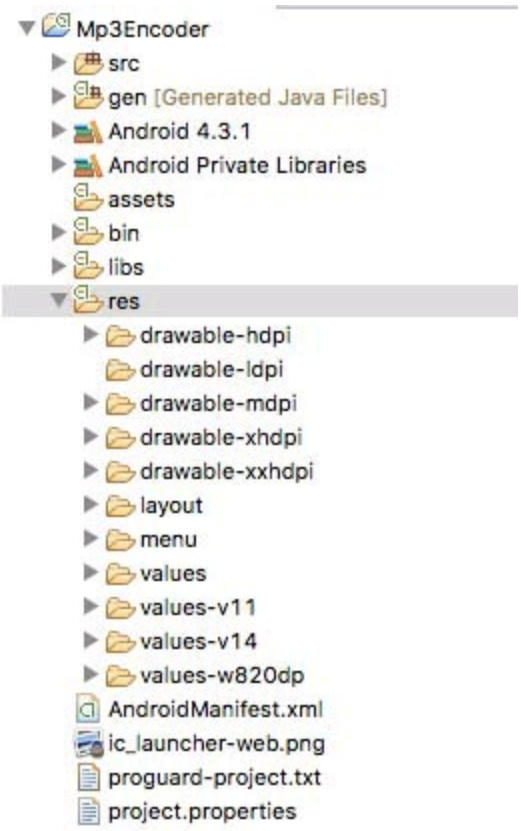


图　2-13

增加C++支持

在Android平台上开发音视频相关的项目，必定会涉及JNI（Java Native Interface）这一概念。它是一种编程框架，允许运行于JVM的Java程序去调用本地代码（C/C++以及汇编语言的代码）。本地代码通常会与硬件或操作系统相关联，因而其会在一定程度上破坏Java语言所宣称的跨平台特性。不过在某些业务场景下这种调用又是必需的，比如Android系统（framework层面）就采用了大量的JNI手段去调用Native层的实现库。如果抛开音视频相关的场景，仅仅考虑Android的开发，那么有哪些场景会用到JNI呢？大约有以下几种情况。

·应用程序需要一些平台相关特性的支持，而Java层没有对应的API支持（比如OpenSL ES的使用）。

·调用成熟的或者已经存在的、用C/C++语言编写的代码库。比如使

用OpenGL ES的视频特效处理库，或者利用FFmpeg、LAME等第三方开源库。

·应用程序的某些关键操作对运行速度有较高的要求。这部分逻辑可以用C或者汇编语言来编写，再通过JNI向Java层提供访问接口。

在本书后面的项目案例中，熟练使用JNI是最基本的要求，所以在这里先详细地讲解一下。JNI主要有两种调用形式：第一种也是最常见的，即Java代码调用Native的代码；第二种则恰好相反，就是在Native层调用Java代码。由于使用场景最多的是第一种方式，所以本节只介绍这种方式，而第二种方式的调用在后续的章节中也会用到，到时候读者自然会学习到。

要完成一个Java层调用Native代码的项目，大致步骤如下。

1）编写一个Java类，并且在某个方法签名的修饰符中加上native修饰符。

2）使用javac命令编译第1步中的Java类，使之成为一个class文件，如果使用的是Eclipse的IDE，则其将会自动执行javac的编译命令，生成的class文件将存在于项目目录下的bin/classes目录下。

3）使用javah命令将第2步的输出作为输入，生成JNI的头文件。

4）将JNI头文件复制到项目下的jni目录，并且建立一个cpp的实现文件实现该JNI头文件中的函数。

5）编写Android.mk文件，加入第4步的本地代码，利用ndk-build生成动态链接库。

6）在Java类中加载第5步生成的动态链接库。

7）在Java类中调用该Native方法。

下面以一个实例来完成上述的步骤，具体如下。

1）在Eclipse的com.phuket.tour.studio包下，建立一个Java文件Mp3Encoder.java。

2）在上述Java文件中编写一个本地方法：

```
public native void encode();
```

3）进入对应的class文件所在的目录下，执行下面的命令生成JNI接口文件：

```
javah -jni com.phuket.tour.studio.Mp3Encoder
```

4）然后把该头文件复制到jni目录下，编写一个Mp3Encoder.cpp来实现该接口文件，这里仅仅输出一行日志。

首先利用“宏定义”定义一个输出日志的宏：

```
#define LOGI(...)  __android_log_print(ANDROID_LOG_INFO, LOG_TAG,
      __VA_ARGS__)
```

然后定义LOG_TAG为Mp3Encoder，并且实现该encode方法：

```
JNIEXPORT void JNICALL
        Java_com_phuket_tour_studio_Mp3Encoder_encode(JNIEnv * env,
        jobject obj) {
    LOGI("encoder encode");
}
```

5）新建立一个Android.mk文件，并键入以下内容：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES = ./Mp3Encoder.cpp
LOCAL_LDLIBS := -L$(SYSROOT)/usr/lib -llog
LOCAL_MODULE := libaudioencoder
include $(BUILD_SHARED_LIBRARY)
```

上述代码具体每一行的含义，在后续章节中会有详细的解释。现在最重要的就是把该项目运行起来。

6）在当前目录下执行ndk-build指令，编译出该动态so库。

7）在MainActivity中写入一个静态代码块：

```
static {
    System.loadLibrary("audioencoder");
}
```

写该代码块的目的是将刚刚编译好的so库加载到我们的项目中。

8）在onCreate中调用Mp3Encoder类的encode方法，最后运行该应用，并且打开logcat视图查看结果。

如果可以在logcat中看到JNI层输出的日志，则代表已经成功地为Android项目添加了C++的支持。如果仅仅是自己编写代码，而不需要调用其他的第三方开源库，这样就完全可以了。但是在开发音视频项目时，肯定会需要依赖很多第三方的库，比如音视频编解码的开源库、音视频特效处理的开源库，等等，那么如何根据自己的需要添加各种依赖库呢？这将在2.3.3节中与大家详细讨论，最终笔者会使用LAME库编码一个MP3的音频文件作为本章的实例。

2.3　交叉编译的原理与实践

本节将学习交叉编译，在音视频的开发中了解交叉编译是必需的，因为无论在哪一种移动平台下开发，第三方库都是需要进行交叉编译的。本节会从交叉编译的原理开始介绍，然后会在两个移动平台下编译出音视频开发常用的几个库，包括X264、FDK_AAC、LAME，最终将以LAME库为例进行实践，完成一个将音频的PCM裸数据编码成MP3文件的实例，以此来证明交叉编译的重要性。

2.3.1 交叉编译的原理

先来看一下，如果要在PC上运行一个二进制程序（以源码的方式进行编译，不要以包管理工具的方式来安装），需要怎样做。

首先，要有这个二进制程序的源代码（有可能是直接下载的，也有可能是自己编写的代码），然后在PC上进行编译链接生成可执行文件，最后在Terminal下面去执行该可执行文件。

上述流程中包含了几个角色，首先是要有源代码，然后是要知道最终运行该二进制程序的机器是哪一个（其实就是本机器），当然，其中最重要的就是编译器和链接器了，对于C或者C++程序来讲，就是使用gcc和g++，而该编译器是需要预先安装在机器上的。分析了这么多角色，总结成一句话就是：使用本机器的编译器，将源代码编译链接成为一个可以在本机器上运行的程序。这就是正常的编译过程，也称为Native Compilation，中文译作本机编译。

了解了本机编译之后，再来看一下何为交叉编译。所谓交叉编译，就是在一个平台（如PC）上生成另外一个平台（Android、iOS或者其他嵌入式设备）的可执行代码。相较于正常编译，下面来看一下交叉编译的相应角色。首先，最终程序运行的设备就是Android或者iOS设备，源代码就是从第三方开源网站上下载的源代码，编译机器就是我们的PC，而编译器也必须要安装到该PC上。但是这里对编译器是有特殊需求的，最终程序运行的系统必须要提供可运行在PC上的编译器，而该编译器就是大家常说的交叉工具编译链。

了解了交叉编译之后，大家应该能够理解交叉编译存在的必要性了。在一般的嵌入式系统开发中，运行程序的目标平台其存储空间和运算能力都是有限的，尽管现在的iOS和Android设备拥有越来越强劲的计算能力，但是在这种嵌入式设备中进行本地编译是不太可能的，一则是因为计算能力的问题，还有一个重要的原因就是编译工具以及整个编译过程异常繁琐，所以在这种情况下，直接在ARM平台下进行本机编译几乎是不可能的。而具有更加强劲的计算能力与更大存储空间的PC才是理想的选择，所以大部分的嵌入式开发平台都提供了本身平台交叉编译所需要的交叉工具编译链，通过该交叉工具编译链，开发者就能在PC上编译出可以运行在ARM平台下的程序了。

无论是自行安装PC上的编译器，还是下载其他平台（Android或者iOS）的交叉工具编译链，它们都会提供以下几个工具：CC、AS、AR、LD、NM、GDB。那么，这几个工具到底是做什么用的呢？下面就来逐一解释一下。

·CC：编译器，对C源文件进行编译处理，生成汇编文件。

·AS：将汇编文件生成目标文件（汇编文件使用的是指令助记符，AS将它翻译成机器码）。

·AR：打包器，用于库操作，可以通过该工具从一个库中删除或者增加目标代码模块。

·LD：链接器，为前面生成的目标代码分配地址空间，将多个目标文件链接成一个库或者是可执行文件。

·GDB：调试工具，可以对运行过程中的程序进行代码调试工作。

·STRIP：以最终生成的可执行文件或者库文件作为输入，然后消除掉其中的源码。

·NM：查看静态库文件中的符号表。

·Objdump：查看静态库或者动态库的方法签名。

了解了这些之后，当读者再进行交叉编译或者使用交叉编译工具链提供的工具时，就不会感到陌生了，接下来将会在iOS平台和Android平台分别演示如何交叉编译出几个常用的音视频开源库。

编译器对比

正常编译一个程序的过程如下：

编译：gcc-c main.cpp./libmad/mad_decoder.cpp-I./libmad/include

打包：ar cr../prebuilt/libmedia.a mad_decoder.o

链接：g++-o main main.o-L../prebuilt-l mdedia

在这个过程中，gcc、ar、g++是我们用到的三个编译工具，在这里没有用到的ranlib、gdb、nm、strip等都会包含在PC的编译器中，同样其他平台提供的交叉工具编译链中也会包含这些命令行工具，比如Android提供的NDK，其交叉工具编译链中的prebuilt/darwin-x86_64/bin中，就包含了对应的gcc、ar、g++、gdb、strip、nm、ranlib等工具。

2.3.2 iOS平台交叉编译的实践

前面提到的目标平台虽然都基于ARM平台，但是随着时间的推移，平台也在不断地演进，就像armv5到armv6、armv7以及到现在的arm64，对于iOS平台来讲，每一代手机对应的指令集到底应该是什么呢？下面就依据iOS设备发布的时间线来逐个看一下。

·armv6：iPhone、iPhone 2、iPhone 3G

·armv7：iPhone 4、iPhone 4S

·armv7s：iPhone 5、iPhone 5S

·arm64：iPhone 5S、iPhone 6（P）、iPhone 6S（P）、iPhone 7（P）

机器对指令集的支持是向下兼容的，因此armv7的指令集是可以运行在iPhone 5S中的，只是效率没那么高而已。借此机会，先来讨论一下iOS项目文件中的一项配置，即Build Settings里面的Architectures选项。Architectures指的是该App支持的指令集，一般情况下，在Xcode中新建一个项目，其默认的Architectures选项值是Standard architectures（armv7、arm64），表示该App仅支持armv7和arm64的指令集；Valid architectures选项指即将编译的指令集，一般设置为armv7、armv7s、arm64，表示一般会编译这三个指令集；Build Active Architecture Only选项表示是否只编译当前适用的指令集，一般情况下在Debug的时候设置为YES，以便可以更加快速、高效地调试程序，而在Release的情况下设置为NO，以便App在各个机器上都能够以最高效率运行，因为Valid architectures选择的对应指令集是armv7、armv7s和arm64，在Release下会为各个指令集编译对应的代码，因此最后的ipa体积基本上翻了3倍。

基于上面的描述，以及设备与指令集平台的对比，大多数情况下，我们在实际的交叉编译过程中只编译armv7与arm64这两个指令集平台下的库，因为armv7s设备的数量比较少，有armv7来保底完全是可以运行的，并且armv7到armv7s指令集的变动又比较少，而arm64的变动则比较大，设备数量也比较多，所以需要单独编译出来，以保证这一批设备可

以享受到最优质的运行状况。

1.LAME的交叉编译

首先来看一下LAME库是如何被交叉编译的，在交叉编译之前先介绍一下LAME库。

LAME简介

LAME是目前非常优秀的一种MP3编码引擎，在业界，转码成MP3格式的音频文件时，最常用的编码器就是LAME库。当达到320Kbit/s以上时，LAME编码出来的音频质量几乎可以和CD的音质相媲美，并且还能保证整个音频文件的体积非常小，因此若要在移动端平台上编码MP3文件，使用LAME便成为唯一的选择。

前面在介绍交叉编译原理的时候曾提到过，不论在何种平台上进行交叉编译，都要知道交叉编译工具链在什么地方。同样，在iOS上进行交叉编译时，也要搞清楚这个最基本的问题。其实在安装iOS的开发环境Xcode时，配套的编译器就已经安装好了。开发iOS平台下的App就是这么方便，不需要再单独下载交叉工具编译链，接下来直接去SourceForge下载最新的LAME版本，访问链接如下：

```
https://sourceforge.net/projects/lame/files/lame/3.99/
```

这里选择3.99.5版本，将源码下载下来之后，编写一个build_armv7.sh脚本，用于编译armv7指令集下的版本，以支持iPhone 5S及以下的设备，Shell脚本里面的内容如下：

```
./configure \
--disable-shared \
--disable-frontend \
--host=arm-apple-darwin \
--prefix="./thin/armv7" \
CC="xcrun -sdk iphoneos clang -arch armv7" \
CFLAGS="-arch armv7 -fembed-bitcode -miphoneos-version-min=7.0" \
LDFLAGS="-arch armv7 -fembed-bitcode -miphoneos-version-min=7.0"
make clean
make -j8
make install
```

下面分别解释一下这几个命令以及选项的意义。configure是符合GNU标准的软件包发布所必备的命令，所以这里是通过configure的方式来生成Makefile文件，然后使用make和make install编译和安装整个库。可使用configure-h命令来查看一下configure的帮助文档，了解LAME的可选配置项，具体如下。

·--prefix：指定将编译好的库放到哪个目录下，这是GNU大部分库的标准配置。

·--host：指定最终库要运行的平台。

·CC：指定交叉工具编译链的路径，其实这里就是指定gcc的路径。

·CFLAGS：指定编译时所带的参数。Shell脚本中指定-march是armv7平台，代表编译的库运行的目标平台是armv7平台；另外Shell脚本中也指定了打开bitcode选项，这使得使用编译出来的这个库的工程，可以将enable-bitcode选项设置为YES，如果没有打开该选项，那么其在Xcode中只能设置为NO，而这对于最终App的运行性能会有一定的影响。Shell脚本中同时也指定了编译出来的这个库所支持的最低iOS版本是7.0，如果不配置该参数的话，则默认是iOS 9.0版本，而所使用的编译出来的这个库的工程，若所支持的最低iOS版本不是9.0的话，Xcode就会给出警告。

·LDFLAGS：指定链接过程中的参数，同样也要带上bitcode的选项以及开发者期望App支持的最低iOS版本的选项参数。

·--disable-shared：通常是GNU标准中关闭动态链接库的选项，一般是在编译出命令行工具的时候，期望命令行工具可以单独使用而不需要动态链接库的配置。

·--disable-frontend：不编译出LAME的可执行文件。

bitcode

bitcode模式是表明当开发者提交应用（App）到App Store上的时候，Xcode会将程序编译为一个中间表现形式（bitcode）。App Store会将该bitcode中间表现形式的代码进行编译优化，链接为64位或者32位的程序。如果程序中用到了第三方静态库，则必须在编译第三方静态库的

时候也开启bitcode，否则在Xcode的Build Setting中必须要关闭bitcode，这对于App来讲可能会造成性能的降低。

同样再看一下arm64指令集下面的编译脚本，建立build_arm64.sh文件，然后输入以下内容：

```
./configure \
--disable-shared \
--disable-frontend \
--host=arm-apple-darwin \
--prefix="./thin/arm64" \
CC="xcrun -sdk iphoneos clang -arch arm64" \
CFLAGS="-arch arm64 -fembed-bitcode -miphoneos-version-min=7.0" \
LDFLAGS="-arch arm64 -fembed-bitcode -miphoneos-version-min=7.0"
make clean
make -j8
make install
```

上述选项配置大部分都与armv7的脚本相同，不同之处在于这里的CFLAGS指定编译的目标平台是arm64的指令集平台。如果想在模拟器上运行，那么就需要编译出i386架构下的静态库，而编译i386平台的Shell脚本也与此类似，仅仅是改变平台架构。为了节省篇幅，后续讨论FDK AAC及X264的编译脚本时，将只提供armv7的编译方式。

待两个脚本执行完毕之后，就可以去thin-lame目录下寻找对应的armv7与arm64目录，并且在这两个目录下会看到bin、lib、include、share这四个目录。由于在配置的时候裁剪掉了可执行文件，所以bin目录下不会有内容；在lib目录下则是链接过程中需要链接的libmp3lame.a静态库文件；在include目录下则是编译过程所需要引用的头文件。

至此已经编译出了两个指令集平台下的静态库文件与include文件目录，其实这两个include文件的目录是一样的，随便使用哪一份都可以，但是对于静态库文件，应该如何用呢？这里就会涉及如何合并静态库的知识，合并静态库应该使用lipo命令，在终端下切换到thin-lame目录下键入：

```
lipo -create ./arm64/lib/libmp3lame.a ./armv7/lib/libmp3lame.a -output
        libmp3lame.a
```

这行命令会把两个平台架构下的静态库文件合并到一个libmp3lame.a的静态库文件中，现在来验证一下最终的libmp3lame.a是否包含armv7与arm64这两个平台架构的静态库，在命令行键入：

```
file libmp3lame.a
```

如果看到如下信息，则说明编译成功了：

```
libmp3lame.a: Mach-O universal binary with 2 architectures: [arm_v7: current
               archive] [arm64: current ar archive]
libmp3lame.a (for architecture armv7):      current ar archive
libmp3lame.a (for architecture arm64):      current ar archive
```

如果在开发过程中编译的第三方库比较多，而同时编译的指令集平台也比较多，则每次都需要新建几个脚本文件，然后编译出各个平台的静态库，最终再用lipo命令进行合并，将会非常麻烦。而软件工程师就是要把重复性的东西做成工具，让工作变得更加简单，所以后续在代码仓库中会有完整的编译脚本，可以编译出所有架构平台下的静态库，并且也已经用lipo命令把所有指令集平台下的静态库合并到了一个静态库文件中（包括即将介绍的FDK_AAC的库及X264库的交叉编译）。

2.FDK_AAC的交叉编译

在交叉编译之前先介绍一下FDK_AAC库。

FDK_AAC简介

FDK_AAC是用来编码和解码AAC格式音频文件的开源库，Android系统编码和解码AAC所用的就是这个库。开发者Fraunhofer IIS是AAC音频规范的核心制定者（MP3时代Fraunhofer IIS也是MP3规范的制定者）。前面章节中已经介绍过AAC有很多种Profile，而FDK_AAC几乎支持大部分的Profile，并且支持CBR和VBR这两种模式，根据笔者个人的听感和频谱分析，在同等码率下FDK_AAC比NeroAAC以及faac和voaac的音质都要好一些。

下面先到SourceForge上下载稳定版本的FDK_AAC：

```
https://sourceforge.net/p/opencore-amr/fdk-aac/ci/v0.1.4/tree/
```

然后在根目录下建立build_armv7.sh脚本，在里面写入以下内容：

```
./configure \
--enable-static \
--disable-shared \
--host=arm-apple-darwin \
--prefix="$FDK_ROOT_DIR/thin/armv7"
CC="xcrun -sdk iphoneos clang" \
AS="gas-preprocessor.pl $CC"
CFLAGS="-arch armv7 -mios-simulator-version-min=7.0" \
LDFLAGS="-arch armv7 -mios-simulator-version-min=7.0"
make clean
make -j8
make install
```

FDK_AAC的配置选项中要求比LAME多配置一项AS参数，并且需要安装gas-preprocessor，首先进入下方链接：

```
https://github.com/applexiaohao/gas-preprocessor
```

下载gas-preprocessor.pl，然后复制到/usr/local/bin/目录下，修改/usr/local/bin/gas-preprocessor.pl的文件权限为可执行权限：

```
chmod 777 /usr/local/bin/gas-preprocessor.pl
```

这样gas-preprocessor.pl就安装成功了，再次执行上面的Shell脚本，成功之后就可以在thin/armv7目录（当然需要提前建立好该目录）下看到include和lib这两个目录，在使用该库时，include目录下包含了编译阶段需要用到的头文件，而lib目录下包含了链接阶段需要用到的静态库文件。类似于上面的脚本，也可以编译arm64以及i386平台下的静态库，最后再用lipo工具合并静态库文件，其实也可以编写一个Shell脚本完成上述所有事情，具体可以查看代码仓库中的完整编译脚本。

3.X264的交叉编译

本节将介绍X264开源库的交叉编译，和之前的章节一样，在做交

又编译之前先向大家介绍一下X264库。

X264简介

X264是一个开源的H.264/MPEG-4 AVC视频编码函数库，是最好的有损视频编码器之一。一般的输入是视频帧的YUV表示，输出是编码之后的H264的数据包，并且支持CBR、VBR模式，可以在编码的过程中直接改变码率的设置，这在直播的场景中是非常实用的（直播场景下利用该特点可以做码率自适应）。

可以到下面这个网站上获取X264的源码：

```
http://www.videolan.org/developers/x264.html
```

当然也可以直接执行：

```
git clone git://git.videolan.org/x264.git
```

同样，在根目录下建立build_armv7.sh脚本，然后在里面写入如下内容：

```
#!/bin/sh
export AS="gas-preprocessor.pl -arch arm -- xcrun -sdk iphoneos clang"
export CC="xcrun -sdk iphoneos clang"
./configure \
--enable-static \
--enable-pic \
--disable-shared \
--host=arm-apple-darwin \
--extra-cflags="-arch armv7 -mios-version-min=7.0" \
--extra-asflags="-arch armv7 -mios-version-min=7.0" \
--extra-ldflags="-arch armv7 -mios-version-min=7.0" \
--prefix="./thin/armv7"
make clean
make -j8
make install
```

在执行上述Shell脚本之前，要求在当前目录下预先建立好thin/armv7目录，然后执行脚本，这样就可以看到在armv7目录下产生的对应的include及lib目录了，当然也有bin目录，各个目录里面存放的内

容，前面已经介绍过很多遍了，在此不再赘述。对于arm64以及i386架构平台的编译，在代码仓库中会有一个Shell脚本，它可以编译出所有架构平台下的静态库，并且已经用lipo工具合并成了一个静态库。

2.3.3 Android平台交叉编译的实践

1.深入了解Android NDK

Android原生开发包（NDK）可用于Android平台上的C++开发，NDK不仅仅是一个单一功能的工具，还是一个包含了API、交叉编译器、链接程序、调试器、构建工具等的综合工具集。

下面大致列举了一下经常会用到的组件。

·ARM、x86的交叉编译器

·构建系统

·Java原生接口头文件

·C库

·Math库

·最小的C++库

·ZLib压缩库

·POSIX线程

·Android日志库

·Android原生应用API

·OpenGL ES（包括EGL）库

·OpenSL ES库

下面来看一下Android所提供的NDK根目录下的结构。

·ndk-build：该Shell脚本是Android NDK构建系统的起始点，一般在项目中仅仅执行这一个命令就可以编译出对应的动态链接库了，后面会

有详细的介绍。

·ndk-gdb：该Shell脚本允许用GUN调试器调试Native代码，并且可以配置到Eclipse的IDE中，可以做到像调试Java代码一样调试Native的代码。

·ndk-stack：该Shell脚本可以帮助分析Native代码崩溃时的堆栈信息，后续会针对Native代码的崩溃进行详细的分析。

·build：该目录包含NDK构建系统的所有模块。

·platforms：该目录包含支持不同Android目标版本的头文件和库文件，NDK构建系统会根据具体的配置来引用指定平台下的头文件和库文件。

·toolchains：该目录包含目前NDK所支持的不同平台下的交叉编译器——ARM、x86、MIPS，其中比较常用的是ARM和x86。构建系统会根据具体的配置选择不同的交叉编译器。

在了解了NDK的目录结构之后，接下来详细了解一下NDK的编译脚本语法——Android.mk和Application.mk。

Android.mk是在Android平台上构建一个C或者C++语言编写的程序系统的Makefile文件，不同的是，Android提供了一系列的内置变量来提供更加方便的构建语法规则。Application.mk文件实际上是对应用程序本身进行描述的文件，它描述了应用程序要针对哪些CPU架构打包动态so包、要构建的是release包还是debug包以及一些编译和链接参数等。

（1）Android.mk

Android.mk分为以下几部分。

·LOCAL_PATH：=$（call my-dir），返回当前文件在系统中的路径，Android.mk文件开始时必须定义该变量。

·include$（CLEAR_VARS），表明清除上一次构建过程的所有全局变量，因为在一个Makefile编译脚本中，会使用大量的全局变量，使用这行脚本表明需要清除掉所有的全局变量。

·LOCAL_SRC_FILES，要编译的C或者Cpp的文件，注意这里不需要列举头文件，构建系统会自动帮助开发者依赖这些文件。

·LOCAL_STATIC_LIBRARIES，所依赖的静态库文件。

·LOCAL_LDLIBS：=-L$（SYSROOT）/usr/lib-llog-lOpenSLES-lGLESv2-lEGL-lz，指定编译过程所依赖的NDK提供的动态与静态库，SYSROOT变量代表的是NDK_ROOT下面的目录$NDK_ROOT/platforms/android-18/arch-arm，而在这个目录的usr/lib/目录下有很多对应的so的动态库以及.a的静态库。

·LOCAL_CFLAGS，编译C或者Cpp的编译标志，在实际编译的时候会发送给编译器。比如常用的实例是加上-DAUTO_TEST，然后在代码中就可以利用条件判断#ifdef AUTO_TEST来做一些与自动化测试相关的事情。

·LOCAL_LDFLAGS，链接标志的可选列表，当对目标文件进行链接以生成输出文件的时候，将这些标志带给链接器。该指令与LOCAL_LDLIBS有些类似，一般情况下，该选项会用于指定第三方编译的静态库，LOCAL_LDLIBS经常用于指定系统的库（比如log、OpenGL ES、EGL等）。

·LOCAL_MODULE，该模块的编译的目标名，用于区分各个模块，名字必须是唯一并且不包含空格的，如果编译目标是so库，那么该so库的名字就是lib项目名.so。

·include$（BUILD_SHARED_LIBRARY），其实类似的include还有很多，都是构建系统提供的内置变量，该变量的意义是构建动态库，其他的内置变量还包括如下几种。

·---BUILD_STATIC_LIBRARY：构建静态库。

·---PREBUILT_STATIC_LIBRARY：对已有的静态库进行包装，使其成为一个模块。

·---PREBUILT_SHARED_LIBRARY：对已有的动态库进行包装，使其成为一个模块。

·---BUILD_EXECUTABLE：构建可执行文件。

构建系统提供的这些内置变量在哪里能够看到呢？它们都在$NDK_ROOT/build/core/目录下，这里面会有所有预先定义好的Makefile，开发者include一个变量，实际上就是把对应的Makefile包含到Android.mk中，包括前面提到的CLEAR_VARS，其也是该目录下面的一个Makefile。

·include$（call all-makefiles-under，$（LOCAL_PATH）），也是构建系统提供的变量，该命令会返回该目录下所有子目录的Android.mk列表。

上面已经清楚地讲解了Android.mk里的基本语法规则，那么，在输入命令ndk-build之后，系统到底会使用哪些编译器以及打包器和链接器来编译我们的程序呢？

会使用$NDK_ROOT/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin/目录（以Mac平台为例）下面的gcc、g++、ar、ld等工具。同样在该目录下的strip工具将会用于清除so包里面的源码，nm工具可以供开发者查看静态库下的符号表。

那么进行gcc编译的时候，头文件将放在哪里呢？

$NDK_ROOT/platforms/android-18/arch-arm/usr/include/目录下会存放编译过程所依赖的头文件。

那么在链接过程中，经常使用的log或者OpenSL ES以及OpenGL ES等库又将放在哪里呢？

答案其实已在前文中提到过，$NDK_ROOT/platforms/android-18/arch-arm/usr/lib/目录下会存放链接过程中所依赖的库文件。

（2）Application.mk

Application.mk分为以下几个部分。

·APP_ABI：=XXX，这里的XXX是指不同的平台，可以选填的有x86、mips、armeabi、armeabi-v7a、all等，值得一提的是，若选择all则会构建出所有平台的so，如果不填写该项，那么将默认构建为armeabi平

台下的库。由于工作的原因，笔者和Intel的员工打过交道，构建armeabi-v7a平台的so之所以可以运行在Intel x86架构的CPU平台下，是因为Intel针对armeabi做了兼容，但是如果想要应用以最小的能耗、最高的效率运行在Intel x86平台上，则还是要指定构建的so为x86平台。因此，如果想要提高App的运行性能，则还需要编译出x86平台。类似于前面介绍的iOS平台，如果不考虑模拟器的话，则仅需要构建armv7与arm64平台架构，那么对于Android平台呢？对于armv7-a，肯定是要编译的；至于arm64-v8a这个平台，其实已经占到了50%以上，最好也将其单独编译出来；同时armv5这个平台的设备还是存在的，当然不同App在不同架构下的比例也不尽相同，读者可以根据实际场景来决定编译的平台数目。这里需要注意的是，编译arm64-v8a的时候使用的交叉工具编译链与之前的armv7所在的目录有比较大的差异，其目录存在于：

·$NDK_ROOT/toolchains/aarch64-linux-android-4.9/prebuilt/darwin-x86_64/bin

·编译过程中使用的编译工具都存在于上述目录下。

·APP_STL：=gnustl_static，NDK构建系统提供了由Android系统给出的最小C++运行时库（/system/lib/libstdc++.so）的C++头文件。然而，NDK带有另一个C++实现，开发者可以在自己的应用程序中使用或链接它，定义APP_STL可选择它们中的一个，可选项包括：stlport_static、stlport_shared、gnustl_static。

·APP_CPPFLAGS：=-std=gnu++11-fexceptions，指定编译过程的flag，可以在该选项中开启exception rtti等特性，但是为了效率考虑，最好关闭rtti。

·NDK_TOOLCHAIN_VERSION=4.8，指定交叉工具编译链里面的版本号，这里指定使用4.8。

·APP_PLATFORM：=android-9，指定创建的动态库的平台。

·APP_OPTIM：=release，该变量是可选的，用来定义“release”或“debug”，“release”模式是默认的，并且会生成高度优化的二进制代码；“debug”模式生成的是未优化的二进制代码，但是可以检测出很多的BUG，经常用于调试阶段，也相当于在ndk-build指令后边直接加上参数NDK_DEBUG=1。

2.如何交叉编译

上文讲解了NDK的结构，以及构建系统的基本语法。本节将直接对LAME、FDK_AAC、X264这三个库进行交叉编译。

（1）LAME的交叉编译

在Android的编译中，一般情况下会使用一个Shell脚本文件，指定好编译器里面的各个工具，然后把对应的Configure的命令与选项开关配置好，最后执行该Shell脚本：

```
#!/bin/bash
NDK_ROOT=/Users/apple/soft/android/android-ndk-r9b
PREBUILT=$NDK_ROOT/toolchains/arm-linux-androideabi-4.6/prebuilt/darwin-x86_6
PLATFORM=$NDK_ROOT/platforms/android-9/arch-arm
export PATH=$PATH:$PREBUILT/bin:$PLATFORM/usr/include:

export LDFLAGS="-L$PLATFORM/usr/lib -L$PREBUILT/arm-linux-androideabi/lib
            -march=armv7-a"
export CFLAGS="-I$PLATFORM/usr/include -march=armv7-a -mfloat-abi=softfp -mfp
       -ffast-math -O2"

export CPPFLAGS="$CFLAGS"
export CFLAGS="$CFLAGS"
export CXXFLAGS="$CFLAGS"
export LDFLAGS="$LDFLAGS"

export AS=$PREBUILT/bin/arm-linux-androideabi-as
export LD=$PREBUILT/bin/arm-linux-androideabi-ld
export CXX="$PREBUILT/bin/arm-linux-androideabi-g++ --sysroot=${PLATFORM}"
export CC="$PREBUILT/bin/arm-linux-androideabi-gcc --sysroot=${PLATFORM}
       -march=armv7-a "
export NM=$PREBUILT/bin/arm-linux-androideabi-nm
export STRIP=$PREBUILT/bin/arm-linux-androideabi-strip
export RANLIB=$PREBUILT/bin/arm-linux-androideabi-ranlib
export AR=$PREBUILT/bin/arm-linux-androideabi-ar

./configure --host=arm-linux \
--disable-shared \
--disable-frontend \
--enable-static \
--prefix=./armv7a

make clean
make -j8
make install
```

下面就来针对该脚本的每一行命令进行详细解释。

第一部分是设置NDK_ROOT，并且声明platform和prebuilt，最终配置可在环境变量中查看。

第二部分主要是声明CFLAGS与LDFLAGS，其目的是在编译和链接阶段找到正确的头文件与链接到正确的库文件。这里需要特别注意的是，在这两个设置的后边都加上了-march=armv7-a，这相当于是让编译器知道要编译的目标平台是armv7-a。

第三部分是声明CC、AS、AR、LD、NM、STRIP等工具，具体每一个工具是做什么用的，前面都已经介绍过了，如果要编译armv5、x86或者arm64-v8a，那么在代码仓库中会提供全量编译的Shell脚本文件。

第四部分就是使用LAME本身的Configure进行编译裁剪。

第五部分就是使用标准的编译链接和安装。

最终执行脚本成功之后，可以看到在指定的Prefix目录下面，包含了lib和include目录，里面分别是静态库文件和头文件，这两个目录的作用在前面已经说过很多遍了，在此不再赘述。

（2）FDK_AAC的交叉编译

```
#!/bin/bash
./configure --host=armv7a \
--enable-static \
--disable-shared \
--prefix=./armv7a/

make clean
make -j8
make install
```

注：这里没有给出声明NDK_ROOT以及各个编译工具的代码。

其实FDK_AAC的交叉编译并没有什么可解说的，配置好环境变量之后，执行Configure，然后安装就可以了。

（3）X264的交叉编译

```
#!/bin/bash
```

```
./configure --prefix=$PREFIX \
--enable-static \
--enable-pic \
--enable-strip \
--disable-cli \
--disable-asm \
--extra-cflags="-march=armv7-a -O2 -mfloat-abi=softfp -mfpu=neon" \
--host=arm-linux \
--cross-prefix=$PREBUILT/bin/arm-linux-androideabi- \
--sysroot=$PLATFORM
```

注：这里没有给出声明NDK_ROOT以及各个编译工具的代码。

对于X264的编译裁剪，这里有如下几个关键点。

·extra-cflags选项的配置。针对armv7-a的CPU打开了NEON的优化运行指令，并且打开了O2编译优化，这是非常重要的一点。

·--disable-asm选项的配置。如果不禁用掉asm指令，则意味着将会禁止neon的指令。

2.3.4 使用LAME编码MP3文件

2.3.3节为两个平台交叉编译了多个库，本节就以上文编译出来的LAME库进行编码工作。本节要实现的目标是，在添加好C++支持的项目中加入编码MP3文件的功能。当点击按钮的时候，输入的是一个PCM文件的路径和一个MP3的路径，等运行完毕，电脑上的播放器直接就可以播放该MP3文件。

首先，用一个C++的类来实现其业务逻辑，输入就是两个char指针类型的路径，编码成功之后，输出一行编码成功的Log，然后在之前的项目基础上集成进这段代码，再运行并测试。

1.编码工具类的编写

首先新建两个文件：mp3_encoder.h和mp3_encoder.cpp。

先看一下头文件应该如何编写，头文件其实就是用于定义该类对外提供的接口。

这里提供的是一个Init接口，输入的是一个PCM FilePath和一个MP3 FilePath，会判定输入文件是否存在、初始化LAME以及初始化输出文件的资源，返回值是该函数是否成功初始化了所有的相关资源，成功则返回true，否则返回false。

此外，还要再提供一个encode方法，负责读取PCM数据，并且调用LAME进行编码，然后将编码之后的数据写入文件。

最后再对外提供一个销毁资源的接口destroy方法，用于关闭所有的资源。

按照上述分析，建立头文件如下：

```
class Mp3Encoder {
    private:
        FILE* pcmFile;
        FILE* mp3File;
        lame_t lameClient;
```

```
    public:
        Mp3Encoder();
        ~Mp3Encoder();
        int Init(const char* pcmFilePath, const char *mp3FilePath, int
                sampleRate, int channels, int bitRate);
        void Encode();
        void Destory();
};
```

既然头文件已经定义好了，接下来就一起来实现。

首先是Init方法的实现，以读二进制文件的方式打开PCM文件，以写入二进制文件的方式打开MP3文件，然后初始化LAME。具体代码如下：

```
int Mp3Encoder::Init(const char* pcmFilePath, const char *
        mp3FilePath, int sampleRate, int channels, int bitRate) {
    int ret = -1;
    pcmFile = fopen(pcmFilePath, "rb");
    if(pcmFile) {
        mp3File = fopen(mp3FilePath, "wb");
        if(mp3File) {
            lameClient = lame_init();
            lame_set_in_samplerate(lameClient, sampleRate);
            lame_set_out_samplerate(lameClient, sampleRate);
            lame_set_num_channels(lameClient, channels);
            lame_set_brate(lameClient, bitRate / 1000);
            lame_init_params(lameClient);
            ret = 0;
        }
    }
    return ret;
}
```

其次是Encode方法的实现，函数主体是一个循环，每次都会读取一段bufferSize大小的PCM数据buffer，然后再编码该buffer，但是在编码buffer之前得把该buffer的左右声道拆分开，再送入到LAME编码器，最后将编码之后的数据写入MP3文件中。具体代码如下：

```
void Mp3Encoder::Encode() {
    int bufferSize = 1024 * 256;
    short* buffer = new short[bufferSize / 2];
    short* leftBuffer = new short[bufferSize / 4];
    short* rightBuffer = new short[bufferSize / 4];
    unsigned char* mp3_buffer = new unsigned char[bufferSize];
    size_t readBufferSize = 0;
    while ((readBufferSize = fread(buffer, 2, bufferSize / 2, pcmFile)) > 0)
```

```
            for (int i = 0; i < readBufferSize; i++) {
                if (i % 2 == 0) {
                    leftBuffer[i / 2] = buffer[i];
                } else {
                    rightBuffer[i / 2] = buffer[i];
                }
            }
            size_t wroteSize = lame_encode_buffer(lameClient, (short
                    int *) leftBuffer, (short int *) rightBuffer,
                    (int)(readBufferSize / 2), mp3_buffer, bufferSize);
            fwrite(mp3_buffer, 1, wroteSize, mp3File);
        }
        delete[] buffer;
        delete[] leftBuffer;
        delete[] rightBuffer;
        delete[] mp3_buffer;
    }
```

最后是Destroy方法，关闭PCM文件，关闭MP3文件，销毁LAME。具体代码如下：

```
void Mp3Encoder::Destory() {
    if(pcmFile) {
        fclose(pcmFile);
    }
    if(mp3File) {
        fclose(mp3File);
        lame_close(lameClient);
    }
}
```

实现结束之后，接下来就是把该类集成到iOS和Android客户端。

2.iOS集成

首先打开2.1节开发的项目——添加了C++支持的iOS工程。

然后在ViewController.mm中直接实例化C++类型的类Mp3Encoder，将vocal.pcm的文件放入沙盒中，利用下面这行代码获取PCM的路径：

```
[[[NSBundle mainBundle] bundlePath] stringByAppendingPathComponent: @"vocal.p
```

接着利用下面的代码获取最终要写入的MP3文件的路径：

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
              NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
[documentsDirectory stringByAppendingPathComponent:@"vocal.mp3"];
```

最后直接调用Mp3Encoder类的encode方法，编码音频文件。最终编码方法执行结束，不要忘记调用该C++类的destroy方法，销毁所有资源。然后运行程序，点击编码按钮，等待编码结束之后，通过Xcode/device中的Download沙盒功能，将MP3文件提取出来，读者可以使用电脑默认的播放器播放该MP3文件，试试看是否能正常播放。

3.Android集成

首先打开2.2节开发的项目——添加了C++支持的Android工程。

然后在com.phuket.tour.studio包下修改Mp3Encoder文件，并写入三个native方法：

```
public native int init(String pcmPath, int audioChannels, int bitRate, int
        sampleRate, String mp3Path);
public native void encode();
public native void destroy();
```

接着重新生成JNI接口文件，并且在实现的C++文件中实现这三个方法。

在init方法中实例化Mp3Encoder类，然后调用初始化方法：

```
const char* pcmPath = env->GetStringUTFChars(pcmPathParam, NULL);
const char* mp3Path = env->GetStringUTFChars(mp3PathParam, NULL);
encoder = new Mp3Encoder();
encoder->Init(pcmPath, mp3Path, sampleRate, channels, bitRate);
env->ReleaseStringUTFChars(mp3PathParam, mp3Path);
env->ReleaseStringUTFChars(pcmPathParam, pcmPath);
```

在encode方法中直接调用Mp3Encoder的encode方法；在destroy方法中则调用Mp3Encoder的destroy方法；最后，修改Android.mk文件，将最新的C++文件加入LOCAL_SRC中，并且在include的路径中增加LAME的头文件所在的路径：

```
LOCAL_C_INCLUDES := \
        $(LOCAL_PATH)/../3rdparty/lame/include
```

当然，还需要在链接阶段加入编译出来的静态库，因此需要键入以下命令：

```
LOCAL_LDLIBS += -L$(LOCAL_PATH)/3rdparty/lame/lib -lmp3lame
```

这样重新执行ndk-build命令就可以将最新的so库打出来了。[1]

现在，在MainActivity中，传入正确的输入路径和输出路径（需要提前准备一段PCM数据放入输入路径下面），等编码结束之后，取出输出路径的MP3文件，在电脑上播放，确认一下是否正确。

[1] so库是gcc或g++编译过后形成的一种动态库，最终可以被加载到程序中使用。

2.4 本章小结

本章主要介绍了如何创建一个Android项目和一个iOS项目，然后为这两个项目增加C++支持，接着讨论了交叉编译，最后利用交叉编译出来的LAME库，在两个平台上完成编码MP3音频文件的功能。本章的内容是音视频开发的基础内容，希望读者熟练掌握。

第3章　FFmpeg的介绍与使用

若要讲解音视频的开发，首先不得不提开源框架FFmpeg。该开源框架为音视频开发者们提供了非常大的帮助，其也是全世界的音视频开发工程师都应该掌握的工具。FFmpeg是一套可以用来记录、处理数字音频、视频，并将其转换为流的开源框架，采用LPL或GPL许可证，提供了录制、转换以及流化音视频的完整解决方案。它的可移植性或者说跨平台特性非常强大，可以用在Linux服务器、PC（包括Windows、Mac OS X等）、移动端设备（Android、iOS等移动设备）等平台。名称中的mpeg来自视频编码标准MPEG，而前缀FF是Fast Forward的首字母缩写。本章会从编译开始讲解，然后介绍命令行工具的使用，接着会介绍FFmpeg在代码层面提供给开发者的API，最后会从源码的角度分析一下整个FFmpeg框架，现在就让我们开始吧。

3.1 FFmpeg的编译与命令行工具的使用

3.1.1 FFmpeg的编译

1.FFmpeg编译选项详解

首先到FFmpeg官网上下载稳定版本的FFmpeg源码，本章将会从下载到的最干净的代码开始逐步进行操作。然后将下载的源码解压到一个目录中，FFmpeg与大部分GNU软件的编译方式类似，都是通过configure脚本来实现编译前定制的，这种方式允许用户在编译前对软件进行裁剪，同时通过对最终运行到的系统以及目标平台的配置来决定对某些模块设定合适的配置。configure脚本运行完毕之后，会生成config.mk和config.h这两个文件，分别作用到makefile和源代码的层次，由这两个部分协同实现对编译选项的控制。所以下面先来看看configure的脚本，可以利用它的help命令来查看其到底提供了哪些选项？

```
./configure -help
```

·标准选项：GNU软件例行配置项目，例如安装路径、--prefix=…等。

·编译、链接选项：默认配置是生成静态库而不是生成动态库，例如--disable-static、--enable-shared等。

·可执行程序控制选项：决定是否生成FFmpeg、ffplay、ffprobe和ffserver等。

·模块控制选项：裁剪编译模块，包括整个库的裁剪，例如--disable-avdevice；一组模块的筛选，例如--disable-decoders；单个模块的裁剪，例如--disable-demuxer。

·能力展示选项：列出当前源代码支持的各种能力集，例如--list-decoders、--list-encoders。

·其他：允许开发者深度定制，如交叉编译环境配置、自定义编译

器参数的设定等。

下面先给出一个总览，大体了解一下FFmpeg的整体结构，如图3-1所示。

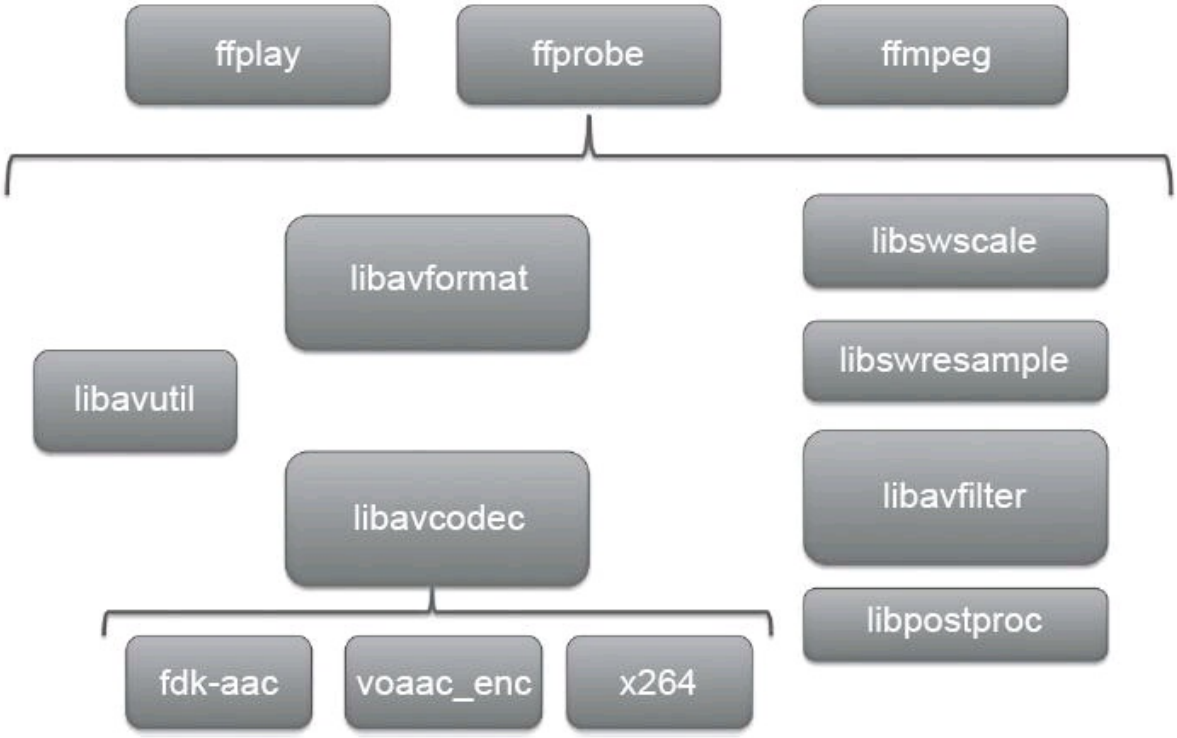


图　3-1

下面这段代码是一个配置实例，用于实现运行于Android系统中的FFmpeg库的编译：

```
./configure -prefix=. \
--cross-prefix=$NDK_TOOLCHAIN_PREFIX \
--enable-cross-compile \
--arch=arm --target-os=linux \
--disable-static -enable-shared \
--disable-ffmpeg --disable-ffplay -disable-ffserver -disable-ffprobe
```

默认的编译会生成4个可执行文件和8个静态库。可执行文件包括用于转码、推流、Dump媒体文件的ffmpeg、用于播放媒体文件的ffplay、用于获取媒体文件信息的ffprobe，以及作为简单流媒体服务器的ffserver。

8个静态库其实就是FFmpeg的8个模块，具体包括如下内容。

·AVUtil：核心工具库，该模块是最基础的模块之一，下面的许多其他模块都会依赖该库做一些基本的音视频处理操作。

·AVFormat：文件格式和协议库，该模块是最重要的模块之一，封装了Protocol层和Demuxer、Muxer层，使得协议和格式对于开发者来说是透明的。

·AVCodec：编解码库，该模块也是最重要的模块之一，封装了Codec层，但是有一些Codec是具备自己的License的，FFmpeg是不会默认添加像libx264、FDK-AAC、lame等库的，但是FFmpeg就像一个平台一样，可以将其他的第三方的Codec以插件的方式添加进来，然后为开发者提供统一的接口。

·AVFilter：音视频滤镜库，该模块提供了包括音频特效和视频特效的处理，在使用FFmpeg的API进行编解码的过程中，直接使用该模块为音视频数据做特效处理是非常方便同时也非常高效的一种方式。

·AVDevice：输入输出设备库，比如，需要编译出播放声音或者视频的工具ffplay，就需要确保该模块是打开的，同时也需要libSDL的预先编译，因为该设备模块播放声音与播放视频使用的都是libSDL库。

·SwrRessample：该模块可用于音频重采样，可以对数字音频进行声道数、数据格式、采样率等多种基本信息的转换。

·SWScale：该模块是将图像进行格式转换的模块，比如，可以将YUV的数据转换为RGB的数据。

·PostProc：该模块可用于进行后期处理，当我们使用AVFilter的时候需要打开该模块的开关，因为Filter中会使用到该模块的一些基础函数。

如果是比较老的FFmpeg版本，那么有可能还会编译出来avresample模块，该模块其实也是用于对音频原始数据进行重采样，但是现在已经被废弃掉了，不再推荐使用该库，而是使用swrresample库进行替代。

如何为FFmpeg平台引入第三方编解码库呢？下面就以最常用的

LAME、X264、FDK-AAC进行举例。前面的章节中已经介绍了这三个库在Android和iOS平台上的交叉编译，现在就假设已经交叉编译出了LAME、X264、FDK-AAC的静态库与头文件，并且在FFmpeg的源码目录下建立了external-libs目录，还在其中建立了LAME、X264、FDK-AAC三个目录，每个目录中的结构都包含了include和lib两个目录，并且将编译出来的头文件和静态库文件分别都放到了这两个目录下面。

现在修改编译脚本如下。

新增X264编码器需要新增以下脚本：

```
--enable-muxer=h264 \
--enable-encoder=libx264 \
--enable-libx264 \
--extra-cflags=”-Iexternal-libs/x264/include” \
--extra-ldflags=”-Lexternal-libs/x264/lib” \
```

新增LAME编码器需要新增以下脚本：

```
--enable-muxer=mp3 \
--enable-encoder=libmp3lame \
--enable-libmp3lame \
--extra-cflags=”-Iexternal-libs/lame/include” \
--extra-ldflags=”-Lexternal-libs/lame/lib” \
```

新增FDK-AAC编码器需要新增以下脚本：

```
--enable-encoder=libfdk_aac \
--enable-libfdk_aac \
--extra-cflags=”-Iexternal-libs/fdk-aac/include” \
--extra-ldflags=”-Lexternal-libs/fdk-aac/lib” \
```

读者可以按照自己的应用场景，把需要编译进来的第三方库以修改脚本文件的方式进行编译，然后以命令行模式或者以API调用的方式进行使用。

在FFmpeg中，有一个类型的filter称为bit stream filter，要想在开发过程中使用该filter，则需要在编译的过程中打开它。该filter存在的意义主要是应对某些格式的封装转换行为。比如AAC编码，常见的有两种封

装格式：一种是ADTS格式的流，是AAC定义在MPEG2里面的格式；另外一种是封装在MPEG4里面的格式，这种格式会在每一帧前面拼接一个用声道、采样率等信息组成的头。开发者完全可以手动拼接该头信息，即将AAC编码器输出的原始码流（ADTS头+ES流）封装进MP4、FLV或者MOV等格式的容器中时，需要先将ADTS头转换为MPEG-4 AudioSpecficConfig（描述了编码器的配置参数）头，并去掉原始码流中的ADTS头（只剩下ES流）。但是使用FFmpeg提供好的aac_adtstoasc类型的bit stream filter可以非常方便地进行转换，FFmpeg为开发者隐藏了实现的细节，并且提供了更好的代码可读性。若想要正常使用这个filter，则需要在编译过程中打开下面这个选项：

```
--enable-bsf=aac_adtstoasc
```

AAC的bit stream filter常常应用在编码的过程中。与音频的AAC编码格式相对应的是视频中的H264编码，它也有两种封装格式：一种是MP4封装的格式；一种是裸的H264格式（一般称为annexb封装格式）。FFmpeg中也提供了对应的bit stream filter，称为H264_mp4toannexb，可以将MP4封装格式的H264数据包转换为annexb封装格式的H264数据（其实就是裸的H264的数据）包。当然，也可以手动写代码来实现这件事情，但是既然FFmpeg提供了这样好用的模块，我们为什么不用呢？要使用它也只需要在编译过程中打开下面这个选项即可：

```
--enable-bsf=h264_mp4toannexb
```

H264的bit stream filter常常应用于视频解码过程中，特别是后期在讲解使用各个平台上提供的硬件解码器时，一定会用到该bit stream filter。

2.FFmpeg的交叉编译

第2章已经介绍了交叉编译的原理，并且交叉编译出了LAME、FDK_AAC、X264等第三方库，本章也已经介绍了FFmpeg中Configure的大部分关键语法与定义，因此这里就直接开始交叉编译FFmpeg吧！不过，对应于Android和iOS平台会有一些通用的配置选项，这里先列出通用的配置选项，代码如下：

```
CONFIGURE_FLAGS=--disable-shared \
--enable-static \
--disable-stripping \
--disable-ffmpeg \
--disable-ffplay \
--disable-ffserver \
--disable-ffprobe \
--disable-avdevice \
--disable-devices \
--disable-indevs \
--disable-outdevs \
--disable-debug \
--disable-asm \
--disable-yasm \
--disable-doc \
--enable-small \
--enable-dct \
--enable-dwt \
--enable-lsp \
--enable-mdct \
--enable-rdft \
--enable-fft \
--enable-version3 \
--enable-nonfree \
--disable-filters \
--disable-postproc \
--disable-bsfs \
--enable-bsf=aac_adtstoasc \
--enable-bsf=h264_mp4toannexb \
--disable-encoders \
--enable-encoder=pcm_s16le \
--enable-encoder=aac \
--enable-encoder=libvo_aacenc \
--disable-decoders \
--enable-decoder=aac \
--enable-decoder=mp3 \
--enable-decoder=pcm_s16le \
--disable-parsers \
--enable-parser=aac  \
--disable-muxers \
--enable-muxer=flv \
--enable-muxer=wav \
--enable-muxer=adts \
--disable-demuxers \
--enable-demuxer=flv \
--enable-demuxer=wav \
--enable-demuxer=aac \
--disable-protocols \
--enable-protocol=rtmp \
--enable-protocol=file \
--enable-libfdk_aac \
--enable-libx264 \
--enable-cross-compile \
--prefix=$INSTALL_DIR
```

可以看到为了达到最小的包体积，需要先关掉所有的模块，然后再打开具体的编解码器、解析器、解复用器、协议，并且这里开启了两个第三方的Codec：一个是FDK_AAC；另外一个是X264。此处还关闭了命令行工具与帮助文档、输入输出设备、动态库生成，同时还打开了静态库的生成。由于要进行交叉编译，所以要在倒数第二行打开交叉编译的选项，在最后一行指定安装库的目标目录。

对于Android平台来讲，Shell脚本需要添加如下内容：

```
ANDROID_NDK_ROOT=/Users/apple/soft/android/android-ndk-r9b
PREBUILT=$ANDROID_NDK_ROOT/toolchains/arm-linux-androideabi-4.8/prebuilt/darw
PLATFORM=$ANDROID_NDK_ROOT/platforms/android-8/arch-arm
./configure \
$CONFIGURE_FLAGS \
--target-os=linux \
--arch=arm \
--cross-prefix=$PREBUILT/bin/arm-linux-androideabi- \
--sysroot=$PLATFORM \
--extra-cflags="-marm -march=armv7-a -Ifdk_aac/include -Ix264 /include" \
--extra-ldflags="-marm -march=armv7-a -Lfdk_aac/lib -Lx264 /lib"
```

可以看到，上述脚本中指定了运行的平台与架构，指定了编译器、链接器的前缀，以用于执行真正的编译与链接操作，然后给出sysroot和编译、链接参数。这里是以armv7a作为编译目标架构的，如果读者想自行编译armv8或者x86的平台，可以查看代码仓库中的编译脚本，这里不再列举。

对于iOS平台来讲，Shell脚本需要添加如下内容：

```
./configure \
$CONFIGURE_FLAGS \
--target-os=darwin
--cc=xcrun -sdk iphoneos clang \
--arch=armv7 \
--extra-cflags="-arch armv7 -mios-version-min=7.0 -Ifdk_aac/include
    -Ix264/include" \
--extra-ldflags="-arch armv7 -mios-version-min=7.0 -Lfdk_aac/lib -Lx264 /lib'
```

可以看到，上述脚本中设置了编译器以及目标运行平台，需要说明的是，这里指定了最小的iOS的运行平台是7.0，否则将这个库集成到Xcode中的时候会遇到平台不匹配的警告，此外，需要注意的是这里并

没有打开bitcode，这会导致集成进入Xcode之后必须将项目的bitcode选项关闭掉。若要为编译的库打开bitcode选项，那么在编译参数中增加下面这行参数就可以了：

```
-fembed-bitcode
```

3.FFmpeg命令行工具的编译与安装

在介绍FFmpeg的命令行之前，应先安装FFmpeg，前面已经介绍过了FFmpeg的编译，但那是基于Android平台的交叉编译，而不是安装在PC上的工具，虽然有些读者可能会说我们做的是移动端上的开发，和PC上的FFmpeg有什么关系呢？但是不可否认的是，开发者用于开发的机器上有一套强大的音视频工具，对开发移动端上的音视频项目是非常重要的。相信有了上文的介绍，在这里介绍FFmpeg的配置与安装应该会很轻松。

下面给出一个编译脚本config_pc.sh：

```
#!/bin/bash
./configure \
--enable-gpl \
--disable-shared \
--disable-asm \
--disable-yasm \
--enable-filter=aresample \
--enable-bsf=aac_adtstoasc \
--enable-small \
--enable-dct \
--enable-dwt \
--enable-lsp \
--enable-mdct \
--enable-rdft \
--enable-fft \
--enable-static \
--enable-version3 \
--enable-nonfree \
--enable-encoder=libfdk_aac \
--enable-encoder=libx264 \
--enable-decoder=mp3 \
--disable-decoder=h264_vda \
--disable-d3d11va \
--disable-dxva2 \
--disable-vaapi \
--disable-vda \
--disable-vdpau \
--disable-videotoolbox \
```

```
--disable-securetransport \
--enable-libx264 \
--enable-libfdk_aac \
--enable-libmp3lame \
--extra-cflags="-Ipc_fdk_aac/include -Ix264_pc/include -Ipc_lame/include" \
--extra-ldflags="-Lpc_fdk_aac/lib -Lx264_pc/lib -Lpc_lame/lib" \
--prefix='/Users/apple/Desktop/ffmpegtmp_1'
```

如果该文件没有执行权限，那么请执行以下命令为该文件增加执行权限：

```
chmod a+x ./config_pc.sh
```

然后就可以执行该Shell脚本文件，对FFmpeg进行配置了：

```
./config_pc.sh
```

该脚本执行结束之后，就来执行以下命令进行编译与安装：

```
make && make install
```

安装结束之后，进入到prefix指定的目录下查看，具体会看到如下几个目录。

·bin：编译结束的命令行工具所在的目录，后文会详细介绍该目录下的工具。

·include：编译结束的头文件都存放在该目录下面，如果要以编写代码的方式调用FFmpeg的API去完成工作（这也是后面会介绍的内容），就需要把include中的目录放到includes的配置中（Android下的makefile文件）或者Header Search Path中（iOS平台下工程文件的配置选项）。

·lib：其中存放的是编译出来的静态库文件，其在以编写代码的方式调用FFmpeg的API时会使用到，在编译阶段会使用到上一步提到的include目录，而在链接阶段则会使用到这个lib目录下面的静态库了。

·share：该目录中存放了一些examples，其中展示了如何使用代码的方式调用FFmpeg的API，其实可以切换到configure脚本所在的目录，然后执行make examples命令及make install，再到doc下面的example里找到对应的二进制文件，这样就可以进行调试或者写出自己的测试程序了。

有的读者可能会发现一个问题，在bin目录的下面没有ffplay，这又是为什么呢？因为ffplay实际上是客户端ffplay.c的C程序编译出来的，该ffplay.c需要依赖avdevice模块，而avdevice模块使用了sdl的API，如果你的PC上没有sdl（1.x版本，最常用的就是1.2.0），那么ffplay就会编译不出来了。所以要想编译出命令行工具ffplay，首先得编译基础库sdl，可以在自己的PC上利用安装软件包工具进行安装，以Mac OS系统为例，使用brew进行安装，如果没有brew的话，则首先安装brew，可以执行下面命令进行Homebrew的安装：

```
ruby -e "$(curl -fsSL \
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

等待一段时间，brew就安装好了，之后即可用brew安装sdl，执行下述命令：

```
brew install sdl
```

等待下载并且安装完毕之后，重新执行上述FFmpeg的配置和安装步骤，待make install结束之后，再去bin目录下就可以找到命令行工具ffplay了。

当然，还可以使用另外一种方式为开发机器安装FFmpeg，即通过安装包管理工具的方式进行安装。在Mac OS系统上直接在命令行下键入以下命令：

```
brew install ffmpeg
```

可以看到brew会先下载X264作为视频的编码库，并安装成功，然后就可以直接使用工具FFmpeg了，但是却没有工具ffplay，如果想安装

ffplay，那么执行如下命令：

```
brew uninstall ffmpeg
brew install ffmpeg --with-ffplay
```

可以看到brew会先下载sdl，然后再安装ffplay，注意使用brew安装的FFmpeg已经是3.0以上的版本，并且使用的sdl也已经是2.0版本了。

至此，关于FFmpeg的编译部分就介绍完毕了，现在回顾一下本节的内容，本节主要介绍了如何控制FFmpeg各个模块的开关，还介绍了如何将第三方编解码器编译到FFmpeg平台中，接着介绍了bit stream filter类型的过滤器，然后把FFmpeg交叉编译到Android平台和iOS平台，最后在PC上成功编译出FFmpeg。3.1.2节将会介绍编译出来的FFmpeg命令行工具，以及在工作中如何使用这些工具来提高处理音视频的效率。

3.1.2　FFmpeg命令行工具的使用

前面讲解了如何安装FFmpeg相关的命令行，其中涉及ffmpeg、ffprobe、ffplay以及ffserver等命令行工具，本节将重点介绍ffmpeg、ffprobe与ffplay这三个命令行工具，而ffserver则是作为简单的流媒体服务器存在的，与客户端开发关系不大，因此本书将不做介绍。前文曾经提到ffmpeg是进行媒体文件转码的命令行工具，ffprobe是用于查看媒体文件头信息的工具，ffplay则是用于播放媒体文件的工具。

下面按照从简单开始的原则，先介绍ffprobe——查看媒体文件格式的工具。

1.ffprobe

首先用ffprobe查看一个音频的文件：

```
ffprobe ~/Desktop/32037.mp3
```

键入上述命令之后，先看如下这行信息：

```
Duration：00:05:14.83，start：0.000000，bitrate：64kb/s
```

这行信息表明，该音频文件的时长是5分14秒零830毫秒，开始播放时间是0，整个媒体文件的比特率是64Kbit/s，然后再看另外一行：

```
Stream#0:0 Audio：mp3，24000Hz，stereo，s16p，64kb/s
```

这行信息表明，第一个流是音频流，编码格式是MP3格式，采样率是24kHz，声道是立体声，采样表示格式是SInt16（short）的planner（平铺格式），这路流的比特率是64Kbit/s。

然后再使用ffprobe查看一个视频的文件：

```
ffprobe ~/Desktop/32037.mp4
```

键入上述命令之后，可以看到第一部分的信息是Metadata信息：

```
Metadata:
    major_brand: isom
    minor_version: 512
    compatible_brands: isomiso2avc1mp41
    encoder: Lavf55.12.100
```

这行信息表明了该文件的Metadata信息，比如encoder是Lavf55.12.100，其中Lavf代表的是FFmpeg输出的文件，后面的编号代表了FFmpeg的版本代号，接下来的一行信息如下：

```
Duration：00:04:34.56 start：0.023220，bitrate：577kb/s
```

上面一行的内容表示Duration是4分34秒560毫秒，开始播放的时间是从23ms开始播放的，整个文件的比特率是577Kbit/s，紧接着再来看下一行：

```
Stream#0:0（un）：Video：h264（avc1/0x31637661），yuv420p，480*480，508kb/s，24f
```

这行信息表示第一个stream是视频流，编码方式是H264的格式（封装格式是AVC1），每一帧的数据表示是YUV420P的格式，分辨率是480×480，这路流的比特率是508Kbit/s，帧率是每秒钟24帧（fps是24），紧接着再来看下一行：

```
Stream#0:1（und）：Audio：aac（LC）（mp4a/0x6134706D），44100Hz，stereo，fltp，63
```

这行信息表示第二个stream是音频流，编码方式是AAC（封装格式是MP4A），并且采用的Profile是LC规格，采样率是44100Hz，声道数是立体声，数据表示格式是浮点型，这路音频流的比特率是63Kbit/s。

以上就是使用ffprobe来提取音频文件和视频文件头信息的方式，以及提取出来信息的含义。当然，ffprobe还有比较高级的用法，下面就来

介绍几个：

```
ffprobe -show_format 32037.mp4
```

上述命令可以输出格式信息format_name、时间长度duration、文件大小size、比特率bit_rate、流的数目nb_streams等。

```
ffprobe -print_format json -show_streams 32037.mp4
```

上述命令可以以JSON格式的形式输出具体每一个流最详细的信息，视频中会有视频的宽高信息、是否有b帧、视频帧的总数目、视频的编码格式、显示比例、比特率等信息，音频中会有音频的编码格式、表示格式、声道数、时间长度、比特率、帧的总数目等信息。

显示帧信息的命令如下：

```
ffprobe -show_frames sample.mp4
```

查看包信息的命令如下：

```
ffprobe -show_packets sample.mp4
```

观影小技巧

日常生活中经常会接触到多媒体文件，比如，在电脑上利用ffprobe工具打开一些国粤双语的文件，一般会看到如下3行Stream。

视频Stream：h264 yuv420P

音频Stream：aac 48000Hz stereo fltp（default）title：粤语

音频Stream：aac 48000Hz stereo fltp title：国语

这就是说明，该媒体文件中有三路流：一路是视频流，另外两路是

音频流，默认播放的是粤语的声音流，在大多数的播放器里面都可以进行音频流的切换，可以切换到国语的声音流进行观看。

以上介绍的基本上就是日常工作中经常会使用到的ffprobe命令了，其实大家只需要掌握最重要的查看指令就可以了，下面继续来看下一个命令行工具ffplay。

2.ffplay

前文已经提到过，ffplay是以FFmpeg框架为基础，外加渲染音视频的库libSDL来构建的媒体文件播放器。它所依赖的libSDL是1.2版本的，所以在安装ffplay之前也要安装对应版本的libSDL作为其依赖的组件。之后使用ffplay就非常简单了，比如我们要播放一个音频文件：

```
ffplay 32037.mp3
```

这时候会弹出一个窗口，一边播放MP3文件，一边将播放声音的语谱图画到该窗口上。针对该窗口的操作如下，点击窗口的任意一个位置，ffplay会按照点击的位置计算出时间的进度，然后跳（seek）到这个时间点上继续播放；按下键盘上的右键会默认快进10s，左键默认后退10s，上键默认快进1min，下键默认后退1min；按ESC键就是退出播放进程；如果按w键则将绘制音频的波形图等。播放一个视频的命令如下所示：

```
ffplay 32037.mp4
```

这时候会直接在新弹出的窗口上播放该视频，如果想要同时播放多个文件，那么只需要在多个命令行下同时执行ffplay就可以了，在对比多个视频质量的时候这是一个操作技巧，此外，如果按s键则可以进入frame-step模式，即按s键一次就会播放下一帧图像，这在观察某些视频内部的帧内容时也是常用的技巧。

业界内开源的ijkPlayer其实就是基于ffplay进行改造的播放器，当然其做了硬件解码以及很多兼容性的工作。ijkPlayer是一款非常优秀的播放器，作为开发者的我们需要很多优秀的开源项目。所以在这里笔者呼吁各家互联网公司开源出自己的部分代码，以提高所在领域的整体水

平。

更多的ffplay命令介绍如下：

```
ffplay 32037.mp4 -loop 10
```

上述命令代表播放视频结束之后会从头再次播放，共循环播放10次。还记得前文中提到过的两路流吗？ffplay也做了这方面的适配，也就是说在ffplay中其实也可以指定使用哪一路音频流或者视频流，命令如下：

```
ffplay 大话西游.mkv -ast 1
```

上述命令表示播放视频中的第一路音频流，如果参数ast后面跟的是2，那么就播放第二路音频流，如果没有第二路音频流的话，就会静音，同样也可以设置参数vst，比如：

```
ffplay 大话西游.mkv -vst 1
```

上述命令表示播放视频中的第一路视频流，如果参数vst后面跟的是2，那么就播放第二路视频流，但是如果没有第二路视频流，就会是黑屏即什么都不显示。

接下来介绍开发工作中常用的几个命令，这些命令在工作中debug的时候非常有用。首先用ffplay播放裸数据，无论是音频的pcm文件还是视频帧原始格式表示的数据（YUV420P或者rgba）。下面先来看看音频pcm文件的播放命令：

```
ffplay song.pcm -f s16le -channels 2 -ar 44100
```

仅键入上述这行命令其实就可以正常播放song.pcm了，当然，前提是格式（-f）、声道数（-channels）、采样率（-ar）必须设置正确，如果其中任何一项参数设置不正确，都不会得到正常的播放结果。第1章在讲音频的基础概念时已经提到过，WAV格式的文件称为无压缩的格

式，其实就是在PCM的头部添加44个字节，用于标识这个PCM的采样表示格式、声道数、采样率等信息，对于WAV格式音频文件，ffplay肯定可以直接播放，但是若让ffplay播放PCM裸数据的话，只要为其提供上述三个主要的信息，那么它就可以正确地播放了。

然后再来看一帧视频帧的播放，首先是YUV420P格式的视频帧：

```
ffplay -f rawvideo -pixel_format yuv420p -s 480*480 texture.yuv
```

其实对于一帧视频帧，或者更直接来说一张PNG或者JPEG的图片，直接用ffplay是可以显示或播放的，当然PNG或者JPEG都会在其头部信息里面指明这张图片的宽高以及格式表示。若想让ffplay显示一张YUV的原始数据表示的图片，那么需要告诉ffplay一些重要的信息，其中包括格式（-f rawvideo代表原始格式）、表示格式（-pixel_format yuv420p）、宽高（-s 480*480）。对于RGB表示的图像，其实是一样的，命令如下：

```
ffplay -f rawvideo -pixel_format rgb24 -s 480*480 texture.rgb
```

上述代码是播放rgb的原始数据，当然还需要指明前面提到的三项基本信息。

另外，对于视频播放器，不得不提的一个问题就是音画同步，在ffplay中音画同步的实现方式其实有三种，分别是：以音频为主时间轴作为同步源；以视频为主时间轴作为同步源；以外部时钟为主时间轴作为同步源。下面就以音频为主时间轴来作为同步源来作为案例进行讲解，这也是后面章节中完成视频播放器项目时要使用到的对齐策略，并且在ffplay中默认的对齐方式也是以音频为基准进行对齐的，那么以音频作为对齐基准是如何实现的呢？

首先要声明的是，播放器接收到的视频帧或者音频帧，内部都会有时间戳（PTS时钟）来标识它实际应该在什么时刻进行展示。实际的对齐策略如下：比较视频当前的播放时间和音频当前的播放时间，如果视频播放过快，则通过加大延迟或者重复播放来降低视频播放速度；如果视频播放慢了，则通过减小延迟或者丢帧来追赶音频播放的时间点。关键就在于音视频时间的比较以及延迟的计算，当然在比较的过程中会设

置一个阈值（Threshold），若超过预设的阈值就应该做调整（丢帧渲染或者重复渲染），这就是整个对齐策略。

对于ffplay可以明确指明使用的到底是哪一种具体的对齐方式，比如：

```
ffplay 32037.mp4 -sync audio
```

上述命令显式地指定了ffplay使用音频为基准进行音视频同步，用来播放文件32037.mp4，当然这也是ffplay的默认设置（就是写与不写都一样）。

```
ffplay 32037.mp4 -sync video
```

上述命令显式地指定了使用以视频为基准进行音视频同步的方式播放视频文件。

```
ffplay 32037.mp4 -sync ext
```

上述命令显式地指定了使用外部时钟作为基准进行音视频同步的方式，用来播放视频文件。

大家可以分别使用这三种方式进行播放，尝试着去听一听，做一些快进操作或者直接跳（seek）到某个位置的操作，观察一下不同的对齐策略对最终的播放具体会造成什么样的影响。

3.ffmpeg

ffmpeg其实是这三个命令行工具里最强大的一个工具，如果说ffprobe是用于探测媒体文件的格式以及详细信息，ffplay是一个播放媒体文件的工具，那么ffmpeg就是强大的媒体文件转换工具。它可以转换任何格式的媒体文件，并且还可以用自己的AudioFilter以及VideoFilter进行处理和编辑，总之一句话，有了它，进行离线处理视频时可以做任何你想做的事情了。下面先介绍总体的参数，然后再列出经典场景下的使用案例。

（1）通用参数

·-f fmt：指定格式（音频或者视频格式）。

·-i filename：指定输入文件名，在Linux下当然也能指定：0.0（屏幕录制）或摄像头。

·-y：覆盖已有文件。

·-t duration：指定时长。

·-fs limit_size：设置文件大小的上限。

·-ss time_off：从指定的时间（单位为秒）开始，也支持[-]hh：mm：ss[.xxx]的格式。

·-re：代表按照帧率发送，尤其在作为推流工具的时候一定要加入该参数，否则ffmpeg会按照最高速率向流媒体服务器不停地发送数据。

·-map：指定输出文件的流映射关系。例如："-map 1：0-map 1：1"要求将第二个输入文件的第一个流和第二个流写入输出文件。如果没有-map选项，则ffmpeg采用默认的映射关系。

（2）视频参数

·-b：指定比特率（bit/s），ffmpeg是自动使用VBR的，若指定了该参数则使用平均比特率。

·-bitexact：使用标准比特率。

·-vb：指定视频比特率（bits/s）。

·-r rate：帧速率（fps）。

·-s size：指定分辨率（320×240）。

·-aspect aspect：设置视频长宽比（4：3，16：9或1.3333，1.7777）。

·-croptop size：设置顶部切除尺寸（in pixels）。

·-cropbottom size：设置底部切除尺寸（in pixels）。

·-cropleft size：设置左切除尺寸（in pixels）。

·-cropright size：设置右切除尺寸（in pixels）。

·-padtop size：设置顶部补齐尺寸（in pixels）。

·-padbottom size：底补齐（in pixels）。

·-padleft size：左补齐（in pixels）。

·-padright size：右补齐（in pixels）。

·-padcolor color：补齐带颜色（000000-FFFFFF）。

·-vn：取消视频的输出。

·-vcodec codec：强制使用codec编解码方式（'copy'代表不进行重新编码）。

（3）音频参数

·-ab：设置比特率（单位为bit/s，老版的单位可能是Kbit/s），对于MP3格式，若要听到较高品质的声音则建议设置为160Kbit/s（单声道则设置为80Kbit/s）以上。

·-aq quality：设置音频质量（指定编码）。

·-ar rate：设置音频采样率（单位为Hz）。

·-ac channels：设置声道数，1就是单声道，2就是立体声。

·-an：取消音频轨。

·-acodec codec：指定音频编码（'copy'代表不做音频转码，直接复制）。

·-vol volume：设置录制音量大小（默认为256）<百分比>。

以上就是日常开发中经常用到的音视频参数以及通用参数，若只介绍这些参数，读者肯定会觉得比较迷茫，下面就结合日常开发中遇到的场景逐个给出具体的实例来实践一下。

1）列出ffmpeg支持的所有格式：

```
ffmpeg -formats
```

2）剪切一段媒体文件，可以是音频或者视频文件：

```
ffmpeg -i input.mp4 -ss 00:00:50.0 -codec copy -t 20 output.mp4
```

表示将文件input.mp4从第50s开始剪切20s的时间，输出到文件output.mp4中，其中-ss指定偏移时间（time Offset），-t指定的时长（duration）。

3）如果在手机中录制了一个时间比较长的视频无法分享到微信中，那么可以使用ffmpeg将该视频文件切割为多个文件：

```
ffmpeg -i input.mp4 -t 00:00:50 -c copy small-1.mp4 -ss 00:00:50 -codec copy
        small-2.mp4
```

4）提取一个视频文件中的音频文件：

```
ffmpeg -i input.mp4 -vn -acodec copy output.m4a
```

5）使一个视频中的音频静音，即只保留视频：

```
ffmpeg -i input.mp4 -an -vcodec copy output.mp4
```

6）从MP4文件中抽取视频流导出为裸H264数据：

```
ffmpeg -i output.mp4 -an -vcodec copy -bsf:v h264_mp4toannexb output.h264
```

注意，上述指令里不使用音频数据（-an），视频数据使用mp4toannexb这个bitstream filter来转换为原始的H264数据，在后续的API章节中也会频繁使用到该bitstream filter，在前面的章节中也曾提到过同一编码会有不同的封装格式。

7）使用AAC音频数据和H264的视频生成MP4文件：

```
ffmpeg -i test.aac -i test.h264 -acodec copy -bsf:a aac_adtstoasc -vcodec cop
        mp4 output.mp4
```

上述代码中使用了一个名为aac_adtstoasc的bitstream filter，AAC格式也有两种封装格式，前面的章节中也曾提到过，而且在后续的章节中也会继续使用API调用该bitstream filter。

8）对音频文件的编码格式做转换：

```
ffmpeg -i input.wav -acodec libfdk_aac output.aac
```

9）从WAV音频文件中导出PCM裸数据：

```
ffmpeg -i input.wav -acodec pcm_s16le -f s16le output.pcm
```

这样就可以导出用16个bit来表示一个sample的PCM数据了，并且每个sample的字节排列顺序都是小尾端表示的格式，声道数和采样率使用的都是原始WAV文件的声道数和采样率的PCM数据。

10）重新编码视频文件，复制音频流，同时封装到MP4格式的文件中：

```
ffmpeg -i input.flv -vcodec libx264 -acodec copy output.mp4
```

11）将一个MP4格式的视频转换成为gif格式的动图：

```
ffmpeg -i input.mp4 -vf scale=100:-1 -t 5 -r 10 image.gif
```

上述代码按照分辨比例不动宽度改为100（使用VideoFilter的scaleFilter），帧率改为10（-r），只处理前5秒钟（-t）的视频，生成gif。

12）将一个视频的画面部分生成图片，比如要分析一个视频里面的每一帧都是什么内容的时候，可能就需要用到这个命令了：

```
ffmpeg -i output.mp4 -r 0.25 frames_%04d.png
```

上述命令每4秒钟截取一帧视频画面生成一张图片，生成的图片从frames_0001.png开始一直递增下去。

13）使用一组图片可以组成一个gif，如果你连拍了一组照片，就可以用下面这行命令生成一个gif：

```
ffmpeg -i frames_%04d.png -r 5 output.gif
```

14）使用音量效果器，可以改变一个音频媒体文件中的音量：

```
ffmpeg -i input.wav -af ‘volume=0.5’ output.wav
```

上述命令是将input.wav中的声音减小一半，输出到output.wav文件中，可以直接播放来听，或者放到一些音频编辑软件中直接观看波形幅度的效果。

15）淡入效果器的使用：

```
ffmpeg -i input.wav -filter_complex afade=t=in:ss=0:d=5 output.wav
```

上述命令可以将input.wav文件中的前5s做一个淡入效果，输出到output.wav中，可以将处理之前和处理之后的文件拖到Audacity音频编辑软件中查看波形图。

16）淡出效果器的使用：

```
ffmpeg -i input.wav -filter_complex afade=t=out:st=200:d=5 output.wav
```

上述命令可以将input.wav文件从200s开始，做5s的淡出效果，并放到output.wav文件中。

17）将两路声音进行合并，比如要给一段声音加上背景音乐：

```
ffmpeg -i vocal.wav -i accompany.wav -filter_complex
        amix=inputs=2:duration=shortest output.wav
```

上述命令是将vocal.wav和accompany.wav两个文件进行mix，按照时间长度较短的音频文件的时间长度作为最终输出的output.wav的时间长度。

18）对声音进行变速但不变调效果器的使用：

```
ffmpeg -i vocal.wav -filter_complex atempo=0.5 output.wav
```

上述命令是将vocal.wav按照0.5倍的速度进行处理生成output.wav，时间长度将会变为输入的2倍。但是音高是不变的，这就是大家常说的变速不变调。

19）为视频增加水印效果：

```
ffmpeg -i input.mp4 -i changba_icon.png -filter_complex
        '[0:v][1:v]overlay=main_w-overlay_w-10:10:1[out]' -map '[out]' output
```

上述命令包含了几个内置参数，main_w代表主视频宽度，overlay_w代表水印宽度，main_h代表主视频高度，overlay_h代表水印

高度。

20）视频提亮效果器的使用：

```
ffmpeg -i input.flv -c:v libx264 -b:v 800k  -c:a libfdk_aac -vf eq=brightness
        -f mp4 output.mp4
```

提亮参数是brightness，取值范围是从-1.0到1.0，默认值是0。

21）为视频增加对比度效果：

```
ffmpeg -i input.flv -c:v libx264 -b:v 800k  -c:a libfdk_aac -vf eq=contrast=1
        mp4 output.mp4
```

对比度参数是contrast，取值范围是从-2.0到2.0，默认值是1.0。

22）视频旋转效果器的使用：

```
ffmpeg -i input.mp4 -vf "transpose=1" -b:v 600k output.mp4
```

23）视频裁剪效果器的使用：

```
ffmpeg -i input.mp4 -an -vf "crop=240:480:120:0" -vcodec libx264 -b:v 600k ou
```

24）将一张RGBA格式表示的数据转换为JPEG格式的图片：

```
ffmpeg -f rawvideo -pix_fmt rgba -s 480*480 -i texture.rgb -f image2 -vcodec
        output.jpg
```

25）将一个YUV格式表示的数据转换为JPEG格式的图片：

```
ffmpeg -f rawvideo -pix_fmt yuv420p -s 480*480 -i texture.yuv -f image2 -vcod
        output.jpg
```

26）将一段视频推送到流媒体服务器上：

```
ffmpeg -re -i input.mp4 -acodec copy -vcodec copy -f flv rtmp://xxx
```

上述代码中，rtmp：//xxx代表流媒体服务器的地址，加上-re参数代表将实际媒体文件的播放速度作为推流速度进行推送。

27）将流媒体服务器上的流dump到本地：

```
ffmpeg -i http://xxx/xxx.flv -acodec copy -vcodec copy -f flv output.flv
```

上述代码中，http://xxx/xxx.flv 代表一个可以访问的视频网络地址，可按照复制视频流格式和音频流格式的方式，将文件下载到本地的output.flv媒体文件中。

28）将两个音频文件以两路流的形式封装到一个文件中，比如在K歌的应用场景中，原伴唱实时切换的场景下，可以使用一个文件包含两路流，一路是伴奏流，另外一路是原唱流：

```
ffmpeg -i 131.mp3 -i 134.mp3 -map 0:a -c:a:0 libfdk_aac -b:a:0 96k -map 1:a -
        libfdk_aac -b:a:1 64k -vn -f mp4 output.m4a
```

其实，FFmpeg的命令工具随意组合的话会有很多种，这里只列举了一部分，如果大家弄清楚了FFmpeg能做什么，那么具体的使用就一目了然了，只要是FFmpeg框架能够实现的功能，那么FFmpeg命令行工具就能将其提供出来了。下面将会介绍如何从API层面调用FFmpeg来完成日常的开发工作。

3.2 FFmpeg API的介绍与使用

首先，需要统一下术语，具体如下。

·容器 / 文件（Conainer/File）：即特定格式的多媒体文件，比如MP4、flv、mov等。

·媒体流（Stream）：表示时间轴上的一段连续数据，如一段声音数据、一段视频数据或一段字幕数据，可以是压缩的，也可以是非压缩的，压缩的数据需要关联特定的编解码器。

·数据帧 / 数据包（Frame/Packet）：通常，一个媒体流是由大量的数据帧组成的，对于压缩数据，帧对应着编解码器的最小处理单元，分属于不同媒体流的数据帧交错存储于容器之中。

·编解码器：编解码器是以帧为单位实现压缩数据和原始数据之间的相互转换的。

前面已经介绍过FFmpeg的各个库以及每个库所承担的职责，本节将要讨论的是，以写代码的方式调用FFmpeg的API完成一些媒体文件的操作，首先会介绍最重要的结构体，然后再以实例的方式完成两个实践。

前面所介绍的术语，其实就是FFmpeg中抽象出来的概念，我们不得不承认，FFmpeg的功能非常强大，同时我们也得承认FFmpeg的代码设计也是非常优秀的。其中AVFormatContext就是对容器或者说媒体文件层次的一个抽象，该文件中（或者说在这个容器里面）包含了多路流（音频流、视频流、字幕流等），对流的抽象就是AVStream；在每一路流中都会描述这路流的编码格式，对编解码格式以及编解码器的抽象就是AVCodecContext与AVCodec；对于编码器或者解码器的输入输出部分，也就是压缩数据以及原始数据的抽象就是AVPacket与AVFrame。

前面是从多媒体概念的角度上自外向内地对FFmpeg进行了剖析，这也正好是FFmpeg抽象的层次，非常易于理解。当然除了编解码之外，对于音视频的处理肯定是针对于原始数据的处理，也就是针对于AVFrame的处理，使用的就是AVFilter，这一点也非常好理解。

至此，FFmpeg中最重要的几个模块都已经介绍完毕了，下面来具体看一个解码的实例，该实例实现的功能非常单一，就是把一个视频文件解码成为单独的音频PCM文件和视频YUV文件，最后将会使用前面章节中介绍的ffplay去验证播放这两个文件，以查看是否可以得到正确的结果。

首先，要使用FFmpeg就必须要引用它的头文件，以及在链接阶段使用它的静态库文件，关于头文件和静态库文件的生成，前面已经介绍过了，这里直接将文件（include文件夹与libffmpeg.a静态库文件）拿过来使用。

1.引用头文件

如果是在iOS下，那么可直接以下面这种方式引用头文件：

```
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libswresample/swresample.h"
#include "libavutil/pixdesc.h"
```

如果是在Android的C++环境下，那么可直接以下面这种方式引用头文件：

```
extern "C" {
    #include "3rdparty/ffmpeg/include/libavformat/avformat.h"
    #include "3rdparty/ffmpeg/include/libswscale/swscale.h"
    #include "3rdparty/ffmpeg/include/libswresample/swresample.h"
    #include "3rdparty/ffmpeg/include/libavutil/pixdesc.h"
}
```

extern“C”的解释

作为一种面向对象的语言，C++支持函数的重载，而面向过程的C语言是不支持函数重载的。同一个函数在C++中编译后与其在C中编译后，在符号表中的签名是不同的，假如对于同一个函数：

```
void decode(float position, float duration)
```

在C语言中编译出来的签名是_decoder，而在C++语言中，一般编译器的生成则类似于_decode_float_float。虽然在编译阶段是没有问题的，但是在链接阶段，如果不加extern“C”关键字的话，那么将会链接_decoder_float_float这个方法签名；而如果加了extern“C”关键字的话，那么寻找的方法签名就是_decoder。而FFmpeg就是C语言书写的，编译FFmpeg的时候所产生的方法签名都是C语言类型的签名，所以在C++中引用FFmpeg必须要加extern“C”关键字。

可以看到，引用头文件的方式是不同的，因为每个平台配置的Header Search Path是不一样的，在iOS的IDE Xcode开发中，可以在工程文件的配置中修改Header Search Path；在Android的底层开发中，可以配置makefile文件中的内置变量LOCAL_C_INCLUDES来指定头文件的搜索路径，当然如果要在跨平台（Android平台和iOS平台）的模块（以C++语言编写）中引用FFmpeg的头文件，则需要编写一个platform_4_ffmpeg.h，并在其中根据各个平台预定义的宏去编译不同的引用方式，代码如下：

```
#ifdef __ANDROID__
    extern "C" {
        #include "3rdparty/ffmpeg/include/libavformat/avformat.h"
        #include "3rdparty/ffmpeg/include/libswscale/swscale.h"
        #include "3rdparty/ffmpeg/include/libswresample/swresample.h"
        #include "3rdparty/ffmpeg/include/libavutil/pixdesc.h"
    }
#elif defined(__APPLE__)            // iOS或OS X
    extern "C" {
        #include "libavformat/avformat.h"
        #include "libswscale/swscale.h"
        #include "libswresample/swresample.h"
        #include "libavutil/pixdesc.h"
    }
#endif
```

2.注册协议、格式与编解码器

使用FFmpeg的API，首先要调用FFmpeg的注册协议、格式与编解码器的方法，确保所有的格式与编解码器都被注册到了FFmpeg框架中，当然如果需要用到网络的操作，那么也应该将网络协议部分注册到FFmpeg框架，以便于后续再去查找对应的格式。代码如下：

```
avformat_network_init();
```

```
av_register_all();
```

文档中还有一个方法是avcodec_register_all（），其用于将所有编解码器注册到FFmpeg框架中，但是av_register_all方法内部已经调用了avcodec_register_all方法，所以其实只需要调用av_register_all就可以了。

3.打开媒体文件源，并设置超时回调

注册了格式以及编解码器之后，接下来就应该打开对应的媒体文件了，当然该文件既可能是本地磁盘的文件，也可能是网络媒体资源的一个链接，如果是网络链接，则会涉及不同的协议，比如RTMP、HTTP等协议的视频源。打开媒体资源以及设置超时回调的代码如下：

```
AVFormatContext *formatCtx = avformat_alloc_context();
AVIOInterruptCB int_cb = {interrupt_callback, (__bridge void *)(self)};
formatCtx->interrupt_callback = int_cb;
avformat_open_input(formatCtx, path, NULL, NULL);
avformat_find_stream_info(formatCtx, NULL);
```

4.寻找各个流，并且打开对应的解码器

上一步中已打开了媒体文件，相当于打开了一根电线，这根电线里面其实还有一条红色的线和一条蓝色的线，这就和媒体文件中的流非常类似了，红色的线代表音频流，蓝色的线代表视频流。所以这一步我们就要寻找出各个流，然后找到流中对应的解码器，并且打开它。

寻找音视频流：

```
for(int i = 0; i < formatCtx->nb_streams; i++) {
    AVStream* stream = formatCtx->streams[i];
    if(AVMEDIA_TYPE_VIDEO == stream->codec->codec_type) {
        // 视频流
        videoStreamIndex = i;
    } else if(AVMEDIA_TYPE_AUDIO == stream->codec->codec_type ){
        // 音频流
        audioStreamIndex = i;
    }
}
```

打开音频流解码器：

```
AVCodecContext * audioCodecCtx = audioStream->codec;
AVCodec *codec = avcodec_find_decoder(audioCodecCtx ->codec_id);
if(!codec){
    // 找不到对应的音频解码器
}
int openCodecErrCode = 0;
if ((openCodecErrCode = avcodec_open2(codecCtx, codec, NULL)) < 0){
    // 打开音频解码器失败
}
```

打开视频流解码器：

```
AVCodecContext *videoCodecCtx = videoStream->codec;
AVCodec *codec = avcodec_find_decoder(videoCodecCtx->codec_id);
if(!codec) {
    // 找不到对应的视频解码器
}
int openCodecErrCode = 0;
if ((openCodecErrCode = avcodec_open2(codecCtx, codec, NULL)) < 0) {
    // 打开视频解码器失败
}
```

5.初始化解码后数据的结构体

知道了音视频解码器的信息之后，下面需要分配出解码之后的数据所存放的内存空间，以及进行格式转换需要用到的对象。

构建音频的格式转换对象以及音频解码后数据存放的对象：

```
SwrContext *swrContext = NULL;
if(audioCodecCtx->sample_fmt ! = AV_SAMPLE_FMT_S16) {
    // 如果不是我们需要的数据格式
    swrContext = swr_alloc_set_opts(NULL,
            outputChannel, AV_SAMPLE_FMT_S16, outSampleRate,
            in_ch_layout, in_sample_fmt, in_sample_rate, 0, NULL);
    if(!swrContext || swr_init(swrContext)) {
        if(swrContext) {
            swr_free(&swrContext);
        }
    }
    audioFrame = avcodec_alloc_frame();
}
```

构建视频的格式转换对象以及视频解码后数据存放的对象：

```
AVPicture picture;
bool pictureValid = avpicture_alloc(&picture,
        PIX_FMT_YUV420P,
        videoCodecCtx->width,
         videoCodecCtx->height) == 0;
if (!pictureValid){
    // 分配失败
    return false;
}
swsContext = sws_getCachedContext(swsContext,
        videoCodecCtx->width,
        videoCodecCtx->height,
        videoCodecCtx->pix_fmt,
        videoCodecCtx->width,
        videoCodecCtx->height,
        PIX_FMT_YUV420P,
        SWS_FAST_BILINEAR,
         NULL, NULL, NULL);
videoFrame = avcodec_alloc_frame();
```

6.读取流内容并且解码

打开了解码器之后，就可以读取一部分流中的数据（压缩数据），然后将压缩数据作为解码器的输入，解码器将其解码为原始数据（裸数据），之后就可以将原始数据写入文件了：

```
AVPacket packet;
int gotFrame = 0;
while(true) {
    if(av_read_frame(formatContext, &packet)) {
        // End Of File
        break;
    }
    int packetStreamIndex = packet.stream_index;
    if(packetStreamIndex == videoStreamIndex) {
        int len = avcodec_decode_video2(videoCodecCtx, videoFrame,
                &gotFrame, &packet);
        if(len < 0) {
            break;
        }
        if(gotFrame) {
            self->handleVideoFrame();
        }
    } else if(packetStreamIndex == audioStreamIndex) {
        int len = avcodec_decode_audio4(audioCodecCtx, audioFrame,
                &gotFrame, &packet);
        if(len < 0) {
            break;
```

```
        }
        if(gotFrame) {
            self->handleVideoFrame();
        }
    }
}
```

7.处理解码后的裸数据

解码之后会得到裸数据，音频就是PCM数据，视频就是YUV数据。下面将其处理成我们所需要的格式并且进行写文件。

音频裸数据的处理：

```
void* audioData;
int numFrames;
    if(swrContext) {
        int bufSize = av_samples_get_buffer_size(NULL, channels,
                (int)(audioFrame->nb_samples * channels),
                AV_SAMPLE_FMT_S16, 1);
        if (!_swrBuffer || _swrBufferSize < bufSize) {
            swrBufferSize = bufSize;
            swrBuffer = realloc(_swrBuffer, _swrBufferSize);
        }
        Byte *outbuf[2] = { _swrBuffer, 0 };
        numFrames = swr_convert(_swrContext, outbuf,
                (int)(audioFrame->nb_samples * channels),
                (const uint8_t **)_audioFrame->data,
                audioFrame->nb_samples);
    audioData = swrBuffer;
} else {
        audioData = audioFrame->data[0];
        numFrames = audioFrame->nb_samples;
}
```

接收到音频裸数据之后，就可以直接写文件了，比如写到文件audio.pcm中。

视频裸数据的处理：

```
uint8_t* luma;
uint8_t* chromaB;
uint8_t* chromaR;
if(videoCodecCtx->pix_fmt == AV_PIX_FMT_YUV420P ||
        videoCodecCtx->pix_fmt == AV_PIX_FMT_YUVJ420P){
    luma = copyFrameData(videoFrame->data[0],
```

```
                videoFrame->linesize[0],
                videoCodecCtx->width,
                videoCodecCtx->height);
        chromaB = copyFrameData(videoFrame->data[1],
                videoFrame->linesize[1],
                videoCodecCtx->width / 2,
                videoCodecCtx->height / 2);
        chromaR = copyFrameData(videoFrame->data[2],
                videoFrame->linesize[2],
                videoCodecCtx->width / 2,
                videoCodecCtx->height / 2);
    } else{
        sws_scale(_swsContext,
                 (const uint8_t **)videoFrame->data,
                videoFrame->linesize,
                0,
                videoCodecCtx->height,
                picture.data,
                picture.linesize);
        luma = copyFrameData(picture.data[0],
                picture.linesize[0],
                videoCodecCtx->width,
                videoCodecCtx->height);
        chromaB = copyFrameData(picture.data[1],
                picture.linesize[1],
                videoCodecCtx->width / 2,
                videoCodecCtx->height / 2);
        chromaR = copyFrameData(picture.data[2],
                picture.linesize[2],
                videoCodecCtx->width / 2,
                videoCodecCtx->height / 2);
    }
```

接收到YUV数据之后也可以直接写入文件了，比如写到文件video.yuv中。

8.关闭所有资源

解码完毕之后，或者在解码过程中不想继续解码了，可以退出程序，当然，退出的时候，要将用到的FFmpeg框架中的资源，包括FFmpeg框架对外的连接资源等全都释放掉。

关闭音频资源：

```
    if (swrBuffer) {
        free(swrBuffer);
        swrBuffer = NULL;
        swrBufferSize = 0;
    }
```

```
if (swrContext) {
    swr_free(&swrContext);
    swrContext = NULL;
}
if (audioFrame) {
    av_free(audioFrame);
    audioFrame = NULL;
}
if (audioCodecCtx) {
    avcodec_close(audioCodecCtx);
    audioCodecCtx = NULL;
}
```

关闭视频资源：

```
if (swsContext) {
    sws_freeContext(swsContext);
    swsContext = NULL;
}
if (pictureValid) {
    avpicture_free(&picture);
    pictureValid = false;
}
if (videoFrame) {
    av_free(videoFrame);
    videoFrame = NULL;
}
if (videoCodecCtx) {
    avcodec_close(videoCodecCtx);
    videoCodecCtx = NULL;
}
```

关闭连接资源：

```
if (formatCtx) {
    avformat_close_input(&formatCtx);
    formatCtx = NULL;
}
```

以上就是利用FFmpeg解码的全部过程了，其中包括打开文件流、解析格式、解析流并且打开解码器、解码和处理，以及最终关闭所有资源的操作。项目实例在代码仓库中是FFmpegDecoder项目。关于FFmpeg的API调用其实也就是这些步骤，只要多使用几次就可以熟练掌握，但是具体到FFmpeg在某一步到底都做了些什么操作呢？3.3节将会带领我们揭秘FFmpeg内部是如何实现这些处理的。

3.3 FFmpeg源码结构

3.2节中已经详细介绍了FFmpeg每个模块的作用，并且通过图3-1基本了解了FFmpeg的内部结构，本节就来详细地看一下FFmpeg具体每个模块里面是如何实现自己的职责的。

3.3.1 libavformat与libavcodec介绍

首先，来看最重要的模块之一的libavformat，其主要组成与层次调用关系如图3-2所示。

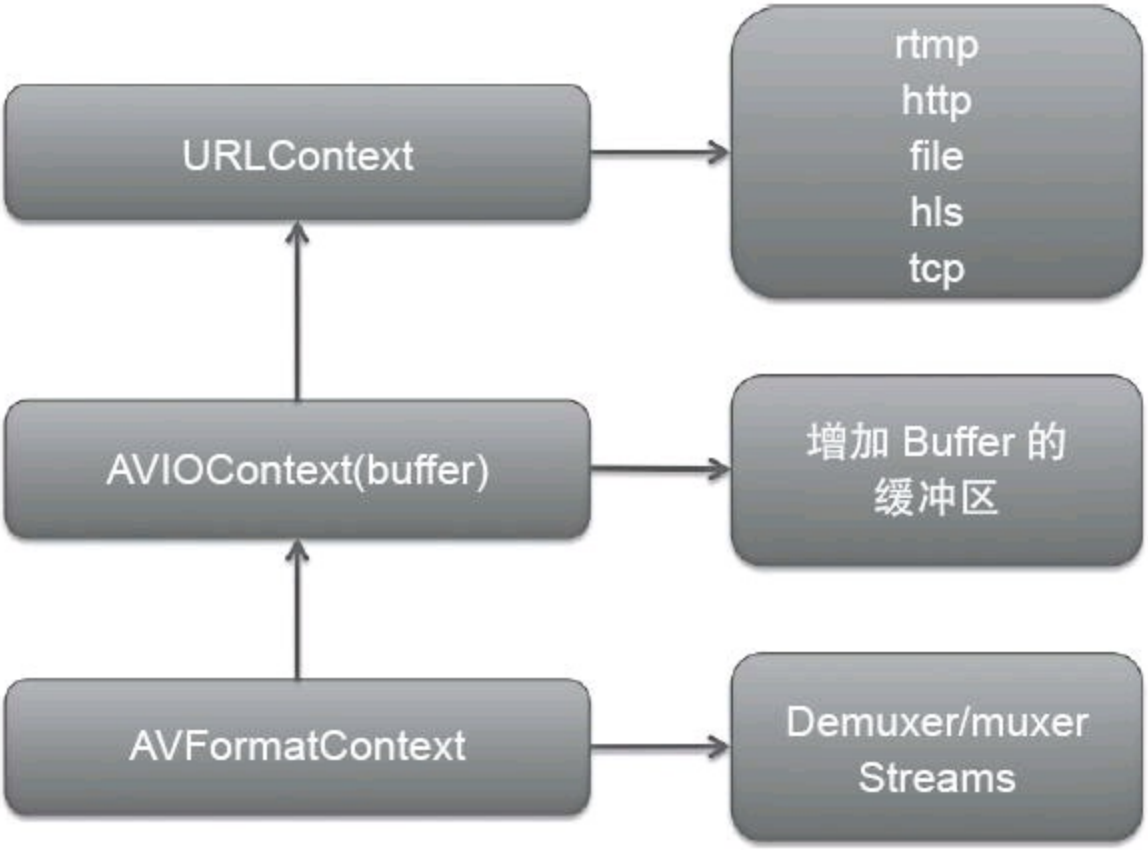


图 3-2

AVFormatContext是API层直接接触到的结构体，它会进行格式的封装与解封装，它的数据部分由底层提供，底层使用了AVIOContext，这个AVIOContext实际上就是为普通的I/O增加了一层Buffer缓冲区，再往底层就是URLContext，也就是到达了协议层，协议层的具体实现有很多，包括rtmp、http、hls、file等，这就是libavformat的内部封装了。

其次，再来看另外一个最重要的模块libavcodec，其主要组成与数据结构如图3-3所示。

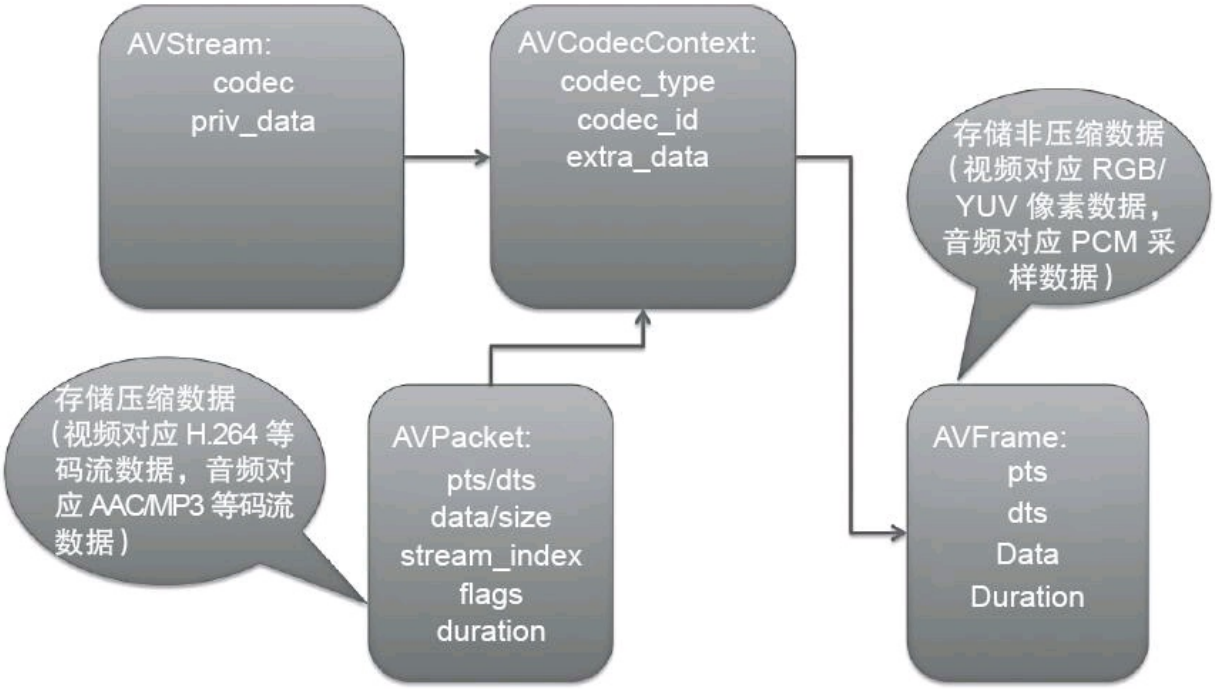


图　3-3

对于开发者来说，这一层我们能接触到的最顶层的结构体就是AVCodecContext，该结构体包含的就是与实际的编解码有关的部分。首先，AVCodecContext是包含在一个AVStream里面的，即描述了这路流的编码格式是什么，其中存放了具体的编码格式信息，根据Codec的信息可以打开编码器或者解码器，然后利用该编码器或者解码器进行AVPacket与AVFrame之间的转换（实际上就是解码或者编码的过程），这是FFmpeg中最重要的一部分。那么，接下来就来看一下在API中调用了FFmpeg的一些方法之后，FFmpeg内部到底做了些什么呢？

3.3.2 FFmpeg通用API分析

1.av_register_all分析

还记得前面最开始编译FFmpeg的时候，做了一个configure的配置吗？其中开启（enable）或者关闭（disable）了很多选项，当初可是留了一句话，configure的配置会生成两个文件：config.mk与config.h。config.mk实际上就是makefile文件需要包含进去的子模块，会作用在编译阶段，帮助开发者编译出正确的库；而config.h是作用在运行阶段，这一阶段将确定需要注册哪些容器以及编解码格式到FFmpeg框架中。所以该函数的内部实现会先调用avcodec_register_all来注册所有config.h里面开放的编解码器，然后会注册所有的Muxer和Demuxer（也就是封装格式），最后注册所有的Protocol（即协议层的东西）。这样一来，在configure过程中开启（enable）或者关闭（disable）的选项就就作用到了运行时，该函数的源码分析涉及的源码文件包括：url.c、allformats.c、mux.c、format.c等文件。

2.av_find_codec分析

这里面其实包含了两部分的内容：一部分是寻找解码器，一部分是寻找编码器。其实在第一步的avcodec_register_all函数里面已经把编码器和解码器都存放到一个链表中了，在这里寻找编码器或者解码器都是从第一步构造的链表中进行遍历，通过Codec的ID或者name进行条件匹配，最终返回对应的Codec。

3.avcodec_open2分析

该函数是打开编解码器（Codec）的函数，无论是编码过程还是解码过程，都会用到该函数，该函数的输入参数有三个：第一个是AVCodecContext，解码过程由FFmpeg引擎填充，编码过程由开发者自己构造，如果想要传入私有参数，则为它的priv_data设置参数，比如在libx264编码器中设置preset、tune、profile等；第二个参数是上一步通过av_find_codec寻找出来的编解码器（Codec）；第三个参数一般会传递NULL。具体到该函数的实现时，就会找到对应的实现文件，那么其是如何找到对应的实现文件的呢？这就需要回到第一步中来看看其是如何注册的，比如libx264的编码器，查看其注册会发现ff_libx264_encoder结

构体的定义存在于libx264.c中，所以该Codec的生命周期方法就会委托给该结构体对应的函数指针所指向的函数，open对应的就是init函数指针所指向的函数，该函数里面就会调用具体的编码库的API，比如libx264这个Codec会调用libx264的编码库的API，而LAME这个Codec会调用LAME的编码库的API，并且会以对应的AVCodecContext中的priv_data来填充对应第三方库所需要的私有参数，如果开发者没有对属性priv_data填充值，那么就使用默认值。

4.avcodec_close分析

如果理解了avcodec_open，那么对应的close就是一个逆过程，找到对应的实现文件中的close函数指针所指向的函数，然后该函数会调用对应第三方库的API来关闭掉对应的编码库。其实FFmpeg所做的事情就是透明化所有的编解码库，用自己的封装来为开发者提供统一的接口。开发者使用不同的编码库时，只需要指明要使用哪一个即可，这也充分体现了面向对象编程中的封装特性，关于FFmpeg面向对象的特性后续还会进一步讨论。

3.3.3 调用FFmpeg解码时用到的函数分析

1.avformat_open_input分析

函数avformat_open_input会根据所提供的文件路径判断文件的格式，其实就是通过这一步来决定使用的到底是哪一个Demuxer。举例来说，如果是flv，那么Demuxer就会使用对应的ff_flv_demuxer，所以对应的关键生命周期的方法read_header、read_packet、read_seek、read_close都会使用该flv的Demuxer中函数指针指定的函数。read_header函数会将AVStream结构体构造好，以便后续的步骤继续使用AVStream作为输入参数。

2.avformat_find_stream_info分析

这个函数非常重要，后续章节中将要介绍的如何在直播场景下的拉流客户端中“秒开首屏”，就是与该函数分析的代码实现息息相关的，该方法的作用就是把所有Stream的MetaData信息填充好。方法内部会先查找对应的解码器，然后打开对应的解码器，紧接着会利用Demuxer中的read_packet函数读取一段数据进行解码，当然解码的数据越多，分析出的流信息就会越准确，如果是本地资源，那么很快就可以得到非常准确的信息了，但是对于网络资源来说，则会比较慢，因此该函数有几个参数可以控制读取数据的长度，一个是probe size，一个是max_analyze_duration，还有一个是fps_probe_size，这三个参数共同控制解码数据的长度，当然，如果配置这几个参数的值越小，那么这个函数执行的时间就会越快，但是会导致AVStream结构体里面一些信息（视频的宽、高、fps、编码类型等）不准确。

3.av_read_frame分析

使用该方法读取出来的数据是AVPacket，在FFmpeg的早期版本中开放给开发者的函数其实就是av_read_packet，但是需要开发者自己来处理AVPacket中的数据不能被解码器完全处理完的情况，即需要把未处理完的压缩数据缓存起来的问题。所以到了新版本的FFmpeg中，其提供了该函数，用于处理此状况。该函数的实现首先会委托到Demuxer的read_packet方法中去，当然read_packet通过解复用层和协议层的处理之后，会将数据返回到这里，在该函数中进行数据缓冲处理。前面曾说

过，对于音频流，一个AVPacket可能包含多个AVFrame，但是对于视频流，一个AVPacket只包含一个AVFrame，该函数最终只会返回一个AVPacket结构体。

4.avcodec_decode分析

该方法包含了两部分内容：一部分是解码视频，一部分是解码音频。在上面的函数分析中，我们知道，解码是会委托给对应的解码器来实施的，在打开解码器的时候就找到了对应解码器的实现，比如对于解码H264来讲，会找到ff_h264_decoder，其中会有对应的生命周期函数的实现，最重要的就是init、decode、close这三个方法，分别对应于打开解码器、解码以及关闭解码器的操作，而解码过程就是调用decode方法。

5.avformat_close_input分析

该函数负责释放对应的资源，首先会调用对应的Demuxer中的生命周期read_close方法，然后释放掉AVFormatContext，最后关闭文件或者远程网络连接。

3.3.4 调用FFmpeg编码时用到的函数分析

1.avformat_alloc_output_context2分析

该函数内部需要调用方法avformat_alloc_context来分配一个AVFormatContext结构体，当然最关键的还是根据上一步注册的Muxer和Demuxer部分（也就是封装格式部分）去找到对应的格式。有可能是flv格式、MP4格式、mov格式，甚至是MP3格式等，如果找不到对应的格式（即在configure选项中没有打开这个格式的开关），那么这里会返回找不到对应的格式的错误提示。在调用API的时候，可以使用av_err2str把返回整数类型的错误代码转换为肉眼可读的字符串，这在调试的时候是一个比较有用的工具函数。该函数最终会将找出来的格式赋值给AVFormatContext类型的oformat。

2.avio_open2分析

首先调用函数ffurl_open，构造出URLContext结构体，这个结构体中包含了URLProtocol（需要去第一步register_protocol中已经注册的协议链表中寻找）；接着会调用avio_alloc_context方法，分配出AVIOContext结构体，并将上一步构造出来的URLProtocol传递进来；然后把上一步分配出来AVIOContext结构体赋值给AVFormatContext的属性，其实这就是图3-2表示的结构了。而该过程恰好是上面所分析的avformat_open_input函数的实现过程的一个逆过程。之前就提到过，编码过程和解码过程从逻辑上来讲本来就是一个逆过程，所以在FFmpeg的实现过程中它们也是一个逆过程。

后面的步骤也都是解码的一个逆过程，解码过程中的av_find_stream_info对应到这里就是avformat_new_stream和avformat_write_header。avformat_new_stream函数会将音频流或者视频流的信息填充好，分配出AVStream结构体，在音频流中分配声道、采样率、表示格式、编码器等信息，在视频流中分配宽、高、帧率、表示格式、编码器等信息；avformat_write_header函数与解码过程中的read_header恰好是一个逆过程，因此这里将不再介绍。接下来就是编码的阶段了，开发者需要将手动封装好的AVFrame结构体，作为avcodec_encode_video方法的输入，将其编码成为AVPacket，然后调用av_write_frame方法输出到媒体文件中。而av_write_frame方法会将编码

后的AVPacket结构体作为Muxer中的write_packet生命周期方法的输入，write_packet函数会加上自己封装格式的头信息，然后调用协议层写到本地文件或者网络服务器上。最后一步就是av_write_trailer，该函数有一个非常大的坑，如果没有执行write_header操作，就直接执行write_trailer操作，程序会直接崩溃（即Crash掉），所以必须保证这两个函数成对出现。write_trailer函数的实现会把没有输出的AVPacket全部丢给协议层去做输出，然后会调用Muxer的write_trailer生命周期方法，对于不同的格式写出的尾部也不尽相同，这里不再逐一介绍。

对于FFmpeg的使用以及源码分析，CSDN上有一个名叫雷霄骅的博主对此分析得非常细致，非常不幸的是，雷霄骅现在已经去世了，笔者曾经和雷霄骅交谈过，对于雷霄骅的分享精神以及他对刚进入视频领域新手的耐心指导都非常钦佩，其实在国内很多使用FFmpeg的人或多或少都听说过雷霄骅这个名字，他无私地帮助了很多人，作为一个技术人员，我为他感到骄傲。

3.3.5 面向对象的C语言设计

笔者重读了FFmpeg的源码之后写下了上面的分析内容，但是这里想要和大家聊一个更高层次的内容，即面向对象与面向结构。有人说C语言是面向结构的语言，写出来的代码虽然性能很高，但是可读性与维护性很差，没有面向对象程序设计的思维，很难做出大型项目。既然已经分析了FFmpeg的源码，那么我们可以想一想FFmpeg这么强大的开源媒体处理库，它的稳定性与迭代速度在业界中其实是非常不错的，并且其扩展性与代码可读性也是相当好的，所以扩展性与可读性真的与语言或者编程思想有关系吗？既然已经读到了这里，那就说明大家对FFmpeg已经有了整体的了解，下面再回到最开始来看看FFmpeg是如何被编译以及使用的。

在编译的时候可以进行配置，既可以纵向裁剪模块（是否还记得configure里面的disable-avdevice、disable-muxers、disable-decoders），又可以横向裁剪支持的格式、编解码器（enable-muxer=flv、enable-decoder=aac、enable-encoder=libfdk_aac）。该框架支持多种格式、多种编解码器、多种音视频滤镜处理，关键是如果后期增加格式、编解码器、音视频滤镜只需要新增代码与修改配置文件（configure），而不需要改动已有框架的代码层面。FFmpeg里面利用C语言结构体的定义（如encoder的init、open、decode、encode、close等方法）与语言特性，实现了面向接口编程，而且完全符合开闭原则——对新增代码开放，对修改代码关闭，这也是FFmpeg能做到代码的可读性与扩展性的精要部分。其实写代码（Coding）提升的过程也是我们对事物认识提升的过程，只要跳出FFmpeg框架的具体实现，再来看它的实现思想，就可以得到更多的收获。

下面就以一个短小的篇幅来分析一下libx264编码器是如何被增加到FFmpeg框架中的。首先新增libx264.c这个文件，在文件中定义结构体：

```
AVCodec libx264_encoder = {
    .name = ”libx264”
    .type = CODEC_TYPE_VIDEO
    .id=CODEC_ID_H264
    .priv_data_size=sizeof(X264Context)
    .init=X264_init
    .encode=X264_frame
    .close=X264_close
```

```
}
```

通过结构体可以看到对于编码器几个生命周期方法的定义，其实，新增的libx264.c实际上就是FFmpeg对于第三方libx264库的一个封装，提供给用户的API都是固定的FFmpeg的API，而libx264.c文件就相当于是第三方库libx264的客户端代码。在使用libx264编码的时候，开发者需要编译静态库x264，然后作为External-libs的形式加入到FFmpeg框架中。

当我们在外界调用avcodec_find_encoder的时候，就会去寻找这个结构体；当调用avcodec_open的时候就会调用到结构体的init方法，进而会调用到新增的这个文件中的X264_init方法，而在X264_init方法中，则会调用真正的libx264库里面的API；当调用avcodec_encode的时候就会调用结构体的encode方法，进而会调用X264_frame方法，而在X264_frame方法中则会真正地使用libx264的API；类似地，当调用avcodec_close的时候，就会调用X264_close这个方法；当然对于一些私有的配置，将统一放到AVCodecContext的priv_data里面去，然后在初始化X264库的时候取出来，并对X264库中的API进行设置。

3.4 本章小结

至此，对于FFmpeg的介绍就结束了，当然FFmpeg库的强大是众所周知的，显然用一章的篇幅要想将其讲得特别透彻几乎是不可能的，本章用了大量篇幅来讲解如何在多个平台上编译、安装、使用命令行，以及如何以API的方式使用FFmpeg，并且对FFmpeg的源码也做了分析，最后进行了一个升华，分析了FFmpeg是如何使用面向结构的C语言来满足开闭原则的，并使得整个FFmpeg代码的可读性与扩展性都能达到一个非常高的级别。

后面的章节将会用到本章的内容，请大家重点掌握FFmpeg在代码层面API方式的使用，当然命令行的使用可以作为工具来了解，用多了就都能记住了。本章的项目实例是代码目录的FFmpegDecoder项目，在Android平台下，需要把resource目录下面的131.mp3复制到sdcard的根目录下。无论是在Android还是在iOS平台下，最终输出的目标文件都可以用ffplay指定声道、采样率以及表示格式，以进行播放，如果可以正常播放则代表我们的解码是成功的。

第4章　移动平台下的音视频渲染

前面的章节基本上都在介绍音视频的概念与工具的使用，如果再不动手敲一些代码，想必工程师读者都会有点手痒了吧。那就动手实践起来吧！本章将主要讲解iOS平台和Android平台上的音视频渲染，即接收到音视频的原始数据之后，如何利用iOS平台与Android平台的API渲染到扬声器（Speaker）或者屏幕（View）上。声音的渲染在iOS平台上会直接使用AudioUnit（AUGraph）之类的API接口，Android平台则使用OpenSL ES或者AudioTrack这两类接口。而对于视频画面的渲染，笔者会为大家介绍一种跨平台的渲染技术，即OpenGL ES，本章将根据两个平台各自提供的API来构造出OpenGL ES环境，然后再利用统一的OpenGL程序（Program）将一张图片渲染到屏幕上。

4.1 AudioUnit介绍与实践

在iOS平台上，所有的音频框架底层都是基于AudioUnit实现的，如图4-1所示。较高层次的音频框架包括：Media Player、AV Foundation、OpenAL和Audio Toolbox，这些框架都封装了AudioUnit，然后提供了更高层次的API（功能更少，职责更单一的接口）。

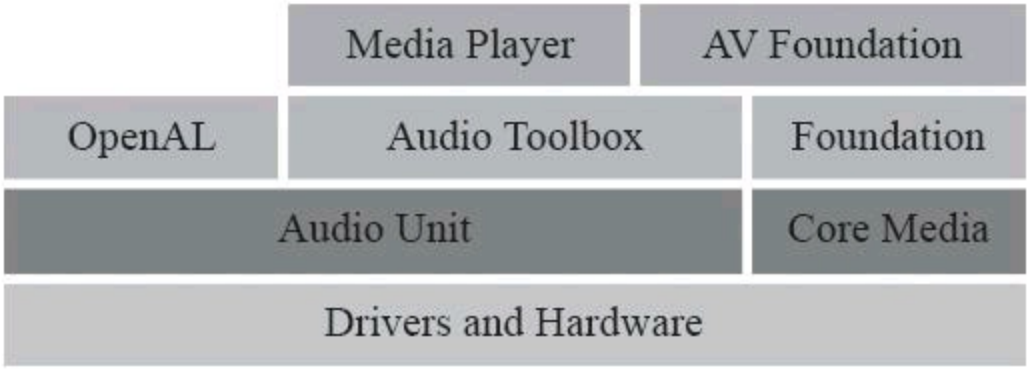


图 4-1

当开发者在开发音频相关产品的时候，如果对音频需要更高程度的控制、性能以及灵活性，或者想要使用一些特殊功能（回声消除）的时候，可以直接使用AudioUnit API。苹果官方文档中描述，AudioUnit提供了音频快速的模块化处理，如果是在以下场景中，更适合使用AudioUnit而不是使用高层次的音频框架。

·想使用低延迟的音频I/O（input或者output），比如说在VoIP的应用场景下。

·多路声音的合成并且回放，比如游戏或者音乐合成器的应用。

·使用AudioUnit里面提供的特有功能，比如：回声消除、Mix两轨音频，以及均衡器、压缩器、混响器等效果器。

·需要图状结构来处理音频，可以将音频处理模块组装到灵活的图状结构中，苹果公司为音频开发者提供了这种API。

本节的讲解将会从创建音频会话开始，然后构建一个AudioUnit，并给AudioUnit设置参数，然后介绍AudioUnit的分类，最终会构建一个

AUGraph来完成一个音频播放的功能，现在让我们开始本节的学习吧。

1.认识AudioSession

在iOS的音视频开发中，使用具体API之前都会先创建一个会话，这里也不例外。但在这之前，先来认识一下音频会话（AudioSession），其用于管理与获取iOS设备音频的硬件信息，并且是以单例的形式存在。可以使用如下代码来获取AudioSession的实例：

```
AVAudioSession *audioSession = [AVAudioSession sharedInstance];
```

获得AudioSession的实例之后，就可以设置以何种方式使用音频硬件做哪些处理了，基本的设置具体如下所示。

1）根据我们需要硬件设备提供的能力来设置类别：

```
[audioSession setCategory:AVAudioSessionCategoryPlayAndRecord error:&error];
```

2）设置I/O的Buffer，Buffer越小则说明延迟越低：

```
NSTimeInterval bufferDuration = 0.002;
[audioSession setPreferredIOBufferDuration:bufferDuration error:&error];
```

3）设置采样频率，让硬件设备按照设置的采样频率来采集或者播放音频：

```
double hwSampleRate = 44100.0;
[audioSession setPreferredSampleRate:hwSampleRate error:&error];
```

4）当设置完毕所有的参数之后就可以激活AudioSession了，代码如下：

```
[audioSession setActive:YES error:&error];
```

2.构建AudioUnit

在创建并启用音频会话之后，就可以构建AudioUnit了。构建AudioUnit的时候需要指定类型（Type）、子类型（subtype）以及厂商（Manufacture）。类型（Type）就是在下一小节提到的四大类型的AudioUnit的Type；而子类型（subtype）就是该大类型下面的子类型（比如Effect该大类型下面有EQ、Compressor、limiter等子类型）；厂商（Manufacture）一般情况下比较固定，直接写成kAudioUnitManufacturer_Apple就可以了。利用以上这三个变量开发者可以完整描述出一个AudioUnit了，比如使用下面的代码创建一个RemoteIO类型的AudioUnit：

```
AudioComponentDescription ioUnitDescription;
ioUnitDescription.componentType = kAudioUnitType_Output;
ioUnitDescription.componentSubType = kAudioUnitSubType_RemoteIO;
ioUnitDescription.componentManufacturer=kAudioUnitManufacturer_Apple;
ioUnitDescription.componentFlags = 0;
ioUnitDescription.componentFlagsMask = 0;
```

上述代码构造了RemoteIO类型的AudioUnit描述的结构体，那么如何使用这个描述来构造真正的AudioUnit呢？有两种方式：第一种方式是直接使用AudioUnit裸的创建方式；第二种方式是使用AUGraph和AUNode（其实一个AUNode就是对AudioUnit的封装，可以理解为一个AudioUnit的Wrapper）来构建。下面就来分别介绍这两种方式。

（1）裸创建方式

首先根据AudioUnit的描述，找出实际的AudioUnit类型：

```
AudioComponent ioUnitRef = AudioComponentFindNext(NULL, &ioUnitDescription);
```

然后声明一个AudioUnit引用：

```
AudioUnit ioUnitInstance;
```

最后根据类型创建出这个AudioUnit实例：

```
AudioComponentInstanceNew(ioUnitRef, &ioUnitInstance);
```

（2）AUGraph创建方式

首先声明并且实例化一个AUGraph：

```
AUGraph processingGraph;
NewAUGraph (&processingGraph);
```

然后按照AudioUnit的描述在AUGraph中增加一个AUNode：

```
AUNode ioNode;
AUGraphAddNode (processingGraph, &ioUnitDescription, &ioNode);
```

接下来打开AUGraph，其实打开AUGraph的过程也是间接实例化AUGraph中所有的AUNode。注意，必须在获取AudioUnit之前打开整个AUGraph，否则我们将不能从对应的AUNode中获取正确的AudioUnit：

```
AUGraphOpen (processingGraph);
```

最后在AUGraph中的某个Node里获得AudioUnit的引用：

```
AudioUnit ioUnit;
AUGraphNodeInfo (processingGraph, ioNode, NULL, &ioUnit);
```

无论是使用上面的哪一种方式创建AudioUnit，都可以创建出我们想要的AudioUnit，而具体应该使用哪一种方式来创建AudioUnit，还需要根据实际的应用场景来决定。但是笔者在实际的工作经验中认识到，使用AUGraph的结构在应用中可以搭建出扩展性更高的系统，所以推荐使用第二种方式，本章后面的实例代码都是使用第二种方式（AUGraph的架构）来搭建整个音频处理系统的。

3.AudioUnit的通用参数设置

本节将以RemoteIO这个AudioUnit为例来讲解AudioUnit的参数设置，RemoteIO这个AudioUnit是与硬件IO相关的一个Unit，它分为输入端和输出端（I代表Input，O代表Output）。输入端一般是指麦克风，输出端一般是指扬声器（Speaker）或者耳机。如果需要同时使用输入输出，即K歌应用中的耳返功能（用户在唱歌或者说话的同时，耳机会将麦克风收录的声音播放出来，让用户能够听到自己的声音），则需要开发者做一些设置将它们连接起来，如图4-2所示。

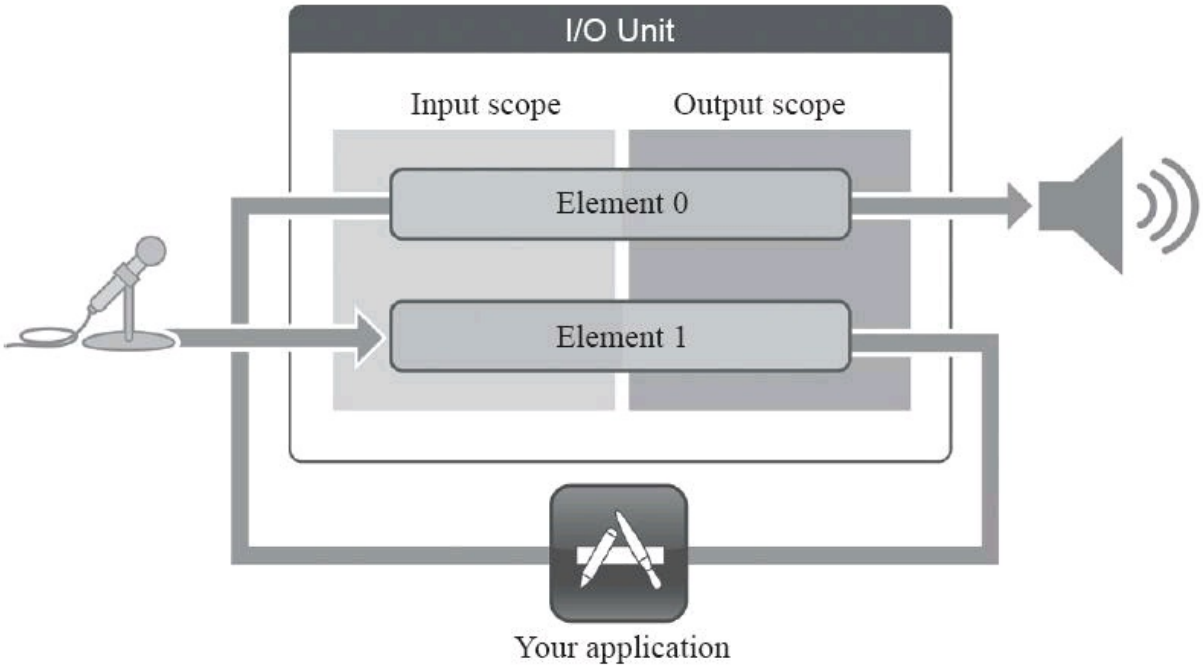


图　4-2

图4-2中RemoteIO Unit分为Element0和Element1，其中Element0控制输出端，Element1控制输入端，同时每个Element又分为Input Scope和Output Scope。如果开发者想要使用扬声器的声音播放功能，那么必须将这个Unit的Element0的OutputScope和Speaker进行连接。而如果开发者想要使用麦克风的录音功能，那么必须将这个Unit的Element1的InputScope和麦克风进行连接。使用扬声器的代码如下：

```
OSStatus status = noErr;
UInt32 oneFlag = 1;
UInt32 busZero = 0;// Element 0
status = AudioUnitSetProperty(remoteIOUnit,
        kAudioOutputUnitProperty_EnableIO,
        kAudioUnitScope_Output,
        busZero,
```

```
        &oneFlag,
        sizeof(oneFlag));
CheckStatus(status, @"Could not Connect To Speaker", YES);
```

上面这段代码就是把RemoteIOUnit的Element0的OutputScope连接到Speaker上，连接过程会返回一个OSStatus类型的值，可以使用自定义的CheckStatus函数来判断错误并且输出Could not Connect To Speaker的提示。具体的CheckStatus函数如下：

```
static void CheckStatus(OSStatus status, NSString *message, BOOL fatal)
{
    if(status != noErr)
    {
        char fourCC[16];
        *(UInt32 *)fourCC = CFSwapInt32HostToBig(status);
        fourCC[4] = '\0';
        if(isprint(fourCC[0]) && isprint(fourCC[1]) && isprint(fourCC[2]) &&
                isprint(fourCC[3]))
            NSLog(@"%@: %s", message, fourCC);
        else
            NSLog(@"%@: %d", message, (int)status);
        if(fatal)
            exit(-1);
    }
```

接下来再来看一下如何启用麦克风的代码：

```
UInt32 busOne = 1; // Element 1
AudioUnitSetProperty(remoteIOUnit,
        kAudioOutputUnitProperty_EnableIO,
        kAudioUnitScope_Input,
        busOne,
        &oneFlag,
        sizeof(oneFlag);
```

上面这段代码就是把RemoteIOUnit的Element1的InputScope连接上麦克风。连接成功之后，就应该给AudioUnit设置数据格式了，AudioUnit的数据格式分为输入和输出两个部分，下面先来看一个Audio Stream Format的描述：

```
UInt32 bytesPerSample = sizeof(Float32);
AudioStreamBasicDescription asbd;
bzero(&asbd, sizeof(asbd));
```

```
asbd.mFormatID = kAudioFormatLinearPCM;
asbd.mSampleRate = _sampleRate;
asbd.mChannelsPerFrame = channels;
asbd.mFramesPerPacket = 1;
asbd.mFormatFlags = kAudioFormatFlagsNativeFloatPacked |
                kAudioFormatFlagIsNonInterleaved;
asbd.mBitsPerChannel = 8 * bytesPerSample;
asbd.mBytesPerFrame = bytesPerSample;
asbd.mBytesPerPacket = bytesPerSample;
```

上面这段代码展示了如何填充AudioStreamBasicDescription结构体，其实在iOS平台做音视频开发久了就会知道：不论音频还是视频的API都会接触到很多StreamBasic Description，该Description就是用来描述音视频具体格式的。下面就来具体分析一下上述代码是如何指定格式的。

·mFormatID参数可用来指定音频的编码格式，此处指定音频的编码格式为PCM格式。

·接下来是设置声音的采样率、声道数以及每个Packet有几个Frame。

·mFormatFlags是用来描述声音表示格式的参数，代码中的第一个参数指定每个sample的表示格式是Float格式，这点类似于之前讲解的每个sample都是使用两个字节（SInt16）来表示；然后是后面的参数NonInterleaved，字面理解这个单词的意思是非交错的，其实对于音频来讲就是左右声道是非交错存放的，实际的音频数据会存储在一个AudioBufferList结构中的变量mBuffers中，如果mFormatFlags指定的是NonInterleaved，那么左声道就会在mBuffers[0]里面，右声道就会在mBuffers[1]里面；而如果mFormatFlags指定的是Interleaved的话，那么左右声道就会交错排列在mBuffers[0]里面，理解这一点对于后续的开发将是十分重要的。

·接下来的mBitsPerChannel表示的是一个声道的音频数据用多少位来表示，前面已经提到过每个采样使用Float来表示，所以这里是使用8乘以每个采样的字节数来赋值。

·最终是参数mBytesPerFrame和mBytesPerPacket的赋值，这里需要根据mFormatFlags的值来进行分配，如果在NonInterleaved的情况下，就赋值为bytesPerSample（因为左右声道是分开存放的）；但如果是

Interleaved的话，那么就应该是bytesPerSample*channels（因为左右声道是交错存放的），这样才能表示一个Frame里面到底有多少个byte。

至此，我们就完全构造好了这个BasicDescription结构体，下面将这个结构体设置给对应的AudioUnit，代码如下：

```
AudioUnitSetProperty( remoteIOUnit,kAudioUnitProperty_StreamFormat,
        kAudioUnitScope_Output, 1, &asbd, sizeof(asbd));
```

4.AudioUnit的分类

介绍完了AudioUnit的通用设置之后，本节就来介绍一下AudioUnit的分类。iOS按照AudioUnit的用途将AudioUnit分为五大类型，本节将从全局的角度出发来认识各大类型以及其下的子类型，并且还会介绍它们的用途，以及对应参数的意义。

（1）Effect Unit

类型是kAudioUnitType_Effect，主要提供声音特效处理的功能。其子类型及用途说明如下。

·均衡效果器：子类型是kAudioUnitSubType_NBandEQ，主要作用是为声音的某些频带增强或者减弱能量，该效果器需要指定多个频带，然后为各个频带设置宽度以及增益，最终将改变声音在频域上的能量分布。

·压缩效果器：子类型是kAudioUnitSubType_DynamicsProcessor，主要作用是当声音较小的时候可以提高声音的能量，当声音的能量超过了设置的阈值时，可以降低声音的能量，当然应合理地设置作用时间、释放时间以及触发值，使得最终可以将声音在时域上的能量压缩到一定范围之内。

·混响效果器：子类型是kAudioUnitSubType_Reverb2，对于人声处理来讲这是非常重要的效果器，可以想象自己身处在一个空房子中，如果有非常多的反射声和原始声音叠加在一起，那么从听感上可能会更有震撼力，但是同时原始声音也会变得更加模糊，原始声音的一些细节会被遮盖掉，所以混响设置的大或者小对于不同的人来讲会很不一致，可

以根据自己的喜好来进行设置。

Effect Unit下最常使用的就是上述三种效果器，当然其下还有很多种子类型的效果器，像高通（High Pass）、低通（Low Pass）、带通（Band Pass）、延迟（Delay）、压限（Limiter）等效果器，大家可以自行尝试使用一下，感受一下各自的效果。

（2）Mixer Units

类型是kAudioUnitType_Mixer，主要提供Mix多路声音的功能。其子类型及用途如下。

·3D Mixer：该效果器在移动设备上是无法使用的，仅仅在OS X上可以使用，所以这里不做介绍。

·MultiChannelMixer：子类型是kAudioUnitSubType_MultiChannelMixer，该效果器将是本书重点介绍的对象，它是多路声音混音的效果器，可以接收多路音频的输入，还可以分别调整每一路音频的增益与开关，并将多路音频合并成一路，该效果器在处理音频的图状结构中非常有用。

（3）I/O Units

类型是kAudioUnitType_Output，它的用途就像其分类的名字一样，主要提供的就是I/O的功能。其子类型及用途说明如下。

·RemoteIO：子类型是kAudioUnitSubType_RemoteIO，从名字上也可以看出，这是用来采集音频与播放音频的，其实当开发者的应用场景中要使用麦克风及扬声器的时候会用到该AudioUnit。

·Generic Output：子类型是kAudioUnitSubType_GenericOutput，当开发者需要进行离线处理，或者说在AUGraph中不使用Speaker（扬声器）来驱动整个数据流，而是希望使用一个输出（可以放入内存队列或者进行磁盘I/O操作）来驱动数据流时，就使用该子类型。

（4）Format Converter Units

类型是kAudioUnitType_FormatConverter，主要用于提供格式转换的功能，比如：采样格式由Float到SInt16的转换、交错和平铺的格式转

换、单双声道的转换等，其子类型及用途说明如下。

·AUConverter：子类型是kAudioUnitSubType_AUConverter，它将是本书要重点介绍的格式转换效果器，当某些效果器对输入的音频格式有明确的要求时（比如3D Mixer Unit就必须使用UInt16格式的sample），或者开发者将音频数据输入给一些其他的编码器进行编码，又或者开发者想使用SInt16格式的PCM裸数据在其他CPU上进行音频算法计算等的场景下，就需要使用到这个ConverterNode了。下面来看一个比较典型的场景，我们自定义一个音频播放器（代码仓库中的AudioPlayer项目），由FFmpeg解码出来的PCM数据是SInt16格式的，因此不能直接输送给RemoteIO Unit进行播放，所以需要构建一个ConvertNode将SInt16格式表示的数据转换为Float32格式表示的数据，然后再输送给RemoteIO Unit，最终才能正常播放出来。

·Time Pitch：子类型是kAudioUnitSubType_NewTimePitch，即变速变调效果器，这是一个很有意思的效果器，可以对声音的音高、速度进行调整，像“会说话的Tom猫”类似的应用场景就可以使用这个效果器来实现。

（5）Generator Units

类型是kAudioUnitType_Generator，在开发中我们经常使用它来提供播放器的功能。其子类型及用途说明如下。

·AudioFilePlayer：子类型是kAudioUnitSubType_AudioFilePlayer，在AudioUnit里面，如果我们的输入不是麦克风，而希望其是一个媒体文件，当然，也可以类似于代码仓库中的AudioPlayer项目自行解码，转换之后将数据输送给RemoteIO Unit播放出来，但是其实还有一种更加简单、方便的方式，那就是使用AudioFilePlayer这个AudioUnit，可以参考代码仓库中的AUPlayer项目，该项目就是利用AudioFilePlayer作为输入数据源来提供数据的。需要注意的是，必须在初始化AUGraph之后，再去配置AudioFilePlayer的数据源以及播放范围等属性，否则就会出现错误，其实数据源还是会调用AudioFile的解码功能，将媒体文件中的压缩数据解压成为PCM裸数据，最终再交给AudioFilePlayer Unit进行后续处理。

5.构造一个AUGraph

实际的K歌应用场景，会对用户发出的声音进行处理，并且立即给用户一个耳返（在50ms之内将声音输出到耳机中，让用户可以听到）。那么如何让RemoteIOUnit利用麦克风采集出来的声音，经过中间效果器的处理，最终输出到Speaker中播放给用户呢？下面就来介绍一下如何以AUGraph的方式将声音采集、声音处理以及声音输出的整个过程管理起来。先来看一下图4-3。

如图4-3所示，首先要知道数据可以在通道中传递是由最右端Speaker（RemoteIO Unit）来驱动的，它会向其前一级——AUNode要数据，然后它的前一级会继续向上一级节点要数据，并最终从RemoteIOUnit的Element1（即麦克风）中要数据，这样就可以将数据按照相反的方向一级一级地传递下去，最终传递到RemoteIOUnit的Element0（即Speaker）并播放给用户听到。当然你可能会想到离线处理的时候应该由谁来进行驱动呢？其实在进行离线处理的时候应该使用Mixer Unit大类型下面子类型为Generic Output的AudioUnit来做驱动端。那么这些AudioUnit或者说AUNode是如何进行连接的呢？有两种方式，第一种方式是直接将AUNode连接起来；第二种方式是通过回调的方式将两个AUNode连接起来。下面就来分别介绍这两种方式。

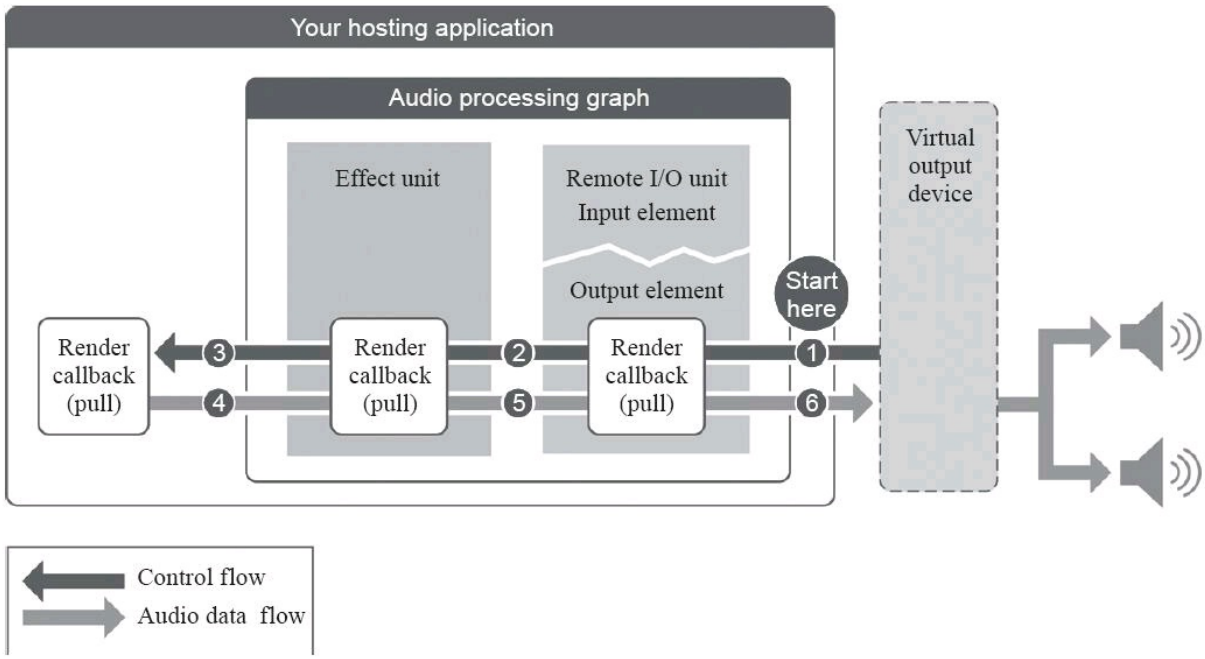


图　4-3

（1）直接连接的方式

```
AUGraphConnectNodeInput(mPlayerGraph, mPlayerNode, 0, mPlayerIONode, 0);
```

以上是本节AUPlayer实例中的一段代码，目标是将Audio File Player Unit和RemoteIO Unit直接连接起来，当RemoteIO Unit需要播放数据的时候，就会调用AudioFilePlayer Unit来获取数据，这样就把这两个AudioUnit连接起来了。

（2）回调的方式

```
AURenderCallbackStruct renderProc;
renderProc.inputProc = &inputAvailableCallback;
renderProc.inputProcRefCon = (__bridge void *)self;
AUGraphSetNodeInputCallback(mGraph, ioNode, 0, &finalRenderProc);
```

这段代码首先是构造一个AURenderCallback的结构体，并指定一个回调函数，然后设置给RemoteIO Unit的输入端，当RemoteIO Unit需要数据输入的时候就会回调该回调函数，回调函数代码如下：

```
static OSStatus renderCallback(void *inRefCon, AudioUnitRenderActionFlags
        *ioActionFlags, const AudioTimeStamp *inTimeStamp, UInt32
        inBusNumber, UInt32 inNumberFrames, AudioBufferList *ioData)
{
    OSStatus result = noErr;
    __unsafe_unretained AUGraphRecorder *THIS = (__bridge
            AUGraphRecorder *)inRefCon;
    AudioUnitRender(THIS->mixerUnit, ioActionFlags, inTimeStamp, 0,
            inNumberFrames, ioData);
    result = ExtAudioFileWriteAsync(THIS->finalAudioFile, inNumberFrames,
            ioData);
    return result;
}
```

该回调函数主要完成两件事情：第一件事情是去Mixer Unit里面要数据，通过调用AudioUnitRender的方式来驱动Mixer Unit获取数据，得到数据之后放入ioData中，从而填充回调方法中的参数，将Mixer Unit与RemoteIO Unit连接了起来；第二件事情则是利用ExtAudioFile将这段声音编码并写入本地磁盘的一个文件中。

本节的代码仓库中包含了两个实例项目：一个是AUPlayer，利用AudioFilePlayer Unit和RemoteIO Unit做了一个最简单的播放器；另外一

个是AudioPlayer，它会利用FFmpeg进行解码操作，解码出来的是SInt16格式表示的数据，然后再通过一个ConvertNode将其转换为Float32格式表示的数据，最终输送给RemoteIO Unit进行播放。将这两个项目对比来看，第二种方式十分不便，其实以第二种方式实现播放器有两个目的：其一是为了让大家体验在开发iOS平台的程序时，优先使用iOS平台自身提供的API，了解它们的便捷性与重要性；其二是为了给后续的视频播放器项目打下基础。大家可以好好学习一下这两个实例，充分感受一下iOS平台为开发者提供了多么强大的多媒体开发API。

4.2 Android平台的音频渲染

Android的SDK（指的是Java层提供的API，对应的NDK是Native层提供的API，即C或者C++层可以调用的API）提供了3套音频播放的API，分别是：MediaPlayer、SoundPool和AudioTrack。这三个API的使用场景各不相同，简单来说具体如下。

·MediaPlayer：适合在后台长时间播放本地音乐文件或者在线的流式媒体文件，它的封装层次比较高，使用方式比较简单。

·SoundPool：适合播放比较短的音频片段，比如游戏声音、按键声音、铃声片段等，它可以同时播放多个音频。

·AudioTrack：适合低延迟的播放，是更加底层的API，提供了非常强大的控制能力，适合流媒体的播放等场景，由于其属于底层API，所以需要结合解码器来使用。

Android的NDK提供了OpenSL ES的C语言的接口，可以提供非常强大的音效处理、低延时播放等功能，比如在Android手机上可实现实时耳返的功能。本书的项目案例中会更多地使用到底层API的功能，下面就来详细地介绍AudioTrack与OpenSL ES这两个API的使用。

4.2.1 AudioTrack的使用

由于AudioTrack是Android SDK层提供的最底层的音频播放API，因此只允许输入裸数据。和MediaPlayer相比，对于一个压缩的音频文件（比如MP3、AAC等文件），它需要自行实现解码操作和缓冲区控制。因为这里只涉及AudioTrack的音频渲染端（解码部分已经在前面章节中介绍过了，对于缓冲区的控制机制，后续章节将会详细讲解），所以本节只介绍如何使用AudioTrack渲染音频PCM数据。

首先来看一下AudioTrack的工作流程，具体如下。

1）根据音频参数信息，配置出一个AudioTrack的实例。

2）调用play方法，将AudioTrack切换到播放状态。

3）启动播放线程，循环向AudioTrack的缓冲区中写入音频数据。

4）当数据写完或者停止播放的时候，停止播放线程，并且释放所有资源。

根据AudioTrack的上述工作流程，本节将以4个小部分分别介绍每个流程的详细步骤。

1.配置AudioTrack

先来看一下AudioTrack的参数配置，要想构造出一个AudioTrack类型的实例，必须先了解其构造函数原型，代码如下所示：

```
public AudioTrack(int streamType, int sampleRateInHz, int channelConfig,
        int audioFormat, int bufferSizeInBytes, int mode);
```

其中构造函数的参数说明如下。

·streamType，Android手机上提供了多重音频管理策略（读者按一下手机侧边的按键，可以看到有多个音量管理，这其实就是不同音频策略的音量控制展示），当系统有多个进程需要播放音频的时候，管理策

略会决定最终的呈现效果，该参数的可选值将以常量的形式定义在类AudioManager中，主要包括以下内容。

```
STREAM_VOCIE_CALL：电话声音
STREAM_SYSTEM：系统声音
STREAM_RING：铃声
STREAM_MUSCI：音乐声
STREAM_ALARM：警告声
STREAM_NOTIFICATION：通知声
```

·sampleRateInHz，采样率，即播放的音频每秒钟会有多少次采样，可选用的采样频率列表为：8000、16000、22050、24000、32000、44100、48000等，大家可以根据自己的应用场景进行合理的选择。

·channelConfig，声道数（通道数）的配置，可选值以常量的形式配置在类AudioFormat中，常用的是CHANNEL_IN_MONO（单声道）、CHANNEL_IN_STEREO（双声道），因为现在大多数手机的麦克风都是伪立体声的采集，为了性能考虑，笔者建议使用单声道进行采集，而转变为立体声的过程可以在声音的特效处理阶段来完成。

·audioFormat，该参数是用来配置“数据位宽”的，即采样格式，可选值以常量的形式定义在类AudioFormat中，分别为ENCODING_PCM_16BIT（16bit）、ENCODING_PCM_8BIT（8bit），注意，前者是可以兼容所有Android手机的。

·bufferSizeInBytes，其配置的是AudioTrack内部的音频缓冲区的大小，AudioTrack类提供了一个帮助开发者确定bufferSizeInBytes的函数，其原型具体如下：

```
int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat);
```

·在实际开发中，强烈建议由该函数计算出需要传入的bufferSizeInBytes，而不是自己手动计算。

·mode，AudioTrack提供了两种播放模式，可选的值以常量的形式定义在类AudioTrack中，一个是MODE_STATIC，需要一次性将所有的数据都写入播放缓冲区中，简单高效，通常用于播放铃声、系统提醒的音频片段；另一个是MODE_STREAM，需要按照一定的时间间隔不间

断地写入音频数据，理论上它可以应用于任何音频播放的场景。

2.将AudioTrack切换到播放状态

首先判断AudioTrack实例是否初始化成功，如果当前状态处于初始化成功的状态，那么就调用它的play方法，并切换到播放状态，代码如下：

```
if (null != audioTrack && audioTrack.getState() != AudioTrack.STATE_UNINITIAL
{
    audioTrack.play();
}
```

3.开启播放线程

首先创建一个播放线程，代码如下：

```
playerThread = new Thread(new PlayerThread(), "playerThread");
playerThread.start();
```

接下来看看该线程中执行的任务，代码如下：

```
class PlayerThread implements Runnable {
    private short[] samples;
    public void run() {
        samples = new short[minBufferSize];
        while(!isStop) {
            int actualSize = decoder.readSamples(samples);
            audioTrack.write(samples, actualSize);
        }
    }
}
```

线程中的minBufferSize是在初始化AudioTrack的时候获得的缓冲区大小，会对其进行换算，即以2个字节表示一个采样的大小，也就是2倍的关系（因为初始化的时候是以字节为单位的）；decoder是一个解码器，假设已经初始化成功，最后将调用write方法把从解码器中获得的PCM采样数据写入AudioTrack的缓冲区中，注意此方法是阻塞的方法，比如：一般要写入200ms的音频数据需要执行接近200ms的时间。

4.销毁资源

首先停止AudioTrack，代码如下：

```
if (null != audioTrack && audioTrack.getState() != AudioTrack.STATE_UNINITIAL
{
    audioTrack.stop();
}
```

然后停止线程：

```
isStop = true;
if (null != playerThread) {
    playerThread.join();
    playerThread = null;
}
```

最后释放AudioTrack：

```
audioTrack.release();
```

具体实例请参看代码仓库中的AudioPlayer项目的AudioTrack部分，需要把项目中resource目录下的音频文件放入目标设备的sdcard根目录下。

4.2.2 OpenSL ES的使用

OpenSL ES全称为Open Sound Library for Embedded Systems，即嵌入式音频加速标准。OpenSL ES是无授权费、跨平台、针对嵌入式系统精心优化的硬件音频加速API。它为嵌入式移动多媒体设备上的本地应用程序开发者提供了标准化、高性能、低响应时间的音频功能实现方法，同时还实现了软/硬件音频性能的直接跨平台部署，不仅降低了执行难度，而且促进了高级音频市场的发展。

图4-4描述了OpenSL ES的架构，在Android中，High Level Audio Libs是音频Java层API输入输出，属于高级API，相对来说，OpenSL ES则是比较低层级的API，属于C语言API。在开发中，一般会使用高级API，除非遇到性能瓶颈，如语音实时聊天、3D Audio、某些Effects等，开发者可以直接通过C/C++开发基于OpenSL ES音频的应用。需要声明的是：本书中使用的是OpenSL ES 1.0.1版本，这套API是在Android系统版本2.3之后才支持的，并且有一些高级功能也会受到一些限制比如解码AAC是在Android系统版本4.0以上才支持的。

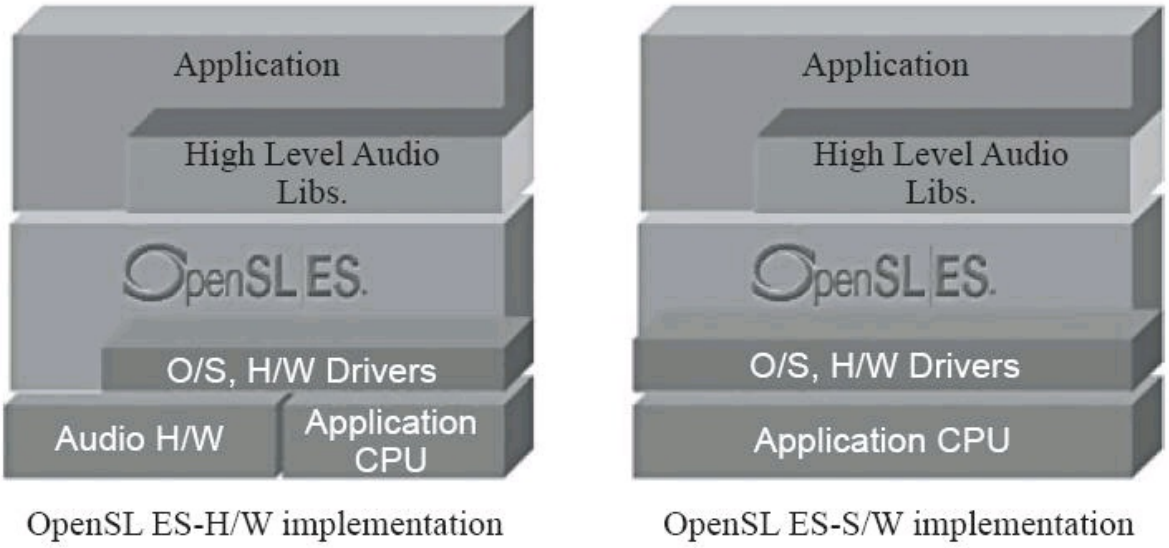


图 4-4

在使用OpenSL ES的API之前，需要引入OpenSL ES的头文件，代码如下：

```
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>
```

由于是在Native层使用该特性，所以要在Makefile文件Android.mk中增加链接选项，以便在链接阶段使用到系统提供的OpenSL ES的so库：

```
LOCAL_LDLIBS += -lOpenSLES
```

前文也提到了OpenSL ES提供的是基于C语言的API，但它是基于对象和接口的方式提供的，会采用面向对象的思想开发API。因此，这里需要先来了解一下OpenSL ES中对象和接口的概念。

·对象：对象是对一组资源及其状态的抽象，每个对象都有一个在其创建时指定的类型，类型决定了对象可以执行的任务集，对象有点类似于C++中类的概念。

·接口：接口是对象提供的一组特征的抽象，这些抽象会为开发者提供一组方法以及每个接口的类型功能，在代码中，接口的类型由接口ID来标识。

需要重点理解的是，一个对象在代码中其实是没有实际的表示形式的，可以通过接口来改变对象的状态以及使用对象提供的功能。对象可以有一个或者多个接口的实例，但是接口实例肯定只属于一个对象。如果读者读到此处已经完全理解了OpenSL ES中对象和接口的概念，那么就继续向下来看看在代码实例中是如何使用它们的。

上面也提到过，对象是没有实际的代码表示形式的，对象的创建也是通过接口来完成的。通过获取对象的方法来获取出对象，进而可以访问对象的其他接口方法或者改变对象的状态，具体的执行步骤如下。

1）创建一个引擎对象接口。引擎对象是OpenSL ES提供API的唯一入口，开发者需要调用全局函数slCreateEngine来获取SLObjectItf类型的引擎对象接口：

```
SLObjectItf engineObject;
SLEngineOption engineOptions[] = { { (SLuint32) SL_ENGINEOPTION_THREADSAFE,
    (SLuint32) SL_BOOLEAN_TRUE } };
slCreateEngine(&engineObject, ARRAY_LEN(engineOptions), engineOptions, 0, 0,
```

2）实例化引擎对象，需要通过在第1步得到的引擎对象接口来实例化引擎对象，否则会无法使用这个对象，其实在OpenSL ES的使用中，任何对象都需要使用接口来进行实例化，所以这里也需要封装出一个实例化对象的方法，代码如下：

```
RealizeObject(engineObject);
SLresult RealizeObject(SLObjectItf object) {
    return (*object)->Realize(object, SL_BOOLEAN_FALSE);
};
```

3）获取这个引擎对象的方法接口，通过GetInterface方法，使用第2步已经实例化好了的对象，获取对应的SLEngineItf类型的对象接口，该接口将会是开发者使用所有其他API的入口：

```
SLEngineItf engineEngine;
(*engineObject)->GetInterface(engineObject, SL_IID_ENGINE, &engineEngine);
```

4）创建需要的对象接口，通过调用SLEngineItf类型的对象接口的CreateXXX方法返回新的对象的接口，比如，调用CreateOutputMix方法来获取一个outputMixObject接口，或者调用CreateAudioPlayer方法来获取一个audioPlayerObject接口。由于篇幅有限，这里仅仅列出创建outputMixObject的接口代码，播放器接口的获取可以参考代码仓库中的代码：

```
SLObjectItf outputMixObject;
(*engineEngine)->CreateOutputMix(engineEngine, &outputMixObject, 0, 0, 0);
```

5）实例化新的对象，任何对象接口获取出来之后，都必须要实例化，与第2步操作其实是一样的：

```
realizeObject(outputMixObject);
realizeObject(audioPlayerObject);
```

6）对于某些比较复杂的对象，需要获取新的接口来访问对象的状态或者维护对象的状态，比如在播放器AudioPlayer或录音器

AudioRecorder中注册一些回调方法等，代码如下：

```
SLPlayItf audioPlayerPlay;
(*audioPlayerObject)->GetInterface(audioPlayerObject, SL_IID_PLAY,
        &audioPlayerPlay);
// 设置播放状态
(*audioPlayerPlay)->SetPlayState(audioPlayerPlay, SL_PLAYSTATE_PLAYING);
// 设置暂停状态
(*audioPlayerPlay)->SetPlayState(audioPlayerPlay, SL_PLAYSTATE_PAUSED);
```

7）待使用完该对象之后，要记得调用Destroy方法来销毁对象以及相关的资源：

```
destroyObject(audioPlayerObject);
destroyObject(outputMixObject);
void AudioOutput::destroyObject(SLObjectItf& object) {
    if (0 != object)
        (*object)->Destroy(object);
    object = 0;
}
```

使用OpenSL ES实现一个播放媒体文件的功能，即一个音频播放器的渲染端逻辑的实例，请读者查看代码仓库中的AudioPlayer项目的OpenSL ES部分。注意，需要把resource目录下的音频文件放入运行的sdcard根目录下。

4.3 视频渲染

4.3.1 OpenGL ES介绍

OpenGL（Open Graphics Library）定义了一个跨编程语言、跨平台编程的专业图形程序接口。可用于二维或三维图像的处理与渲染，它是一个功能强大、调用方便的底层图形库。对于嵌入式的设备，其提供了OpenGL ES（OpenGL for Embedded Systems）版本，该版本是针对手机、Pad等嵌入式设备而设计的，是OpenGL的一个子集。到目前为止，OpenGL已经经历过很多版本的迭代与更新，最新版本为3.0，而使用最广泛的还是OpenGL ES 2.0版本。本书所讲解的案例就是基于OpenGL ES 2.0接口进行编程并实现图像的处理与渲染的。本书只讨论OpenGL ES 2D部分的内容，不涉及3D部分的介绍，因为在视频应用这一场景下，绝大部分都是使用2D的处理与渲染，所以只需要具备2D部分的知识就可以完成绝大部分的工作了。

由于OpenGL是基于跨平台的设计，所以在每个平台上都要有它的具体实现，即要提供OpenGL ES的上下文环境以及窗口的管理。在OpenGL的设计中，OpenGL是不负责管理窗口的，窗口的管理将交由各个设备自己来完成，上下文环境也是一样的，其在各个平台上都有自己的实现。具体来讲，在iOS平台上使用EAGL提供本地平台对OpenGL ES的实现，在Android平台上使用EGL提供本地平台对OpenGL ES的实现。所以如果想要OpenGL程序运行在多个平台上，那么也要为每个平台编写自己的上下文环境的实现。

这里需要介绍一下另外一个库——libSDL，它可以为开发者提供面向libSDL的API编程，libSDL内部会解决多个平台的OpenGL上下文环境以及窗口管理问题，开发者只需要交叉编译这个库到各自的平台上就可以做到只写一份代码即可运行到多个平台。其中FFmpeg中的ffplay这一工具就是基于libSDL进行开发的。但是对于移动开发者来讲，这样就会失去一些更加灵活的控制，甚至某些场景下的功能不能实现，所以本书不会基于libSDL，而是基于每个平台裸用自己平台的API来提供OpenGL ES的本地实现。

上面介绍了OpenGL（ES）是什么，下面再来介绍一下OpenGL（ES）能做什么。其实从名字上就可以看出来，OpenGL主要是

做图形图像处理的库，尤其是在移动设备上进行图形图像处理，它的性能优势更能体现出来。前文曾提到过最主要的是使用OpenGL ES 2.0版本进行开发工作，若要使用OpenGL ES 2.0就不得不提到GLSL，GLSL（OpenGL Shading Language）是OpenGL的着色器语言，开发人员利用这种语言编写程序运行在GPU（Graphic Processor Unit，图形图像处理单元，可以理解为是一种高并发的运算器）上以进行图像的处理或渲染。GLSL着色器代码分为两个部分，即Vertex Shader（顶点着色器）与Fragment Shader（片元着色器）两部分，分别完成各自在OpenGL渲染管线中的功能，具体的语法与流程会在4.3.2节中介绍。对于OpenGL ES，业界有一个著名的开源库GPUImage，它的实现非常优雅，尤其是在iOS平台上实现得非常完备，不仅有摄像头采集实时渲染、视频播放器、离线保存等功能，更有强大的滤镜实现。在GPUImage的滤镜实现中，可以找到大部分图形图像处理Shader的实现，包括：亮度、对比度、饱和度、色调曲线、白平衡、灰度等调整颜色的处理，以及锐化、高斯模糊等图像像素处理的实现等，还有素描、卡通效果、浮雕效果等视觉效果的实现，最后还有各种混合模式的实现等。当然，除了GPUImage提供的这些图像处理的Shader之外，开发者也可以自己实现一些有意思的Shader，比如美颜滤镜效果、瘦脸效果以及粒子效果等。

那么如何利用OpenGL来完成上面所述的工作呢？请阅读4.3.2节，我们会在Android和iOS平台上分别实践如何使用OpenGL。

4.3.2 OpenGL ES的实践

1.OpenGL渲染管线

要想学习着色器，并理解着色器的工作机制，就要对OpenGL固定的渲染管线有深入的了解。同样，先来统一一下术语。

·几何图元：包括点、直线、三角形，均是通过顶点（vertex）来指定的。

·模型：根据几何图元创建的物体。

·渲染：计算机根据模型创建图像的过程。

最终渲染过程结束之后，人眼所看到的图像就是由屏幕上的所有像素点组成的，在内存中，这些像素点可以组织成一个大的一维数组，每4个Byte即表示一个像素点的RGBA数据，而在显卡中，这些像素点可以组织成帧缓冲区（FrameBuffer）的形式，帧缓冲区保存了图形硬件为了控制屏幕上所有像素的颜色和强度所需要的全部信息。理解了帧缓冲区的概念，接下来就来讨论一下OpenGL的渲染管线，这部分内容对于OpenGL来说是非常重要的。

那么OpenGL的渲染管线具体是做什么的呢？其实就是OpenGL引擎渲染图像的流程，也就是说OpenGL引擎是一步一步地将图片渲染到屏幕上去的过程。渲染管线分为以下几个阶段。

阶段一：指定几何对象

所谓几何对象，就是上面说过的几何图元，这里将根据具体执行的指令绘制几何图元。比如，OpenGL提供给开发者的绘制方法glDrawArrays，这个方法里面的第一个参数是mode，就是制定绘制方式，可选值有以下几种。

·GL_POINTS：以点的形式进行绘制，通常用在绘制粒子效果的场景中。

·GL_LINES：以线的形式进行绘制，通常用在绘制直线的场景中。

·GL_TRIANGLE_STRIP：以三角形的形式进行绘制，所有二维图像的渲染都会使用这种方式。

具体选用哪一种绘制方式决定了OpenGL渲染管线的第一阶段应如何去绘制几何图元，所以这就是第一阶段指定的几何对象。

阶段二：顶点处理

不论以上的几何对象是如何指定的，所有的几何数据都将会经过这个阶段。这个阶段所做的操作就是，根据模型视图和投影矩阵进行变换来改变顶点的位置，根据纹理坐标与纹理矩阵来改变纹理坐标的位置，如果涉及三维的渲染，那么这里还要处理光照计算和法线变换（本书不会涉及三维的渲染）。这里的输出是以gl_Position来表示具体的顶点位置的，如果是以点（GL_POINTS）来绘制几何图元，那么还应该输出gl_PointSize。

阶段三：图元组装

在经过阶段二的顶点处理操作之后，不论是模型的顶点，还是纹理坐标都是已经确定好了的。在这个阶段，顶点将会根据应用程序送往图元的规则（如GL_POINTS、GL_TRIANGLES等），将纹理组装成图元。

阶段四：栅格化操作

由阶段三传递过来的图元数据，在此将会被分解成更小的单元并对应于帧缓冲区的各个像素。这些单元称为片元，一个片元可能包含窗口颜色、纹理坐标等属性。片元的属性是根据顶点坐标利用插值来确定的，这其实就是栅格化操作，也就是确定好每一个片元是什么。

阶段五：片元处理

通过纹理坐标取得纹理（texture）中相对应的片元像素值（texel），根据自己的业务处理（比如提亮、饱和度调节、对比度调节、高斯模糊等）来变换这个片元的颜色。这里的输出是gl_FragColor，用于表示修改之后的像素的最终结果。

阶段六：帧缓冲操作

该阶段主要执行帧缓冲的写入操作，这也是渲染管线的最后一步，负责将最终的像素值写到帧缓冲区中。

前面也提到过，OpenGL ES 2.0版本与之前的版本相比，更出色的功能就是提供了可编程的着色器来代替OpenGL ES中渲染管线的某一阶段。那么具体是哪一个着色器，又可以替换渲染管线的哪一个阶段呢？具体如下所示。

·Vertex Shader（顶点着色器）用来替换顶点处理阶段。

·Fragment Shader（片元着色器，又称像素着色器）用来替换片元处理阶段。

glFinish和glFlush

提交给OpenGL的绘图指令并不会马上发送给图形硬件执行，而是放到一个缓冲区里面，等待缓冲区满了之后再将这些指令发送给图形硬件执行，所以指令较少或较简单时是无法填满缓冲区的，这些指令自然不能马上执行以达到所需要的效果。因此每次写完绘图代码，需要让其立即完成效果时，开发者都需要在代码后面添加glFlush（）或glFinish（）函数。

·glFlush（）的作用是将缓冲区中的指令（无论是否为满）立刻发送给图形硬件执行，发送完立即返回。

·glFinish（）的作用也是将缓冲区中的指令（无论是否为满）立刻发送给图形硬件执行，但是要等待图形硬件执行完成之后才返回这些指令。

在下面讲解GLSL语法以及内嵌函数时将会学习到具体如何实现顶点着色器和片元着色器，并且还会实现如何写出一组着色器（包括顶点着色器与片元着色器）为图片增加对比度的功能。

2.GLSL语法与内建函数

本节的目标是实现一组着色器来完成增强对比度的功能，但是还不能直接看到这组着色器的效果，因为着色器需要运行到显卡中，要想看到效果还需要继续完成后续的学习，所以先不要着急，我们慢慢来。

前面已经粗略介绍过GLSL是什么了，但是一直没有为它下过准确的定义，就是担心读者看到它的定义之后，觉得不可理解，现在，我们已经了解了渲染管线，应该也已经充分理解了着色器到底是做什么用的了，GLSL全称为OpenGL Shading Language，是为了实现着色器的功能而向开发人员提供的一种开发语言，对其只要能理解到这个层次就可以了。

（1）GLSL的修饰符与基本数据类型

具体来说，GLSL的语法与C语言非常类似，学习一门语言，首先要看它的数据类型表示，然后再学习具体的运行流程。对于GLSL，其数据类型表示具体如下。

首先是修饰符，具体如下：

·const：用于声明非可写的编译时常量变量。

·attribute：用于经常更改的信息，只能在顶点着色器中使用。

·uniform：用于不经常更改的信息，可用于顶点着色器和片元着色器。

·varying：用于修饰从顶点着色器向片元着色器传递的变量。

然后是基本数据类型，int、float、bool，这些与C语言都是一致的，需要强调的一点就是，这里面的float是有一个修饰符的，即可以指定精度。三种修饰符的范围（范围一般视显卡而定）和应用情况具体如下。

·highp：32bit，一般用于顶点坐标（vertex Coordinate）。

·medium：16bit，一般用于纹理坐标（texture Coordinate）。

·lowp：8bit，一般用于颜色表示（color）。

接下来是向量类型，向量类型是Shader中非常重要的一个数据类型，因为在做数据传递的时候需要经常传递多个参数，相较于写多个基本数据类型，使用向量类型是非常好的选择。列举一个最经典的例子，要将物体坐标和纹理坐标传递到Vertex Shader中，用的就是向量类型，

每一个顶点都是一个四维向量，在Vertex Shader中利用这两个四维向量即可完成自己的纹理坐标映射操作。声明方式如下（GLSL代码）：

```
attribute vec4 position;
```

之后是矩阵类型，矩阵类型在Shader的语法中也是一个非常重要的类型，有一些效果器需要开发者传入矩阵类型的数据，比如后面会接触到的怀旧效果器，就需要传入一个矩阵来改变原始的像素数据。声明方式如下（GLSL代码）：

```
uniform lowp mat4 colorMatrix;
```

上面的代码表示了一个4×4的浮点矩阵，如果是mat2就是2×2的浮点矩阵，如果是mat3就是3×3的浮点矩阵。若要传递一个矩阵到实际的Shader中，则可以直接调用如下函数（客户端代码）：

```
glUniformMatrix4fv(mColorMatrixLocation, 1, false, mColorMatrix);
```

紧接着是纹理类型，本章的最后一节（4.3.4节）将会介绍应该如何加载以及渲染纹理，但是这里要讲解的是如何声明这个类型，一般仅在Fragment Shader中使用这个类型，二维纹理的声明方式如下（GLSL代码）：

```
uniform sampler2D texSampler;
```

当客户端接收到这个句柄时，就可以为它绑定一个纹理，代码如下（客户端代码）：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);
glUniform1i(mGLUniformTexture, 0);
```

注意上述代码中第一行激活的是哪一个纹理句柄，第三行代码中的

第二个参数需要传递对应的Index，就像代码中激活的纹理句柄是GL_TEXTURE0，对应的Index就是0，如果激活的纹理句柄是GL_TEXTURE1，那么对应的Index就是1，在不同的平台上句柄的个数也不一样，但是一般都会在32个以上。

最后来看一下比较特殊的传递类型，在GLSL中有一个特殊的修饰符就是varying，这个修饰符修饰的变量均用于在Vertex Shader和Fragment Shader之间传递参数。首先在顶点着色器中声明这个类型的变量代表纹理的坐标点，并且对这个变量进行赋值，代码如下：

```
attribute vec2 texcoord;
varying vec2 v_texcoord;
void main(void)
{
    // 计算顶点坐标
    v_texcoord = texcoord;
}
```

紧接着在Fragment Shader中也声明同名的变量，然后使用texture2D方法取出二维纹理中该纹理坐标点上的纹理像素值，代码如下（GLSL代码）：

```
varying vec2 v_texcoord;
vec4 texel = texture2D(texSampler, v_texcoord);
```

取出了该坐标点上的像素值之后，就可以进行像素变化操作了，比如说提高对比度，最终将改变的像素值赋值给gl_FragColor。

（2）GLSL的内置函数与内置变量

首先来看内置变量，最常见的是两个Shader的输出变量。

先来看Vertex Shader的内置变量（GLSL代码）：

```
vec4 gl_position;
```

上述代码用来设置顶点转换到屏幕坐标的位置，Vertex Shader一定

要去更新这个数值。另外还有一个内置变量，代码如下（GLSL代码）：

```
float gl_pointSize;
```

在粒子效果的场景下，需要为粒子设置大小，改变该内置变量的值就是为了设置每一个粒子矩形的大小。

其次是Fragment Shader的内置变量，代码如下（GLSL代码）：

```
vec4 gl_FragColor;
```

上述代码用于指定当前纹理坐标所代表的像素点的最终颜色值。

然后是内置函数，具体的函数可以去官方文档中查询，这里仅介绍几个常用的函数。

·abs（genType x）：绝对值函数。

·floor（genType x）：向下取整函数。

·ceil（genType x）：向上取整函数。

·mod（genType x，genType y）：取模函数。

·min（genType x，genType y）：取得最小值函数。

·max（genType x，genType y）：取得最大值函数。

·clamp（genType x，genType y，genType z）：取得中间值函数。

·step（genType edge，genType x）：如果x<edge，则返回0.0，否则返回1.0。

·smoothstep（genType edge0，genType edge1，genType x）：如果x≤edge0，则返回0.0；如果x≥edge1，则返回1.0；如果edge0<x<edge1，则执行0～1之间的平滑差值。

·mix（genType x，genType y，genType a）：返回线性混合的x和y，用公式表示为：x*（1-a）+y*a，这个函数在mix两个纹理图像的时候非常有用。

其他的角度函数、指数函数、几何函数在这里就不再赘述了，大家可以去官方文档进行查询。对于一个语言的语法来讲，剩下的就是控制流部分了，而GLSL的控制流与C语言非常类似，既可以使用for、while以及do-while实现循环，也可以使用if和if-else进行条件分支的操作，在后面的实践过程中及GLSL代码中都会用到这些控制流，在这里将不再讲解这些枯燥的语法。

至此，GLSL的语法部分已经讲解得差不多了，毕竟本节所写的程序（Shader）都是运行在GPU上的，那么在CPU上运行的程序（应用程序）应该如何将这一组Shader交给OpenGL ES的渲染管线呢？下面就来介绍如何在应用程序中使用Shader。

3.创建显卡执行程序

前面已经学习了GLSL的语法以及内嵌函数，并且也已经完成了一组Shader的实例，那么，如何让显卡来运行这一组Shader呢？或者说如何用Shader来替换掉OpenGL渲染管线中的那两个阶段呢？下面就来学习一下如何将Shader传递给OpenGL的渲染管线。先来看一下图4-5，该图描述了如何创建一个显卡的可执行程序，后文中将其统称为Program。

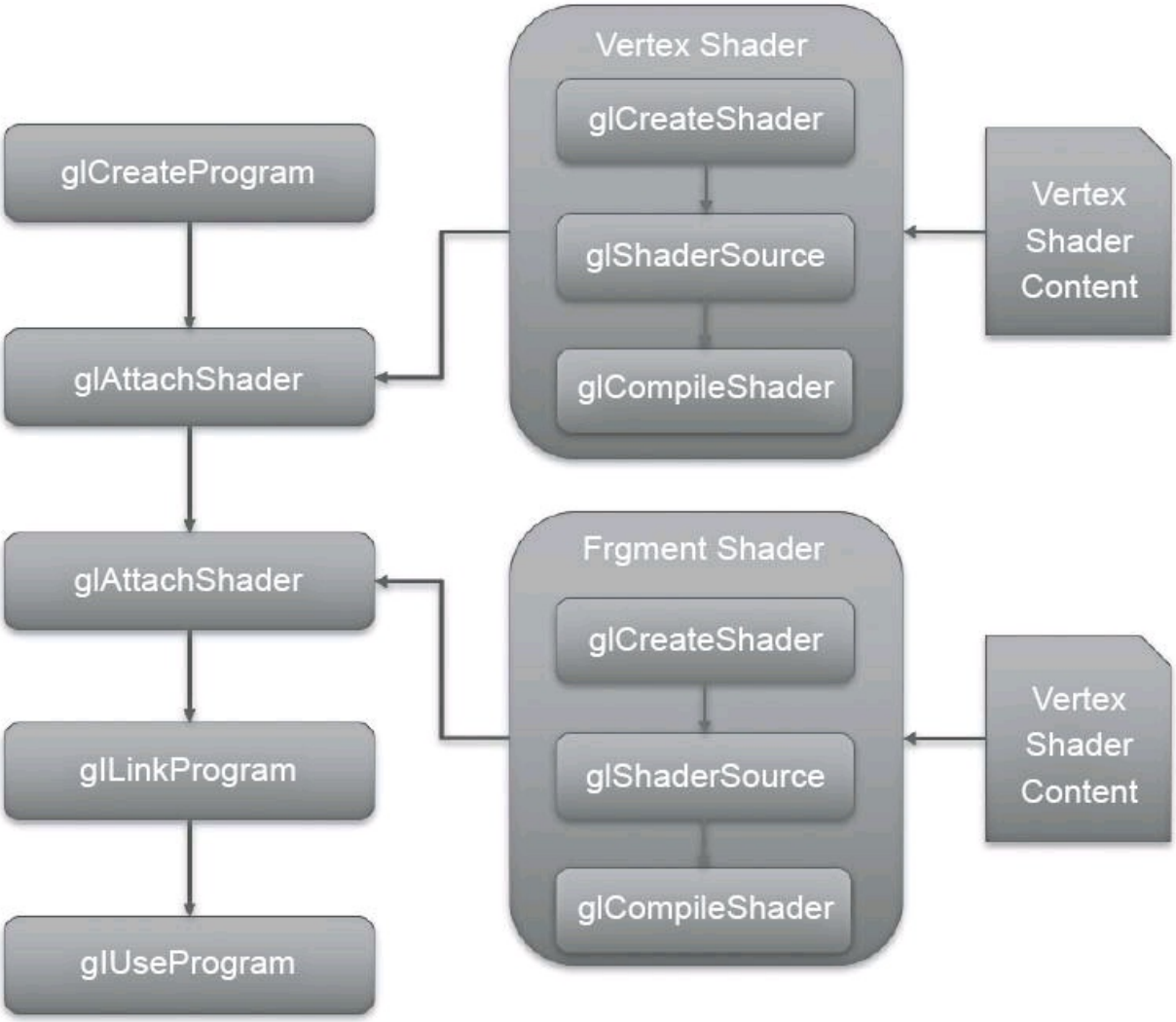


图　4-5

下面就按照图4-5逐一解释一下如何创建该Program。首先来看图4-5的右半部分，即创建shader的过程，第一步是调用glCreateShader方法创建一个对象，作为shader的容器，该函数会返回一个容器的句柄，函数的原型如下：

```
GLuint glCreateShader(GLenum shaderType);
```

函数原型中的参数shaderType有两种类型，当要创建VertexShader时，开发者应该传入类型GL_VERTEX_SHADER；当要创建FragmentShader时，开发者应该传入GL_FRAGMENT_SHADER类型。下一步就是为创建的这个shader添加源代码，即图4-5中最右边的这两个

Shader Content，它们就是前面讲解过的根据GLSL语法和内嵌函数编写的两个着色器程序（Shader），其为字符串类型。函数原型如下：

```
void glShaderSource(GLuint shader, int numOfStrings, const char **strings, ir
        *lenOfStrings)
```

上述函数的作用就是把开发者编写的着色器程序加载到着色器句柄所关联的内存中，最后一步就是编译该Shader，编译Shader的函数原型如下：

```
void glCompileShader(GLuint shader);
```

待编译完成之后，还需要验证该Shader是否编译成功了。那么，应该如何验证呢？使用下面的函数即可进行验证：

```
void glGetShaderiv (GLuint shader, GLenum pname, GLint* params);
```

其中第一个参数就是需要验证的Shader句柄；第二个参数是需要验证的Shader的状态值，这里一般是验证编译是否成功，该状态值一般是选取GL_COMPILE_STATUS；第三个参数是返回值。当返回值为1时，则说明该Shader是编译成功的；如果为0，则说明该Shader没有被编译成功，如果编译没有成功，那么开发者肯定需要知道到底是着色器代码中的哪一行出了问题，所以还需要调用上面的函数，只不过此时获取的是该Shader的另外一个状态，该状态值应该选取GL_INFO_LOG_LENGTH，返回值返回的则是错误原因字符串的长度，我们可以利用这个长度分配出一个buffer，然后调用获取Shader的InfoLog函数，函数原型如下：

```
void glGetShaderInfoLog(GLuint object, int maxLen, int *len, char *log);
```

之后可以把InfoLog打印出来，以帮助我们调试实际Shader中的错误。按照上面的步骤可以创建出Vertex Shader和Fragment Shader。接下来再来看图4-5的左半部分，即如何通过这两个Shader来创建Program（显卡可执行程序）。

首先创建一个对象，作为程序的容器，此函数将返回容器的句柄。函数原型如下：

```
GLuint glCreateProgram(void);
```

下面将把前文编译的Shader附加到刚刚创建的程序中，调用的函数名称如下：

```
void glAttachShader(GLuint program, GLuint shader);
```

第一个参数就是传入上一步返回的程序容器的句柄，第二个参数就是编译的Shader容器的句柄，当然要为每一个Shader都调用一次这个方法才能把两个Shader都关联到Program中去。最后一步就是链接程序，链接函数原型如下：

```
void glLinkProgram(GLuint program);
```

传入参数就是程序容器的句柄，那么这个程序到底有没有链接成功呢？OpenGL提供了一个函数来检查该程序的状态，函数原型如下：

```
glGetProgramiv (GLuint program, GLenum pname, GLint* params);
```

第一个参数就是传入程序容器的句柄，第二个参数代表需要检查该程序的哪一个状态，这里传入的是GL_LINK_STATUS，最后一个参数就是返回值。返回值为1则代表链接成功，如果返回值为0则代表链接失败，类似于编译Shader的操作，如果链接失败了，也可以获取错误信息，以便修改程序。如果想获取具体的错误信息，应该调用下属函数，但是第二个参数传递的是GL_INFO_LOG_LENGTH，代表获取该程序的InfoLog的长度，获取到长度之后我们分配出一个char*的内存空间以获取InfoLog，函数原型如下：

```
void glGetProgramInfoLog(GLuint object, int maxLen, int *len, char *log);
```

调用该函数返回InfoLog之后可以将其打印出来，以便于后续修改程序。至此就可以创建一个Program（显卡可执行程序）了，回顾一下整个过程，其实有点类似于C语言的编译和链接阶段，而构造OpenGL Program也是一样的，在构造Shader的过程中需要编译Shader，然后将两个Shader关联到具体的Program之后还需要链接该Program。接下来就是如何使用该程序，使用这个构建出来的程序也很简单，调用glUseProgram方法就可以了。至此本节要介绍的内容已基本介绍完毕，但是要想完全运行到手机上，还需要为OpenGL ES的运行提供一个上下文环境，下面就来学习在两个平台上如何为OpenGL ES提供上下文环境。

4.3.3 上下文环境搭建

就像前面提到的，OpenGL不负责窗口管理及上下文环境管理，该职责将由各个平台或者设备自行完成。为了在OpenGL的输出与设备的屏幕之间架接起一个桥梁，Khronos创建了EGL的API，EGL是双缓冲的工作模式，即有一个Back Frame Buffer和一个Front Frame Buffer，正常绘制操作的目标都是Back Frame Buffer，操作完毕之后，调用eglSwapBuffer这个API，将绘制完毕的FrameBuffer交换到Front Frame Buffer并显示出来。而在Android平台上，使用的是EGL这一套机制，EGL承担了为OpenGL提供上下文环境以及窗口管理的职责。iOS平台为OpenGL提供的实现则是EAGL（可以按照Eagle来发音），比较重要的是，iOS平台不允许直接渲染到屏幕上，因此要使用renderBuffer来代替。对于这两个平台的实现下面将分别给出详细的代码以及解释。

1.Android下的环境搭建

要在Android平台上使用OpenGL ES，第一种方式是直接使用GLSurfaceView，通过这种方式使用OpenGL ES比较简单，因为不需要开发者搭建OpenGL ES的上下文环境，以及创建OpenGL ES的显示设备。但是凡事都有两面，有好处也就有坏处，使用GLSurfaceView不够灵活，很多真正的OpenGL ES的核心用法（比如共享上下文来达到多线程共同操作一份纹理）都不能直接使用。所以本书的OpenGL ES上下文环境，都是直接使用EGL的API来搭建的，并且是基于C++的环境搭建的。因为如果仅仅在Java层编写，那么对于普通的应用也许可行，但是对于要进行解码或使用第三方库的场景（比如人脸识别），则需要到C++层来实施。出于效率和性能的考虑，这里的架构将直接使用Native层的EGL搭建一个OpenGL ES的开发环境。要想在Native层使用EGL，那么就必须要在Makefile文件（Android.mk）中加入EGL库，并在使用该库的C++文件中引入对应的头文件。

需要包含的头文件：

```
#include <EGL/egl.h>
#include <EGL/eglext.h>
```

需要引入的so库：

```
LOCAL_LDLIBS += -lEGL
```

这样就可以在Android的C++开发中使用EGL了，不过要想使用OpenGL ES，还需要引入OpenGL ES对应的头文件与库。

需要包含的头文件：

```
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>
```

需要引入的so库，注意这里使用的是OpenGL ES的2.0版本：

```
LOCAL_LDLIBS += -lGLESv2
```

至此，对于OpenGL的开发需要用到的头文件以及库文件就引入完毕了，下面再来看一下如何使用EGL搭建出OpenGL的上下文环境以及渲染的目标屏幕。

首先EGL需要知道绘制内容的目标在哪里，EGLDisplay是一个封装系统物理屏幕的数据类型（可以理解为绘制目标的一个抽象），通常会调用eglGetDisplay方法返回EGLDisplay来作为OpenGL ES渲染的目标。在调用该方法的时候，常量EGL_DEFAULT_DISPLAY会被传进该方法中，每个厂商通常都会返回默认的显示设备，代码如下：

```
if ((display = eglGetDisplay(EGL_DEFAULT_DISPLAY)) == EGL_NO_DISPLAY) {
    LOGE("eglGetDisplay() returned error %d", eglGetError());
    return false;
}
```

然后调用eglInitialize来初始化这个显示设备，该方法会返回一个布尔型变量来代表执行状态，后面两个参数则代表Major和Minor的版本，比如EGL的版本号是1.0，那么Major将返回1，Minor则返回0。如果不关心版本号，则可都传入0或者NULL，代码如下：

```
if (!eglInitialize(display, 0, 0)) {
    LOGE("eglInitialize() returned error %d", eglGetError());
    return false;
}
```

接下来就需要准备配置选项了，一旦EGL有了Display之后，它就可以将OpenGL ES的输出和设备的屏幕桥接起来，但是需要指定一些配置项，类似于色彩格式、像素格式、RGBA的表示以及SurfaceType等，不同的系统以及平台使用的EGL标准是不同的，Android平台下的配置代码如下所示：

```
const EGLint attribs[] = {EGL_BUFFER_SIZE, 32,
        EGL_ALPHA_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_RED_SIZE, 8,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
        EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
        EGL_NONE };
if (!eglChooseConfig(display, attribs, &config, 1, &numConfigs)) {
    LOGE("eglChooseConfig() returned error %d", eglGetError());
        return false;
}
```

最终可通过调用eglChooseConfig方法得到配置选项信息，接下来就需要创建OpenGL的上下文环境——EGLContext了，这里需要用到之前介绍过的EGLDisplay和EGLConfig，因为任何一条OpenGL指令都必须在自己的OpenGL上下文环境中运行，所以可以按照如下代码构建出OpenGL的上下文环境：

```
EGLint attributes[] = { EGL_CONTEXT_CLIENT_VERSION, 2, EGL_NONE };
if (!(context = eglCreateContext(display, config, NULL,
        eglContextAttributes))) {
    LOGE("eglCreateContext() returned error %d", eglGetError());
    return false;
}
```

函数eglCreateContext的第三个参数可以由开发者传入一个EGLContext类型的变量，该变量的意义是指可以与正在创建的上下文环境共享OpenGL资源，包括纹理ID、FrameBuffer以及其他的Buffer资源。这里暂时填写为NULL，代表不需要与其他的OpenGL ES上下文共

享任何资源，后文介绍的项目中在一些条件下其实是需要共享上下文的，这点将在后面进行讨论。

通过上面这三步创建OpenGL的上下文之后，说明EGL和OpenGL ES端的环境已经搭建完毕，即OpenGL ES的输出已经可以获取到了，那么应该如何将该输出渲染到设备的屏幕上呢？应该将EGL和设备的屏幕连接起来，只有这样EGL才是一个“桥”的功能，从而使得OpenGL ES的输出可以渲染到设备的屏幕上。那么如何将EGL和设备的屏幕连接起来呢？答案是使用EGLSurface，Surface实际上是一个FrameBuffer，通过EGL库提供的eglCreateWindowSurface可以创建一个可实际显示的Surface，通过EGL库提供的eglCreatePbufferSuface可以创建一个OffScreen的Surface，当然Surface也有很多属性，其中最基础的属性包括EGL_WIDTH、EGL_HEIGHT等，代码如下：

```
EGLSurface surface = NULL;
EGLint format;
if (!eglGetConfigAttrib(display, config, EGL_NATIVE_VISUAL_ID,
        &format)) {
    LOGE("eglGetConfigAttrib() returned error %d", eglGetError());
    return surface;
}
ANativeWindow_setBuffersGeometry(_window, 0, 0, format);
if (!(surface = eglCreateWindowSurface(display, config, _window, 0))) {
    LOGE("eglCreateWindowSurface() returned error %d", eglGetError());
}
```

有的读者可能会问_window是什么？这里需要重点解释一下，这个_window就是通过Java层的Surface对象创建出的ANativeWindow类型的对象，即本地设备屏幕的表示，在Android里面可以通过Surface（通过SurfaceView或者TextureView来得到或者构建出的Surface对象）构建ANativeWindow。这需要我们在使用的时候引用头文件：

```
#include <android/native_window.h>
#include <android/native_window_jni.h>
```

调用ANAtiveWindow API的代码如下：

```
ANativeWindow* window = ANativeWindow_fromSurface(env, surface);
```

env就是JNI层的JNIEnv指针类型的变量，surface就是jobject类型的变量，由Java层Surface对象传递而来。这样就可以把EGL和Java层的View（即设备的屏幕）连接起来了。如果要做离线的渲染，即在后台使用OpenGL进行一些图像的处理，就需要用到离线处理的Surface了，创建离线处理Surface的代码如下：

```
EGLSurface surface;
EGLint PbufferAttributes[] = { EGL_WIDTH, width, EGL_HEIGHT, height, EGL_NONE
        EGL_NONE };
if (!(surface = eglCreatePbufferSurface(display, config, PbufferAttributes)))
    LOGE("eglCreatePbufferSurface() returned error %d", eglGetError());
}
```

进行离线渲染的时候，可以使用这个Surface进行操作。

现在，EGL的准备工作已经做好了，一方面为OpenGL ES的渲染准备好了上下文环境，可以接收到OpenGL ES渲染出来的纹理，另外一方面连接好了设备的屏幕（Java层提供的SurfaceView或者TextureView），那么接下来就来具体看一下如何使用创建好的EGL环境进行工作。

首先需要明确一点，开发者需要开辟一个新的线程，来执行OpenGL ES的渲染操作，而且还必须为该线程绑定显示设备（Surface）与上下文环境（Context），因为每个线程都需要绑定一个上下文，这样才可以执行OpenGL的指令，所以首先需要调用eglMakeCurrent，来为该线程绑定Surface与Context。

```
eglMakeCurrent(display, eglSurface, eglSurface, context);
```

然后就可以执行RenderLoop循环了，每次循环都将调用OpenGL ES指令绘制图像。前文曾经提到过，EGL的工作模式是双缓冲模式，其内部有两个FrameBuffer（帧缓冲区，可以理解为是一个图像的存储区域），当EGL将一个FrameBuffer显示到屏幕上的时候，另外一个FrameBuffer就在后台等待OpenGL ES进行渲染输出了。直到调用函数eglSwapBuffers这条指令的时候，才会把前台的FrameBuffer和后台的FrameBuffer进行交换，这样用户就可以在屏幕上看到刚才OpenGL ES渲染输出的结果了。

最后所有的绘制操作执行完毕之后，需要销毁资源。注意销毁资源也必须在这个线程中，首先要销毁显示设备（EGLSurface）：

```
eglDestroySurface(display, eglSurface);
```

然后销毁上下文（Context）：

```
eglDestroyContext(display, context);
```

至此在Android平台上的Native层中，就可以使用EGL搭建起来的OpenGL ES的开发环境了，后面的章节中也会基于此进行业务开发。

2.iOS下的环境搭建

在iOS平台上不允许开发者使用OpenGL ES直接渲染屏幕，必须使用FrameBuffer与RenderBuffer来进行渲染。若要使用EAGL，则必须先创建一个RenderBuffer，然后让OpenGL ES渲染到该RenderBuffer上去。而该RenderBuffer则需要绑定到一个CAEAGLLayer上面去，这样开发者最后调用EAGLContext的presentRenderBuffer方法，就可以将渲染结果输出到屏幕上去了。实际上，在调用这个方法时，EAGL也会执行类似于前面的swapBuffer过程，将OpenGL ES渲染的结果绘制到物理屏幕上去（View的Layer），具体使用步骤如下。

首先编写一个View类，继承自UIView，然后重写父类UIView的一个方法layerClass，并且返回CAEAGLLayer类型：

```
+ (Class) layerClass
{
    return [CAEAGLLayer class];
}
```

然后在该View的initWithFrame方法中，获得layer并且强制类型转换为CAEAGLLayer类型的变量，同时为layer设置参数，其中包括色彩模式等属性：

```
- (id) initWithFrame:(CGRect)frame{
```

```
    if ((self = [super initWithFrame:frame])){
        CAEAGLLayer *eaglLayer = (CAEAGLLayer *)[self layer];
        NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
                [NSNumber numberWithBool:NO],
                kEAGLDrawablePropertyRetainedBacking,
                kEAGLColorFormatRGB565,
                kEAGLDrawablePropertyColorFormat,
                nil];
         [eaglLayer setOpaque:YES];
         [eaglLayer setDrawableProperties:dict];
    }
    return self;
}
```

接下来构造EAGLContext与RenderBuffer并且绑定到Layer上，之前也提到过，必须为每一个线程绑定OpenGL ES上下文。所以首先必须开辟一个线程，开发者在iOS中开辟一个新线程有多种方式，可以使用dispatch_queue，也可以使用NSOperationQueue，甚至使用pthread也可以，反正必须在一个线程中执行以下操作，首先创建OpenGL ES的上下文：

```
EAGLContext* _context;
_context = [[EAGLContext alloc]initWithAPI:kEAGLRenderingAPIOpenGLES2];
```

然后实施绑定操作，代码如下：

```
[EAGLContext setCurrentContext:_context];
```

此时就已经为该线程绑定了刚刚创建好的上下文环境了，也就是说已经建立好了EAGL与OpenGL ES的连接，接下来再建立另一端的连接。

创建帧缓冲区：

```
glGenFramebuffers(1, &_FrameBuffer);
```

创建绘制缓冲区：

```
glGenRenderbuffers(1, &renderbuffer);
```

绑定帧缓冲区到渲染管线：

```
glBindFramebuffer(GL_FRAMEBUFFER, _FrameBuffer);
```

绑定绘制缓存区到渲染管线：

```
glBindRenderbuffer(GL_RENDERBUFFER, _renderbuffer);
```

为绘制缓冲区分配存储区，此处将CAEAGLLayer的绘制存储区作为绘制缓冲区的存储区：

```
[_context renderbufferStorage:GL_RENDERBUFFER fromDrawable:(CAEAGLLayer*)
        self.layer]
```

获取绘制缓冲区的像素宽度：

```
glGetRenderBufferParameteriv(GL_RENDER_BUFFER, GL_RENDER_BUFFER_WIDTH,
        &_backingWidth);
```

获取绘制缓冲区的像素高度：

```
glGetRenderBufferParameteriv(GL_RENDER_BUFFER, GL_RENDER_BUFFER_HEIGHT,
        &_backingHeight);
```

将绘制缓冲区绑定到帧缓冲区：

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFF
        _renderbuffer);
```

检查FrameBuffer的status：

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE){
    // failed to make complete frame buffer object
}
```

至此我们就将EAGL与Layer（设备的屏幕）连接起来了，绘制完一帧之后（当然绘制过程也必须在这个线程之中），调用以下代码：

```
[_context presentRenderbuffer:GL_RENDERBUFFER];
```

这样就可以将绘制的结果显示到屏幕上了。至此我们就搭建好了iOS平台的OpenGL ES的上下文环境，后面章节会在此基础上进行业务开发。

4.3.4 OpenGL ES中的纹理

OpenGL中的纹理可以用来表示图像、照片、视频画面等数据，在视频渲染中，只需要处理二维的纹理，每个二维纹理都由许多小的纹理元素组成，它们都是小块数据，类似于前面章节所说的像素点。要使用纹理，最常用的方式是直接从一个图像文件加载数据。

为了访问到每一个纹理元素，每个二维纹理都有其自己的坐标空间，其范围是从左下角的（0，0）到右上角的（1，1）。按照惯例，一个维度称为S，而另一个则称为T。

如图4-6所示，对于OpenGL内部的纹理，坐标的方向性是规定的，t方向下面是0，上面是1，而对于s方向，左边是0，右边是1，从而构成了上述四个顶点的坐标位置，而中间的位置就是（0.5，0.5）。但是在这里还有另外一个坐标系的概念，那就是计算机系统里的坐标系，通常将x轴称为横轴，y轴称为纵轴，如图4-7所示。

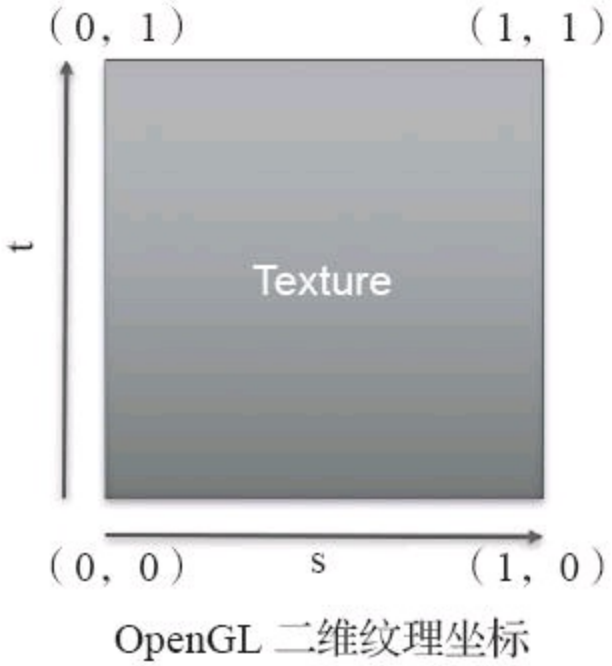


图 4-6

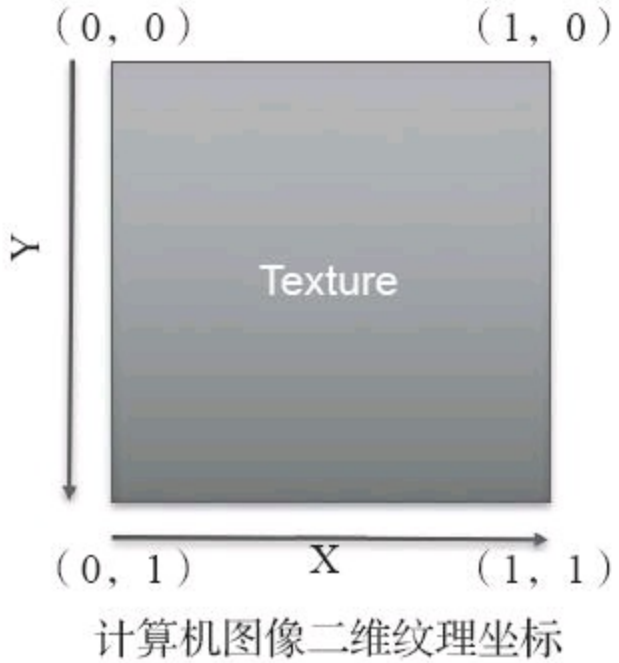


图　4-7

我们所熟知的不论是计算机还是手机的屏幕坐标系，x轴从左到右都是从0到1，y轴从上到下是从0到1，与图片的存储恰好是一致的，假设图片的存储是把所有的像素点都存储到一个大数组中，图片存储的第一个像素点也是左上角的像素点（即第一排第一列的像素点），然后是第二个像素点（第一排第二列）存储在数组的第二个元素中，那么，这里的坐标和OpenGL中的纹理坐标正好是做了一个180度的旋转，后面将会看到如何从本地图片中加载一张纹理并且渲染到界面上，而对于纹理坐标和计算机系统坐标的理解，在那时就会显得非常重要了。

下面再来看一下如何加载一张图片作为OpenGL中的纹理，首先要在显卡中创建一个纹理对象，OpenGL ES提供的方法原型如下：

```
void glGenTextures (GLsizei n, GLuint* textures)
```

这个方法传递进去的第一个参数是需要创建几个纹理对象，并且把创建好的纹理对象的句柄放到第二个参数中去，所以第二个参数是一个数组（指针）的形式。如果只需要创建一个纹理对象的话，则只需要声明一个GLuint类型的texId，然后针对该纹理ID取地址，并将其作为第二个参数，就可以创建出这个纹理对象了，代码如下：

```
glGenTextures(1, &texId);
```

执行完这行代码之后，就会在显卡中创建出一个纹理对象，并且把该纹理对象的句柄返回给texId变量。紧接着开发者要操作该纹理对象，但是在OpenGL ES的操作过程中必须告诉OpenGL ES具体操作的是哪一个纹理对象，所以必须调用OpenGL ES提供的一个绑定纹理对象的方法，调用代码如下：

```
glBindTexture(GL_TEXTURE_2D, texId);
```

执行完毕上面这行代码之后，下面的操作就都是针对于texId这个纹理对象的了，最终对该纹理对象操作完毕之后，我们可以调用一次解绑定的代码：

```
glBindTexture(GL_TEXTURE_2D, 0);
```

这行代码执行完毕之后，代表开发者不会对texId纹理对象做任何操作了，所以上面这行代码只在最后的时候才调用。接下来就是最关键的部分，即如何将本地磁盘上的一个PNG的图片上传到显卡中的这个纹理对象上。在将图片上传到这个纹理上之前，首先应该要对这个纹理对象设置一些参数，具体参数有哪些？其实就是纹理的过滤方式，当纹理对象（可以理解为一张图片）被渲染到物体表面上的时候（实际上是OpenGL绘制管线将纹理的元素映射到OpenGL生成的片段上的时候），有可能要被放大或者缩小，而当其放大或者缩小的时候，具体应该如何确定每个像素是如何被填充的，就由开发者配置的纹理对象的纹理过滤器来指明。

magnification（放大）：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

minification（缩小）：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

一般在视频的渲染与处理的时候使用GL_LINEAR这种过滤方式，该过滤方式称为双线性过滤，可使用双线性插值平滑像素之间的过渡，OpenGL会使用四个邻接的纹理元素，并在它们之间用一个线性插值算法做插值，该过滤方式是最主要的过滤方式，当然OpenGL中还提供了另外几种过滤方式。常见的有GL_NEAREST，称为最邻近过滤，该方式将为每个片段选择最近的纹理元素，但是当其放大的时候会有很严重的锯齿效果（因为相当于将原始的直接放大，其实就是降采样），而当其缩小的时候，因为没有足够的片段来绘制所有的纹理单元（这个是真正的降采样），许多细节都会丢失；相较于这两种过滤方式，本书在使用纹理的过滤方式时都会选用双线性过滤的过滤方式（GL_LINEAR）；其实OpenGL还提供了另外一种技术，称为MIP贴图，但是这种技术会占用更多的内存，其优点是渲染也会更快。当缩小和放大到一定程度之后效果也比双线性过滤的方式更好，但是其对纹理的尺寸以及内存的占用是有一定限制的，不过，在视频的处理以及渲染的时候不需要放大或者缩小这么多倍，所以在进行视频的处理以及渲染的场景下，MIP贴图并不适用。

紧接着来看一下对于纹理对象的另外一个设置，也就是在纹理坐标系的s轴和t轴的纹理映射过程中用到的重复映射或者约简映射的规则，代码如下：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

上述代码所表示的含义是，将该纹理的s轴和t轴的坐标设置为GL_CLAMP_TO_EDGE类型，因为纹理坐标可以超出（0，1）的范围，而按照上述设置规则，所有大于1的纹理值都要设置为1，所有小于0的值都要置为0。

接下来，就是将PNG素材的内容放到该纹理对象上，OpenGL的大部分纹理一般都只接受RGBA类型的数据（否则还得去做转化，后续会讲到YUV420P格式的视频帧在显卡中是如何转换为RGBA格式的），所以我们需要对PNG这种压缩格式进行解码操作，如果想要采用一种更通用的方式，那么可以引用libpng库来进行解码操作，当然也可以使用各自平台的API进行解码，最终可以得到RGBA的数据。待得到RGBA的数

据之后，记为uint8_t数组类型的pixels，然后执行如下操作：

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
        GL_UNSIGNED_BYTE, pixels);
```

这样就可以将该RGBA的数组表示的像素内容上传到显卡里面texId所代表的纹理对象中去了，以后只要使用该纹理对象，其实表示的就是这个PNG图片。

OpenGL中的纹理表示如何为物体增加细节，现在我们已经准备好了该纹理，那么如何把这张图片（或者说这个纹理）绘制到屏幕上呢？首先来看一下OpenGL中的物体坐标系，如图4-8所示，物体坐标系中x轴从左到右是从-1到1变化的，y轴从下到上是从-1到1变化的，物体的中心点恰好是（0，0）的位置。

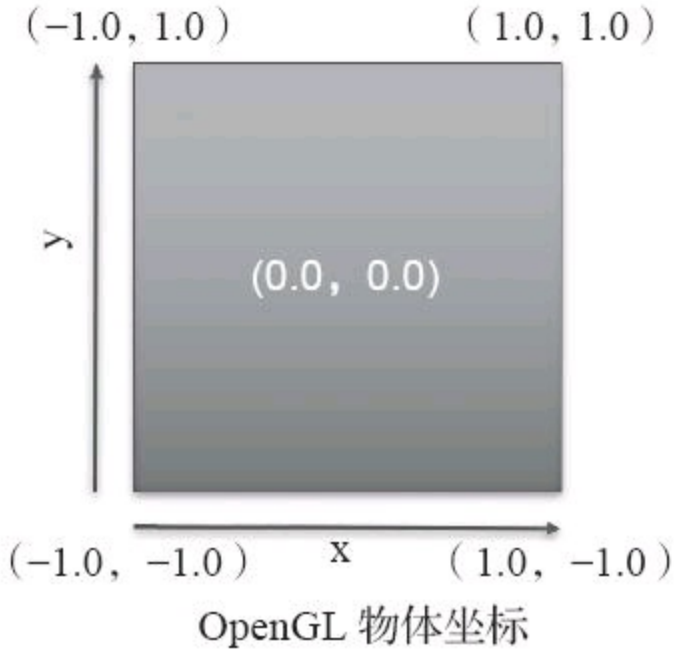


图　4-8

接下来的任务就是如何将这个纹理绘制到屏幕上，其实相关的基础知识已经都讲解过了，首先是搭建好各自平台的OpenGL ES的环境（包括上下文与窗口管理），然后创建显卡可执行程序，书写Vertex Shader，代码如下：

```
static char* COMMON_VERTEX_SHADER =
        "attribute vec4 position;                \n"
        "attribute vec2 texcoord;                \n"
        "varying vec2 v_texcoord;                \n"
        "                                        \n"
        "void main(void)                         \n"
        "{                                       \n"
        "   gl_Position = position;              \n"
        "   v_texcoord = texcoord;               \n"
        "}                                       \n";
```

在客户端代码中，开发者要从VertexShader中读取出两个attribute，并放置到全局变量的mGLVertexCoords与mGLTextureCoords中，接下来是Fragment Shader的内容，代码如下所示：

```
static char* COMMON_FRAG_SHADER =
        "precision highp float;                                  \n"
        "varying highp vec2 v_texcoord;                          \n"
        "uniform sampler2D texSampler;                           \n"
        "                                                        \n"
        "void main() {                                           \n"
        "    gl_FragColor = texture2D(texSampler, v_texcoord);   \n"
        "}                                                       \n";
```

从FragmentShader中读取出来的uniform会放置到mGLUniformTexture变量里，利用上面两个Shader创建好的Program，称为mGLProgId。紧接着进行真正的绘制操作，下面将详细地讲解一下绘制部分。

1）规定窗口的大小：

```
glViewport(0, 0, screenWidth, screenHeight);
```

假定screenWidth表示绘制区域的宽度，screenHeight表示绘制区域的高度。

2）使用显卡绘制程序：

```
glUseProgram(mGLProgId);
```

3）设置物体坐标：

```
GLfloat vertices[] = { -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f };
glVertexAttribPointer(mGLVertexCoords, 2, GL_FLOAT, 0, 0, vertices);
glEnableVertexAttribArray(mGLVertexCoords);
```

4）设置纹理坐标：

```
GLfloat texCoords1[] = { 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f };
GLfloat texCoords2[] = { 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f };
glVertexAttribPointer(mGLTextureCoords, 2, GL_FLOAT, 0, 0,  texCoords2);
glEnableVertexAttribArray(mGLTextureCoords);
```

这里需要注意的是texCoords2这个纹理坐标，因为其纹理对象是将一个PNG图片的RGBA格式的形式上传到显卡上（即计算机坐标），如果该纹理对象是OpenGL中的一个普通纹理对象，则需要使用texCoords1，这两个纹理坐标恰恰就是要做一个上下的翻转，从而将计算机坐标系和OpenGL坐标系进行转换。

5）指定将要绘制的纹理对象并且传递给对应的FragmentShader：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);
glUniform1i(mGLUniformTexture, 0);
```

6）执行绘制操作：

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

至此就可以在绘制区域（屏幕）绘制出最初的PNG图片了。

如果该纹理对象不再使用了，则需要将其删除掉，需要执行的代码是：

```
glDeleteTextures(1, &texId);
```

当然，只有在最终不再使用这个纹理的时候才会调用上述这个方法，如果不调用该方法则会造成显存的泄漏。

具体的实例请读者参考代码仓库中的OpenGLRenderer项目，注意，Android项目首先需要把resource目录下的PNG图片放到运行设备的sdcard的根目录之下。

4.4 本章小结

本章主要介绍了移动端音视频的渲染，音频使用对应平台提供的API进行渲染，其中在iOS平台使用AudioUnit进行渲染音频，在Android平台分别讲解了使用AudioTrack以及OpenSL ES来渲染音频。视频使用跨平台的OpenGL ES方案进行渲染，但是OpenGL ES需要各自平台提供对应的窗口管理与上下文环境，所以也讲解了Android与iOS平台对于OpenGL ES的窗口管理与上下文环境的创建。最后还讲解了OpenGL ES中的纹理，理解纹理的概念以及用法是非常重要的，因为在本书后边所有与OpenGL ES相关的处理都与纹理息息相关。本章作为移动端音视频领域开发的基础部分，请读者结合源码仓库中的实例加深理解。

第5章　实现一款视频播放器

前3章讨论了许多音视频相关的知识，包括音视频的基本概念，如何搭建移动平台下的开发环境，并学习了FFmpeg以及使用FFmpeg解码的方法，第4章学习了如何将音视频的裸数据渲染到硬件设备上。所以现在我们完全可以做一个实际的大项目——视频播放器，该项目能够把之前所学的知识串联起来，并且还可以学习到多线程控制、音视频同步等一些知识，那么实现一款视频播放器具体需要开发者做哪些工作呢，本章将带领大家逐步来实现。

5.1 架构设计

首先来看一下播放器需要为用户提供哪些功能：能够从零开始播放（当然要保证音画对齐）；支持暂停和继续播放功能；支持seek功能（就是可以随意拖动到任意位置并仍然可以继续播放），有的播放器还支持快进、快退15s。下面先来实现最基本的功能，即播放器能够从零开始播放、暂停和继续。

首先来思考一下要实现的场景，播放器可以从零开始播放直到结束。如果直接抛出这样一个项目，我们很容易找不到任何头绪，但是作为一个开发人员，要做的事情就是把复杂的问题简单化，简单的问题条理化，最终按照拆分得非常细的模块来逐个实现。基于这个项目，我们需要思考以下几个问题：

·输入是什么？

·输出是什么？

·可以划分为几个模块？

·每个模块的职责是什么？

下面对问题逐个进行梳理，首先要搞清楚输入是什么，输入既可以是本地磁盘上的一个媒体文件（可能是FLV、MP4、AVI、MOV等格式的文件），也可以是网络上的一个媒体文件（可能是HTTP、RTMP、HLS等协议），这就是我们确定的输入；那么输出又是什么呢？输出就是让扬声器播放视频中的音频使用户的耳朵可以听到声音，让屏幕显示视频画面使用户的眼睛可以看到画面，同时，听到的声音和看到的画面必须是同步的（也就是说不能让用户听到的是“你好”的发音，看到的却是“吃了”的画面）；然后再根据输入和输出拆分模块，并为模块分配合理的职责。

对于输入部分的分析具体如下，输入有可能是不同的协议，比如说file（本地磁盘的文件），或者是HTTP、RTMP、HLS协议等；也有可能是不同的封装格式，比如说MP4、FLV、MOV等封装格式；而对于这些封装格式里面的内容，会有两路流，分别是音频流和视频流，我们需要将这两路流都解码为裸数据。待视频流和音频流都解码为裸数据之

后，需要为音视频各自建立一个队列将裸数据存储起来，不过，如果是在需要播放一帧的时候再去做解码，那么这一帧的视频就有可能产生卡顿或者延迟，所以这里引出了第一个线程，即为播放器的后台解码分配一个线程，该线程用于解析协议，处理解封装以及解码，并最终将裸数据放到音频和视频的队列中，这个模块称为输入模块。

下面再来看输出部分，输出部分其实是由两部分组成的：一部分是音频的输出，另一部分是视频的输出。不过可以确定的是，不论是音频的输出还是视频的输出，它们都会用一个线程来进行管理，这两个模块应该先从队列中获取音视频的裸数据，然后分别进行音视频的渲染，并最终发布到扬声器和屏幕上，使得用户可以听得到、看得到，这两个模块称为音频输出和视频输出模块。

下面再来思考一件事情，由于输出模块都在各自的线程中，音频和视频均是单独播放，这就导致了两个输出模块的播放频率以及线程控制没有任何关系，从而无法保证音画对齐。我们规划的各个模块里，好像还没有一个模块的职责是负责音视频同步的，所以需要再建立一个模块来负责相关的工作，这个模块称为音视频同步模块。

至此，模块都已拆分完毕，具体的模块分布如图5-1所示。

不过，我们还应该再写一个调度器，将这几个模块组装起来。也就是说，先把输入模块、音频队列、视频队列都封装到音视频同步模块中，然后为外界提供获取音频数据、视频数据的接口，这两个接口必须保证音视频的同步，内部将负责解码线程的运行与暂停的维护。然后把音视频同步模块、音频输出模块、视频输出模块都封装到调度器中，调度器模块会分别向音频输出模块和视频输出模块注册回调函数，回调函数允许两个输出模块获取音频数据和视频数据。这样就可以对类图设计做进一步整理了，如图5-2所示。

图5-2详细的解释具体如下。

·VideoPlayerController：调度器，内部维护音视频同步模块、音频输出模块、视频输出模块，为客户端代码提供开始播放、暂停、继续播放、停止播放接口；为音频输出模块和视频输出模块提供两个获取数据的接口。

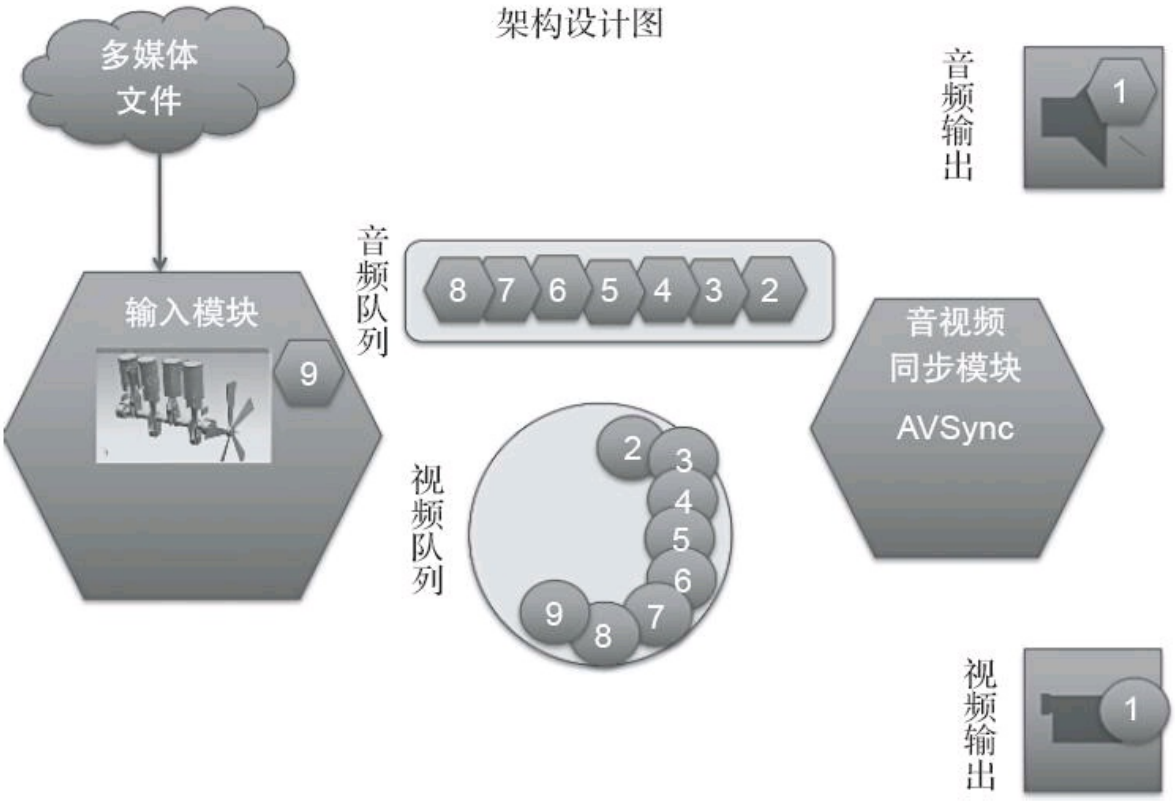

架构设计图
多媒体
文件
输入模块
9
音频队列
8 7 6 5 4 3 2
音频输出
1
音视频
同步模块
AVSync
视频队列
2 3 4 5 6 7 8 9
视频输出
1

图　5-1

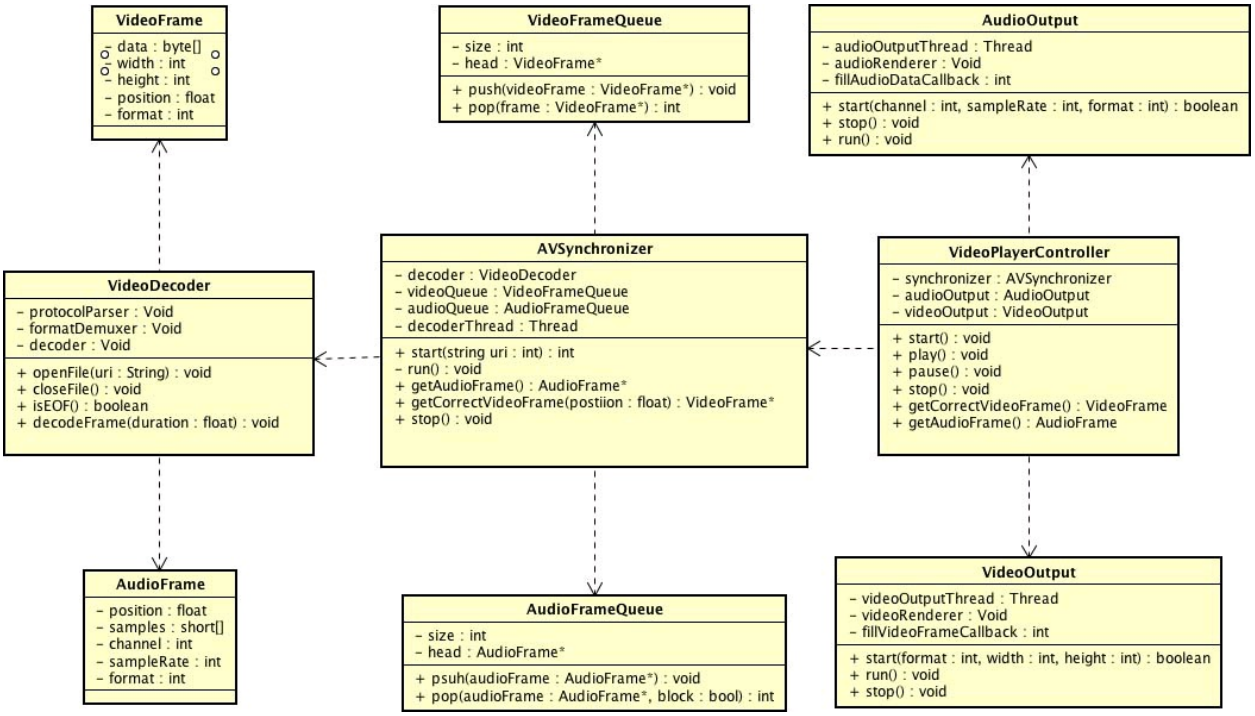

VideoFrame
- data : byte[]
- width : int
- height : int
- position : float
- format : int
VideoFrameQueue
- size : int
- head : VideoFrame*
+ push(videoFrame : VideoFrame*) : void
+ pop(frame : VideoFrame*) : int
AudioOutput
- audioOutputThread : Thread
- audioRenderer : Void
- fillAudioDataCallback : int
+ start(channel : int, sampleRate : int, format : int) : boolean
+ stop() : void
+ run() : void
VideoDecoder
- protocolParser : Void
- formatDemuxer : Void
- decoder : Void
+ openFile(uri : String) : void
+ closeFile() : void
+ isEOF() : boolean
+ decodeFrame(duration : float) : void
AVSynchronizer
- decoder : VideoDecoder
- videoQueue : VideoFrameQueue
- audioQueue : AudioFrameQueue
- decoderThread : Thread
+ start(string uri : int) : int
- run() : void
+ getAudioFrame() : AudioFrame*
+ getCorrectVideoFrame(postiion : float) : VideoFrame*
+ stop() : void
VideoPlayerController
- synchronizer : AVSynchronizer
- audioOutput : AudioOutput
- videoOutput : VideoOutput
+ start() : void
+ play() : void
+ pause() : void
+ stop() : void
+ getCorrectVideoFrame() : VideoFrame
+ getAudioFrame() : AudioFrame
AudioFrame
- position : float
- samples : short[]
- channel : int
- sampleRate : int
- format : int
AudioFrameQueue
- size : int
- head : AudioFrame*
+ psuh(audioFrame : AudioFrame*) : void
+ pop(audioFrame : AudioFrame*, block : bool) : int
VideoOutput
- videoOutputThread : Thread
- videoRenderer : Void
- fillVideoFrameCallback : int
+ start(format : int, width : int, height : int) : boolean
+ run() : void
+ stop() : void

图　5-2

·AudioOutput：音频输出模块，由于在不同平台上有不同的实现，所以这里真正的声音渲染API为Void类型，但是音频的渲染要放在一个单独的线程（不论是平台API自动提供的线程，还是我们主动建立的线程）中进行，所以这里有一个线程的变量，在运行过程中会调用注册过来的回调函数来获取音频数据。

·VideoOutput：视频输出模块，虽然这里统一使用OpenGL ES来渲染视频，但是前文已提到过，OpenGL ES的具体实现在不同的平台上也会有自己的上下文环境，所以这里采用了Void类型的实现，当然，必须由我们主动开启一个线程来作为OpenGL ES的渲染线程，它会在运行过程中调用注册过来的回调函数来获取视频数据。

·AVSynchronizer：音视频同步模块，会组合后文将要讲到的输入模块、音频队列和视频队列，其主要为它的客户端代码VideoPlayerController调度器提供接口，包括：开始、结束，以及最重要的获取音频数据和获取对应时间戳的视频帧。此外，它还会维护一个解码线程，并且根据音视频队列里面的元素数目来继续或者暂停该解码线程的运行。

·AudioFrame：音频帧，其中记录了音频的数据格式以及这一帧的具体数据、时间戳等信息。

·AudioFrameQueue：音频队列，主要用于存储音频帧，为它的客户端代码音视频同步模块提供压入和弹出操作，由于解码线程和声音播放线程会作为生产者和消费者同时访问该队列中的元素，所以该队列要保证线程安全性。

·VideoFrame：视频帧，记录了视频的格式以及这一帧的具体的数据、宽、高以及时间戳等信息。

·VideoFrameQueue：视频队列，主要用于存储视频帧，为它的客户端代码音视频同步模块提供压入和弹出操作，由于解码线程和视频播放线程会作为生产者和消费者同时访问该队列中的元素，所以该队列要保证线程安全性。

·VideoDecoder：输入模块，其职责在前面已经分析过了，由于还没有确定具体的技术实现，所以这里先根据前面的分析暂时写了三个实例变量，一个是协议层解析器，一个是格式解封装器，一个是解码器，并

且它主要向AVSynchronizer提供接口：打开文件资源（网络或者本地）、关闭文件资源、解码出一定时间长度的音视频帧。

至此我们根据用户场景（Case）把视频播放器拆解成了各个模块，并且根据模块的调用关系画出了类图，那么接下来要做的事情就应该是拆分每个模块的具体实现。

首先是输入模块，如果是自己编写代码处理这些不同的协议、封装格式，以及编解码格式（更专业地来讲是各种解码器），肯定会是非常复杂也极其不合理的，因为这套东西已经非常成熟了，自行实现需要付出很大的开发与测试成本，并且最终效果也不会太理想。而基于我们现在所掌握的知识，可选择FFmpeg开源库的libavformat模块来处理各种不同的协议以及不同的封装格式。在解封装成每一路流之后，接下来就需要进行解码的操作，当然，最简单的也可以直接使用FFmpeg的libavcodec模块来进行，但是如果需要更高性能的解码，那么开发者可以使用Android和iOS平台各自的硬件解码器。本章暂时先不考虑优化，只是先快速实现出一套方案，使用软件解码会是一种比较好的选择，所以本章将使用FFmpeg的libavcodec模块作为解码器模块的技术选型。

其实对于架构来说，没有最好的设计，只有最合适的设计。在这里，硬件解码器对于系统平台是有限制的，同时还会有一些兼容性问题，并且两个平台还需要分别编写代码来实现各自的硬件解码器，并将硬件解码器解码出来的数据转换为可用于显示的视频帧数据结构。因为本章需要快速实现各个模块，所以这里选择使用软件解码器，同时它有更高的兼容性以及更简单的API调用。以后很可能需要通过硬件解码来提升性能，所以在设计解码模块的时候可以更多地使用面向接口的设计，以便日后更加方便地替换实现。

其次是音频输出模块，音频的输出其实有很多种方式，下面就来分别分析一下，首先是Android平台，最常用的就是Java层的AudioTrack和Native层的OpenSL ES。主要代码处于Native层，在AudioTrack和OpenSL ES之间，应该选择OpenSL ES，因为其省去了JNI的数据传递，并且OpenSL ES在播放声音方面的延迟更低，其缺点是所提供的API比起AudioTrack的不够友好，调试也不太方便，但是从总体来分析，还是选择OpenSL ES更合适；对于iOS平台，其实也有很多种方式，比较常见的就是AudioQueue和AudioUnit，AudioQueue是更高层次的音频API，是建立在AudioUnit的基础之上的，其所提供的API更加简单，在这里其实选用AudioQueue可能会更加合适，但是我们最终还是会选用

AudioUnit，对此，有如下几个原因：首先可能存在音频格式的转换，这时AudioUnit会更加方便，并且这里还需要为后续的录音、音效处理打下使用AudioUnit的基础，所以这里将直接选择AudioUnit作为实现。

然后是视频输出模块，对此，技术选型肯定是选择OpenGL ES，因为我们可以利用它非常高效率的渲染视频，不论是在Android平台还是在iOS平台，前面也已经学习了如何在Android平台和iOS平台搭建OpenGL ES的环境。此外，在这里使用OpenGL ES还有一个好处，那就是我们可以利用OpenGL ES处理图像的巨大优势，来对视频做一个后期处理（通过去块滤波器、增加对比度等效果器的使用），让用户感觉视频更加清晰。在Android平台上使用EGL来为OpenGL ES提供上下文环境，使用SurfaceView的Surface来构造显示对象，并最终输出到SurfaceView上；在iOS平台上使用EAGL来为OpenGL ES提供上下文环境，自己定义一个View继承自UIView，使用EAGLLayer作为渲染对象，并最终渲染到这个自定义的View上。

至于音视频同步模块，这里不会涉及任何与平台相关的API，不过，考虑到它要维护解码线程，因此pthread其实是一个很好的选择，因为两个平台都支持这种线程模型。此外，它还需要维护两个队列，由于STL中提供的队列不能保证线程的安全性，所以对于音视频队列，我们可以自行编写一个保证线程安全的链表来实现。最后要负责音视频的同步，由于音视频同步的策略在前面的章节中已经提到过，因此这里采用视频向音频对齐的策略，即只需要把同步这块逻辑放到获取视频帧的方法里面就好了。

最后是控制器，控制器需要将上述的三个模块合理地组装起来。在开始播放的时候，需要把资源的地址（有可能是本地的文件，也有可能是网络的资源文件）传递给AVSynchronizer，如果能够成功地打开文件，那么就去实例化VideoOutput和AudioOutput，在实例化这两个类的同时，要传入回调函数，这两个回调函数又将分别调用AVSynchronizer里面的获取音频和视频帧的方法，这样就可以有序地组织多个模块，最终如果要调用暂停、继续或停止的指令，自然也就会调用各个模块对应的生命周期方法。

前文中笔者将每个模块的具体实现梳理了一遍，这样，其实架构已经基本成型了，但是对于架构师来说，做到这些还是不够的，因为优秀的架构师必须在做完整个架构之后，再针对该架构给出风险评估与部分测试用例，下面也将来逐一分析一下。

首先是风险评估，由于我们所做的最终项目是运行在移动平台上的，所以对于移动平台的碎片化设备（尤其是Android平台的碎片化更加严重）这一特点，必须要有足够多的设备作为测试目标，以保证没有兼容性方面的问题，设备所述的平台架构也应该覆盖到arm、armv7、arm64等平台。然后必须要测试性能问题，性能包括CPU消耗、内存占用、耗电量与发热量，而针对这些风险，在这一期项目中可能会有一些问题无法得到解决，因此我们的长期计划就应该在这些方面进行改进。其实，目前来说最大的风险就是软件解码这部分，因此从长期来看，需要有硬件解码的替代方案。

对于测试用例，我们应该从以下几个方面进行测试，首先是输入模块，包括协议层（网络资源、本地资源）、封装格式（FLV、MP4、MOV、AVI等）、编码格式（H264、AAC、WAV）等；其次是音视频同步模块，应该在低网速的条件下观看网络资源的对齐程度；最后是两个输出模块，测试应该要覆盖iOS系统和Android系统的大部分系统版本，以及最终应用运行的Top10的所有设备的音频和视频播放的兼容性。

完成了风险评估和基本的测试用例之后，至此我们的架构算是比较完善了，接下来会逐一实现每个模块。

5.2 解码模块的实现

本节先来介绍输入模块的具体实现，即类图（见图5-2）中的VideoDecoder类的实现，前面在讨论技术定型的时候已经说过，我们会直接使用FFmpeg开源库来负责输入模块的协议解析、封装格式拆分、解码操作等行为，整体流程如图5-3所示。

首先，来看一下整体的运行流程，整个运行流程分为以下几个阶段：

1）建立连接、准备资源阶段。

2）不断读取数据进行解封装、解码、处理数据阶段。

3）释放资源阶段。

以上就是输入端的整体流程，其中第二个阶段会是一个循环，并且放在单独的线程中来运行，由于具体的API调用已经在前面章节中做过详细的介绍，并且在本章的代码仓库中也可以找到对应的源码，因此这里就不再罗列源码了，而是具体来看一下这个类中几个重要的接口是如何设计与实现的。

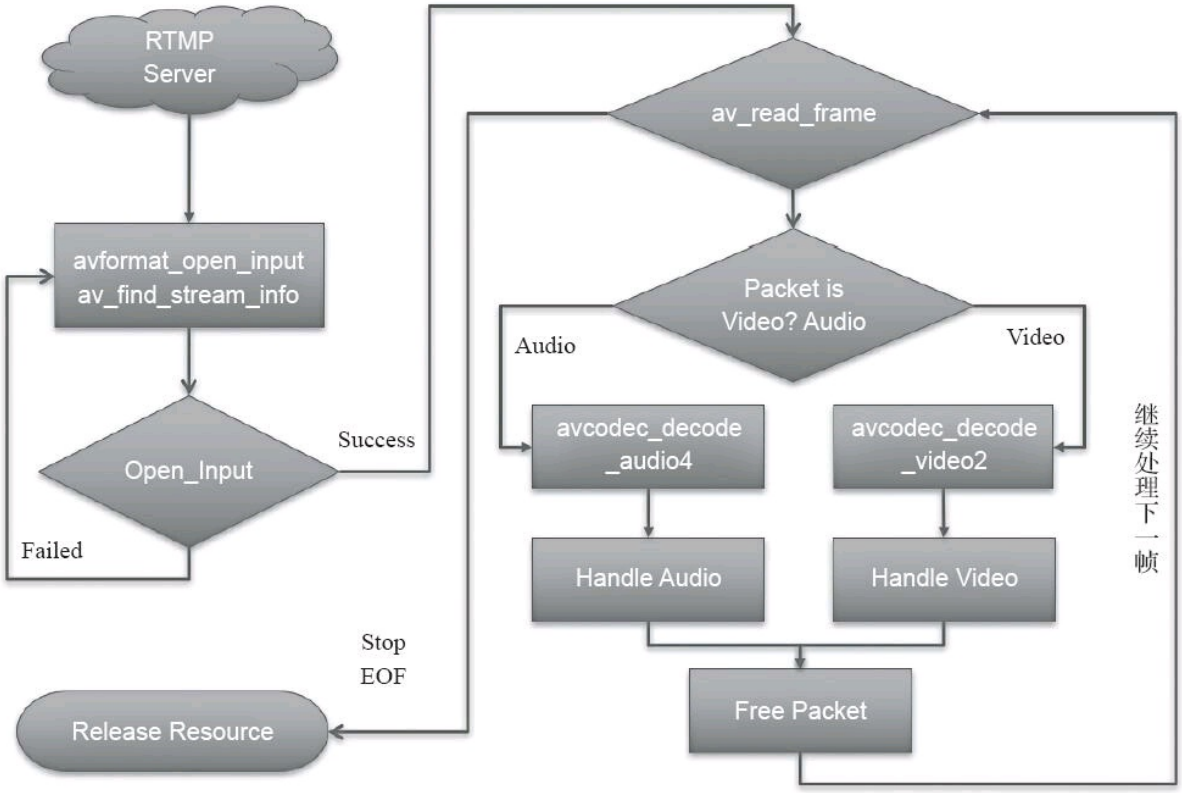


图　5-3

先来看openFile接口的具体实现，该接口主要负责建立与媒体资源的连接通道，并且分配一些全局需要用到的资源，将建立连接的通道与分配资源的结果返回到调用端。在此过程中，首先是与媒体资源建立连接通道，然后找出该资源所包含的流的信息（其实是对应的各个Stream的MetaData，比如声音轨的声道数、采样率、表示格式或者视频轨的宽、高、fps等）。如果是网络资源，那么在找出流信息失败的时候可以进行重试（具体的重试逻辑可以根据不同的业务场景进行设置，在我们的代码中设置的是重试3次的策略）。在找出流信息的这一阶段需要使用到前面建立起来的连接通道，并且FFmpeg提供的找出流信息（av_find_stream_info）的API其实是可以设置参数来控制方法的执行时间的。对于本地资源，该过程寻找MetaData是很快的，不过，如果是网络资源，那就需要一段时间了。第3章中已经介绍过find_stream_info函数的内部实现，因为会发生实际的解码行为，所以解码的数据越多，所花费的时间也会越长，对应得到的MetaData也会越准确，对此，一般是通过设置probesize和max_analyze_duration这两个参数来给出探测数据量的大小和最大的解析数据的长度，其值常设置为50×1024和75000，如果达到了设置的值却还没有解析出对应的视频流和音频流的MetaData，那

么就返回失败，紧接着会进入重试策略，如果可以解析到就将对应的流信息填充到对应的结构体中。在找出对应的流信息之后，接下来就是要打开每个流的解码器（有时声音有两路流，有的播放器中可以允许切换，就像我们之前所说的ffplay就支持带入参数进行选择，VLC播放器可以实时切换，本项目中只选择第一个音频流）。最后，对于每个流都要分配一个AVFrame作为解码之后数据存放的结构体，对于音频流，需要额外分配一个重采样的上下文，对解码之后的音频格式进行重采样，使其成为我们需要的PCM格式，这里仅进行资源分配，具体的解码和转换行为后续再讲。

其次是decodeFrames接口的实现，该接口主要负责解码音视频压缩数据成为原始格式，并且封装成为自定义的结构体，最终全部放到一个数组中，然后返回给调用端。首先需要读取一个压缩数据帧，对应于FFmpeg里面的AVPacket结构体，对此，前文中也有过详细的介绍；对于视频帧，一个AVPacket就是一帧视频帧；对于音频帧，一个AVPacket有可能包含多个音频帧，所以对于一个AVPacket判定了类型（音频类型还是视频类型）之后，其所调用的解码方法是不一样的，视频部分仅需要解码一次，而音频部分则需要判定该AVPacket里面的压缩数据是否已被全部消耗干净了，并以此作为结束的条件。解码结束之后，需要提取出对应的裸数据填充到我们自定义的结构体，并全部存入数组中，然后返回给外界调用端，为什么要这么做呢？因为我们不希望向外界暴露Input模块内部所使用的技术细节，即不希望向客户端代码暴露其中使用的到底是FFmpeg的解码器库还是硬件解码器或者是其他的解码器等细节。所以解码之后，需要封装成自定义的结构体的AudioFrame和VideoFrame，具体如何操作，前面章节中也已经提到过，或者直接查看源码也可以了解。

这里有一点比较特殊的是，如果音频或者视频解码出来的表示方式与我们预期的表示方式不一样，那么就需要做个转换。对于音频和视频的格式转换，FFmpeg分别提供了不同的API来完成转换操作，具体如下所示。

·对于音频的格式转换，FFmpeg提供了一个libswresample库，开发者只需要把原始音频的格式（包括声道、采样率、表示格式）和目标音频的格式（包括声道、采样率、表示格式）传递给这个库的初始化上下文方法（swr_alloc_set_opts），就可以构造出一个重采样上下文，然后调用swr_convert将解码器输出的AVFrame传递进来，那么重采样之后的

数据就是开发者所期望的音频格式的数据了，最终使用完毕之后再调用swr_free方法即可释放掉重采样上下文。

·对于视频帧的格式转换，FFmpeg提供了一个libswscale的库，用于转换视频的裸数据的表示格式，如果原始视频的裸数据其表示格式不是YUV420P，那么就需要使用该库来将非YUV420P格式的视频数据转换为YUV420P格式。转换方式也很简单，就是把源的格式（包括视频宽、高、表示格式）和目标的格式（包括视频宽、高、表示格式）传递给该库的获取上下文方法（sws_getCachedContext），以构造出转换视频的上下文，然后在需要使用的时候调用sws_scale将解码器输出的AVFrame传递进来，那么转换之后的视频数据就存在于AVPicture结构体里面了，最终我们可以从AVPicture里面再提取出对应的数据封装到自定义的结构体中，使用完毕之后调用sws_freeContext来销毁掉该转换上下文。

销毁资源阶段的实现，与打开流阶段恰恰相反，首先要销毁掉音频相关的资源，包括分配的AVFrame以及音频解码器（如果分配了重采样上下文与重采样的buffer，那么也需要销毁掉）；然后再销毁掉视频的相关资源，包括分配的AVFrame与视频解码器（如果分配了格式转换上下文与转换后的AVPicture，那么也需要销毁掉）；最后断开连接通道，最终所有的资源都销毁掉了。

超时设置

在FFmpeg的API中，有一个判断超时的设置，如果资源是网络上的媒体文件，那么该设置将会非常有用，首先要在建立连接通道之前为AVFormatContext类型的结构体的interrupt_callback变量赋值即设置回调函数，接下来FFmpeg将在以后需要用到该连接通道读取数据的时候（寻找流信息阶段、实际的read_frame阶段）由另外一个线程调用开发者设置的这个回调函数，询问是否达到超时的条件，如果返回1则代表超时，FFmpeg会主动断开该连接通道，返回0则代表不超时，FFmpeg则不进行其他的任何处理。所以如果网络情况不好的话，在我们关闭资源的时候就有可能会出现阻塞很长时间的情况，所以开发者可以在超时回调中直接返回1，并且可以很快关闭掉连接通道，然后释放掉整个资源了。

5.3 音频播放模块的实现

本节就来介绍音频播放模块的实现，即类图（图5-2）中的AudioOutput类的实现，这一部分的实现对于Android和iOS平台其实是不同的，第4章中已经详细介绍了OpenSL ES和AudioUnit的使用，这里面将会结合现在的这个项目再来具体地调整一下其实现结构。

5.3.1 Android平台的音频渲染

Android平台上使用OpenSL ES进行音频的渲染，首先要建立AudioOutput类，按照之前的架构设计，需要在该类中定义一个回调函数，让外界来实现该函数，用来为此模块填充所需要播放的音频PCM数据，所以该回调函数定义如下：

```
typedef int(*audioPlayerCallback)(byte* , size_t, void* ctx);
```

该函数的第一个参数需要外界填充PCM数据的缓冲区，第二个参数是该缓冲区的大小，第三个参数是客户端代码自己填充的上下文对象（由于C++中的回调函数是静态的函数，所以要传递对象自身作为该上下文对象，以便被调用的时候可以将该上下文对象强制转换成为目标对象，来访问对象中的属性以及方法）。

接下来，就来具体实现AudioOutput类中的几个接口方法，其实面向对象的特征之一就是封装，即将类内部的具体实现细节进行封装，对外提供接口，用来完成客户端代码想要该类完成的所有行为。所以这几个接口肯定不需要暴露AudioOutput内部到底是使用OpenSL ES来实现的音频播放还是AudioTrack来实现的播放，我们现在使用OpenSL ES实现音频播放，如果以后还有一些特殊的需求，那么可以换成AudioTrack或者其他的实现方式，但是对于外界的接口以及回调函数是不会改变的。这对于整个系统的扩展性以及维护性都是非常重要的，其实前面VideoDecoder不向客户端代码暴露AVFrame，而是暴露自己封装的VideoFrame或者AudioFrame结构体，道理是一样的。那么下面就来实现第一个接口。

（1）初始化方法

传入的参数就是声道数、采样率、表示格式、回调函数以及回调函数的上下文对象，返回值就是OpenSL ES是否可以正常完成初始化，对于具体的OpenSL ES是如何初始化的此处将不再赘述，因为第4章中已经花费了很大的篇幅来讲解这点。核心流程中有一步是为audioPlayerBufferQueue设置回调函数，也就是说，当OpenSL ES需要数据进行播放的时候会回调该函数，由开发者来填充PCM数据，而此时我

们在第一步中定义的回调函数就有用了，此处可调用该回调函数来填充音频裸数据，然后调用audioPlayerBufferQueue的Enqueue方法，把客户端代码填充过来的PCM数据放到OpenSL ES的BufferQueue中去。

（2）暂停和继续播放方法

上面的步骤在初始化OpenSL ES的时候，已经获得了audioPlayerObject中的play接口，这里只需要设置playState就可以了，代码如下：

```
int state = play ? SL_PLAYSTATE_PLAYING : SL_PLAYSTATE_PAUSED;
(*audioPlayerPlay)->SetPlayState(audioPlayerPlay, state);
```

（3）停止方法

首先应暂停现在的播放，然后最重要的一步就是设置一个全局的状态，以保证如果再有audioPlayerBufferQueue的回调函数需要调用的时候，不需要再进行数据填充，最好再调用usleep方法来暂停一段时间（比如50ms），以使得buffer缓冲区里面的数据全部播放完毕，最终再调用OpenSL ES的API销毁所有的资源，包括audioPlayerObject与outputMixObject。

5.3.2 iOS平台的音频渲染

在iOS平台，可使用AudioUnit（AUGraph封装的实际上就是AudioUnit）来渲染音频，类似于Android平台的实现，这里也要有一个类似于回调函数的形式来要求客户端代码填充音频的PCM数据。相比较于回调函数，OC中的常用实现是定义一个协议（Protocol），由客户端代码实现该协议，并重写协议里面定义的方法，下面就来看看这个Protocol的定义：

```
@protocol FillDataDelegate <NSObject>
    - (NSInteger) fillAudioData:(SInt16*) sampleBuffer
        numFrames:(NSInteger)frameNum numChannels:(NSInteger)channels;
@end
```

其中，第一个参数就是要填充的缓冲区，第二个参数是该缓冲区中有多少个音频帧，第三个参数是声道数。客户端代码在实现中要按照帧的个数和声道数来填充该缓冲区。其实OC中的这种Protocol方式会更加地面向对象，让开发者能够更加合理地实现代码，用客户端代码来实现这个协议就意味着需要承担该协议所要求的职责，比起C++的回调函数OC语言的这种写法会更加地面向对象，读者可以自行体会一下。

然后是该类的初始化方法，传入包括声道数（NSInteger channels）、采样率（NSInteger sampleRate）、采样的表示格式（NSInteger bytesPerSample），以及具体的所规定的协议实现的对象（id<FilleDelegate>fillAudioDelegate）。在该方法的实现中首先需要构造一个AVAudioSession，然后为该session设置用途类型以及采样率等；接下来需要设置音频被中断的监听器，以方便应用程序在特殊情况下也可以给出相应的处理；最后就是核心流程——构造AUGraph，用来实现音频播放，这个具体的流程已经在前面的章节中详细讲解过了，这里需要注意的是，应配置一个ConvertNode将客户端代码填充的SInt16格式的音频数据转换为RemoteIONode可以播放的Float32格式的音频数据（采样率、声道数以及表示格式应对应上），这一点是非常关键的，当然需要为ConvertNode配置上InputCallback，在InputCallback的实现中调用Delegate的fillAudioData方法，让客户端代码填充数据，配置好整个AudioGraph之后，调用AUGraphInitialize方法来初始化整个AUGraph。最终构造出来的AUGraph以及其与客户端代码的调用关系如图5-4所

示。

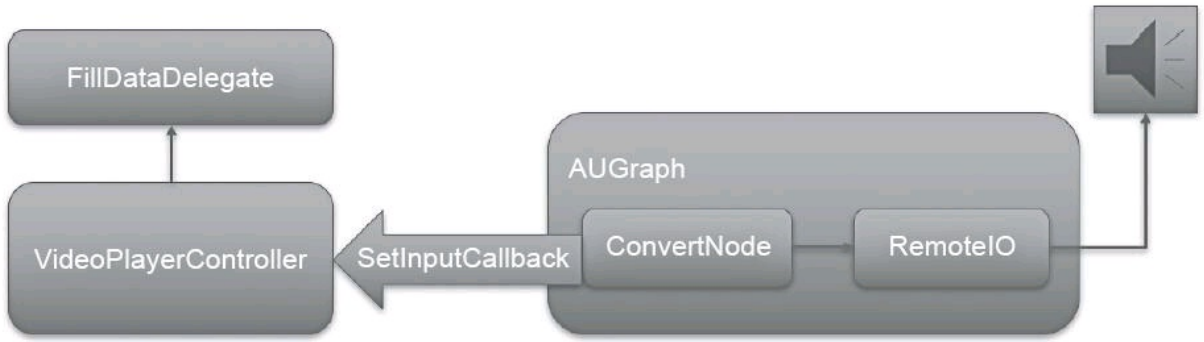


图　5-4

接下来是play方法，该方法的实现就非常简单了，直接调用AUGraphStart方法，启动该Graph就可以了，一旦启动了之后，就会从RemoteIO这个AudioUnit开始播放音频数据，如果需要音频数据，就向它的前一级AudioUnit即ConvertNode去获取数据，而ConvertNode则会寻找自己的InputCallback，在InputCallback的实现中其将从delagate（即VideoPlayerController）处获取数据，然后就向实现该Protocol的客户端代码中填充数据，最终就可以播放出来了。

至于pause方法，其实现就更简单了，直接调用AUGraphStop方法，这样就可以停止AUGraph的运行，声音自然就不会播放出来了。

最后是销毁方法，在销毁方法中需要停止AUGraph，然后调用AUGraphClose方法关闭AUGraph，并移除AUGraph里面所有的Node，最终调用DisposeAUGraph，这样就可以彻底销毁掉整个AUGraph了。

5.4 画面播放模块的实现

本节将介绍视频（画面）播放模块的实现，即类图（见图5-2）中的VideoOutput类的实现，这部分的实现也是依赖于平台的，虽然原理上使用的都是OpenGL ES，但是对于不同的平台其实现也是不同的，第4章中已经详细地讲解了具体的使用实例，下面就结合该项目再来调整一下具体结构。

5.4.1 Android平台的视频渲染

前面提到过，无论是在哪一个平台上使用OpenGL ES渲染视频的画面，都需要单独开辟一个线程，并且为该线程绑定一个OpenGL ES的上下文，Android平台肯定也不例外，由于是在Native层进行OpenGL ES的开发，所以这里首先选用线程模型，前面的章节中，笔者曾经分析过各种线程模型的优缺点，所以这里将直接选用POSIX线程模型即PThread。在开始实现初始化函数之前，需要先定义一个回调函数，当VideoOutput模块需要渲染视频帧的时候，就调用该回调函数获取需要渲染的视频帧，然后再进行真正的渲染，回调函数的代码原型如下：

```
typedef int (*getTextureCallback)(VideoFrame** texture, void* ctx);
```

该函数的参数列表中，第一个就是要获取的视频帧，第二个是回调函数的上下文，返回值为int类型（当成功获取到一帧视频帧之后返回大于零的值，否则返回负值）。接下来再来看一下初始化函数的实现。

对于初始化函数，传入的第一个参数是ANativeWindow类型的指针，该Window实际上是从Java层传递过来的一个Surface中构造出来的，而Java层的这个Surface就是从SurfaceView的onSurfaceCreated生命周期方法中的SurfaceHolder中获取出来的。第二个和第三个参数则是绘制的View的宽和高，第四个参数是获取视频帧的回调函数以及回调函数的上下文对象。初始化函数的实现如下，首先创建一个线程作为OpenGL ES的渲染线程，线程执行的第一个步骤就是初始化OpenGL ES环境，它会利用EGL构建出OpenGL ES的上下文，并且利用ANativeWindow构造出EGLDisplay作为显示目标，然后利用vertexShader和fragmentShader构造出一个Program。

VertexShader负责顶点的操作，其代码如下：

```
static char* OUTPUT_VIEW_VERTEX_SHADER =
        "attribute vec4 vPosition;                  \n"
        "attribute vec4 vTexCords;                  \n"
        "varying vec2 yuvTexCoords;                 \n"
        "                                           \n"
        "void main() {                              \n"
        "    yuvTexCoords = vTexCords.xy;           \n"
```

```
        "    gl_Position = vPosition;                \n"
        "}                                           \n";
```

VertexShader中直接将顶点赋值给gl_Position，然后将纹理坐标传递给FragmentShader，代码具体如下：

```
static char* YUV_FRAME_FRAGMENT_SHADER =
        "varying highp vec2 yuvTexCoords;                                  \
        "uniform sampler2D s_texture_y;                                    \
        "uniform sampler2D s_texture_u;                                    \
        "uniform sampler2D s_texture_v;                                    \
        "void main(void)                                                   \
        "{                                                                 \
        "    highp float y = texture2D(s_texture_y, yuvTexCoords).r;       \
        "    highp float u = texture2D(s_texture_u, yuvTexCoords).r - 0.5; \
        "    highp float v = texture2D(s_texture_v, yuvTexCoords).r - 0.5; \
        "                                                                  \
        "    highp float r = y + 1.402 * v;                                \
        "    highp float g = y - 0.344 * u - 0.714 * v;                    \
        "    highp float b = y + 1.772 * u;                                \
        "    gl_FragColor = vec4(r,g,b,1.0);                               \
        "}                                                                 \
```

由于视频帧是由YUV420P的数据格式表示的，所以在FragmentShader中需要把YUV格式的数据转换为RGBA格式的数据。首先获取对应的YUV格式的数据，因为UV的默认值是127，所以我们这里要减去0.5（OpenGL ES的Shader中会把内存中0～255的整数数值换算为0.0～1.0的浮点数值），然后按照YUV到RGBA的计算公式将YUV的格式转换为RGBA的格式，而这就是FragmentShader要完成事情。

然后是渲染方法，当客户端代码需要VideoOutput来渲染视频帧的时候，VideoOutput模块将会利用回调函数先获得视频帧，然后利用第一步创建的线程中构造的Program执行渲染操作，最终调用eglSwapBuffers方法将渲染的内容绘制到EGLSurface中去（EGLSurface内部是ANativeWindow，ANativeWindow内部又组合了Surface，而Surface代表的就是SurfaceView，所以最终就是绘制到Java层的SurfaceView中去）。

最后是销毁方法，也必须在第一步创建的线程中进行，因为在该线程中创建了OpenGL上下文、EGLDisplay、Program、渲染过程中使用到的纹理对象、FrameBuffer对象等OpenGL ES的对象，所以必须在该线程中销毁这一系列的对象。

5.4.2 iOS平台的视频渲染

前面章节中曾介绍过如何在iOS平台上使用OpenGL ES，在本节的实现中，首先会书写一个VideoOutput类继承自UIView，然后重写父类的layerClass方法，并且返回CAEAGLLayer类型，重写该方法的目的是该UIView可以被OpenGL ES进行渲染；然后在初始化方法中，将OpenGL ES绑定到Layer上，iOS平台上的线程模型，采用NSOperationQueue来实现，也就是把OpenGL ES的所有操作都封装在NSOperationQueue中来完成，为什么要使用这种线程模型呢？其实笔者在很多设备中都做过测试，由于某些低端设备，比如iPod、iPhone 4，在一次OpenGL的绘制中耗费的时间可能会比较多，如果使用的是GCD的线程模型，那么会导致DispatchQueue里面的绘制操作越积累越多，并且不能清空；而使用NSOperationQueue，则可以在检测到Queue里面的Operation超过定义的阈值（Threshold）时，清空最久的Operation，只保留最新的绘制操作，这样才能完成正常的播放。前面的章节中也曾提到过，iOS平台有一个比较特殊的地方就是如果App进入后台之后，就不能再进行OpenGL ES的渲染操作，所以这里还需要注册两个监听事件：一个是WillResignActiveNotification，即当App从活跃状态变为非活跃状态的时候，或者即将进入后台的时候，系统会调用该监听事件；另外一个是DidBecomeActiveNotification，即当App从后台到前台时系统会调用该监听事件。我们可以在这两个回调方法中控制一个布尔型变量：enableOpenGLRendererFlag，并在进入后台的监听事件中将它设置为NO，在回到前台的监听事件中将它设置为YES。在线程的绘制过程中应该先判定这个变量是否为YES，是YES则进行绘制，否则不进行绘制，其整体实现结构如上所述。

接下来看一下初始化方法的实现，首先为layer设置属性，然后初始化NSOperation-Queue，并且将OpenGL ES的上下文构建以及OpenGL ES的渲染Program的构建作为一个Block（可以理解为一个代码块）直接加入到该Queue中。该Block中的具体行为如下：先分配一个EAGLContext，然后为该NSOperationQueue线程绑定OpenGL ES上下文，接着再创建FrameBuffer和RenderBuffer，将RenderBuffer的storage设置为UIView的layer（就是前面提到的CAEAGLLayer），然后再将FrameBuffer和RenderBuffer绑定起来，这样绘制在FrameBuffer上的内容就相当于绘制到了RenderBuffer上，最后使用前面提到的VertexShader和FragmentShader构造出实际的渲染Program，至此，初始化就完成了。

然后是关键的渲染方法，这里先判断当前OperationQueue中operationCount的值，如果其数目大于我们规定的阈值（一般设置为2或者3），则说明每一次绘制所花费的时间都比较多，这将导致很多绘制的延迟，所以可以删除掉最久的绘制操作，仅仅保留等于阈值个数的绘制操作，然后将本次绘制操作加入到OperationQueue中，该绘制操作的执行已经委托到OperationQueue的线程中了。由于在初始化的过程中我们已经为该线程绑定了OpenGL ES的上下文，所以可以在该线程中直接进行OpenGL ES的渲染操作。首先判定布尔型变量enableOpenGLRendererFlag的值，如果是YES，就绑定FrameBuffer，然后使用Program进行绘制，最后绑定RenderBuffer并且调用EAGLContext的PresentRenderBuffer将刚刚绘制的内容显示到layer上去，因为layer就是UIView的layer，所以能够在UIView中看到我们刚刚绘制的内容了。

至于销毁方法，也要保证这步操作是放在OperationQueue中执行的，因为涉及OpenGL ES的所有操作都要放到绑定了上下文环境的线程中去操作。具体实现中，首先要释放掉Program，然后释放掉FrameBuffer和RenderBuffer，最后将本线程与OpenGL上下文解除绑定。

对于UIView的dealloc方法，其功能主要是负责回收所有的资源，首先移除所有的监听事件，然后清空OperationQueue中未执行的操作，最后释放掉所有的资源。至此该VideoOutput就实现完毕了。

5.5 AVSync模块的实现

本节将介绍音视频同步模块的实现，即类图中AVSynchronizer类的实现，对于该类的职责，从它的名字上就可以看出来，主要是用于实现音视频同步的，我们在架构设计阶段就曾说过，不想把该系统拆得太细，所以该类的职责还包括维护解码线程，即创建、暂停、运行、销毁解码线程，基于以上分析，该类可分为两部分来实现：第一部分是维护解码线程，第二部分就是音视频同步。其主要接口与实现具体如下。

·当外界调用该模块的初始化方法的时候，根据要打开的媒体资源的URI对解码器进行实例化，并且将解码器维护为一个全局变量以便后续使用。

·当外界需要使用该类填充音频数据的时候，如果音频队列中已存在音频则直接填充即可，同时要记录下该音频帧的时间戳，如果音频队列中没有音频则填充空数据。

·当外界需要该类返回视频帧的时候，会根据当前播放的音频帧时间戳找到合适的视频帧并返回。

·当外界调用销毁方法的时候，首先会停止解码线程，然后销毁解码器，最后再销毁音视频队列。

5.5.1 维护解码线程

AVSync模块开辟的解码线程扮演了生产者的角色，其生产出来的数据所存放的位置就是音频队列和视频队列，而AVSync模块对外提供的填充音频数据和获取视频的方法则扮演了消费者的角色，从音视频队列中获取数据，其实这就是标准的生产者消费者模型。当客户端代码调用start方法的时候，就应该利用POSIX线程模型来创建一个解码线程，并且让该解码线程开始运行，从而解码音频帧和视频帧，解码出来的音视频帧要转换为我们自定义的结构体AudioFrame和VideoFrame，并且要把这两种类型的帧分别放入音频队列和视频队列中。那么在解码线程中具体应该如何调用VideoDecoder（输入模块）来进行解码的操作呢？其实现代码具体如下：

```
while(isOnDecoding) {
    pthread_mutex_lock(&videoDecoderLock);
    pthread_cond_wait(&videoDecoderCondition, &videoDecoderLock);
    pthread_mutex_unlock(&videoDecoderLock);
    isDecodingFrames = true;
    decodeFrames();
    isDecodingFrames = false;
}
```

上述代码在该线程中会是一个循环，只要不销毁该模块（销毁的时候会把全局变量isOnDecoding设置为false），就会一直运行该循环。在该循环内部首先会看到有一个条件锁，即每循环一次之后就会停在wait的地方，等待signal指令发送过来才可以进行下一次解码操作，为什么要这样安排呢？因为播放器播放的视频是随时间逐一进行播放的，而后台解码线程没必要将视频一次性全部解码完毕并放入队列中（一个原因是其在内存中基本上存储不开，因为视频所占用的空间实在是太大了；第二个原因是用户可能看一会儿就不看了，也就是说我们解码出来音视频帧就都作废了，没必要白白浪费CPU，如果是网络资源则还浪费了带宽），所以需要将解码线程设置成这种模式。这里规定两个值：min_bufferDuration和max_bufferDuration，比如将它们分别设置0.2s和0.4s，前面已经提到过解码线程其实是充当了生产者的角色，每调用一次decodeFrames方法时都会将两个队列填充至max_bufferDration的刻度之上，然后解码线程就会进入下一次循环，就在上面代码中的wait处等待signal指令。而当消费者线程每消费一次数据的时候，我们都会判断

队列中所有视频帧的长度是否在min_bufferDuration刻度之下，如果是在该刻度之下，就发送signal指令让解码线程进行解码。实现代码具体如下：

```
bool isBufferedDurationDecreasedToMin = bufferedDuration <= minBufferedDurati
if (isBufferedDurationDecreasedToMin && !isDecodingFrames) {
    int getLockCode = pthread_mutex_lock(&videoDecoderLock);
    pthread_cond_signal(&videoDecoderCondition);
    pthread_mutex_unlock(&videoDecoderLock);
}
```

当解码线程收到signal指令之后，就可以进行下一次解码了，如此一来，伴随着生产者线程和消费者线程的协同工作，整个视频播放器就可以播放出视频来了。

还有一点需要注意的是，在最后销毁该模块的时候，需要先将isOnDecoding变量设置为false，然后还需要额外发送一次signal指令，让解码线程有机会结束，如果不发送该signal指令，那么解码线程就有可能一直wait在这里，成为一个僵尸线程。

5.5.2 音视频同步

音视频同步的策略在前文中也曾提到过，这里再重点介绍一下，音视频同步一般分为三种：音频向视频同步、视频向音频同步、音频视频统一向外部时钟同步。在第3章学习FFmpeg框架时，也学习过ffplay，其中使用ffplay播放视频文件的时候，所指定的对齐方式就是上面所说的三种方式，下面就来逐一分析这三种对齐方式分别是如何实现的，以及各自的优缺点。

（1）音频向视频同步

先来看一下这种同步方式是如何实现的，音频向视频同步，顾名思义，就是视频会维持一定的刷新频率，或者根据渲染视频帧的时长来决定当前视频帧的渲染时长，或者说视频的每一帧肯定可以全都渲染出来，当我们向AudioOutput模块填充音频数据的时候，会与当前渲染的视频帧的时间戳进行比较，这个差值如果不在阈值的范围内，就需要做对齐操作；如果其在阈值范围内，那么就可以直接将本帧音频帧填充到AudioOutput模块，进而让用户听到该声音。那如果不在阈值范围内，又该如何进行对齐操作呢？这就需要我们去调整音频帧了，也就是说如果要填充的音频帧的时间戳比当前渲染的视频帧的时间戳小，那就需要进行跳帧操作（具体的跳帧操作可以是加快速度播放的实现，也可以是丢弃一部分音频帧的实现）；如果音频帧的时间戳比当前渲染的视频帧的时间戳大，那么就需要等待，具体实现可以是向AudioOutput模块填充空数据并进行播放，也可以是将音频的速度放慢播放给用户听，而此时视频帧是继续一帧一帧进行渲染的，一旦视频的时间戳赶上了音频的时间戳，就可以将本帧音频帧的数据填充到AudioOutput模块了。这就是音频向视频同步的实现，其优点就是视频可以将每一帧都播放给用户看，画面看上去是最流畅的，但是音频就会有所丢帧或者会插入静音帧，所以这种对齐方式会有一个明显的缺点，那就是音频有可能会加速（或者跳变）也有可能会有静音数据（或者慢速播放），如果变速系数不太大，那么用户感知可能不太强（但是如果系数变化比较大那么用户感知就会非常强烈了），发生丢帧或者插入空数据的时候，用户的耳朵是可以明显感觉到的。

（2）视频向音频同步

再来看一下视频向音频同步的方式是如何实现的，这与上面提到的方式恰好相反，由于不论是哪一个平台播放音频的引擎，都可以保证播放音频的时间长度与实际这段音频所代表的时间长度是一致的，所以我们可以依赖于音频的顺序播放为我们提供的时间戳，当客户端代码请求发送视频帧的时候，会先计算出当前视频队列头部的视频帧元素的时间戳与当前音频播放帧的时间戳的差值。如果在阈值范围内，就可以渲染这一帧视频帧；如果不在阈值范围内，则要进行对齐操作。具体的对齐操作方法就是：如果当前队列头部的视频帧的时间戳小于当前播放音频帧的时间戳，那么就进行跳帧操作；如果大于当前播放音频帧的时间戳，那么就进行等待（重复渲染上一帧或者不进行渲染）的操作。其优点是音频可以连续地播放，缺点是视频画面有可能会有跳帧的操作，但是对于视频画面的丢帧和跳帧，用户的眼睛是不太容易分辨得出来的。

（3）统一向外部时钟同步

这种策略其实更像是上述两种对齐方式的合体，其实现就是在外部单独维护一轨外部时钟，我们要保证该外部时钟的更新是按照时间的增加而慢慢增加的，当我们获取音频数据和视频帧的时候，都需要与这个外部时钟进行对齐，如果没有超过阈值，那么就直接返回本帧音频帧或者视频帧，如果超过了阈值就要进行对齐操作。具体的对齐操作是：使用上述两种方式里面的对齐操作，将其分别应用于音频的对齐和视频的对齐。优点是可以最大限度地保证音视频都可以不发生跳帧的行为，缺点是如果控制不好外部时钟，极有可能引发音频和视频都跳帧的行为。

根据人眼睛和耳朵的生理构造因素，得出了一个理论，那就是人的耳朵比人的眼睛要敏感得多，也就是说，如果音频有跳帧的行为或者填空数据的行为，那么我们的耳朵是十分容易察觉得到的；而视频如果有跳帧或者重复渲染的行为，我们的眼睛其实不容易分辨出来。根据这个理论，我们所实现的播放器将采用音视频对齐策略的第二种方式，即视频向音频对齐的方式。

5.6 中控系统串联起各个模块

下面介绍中控模块的实现，即类图中VideoPlayerController类的实现，这一部分其实是将上面提到的各个模块有序地组织起来，让单独运行的各个模块可以协同起来配合工作。由于每个模块都有各自的线程在运行，所以对于这部分代码，必须要负责好各个模块的生命周期的维护，否则极易产生多线程的问题，下面就分为三个阶段来讲解该模块，分别是初始化、运行和销毁三个阶段。

5.6.1 初始化阶段

1.Android平台

虽然我们的项目称为视频播放器，但即使客户端代码没有提供渲染的View，播放器也应该能够播放出声音来，而这是一个比较有用的功能（在一些产品中可以为用户提供非常好的体验，比如：在直播产品中秒开首屏、在一些视频播放器中正在播放的时候退出播放界面但还是有声音在播放、在切换回来时画面可以立即渲染类似于YouTube的播放界面的体验），所以在初始化阶段必须将播放器的初始化与渲染界面的初始化分离开来。如上述分析，初始化阶段应该分为两部分：一部分是播放器的初始化，另外一部分是渲染界面的初始化。

首先来看播放器的初始化，因为在初始化的过程中需要I/O操作，因此需要调用AVSync模块打开媒体资源，如果媒体资源是本地资源则还好；如果是网络资源，则建立连接的时间就不确定了（因为建立连接操作是阻塞的，直到建立连接成功之后才会返回），所以这里必须要开辟一个线程来进行初始化的操作，即利用PThread开辟一个initThread出来。在这个线程中，先实例化AVSynchronizer对象，然后调用该对象的init方法来建立与媒体资源的连接通道。如果打开连接失败，那么回调客户端会提示打开资源失败；如果打开连接成功，则根据媒体资源的Channel、SampleRate、SampleFormat以及fillAudioDataCallback回调函数和对象本身来初始化AudioOutput。如果可以初始化成功，则代表初始化步骤可以完成了，然后直接调用AVSync模块的start方法以及AudioOutput的播放方法。还有最后一步，由于上述操作都是在新开启的线程里面执行的，所以无法将初始化成功或者失败以及一系列的参数返回给客户端，故而最后一步就是将初始化成功与否回调给客户端对象，告诉客户端初始化播放器的状态，至此播放器就可以正常地播放音频了，但是视频呢？

接下来是渲染界面初始化的阶段，如果客户端调用层觉得现在这个时机可以显示视频的画面部分了，那么就会让SurfaceView进行显示操作，按照SurfaceView的生命周期，应该会调用设置Callback的onSurfaceCreated方法，也就是调用中控系统的initVideoOutput方法，这就是用来初始化渲染界面的，这里会直接初始化VideoOutput对象，然后用传递进来的ANativeWindow对象与界面的宽和高以及获取视频帧的

回调函数来初始化VideoOutput对象。以上就是初始化阶段所有的执行步骤了。

2.iOS平台

与Android平台不同的是，iOS的播放器在一个ViewController中，所以整个播放器的中控系统就是ViewController，从而播放器在iOS平台上的实现要简单一些，毕竟不需要两种语言的交互（Java层到Native层的数据和指令传递），所以这里的初始化就是整个播放器的初始化。还记得在AVSync模块中定义的PlayerStateCallback的Protocol吗？其中包含如下两个方法：

```
- (void) openSucceed;
- (void) connectFailed;
```

上述代码中的两个方法分别是在初始化方法执行成功或失败时，回调客户端代码时使用的。与Android平台类似，调用AVSync模块放在一个异步线程中来打开连接会更加合理，所以这里使用GCD线程模型，将初始化的操作放在一个DispatchQueue中。首先也是调用AVSync模块的openFile方法，如果可以打开媒体资源连接，则继续初始化VideoOutput对象。还记得VideoOutput实际上是一个继承自UIView的自定义View吗？我们需要把该View加入到ViewController中，但是当前是在一个子线程中初始化的ViewOutput的对象，所以我们必须dispatch到主线程中，然后调用如下代码：

```
[self.view insertSubview:_videoOutput atIndex 0];
```

接着还是在子线程中根据媒体文件中的声道数、采样率以及对象本身（作为实现AudioOutput类中声明的Protocol的实现者）来初始化AudioOutput对象，最终调用AudioOutput的开始播放方法。由于上述操作一直是在子线程中执行的操作，所以当执行完毕之后，可以使用前面提到的playerStateCallback来回调客户端代码，并告知客户端播放器初始化的状态，是成功还是失败。

5.6.2 运行阶段

1.Android平台

由于在初始化阶段已经开启了音频输出模块（调用了AudioOutput对象的start方法），因此，在OpenSL ES中将自己缓冲区里面的音频播放完毕之后，就会通过回调方法立马回调到中控模块，由中控模块来填充数据，而填充音频的方法就是最核心的实现。具体实现如下，首先会判断当前播放器的状态，如果处于暂停状态就不会再向AVSync模块请求数据，而是将静音数据（即全0的数据）填充到OpenSL ES并播放；当然如果AVSync已经被销毁了或者解码完毕了，那么也要将空数据填充到OpenSL ES并播放；如果上述情况都不满足的话，就需要调用AVSync模块填充音频数据的方法，待填充了这一帧的音频数据之后，就向VideoOutput（视频输出模块）发送一个指令，让VideoOutput模块来更新视频画面的一帧，当VideoOutput模块收到该指令时，就可以再调用自己的回调方法（由于在初始化的时候已经把中控系统的回调方法传递给了VideoOutput），从而调用到VideoPlayerController的获取视频帧的方法，调用AVSync模块的获取视频帧的方法之后，返回给视频播放模块，并将最新的一帧视频帧更新到画面中。

在运行阶段还有暂停和继续播放的接口实现，在Android平台上，由于整个播放器的驱动是由音频播放模块来驱动的，所以仅需要让音频播放模块暂停和继续就好了，所以这一块的实现是非常简单的。

2.iOS平台

由于iOS的中控系统实现了AudioOutput模块的FillDataDelegate这个Protocol，所以就需要实现该协议里面填充音频数据的方法，而实现该方法实际上就是运行阶段的核心控制。这里先判断AVSync模块是否播放完成或者播放器的当前状态是否处于暂停状态，如果已经播放完成或者是暂停状态了，那么就需要填充为静音数据（即全0的数据），如果没有播放完成，则调用AVSync模块的获取音频帧的方法，并且发送一个指令，让VideoOutput模块来更新画面数据。这就是运行中的最核心的部分了，其实很简单，就是为AudioOutput模块填充数据，并且通知VideoOutput模块来更新画面。

再就是暂停和继续播放，其实现与Android平台很类似，当外界调用暂停和继续的时候，调用AudioOutput模块的暂停和继续就可以了，其实就是让我们的播放器驱动端来暂停和继续。

5.6.3 销毁阶段

1.Android平台

销毁阶段其实就是初始化阶段的逆过程，首先应该中断媒体资源的连接通道，可调用AVSync模块的interruptRequest方法来实现，然后再来看一下初始化阶段的线程有没有执行结束，如果没有执行结束则等待它的结束，所以这里需要使用排程的方法pthread_join等待初始化线程执行结束。然后优先停止VideoOutput，直接调用VideoOutput的stopOutput方法，紧接着再暂停音频输出模块，然后销毁AVSync模块，该模块会等待解码线程的结束并且销毁解码器（输入模块），最后再调用音频输出模块的销毁方法，这样就可以销毁掉所有的模块了。

2.iOS平台

根据运行阶段的介绍我们可以知道，由于音视频对齐策略的影响，整个播放过程其实是由音频来驱动的，所以在销毁阶段肯定需要首先停止音频，所以这里首先调用AudioOutput对象的stop方法；然后应该停止AVSync模块，由于该模块包含了解码线程，所以需要断开输入模块的连接，这里首先应该判断输入模块打开连接通道是否成功，如果没有打开成功，则应该中断连接；如果打开成功，则应该调用AVSync模块的销毁方法（里面会把音频队列、视频队列、解码线程以及解码器都销毁掉，具体实现请参考前面的销毁方法）；最后一步应该是停止VideoOutput模块，通过调用VideoOutput的销毁资源的方法（里面将会销毁FrameBuffer、Renderbuffer、Program等，具体实现请参考前面章节的销毁方法）来实现，最终再将VideoOutput这个自定义的view从ViewController中移除，至此销毁阶段就实现完毕了。

5.7　本章小结

视频播放器已经实现完毕，下面来回顾一下整个设计与开发阶段。在此之前，需要说明的是，在书中大量罗列代码可不是一件好事，因此本书会尽量少地罗列代码，而是引导大家一起逐步设计并实现这款播放器。本章先将其拆分为各个子模块并逐一实现。

·首先实现了输入模块（或者称为解码模块），输出音频帧是AudioFrame，其中的主要数据是PCM裸数据；输出视频帧是VideoFrame，其中的主要数据是YUV420P的裸数据。

·然后实现了音频播放模块，输入是解码出来的AudioFrame，直接就是SInt16表示的sample格式的数据，输出则是输出到Speaker用户能够直接听到声音。

·接着实现了视频播放模块，输入是解码出来的VideoFrame，其中存放的是YUV420P格式的数据，在渲染过程中可使用OpenGL ES的Program将YUV格式的数据转换为RGBA格式的数据，并最终显示到物理屏幕上。

·之后就是音视频同步模块了，它的工作主要由两部分组成：第一部分是负责维护解码线程，即负责输入模块的管理；另外一部分是音视频同步，可向外部提供填充音频数据的接口和获取视频帧的接口，以保证所提供的数据是同步的。

最后编写一个中控系统，负责将AVSync模块、AudioOutput模块、VideoOutput模块组织起来，最重要的就是维护这几个模块的生命周期，由于其中存在多线程的问题，所以需要重点注意的是，应在初始化、运行、销毁各个阶段保证这几个模块可以协同有序地运行，同时中控系统应对外提供用户可以操作的接口，比如开始播放、暂停、继续、停止等接口。

第6章　音视频的采集与编码

前面4章探讨了声音与画面的解码与渲染，第5章完成了一个视频播放器的项目。对于一个完整的多媒体App来说，只有播放而没有录制是不够的，对于视频的录制，最重要的就是声音和画面采集和编码，本章就来学习音视频的采集和编码，紧接着第7章将会通过一个完整的录制视频的项目来巩固本章所学的内容。

6.1 音频的采集

本节将会学习Android平台与iOS平台音频采集的知识，请不太了解音频基础知识的读者先回头看第1章的内容，因为在音视频的开发过程中，经常要涉及这些基础知识，掌握了这些重要的概念之后，开发过程中遇到的很多参数和流程就会更加容易理解。

6.1.1 Android平台的音频采集

Android SDK提供了两套音频采集的API，分别是：MediaRecorder和AudioRecord。前者是一个更加上层的API，它可以直接对手机麦克风录入的音频数据进行编码压缩（如AMR、MP3等），并存储为文件；后者则更加接近底层，能够更加自由灵活地控制，其可以让开发者得到内存中的PCM音频流数据。如果想做一个简单的录音机，输出音频文件，则推荐使用MediaRecorder；如果需要对音频做进一步的算法处理，或者需要采用第三方的编码库进行压缩，又或者需要用到网络传输等场景中，那么只能使用AudioRecord或者OpenSL ES，其实MediaRecorder底层也是调用了AudioRecord与Android Framework层的AudioFlinger进行交互的。而我们的项目场景显然更倾向于第二种方式，即使用AudioRecord来采集音频，至于OpenSL ES录制音频的API，由于其属于Native层提供的接口，因此本章不做讲解。

既然选择了AudioRecord这个Android平台所提供的SDK，那么取得内存中的PCM数据之后又该如何处理呢？在多媒体App中，一般会对声音进行特效处理，然后将其编码为一个AAC或者MP3文件，至于音频的处理前面还没有讲解，而对于音频的编码本章后续会进行学习，所以这里先暂时简单地写成PCM文件。在录制结束之后，可以从SD卡中读取该PCM文件，然后使用ffplay进行播放（如何用ffplay播放PCM文件，是第3章所讲的内容，如果忘记了可以回头去查看命令）以试听效果。

如果想要使用AudioRecord这个API，则需要在应用AndroidManifest.xml的配置文件中增加权限，代码如下：

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

若要把录制出来的数据写入文件中，则需要配置写入文件的权限，代码如下：

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

接下来了解一下AudioRecord的工作流程。

1.配置参数，初始化内部的音频缓冲区

首先来看一下AudioRecord的配置参数，AudioRecord是通过构造函数来配置参数的，其函数原型如下：

```
public AudioRecord(int audioSource, int sampleRateInHz, int channelConfig, ir
    audioFormat, int bufferSizeInBytes).
```

上述参数所代表的含义及其在各种场景下应该传递的值具体说明如下。

·audioSource，该参数指的是音频采集的输入源，可选值以常量的形式定义在类AudioSource（MediaRecorder的一个内部类）中，常用的值包括：

·DEFAULT（默认）

·VOICE_RECOGNITION（用于语音识别，等同于DEFAULT）

·MIC（由手机麦克风输入）

·VOICE_COMMUNICATION（用于VoIP应用）等

·sampleRateInHz，用于指定以多大的采样频率来采集音频，现在用得最多的就是44100的采样频率（就是我们常说的44.1kHz的采样率），如果使用该采样率初始化录音器失败的话，则可以使用16000的采样频率（就是我们常说的16kHz的采样率）来尝试一下。

·channelConfig，该参数用于指定录音器采集几个声道的声音，可选值以常量的形式定义在AudioFormat类中，常用的值包括：

·CHANNEL_IN_MONO（单声道）

·CHANNEL_IN_STEREO（立体声）

·由于现在的移动设备都是伪立体声的采集，所以出于性能考虑，一般按照单声道进行采集，然后在后期处理中将数据转换为立体声。

·audioFormat，这就是基础概念里面介绍过的采样的表示格式，可选值以常量的形式定义在AudioFormat类中，常用的值包括：

·ENCODING_PCM_16BIT（16bit）

·ENCODING_PCM_8BIT（8bit）

·注意，前者可以保证兼容大部分的Android手机。

·bufferSizeInBytes，这是最难理解但最重要的一个参数，其配置的是AudioRecord内部的音频缓冲区的大小，而具体的大小，不同的厂商会有不同的实现，该音频缓冲区越小，产生的延时就会越小。AudioRecord类提供了一个静态方法用于确定该bufferSizeInBytes的函数，其原型如下：

```
int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat);
```

·在实际开发中，强烈建议由该函数计算出需要传入的bufferSizeInBytes，而不是使用自己计算的值。

配置好AudioRecord之后，应该检查一下AudioRecord的当前状态，因为极有可能会因为权限（用户未对我们的应用授权访问麦克风的权限），或者由于其他应用没有正确地释放录音器等原因，使得构造的AudioRecord的状态不正常。可以通过AudioRecord的方法getState来获取当前录音器的状态，然后与AudioRecord.STATE_INITAILIZED进行比较，如果不相等，则提示用户尚未获得录音权限。

2.开始采集

待创建好AudioRecord对象之后，就可以开始进行音频数据的采集了，可通过下面的函数来控制麦克风音频采集的开始：

```
audioRecord.startRecording();
```

3.提取数据

执行完上述命令之后，需要一个线程，从AudioRecord的缓冲区中将音频数据不断地读取出来。注意，该过程一定要及时，否则会出现“overrun”的错误，该错误在音频开发中比较常见，其意味着应用层没有及时地“取走”音频数据，从而导致内部的音频缓冲区溢出。读取录音器采集到的PCM数据的方法原型如下所示：

```
public int read(byte[] audioData, int offsetInBytes, int sizeInBytes)
```

当然，也可以读取short数组类型的数据，因为其更符合底层的一些处理，毕竟我们的设置是每个sample都由一个short来表示，方法原型如下：

```
public int read(short[] audioData, int offsetInShorts, int sizeInShorts)
```

拿到数据之后就可以直接写入文件了，可以通过Java层提供的FileOutputStream将数组直接写到文件中。

4.停止采集，释放资源

当我们想要停止录音的时候，可调用AudioRecord的stop方法来实现，并且最终要对该AudioRecord的实例调用release，代表释放掉了录音器，以便设备的其他应用可以正常使用录音器，这一点是十分重要的。可以通过布尔型变量的控制先读取数据的线程并结束，然后再停止和释放AudioRecord实例，最终再关闭写入数据的文件，否则会有文件写出不完全的问题。

本节的实例在代码仓库里是AudioRecorder项目，运行该项目，进入录音界面，点击record开始录音，点击stop停止录音，然后利用adb pull导出PCM文件：

```
adb pull /mnt/sdcard/vocal.pcm ~/Desktop/
```

利用ffplay播放声音：

```
ffplay -f s16le  -sample_rate 44100  -channels 1 -i vocal.pcm
```

也可以利用FFmpeg将PCM文件转换为WAV文件，然后使用PC上的系统播放器进行播放：

```
ffmpeg -f s16le -sample_rate 44100 -channels 1 -i vocal.pcm -acodec pcm_s16le
    vocal.wav
```

6.1.2 iOS平台的音频采集

iOS平台提供了多套API采集音频，如果开发者想要直接指定一个路径，则可以将录制的音频编码到文件中，可以使用AVAudioRecorder这套API，其优点是简单易用，但是其对于想要实时地在内存中获得录音的数据来说，限制性非常强。对此，iOS平台提供了两个层次的API来协助实现，第一种方式是使用AudioQueue，第二种方式是使用AudioUnit，实际上AudioQueue是AudioUnit更高级的封装，相比较于AudioUnit，它提供的功能更单一，使用的接口调用更简单，具体使用哪个层次的API需要根据应用场景来决定。如果仅仅是要获取内存中的录音数据，然后再进行编码输出（有可能是输出到本地磁盘，也有可能是网络），那么使用更高级的AudioQueue的API会更好一些；如果要使用更多的音效处理，以及实时的监听（在耳机中可以听到自己说的话，在唱歌的App中这是一个最基础的功能），那么使用AudioUnit会更加方便一些。本书的代码案例中，使用的都是AudioUnit，因为本书案例实现的场景不仅需要耳返，还需要音效的实时处理等功能。

要想使用iOS的麦克风进行录音，首先要为App声明使用麦克风的权限，在新建的目录下面找到info.plist，然后在其中新增麦克风权限的声明：

```
<key>NSMicrophoneUsageDescription</key>
<string>microphoneDesciption</string>
```

这样添加之后，系统就知道了App要访问系统的麦克风权限。这里使用的AudioUnit，本书的第4章在讨论音频渲染的时候已经有很详尽的讲解，所以这里直接来看如何使用AudioUnit实现人声录制，同时，还会发送给耳机一个监听耳返（等完成这一功能之后，读者肯定会感叹，这些在iOS平台上实现起来是多么的简单）。

与第4章讲解的一样，要使用AudioUnit，首先需要通过AVAudioSession来开启硬件设备以及对硬件设备做一些设置，然后才能使用AudioUnit，而AVAudioSession的具体使用方法也和前面讲解的一致，下面再来熟悉一下：

1）获得AVAudioSession的实例，由于AVAudioSession是单例模式设计的，所以在这里只需要调用它的单例方法就可以获得。

2）为AudioSession设置使用类别，由于要在录音的同时为用户输送一路监听耳返，所以这里选择使用类别AVAudioSessionCategoryPlayAndRecord，本书前面的章节中，仅在做音频渲染时使用到类别AVAudioSessionCategoryPlayback。所以设置这个类别的目的是告诉系统的硬件应该为我们的App提供什么样的服务。

3）为AudioSession设置预设的采样率。

4）启用AudioSession。

5）为AudioSession设置路由监听器，目的就是在采集音频或者音频输出的线路发生变化的时候（比如插拔耳机、蓝牙设备连接成功等）回调此方法，以便开发者可以重新设置使用当前最新的麦克风或扬声器。

至此AudioSession就设置好了，接下来的事情就是构造该应用所使用的AUGraph，其构造步骤与第4章中音频渲染构造的AUGraph也很类似，因为这里要使用录音功能，所以需要启用RemoteIO这个AudioUnit的InputElement。RemoteIO这个AudioUnit比较特别，Input-Element实际上使用的是麦克风，而OutputElement使用的则是扬声器，所以这里首先会启用RemoteIOUnit的InputElement。为了支持所开发的App可以在后续Mix一轨伴奏这一扩展功能，在AUGraph中需要增加MultiChannelMixer这个AudioUnit。由于每个AudioUnit的输入输出格式并不相同，所以这里还要使用AudioConvert这个AudioUnit将输入的AudioUnit连接到MixerUnit上。最终将MixerUnit连接到RemoteIO这个AudioUnit的OutputElement，将声音发送到耳机的扬声器中（如果直接发送到手机的扬声器中就会出现啸叫），这样就将AUGraph整体地建立起来了，如图6-1所示。

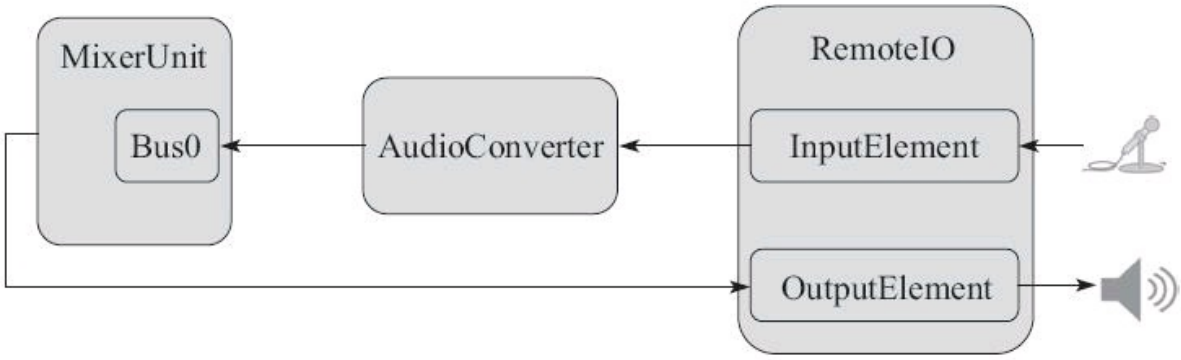


图　6-1

此外，这里还需要实现一个功能，就是将录音得到的数据存储为一个文件。若我们想要获取某个AudioUnit的数据，并将其写入文件，那么可以在它后一级的AudioUnit中增加一个回调，然后在回调方法中将该AudioUnit的数据渲染出来并发送给下一级的AudioUnit，同时也可以去写文件，这种方式已经在音频渲染的时候使用过了。这里就为RemoteIO这个AudioUnit的OutputElement增加一个回调，代码如下：

```
AURenderCallbackStruct finalRenderProc;
finalRenderProc.inputProc = &renderCallback;
finalRenderProc.inputProcRefCon = (__bridge void *)self;
status = AUGraphSetNodeInputCallback(_auGraph, _ioNode, 0, &finalRenderProc);
```

然后在上述回调方法的实现中，将它的前一级MixerUnit的数据渲染出来，同时写文件，代码如下：

```
static OSStatus renderCallback(void *inRefCon,
        AudioUnitRenderActionFlags *ioActionFlags, const AudioTimeStamp
        *inTimeStamp, UInt32 inBusNumber, UInt32 inNumberFrames, AudioBufferL
        *ioData){
    OSStatus result = noErr;
    __unsafe_unretained AudioRecorder *THIS = (__bridge AudioRecorder *)inRef
    AudioUnitRender(THIS->_mixerUnit, ioActionFlags,
            inTimeStamp, 0, inNumberFrames, ioData);
    // Write To File
    return result;
}
```

对于写文件，后面的6.3节中会使用AudioToolbox来编码文件，但是这里将使用一个更高级的API来写文件，即ExtAudioFile，iOS提供的这

个API只需要设置好输入格式、输出格式以及输出文件路径和文件格式即可，代码如下：

```
AudioStreamBasicDescription destinationFormat;
CFURLRef destinationURL;
result = ExtAudioFileCreateWithURL(destinationURL, kAudioFileCAFType,
        &destinationFormat, NULL, kAudioFileFlags_EraseFile, &audioFile);
result = ExtAudioFileSetProperty(audioFile,
        kExtAudioFileProperty_ClientDataFormat, sizeof(clientFormat),
        &clientFormat);
UInt32 codec = kAppleHardwareAudioCodecManufacturer;
result = ExtAudioFileSetProperty(audioFile,
        kExtAudioFileProperty_CodecManufacturer, sizeof(codec), &codec);
```

在需要编码文件时直接写入数据：

```
ExtAudioFileWriteAsync(audioFile, inNumberFrames, ioData);
```

在停止写入的时候调用关闭方法即可：

```
ExtAudioFileDispose(audioFile);
```

最终就可以得到我们想要的文件了，大家可以从应用的沙盒中将保存的文件读取出来（在XCode中利用Device取出文件或者使用iExplorer等软件取出），然后播放试听一下。

本节的代码示例是代码仓库中的AudioRecorder，建议读者运行该项目的时候插上耳机，以免设备发出啸叫的声音，另外，可点击recorder开始录音，点击stop结束录音，并且可取出对应的录音文件在PC上用系统播放器进行播放试听。

6.2 视频画面的采集

视频画面的采集主要是使用各个平台提供的摄像头API来实现的，在为摄像头设置了合适的参数之后，将摄像头实时采集的视频帧渲染到屏幕上提供给用户预览，然后将该视频帧编码到一个视频文件中，其使用的编码格式一般是H264。当然，最终我们还要配上音频，否则没有音频文件的视频就成了早期的默片电影了。

本节将主要学习如何在Android和iOS平台上利用各自平台提供的摄像头API，采集出正确的视频帧并绘制到屏幕上，具体的编码将会在后续进行讨论。

6.2.1 Android平台的视频画面采集

1.权限配置

要想使用Android平台提供的摄像头，首先必须在配置文件中添加如下权限要求：

```
<uses-permission android:name="android.permission.CAMERA" />
```

伴随着Android系统的发展，Android的摄像头API也已经有了非常大的变化，现在选用的Camera的使用方式为设置预览纹理的形式，而不是设置YUV数据回调的方式，这是因为得到纹理ID之后，可以很方便地进行视频滤镜处理，并且很容易渲染到界面上。

2.打开摄像头

Android平台提供了打开摄像头的API，其函数原型如下：

```
public static Camera open(int cameraId)
```

需要传入的参数就是摄像头的ID，从手机的发展历史可以知道，先有后置摄像头，然后才有前置摄像头，甚至目前已经有部分手机有了更多的辅助摄像头，所以摄像头的ID排列是后置摄像头是0，前置摄像头是1，然后才是其他的摄像头，即便是这样，我们也要使用CameraInfo类里面的两个常量，它们分别如下。

·CAMERA_FACING_BACK代表后置摄像头。

·CAMERA_FACING_FRONT代表前置摄像头。

该函数返回的就是一个摄像头的实例，如果返回的是NULL，或者抛出异常（因为不同厂商所给出的返回是不一样的），则代表用户没有授权该应用访问摄像头。

3.配置摄像头参数

获取到该摄像头实例之后，要为该摄像头实例设置对应的参数，参数的配置主要涉及如下两个参数。

第一个参数是预览格式，一般设置为NV21格式的，实际上就是YUV420SP的格式，即UV是interleaved（交错UVUVUV）的存放，代码设置如下：

```
List<Integer> supportedPreviewFormats = parameters.getSupportedPreviewFormats
if (supportedPreviewFormats.contains(ImageFormat.NV21)) {
    parameters.setPreviewFormat(ImageFormat.NV21);
} else {
    throw new CameraParamSettingException("视频参数设置错误:设置预览图像格式异常");
}
```

上述代码先取出摄像头所支持的所有预览格式，然后判断其是否包含我们要设定的格式，如果包含，则设置进去；如果不包含，则抛出异常，让客户端代码进行处理。

第二是设置预览的尺寸，分辨率的尺寸一般设置为1280×720，当然对于某些应用来说，可能也会设置为640×480的分辨率，代码设置如下：

```
List<Size> supportedPreviewSizes = parameters.getSupportedPreviewSizes();
int previewWidth = 640;// 1280
int previewHeight = 480;// 720
boolean isSupportPreviewSize = isSupportPreviewSize(
        supportedPreviewSizes, previewWidth, previewHeight);
if (isSupportPreviewSize) {
    parameters.setPreviewSize(previewWidth, previewHeight);
} else {
    throw new CameraParamSettingException("视频参数设置错误:设置预览的尺寸异常");
}
```

上述代码会取出摄像头所支持的所有分辨率列表，然后判断要设置的分辨率是否在支持的列表中，如果包含，则设置进去，否则抛出异常，让客户端代码进行处理。

配置完上述参数的设置之后，就需要将该参数设置给Camera实例了，代码如下：

```
try {
    mCamera.setParameters(parameters);
} catch (Exception e) {
    throw new CameraParamSettingException("视频参数设置错误");
}
```

在宽高的设置中，细心的读者可能已经注意到了宽是1280（或者640），高是720（或者480），这是因为摄像头默认采集出来的视频画面是横版的，在显示的时候，需要获取当前这个摄像头采集出来的画面的旋转角度，那么具体的旋转角度应该如何获取呢？代码如下：

```
int degrees = 0;
CameraInfo info = new CameraInfo();
Camera.getCameraInfo(cameraId, info);
if (info.facing == Camera.CameraInfo.CAMERA_FACING_FRONT) {
    degrees = (info.orientation) % 360;
} else { // back-facing
    degrees = (info.orientation + 360) % 360;
}
```

根据不同的摄像头取出对应的CameraInfo，该CameraInfo中的orientation变量表示的就是该摄像头采集到的画面的旋转角度，不过，要想正确地旋转还需要再处理一下，如果是前置摄像头，则直接对360进行取模；如果是后置摄像头，则先加上360度再取模360，从而就能得到想要旋转的角度。得到的这个角度对于后续可以将视频帧正确地显示到屏幕上来说是至关重要的，下面讲述摄像头的预览时就会用到该角度参数。

4.摄像头的预览

配置好摄像头之后，剩下的事情就是配置摄像头采集每一帧图像的回调，并且获取到图像之后将图像渲染到屏幕上。本书的第4章已经讲解过了如何通过OpenGL ES来渲染图像，这里先来回顾一下：首先把图像解码为RGBA格式；然后将RGBA格式的字节数组上传到一个纹理上；最终将该纹理渲染到屏幕上。所以这里的渲染到屏幕上也会使用OpenGL ES来实现。由于这里要显示的纹理是摄像头按照一定的刷新频率（fps）来更新的，所以最终显示出来的就是我们预期的预览效果了。

整个预览过程分为三个阶段，分别为开始预览、刷新预览与结束预

览。我们首先讲解开始预览阶段，整体流程如图6-2所示。

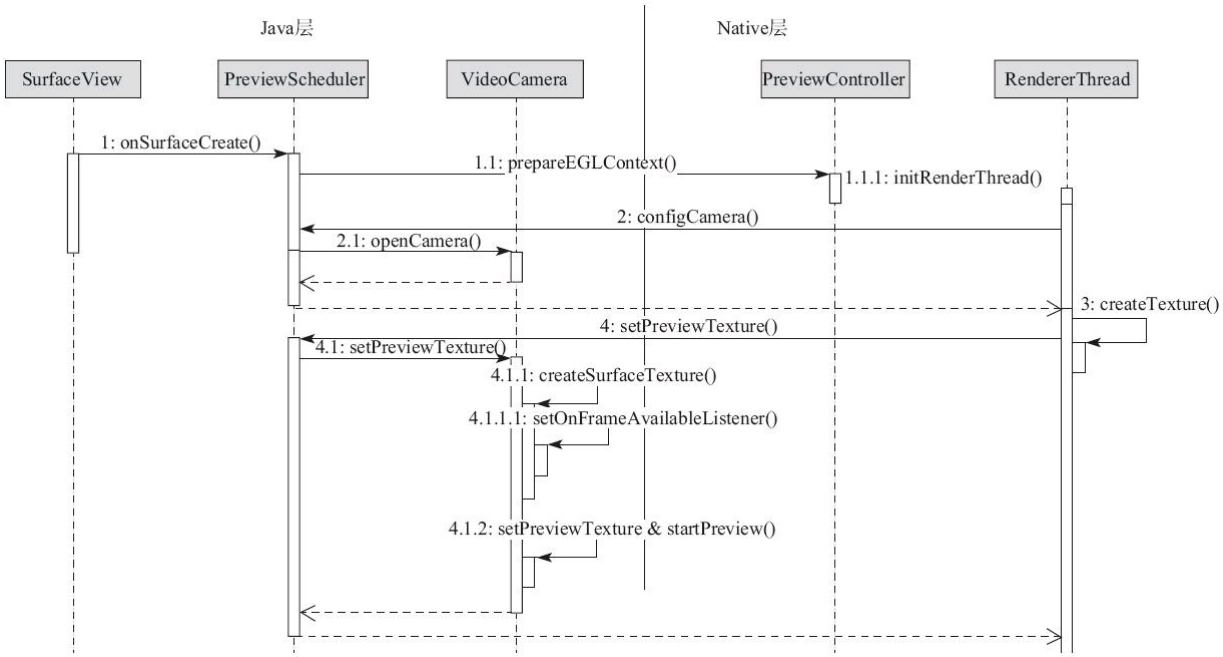


图　6-2

如图6-2所示，首先在Activity的界面层构造一个SurfaceView用于显示渲染结果；然后在Native层用EGL和OpenGL ES构造一个渲染线程用于渲染该SurfaceView，同时在该渲染线程中生成一个纹理ID并传递到Java层；Java层利用该纹理ID构造出一个Surface-Texture，之后再将该SurfaceTexture作为Camera的预览目标。最终调用Camera的开始预览方法，这样就可以将摄像头采集到的视频帧渲染到设备屏幕上了。

但是如何让摄像头按照频率采集出来的视频帧依次进行渲染呢？答案是在图6-3中构造好了SurfaceTexture对象之后，要为该对象设置视频帧可用时的监听器，实际上就是当SurfaceTexture在可以更新的时候调用该监听器（即当Camera设备采集到一帧视频帧的时候会回调该监听器方法）。将纹理ID设置给摄像头的代码如下：

```
mCameraSurfaceTexture = new SurfaceTexture(textureId);
try {
    mCamera.setPreviewTexture(mCameraSurfaceTexture);
    mCameraSurfaceTexture.setOnFrameAvailableListener(frameAvailableListener)
    mCamera.startPreview();
} catch (Exception e) {
    throw new CameraParamSettingException("设置预览纹理错误");
```

```
}
```

如上所述，代码中的frameAvailableListener是继承自OnFrameAvailableListener内部类的一个实例，在该内部类中重写onFrameAvailable方法，在该方法中调用Native层的方法来渲染摄像头刚刚捕捉的图像。调用到了Native层之后，将会委托到渲染线程中去调用Java层的SurfaceTexture的updateTexImage方法（因为必须在OpenGL ES的渲染线程中才可以调用该方法，所以绕了一大圈）。更新视频帧的整体流程如图6-3所示。

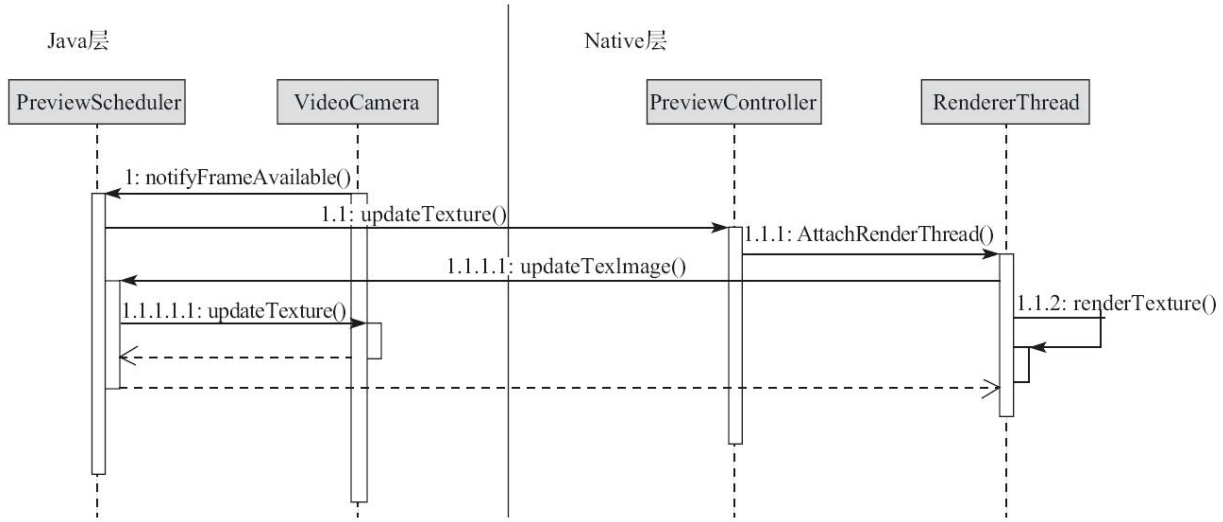


图　6-3

图6-3中，当VideoCamera的方法updateTexture执行完毕之后，就说明摄像头采集的视频帧已经更新到Native层生成的纹理ID上了，渲染线程就可以把该纹理ID渲染到界面上去了。当摄像头再次采集到一帧新视频帧的时候，就会周而复始地执行上述过程，这样在设备屏幕上就可以流畅地看到摄像头的预览了。

对于渲染线程的搭建以及如何将一帧纹理绘制到上层界面的知识已经在前面章节中讲解过了，那么摄像头采集到这一帧视频帧之后是如何进行渲染的呢？这也是接下来的重点，下面一起来看看。

前面提到过，要在渲染线程中生成一个纹理ID，然后传递到Java层，再由Java层构造成一个SurfaceTexture类型的对象，并将Camera的PreviewCallback设置为该SurfaceTexture对象。由于摄像头采集出来的视

频帧的格式是NV21，即采集出来的一帧的格式是YUV420SP，width*height个像素点共占用了width*height*3/2个字节数，即每个像素点都会有一个Y放到数据存储的前width*height个数据中，每四个像素点共享一个UV放到后半部分进行交错存储。而在OpenGL中使用的绝大部分纹理ID都是RGBA的格式，另外之前在讲解播放器项目的时候也曾讲过Luminance格式，但是那里是开辟3个纹理ID来表示一张YUV的图片，这里必须使用一个纹理ID来为Camera更新数据，那么应该如何将3个Luminance的纹理ID合并成一个纹理ID呢？幸好OpenGL ES的扩展GL_OES_EGL_image_external定义了一个纹理的扩展类型，即GL_TEXTURE_EXTERNAL_OES，否则整个转换过程将会非常复杂。同时这种纹理目标对纹理的使用方式也会有一些限制，纹理绑定需要绑定到类型GL_TEXTURE_EXTERNAL_OES上，而不是类型GL_TEXTURE_2D上，对纹理设置参数也要使用GL_TEXTURE_EXTERNAL_OES类型，生成纹理与设置纹理参数的代码如下：

```
glGenTextures(1, &texId);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, texId);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE)
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE)
```

在实际的渲染过程中绑定纹理的代码如下：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, texId);
glUniform1i(uniformSamplers, 0);
```

在OpenGL ES的shader中，任何需要从纹理中采样的OpenGL ES 2.0的shader都需要声明其对此扩展（GL_OES_EGL_image_external）的使用，使用指令如下：

```
#extension GL_OES_EGL_image_external : require
```

这些shader也必须使用samplerExternalOES采样方式来声明纹理，其在FragmentShader中的代码如下：

```
static char* GPU_FRAME_FRAGMENT_SHADER =
"#extension GL_OES_EGL_image_external : require              \n"
"precision mediump float;                                     \n"
"uniform samplerExternalOES yuvTexSampler;                   \n"
"varying vec2 yuvTexCoords;                                  \n"
"                                                            \n"
"void main() {                                               \n"
"    gl_FragColor = texture2D(yuvTexSampler, yuvTexCoords);  \n"
"}                                                           \n";
```

至此，这种扩展类型的纹理ID从创建到设置参数，再到真正的渲染整个过程已经处理完毕，弄清楚了这种特殊纹理ID的使用方法之后，接下来再看一下具体的旋转角度问题，因为在使用摄像头的时候很容易在这个地方踩到坑，比如手机摄像头预览的时候会出现倒立、镜像等问题，下面就来彻底地解决这类问题。

摄像头采集出来的视频都是横屏的，比如开发者为摄像头设置的预览大小是640×480，实际上摄像头采集出来的视频帧宽是640，高是480，并且图片也是横向采集的。正常来讲，用户使用手机时都是竖直方向的，所以需要旋转90度或者270度用户才可以正确地看到自己的预览效果。而具体旋转多大角度需要在当前这颗摄像头的CameraInfo中获得，不同的手机甚至是不同的系统都会不一样。并且如果是前置摄像头的话，还需要再做一个VFlip（假设图像是横向采集出来的所以要做竖直翻转，如果是已经旋转过了的就要做横向翻转）用于修复镜像的问题，下面就用实际的图片来分别看一下前置摄像头和后置摄像头的具体渲染流程。

首先我们来看一张摄像头要实际采集的物体，如图6-4所示。

图　6-4

在使用手机的摄像头去采集这个物体时，如果是前置摄像头，那么采集得到的图片将如图6-5最左边的图片所示（摄像头的CameraInfo中取出来的角度是270度），对此，应该按照摄像头的旋转角度将图片顺时针旋转（注意这里一定是顺时针），旋转270度之后将得到如图6-5中间的图片，最后再进行镜像处理，得到如图6-5最右边的图片，最终用户在手机屏幕中看到的预览才是预期的图像。

图　6-5

如果是后置摄像头，那么一般情况下从摄像头的CameraInfo中取出来的角度是90度，当然这一点会根据ROM厂商来决定，比如LG厂商的Nexus 5X设备取出来的角度就是270度，不论是多少度，摄像头采集出来的图像在旋转过该角度之后肯定会是一个正常的图像，旋转流程如图6-6所示。

图　6-6

如果是LG厂商的Nexus 5X或者HUAWEI厂商的Nexus 6P这两款设备系统升级之后，我们对图像后置摄像头的处理将如图6-7所示。

图　6-7

那么接下来就讲解一下图像的旋转和镜像。在OpenGL ES中这个问题其实就是如何来确定物体的坐标和纹理坐标，尽管在第4章中已经讲解过这部分内容，而在这里由于既要做旋转又要做镜像操作，所以需要

先回顾一下物体的坐标系，如图6-8所示。

通过如下数组来规定物体坐标：

```
GLfloat squareVertices[8] = {
    -1.0, -1.0,       // 物体左下角
    1.0，-1.0，       // 物体右下角
    -1.0，1.0，       // 物体左上角
    1.0，1.0          // 物体右上角
};
```

下面再回顾一下OpenGL的纹理坐标系，如图6-9所示。

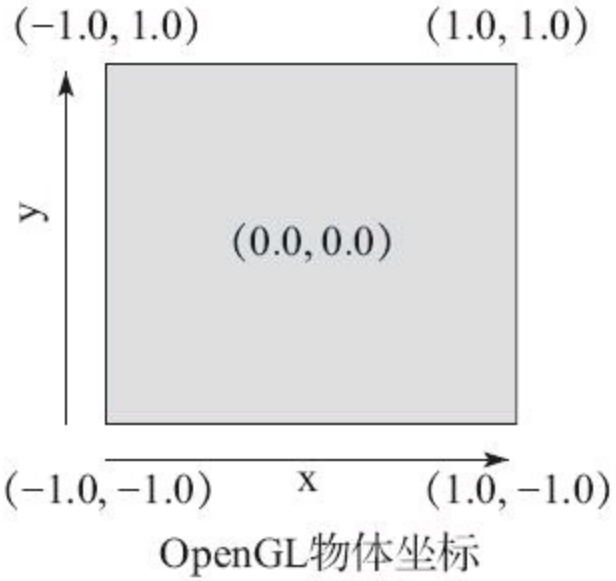


图　6-8

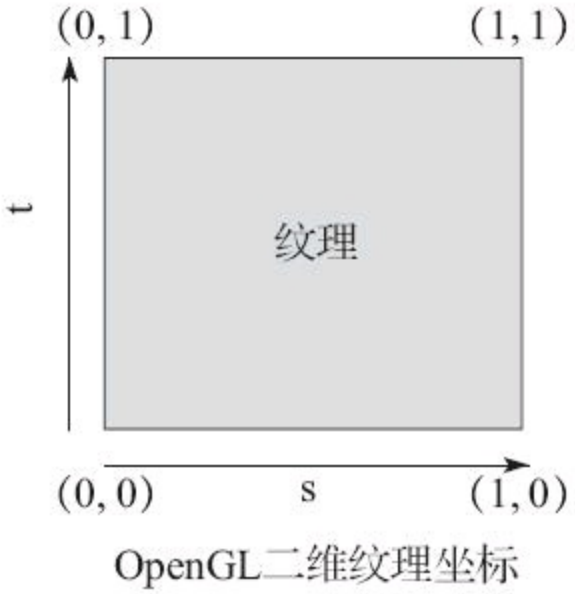


图　6-9

然后给出不做任何旋转的纹理坐标：

```
GLfloat textureCoordNoRotation[8] = {
    0.0，0.0，       // 图像的左下角
    1.0，0.0，       // 图像的右下角
    0.0，1.0，       // 图像的左上角
    1.0，1.0         // 图像的右上角
};
```

再给出顺时针旋转90度的纹理坐标，大家可以想象一下，将图6-9顺时针旋转90度，然后再把对应的左下、右下、左上、右上的坐标点写下来，如下所示：

```
GLfloat textureCoords[8] = {
    1.0，0.0，       // 图像的右下角
    1.0，1.0，       // 图像的右上角
    0.0，0.0，       // 图像的左下角
    0.0，1.0         // 图像的左上角
};
```

现在，再给出顺时针旋转180度的纹理坐标：

```
GLfloat textureCoords[8] = {
    1.0，1.0，       // 图像的右上角
    0.0，1.0，       // 图像的左上角
    1.0，0.0，       // 图像的右下角
    0.0，0.0         // 图像的左下角
};
```

之后给出顺时针旋转270度的纹理坐标：

```
GLfloat textureCoords[8] = {
    0.0，1.0，       // 图像的左上角
    0.0，0.0，       // 图像的左下角
    1.0，1.0，       // 图像的右上角
    1.0，0.0         // 图像的右下角
};
```

还记得第4章中讲过的计算机图像的坐标系与OpenGL的坐标系有什么不同吗？它们的y轴坐标恰好是相反的，所以这里要对每一个纹理坐标做一个VFlip的变换（即把每一个顶点的y值由0变为1或者由1变为0），这样就可以得到一个正确的图像旋转了。而前置摄像头还存在镜像的问题，因此需要对每一个纹理坐标做一个HFlip的变换（即把每一个顶点的x值由0变为1或者由1变为0），从而让图片在预览界面中看起来就像在镜子中的一样。

上面的步骤其实就是将一个特殊格式（OES）的纹理ID经过处理和旋转，使其变成正常格式（RGBA），那么接下来就可以把该纹理ID渲染到屏幕上去了，但是在这里还要再啰唆一句，因为该纹理ID的宽和高其实就是摄像头捕捉过来的高和宽（因为我们做了一个90度或者270度的旋转），目标是要将其渲染到SurfaceView上面去，但是如果Java层为我们提供的SurfaceView的宽高和处理过后的该纹理ID的宽高不一致，那么这一帧图像就会出现压缩或者拉伸的问题，所以在渲染到屏幕上的时候需要进行一个自适配，让纹理按照屏幕比例自动填充。

首先来看一下前面的纹理坐标，x从0.0到1.0就说明要把纹理的x轴方向全都绘制到物体表面（整个SurfaceView）上去，而如果我们只想绘制一部分，比如中间的一半，那么就可以将x轴的坐标写成0.25到0.75，相同的原则一样被应用到y轴上。那么这个0.25和0.75是如何出来的呢？答案很简单，要想不被拉伸，那么SurfaceView的宽高比例和纹理的宽高比例就应该是相同的。假设这一张纹理的宽为texWidth，纹理的高为texHeight以及物体的宽为screenWidth，物体的高为

screenHeight，且无论是宽还是高，都是float类型的，那么就可以利用下面的公式来完成自动填充的坐标计算：

```
float textureAspectRatio = texHeight / texWidth;
float viewAspectRatio = screenHeight / screenWidth;
float xOffset = 0.0f;
float yOffset = 0.0f;
if(textureAspectRatio > viewAspectRatio){
    // Update Y Offset
    int expectedHeight = texHeight*screenWidth/texWidth+0.5f;
    yOffset = (expectedHeight - screenHeight) / (2 * expectedHeight);
} else if(textureAspectRatio < viewAspectRatio){
    // Update X Offset
    int expectedWidth = texHeight * screenWidth / screenHeight + 0.5);
    xOffset = (texWidth - expectedWidth)/(2*texWidth);
}
```

计算得到的xOffset与yOffset分别用于在纹理坐标中替换掉0.0的位置，利用1.0-xOffset以及1.0-yOffset来替换掉1.0的位置，最终将得到一个纹理坐标矩阵如下：

```
GLfloat textureCoordNoRotation[8] = {
    xOffset,            yOffset,
    1.0 - xOffset,      yOffset,
    xOffset,            1.0 - yOffset,
    1.0 - yOffset,      1.0 - yOffset
};
```

至此，摄像头预览流程就可以随着摄像头所采集的各帧图像正常地绘制下去了，从而实现整个预览的过程。

当用户切换摄像头的时候，可以向Native层发送一个指令，Native层会在渲染线程中关闭当前摄像头，然后重新打开另外一个摄像头，并配置参数，以及设置预览的Surface-Texture，最后调用开始预览方法，这样就可以切换成功，用户看到的就是摄像头切换之后的预览画面了。

当我们最终关闭预览时，首先要停止整个渲染线程，然后释放掉所建立的Surface-Texture，之后再将摄像头的PreviewCallback设置为null，最终关闭并且释放摄像头。整个流程代码如下：

```
if (mCameraSurfaceTexture != null) {
    mCameraSurfaceTexture.release();
```

```
        mCameraSurfaceTexture = null;
    }
    if (null != mCamera) {
        mCamera.setPreviewCallback(null);
        mCamera.release();
        mCamera = null;
    }
```

至此，摄像头预览部分已经全部讲解完毕，这是十分重要的，对于后面搭建整个录制视频的项目以及后续视频直播的项目来说，这都是最基础的部分，所以请读者好好熟悉这一部分。

本节的代码实例在代码仓库中的CameraPreview项目中，运行项目之后进入摄像头预览界面，可以看到摄像头的预览，点击右上角的切换摄像头按钮可以进行摄像头的切换操作。

6.2.2 iOS平台的视频画面采集

在iOS平台上使用Camera来采集画面要比在Android平台上简单得多，毕竟这里仅用一种语言就可以实现，不必像Android平台那样需要在Java层和Native层之间进行频繁的切换，但是要想设计出一个优秀的可扩展的架构也不是一件容易的事情。本节中笔者会带领大家设计并实现一个基于摄像头采集，最终用OpenGL ES渲染到UIView上，并且可以支持后期视频特效处理，以及编码视频帧的架构。首先来看一下整体架构图，如图6-10所示。

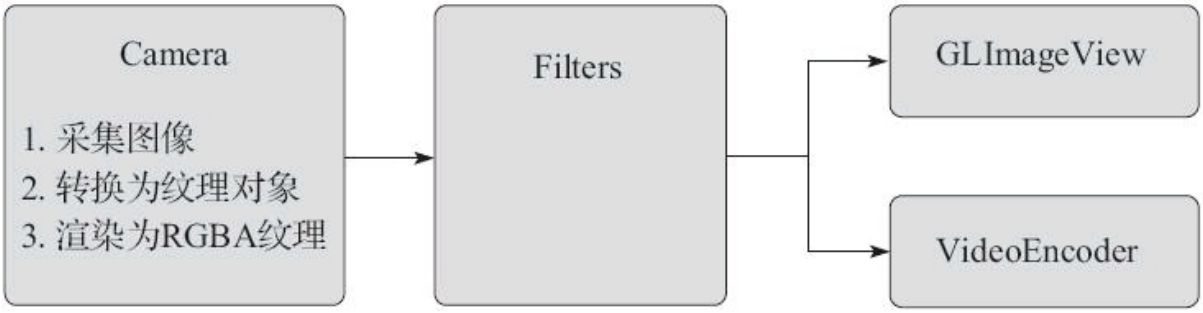


图 6-10

图6-10中所描述的是，首先利用系统的Camera采集出YUV的图像，然后渲染到一个纹理对象上，为了方便扩展，将该纹理对象作为Filter的输入纹理对象，调用Filter进行图像处理（使用OpenGL ES来处理），该Filter还会有一个输出纹理对象，可将该输出纹理对象作为下一级组件GLImageView或者将来扩展出来的组件VideoEncoder的输入纹理对象，而这两个组件将分别用于进行屏幕渲染和编码操作。这样就可以实现预览的场景，并且还可以满足我们将来要做编码以及做图像处理的需求。

接下来分析一下该架构。

要知道，每个节点的处理都是一个OpenGL的渲染过程，所以为每个节点建立一个Program是必不可少的，我们不可能在每个节点中都去编写编译Shader、链接Program等基本操作，所以要先抽取出一个类——ELImageProgram（EL是整个项目的前缀），用于把OpenGL的Program的构建、查找属性、使用等这些操作以面向对象的形式封装起来，每个节点都会有一个该类的引用实例。

观察每个节点的输入，会发现它们都是一个纹理对象（实际上是一个纹理ID），实际渲染到一个目标纹理对象的时候，还需要建立一个帧缓存对象，并且还要将该目标纹理对象Attach到这个帧缓存对象上，所以这里建立一个类ELImageTextureFrame，用于将纹理对象和帧缓存对象的创建、绑定、销毁等操作以面向对象的方式封装起来，使得每个节点的使用都更加方便。

要想使用OpenGL ES，必须要有上下文以及关联的线程，之前也提到过iOS平台为OpenGL ES提供了EAGL作为OpenGL ES的上下文。在整个架构的所有组件中，要针对编码器组件单独开辟一个线程，因为我们不希望它阻塞预览线程，从而影响预览的流畅效果，所以它也需要一个单独的OpenGL上下文，并且需要和渲染线程共享OpenGL上下文（两个OpenGL ES线程共享上下文或者共享一个组，则代表可以互相使用对方的纹理对象以及帧缓存对象），只有这样，在编码线程中才可以正确访问到预览线程中的纹理对象、帧缓存对象。

继续观察图6-8，会发现Camera和Filter这些节点是可以输出纹理对象的，也就是它们的目标纹理对象要作为后一级节点的输入纹理对象，另外观察架构图还会发现，Filter、GLImageView以及VideoEncoder需要上一级节点提供输入的纹理对象。发现了这两个特点之后，我们就可以抽象出以下两个规则。

·规则一：凡是需要输入纹理对象的节点都是Input类型。

·规则二：凡是需要向后级节点输出纹理对象的节点都是Output类型。

基于规则一，我们可以定义ELImageInput这样一个Protocol，由于需要别的组件为它提供输入所需的纹理对象，所以该Protocol里面定义了两个方法，第一个方法是设置输入纹理对象：

```
- (void)setInputTexture:(ELImageTextureFrame *)textureFrame;
```

节点中的Filter、GLImageView以及VideoEncoder都属于ELImageInput的类型，所以应该实现该方法，在该方法的实现中应该将输入纹理对象保存为一个全局变量。

此外，这些节点还有一个共同点，那就是都需要进行渲染操作，所以接下来第二个方法是执行渲染操作：

```
- (void)newFrameReadyAtTime:(CMTime)frameTime timimgInfo:
    (CMSampleTimingInfo)timimgInfo;
```

这是上一级节点（实际上是一个Output节点）处理完毕之后会调用的方法，在这个方法的实现中可完成渲染操作。

基于规则二，再来建立新类ELImageOutput，其中Camera、Filter节点需要继承自该类，该类描述为凡是继承自它的节点都可以向自己的后级节点输出目标纹理对象，所以我们首先建立两个属性：一个是后级节点列表；另一个是目标纹理对象。代码如下：

```
ELImageTextureFrame *outputTexture;
NSMutableArray *targets;
```

为什么后级节点是列表类型？因为后级节点有可能包含了多个目标对象，像Filter节点，既要输出给GLImageView又要输出给VideoEncoder，而这个targets里面的对象又是什么呢？实际上就是之前定义的协议ELImageInput类型的对象，这是因为Output节点的后级节点肯定是一个Input的节点。另外该类还得提供增加和删除目标节点的方法：

```
- (void)addTarget:(id<ELImageInput>)target;
- (void)removeTarget:(id<ELImageInput>)target;
```

这两个方法可用于操作targets属性，在每一个真正继承ELImageInput类的节点执行渲染过程结束之后，都会遍历targets中所有的目标节点（即ELImageInput）执行设置输出纹理对象方法，并执行下一个节点的渲染过程。代码如下：

```
// Do Render Work
for (id<ELImageInput> currentTarget in targets){
    [currentTarget setInputTexture:outputTextureFrame];
    [currentTarget newFrameReadyAtTime:frameTime timimgInfo:timimgInfo];
}
```

基于上面的分析，我们可以画出节点的类图关系，如图6-11所示。图中的类ELImage-Context是需要重点讲解的一个地方，由于OpenGL ES渲染操作必须执行在绑定了OpenGL上下文的线程中，而且由于其对于客户端代码的调用，需要在调用线程和OpenGL ES的线程之间进行频繁的切换，所以在该类中提供一个静态方法可以获得具有OpenGL上下文的渲染线程，让一些OpenGL ES渲染操作可以在该线程中直接执行，具体的代码如下：

```
+ (void)useImageProcessingContext;
{
    [[ELImageContext sharedImageProcessingContext] useAsCurrentContext];
}
- (void)useAsCurrentContext;
{
    EAGLContext *imageProcessingContext = [self context];
    if ([EAGLContext currentContext] != imageProcessingContext)
    {
        [EAGLContext setCurrentContext:imageProcessingContext];
    }
}
```

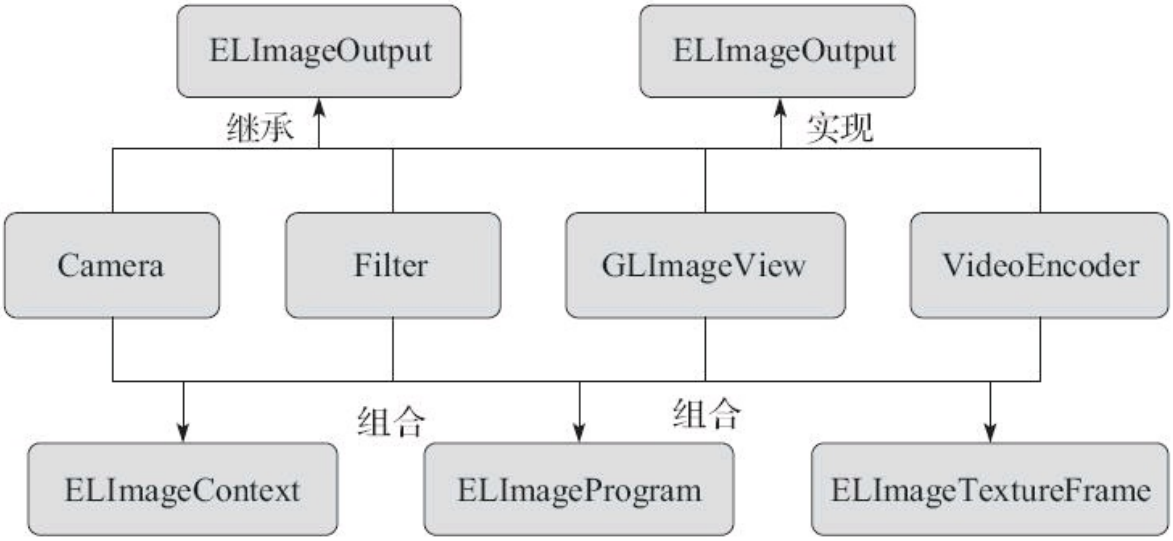


图　6-11

由于GLImageView在前面的章节中已经实现过了，这里就不再赘述，读者可以直接到实例中查看源码，本节中只会实现Camera，6.4.3节将会实现VideoEncoder，将在第9章中再去实现Filter。那么接下来就来学习Camera的实现，这将分为两部分来进行讲解，第一部分是摄像头的

配置，第二部分是摄像头采集到的YUV数据应该如何构转换为纹理对象，以便可以输出给后续的节点进行处理与渲染。

1.摄像头配置

既然要使用摄像头，就要使用AVCaptureSession，因为在iOS平台开发中只要是与硬件相关的都要从会话开始进行配置：

```
AVCaptureSession* captureSession;
captureSession = [[AVCaptureSession alloc] init];
```

然后需要配置AVCaptureDeviceInput，其实该变量就是用于指定需要使用哪一个摄像头，比如，以下是使用前置摄像头的代码：

```
AVCaptureDevice * captureDevice = nil;
NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
for (AVCaptureDevice *device in devices) {
    if ([device position] == AVCapturePositionFront) {
        captureDevice = device;
    }
}
captureInput = [[AVCaptureDeviceInput alloc] initWithDevice:
        captureDevice error:nil];
```

接着需要配置AVCaptureVideoDataOutput，其实该变量是用于配置如何接收摄像头采集的数据的：

```
dispatch_queue_t dataCallbackQueue;
dataCallbackQueue = dispatch_queue_create("dataCallbackQueue",
        DISPATCH_QUEUE_SERIAL);
captureOutput = [[AVCaptureVideoDataOutput alloc] init];
[_captureOutput setSampleBufferDelegate:self queue:dataCallbackQueue];
```

如上述代码所示，在构建出captureOutput实例之后，要想获取摄像头采集的数据，就需要传入类型为AVCaptureVideoDataOutputSampleBufferDelegate的实例和一个dispatch_queue，所以这里分配了线程队列以及实现该类本身所要求的Delgate，然后配置给captureOutput实例对象。接下来需要设置像素格式，摄像头默认使用的表示格式为YUVFullRange类型，FullRange表示

YUV的取值范围是从0～255，而另外一种类型YUVVideoRange则是为了防止溢出，将YUV的取值范围设置为从16～235。不同的Range对于后续将YUV格式转换为RGBA格式的时候所使用的矩阵（转换公式）是不同的，所以这里需要根据所支持的格式来对摄像头进行设置，并且记录下来以供后面确定RGBA的转换矩阵所用。

现在，将captureInput实例和captureOutput实例配置到CaptureSession中，代码如下：

```
if ([self.captureSession canAddInput:self.captureInput]) {
        [self.captureSession addInput:self.captureInput];
    }
if ([self.captureSession canAddOutput:self.captureOutput]) {
        [self.captureSession addOutput:self.captureOutput];
}
```

调用captureSession设置分辨率的方法，常见的分辨率及其设置代码如下：

```
NSString* highResolution = AVCaptureSessionPreset1280x720;
NSString* lowResolution = AVCaptureSessionPreset640x480;
[_captureSession setSessionPreset:[NSString stringWithString: highResolution]
```

调用CaptureSession的beginConfiguration方法，配置整个摄像头会话。最后取出capture-Output中的AVCaptureConnection来配置摄像头输出的方向，这一点是非常重要的，如果不配置该参数，那么摄像头默认输出的就是横向的图片，设置为纵向图片输出的代码如下所示：

```
conn.videoOrientation = AVCaptureVideoOrientationPortrait;
```

当然，也可以为CaptureInput设置帧率等信息，这里将不再赘述，大家可以参考代码仓库中的源码进行设置。

2.摄像头采集数据处理

上文已经为本类实现了AVCaptureVideoDataOutputSampleBufferDelegate接口（协议），由于我

们要获取摄像头采集的数据，所以这里需重写该Protocol里面约定的方法，也就是摄像头用来输出数据的方法，签名如下：

```
-(void) captureOutput:(AVCaptureOutput*)captureOutput
        didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
        fromConnection:(AVCaptureConnection*)connection
```

上述方法会返回具体是哪一个captureOutput以及connection，但是最重要的还是CMSample-Buffer类型的sampleBuffer，其中实际存储着摄像头采集到的图像，CMSampleBuffer结构体由以下三个部分组成。

·CMTime：代表了这一帧图像的时间。

·CMVideoFormatDescription：代表了对于这一帧图像格式的描述。

·CVPixelBuffer：代表了这一帧图像的具体数据。

在该回调函数中，需要把摄像头采集到的YUV数据转换为纹理对象，所以要进行OpenGL的渲染操作，前文中提到过一个问题，iOS平台不允许App进入后台的时候还进行OpenGL的渲染操作，如果App依然进行渲染操作的话，那么系统就会强制杀掉该App。通用的处理方式就是之前在播放器中也用到过的方式，即为App注册applicationWillResignActive和applicationDidBecomeActive的通知，在这两个方法中将该类中的实例变量shouldEnableOpenGL设置为NO和YES。而在回调函数中，处理代码如下：

```
-(void) captureOutput:(AVCaptureOutput*)captureOutput didOutputSampleBuffer:
        (CMSampleBufferRef)sampleBuffer fromConnection:
        (AVCaptureConnection*)connection
{
    if (self.shouldEnableOpenGL) {
        if (dispatch_semaphore_wait(_frameRenderingSemaphore,
                DISPATCH_TIME_NOW) != 0) {
            return;
        }

        CFRetain(sampleBuffer);
        runAsyncOnVideoProcessingQueue(^{
            [self processVideoSampleBuffer:sampleBuffer];
            CFRelease(sampleBuffer);
            dispatch_semaphore_signal(_frameRenderingSemaphore);
        });
    }
```

```
}
```

上述代码中，首先要判断该应用程序进入后台的布尔型变量，如果是NO则不执行任何操作；如果是YES则需要保证上一次渲染已经结束了（通过dispatch_semaphore的wait来确定），否则丢弃当前帧，然后使用CFRetain锁定该sampleBuffer，因为真正操作sample-Buffer的地方是在OpenGL ES线程中，所以这里必须先将其保留（Retain）住，等这一次OpenGL ES的渲染操作执行完毕之后，再使用CFRelease释放掉该sampleBuffer，最后向semaphore发送一个signal指令表明可以继续处理下一帧视频帧。那么剩下的就是如何将该sampleBuffer渲染成为一个纹理对象，并且调用后续的targets进行渲染。

首先，取出该sampleBuffer中的真实数据，即CVPixelBuffer类型的实例，然后拿出该CVPixelBuffer里面YUV格式的矩阵key，该矩阵key用于判断转换格式是ITU601格式还是ITU709格式。ITU601是标清电视（SDTV）的标准，而ITU709是超清电视（HDTV）的标准，YUV转换为RGB的矩阵是根据标清标准或者超清标准，以及前面提到的YUVFullRange和YUVVideoRange共同决定的。其中ITU601标准分为YUVFullRange和YUVVideoRange，而ITU709标准不区分Rang，得到矩阵之后，在渲染过程中将使用该矩阵来做YUV格式到RGB格式的转换，下面先来看一下三个矩阵：

```
GLfloat colorConversion601Default[] = {
    1.164,  1.164,   1.164,
    0.0,    -0.392,  2.017,
    1.596, -0.813,   0.0,
};
GLfloat colorConversion601FullRangeDefault[] = {
    1.0,    1.0,     1.0,
    0.0,    -0.343, 1.765,
    1.4,    -0.711, 0.0,
};
GLfloat colorConversion709Default[] = {
    1.164,  1.164,   1.164,
    0.0,    -0.213,  2.112,
    1.793, -0.533,   0.0,
};
```

然后调用ELImageContext为当前线程设置OpenGL上下文，由于ELVideoCamera类继承自ELImageOutput，所以要根据宽高分配出一个outputTexture，并且绑定该Texture的Frame-Buffer（代表该渲染过程的

目标就是这个纹理对象）。在渲染之前，需要将CVPixelBuffer中的YUV数据关联到两个纹理ID上，如果是在其他平台上，则只能通过OpenGL ES提供的glTexImage2D将内存中的数据上传到显卡的一个纹理ID之上，但是这种内存和显存之间的数据交换效率是很低的，在iOS平台上的CoreVideo这个framework中提供了CVOpenGLESTextureCacheCreateTextureFromImage方法，可以使得整个交换过程更加高效，因为CVPixelBuffer是YUV数据格式的，所以可以分配以下两个纹理对象：

```
CVOpenGLESTextureRef luminanceTextureRef = NULL;
CVOpenGLESTextureRef chrominanceTextureRef = NULL;
```

之后，必须锁定CVPixelBuffer，因为CVPixelBuffer这个API在官方文档上描述的是像素数据存储在主内存中。对于该主内存，笔者个人理解应该不是普通操作的内存，所以需要在使用该内存区域之前先锁定该对象，在使用完毕之后进行解锁。以下代码可锁定该PixelBuffer：

```
CVPixelBufferLockBaseAddress(pixelBuffer, 0);
```

将其中的Y通道部分的内容上传到luminanceTexture中：

```
CVOpenGLESTextureCacheCreateTextureFromImage(
        kCFAllocatorDefault, coreVideoTextureCache, pixelBuffer,
        NULL, GL_TEXTURE_2D, GL_LUMINANCE, bufferWidth,
        bufferHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE, 0,
        &luminanceTextureRef);
```

这里传入了pixelBuffer以及格式GL_LUMINANCE，还需要传入宽和高，这样该API内部就知道应该访问pixelBuffer的哪一部分数据了，当然，还必须传入一个纹理缓存，并从该纹理缓存中读取出所创建的纹理对象，创建该纹理缓存的代码如下：

```
CVOpenGLESTextureCacheCreate(kCFAllocatorDefault, NULL, context, NULL,
                &coreVideoTextureCache)
```

只有传入OpenGL上下文才可以创建出该纹理缓存，然后就可以通

过CVOpenGLESTexture-GetName方法来获取luminanceTextureRef的纹理ID了，使用该纹理ID就可以和普通纹理对象一样进行操作了。以下代码可将UV通道部分上传到chrominanceTextureRef里面去：

```
CVOpenGLESTextureCacheCreateTextureFromImage(
        kCFAllocatorDefault, coreVideoTextureCache, pixelBuffer,
        NULL, GL_TEXTURE_2D, GL_LUMINANCE_ALPHA, bufferWidth/2,
        bufferHeight/2, GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, 1,
        &chrominanceTextureRef)
```

因为U和V各占width*height的四分之一个像素，即每四个像素会有一个U和一个V，所以可使用GL_LUMINANCE_ALPHA来表示，即将U放到Luminance部分，将V放到Alpha部分，理解这一点是非常重要的，因为这关乎后面在FragmentShader中如何拿到正确的YUV数据。

接下来就是整个渲染过程了，其实在渲染过程中有两点是需要注意的：第一点就是物体坐标和纹理坐标的确定；第二点就是在FragmentShader中如何将YUV转换为RGBA的表示格式。

首先来看物体坐标和纹理坐标的确定，物体坐标是固定的，具体如下：

```
GLfloat squareVertices[8] = {
        -1.0,  -1.0,        // 物体左下角
         1.0,   -1.0,       // 物体右下角
        -1.0,    1.0,       // 物体左上角
         1.0,    1.0        // 物体右上角
};
```

而纹理坐标就比较特殊了，首先有一点（这点在前面的章节中已经讲过很多次了），就是OpenGL纹理坐标系和计算机坐标系不同，所以默认情况下，纹理坐标如下：

```
GLfloat textureCoords[8] = {
        0.0，1.0，
        1.0，1.0，
        0.0，0.0，
        1.0，0.0
};
```

进行旋转以及镜像的时候都是根据上述纹理坐标来实施的。图6-12是前置摄像头默认为我们显示的图像。

图　6-12

对图于6-12，需要先按照顺时针旋转90度，由于是前置摄像头，所以还得做一个镜像处理，其纹理坐标具体如下：

```
GLfloat textureCoords[8] = {
        1.0, 0.0,
        1.0, 1.0,
        0.0, 0.0,
        0.0, 1.0
};
```

如果是后置摄像头，那么所做的操作如图6-13所示。

图　6-13

纹理坐标如下所示：

```
GLfloat textureCoords[8] = {
        1.0, 1.0,
        1.0, 0.0,
        0.0, 1.0,
```

```
        0.0, 0.0
};
```

这里需要特别注意就是，由于对纹理做了90度的旋转，所以目标纹理对象（output-Texture）的宽和高就是CVPixelBuffer的高和宽了，这里不要再弄错。

上述处理过程是默认摄像头给出的图像处理过程，还记得前面我们为摄像头做了一个特殊的设置吗？即为AVCaptureConnection设置videoOrientation参数，默认是横向视频输出，当将该参数设置为Portrait时，即要求摄像头竖直方向的视频输出。这时候目标纹理对象（outputTexture）的宽和高就是CVPixelBuffer的宽和高了，后置摄像头采集出来的图像操作如图6-14所示。

图　6-14

所对应的纹理坐标为：

```
GLfloat textureCoords[8] = {
        0.0, 1.0,
        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0
};
```

而前置摄像头由于镜像的原因，所以处理过程如图6-15所示。

图　6-15

此时的纹理坐标与后置摄像头的每一个坐标的X点正好相反：

```
GLfloat textureCoords[8] = {
        1.0, 1.0,
        0.0, 1.0,
        1.0, 0.0,
        0.0, 0.0
};
```

其次，来看一下如何在FragmentShader中将YUV转换为RGBA格式。可能有读者会有一个疑问，为什么非要转换为RGBA格式呢？因为在OpenGL中纹理的默认格式都是RGBA格式的，并且也要为后续的纹理处理以及渲染到屏幕上打下基础，最终编码器也是以RGBA格式为基础进行转换和处理的。在播放器的章节中已做过一次YUV转RGB的操作，但是由于使用了CoreVideo这个framework下的快速上传，纹理变得不一样了，下面就来看一下具体的FragmentShader：

```
varying highp vec2 textureCoordinate;
uniform sampler2D luminanceTexture;
uniform sampler2D chrominanceTexture;
uniform mediump mat3 colorConversionMatrix;
void main(){
    mediump vec3 yuv;
    lowp vec3 rgb;
    yuv.x = texture2D(luminanceTexture, textureCoordinate).r;
    yuv.yz = texture2D(chrominanceTexture, textureCoordinate).
    ra - vec2(0.5, 0.5);
    rgb = colorConversionMatrix * yuv;
    gl_FragColor = vec4(rgb, 1);
}
```

其中，textureCoordinate就是纹理坐标，而两个sampler2D类型就是我们千辛万苦从CV-PixelBuffer里面上传到显存中的纹理对象，而3×3的矩阵就是前面根据像素格式以及是否为FullRange选择的变换矩阵。在前面Y使用的是GL_LUMINANCE格式，所以这里是使用texture2D函数读取出像素点，然后访问像素的r通道就可以获得Y通道的值了。而UV通道使用的是格式GL_LUMINANCE_ALPHA，所以这里是通过texture2D函数获取像素点，然后访问r和a通道作为UV的值，但是为什么UV的值要减去0.5呢？换算为0-255就是减去127，这是因为UV是色彩分量，当整张图片都是黑白的时候，UV分量是默认值127，所以这里需要先减去127，然后再转换为RGB，否则会出现色彩不匹配的错误。最终可以使用传递进来的转换矩阵乘以YUV得到该像素点的RGBA表示格式。

6.3 音频的编码

本节将讨论音频的编码，第1章中已经分析过音频编码的原理与发展历史，本章将以实践为主，利用编码器完成编码场景下的编码需求。MP3格式是兼容性最好的格式，而AAC在低码率（128Kbit/s以下）场景下，其音频品质大大超过MP3，并且在移动平台上，无论是单独的音频编码，还是视频编码中的音频流部分，使用得最广泛的都是AAC的编码格式。本节将通过软件编码库（libfdk_aac这个第三方开源库）以及各个平台提供的硬件编码库来编码AAC码流。对于一些其他的编码格式，比如适用于VOIP通话的Ogg格式，适用于语言聊天的AMR格式等在本节中不会涉及，但是这里使用的软件编码方式是以FFmpeg平台为基础的，所以后期无论想用其他的什么格式，都可以自行配置编码库来实现。本节的输入就是6.1节的录音器采集的音频数据（其实就是普通的PCM流），当然读者也可以自行尝试使用第5章中解码器模块的输出作为编码器的输入，其实这就是一个后处理（post-processing）过程的实践，后面还会有一个单独的项目来完成这种用户的后处理场景。

6.3.1 libfdk_aac编码AAC

软件编码AAC，将基于FFmpeg的API来编写，而不像第2章那样直接使用LAME库的API来编码MP3。这样做的好处是，只需要编写一份音频编码的代码即可，对于不同的编码器，只需要调整相应的编码器ID或者编码器Name，就可以编码出不同格式的音频文件。当然，既然要使用第三方库libfdk_aac编码AAC文件，那么必须在做交叉编译的时候将libfdk_aac库编译到FFmpeg中去。可编写一个C++的类，命名为audio_encoder，然后对外提供三个接口，分别是初始化、编码以及销毁方法，并且要求该类可以同时运行在Android平台和iOS平台之上。首先来看一下初始化接口：

```
int init(int bitRate, int channels, int sampleRate, int bitsPerSample,
    const char* aacFilePath, const char * codec_name);
```

对于其中的传入参数，说明如下。

首先是比特率，也就是最终编码出来的文件的码率，接着是声道数、采样率，这两个将不再赘述，然后是最终编码的文件路径，最后是编码器的名字。其实现步骤具体如下。

首先调用方法av_register_all（）将所有的封装格式以及编解码器注册到FFmpeg框架中；然后调用avformat_alloc_output_context2方法传入输出文件格式，分配出上下文，即分配出封装格式；之后调用avio_open2方法传入AAC的编码路径，相当于打开文件连接通道。前面分析过FFmpeg的源码，通过上述两行代码可以确定出Muxer与Protocol，然后就是分配Codec的相关内容了，所以接下来要为AVFormatContext上下文填充一轨AVStream：

```
audioStream = avformat_new_stream(avFormatContext, NULL);
```

现在，要为audioStream的Codec属性填充内容了。Codec属性是一个AVCodecContext的结构体类型，需要为该结构体填充如下几个属性，首先是codec_type，赋值为AVMEDIA_TYPE_AUDIO，代表其是音频类

型；其次是bit_rate、sample_rate、channels等基本属性，然后是channel_layout（其表示的意义与channels是一样的，只不过可选值是两个常量，分别是AV_CH_LAYOUT_MONO代表单声道、AV_CH_LAYOUT_STEREO代表立体声），最后也是最重要的sample_fmt，代表了如何数字化表示采样，使用的是AV_SAMPLE_FMT_S16，即用一个short来表示一个采样点，这样就把AVCodecContext结构体构造完成了。

下面要准备最重要的编码器了，通过调用avcodec_find_encoder_by_name函数来找出对应的编码器，然后调用方法avcodec_open2来为该编码器上下文打开这个编码器，接下来为编码器指定frame_size的大小，一般指定1024作为一帧的大小，至此我们就把编码器部分给分配好了。

在此需要注意的是，某些编码器只允许特定格式的PCM作为输入源，比如声道数、采样率、表示格式（比如LAME编码器就不允许SInt16的表示格式）的要求，所以需要构造一个重采样器来将PCM数据转换为可适配编码器输入的PCM数据，即前面讲解过的需要将输入的声道、采样率、表示格式和输出的声道、采样率、表示格式传递给初始化方法，然后分配出重采样上下文SwrContext。此外，还要分配一个AVFrame类型的inputFrame，作为客户端代码输入的PCM数据存放的地方，这里需要知道inputFrame分配的buffer的大小，如上一步所述，默认一帧的大小是1024，所以对应的buffer（按照uint8_t类型作为一个元素来分配）大小就应该是：

```
bufferSize = frame_size * sizeof(SInt16) * channels;
```

当然无需自己进行计算，可以调用FFmpeg提供的方法av_samples_get_buffer_size来帮助开发者计算。其实这个方法内部的计算公式就是上面所列的公式。如果需要进行重采样处理，那就需要额外分配一个重采样之后的AVFrame类型的swrFrame，作为最终得到结果的AVFrame。

在初始化方法的最后，需要调用FFmpeg提供的方法avformat_write_header将该音频文件的Header部分写进去，然后记录一个标志isWriteHeaderSuccess，使其为true，因为后续在销毁资源的阶段需要根据该标志来判断是否调用write trailer方法，即写入文件尾部的方

法，否则会造成Crash，这在前面分析FFmpeg源码的时候就已经讲解过了。

接下来看一下提供的第二个接口方法：

```
void encode(byte* buffer, int size);
```

这里传入的参数是uint8_t类型的指针，以及这块内存所表示的数据长度，具体的实现是将该buffer填充入inputFrame，由于前面已经知道了每一帧buffer需要填充的大小是多少，所以这里可以利用一个while循环来做数据的缓冲，一次性填充到AVFrame中去，然后调用编码方法avcodec_encode_audio2，该方法会将编码好的数据放入AVPacket的结构体中，紧接着就可以将该AVPacket类型的结构体写到文件中去了，而调用av_interleaved_write_frame方法，则可以将该packet输出到最终的文件中去。

最后，来看一下第三个接口方法：

```
void destroy();
```

上述代码所述方法需要销毁前面所分配的资源以及打开的连接通道。

如果初始化了重采样器，那么就销毁重采样的数据缓冲区以及重采样上下文；然后销毁为输入PCM数据分配的AVFrame类型的inputFrame，再判断标志isWriteHeaderSuccess变量，决定是否需要填充duration以及调用方法av_write_trailer，最终关闭编码器以及连接通道。

这个类写完之后，就可以集成到Android和iOS平台了，外界控制层需要初始化该类，然后负责读写文件调用encode方法，最终调用销毁资源的方法。

本节的代码实例，Android工程是Android代码仓库中的FDKAACAudioEncoder工程，iOS工程是iOS的代码仓库中的FDKAACEncoder工程，两个工程都是将PCM文件编码为一个AAC的文件。运行Android工程之前，需要将resource目录下面的PCM文件放入

SDCard目录下的根目录下面，最终可以播放编码后的AAC文件进行试听。

6.3.2 Android平台的硬件编码器MediaCodec

本节就来看一下如何使用Android平台提供的MediaCodec编码AAC。使用MediaCodec编码AAC对Android系统是有要求的，必须是4.1系统以上，即要求Android的版本代号在Jelly_Bean以上。MediaCodec是Android系统提供的硬件编码器，它可以利用设备的硬件来完成编码，从而大大提高编码的效率，还可以降低电量的使用，但是其在兼容性方面不如软件编码好，因为Android设备的碎片化太严重，所以读者可以自己衡量在应用中是否使用Android平台的硬件编码特性。

第3章中讲解过AAC编码格式，其中有一种是ADTS封装格式，另外一种就是裸的AAC的封装格式。不论这里所说的是MediaCodec编码出来的AAC，还是在6.3.3节将要讲解的AudioToolbox编码出来的AAC，都是裸的AAC，即AAC的原始数据块，一个AAC原始数据块的长度是可变的，对原始帧加上ADTS头进行封装，就形成了ADTS帧。ADTS的全称是Audio Data Transport Stream，是AAC音频的传输流格式，通常我们将得到的AAC原始帧加上ADTS头进行封装后写入文件，该文件使用常用的播放器即可播放，这是个验证AAC数据是否正确的方法。进行封装之前，需要了解相关的参数，如采样率、声道数、原始数据块的长度等。下面将AAC原始数据帧加工为ADTS格式的帧，根据相关参数填写组成7字节的ADTS头。下面就来分配一下这7个字节：

```
int adtsLength = 7;
char *packet = malloc(sizeof(char) * adtsLength);
```

其中，前两个字节是ADTS的同步字：

```
packet[0] = (char)0xFF;
packet[1] = (char)0xF9;
```

紧接着第三个字节是编码的Profile、采样率下标（注意是下标，而不是采样率）、声道配置（注意是声道配置，而不是声道数）、数据长度的组合（注意packetLen是原始数据长度加上ADTS头的长度）：

```
int profile = 2; // AAC LC
int freqIdx = 4; // 44.1kHz
int chanCfg = 2;
packet[2] = (byte) (((profile - 1) << 6) + (freqIdx << 2) + (chanCfg >> 2));
packet[3] = (byte) (((chanCfg & 3) << 6) + (packetLen >> 11));
packet[4] = (byte) ((packetLen & 0x7FF) >> 3);
packet[5] = (byte) (((packetLen & 7) << 5) + 0x1F);
```

其中具体的编码Profile、采样率的下标以及声道数都可以从下方这个链接中查到相关的所有表示：

```
https:// wiki.multimedia.cx/index.php?title=MPEG-4_Audio#Channel_Configuratic
```

最后一个字节也是固定的：

```
packet[6] = (byte) 0xFC
```

至此，ADTS头就拼接上了，对于编码出一个可以播放的AAC文件来讲，这是非常重要的，而对于MediaCodec以及6.3.3节将要讲到的AudioToolbox编码出来的AAC来说，都需要拼接上该ADTS头，最终文件就可以正确地播放出来了。

下面就来看看MediaCodec如何使用。类似于软件编码提供的三个接口方法，这里也提供三个接口方法，分别用于完成初始化、编码数据和销毁编码器的操作。首先来看一下初始化方法，先构造一个MediaCodec实例，通过该类的静态函数来实现。构造一个AAC的Codec，其代码如下：

```
MediaCodec mediaCodec = MediaCodec.createEncoderByType("audio/mp4a-latm");
```

其实该方法有点类似于前面所讲的FFmpeg中根据Codec的name来找出编码器。构造出该实例之后，就需要配置该编码器了，配置编码器最重要的是需要传递一个MediaFormat类型的对象，该对象中配置的是比特率、采样率、声道数以及编码AAC的Profile，此外，还需要配置输入Buffer的最大值，代码如下：

```
MediaFormat encodeFormat = MediaFormat.createAudioFormat(MINE_TYPE, sampleRat
        channels);
encodeFormat.setInteger(MediaFormat.KEY_BIT_RATE, bitRate);
encodeFormat.setInteger(MediaFormat.KEY_AAC_PROFILE, Media
        CodecInfo.CodecProfileLevel.AACObjectLC);
encodeFormat.setInteger(MediaFormat.KEY_MAX_INPUT_SIZE, 10 * 1024);
```

与FFmpeg中的配置编码器类似，上述代码也配置了编码器要求的输入，现在就将该对象配置到编码器内部：

```
mediaCodec.configure(encodeFormat, null, null,
    MediaCodec.CONFIGURE_FLAG_ENCODE);
```

最后一个参数代表需要配置一个编码器，而非解码器（如果是解码器则传递为0）。调用start方法，代表开启该编码器。

至此，编码器已经完全配置好了，打开编码器，现在开发者可以从MediaCodec实例中取出两个buffer，一个是inputBuffer，用于存放输入的PCM数据（类似于FFmpeg编码的AVFrame）；另外一个是outputBuffer，用于存放编码之后的原始AAC的数据（类似于FFmpeg编码的AVPacket），代码如下：

```
ByteBuffer[] inputBuffers = mediaCodec.getInputBuffers();
ByteBuffer[] outputBuffers = mediaCodec.getOutputBuffers();
```

到此，初始化方法已实现完毕，下面来看一下编码方法，MediaCodec的工作原理如图6-16所示，图6-16左边的Client元素代表要将PCM放到inputBuffer中的某个具体的buffer中去；图6-16右边的Client元素代表要将编码之后的原始AAC数据从outputBuffer中的某个具体的buffer中取出来，图6-16中，左边的小方块代表各个inputBuffer元素，右边的小方块则代表各个outputBuffer元素。

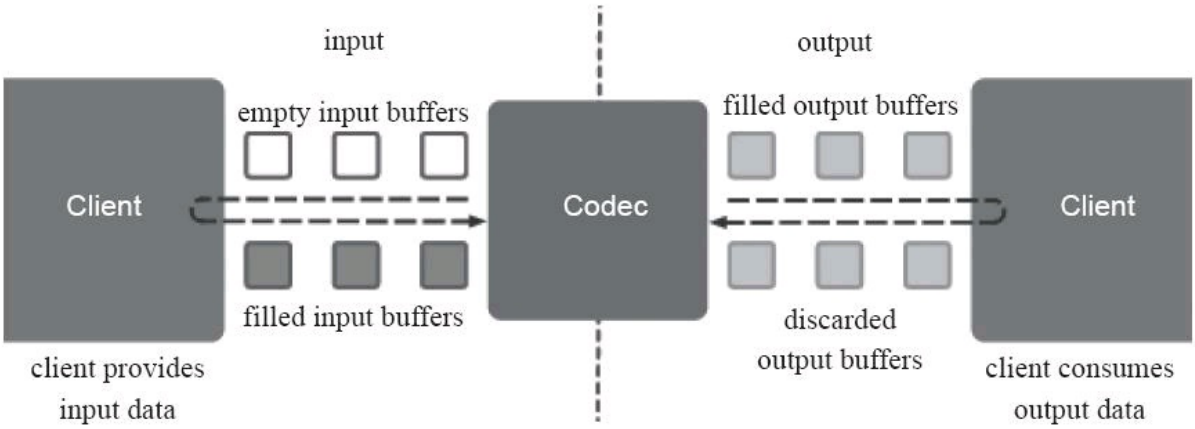


图　6-16

代码实现具体如下。

首先，从mediaCodec中读取出一个可以用来输入的buffer的Index，然后填充数据，并且把填充好的buffer发送给Codec：

```
int bufferIndex = codec.dequeueInputBuffer(-1);
if (inputBufferIndex >= 0) {
    ByteBuffer inputBuffer = inputBuffers[bufferIndex];
    inputBuffer.clear();
    inputBuffer.put(data);
    long time = System.nanoTime();
    codec.queueInputBuffer(bufferIndex, 0, len, time, 0);
}
```

然后，从Codec中读取出一个编码好的buffer的Index，通过Index读取出对应的output-Buffer，然后将数据读取出来，添加上ADTS头部，写文件，之后再把该outputBuffer放回到待编码填充队列中去：

```
BufferInfo info = new BufferInfo();
int index = codec.dequeueOutputBuffer(info, 0);
while (index >= 0) {
    ByteBuffer outputBuffer = outputBuffers[index];
    if (outputAACDelegate != null) {
        int outPacketSize = info.size + 7;
        outputBuffer.position(info.offset);
        outputBuffer.limit(info.offset + info.size);
        byte[] outData = new byte[outPacketSize];
        // 添加ADTS头信息
        addADTStoPacket(outData, outPacketSize);
        // 将编码得到的AAC数据取出到目标数组中，其中7代表偏移量
```

```
            outputBuffer.get(outData, 7, info.size);
            outputBuffer.position(info.offset);
            outputAACDelegate.outputAACPacket(outData);
        }
        codec.releaseOutputBuffer(index, false);
        index = mediaCodec.dequeueOutputBuffer(bufferInfo, 0);
    }
```

需要注意的是，上述代码中的第一行是构造出一个BufferInfo类型的对象，当开发者从Codec的输出缓冲区中读取一个buffer的时候，Codec会将该buffer的描述信息放入该对象中，后续会按照该对象中的信息来处理实际的内容。

最后来看一下销毁方法，使用完了MediaCodec编码器之后，就需要停止运行并释放编码器，代码如下：

```
    if (null != mediaCodec) {
        mediaCodec.stop();
        mediaCodec.release();
    }
```

外界调用端需要做的事情是先初始化该类，然后读取PCM文件并调用该类的编码方法，实现该类的Delegate接口，在重写的方法中将输出带有ADTS头的AAC码流直接写文件，最终编码结束之后，调用该类的停止编码方法。

本节的代码实例是代码仓库中的MediaCodecAudioEncoder工程，运行之前，需要将resource目录下的PCM文件放入SDCard目录下的根目录下面，编码结束之后，可以在对应的目录下获取AAC文件并播放试听。

6.3.3 iOS平台的硬件编码器AudioToolbox

本节将讲解iOS平台下的硬件编码器，可使用AudioToolbox下的Audio Converter Services来完成硬件编码，iOS平台提供的该服务从名字上来看就是一个转换服务，那么它能够提供哪几个方面的转换呢？

PCM到PCM，转换位深度、采样率以及表示格式，也包括交错存储还是平铺存储，这些与前面讲解的FFmpeg里的重采样器非常类似；最重要的是还可以做PCM到压缩格式的转换，所谓转换，在这种场景下其实就是可以做编码或者解码操作。可见iOS平台在多媒体方面提供的API是多么的强大，并且其兼容性也非常好，随着学习的深入，大家会逐渐了解到更多更方便的API，而本节就是利用它所提供的编码服务，将PCM数据编码为AAC格式的数据。

AudioToolbox中编码出来的AAC数据也是裸数据，在写入文件之前也需要添加上ADTS头信息，最终写出来的文件才可以被系统播放器播放，添加头信息的具体操作和6.3.2节中的操作是一样的，在此不再赘述。

类似于软件编码提供的三个接口方法，这里也提供了三个接口方法，分别用于完成初始化、编码数据和销毁编码器的操作。在介绍这三个方法的实现之前，先定义一个Protocol，命名为FillDataDelegate，需要客户端代码来实现该Delegate，这里面定义了三个方法，第一个方法的代码原型如下：

```
- (UInt32) fillAudioData:(uint8_t*) sampleBuffer bufferSize:(UInt32) bufferSi
```

当编码器（或者说转换器）需要编码一段PCM数据的时候，就通过该方法来调用客户端代码，让实现该Delegate的客户端代码来填充PCM数据。第二个方法的代码原型如下：

```
- (void) outputAACPakcet:(NSData*) data presentationTimeMills:
        (int64_t)presentationTimeMills error:(NSError*) error;
```

待编码器（或者说转换器）成功编码一段AAC的Packet之后，紧接着就是为这段数据添加ADTS头信息，然后通过上面的方法来调用客户端代码，由客户端代码来输出相关的数据。最后一个方法的代码原型如下：

```
- (void) onCompletion;
```

待编码器（或者说转换器）结束之后，调用客户端代码的这个方法，让客户端代码可以对自己的资源进行销毁与关闭等操作。

接下来介绍编码器类提供的三个接口方法的实现，首先是初始化方法的实现，之前笔者曾多次提到过，iOS平台提供了音视频的API，如果需要用到硬件Device相关的API，就需要配置各种Session；如果要用到与提供的软件相关的API，就需要配置各种Description以描述配置的信息，而在这里需要配置的Description就是前面介绍的AudioUnit部分所配置的Description。一说到这里相信大家就不再陌生了，我们需要分别配置一个input和一个output部分的Description，用于描述所提供数据的声道、采样率、表示格式、存储格式、编码方式等信息，具体代码如下：

```
// 构建InputABSD
AudioStreamBasicDescription inASBD = {0};
UInt32 bytesPerSample = sizeof (SInt16);
inASBD.mFormatID = kAudioFormatLinearPCM;
inASBD.mFormatFlags = kAudioFormatFlagIsSignedInteger | kAudioFormatFlagIsPac
inASBD.mBytesPerPacket = bytesPerSample * channels;
inASBD.mBytesPerFrame = bytesPerSample * channels;
inASBD.mChannelsPerFrame = channels;
inASBD.mFramesPerPacket = 1;
inASBD.mBitsPerChannel = 8 * channels;
inASBD.mSampleRate = inputSampleRate;
inASBD.mReserved = 0;
```

这里配置的input的Description是PCM格式的，表示格式是整数表示并且是交错存储的，这一点十分关键，因为需要按照设置的格式填充PCM数据，或者反过来说，客户端代码填充的PCM数据的格式是什么样的，这里配置给input描述的mFormatFlags就应该是什么样的，因为我们提供的数据就是交错存储的，所以填充的后续几个关键值都得乘以channels。

在iOS的音频流描述的配置中，最重要的就是存储格式和表示格式的配置，表示格式是指用整数或者浮点数表示一个sample；存储格式是指交错存储或非交错存储，输出或者输入数据都存储于AudioBufferList中的属性ioData中。假设声道是双声道的，那么对于交错存储（IsPacked）来讲，对应的数据格式如下：

```
ioData->mBuffers[0]: LRLRLRLRLRLR…
```

而对于非交错的存储（NonInterleaved）来讲，对应的数据格式如下：

```
ioData->mBuffers[0]: LLLLLLLLLLLL…
ioData->mBuffers[1]: RRRRRRRRRRRR…
```

这也就要求客户端代码需要按照配置的格式描述来填充或者获取数据，否则就会出现不可预知的问题。接下来再看一下，如果要转换成为AAC的格式，那么应该如何配置output的Description：

```
// 构造OutputABSD
AudioStreamBasicDescription outASBD = {0};
outASBD.mSampleRate = inASBD.mSampleRate;
outASBD.mFormatID = kAudioFormatMPEG4AAC;
outASBD.mFormatFlags = kMPEG4Object_AAC_LC;
outASBD.mBytesPerPacket = 0;
outASBD.mFramesPerPacket = 1024;
outASBD.mBytesPerFrame = 0;
outASBD.mChannelsPerFrame = inASBD.mChannelsPerFrame;
outASBD.mBitsPerChannel = 0;
outASBD.mReserved = 0;
```

这里面需要注意的是，mFormatID需要配置成AAC的编码格式，Profile需要配置为低运算复杂度的规格（LC），最后需要注意的一点是，配置一帧数据时，其大小为1024，这是AAC编码格式要求的帧大小。

至此，输入和输出的Description就配置好了。当然，还需要构造一个编码器类的描述，用于提供编码器的类型以及编码器的实现方式，因为是编码AAC，所以其所使用的编码器类型是：

kAudioFormatMPEG4AAC，编码的实现方式是使用兼容性更好的软件编码方式（虽然是软件编码方式，但是也是有硬件加速的）：kAppleSoftwareAudioCodecManufacturer。通过这两个输入可构造出一个编码器类的描述，它将告诉iOS系统开发者想要使用的到底是哪一个编码器。

有了上述的三个Description（一个是输入数据的描述，一个是输出数据的描述，还有一个是编码器的描述），调用如下方法就可以构造出一个AudioConverterRef实例了：

```
OSStatus status = AudioConverterNewSpecific(&inABSD, &outABSD, 1,
    codecDescription, &_audioConverter);
```

第三个参数1是指明要创建编码器的个数，最后一个参数就是我们想要构造的转码器实例对象，返回值是OSStatus类型的变量，如果返回的不是0，则表示出错了，如果返回值是0或者是iOS定义的一个常量noErr则表示成功了。

现在可以对该转码实例设置比特率了，代码如下所示：

```
AudioConverterSetProperty(_audioConverter, kAudioConverterEncodeBitRate,
    sizeof(bitRate), &bitRate);
```

之后需要获取编码之后输出的AAC其Packet size的最大值是多少，因为需要按照该值来分配编码后数据的存储空间，从而让编码器输出到该存储区域中，代码如下：

```
UInt32 size = sizeof(_aacBufferSize);
AudioConverterGetProperty(_audioConverter, kAudioConverter
        PropertyMaximumOutputPacketSize, &size, &_aacBufferSize);
_aacBuffer = malloc(_aacBufferSize * sizeof(uint8_t));
        memset(_aacBuffer, 0, _aacBufferSize);
```

至此初始化方法就结束了，这里面除了分配出编码器以及为编码器设置比特率之外，还需要根据获取到的输出AAC的Packet大小分配存储AAC的Packet的存储空间。

下面来看第二个接口，真正编码函数的实现。首先要利用前面已初始化好的_aacBuffer构造出一个AudioBufferList的结构体，作为编码器输出AAC数据的存储容器，代码如下：

```
AudioBufferList outAudioBufferList = {0};
outAudioBufferList.mNumberBuffers = 1;
outAudioBufferList.mBuffers[0].mNumberChannels = _channels;
outAudioBufferList.mBuffers[0].mDataByteSize = _aacBufferSize;
outAudioBufferList.mBuffers[0].mData = _aacBuffer;
```

构造出该结构体之后，就可以调用编码器的编码（转换）函数了。但是有的读者会有疑问，我们还没有拿到PCM数据，又该如何调用编码器编码呢？也就是数据源从哪里来呢？其实在iOS中提供的API一般都是按照回调函数的方式获取数据源的，这里就是一个非常典型的应用：

```
UInt32 ioOutputDataPacketSize = 1;
OSStatus status = AudioConverterFillComplexBuffer(
        _audioConverter, inInputDataProc, (__bridge void *)(self),
        &ioOutputDataPacketSize, &outAudioBufferList, NULL);
```

再来总体解释一下该函数，其中第一个参数是实例化好的编码器（或者说是转换器）；第二个参数就是一个回调函数，即当编码器（或者说是转换器）需要开发者填充数据的时候，编码器（或者转换器）就会调用这个回调函数来获得PCM数据；接下来的参数就是对象本身，回调函数一般都会传入一个context，以便调用本对象的方法；接下来的参数就是输出的AAC的Packet的大小；再就是编码之后的AAC的Packet存放的容器；最后一个参数是输出AAC的Packet的Description，一般填充为NULL。

下面来看一下该回调函数的原型以及在回调函数中如何填充PCM数据，该回调函数的原型如下：

```
OSStatus inInputDataProc(AudioConverterRef inAudioConverter, UInt32
        *ioNumberDataPackets, AudioBufferList *ioData,
        AudioStreamPacketDescription **outDataPacketDescription,
        void *inUserData)
```

下面来看一下该回调函数的具体参数，第一个参数是编码器（或者

说转换器）的实例；第二个参数是需要填充多少个PCM的packet（或者说是frame）；第三个参数是开发者实际填充PCM数据的容器；第四个参数是填充输出Packet的Description，但是在这里不使用；最后一个参数就是上下文，即在调用编码函数（或者说转换函数）的时候传入的对象本身。像大多数的回调函数一样，我们可以将inUserData强制转换为本类类型的一个实例对象，然后就可以调用该对象的方法了，代码如下：

```
AudioToolboxEncoder *encoder = (__bridge AudioToolboxEncoder *)(inUserData);
return [encoder fillAudioRawData:ioData ioNumberDataPackets:
        ioNumberDataPackets];
```

在这个静态的回调函数中，通过上下文对象的强制类型转换，就可以得到对象本身，进而可以调用到fillAudioRawData方法。接下来再介绍一下该方法的具体实现，首先根据需要填充的帧的数目、当前声道数以及表示格式计算出需要填充的uint8_t类型的buffer的大小，计算公式如下所示：

```
int bufferLength = ioNumberDataPackets * channels * sizeof(short);
```

然后根据上述公式算出来的bufferLength来分配pcmBuffer，接下来调用delegate里面的fill-AudioData：bufferSize：方法来填充数据，最后将客户端代码填充好的pcmBuffer放入ioData容器中并返回，这样就完成了为编码器（或者说是转换器）提供PCM数据的回调函数。

编码函数（或者说是转换函数）执行结束之后，就可以读取出前面拿出定义好的out-AudioBufferList结构体，编码好的数据就存放在这里。从该结构体的属性mBuffers[0].mData中读取出AAC的原始Packet，添加上ADTS头信息，并调用delegate的方法outputAACPakcet：presentationTimeMills：error：，该方法约定由客户端来输出编码之后的带有ADTS头信息的AAC数据，最终如果输入数据为空，则代表结束，我们就可以调用delegate的方法onCompletion，让客户端对自己的资源进行关闭以及销毁的操作了。

最后来看一下销毁编码器的接口实现，先释放已分配的填充PCM数据的pcmBuffer，然后释放分配的接受编码器输出的aacBuffer，最后释

放编码器，代码如下：

```
if(_pcmBuffer) {
    free(_pcmBuffer);
    _pcmBuffer = NULL;
}
if(_aacBuffer) {
    free(_aacBuffer);
    _aacBuffer = NULL;
}
AudioConverterDispose(_audioConverter);
```

至此编码类的实现就全部实现完毕了，集成阶段的实现具体如下。

首先客户端代码需要实现该类中定义的FillDataDelegate类型的Protocol，并且需要重写其中的fillAudioData方法，以便为该编码器类提供PCM数据，此外，还需要重写outputAAC-Packet方法来输出编码添加上ADTS头的AAC的码流数据，重写onCompletion方法以关闭自己的读写文件等操作；然后实例化编码器，开启一个线程（使用GCD）来调用编码方法；最终在编码结束之后或者在dealloc方法中调用结束编码的方法。

本节的代码实例是代码仓库中的AudioToolboxEncoder工程，输入就是bundle目录下面的vocal.pcm，编码之后在App的document目录下面，可以利用Xcode导出App的container，然后读取出AAC文件使用ffplay播放或者使用ffprobe查看格式描述。

6.4 视频画面的编码

本节将讨论视频的编码，对于视频编码的原理与发展历史，本书的第1章中已经有过简单的介绍，本节讨论的主要内容是如何以软件编码和硬件编码的方式在两个平台上分别完成H264格式的编码工作。当然软件编码实际使用的库是libx264库，但是开发是基于FFmpeg的API进行的；对于硬件编码，可使用各自平台提供的硬件编码器来实现。而编码的输入就是6.3节中摄像头捕捉的纹理图像（显存中的表示），输出是H264的Annexb封装格式的流。当然，如果读者愿意自行尝试的话，也完全可以使用前面第5章中解码器的输出作为编码器的输入，其实这就是后期处理保存的过程，本书会在后续的章节中进行单独介绍。

6.4.1 libx264编码H264

在iOS平台上，由于硬件编码器的兼容性比较好，所以基本上用不到libx264这些软件的编码器，毕竟在移动平台上只要兼容性没有问题，肯定会以性能作为第一考虑因素，所以本节的实例只需要运行在Android平台上即可。但是编码部分的代码都是使用C++来编写的，是跨平台的，因此如果在iOS平台上有需要的话可以直接使用编码部分的代码。由于输入是一张纹理，输出是H264的裸流，因此可将实例分为以下几个部分来讲解。

首先，编码模块的输入肯定是6.3节中讲解的摄像头预览控制器渲染到屏幕之前的纹理ID。先来看一下编码器模块的两种实现，本节讨论的是软件编码器，6.4.2节将讨论硬件编码器，对此，经典的设计就是抽象出一个接口，然后提供一个软件编码器的实现和一个硬件编码器的实现。接下来看一下在摄像头预览的控制器这个类里，如何与编码器模块进行交互，其实就是抽象出的该编码器模块的接口，首先是初始化接口：

```
void init(const char* h264Path, int width, int height, int videoBitRate, floa
        frameRate)
```

该接口传入的参数分别是编码之后的H264文件的存储路径，编码视频的宽、高，编码H264的比特率，视频的帧率等。该接口负责将这些视频编码的MetaData信息存储到全局变量中，并且执行打开文件的操作。接下来看一下创建编码器的接口：

```
virtual int createEncoder(EGLCore* eglCore, int inputTexId) = 0;
```

可以看到该方法是一个纯虚的方法，代表由具体的子类来完成操作，由于编码器模块的输入是渲染到屏幕上之前的纹理ID，所以需要对纹理ID进行操作，即把OpenGL ES的上下文环境的封装类EGLCore以及要渲染的纹理ID传入进来，该接口负责创建编码器资源以及转换纹理对象以适配编码器，这也引出了编码器模块入口的接口名字：VideoEncoderAdapter。为一个类命名其实就是根据该类的职责而确定

的，上面这个类实际上就是将输入的纹理ID做一个转换，使得转换之后的数据可以作为具体编码器的输入，所以这也是该接口的名字所代表的意义，而本节要完成的就是该软件编码器适配器的实现，即SoftEncoderAdapter。6.4.2节将会完成硬件编码器适配器的实现，即HWEncoderAdapter。该函数需要构建出OpenGL ES环境，并且创建编码器，如果成功则返回0，否则返回负数。接下来看一下编码接口：

```
virtual void encode() = 0;
```

同样的，这也是一个纯虚的方法，由子类自己来完成编码操作，实际上就是利用自己构造的OpenGL ES渲染线程来完成适配工作的过程。

下面来看下一个接口，即销毁编码器的接口：

```
virtual void destroyEncoder() = 0;
```

这也是一个纯虚的方法，由子类自己来完成销毁编码器以及销毁OpenGL ES的渲染线程，这就是我们抽象出来的接口的三个方法，通过图6-17可以更加清晰地看到它们之间的调用关系。

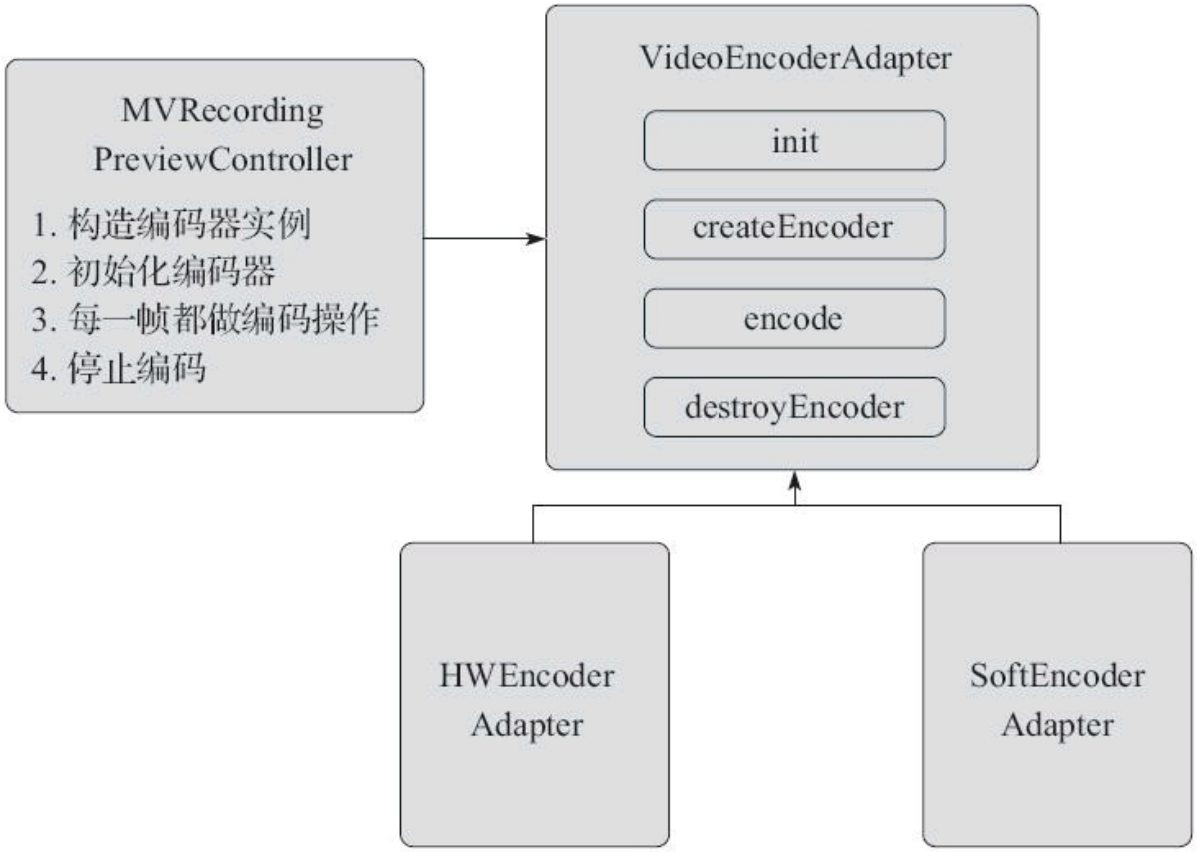


图　6-17

本节的目标就是完成图6-17中软件编码器适配器的实现部分，即SoftEncoderAdapter部分，首先从全局来看一下软件编码器的整体结构，如图6-18所示。

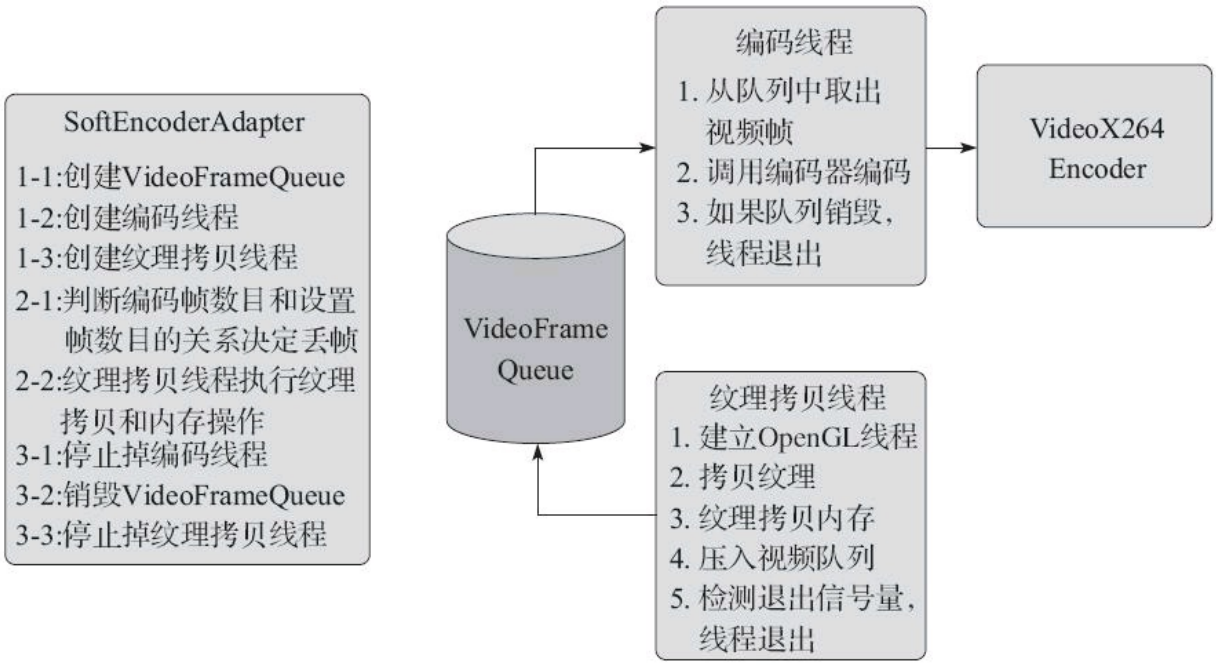


图　6-18

从图6-18中可以看到整个软件编码器模块的整体结构，其实，纹理拷贝线程是一个生产者，它生产的视频帧会放入VideoFrameQueue中；而编码线程则是一个消费者，其可从VideoFrameQueue中取出视频帧，再进行编码，编码好的H264数据将输出到目标文件中。

在createEncoder方法的实现中，即图6-18中左边框图的1-1、1-2以及1-3部分，分别创建存储视频帧的队列VideoFrameQueue、编码线程以及纹理拷贝线程。这三个对象的职责一目了然，因此不作赘述，这里主要来看一下这三个对象是如何实现的。首先来看Video-FrameQueue，这是一个我们自己实现的保证线程安全的队列，实际上就是一个链表，链表中每个Node节点内部的元素均是一个VideoFrame的结构体，该结构体中的元素具体如下：

```
typedef struct VideoFrame_t {
    unsigned char * buffer;      // YUV420P的图像数据
    int size;                    // 图像数据的大小
    int timeMills;               // 所代表的时间戳
    int duration;                // 这一帧图像所代表的时间长度
} VideoFrame;
```

VideoFrameQueue对象提供了以下几个接口：第一个是init方法，该方法会初始化锁来保证线程的安全，同时也会初始化头指针和尾部指针为空，此外，还将初始化一个布尔形变量，代表是否要丢弃所有帧，即mAbortRequest；第二个就是put方法，在保证线程安全（即在操作指针前后上锁和解锁）的前提下，将新的元素连接到该链表的最后面，并且发出signal指令；第三个是get方法，在保证线程安全性（即在操作指针前后上锁和解锁）的前提下，取出头部指针指向的元素，并且将指针指向下一个元素；如果没有元素可以取出的话，那就wait，等接收到signal指令之后再去取出元素，signal指令有可能是put函数或者abort函数被调用前发出的；第四个是abort方法，即我们要放弃队列中的所有元素了，当put方法和get方法再次被调用时就会被忽略；最后在析构函数中会把剩余的所有元素都逐一取出，并且释放掉，以防止内存泄漏。

下面再来看编码线程，在编码线程中首先需要实例化编码器，然后进入一个循环，不断从VideoFrameQueue里面取出视频帧元素，调用编码器进行编码，如果从VideoFrameQueue中获取元素的返回值是-1，则跳出循环，最后销毁编码器，代码如下：

```
encoder = new VideoX264Encoder();
encoder->init(videoWidth, videoHeight, videoBitRate, frameRate, h264File);
LiveVideoFrame *videoFrame = NULL;
while(true){
    if (videoFramePool->getYUY2Packet(&videoFrame, true) < 0) {
        break;
    }
    if(videoFrame){
        // 调用编码器编码这一帧的数据
        encoder->encode(videoFrame);
        delete videoFrame;
        videoFrame = NULL;
    }
}
if(NULL != encoder){
    encoder->destroy();
    delete encoder;
    encoder = NULL;
}
```

现在分析纹理拷贝线程，由于将纹理从显存中拷贝到内存中需要耗费的时间比较长，为了尽量不阻塞预览界面的渲染线程，因此建立了该纹理拷贝线程。该线程首先需要初始化OpenGL ES的上下文环境，然后绑定到新建立的这个纹理拷贝线程之上。该纹理拷贝线程拷贝的纹理ID是在客户端代码调用createEncoder方法的时候传递进来的，而该纹理是

在预览界面渲染线程的上下文中创建的，那么为什么在新建立的渲染线程中可以访问到呢？这也是我们要将EGLCore从预览控制类中传递过来的原因，因为我们要使用OpenGL中共享上下文的概念，即在创建OpenGL ES上下文的时候，使用已经存在的EGLContext而不是EGL_NO_CONTEXT，这样，新创建的这个上下文就可以和已经存在的EGLContext共享所有的OpenGL对象了，包括纹理对象、帧缓存对象，等等。而这里已经将渲染线程中封装的EGLCore这个指针传递过来了，也就是可以获得预览界面渲染线程的OpenGL ES的上下文了。此外，创建OpenGL ES上下文的时候会将传递过来的上下文当作第三个参数传递进去，这样我们在当前纹理拷贝线程中就可以访问预览界面渲染线程所创建的纹理对象了。代码如下：

```
eglCore = new EGLCore();
eglCore->init(previewGLContext);
copyTexSurface = eglCore->createOffscreenSurface(videoWidth, videoHeight);
eglCore->makeCurrent(copyTexSurface);
```

是时候创建一个输出纹理ID了，我们拷贝的目标就是该输出纹理对象，此外，还需要创建一个帧缓存对象，帧缓存对象是任何一个OpenGL Program渲染的目标。当然像之前直接渲染到屏幕上的都会有一个默认的帧缓存对象，但目前的场景并不是向屏幕上绘制，而是进行纹理拷贝，所以我们需要自行创建一个帧缓存对象。创建纹理ID与帧缓存对象的代码如下：

```
glGenFramebuffers(1, &mFBO);
glGenTextures(1, &outputTexId);
```

在进行真正的拷贝之前，需要显式地绑定该帧的缓存对象，然后使用一个通用的renderer（使用OpenGL的Program将输入纹理ID绘制到我们绑定的帧缓存对象上），这个renderer是在播放器项目中封装的通用的渲染代码，renderer的渲染目标就是绑定的这个帧的缓存对象，又因为我们把输出纹理ID绑定到了这个帧缓存对象之上，所以就相当于是将输入纹理的内容绘制到了输出纹理上面去了。完成拷贝之后需要再解绑定这个帧缓存对象，代码如下：

```
glViewport(0, 0, videoWidth, videoHeight);
glBindFramebuffer(GL_FRAMEBUFFER, mFBO);
```

```
checkGlError("glBindFramebuffer FBO");
long startTimeMills = getCurrentTime();
renderer->renderToTexture(texId, outputTexId);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

在拷贝到目标纹理之后，就可以让预览线程继续进行自己的工作了，但是在拷贝成功之前，需要阻塞预览线程，具体实现就是在调用encode方法的时候使用条件锁wait，待纹理拷贝线程拷贝成功了之后，发送一个signal指令过来，encode方法接受到signal指令之后就可以结束了，以让预览线程继续运行。这样就可以达到最短时间的阻塞预览线程，以防止发生预览界面的FPS降低，然后我们需要做的就是，将该输出纹理ID的内容拷贝到内存中，这就涉及显存和内存的数据传递了。在做视频处理以及编解码的时候，需要遵从一个原则：尽量少地进行显存和内存的交换。但是在不得已的情况下，必须要将显存中的数据传递到内存中，因为我们是使用X264进行编码操作的，而X264的输入必须是内存中的数据。OpenGL中提供了显存到内存的数据转换API：glReadPixels，它一共有7个参数，第一个和第二个参数表示了矩形左下角的横、纵坐标，坐标系就是OpenGL的纹理坐标系；第三个和第四个参数表示了矩形的宽度和高度；第五个参数是读取的内容格式，一般是读取RGBA格式的数据；第六个参数是读取的内容在内存中的表示格式；第七个参数就是我们要存储到的内存区域的指针。该函数默认会读取出RGBA格式的数据，并且会非常耗时，读取的内容区域越大，所消耗的时间也会越多，所以对于分辨率大的纹理ID，读取一帧数据所耗费的时间就比较多了。因此需要把显存到内存中的数据交换（耗时、性能低）和拷贝纹理（速度快）分为两个阶段，使得最终效果不会阻塞预览界面的渲染线程。而对于显存到内存的数据转换的优化，其主要的思路就是减少数据量的读取，那么如何减少数据量的读取呢？可以把一张RGBA格式的纹理ID先转换为YUY2的格式，YUY2的格式将为每个像素保留Y分量，而UV分量在水平方向上将每两个像素采样一次，即一个像素RGBA格式使用4个字节来表示，而YUY2格式使用2个字节来表示，这样从显存到内存转换数据的时候数据量就减小了一半，而读取所耗费的时间也几乎减少到了原始时间的一半，但是我们需要做的额外工作是，在显存中通过一个OpenGL Program将RGBA格式转换为YUY2格式，然后再进行读取，具体请查看代码示例中的host_gpu_copier.cpp实现文件，最后将YUY2格式的数据交给编码器进行编码。

接下来介绍VideoX264Encoder的实现，该类主要完成的任务是将YUY2的原始图像数据编码成H264的压缩数据，然后写到H264的文件

中。

首先，使用libx264时，并不是直接使用它的API，而是基于FFmpeg的API来开发的，当然，在此之前要将libx264交叉编译到FFmpeg中去，这点在前面已经编译进去了。现在将FFmpeg的头文件以及静态库文件拉入到工程中的jni目录下，并且在Android.mk中进行合适的配置，然后新建一个C++文件video_x264_encoder，下面来看一下该文件对外提供的接口。

初始化接口：

```
int init(int width, int height, int videoBitRate, float  frameRate,
    FILE* h264File)
```

上述接口需要传入要编码视频的宽、高以及视频的码率和帧率，最后一个参数是编码之后要写入的H264文件，该接口负责进行初始化编码器上下文以及编码之前的AVFrame结构体，如果成功则返回0，否则返回负数。接下来看一下第二个接口，实际的编码接口：

```
int encode(VideoFrame *videoFrame);
```

该接口需要传入的参数是一个VideoFrame的结构体，该结构体中实际上包含了这一帧图片的YUY2数据以及时间信息，之前也提到过，为了性能考虑，从显卡中读出来的格式是YUY2的格式，所以这里的输入格式是YUY2的视频帧表示格式，而libx264输入的格式一般都是YUV420P的格式，所以要在将数据发送到libx264之前将其转换成为YUV420P格式的数据，然后把这一帧图像编码成为H264的数据，并写入文件。如果编码失败则返回负数，如果编码成功则返回0。从YUY2到YUV420P格式的具体转换过程已做优化，在armv7平台上利用Neon指令集来做加速，在X86平台使用SSE指令集来做加速，这些加速操作其实都是SIMD指令集的应用，就是单指令多数据的操作，由于篇幅关系具体代码部分请读者参考实例代码。

接下来再看最后一个销毁接口：

```
void destroy();
```

该接口负责销毁编码器的上下文以及销毁分配的AVFrame等资源。

本节的代码示例是代码仓库中的CameraPreviewRecorder的Android工程，读者可以更改CameraPreviewActivity中的第80行，调整为软件编码，并且给出自己的路径，进入预览界面之后点击编码按钮开始编码，当点击停止按钮之后，编码结束，然后可以利用adb pull命令将编码之后的H264文件导出到电脑上，最后再利用ffplay进行播放以观看效果。

6.4.2 Android平台的硬件编码器MediaCodec

在Android 4.3系统之后，用MediaCodec编码视频成为了主流的使用场景，尽管Android的碎片化比较严重，会导致一些兼容性问题，但是硬件编码器的性能以及速度是非常可观的，并且在4.3系统之后可以通过Surface来配置编码器的输入，大大降低了显存到内存的交换过程所使用的时间，从而使得整个应用的体验得到大大提升。由于输入和输出已经确定，因此接下来将直接编写MediaCodec编码视频帧的过程。

6.4.1节中已经介绍了预览控制器类如何调用编码器模块进行编码，并且也已经实现了软件编码器的子类，本节要完成的目标就是实现硬件编码器子类的编写。新建的hw_encoder_adapter类继承自video_encoder_adapter类，然后实现所有的虚函数，包括创建编码器的createEncoder方法，实际编码的encode方法以及销毁编码器的destroyEncoder方法。MediaCodec的具体运转流程如图6-16所示。因为MediaCodec是Android提供的Java层的API，因此需要在C++层调用Java层的代码，所以需要在该类的构造方法中将JavaVM以及jObject传递进来。

首先来看一下创建编码器的方法实现，6.4.1节中libx264编码器是以内存中的数据作为输入的，所以我们需要进行显存到内存的数据交换，而本节使用的MediaCodec是允许直接以显存中的纹理对象作为输入的，这就在提供数据的速度层面有了很高的保证，同时也减少了显存到内存的数据交换，这一点也是十分关键的。

调用Java层对象封装的创建编码器的方法，把视频的宽、高、比特率、帧率等传递上去，然后在Java层利用这些参数创建编码类型为“video/avc”的MediaCodec实例；然后调用该实例对象的configure方法配置编码器；当编码器配置成功之后，再调用实例对象的createInputSurface方法创建该MediaCodec的输入Surface；然后调用实例对象的start方法来开启该编码器。接下来我们将Java层通过MediaCodec创建的Surface对象传递给Native层，在Native层构造一个ANativeWindow，然后与预览控制器传递过来的EGLCore共同创建EGLSurface，最后再创建一个renderer，即我们封装的一个OpenGL Program，目的是利用这个renderer将输入纹理ID渲染到目标Surface上去。注意该Surface是Java层MediaCodec的输入Surface，而不是预览控制

类中的Surface（预览控制器的Surface是Java层的Surface-View的Surface）。由于上述过程比较复杂，其中涉及Java层和Native层的交互，又涉及MediaCodec的使用，所以笔者画了一个时序图以方便读者理解，如图6-19所示。紧接着创建一个jbyteArray类型的buffer，用于在MediaCodec中拉取编码之后的H264数据。为了不影响预览线程的刷新频率，这里把从MediaCodec中拉取数据的操作放到了一个新的线程中，所以这里需要创建出一个线程来单独进行拉取编码器中编码数据的操作。另外由于MediaCodec编码器编码H264数据的时候，会在前几帧中返回SPS和PPS信息，因此需要将SPS和PPS作为全局变量存储下来，放到每一个关键帧的前面，从而组成H264文件。关于SPS和PPS的具体概念，以及如何判断H264这一帧的NALU Type，前面的章节中已经有过介绍，这里不再赘述。

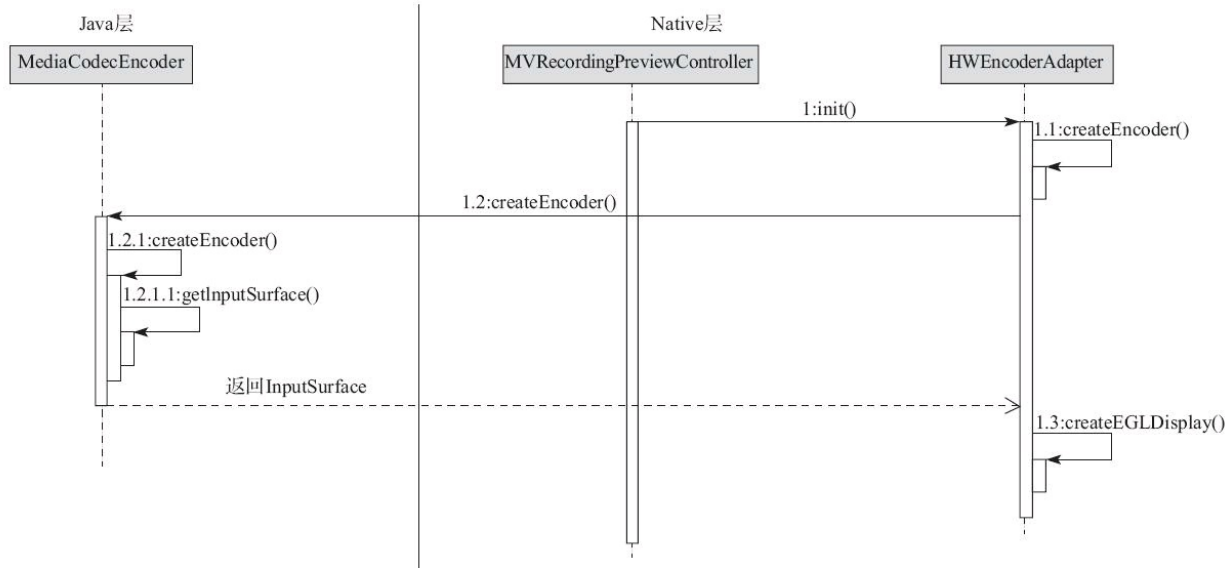


图　6-19

下面再来看一下编码方法的实现，把第一次编码取得的当前时间戳记录为开始编码的时间戳startTime，之后再调用编码操作的时候，取出当前时间减去startTime，算出编码时长，然后根据帧率和编码时长算出我们期待的编码数目expectedFrameCnt，并且在每编码一帧的时候就为全局变量encodedFrameCnt加1。通过比较这两者的关系，来控制是否将这一帧视频帧发送给编码器，并且发送给编码器的这一帧视频帧的时间戳就是上面计算的时长。如何发送给编码器呢？首先调用库EGL的makeCurrent方法，将encoderSurface作为渲染目标，接下来调用renderer将输入的纹理ID渲染到Surface上去，然后为编码器设置编码的时间，最

后向拉取编码数据的线程发送一个指令，让其到MediaCodec中拉取H264的数据，并调用库EGL的swapBuffer渲染数据。代码如下：

```
if (startTime == -1)
startTime = getCurrentTime();
int64_t curTime = getCurrentTime() - startTime;
int expectedFrameCount = (int)(curTime / 1000.0f * frameRate + 0.5f);
if (expectedFrameCount < encodedFrameCount) {
    // need drop frames
    return;
    }
encodedFrameCount++;
if(EGL_NO_SURFACE != encoderSurface){
    eglCore->makeCurrent(encoderSurface);
    renderer->renderToView(texId, videoWidth, videoHeight);
    eglCore->setPresentationTime(encoderSurface,
             ((khronos_stime_nanoseconds_t) curTime) * 1000000);
    handler->postMessage(new Message(FRAME_AVAILIBLE));
    eglCore->swapBuffers(encoderSurface)
    }
```

那么，拉取MediaCodec中H264数据的这个线程是如何工作的呢？首先传递到Java层的参数就是上述第一步中创建的jbyteArray的buffer，在Java层会调用MediaCodec的dequeue-OutputBuffer方法，该方法的返回值包含以下几种状态。

·INFO_TRY_AGAIN_LATER，代表获取数据超时，稍后再试。

·INFO_OUTPUT_FORMAT_CHANGED，输出格式发生变化，需要重新获取新的输出格式。

·INFO_OUTPUT_BUFFERS_CHANGED，输出的缓冲区发生变化，需要重新获取新的缓冲区。

若返回值不是上述的几个状态，并且又大于0的话，我们就可以取出对应的编码数据了，然后将buffer中的数据复制出来，并调用MediaCodec的release方法来将该buffer放回到缓冲队列中。之后在C++层就可以拿到该数据了，我们会用该buffer数据中index为4的数据与0X1F做“相与”的操作来判断NALU类型，如果是SPS和PPS，就存储到全局变量中，因为要在每一个关键帧前面写入SPS和PPS。代码如下：

```
int nalu_type = (outputData[4] & 0x1F);
```

```
if (H264_NALU_TYPE_SEQUENCE_PARAMETER_SET == nalu_type) {
    spsppsBufferSize = size;
    spsppsBuffer = new byte[spsppsBufferSize];
    memcpy(spsppsBuffer, outputData, spsppsBufferSize);
} else if(NULL != spsppsBuffer){
    if(H264_NALU_TYPE_IDR_PICTURE == nalu_type) {
        fwrite(spsppsBuffer, 1, spsppsBufferSize, h264File);
    }
    fwrite(outputData, 1, size, h264File);
}
```

最后再来看一下销毁编码器方法的实现，首先要停止拉取编码器数据的线程，然后调用Java层的方法关闭MediaCodec，并释放相关的编码器资源，Java层会调用MediaCodec的stop与release方法，最后，再释放分配的jbyteArray类型的buffer，同时释放全局的SPS和PPS的buffer，并且关闭文件。

本节的代码示例是代码仓库中的CameraPreviewRecorder的Android工程，读者可以更改CameraPreviewActivity中的第80行，调整为硬件编码，并且可以输入自己的路径，进入预览界面之后点击编码按钮开始编码，点击停止按钮之后，编码结束，然后可以利用adb pull命令将编码之后的H264文件导出到电脑上，最后利用ffplay进行播放以观看效果。

6.4.3　iOS平台的硬件编码器

6.4.1节中完成了利用软件编码器libx264编码H264数据的实例，但其是基于Android平台的，基于iOS平台的软件编码器这里就不做实现了，如果读者有兴趣，可以直接利用6.4.1节编码器的封装类，做一下iOS平台上的输入适配就可以了，因为在iOS平台上硬件编码器的兼容性比较好，所以对于H264编码格式一般不需要使用软件编码器。

在iOS 8.0以后，系统提供了VideoToolbox编码API，该API可以充分使用硬件来做编码工作以提升性能和编码速度。本节就来讲解一下如何将一系列连续的纹理，利用VideoToolbox编码成一段H264的数据。首先来介绍VideoToolbox如何将一帧视频帧数据编码为H264的压缩数据，并把它封装到H264HWEncoderImpl类中，然后再将封装好的这个类集成进6.2.2节的预览系统中，集成进去之后，对于原来仅仅是预览的项目，也可以将其保存到一个H264文件中了。

1.使用VideoToolbox构造自己的编码器

使用VideoToolbox可以为系统带来以下几个优点，提高编码性能（使得CPU的使用率大大降低），增加编码效率（使得编码一帧的时间缩短），延长电量使用（耗电量大大降低），而VideoToolbox是iOS 8.0以后才公开的API，既可以做编码又可以做解码工作。本节主要介绍编码的工作流程，后续的章节也会对使用VideoToolbox完成解码场景内容进行讲解。

首先，来看一下使用VideoToolbox进行编解码的输入输出分别是什么，只有明确了这一点，才可以知道如何为VideoToolbox编码器提供输入数据，并且知道如何从VideoToolbox中获取编码之后的数据。VideoToolbox的编码原理如图6-20所示。

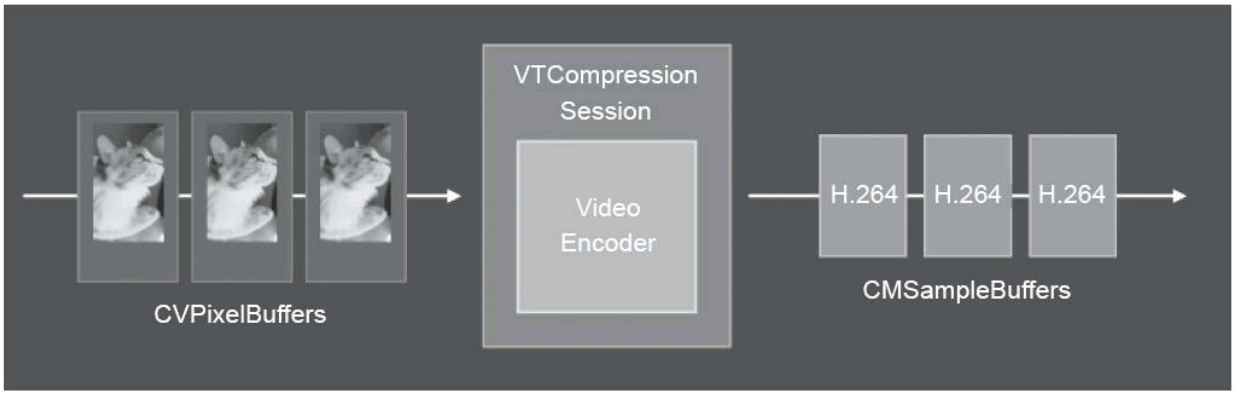


图　6-20

如图6-20所示，左边的三帧视频帧是发送给编码器之前的数据，开发者必须将原始图像数据封装为CVPixelBuffer的数据结构，该数据结构是使用VideoToolbox编解码的核心，我们必须要理解清楚，Apple Developer官网上针对CVPixelBuffer给出的解释是其是主内存中存储所有像素点数据的一个对象，那么什么是主内存？笔者做过实验之后理解为主内存其实并不是我们平时所操作的内存，但是两者的概念是可以关联起来的，所以笔者将主内存理解为这块存储区域存在于缓存之中，我们在访问这块区域之前必须先锁定这块区域：

```
CVPixelBufferLockBaseAddress(pixel_buffer, 0);
```

然后才可以访问这块内存区域：

```
void* data = CVPixelBufferGetBaseAddress(pixel_buffer)
```

操作data变量可以向该内存区域填充内容或者从中读取内容，最终使用完毕之后，要解锁该区域，以确保后续操作可以正常进行：

```
CVPixelBufferRelease(pixel_buffer);
```

从它的使用方式来看，该区域肯定不是普通的内存访问，否则不会在访问内存区域之前和之后加上锁定、解锁定等操作，前面的章节中也说过，做视频开发有一个原则：尽量少地进行显存和内存的交换。所以

在iOS的开发中也要尽量少地访问它的内存区域，我们应该使用iOS平台提供的对应的API来完成相应的操作。其实，可以回顾一下6.2.2节中所讲的Camera回调，它提供给我们的数据其实就是CVPixelBuffer，只不过当时使用的引用类型是CVImageBufferRef，但是可以在iOS头文件定义中找到它，其实就是CVPixelBuffer的另一个定义。大家可以回过头来看一下Camera的回调中是如何处理的，其核心就是如何将CVPixelBuffer中的内容构造成纹理对象，以供OpenGL ES使用，而不是手动获取CVPixelBuffer的内存地址，然后利用OpenGL ES提供的glTexImage2D方法来将内存中的数据上传到显存，以构建纹理对象，再将其发送给OpenGL ES使用。而iOS的CoreVideo这个framework提供的方法CVOpenGLESTextureCacheCreateTextureFromImage就是专门用来将纹理对象关联到CVPixelBuffer表示视频帧的方法。

下面来看这个编码器输出的对象，可以看到图6-21表示的是一个CMSampleBuffer的对象，如果大家还有印象的话，6.2.2节中讲解的Camera回调给我们的视频帧也是CMSample-Buffer的对象，但是它们所包含的内容完全不一样，Camera预览返回的CMSampleBuffer中存储的数据是一个CVPixelBuffer，而经过VideoToolbox编码输出的CMSampleBuffer中存储的数据是一个CMBlockBuffer的引用，如图6-21所示。

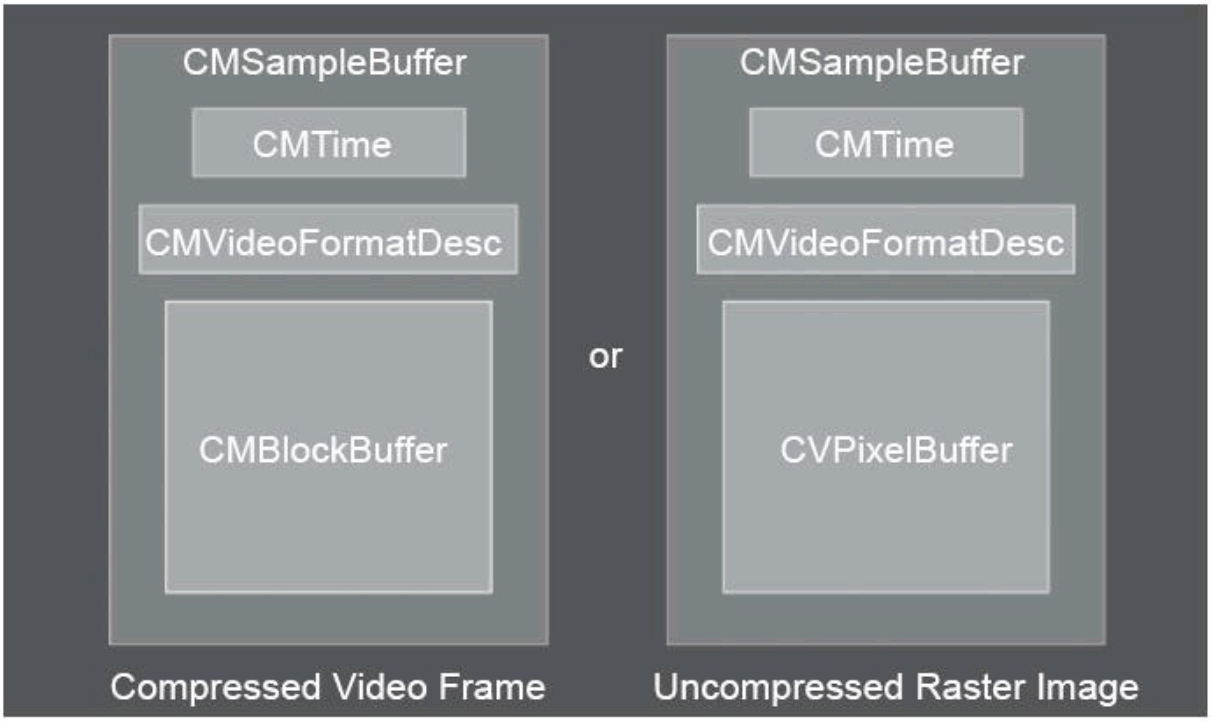


图　6-21

图6-21展示了CMSampleBuffer的构成方式，左边代表了压缩格式（编码器输出的数据）的构成，右边代表了非压缩格式（摄像头采集到的数据或者解码器解码出来的数据）的构成。而CMBlockBuffer就是编码之后数据存放的对象，我们可以调用CMBlockBufferGet-DataPointer方法来获取内存中的指针，然后使用内存中对应的数据去做自己的处理（写文件或者发送到网络）。

既然弄清楚了编码器的输入和输出，下面就来看一下如何构建编码器。还记得之前一直说的一句关于iOS多媒体API的话吗？那就是使用任何硬件设备时都要使用对应的Session，使用麦克风以及Speaker的时候使用的是AudioSession，使用Camera的时候使用的是AVCaptureSession，而这里使用的会话就是VTCompressionSession，这个会话就代表要使用编码器，等后续讲到硬件解码场景时将要使用的会话就是VTDecompressionSessionRef。那么下面就来讲解如何定制一个我们需要的编码器的会话。

首先，调用VTCompressionSessionCreate方法将要编码的视频的宽、高、编码器类型（kCMVideoCodecType_H264）、回调函数以及回调函数上下文传递进去，然后构造出一个编码器会话，该函数的返回值是一个OSStatus，如果构造成功则返回的是0，如果不成功则要给出提示，用于通知客户端代码初始化编码器会话失败。构造成功之后要为该会话设置参数，具体代码如下：

```
VTSessionSetProperty(EncodingSession, kVTCompressionPropertyKey_RealTime,
    kCFBooleanTrue);
VTSessionSetProperty(EncodingSession, kVTCompressionPropertyKey_ProfileLevel,
    kVTProfileLevel_H264_High_AutoLevel);
VTSessionSetProperty(EncodingSession ,
    kVTCompressionPropertyKey_AllowFrameReordering, kCFBooleanFalse);
VTSessionSetProperty(EncodingSession,
    kVTCompressionPropertyKey_MaxKeyFrameInterval,
    (__bridge CFTypeRef)(@(fps)));
VTSessionSetProperty(EncodingSession,
    kVTCompressionPropertyKey_ExpectedFrameRate, (__bridgeCFTypeRef)(@(fps)))
VTSessionSetProperty(EncodingSession, kVTCompressionPropertyKey_DataRateLimit
    (__bridge CFArrayRef)@[@(maxBitRate / 8), @1.0]);
VTSessionSetProperty(EncodingSession, kVTCompressionPropertyKey_AverageBitRat
    (__bridge CFTypeRef)@(avgBitRate));
```

这里面第一个参数设置的是需要实时编码，第二个参数设置的是使用的H264的Profile是High的AutoLevel规格，第三个参数是我们不产生B帧，第四个参数是设置关键帧间隔，也就是通常所指的gop size，第五个参数是设置帧率，第六个参数和第七个参数共同用于控制编码器输出的码率。设置完这些参数之后，调用VTCompressionSessionPrepareToEncodeFrames方法，告诉编码器开始编码。下面把构建会话与设置参数的这部分代码封装到H264HW-Encoder类的初始化方法中，该初始化方法签名如下：

```
- (void)initEncode:(int)width height:(int)height fps:(int)fps
        maxBitRate:(int)maxBitRate avgBitRate:(int)avgBitRate;
```

在最开始创建该会话的时候指定了一个回调函数，该回调函数是在编码器编码成功一帧之后，把编码成功的这一帧数据构造成一个CMSampleBuffer结构体以回调这个函数，开发者要在这个回调函数里处理数据。我们在第一步中仅仅明确了编码器的输入和输出，其实并没有说明编码器具体是如何使用的，具体来说是在使用我们封装的H264HWEncoder类的encode方法的时候，输入参数是一个CVPixelBuffer，然后构造当前编码视频帧的时间戳以及时长，最后调用编码会话对这三个参数进行编码。代码如下：

```
int64_t currentTimeMills = CFAbsoluteTimeGetCurrent() * 1000;
if(-1 == encodingTimeMills){
    encodingTimeMills = currentTimeMills;
}
int64_t encodingDuration = currentTimeMills - encodingTimeMills;
CVImageBufferRef imageBuffer = (CVImageBufferRef)
        CMSampleBufferGetImageBuffer(sampleBuffer);
CMTime pts = CMTimeMake(encodingDuration, 1000.); // timestamp is in ms.
CMTime dur = CMTimeMake(1, m_fps);
VTEncodeInfoFlags flags;
OSStatus statusCode = VTCompressionSessionEncodeFrame(EncodingSession,
                                    imageBuffer,
                                    pts,
                                    dur,
                                    NULL, NULL, &flags);
if (statusCode != noErr) {
    error = @"H264: VTCompressionSessionEncodeFrame failed ";
    return;
}
```

待编码器编码成功之后，就会回调最开始初始化编码器会话时传入

的回调函数，回调函数的原型如下：

```
void didCompressH264(void *outputCallbackRefCon,
void *sourceFrameRefCon,
OSStatus status, VTEncodeInfoFlags infoFlags,
CMSampleBufferRef sampleBuffer )
```

开发者应该在该回调函数中处理编码之后的数据，首先判断status，如果编码成功则返回0（实际上头文件中定义了一个枚举类型是noErr）；如果不成功则不处理。成功的话首先来判断编码成功之后的当前帧是否为关键帧，判断关键帧的方法如下：

```
CFArrayRef array = CMSampleBufferGetSampleAttachmentsArray(sampleBuffer, true
CFDictionaryRef dic = (CFDictionaryRef)CFArrayGetValueAtIndex(array, 0);
BOOL keyframe = !CFDictionaryContainsKey(dic, kCMSampleAttachmentKey_NotSync)
```

为什么要判断关键帧呢？因为VideoToolbox编码器在每一个关键帧前面都会输出SPS和PPS信息，所以如果本帧是关键帧，则取出对应的SPS和PPS信息。那么如何取出对应的SPS和PPS信息呢？图6-21中提到CMSampleBuffer中有一个成员是CMVideoFormatDesc，而SPS和PPS信息就存在于这个对于视频格式的描述里面。取出SPS的代码如下：

```
CMFormatDescriptionRef fmt =CMSampleBufferGetFormatDescription(sampleBuffer);
size_t sparameterSetSize, sparameterSetCount;
const uint8_t *sparameterSet;
size_t paramSetIndex = 0;// 代表sps
OSStatus statusCode = CMVideoFormatDescriptionGetH264
ParameterSetAtIndex(fmt, paramSetIndex, &sparameterSet, &sparameterSetSize,
        &sparameterSetCount, 0 );
```

同样，取出PPS的代码如下：

```
size_t pparameterSetSize, pparameterSetCount;
const uint8_t *pparameterSet;
size_t paramSetIndex = 1;// 代表pps
OSStatus statusCode = CMVideoFormatDescriptionGetH264
ParameterSetAtIndex(fmt, paramSetIndex, &pparameterSet, &pparameterSetSize,
        &pparameterSetCount, 0 );
```

这样就可以取出SPS和PPS的信息了，接着再把这一帧（有可能是关键帧也有可能是非关键帧）的实际内容提取出来进行处理。首先，取出这一帧的时间戳，代码如下：

```
CMTime pts = CMSampleBufferGetPresentationTimeStamp(buffer);
double presentationTimeMills = CMTimeGetSeconds(pts)*1000;
```

然后再取出具体的压缩后的数据，代码如下：

```
CMBlockBufferRef data = CMSampleBufferGetDataBuffer(buffer);
```

取出真正的压缩后的数据CMBlockBuffer之后，然后就可以访问这块内存并取出具体的数据了，然后写文件，具体的代码可以参考项目实例中的代码。

最后是释放编码器，首先调用VTCompressionSessionCompleteFrames方法强制编码器完成编码行为，然后调用VTCompressionSessionInvalidate方法结束编码器会话，最终调用CFRelease方法释放编码器会话。释放编码器方法的签名为：

```
- (void)endCompresseion
```

2.将编码器集成进系统

前面已将VideoToolbox成功地封装到我们自己的类中了，并且提供了调用的接口，这里会把该H264HWEncoder集成到6.2.2节给出的预览实例中，然后提供一个按钮，点击编码按钮可以将预览的内容编码并且写入一个H264文件中，点击停止则停止编码。

6.2.2节已经按照我们的设计将Camera连接到了GLImageView上面了，而基于之前的设计，Camera是输入端，它将处理完的纹理对象传递给GLImageView，而GLImageView是一个输出节点，是让用户可以在屏幕上看见预览图像的输出节点。本节即将编写的Video-Encoder也是一个输出节点，该输出节点是编码并写到磁盘中的，所以先编写一个Video-Encoder，让Camera也连接到我们的VideoEncoder上。首先建立

ELImageVideoEncoder实现ELImageInput这个Protocol，代表我们新建立的这个节点是可以被输入纹理对象的，然后编写初始化方法，可以让客户端代码创建该节点的时候将编码参数传递进来，初始化方法定义如下：

```
- (id) initWithFPS: (float) fps maxBitRate:(int)maxBitRate
    avgBitRate:(int)avgBitRate encoderWidth:(int)encoderWidth
    encoderHeight:(int)encoderHeight;
```

由于我们实现了ELImageInput这个Protocol，所以需要实现保存输入纹理的方法和渲染纹理的方法。新建一个ELImageTextureFrame指针类型的纹理来保存输入的纹理对象，然后建立一个EncoderRenderer来渲染输入的纹理对象，而这个渲染过程其实就是将输入纹理对象渲染到编码纹理对象之上，但是这里有两点需要注意。

第一点，由于要将纹理对象渲染之后再放到编码器中，因此会涉及OpenGL坐标系到计算机坐标系的转换，故而这里在渲染到目标纹理对象的时候要把整个图像HFlip一下，即将每个坐标的Y顶点0、1互换一下。物体坐标如下：

```
static const GLfloat imageVertices[] = {
    -1.0f, -1.0f,
     1.0f, -1.0f,
    -1.0f,  1.0f,
     1.0f,  1.0f,
};
```

纹理坐标如下：

```
static const GLfloat hFlipTextureCoordinates[] = {
    0.0f, 1.0f,
    1.0f, 1.0f,
    0.0f, 0.0f,
    1.0f, 0.0f,
    };
```

第二点，由于渲染到的目标纹理对象需要交给编码器进行编码，即我们的目标纹理对象必须与一个CVPixelBuffer关联起来，所以在构建目标纹理对象的时候势必不能与创建普通的纹理对象一样。其实，在

ELImageVideoCamera节点中已经实现了将一个纹理对象和一个CVPixelBuffer关联起来的操作，只不过这里关联的纹理格式是RGBA的格式，而存储的像素格式会使用iOS特有的BGRA格式。代码如下：

```
CVOpenGLESTextureCacheCreateTextureFromImage (kCFAllocatorDef
        ault, coreVideoTextureCache, renderTarget, NULL, GL_TEXTURE_2D,
        GL_RGBA, _width, _height, GL_BGRA,GL_UNSIGNED_BYTE, 0,
        &renderTexture);
```

其中renderTarget就是一个CVPixelBuffer的引用，如此一来renderTexture这个纹理对象就是该节点的渲染目标了，只不过这里的渲染行为是将图像做一个上下翻转。再来看最关键的一步，即渲染完成之后，实际上渲染的内容会存在于这个CVPixelBuffer中，这样我们就可以将该renderTarget传递给编码器进行编码操作了。当然，在交给编码器之前先要进行锁定，编码器使用完毕之后要解锁这个CVPixelBuffer。具体代码可以参照本节的代码示例。

3.iOS平台高层次的硬件编解码API的理解

先扩展一下本节的知识面，了解一下iOS系统为开发者提供的非常强大的多媒体API库，当然万变不离其宗，这些高级的API都是基于我们讲解过的最底层的VideoToolbox进行封装的，并且提供了单一的接口来完成某些具体的事情。

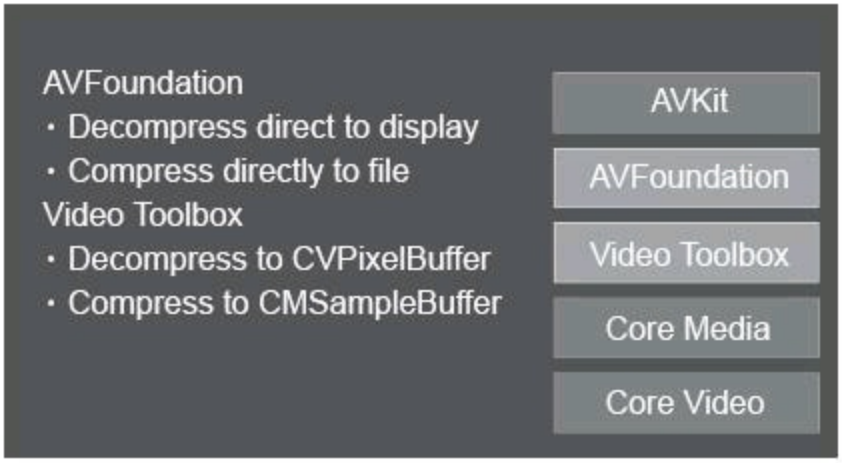


图　6-22

如图6-22所示，iOS平台提供的多媒体接口是从底层到上层的结

构，之前都是直接使用VideoToolbox，而AVFoundation是基于VideoToolbox进行的封装。它们的关注点不一样：VideoToolbox更关注编码成为内存中的CM-SampleBuffer结构体，以及解码成为主内存（或者理解为显存）中的CVPixelBuffer结构体，而AVFoundation则更关注于解码后直接显示以及直接编码到文件中。下面重点来看一下AVFoundation这个层次提供的几个主要API。

（1）AVAssetWriter

从名字上可以看出，这是为了写入本地文件而提供的API，该类可以方便地将图像和音频写成一个完整的本地视频文件，首先来看一下前面是如何利用VideoToolbox编码视频文件的，如图6-23所示。

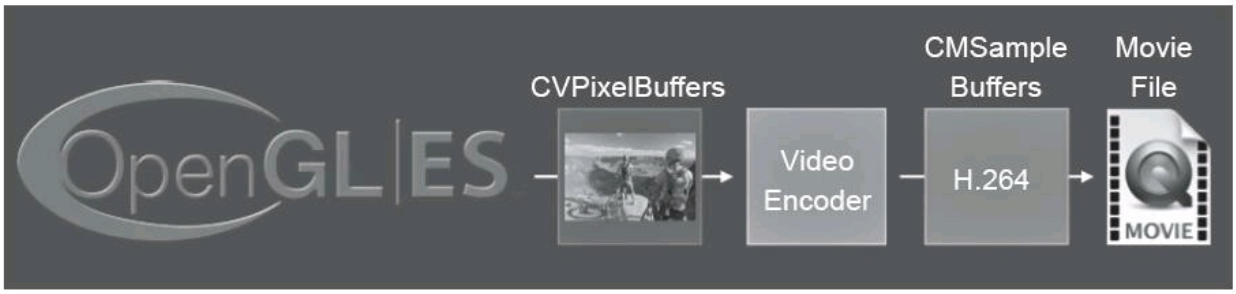


图　6-23

先从OpenGL ES中拿到纹理对象，然后关联到CoreVideo这个framework里面提供的CVPixelBuffer中去，然后提供给VideoToolbox进行编码，最终将其写成H264文件或者封装到一个视频文件中去。但是如果仅仅是写本地文件，其实并不需要这么麻烦，我们可以利用iOS封装好的API很简单地完成这种场景，如图6-24所示。

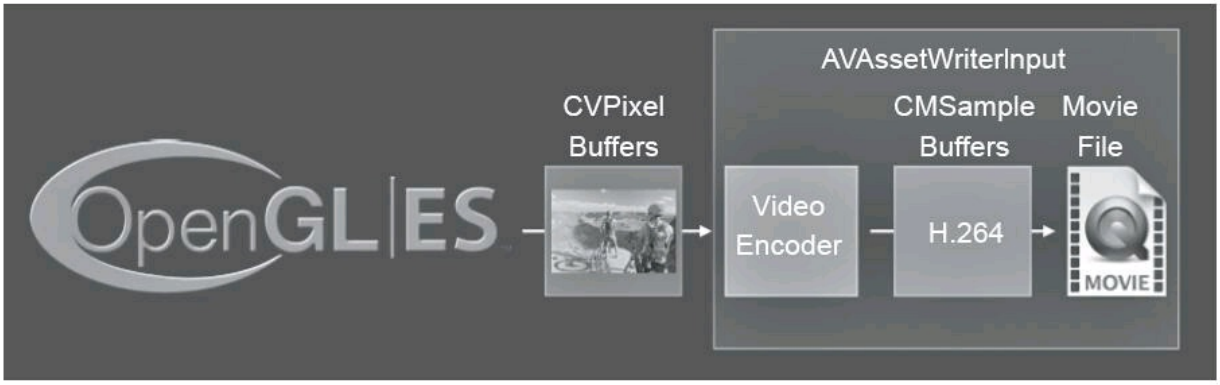


图　6-24

可以看到AVAssetWriterInput将编码器以及后续的处理和封装的工作组装到了一起，提供了更单一的接口调用，其完成的功能也更加清晰，这也是iOS平台提供的多媒体API强大的地方，当然这也仅限于本地文件。如果是直播场景，将编码后的码流推送到流媒体服务器的话，就不能使用这个API了，所以在工作中需要根据场景选择不同的技术实现。具体代码就不做展示了，读者想要继续了解的话，建议在GitHub上找到GPUImage框架，里面有一个类GPUImageMovieWriter非常详尽地使用了这个API，大家可以阅读其源码。

（2）AVAssetReader

从名字上来看，这是为了读取本地文件而存在的一个类，该类可以方便地将本地文件中的音频和视频解码出来。下面来看一下如何利用VideoToolbox解码视频，然后再使用Video-Toolbox编码成为一个本地的视频文件的整体过程，如图6-25所示。

虽然前面还没有介绍如何使用VideoToolbox解码H264数据，但是大家可以认为解码就是编码的一个逆过程，开发者需要自己编写很多代码来控制解码器的很多状态，包括输入、输出等，而在AVFoundation中，iOS平台则直接将其封装成为一个更高级的API，如图6-26所示。

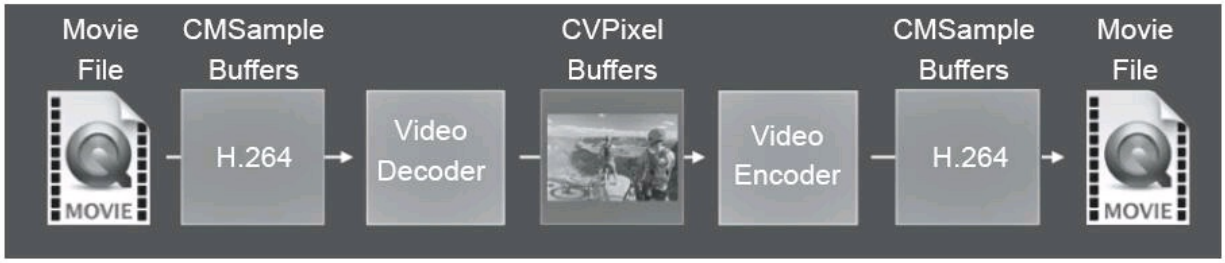


图　6-25

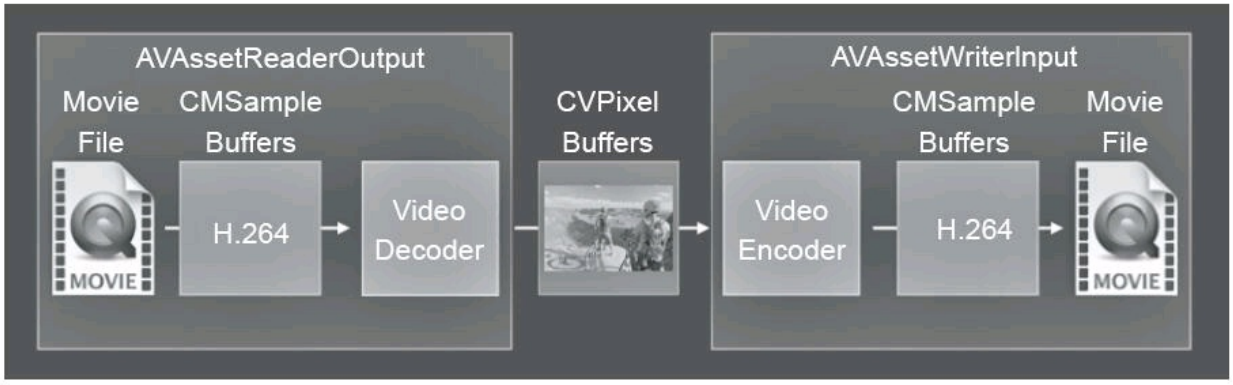


图　6-26

AVAssetReaderOutput也只支持本地文件的解码，但不支持网络媒体文件的输入，同样，这里也不再进行代码展示，如果读者有兴趣，可以参考GPUImage框架中的GPUImageMovie这个类，其在离线处理操作的场景下对AVAssetReaderInput这个API的使用有比较详尽的展开。

（3）AVAssetExportSession

这个类的使用场景比较多，比如拼接视频、合并音频与视频、转换格式，以及压缩视频等多种场景，其实是一个更高层次的封装，如图6-27所示。

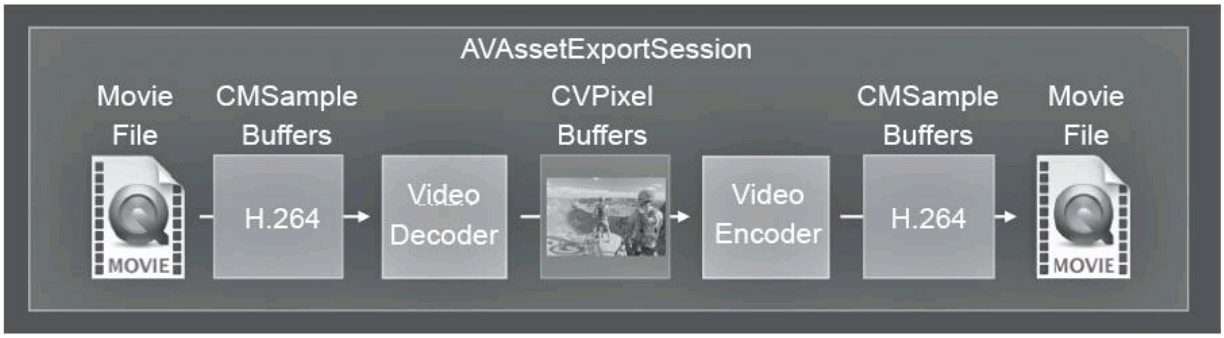


图　6-27

通过图6-25可以看到，该类不允许我们做中间的处理操作，很显然其是一个更高层次的封装，它只允许我们设置一些预设值和提供输入输出文件等，所以相比使用VideoToolbox，或者使用AVAssetReader和AVAssetWriter来实现，AVAssetExportSession提供的功能更单一，接口也更简单。该API只能按照预设去编码文件，而不能指定实际的码率以及帧率等细节参数，如果要实现这个功能，则只能通过AVAssetReader和AVAssetWriter来完成。所以在我们的工作中，不同的场景下选用不同的技术实现是非常重要的，这不单单会影响开发的效率，还会直接影响产品的体验。学习iOS平台多媒体的开发，了解这些API是非常有好处的。

本节的代码示例是代码仓库中的VideoToolboxEncoder工程，进入预览界面之后点击编码按钮开始编码，点击停止按钮之后，编码结束，然后可以利用Xcode的Devices将编码之后的H264文件导出到电脑上，最后可以利用ffplay播放这个H264文件，以观看效果。

当使用ffplay观看H264文件的时候，大家可以看到，刚才所预览的视频现在播放的速度非常快，这是因为裸H264的流是没有时间戳概念的，它不像音频那样只要指定好采样率、声道数、表示格式，声音的渲染端就可以以固定的频率来渲染音频数据了，虽然SPS和PPS中也指明了视频的宽、高以及fps等信息，但是普通的播放器并不支持按照一定的频率进行播放，播放这个H264文件的时候也没有声音，不过不要着急，第7章中将会完成一个实例——视频的录制App，其会输出一个完整的视频，视频播放的时候会按照正常的速率，当然声音也可以正常播放。

6.5　本章小结

本章的内容比较多，涉及Android和iOS两个平台音视频的采集和编码，第7章将会完成一个视频录制的应用，第7章完全是以本章内容作为基础的。所以读者在学习本章的时候可以放慢脚步，深入学习，因为只有充分了解本章的一些细节，才能在接下来章节的学习中更加游刃有余，同时在日常工作中遇到问题时也会比较容易解决。

第7章　实现一款视频录制应用

第6章讲解了音频和视频的采集以及相应的编码知识，本章会利用前面章节的基础知识完成一款视频录制的应用，分别从Android和iOS两个平台进行实现，大家在学习完本章内容之后会对视频录制有一个整体的认识，现在让我们开始吧！

7.1　视频录制的架构设计

本节先来看一下视频录制的架构设计，大家在工作中完成一个项目或者产品的某一个迭代时，首先应该根据用户场景进行设计，这里所说的设计绝不是写一大堆设计文档，而是对功能点进行拆分、细化，然后再为每个模块找到最合理的实现。只有先对全局有一个整体的认识，才能清楚接下来应该如何实现每一个模块。

下面分析一下需要完成的场景：将用户的声音和画面全部录制下来，生成一个MP4文件，同时用户可以自己选择是否需要开启背景音乐。场景看上去挺简单，然而针对这些场景如何做出一个合理的架构设计却是一项比较复杂的工作，从录制视频的角度来讲，每个平台都有自己独特的API可供开发者调用，但是要想合理地使用这些API，还得将业务场景拆分为技术模块，才能确定其实现细节。基于业务场景分析，该应用可拆分为两部分：一部分是音频部分，一部分是画面部分。基于第6章中的基础知识，可以先对音频做出以下架构设计，如图7-1所示。

乍一看，可能会觉得这个架构比较复杂，毕竟该架构图（见图7-1）中包含了两个平台的音频架构。不过，大家不要着急，先来逐一梳理一下，等分析结束之后，大家就会觉得非常清晰明了了。

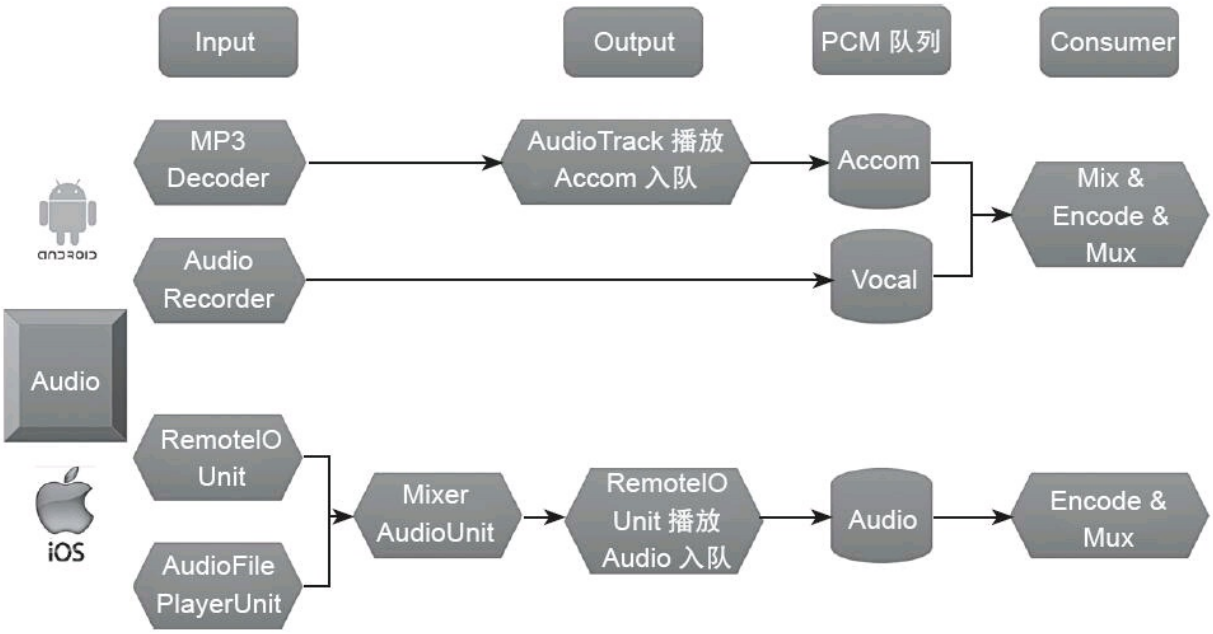


图　7-1

先从高层次来解读这张架构图，图7-1的最上面的那一部分，从左到右依次如下。

第一个模块是Input模块，代表输入部分，第一个输入是麦克风用来采集用户的声音；另外一个输入是伴奏文件解码器，用来解码用户选择的背景音乐，所以输入部分将由这两部分共同实现。

第二个模块是Output，代表输出模块，第一个输出模块就是利用渲染音频的API将背景音乐的声音播放出来，在iOS平台上如果用户戴着耳机的话，可以将用户自己发出的声音同时播放出来，以达到耳返监听的功能；第二个输出模块是记录数据，需要将背景音乐和用户声音的数据保存下来；那保存到哪里呢？就是接下来介绍的第三个模块。

第三个模块是PCM队列，应该把背景音乐和用户声音的PCM数据存入队列中，该队列应该能够保证多线程访问时候的线程安全性。

最后一个模块就是Consumer模块，该模块负责从第三个模块的队列里面取出音频PCM数据，进行音频AAC的编码，并且最终封装到MP4文件中，属于一个消费者的角色，所以我们称之为消费者模块。

在对模块进行了拆分之后，再在Android和iOS平台上确定其技术实现，对应到每一个平台都应该从这几个模块来进行分析，确定应该使用什么技术来实现模块理应完成的职责。

所以下面再来看一下图7-1的第二行，Android平台的实现。首先是Input模块，基于之前掌握的知识我们知道，对于音频的采集需要使用AudioRecord这个API来采集用户的声音，同时需要将采集出的音频放入第三个模块即人声的PCM队列中，对于提供的伴奏文件（MP3格式或者M4A格式），可以利用FFmpeg写一个伴奏的解码器，将背景音乐文件解码为PCM数据并放入PCM队列（解码器内部维护的队列）中，以供后续的播放器API播放给用户；然后是Output模块，从目前来讲，在Android设备上对于耳返监听的功能并没有一个特别成熟的方案，所以在这里就不实现Android的耳返功能了，但是必须要让用户听得到背景音乐，所以我们使用AudioTrack来播放前面解码好的伴奏，同时把播放的伴奏放入第三个模块中的伴奏PCM队列中；而对于PCM队列，可以使用C++自行编写一个线程安全的链表来提供先入先出的接口以完成队列的功能，并且为了性能考虑，可以将该队列改造成为一个Blocking Queue的形式；最后一个模块是Consumer模块，即开启一个线程在后台

轮询伴奏和人声的PCM队列，取出伴奏的数据和人声的数据，合并（Mix）成一轨音频数据，利用MediaCodec或者libfdk_aac进行编码，最终利用FFmpeg的Muxer模块将编码后的AAC数据封装到MP4文件的声音轨道中。这样Android平台上的技术选型分析就结束了，读者可以对照架构图7-1的Android部分梳理一下。

接下来看一下图7-1的第三行，iOS平台的实现，首先是Input模块，对于音频的采集，应该使用RemoteIO这个AudioUnit，启用它的InputElement来采集人声数据，伴奏的播放则采用第4章中掌握的知识，使用AudioFilePlayer这个AudioUnit进行解码伴奏，然后使用一个Mixer的AudioUnit将两轨声音Mix起来，为后续节点提供输出；其次是Output模块，这里使用RemoteIO该AudioUnit的OutputElement将MixerUnit输出的音频播放给用户听，同时将该AudioUnit注册一个回调函数，利用Converter的AudioUnit转换为SInt16采样格式表示的PCM数据放入到音频队列中；第三是PCM队列模块，可以使用C++自行编写一个线程安全的链表，提供先入先出的接口以完成队列的功能，并且为了性能考虑，可以将该队列改造成为一个Blocking Queue的形式，而该队列的代码可以与Android平台共享一份代码进行使用；最后是Consumer模块的实现，开启一个线程在后台轮询PCM队列，取出PCM数据之后使用AudioToolbox或者libfdk_aac进行编码，最后利用FFmpeg将编码后的AAC数据封装到MP4文件的声音轨道中，这个模块主要是使用C++语言调用FFmpeg的API来实现的，所以可以与Android平台共享一份代码。读者可以对照架构图7-1的iOS部分再梳理一下。

音频架构就为大家分析到这里，接下来再分析一下视频部分的架构，如图7-2所示。

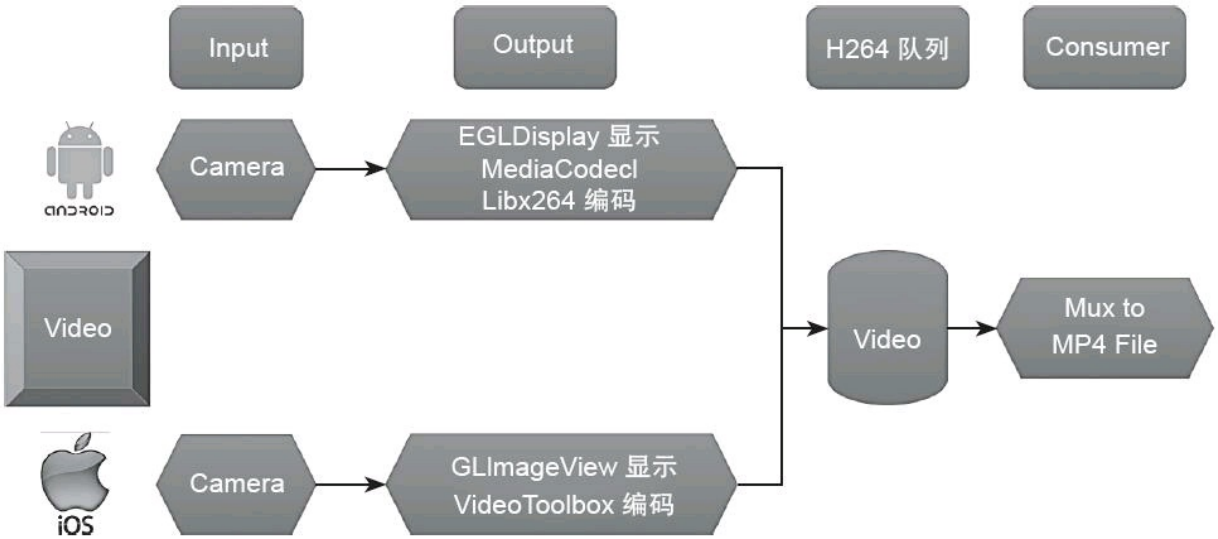


图　7-2

相比于音频的架构设计，视频的架构相对更简单一些，可以看到图7-2的第一行，也是拆分为输入、输出、队列和消费者四个模块，每个模块所完成的功能这里就不再详细的叙述了，相信读者可以很直观的理解，接下来确定一下各个模块在两个平台的技术选型。

对于Android平台，本书的第6章已经开发出了一个预览和编码的实例，对于Input模块的实现，可以直接使用Camera这个API。而对于Output模块则分为两部分：一部分是预览，通过使用EGL和OpenGL并且结合Java层的SurfaceView来实现；另外一部分是编码，对于视频的编码，优先使用硬件编码，如果存在兼容性问题，则使用libx264软件编码作为保底的方案来实现，最终编码成为H264的数据。与第6章的案例不同的是，编码之后的H264数据不可以再写入文件，而是要放到第三个模块即H264的队列中，对于H264的队列，同样可以使用一个线程安全的链表来实现，链表中的每一个节点元素都是H264的数据包。最后一个模块是消费者模块，我们在Consumer这个模块中取出队列中的H264数据包利用FFmpeg的Mux模块封装到MP4的视频轨道中，其恰好与之前封装到该文件中的音频轨道组成一个完整的MP4文件。

在iOS平台上，Input模块的实现自然会使用系统提供给开发者的Camera这个API来实现。Output模块分为预览和编码两个部分，其实在第6章中已经开发出了一个预览和编码的实例，预览的实现直接使用EAGL和OpenGL再结合自定义的一个UIView来完成；编码的实现则使用VideoToolbox进行硬件编码，不同于第6章实例的是，编码之后的

H264数据不要直接写入文件中，而是应该放到H264的队列中。所以第三个模块就是H264队列模块，对于H264的队列，可以使用一个线程安全的链表来实现，链表中的每一个节点元素都是H264的数据包，该模块的实现可以与Android平台共享一份代码。最后一个模块是消费者模块，从Consumer模块取出队列中的H264数据包，然后利用FFmpeg的Mux模块封装到MP4的视频轨道部分，正好与之前封装到该文件中的音频轨道共同组成一个完整的MP4文件，该模块主要使用C++语言调用FFmpeg的API来实现，所以可以与Android平台共享一份代码。

其实，通过上述的分析不难看出来，在队列的实现部分以及Consumer部分，两个平台的实现是一模一样的，所以可以将这部分代码抽象出来的统一接口，使用C++语言来完成，共同运行在两个平台之上，这样做不仅可以提升开发效率还能够降低维护成本。上述的架构设计阶段还缺少风险分析，对于整个架构的风险分析来说，主要是硬件编码器的兼容性问题，以及如果在Android平台使用软件编码器的话，软件编码器的性能问题，还有一个风险就是音视频是分开采集的，是否会存在音视频同步的问题。对于测试用例来说，在测试完APP定位的主流机型和主流系统之后，可以针对于iOS 8.0系统以及Android 4.3系统来做一个边缘测试，因为硬件编码的API都是以这两个版本的系统为基础开放给开发者的。接下来的每一节均将会完成一个模块的代码实现，本章最终会按照整体架构图实现一个完整的系统。

7.2 音频模块的实现

本节会基于第6章的音频采集代码继续开发，与第6章不同的是，这里采集的音频数据需要放入队列中，而不是写入文件。本节首先来介绍音频队列的实现，然后在Android平台和iOS平台分别给出具体的实现；由于第6章中对于具体如何采集音频已经介绍得非常详细了，所以本节会偏重于如何将采集的数据放入队列中，以及如何播放背景音乐。

7.2.1 音频队列的实现

无论是Android平台还是iOS平台，所使用的音频队列都是一致的，使用C++来实现一个音频队列，就可以保证在两个平台上可以共同使用。下面来看一下这个队列的具体实现。

首先，确定该队列中存放的元素，结构体定义如下：

```
typedef struct AudioPacket {
    short * buffer;
    int size;
    AudioPacket() {
        buffer = NULL;
        size = 0;
    }
    ~AudioPacket() {
        if (NULL != buffer) {
            delete[] buffer;
            buffer = NULL;
        }
    }
} AudioPacket;
```

该结构体定义了一个AudioPacket，每采集一段时间的PCM音频数据，就封装成为一个这样的结构体对象，然后放入到PCM队列中。这里提到的队列是线程安全的队列，可以自行编写一个链表来实现队列的功能，链表节点的元素定义如下：

```
typedef struct AudioPacketList {
    AudioPacket *pkt;
    struct AudioPacketList *next;
    AudioPacketList(){
        pkt = NULL;
        next = NULL;
    }
} AudioPacketList;
```

可在上述队列中定义一个mFirst节点来指向头部结点，定义一个mLast节点指向尾部节点，为了保证线程的安全性，需要定义以下两个变量：

```
pthread_mutex_t mLock;
pthread_cond_t mCondition;
```

该队列提供了两个最重要的接口，分别是push和pop，下面分别定义为put和get方法，代码如下：

```
int put(AudioPacket* audioPacket);
```

在put方法的实现中，首先会判断是否abort了该队列，如果abort了则代表队列不再需要操作，直接返回；如果不是abort状态，那么需要将客户端代码封装好的AudioPacket实例组装成一个链表节点放入链表中。当然为了保证线程的安全性，在放入链表的过程中要先上锁，然后操作链表。在放入链表结束之后发出一个signal指令，因为get方法是有可能被block的（由于这里实现的是一个Blocking Queue），所以要通过发送signal指令来告诉block住的线程可以继续从队列中获取元素，最后释放锁。代码如下：

```
if (mAbortRequest) {
    delete pkt;
    return -1;
}
AudioPacketList *pkt1 = new AudioPacketList();
if (!pkt1)
    return -1;
pkt1->pkt = pkt;
pkt1->next = NULL;
pthread_mutex_lock(&mLock);
if (mLast == NULL) {
    mFirst = pkt1;
} else {
    mLast->next = pkt1;
}
mLast = pkt1;
pthread_cond_signal(&mCondition);
pthread_mutex_unlock(&mLock);
return 0;
```

下面是该队列提供的get接口，方法原型如下：

```
int get(AudioPacket **audioPacket);
```

该get方法主要实现将mFirst指向的节点拿出来返回给客户端代码，并且将mFirst指向它的下一级节点，如果当前队列为空则block住（用来实现Blocking Queue）get方法，等待有元素再放进来或者abort该队列之后才会返回，实现代码如下：

```
AudioPacketList *pkt1;
int ret = 0;
pthread_mutex_lock(&mLock);
for (;;) {
    if (mAbortRequest) {
        ret = -1;
        break;
    }
    pkt1 = mFirst;
    if (pkt1) {
        mFirst = pkt1->next;
        if (!mFirst)
            mLast = NULL;
        mNbPackets--;
        *pkt = pkt1->pkt;
        delete pkt1;
        pkt1 = NULL;
        ret = 1;
        break;
    else {
        pthread_cond_wait(&mCondition, &mLock);
    }
}
pthread_mutex_unlock(&mLock);
return ret;
```

现在，最核心的两个方法已经全部实现了，剩下的就是abort方法了，该方法需要将布尔型变量mAbortRequest设置为true，并且同时要发送一个signal指令，以防止别的线程会被block在获取数据的接口中（即get方法中）。还有一个销毁方法，就是遍历队列中所有的元素，然后释放它们。至此队列实现就完成了，读者可以到代码目录中去查看一下源码，对比进行分析。

7.2.2 Android平台的实现

本节将完成Android平台上Input模块的实现，相关的基本代码在第6章中已经实现过了，本节会根据当前的整个项目做一下结构调整，最重要的是，这里要添加上背景音乐，这在之前的第4章中也有过介绍，所以本节就将它们综合运用起来，最终将采集到的人声和解码的背景音乐的两部分PCM音频数据分别入队。结构图如图7-3所示。

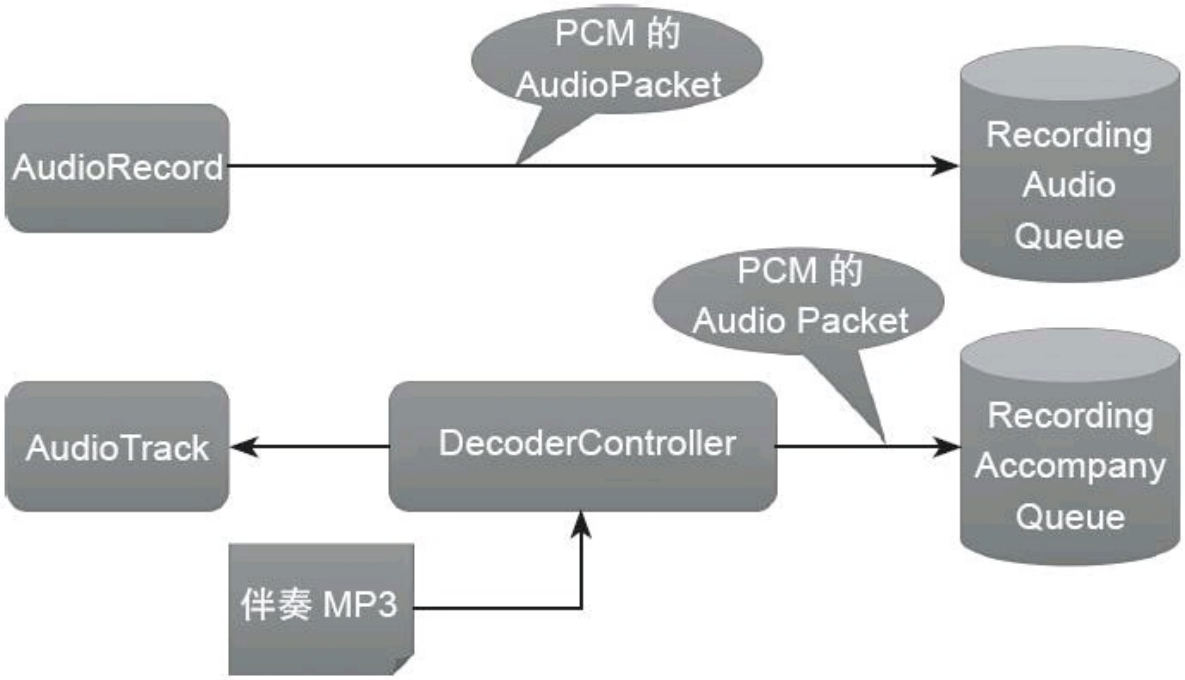


图 7-3

1.伴奏的解码与播放

因为要为录制的视频增加背景音乐，所以这里又要回到伴奏的解码与播放中，第4章中已经实现过伴奏的解码与播放，所以这里将基于第4章的项目继续进行开发。

先来回顾一下第4章中播放器是如何工作的，首先，基于FFmpeg建立一个伴奏的解码器；然后建立一个解码控制器来开启一个线程不断地调用解码器，解码出来的音频数据将放入队列之中，当队列元素达到一个阈值的时候就暂停解码，如果收到signal指令就继续调用解码器解

码；同时解码控制器将为客户端提供一个readSamples方法，该方法的实现中，会不断地从队列中获取解码好的伴奏PCM数据并返回给客户端，此外，它还会监测队列的大小，当队列中元素数目小于某个阈值的时候，发送一个signal指令让解码线程继续工作；接着客户端将使用AudioTrack作为播放PCM数据的播放器，客户端会创建一个播放器线程，从而不断地调用解码控制器的readSamples方法，最后将获取出来的PCM数据提供给AudioTrack进行播放；当停止播放的时候，首先停止解码控制器，然后停止并释放AudioTrack的资源，这就是播放器的流程。总体来描述就是，解码控制器中的解码线程作为生产者将解码的PCM数据放入到一个解码队列当中，而在客户端中，AudioTrack所在的播放线程则作为消费者（调用readSamples方法），不断地从解码队列中拿出PCM数据播放给用户。

在当前场景之下，除了让用户可以听到伴奏，同时还需要将用户听到的伴奏存储下来，所以首先要修改初始化方法，在初始化方法中需要添加一个伴奏音频队列的初始化。注意该队列和上述的解码队列不是同一个队列，解码队列会作为解码线程和播放器这一对生产者和消费者之间的桥梁，而这里的队列是指以播放器作为生产者，Consumer模块的编码线程作为消费者，所以该队列是这两个模块之间的桥梁。为了全局都可以访问到该队列，需要将所有队列都放到一个单例模式设计的池子中，初始化两个队列的代码如下：

```
packetPool = LiveCommonPacketPool::GetInstance();
packetPool->initDecoderAccompanyPacketQueue();
packetPool->initAccompanyPacketQueue(sampleRate, CHANNEL);
```

接下来要修改readSamples方法，因为该方法不再只是为上层的播放器PCM提供数据，同时也是第三个模块的伴奏音频队列的生产者，职责发生了变化，所以需要将该解码控制器中的方法名readSamples修改为readSamplesAndProducePacket，在方法的实现中，将该伴奏的AudioPacket中的数据复制到客户端之后，不要删除该伴奏的AudioPacket，而是直接将它放入伴奏音频队列之中。代码如下：

```
packetPool->pushAccompanyPacketToQueue(accompanyPacket);
```

这样就把第4章中的音频播放器项目适配到我们的系统之中了，是

不是很简单呢？读者可以到代码仓库中找到代码从头再分析一遍。

2.音频入队

第6章中直接把AudioRecord采集出来的音频写到文件中去了，但是在当前的场景下是不能够直接写文件的，而是应该放到人声的PCM队列之中，所以要在Native层新建立一个RecordProcessor类，主要负责维护人声的采集以及声音的编码线程。下面来看一下这个类中的方法及其实现，首先是初始化方法：

```
void initAudioBufferSize(int sampleRate, int audioBufferSize);
```

该方法主要是将采样率以及队列中每一个元素的大小作为参数传入进来，在该方法的实现中，首先将这两个参数赋值给全局变量，并且按照audioBufferSize分配一块audioBuffer的存储空间，用于积攒采集到的音频数据，最后构造出编码器，并且初始化这个编码器，编码器模块将在7.2.3节中进行实现。

接下来再看第二个方法，该方法用于积攒本来要在Java层写入文件的PCM的Buffer，并且将其放入队列中。为什么要积攒呢？因为在不同的设备上Java层的AudioRecorder每次采集出来的buffer其大小有可能是不同的，所以要在该方法内部做一个积攒，当积攒到初始化方法中设定的audioBufferSize时，再将这一段buffer构造成为一个AudioPacket，然后放入到人声队列中，下面先来看一下积攒的逻辑，代码如下：

```
void pushAudioBufferToQueue(short* samples, int size) {
    int samplesCursor = 0;
    int samplesCnt = size;
    while (samplesCnt > 0) {
        if ((audioSamplesCursor + samplesCnt) < audioBufferSize) {
            cpyToAudioSamples(samples + samplesCursor, samplesCnt);
            audioSamplesCursor += samplesCnt;
            samplesCursor += samplesCnt;
            samplesCnt = 0;
        } else {
            int subFullSize = audioBufferSize – audioSamplesCursor;
            cpyToAudioSamples(samples + samplesCursor, subFullSize);
            audioSamplesCursor += subFullSize;
            samplesCursor += subFullSize;
            samplesCnt -= subFullSize;
            flushAudioBufferToQueue();
        }
```

```
    }
}
```

对这段代码的分析具体如下，首先定义一个游标来表示访问到samples这个buffer的哪个位置，然后进入一个循环，将该buffer的数据全都拷贝出来，直到全部都使用完毕了再退出这个循环，循环内部首先会判断在全局的audioBuffer中是否有足够的空间用来存放该buffer的有效数据（就是buffer中还没被取出来的数据），如果可以存放，则将有效数据全部都拷贝到全局audioBuffer中，然后对全局的audioBuffer的游标和当前buffer的游标增加对应的数值，将sampleCnt设置为0，表示当前buffer使用完毕；如果全局的audioBuffer存放不了的话，那么就先计算出全局的audioBuffer还能存放多少，代码如下：

```
int subFullSize = audioBufferSize – audioSamplesCursor;
```

上述代码可实现将subFullSize个sample从当前buffer中拷贝到全局的audioBuffer中去，然后将全局audioBuffer对应的游标增加subFullSize，将当前buffer的有效采样数目减去这个subFullSize，最后再把全局的audioBuffer封装成为一个AudioPacket放入人声队列中。将audioBuffer封装为AudioPacket，并且放入队列中的代码如下：

```
short* packetBuffer = new short[audioSamplesCursor];
memcpy(packetBuffer, audioSamples, audioSamplesCursor * sizeof(short));
AudioPacket * audioPacket = new AudioPacket();
audioPacket->buffer = packetBuffer;
audioPacket->size = audioSamplesCursor;
packetPool->pushAudioPacketToQueue(audioPacket);
```

上述代码就是flushAudioBufferToQueue方法的具体实现，实际上就是分配一个新的内存空间，将全局变量audioBuffer的内容拷贝进去，然后封装成AudioPacket，最终放入到队列中。

该类的最后一个方法就是销毁方法，该方法的实现非常简单，首先要释放掉在初始化方法中分配的全局audioBuffer这块内存，然后需要调用编码器的销毁方法，最终再释放编码器这个类。

至此就完成了伴奏和人声入队的操作，读者如果想了解全部的代码

可以在代码仓库中参考对应的代码。

7.2.3　iOS平台的实现

本节要完成iOS平台的音频采集与伴奏播放，同时将采集的人声PCM数据和播放的伴奏PCM数据合并（Mix）成一轨PCM数据，并存入队列中，人声的采集在第6章中已经有了具体的实现，并且直接编码到磁盘的文件中去了，伴奏的播放在第4章中也有过实现，本节会基于这两个实例加以改造，来满足当前系统的需求。

1.伴奏的解码与播放

第4章介绍的播放器有两种实现方式，第一种是使用AudioFilePlayer这个AudioUnit再加上RemoteIO来播放伴奏。还记得第6章中曾经提到过，有一个MixerUnit是为了后续扩展使用的吗？扩展的地方就在这里了，就是将第4章中提到的AudioFilePlayer的AudioUnit也连接上MixerUnit，这样用户听到的就是伴奏和人声Mix到一块的声音了，结构如图7-4所示。

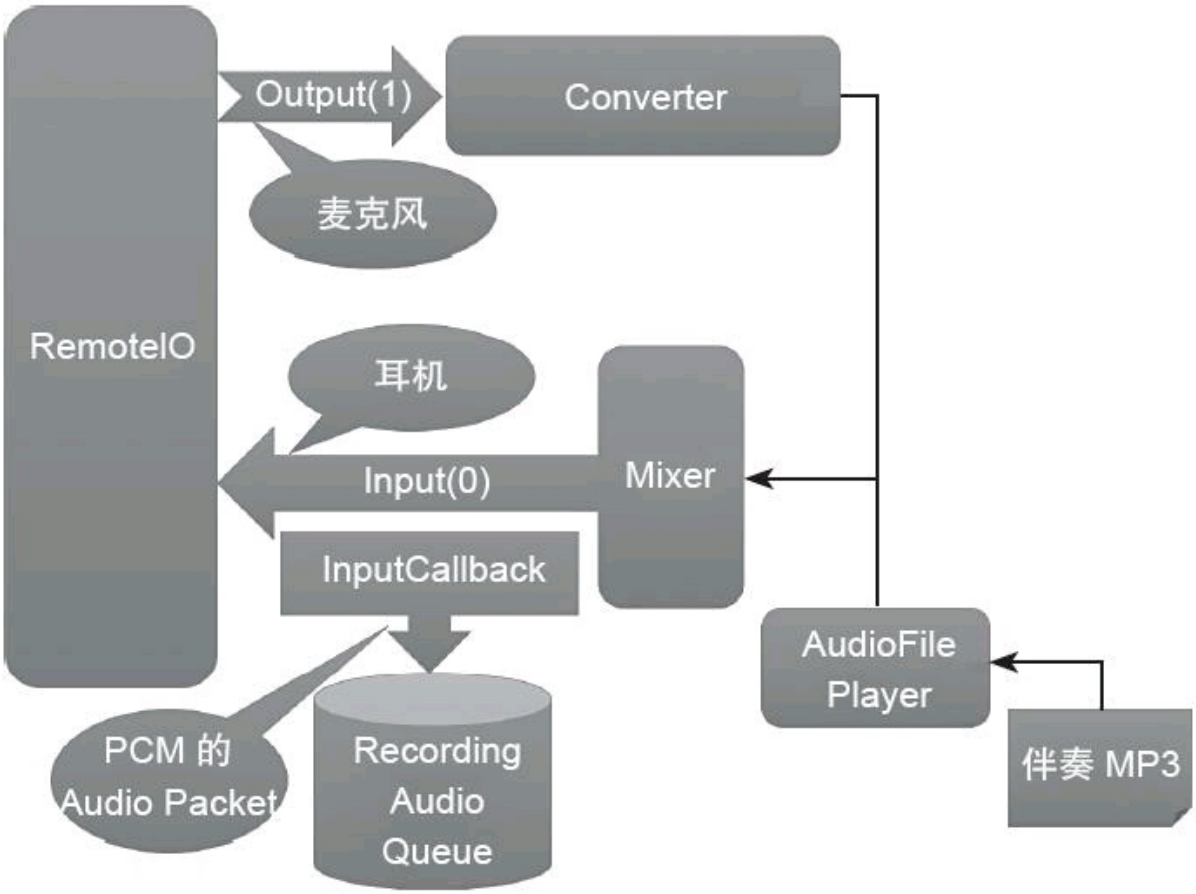


图　7-4

本节将重点来看一下伴奏的解码以及其与MixerUnit的连接操作，具体的入队操作将会在第2个小节中介绍。下面将基于第6章中的AudioRecorder录音实例继续进行开发，首先从构造AUGraph的类中找到方法addAudioUnitNodes，添加以下代码：

```
AudioComponentDescription playerDescription;
bzero(&playerDescription, sizeof(playerDescription));
playerDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
playerDescription.componentType = kAudioUnitType_Generator;
playerDescription.componentSubType = kAudioUnitSubType_AudioFilePlayer;
AUGraphAddNode(_auGraph, &playerDescription, &mPlayerNode);
```

上述代码的含义其实就是向AUGraph中加入AudioFilePlayer这个AUNode，然后在get-UnitsFromNode方法中加入以下代码：

```
AUGraphNodeInfo(mPlayerGraph, mPlayerNode, NULL, &mPlayerUnit);
```

上述代码是找出mPlayerNode这个AUNode对应的AudioUnit，并赋值给全局变量mPlayer-Unit，以方便后续设置参数。然后在setUnitProperties方法的最后一行为新找出来的Audio-Unit设置声音格式：

```
AudioUnitSetProperty(mPlayerUnit, kAudioUnitProperty_StreamFormat,
        kAudioUnitScope_Output, 0, &_clientFormat32float,
        sizeof(_clientFormat32float));
```

下一步就将该mPlayerUnit连接到Mixer上，首先需要更改Mixer这个AudioUnit的输入Unit的数目，即将在该函数中定义的局部变量mixerElementCount更改为2，代表有两路输入要给到Mixer这个AUNode。由于现在新增的这个Player的AUNode在整个AUGraph中还是独立存在的，所以要将该AUNode连接到mixerNode的bus为1的输入端，代码如下：

```
AUGraphConnectNodeInput(_auGraph, mPlayerNode, 0, _mixerNode, 1);
```

在执行完AUGraphInitialize方法之后，要为Player这个AudioUnit配置参数，配置过程如下，首先客户端代码将需要播放的本地伴奏路径传递过来，将其设置为全局变量playPath，所以首先需要将想要播放的文件设置给Player Unit，代码如下：

```
AudioFileID musicFile;
CFURLRef songURL = (__bridge  CFURLRef) _playPath;
AudioFileOpenURL(songURL, kAudioFileReadPermission, 0, &musicFile);
AudioUnitSetProperty(mPlayerUnit, kAudioUnitProperty_
        ScheduledFileIDs, kAudioUnitScope_Global, 0, &musicFile,
        sizeof(musicFile));
```

紧接着来看一个对于AudioFilePlayer这个AudioUnit来说非常重要的概念，即Scheduled-AudioFileRegion，该结构体是AudioToolbox里面提供的一个结构体，从名字上来看即可知道，Scheduled Audio File Region是对于AudioFile进行访问计划的区域，其实该结构体就是用来控制AudioFilePlayer的，结构体里面可以设置的内容具体如下。

·mAudioFile：设置为要播放的音频文件的AudioFileID。

·mFramesToPlay：要播放的音频帧数目，可以通过获取要播放的AudioFile的Format以及总的Packet数目来计算。

·mLoopCount：用来设置循环播放的次数。

·mStartTime：用来设置开始播放的时间，拖动（Seek）操作就是通过这个参数来设置的。

·mCompletionProc：用来设置音频播放完成之后的回调函数。

·mCompletionProcUserData：用来设置回调函数的上下文。

在配置好该结构体之后，将它设置给AudioFilePlayer这个Unit：

```
AudioUnitSetProperty(mPlayerUnit, kAudioUnitProperty_ScheduledFileRegion,
        kAudioUnitScope_Global, 0,&rgn, sizeof(rgn))
```

最后给出AudioFilePlayer的最后一部分配置：

```
UInt32 defaultVal = 0;
AudioUnitSetProperty(mPlayerUnit, kAudioUnitProperty_ScheduledFilePrime,
        kAudioUnitScope_Global, 0, &defaultVal, sizeof(defaultVal));
AudioTimeStamp startTime;
memset (&startTime, 0, sizeof(startTime));
startTime.mFlags = kAudioTimeStampSampleTimeValid;
startTime.mSampleTime = -1;
AudioUnitSetProperty(mPlayerUnit, kAudioUnitProperty_ScheduleStartTimeStamp,
        kAudioUnitScope_Global, 0, &startTime, sizeof(startTime));
```

注意该配置过程必须在AUGraph初始化之后，否则是不生效的，因为在构造的AUGraph初始化之后才会初始化AudioFilePlayer这个AudioUnit，所以配置放到这个位置所设置的参数才是有效的，否则就是无效的，这与将AUGraph中的AUNode找出来赋值给AudioUnit非常类似，如果没有打开AUGraph的话，就相当于这个Graph里面的AUNode还没有被实例化，我们也不可能找出所对应的AudioUnit。

在播放的过程中，可以通过获取

kAudioUnitProperty_CurrentPlayTime来得到相对于所设置的开始时间的播放时长，从而计算出当前播放到的位置。

正确配置播放器的时机是最重要的，具体的配置信息比较简单，大家可以对照整个项目中的这一块代码示例来梳理一遍。

2.音频入队

第6章中已经使用过RemoteIO这个AudioUnit来采集音频，然后直接编码到文件中，本节不能直接写入文件中，而是封装成AudioPacket，然后放入队列中，首先找到为RemoteIO设置的回调函数，在这里，我们之前的操作是从前一级MixerNode中取出数据，然后写文件，去掉该函数中关于写文件的所有操作，然后将取出来的数据封装成为AudioPacket并放入到队列中，但是队列中要求的PCM格式是SInt16格式的数据，而从MixerNode中取出来的格式是Float32格式的数据，那么如何进行转换呢？答案非常简单，应该是使用Convert-Node，其整体流程具体如下。

第一步，先在MixerNode后面添加上一个Float32转换为SInt16的ConvertNode，但是，要直接将该ConvertNode连接到RemoteIO的输出端实际上是不可以的，因为SInt16的表示格式和RemoteIO要求的输入格式不匹配，所以需要在该ConvertNode之后再添加一级ConvertNode，用于将SInt16格式转换为Float32格式，然后将第二个ConvertNode连接到原来的RemoteIO之上，这样改造之后就可以形成当前场景下的AUGraph了。对于数据的获取，可以在第一个ConvertNode对应的Unit上添加一个RenderNotify回调函数，在函数中将数据封装为AudioPacket并送入队列。这就是整个流程，我们来共同实现一下。

首先构建Float32转换SInt16的ConvertNode，代码如下：

```
AudioComponentDescription convertDescription;
bzero(&convertDescription, sizeof(convertDescription));
convertDescription.componentManufacturer = kAudioUnitManufacturer_Apple;
convertDescription.componentType = kAudioUnitType_FormatConverter;
convertDescription.componentSubType = kAudioUnitSubType_AUConverter;
AUGraphAddNode(_auGraph, &convertDescription, &_c32fTo16iNode);
```

然后取出该AUNode对应的AudioUnit，分别构造Float32的Format与SInt16的Format，然后将这两个Format分别设置给AudioUnit的输入和输出，代码如下：

```
AudioUnitSetProperty(_c32fTo16iUnit,
        kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0,
        &c16iFmt, sizeof(c16iFmt));
AudioUnitSetProperty(_c32fTo16iUnit,
        kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0,
        &c32fFmt, sizeof(c32fFmt));
```

这里为该ConvertNode的AudioUnit增加了一个RenderNotify，注意这里设置的Render-Notify和之前设置的InputCallback是不一样的：InputCallback是当下一级节点需要数据的时候将会调用的方法，让配置的这个方法来填充数据；但是RenderNotify是不同的调用机制，RenderNotify是在这个节点从它的上一级节点获取到数据之后才会调用该函数，可以让开发者做一些额外的操作（比如音频处理或者编码文件等），所以在这个场景下使用RenderNotify这个方法会更合理，代码设置如下：

```
AudioUnitAddRenderNotify(c32fTO16iUnit, &mixerRenderNotify,
         (__bridge void *)self);
```

在mixerRenderNotify这个回调函数中，拿到的PCM数据就是SInt16格式的了，之后将PCM数据封装成为AudioPacket并放入到音频队列中，代码如下：

```
AudioBuffer buffer = ioData->mBuffers[0];
int sampleCount = buffer.mDataByteSize / 2;
short *packetBuffer = new short[sampleCount];
memcpy(packetBuffer, buffer.mData, buffer.mDataByteSize);
AudioPacket *audioPacket = new AudioPacket();
audioPacket->buffer = packetBuffer;
audioPacket->size = buffer.mDataByteSize / 2;
packetPool->pushAudioPacketToQueue(audioPacket);
```

接下来，我们来构建将SInt16格式转换为Float32格式的ConvertNode，构建AUNode的方法和之前是一样的，但设置的参数和之前恰好是相反的，代码如下：

```
AudioUnitSetProperty(_c16iTo32fUnit, kAudioUnitProperty_StreamFormat,
    kAudioUnitScope_Output, 0, &c32fFmt, sizeof(c32fFmt));
AudioUnitSetProperty(_c16iTo32fUnit, kAudioUnitProperty_StreamFormat,
    kAudioUnitScope_Input, 0, &c16iFmt, sizeof(c16iFmt));
```

最后将c32fTo16iNode连接上c16iTo32fNode，将c16iTo32fNode连接到RemoteIO上并进行播放，代码如下：

```
AUGraphConnectNodeInput(_auGraph, _c32fTo16iNode, 0, _c16iTo32fNode, 0);
AUGraphConnectNodeInput(_auGraph, _c16iTo32fNode, 0, _ioNode, 0);
```

通过上述的一系列改造，可以将PCM数据读取出来并且存入到音频队列中，同时也使得用户可以听到伴奏以及自己声音的耳返。那么PCM数据在音频队列中后续又是如何处理的呢？答案是编码，下面就来进入7.3节完成音频编码模块的实现吧。

7.3　音频编码模块的实现

本节我们来看看音频的编码部分，音频编码可以使用硬件编码也可以使用软件编码，关于如何使用FFmpeg（软件编码方式）、MediaCodec（Android平台的硬件编码方式）和AudioToolbox（iOS平台的硬件编码方式）将PCM编码成为AAC，这些内容在第6章中已经详细介绍过了，本章中音频编码模块的输入是7.2节中PCM格式的音频队列，输出则存放于另外一个AAC格式的音频队列之中。本节不会再像第6章那样讲解如何使用API编码PCM数据了，而是重点讲解如何将该编码器集成到整个系统中来，本节将以软件编码作为示例来讲解，如果读者有兴趣，可以自行将各自平台的硬件编码的实现集成进来。

类似于第6章中视频编码，音频的编码也应该放到一个单独的线程中，所以我们建立一个类AudioEncoderAdapter，利用PThread维护一个编码线程，不断地从音频队列中取出PCM数据，然后调用编码器进行编码，使其成为AAC数据，最后将AAC数据封装成为AudioPacket数据结构，并放入到AAC的队列中。其中，编码器是我们自己封装的一个AudioEncoder类，实际上就是在第6章的编码器类基础上进行的修改，下面来逐一看一下各个类的具体实现。

7.3.1 改造编码器

首先来看一下AudioEncoder类，在此之前，先回顾一下第6章中编码器的结构，当时在初始化函数中直接给出了一个文件路径，让FFmpeg帮我们输出到这个文件中，在这里则需要改造一下。

由于该类仅仅负责编码而不需要完成封装格式以及写文件的工作，而使用FFmpeg其实用不到libavformat模块，仅仅使用它的libavcodec模块就可以完成工作了，因此要将初始化方法改造一下，仅需要分配出编码器，并且把对编码器的要求设置进去，分配出存放PCM数据的缓冲区以及输送给编码器之前的AVFrame结构体就可以了。

实际的编码过程也需要修改一下，首先在AudioEncoder类中定义一个回调方法：

```
static int fill_pcm_frame_callback(int16_t *samples, int frame_size,
    int nb_channels, void *context)
```

该回调函数用于让客户端代码（即调用AudioEncoder的地方）提供PCM数据。编码函数的实现会对PCM数据进行编码，编码之后是FFmpeg中AVPacket的结构体，最后将该FFmpeg的数据结构转换为我们自己定义的数据结构AudioPacket，并返回给调用客户端。

最终的销毁方法比较简单，释放掉所分配的存放PCM数据的缓存区，以及编码前的AVFrame数据结构，关闭编码器上下文并且释放即可。

这就是编码器类的改造过程了，由于其实现比较简单，代码就不在这里展示了，读者可以参照代码仓库中的源码进行分析。

7.3.2 编码器适配器

本节将完成编码器的适配器，即AudioEncoderAdapter，从名字上就可以看出该类承担的职责，下面就来实现它。

首先，提供的初始化方法代码如下：

```
void init(LivePacketPool* pcmPacketPool, int audioSampleRate, int audioChanne
    int audioBitRate, const char* audio_codec_name);
```

初始化方法需要客户端（调用这个方法的类）将PCM数据存放的队列传递进来，因为这个类需要将该队列作为数据源；此外，客户端还需要将编码文件的采样率、声道数、比特率以及编码器的名称传递进来，因为这个类需要根据这些参数去寻找编码器并配置编码器。这个方法的实现需要构建出编码后的AAC存放队列，并且启动一个编码线程来进行编码工作。接下来看一下编码线程的主体流程是如何工作的：

```
audioEncoder = new AudioEncoder();
audioEncoder->init(audioBitRate,audioChannels,audioSampleRate,
        audioCodecName,fill_pcm_frame_callback,this);
while(isEncoding){
    AudioPacket *audioPacket = NULL;
    int ret = audioEncoder->encode(&audioPacket);
    if(ret >= 0 && NULL != audioPacket){
        aacPacketPool->pushAudioPacketToQueue(audioPacket);
    }
}
if (NULL != audioEncoder) {
    audioEncoder->destroy();
    delete audioEncoder;
    audioEncoder = NULL;
}
```

编码线程的最开始，将会利用初始化函数的参数来实例化一个7.3.1节讲解的编码器，可以看到初始化编码器除了需要上述参数以外，还要加上一个回调函数，该回调函数负责按照帧大小和声道数取出PCM数据队列中的PCM数据。接下来再来看一下该回调函数的实现：

```
int AudioEncoderAdapter::getAudioFrame(int16_t * samples,
```

```
        int frame_size, int nb_channels) {
    int sampleCnt = frame_size * nb_channels;
    int samplesInShortCursor = 0;
    while (true) {
        if (packetBufferSize == 0) {
            int ret = this->getAudioPacket();
            if (ret < 0) {
                return ret;
            }
        }
        int copyToSamplesInShortSize = sampleCnt - samplesInShortCursor;
        if (packetBufferCursor + copyToSamplesInShortSize <= packetBufferSize
            memcpy(samples + samplesInShortCursor, packetBuffer
                    + packetBufferCursor, copyToSamplesInShortSize *
                    sizeof(short));
            packetBufferCursor += copyToSamplesInShortSize;
            samplesInShortCursor = 0;
            break;
        } else {
            int subPacketBufferSize = packetBufferSize – packetBufferCursor;
            memcpy(samples + samplesInShortCursor, packetBuffer
                    + packetBufferCursor, subPacketBufferSize *
                    sizeof(short));
            samplesInShortCursor += subPacketBufferSize;
            packetBufferSize = 0;
            continue;
        }
    }
    return frame_size * nb_channels;
}
```

这个回调函数看起来很复杂，下面就来逐一分析一下，在这个函数的输入参数中，希望填充的帧大小是frame_size，声道数是nb_channels，填充目标是samples这一块内存区域，所以计算得出需要填充进去的采样的数目就是：

```
int sampleCnt = frame_size * nb_channels;
```

由于从PCM队列中取出的PCM的buffer大小与编码器需要我们填充的sampleCnt的大小并不一定相同，所以如果PCM队列的buffer小于sampleCnt，则需要积攒多个buffer放入这个内存区域中，如果大于sampleCnt，则应该根据要求进行拆分，并放入下一次编码器要求放入的内存区域中。所以这里需要设置一个samplesInShortCursor的变量，代表已经为该填充区域填充进去了多少个采样，对于PCM队列读取出来的buffer，可用packetBuffer来代表，使用packetBufferSize代表该buffer的大小，使用packetBufferCursor代表已经使用了该buffer的多少个数据。

如果PCM队列中读取出来的buffer已经被耗尽了，则调用getAudioPacket方法（具体的实现在下面再给出）去PCM队列中读取一个新的buffer：如果返回的值小于0，则代表已经结束，就返回小于0的值，编码器则结束；如果返回的值大于0，则代表正确读出了PCM数据，并且将采样存放到了全局变量packetBuffer中，采样数目存放到了packetBufferSize之中，首先来计算还需要为编码器填充的内存区域填充多少数据：

```
int copyToSamplesInShortSize = sampleCnt - samplesInShortCursor;
```

然后判断当前PCM队列读取出来的Buffer是否还有这么多的数据可用于填充：

```
if (packetBufferCursor + copyToSamplesInShortSize <= packetBufferSize)
```

如果packetBuffer里面还有足够多的数据，那么就拷贝数据，然后将packetBufferCursor的大小加上拷贝的大小；如果数据不够用，那么就先计算一下packetBuffer里面还有多少数据，并把剩余的数据全部都拷贝到编码器需要我们填充的内存区域中，然后将packet-BufferSize设置为0，进行下一次循环，再一次从PCM队列中读取出新的packetBuffer，重复进行该循环中的操作，直到将编码器需要我们填充的内存区域填充满了为止。

再来看一下这部分代码中的getAudioPacket方法，由于在两个平台之上音频采集的实现不同，因此需要进行特殊处理。在Android平台上，伴奏和人声都要单独入队；而在iOS平台上，人声和伴奏是合并（Mix）之后再入队，所以这个getAudioPacket的实现就有所不同了。此类中的实现就是iOS平台的实现，因为iOS平台的实现是把伴奏和人声合并（Mix）之后再放入到人声队列之中的，所以在iOS平台上就从人声队列中取出数据直接填充packetBuffer了。对于Android平台，我们需要实现一个子类，继承自当前类并且重写getAudioPacket方法，在该方法的实现中首先调用父类方法获取到人声数据，然后再读取伴奏队列中的PCM数据，将读取出来的数据与父类填充的packetBuffer进行合并（Mix），之后再存入到packetBuffer中，这样就可以无缝地接入到整个系统中了。

继续回到上述编码线程的主体流程中，有一个判断isEncoding的while循环，这个布尔型变量就是为了控制编码循环的，在循环中不断地调用编码器的编码方法。当然，编码器进行编码时，第一步就是调用上面大篇幅讲解的回调函数以填充PCM数据，然后将其编码成为一个我们自己定义的结构体AudioPacket并返回。客户端代码获取到AudioPacket之后，若判断返回值是正确编码，则将它放入到AAC的队列之中；如果返回值是编码失败的话则跳出循环，并最终销毁编码器。该适配器还要提供一个销毁的方法，进入销毁方法之后首先将isEncoding这个变量设置为false，进而丢弃PCM的队列，然后等待编码线程结束，最后销毁我们自己分配的packetBuffer等资源。这样一来，编码器适配器类就完成了，大家可以参照代码仓库中的实际代码进行分析。

7.4 画面采集与编码模块的实现

本节会基于第6章的摄像头画面采集的工程继续进行开发，不同之处在于编码之后的H264数据不会直接写入文件，而是会放到视频队列之中，所以本节将会重点介绍视频队列的实现以及如何将编码之后的H264数据放入队列之中，如果读者对于视频画面的采集与编码还不熟悉的话，那么请回到第6章中再学习一下。

7.4.1 视频队列的实现

视频队列的实现与音频队列是非常类似的，具体实现在此就不再赘述了，但是要看一下具体的接口，以便后续使用。

初始化接口，用于队列的初始化工作，要想使用队列，第一步就应调用如下这个方法：

```
void init();
```

入队接口，如果要向队列中存入一个视频帧，则调用该接口方法，如果存入成功则返回0，否则返回负数：

```
int put(VideoPacket* videoPacket);
```

从队列中拿出一个视频帧，此方法为一个阻塞方法，即如果队列为空就会阻塞住，直到队列被丢弃或者有了新的视频帧。如果正确获取到视频帧则返回0，如果返回的值小于0，则代表队列被丢弃掉了：

```
int get(VideoPacket** packet);
```

丢弃队列，即不再接受任何入队和出队的请求，一般在队列使用结束之前调用这个方法：

```
void abort();
```

接下来的这个方法为私有方法，当队列被销毁的时候调用，该方法会把队列中的所有元素全都取出来并且销毁掉：

```
void flush();
```

另外，其所存放的元素与音频队列也是不一样的，所以在这里有必要声明一下各个视频帧结构体的具体构成：

```
typedef struct VideoPacket {
    byte * buffer;
    int size;
    int timeMills;
    int duration;
    VideoPacket() {
        buffer = NULL;
        size = 0;
        timeMills = -1;
        duration = 0;
    }
    ~VideoPacket() {
        if (NULL != buffer) {
            delete[] buffer;
            buffer = NULL;
        }
    }
} VideoPacket;
```

可以看到该结构体中记录了这一帧视频帧的数据和数据长度，以及这一帧视频帧的时间戳与时长，在析构函数中会释放掉视频帧所占的内存。

7.4.2 Android平台画面编码后入队

在Android平台上的实现将基于第6章Android平台的硬件编码项目进行改动，以适配当前场景下的需求。

先找到HWEncoderAdapter类，然后找到方法drainEncodedData，目前这个方法的实现是从MediaCodec中取出编码之后的H264数据，然后写入文件。我们要做的改造就是将写文件的代码部分全部都删除掉，然后将H264的数据放入队列之中。在改造之前，我们再来回顾一下比较重要的一点，就是MediaCodec编码器会在开始编码之后给出SPS和PPS信息，并且PPS是放在SPS之后的，不过，两者会放在同一帧之中。具体应该如何判断当前帧是否是SPS和PPS所代表的数据帧呢？方法如下。

由于MediaCodec编码器为开发者提供的H264数据是有一个开始码的，即一定是从00000001开始，之后才是帧类型（NALU Type），所以可以通过以下代码来取出帧类型（NALU Type）：

```
int nalu_type = (outputData[4] & 0x1F);
```

拿出数组中下标为4的字节，使其和0X1F进行“相与”的操作，得到nalu_type，将nalu_type和H264协议中预设的Type进行比较，从而判断其到底是何种类型的NALU类型。我们之前的操作是将SPS和PPS保存下来，然后在每一个关键帧前面拼接上SPS和PPS信息。但是在Mux模块（7.4.3节中将会讲到）中，将H264数据封装（Mux）到一个MP4文件中的时候，仅需要一次SPS和PPS的设置，所以这里也仅需要将SPS和PPS信息放入队列中一次。在判断了NALU Type是SPS之后，就将其填入队列之中，并且确保即便再有SPS和PPS信息，也不会再一次填入队列之中，代码如下：

```
if(isSPSUnWriteFlag){
    VideoPacket* videoPacket = new VideoPacket();
    videoPacket->buffer = new byte[size];
    memcpy(videoPacket->buffer, outputData, size);
    videoPacket->size = size;
    videoPacket->timeMills = timeMills;
    packetPool->pushRecordingVideoPacketToQueue(videoPacket);
```

```
        isSPSUnWriteFlag = false;
    }
```

如果不是SPS和PPS的话，那么就直接进行封装，使其成为VideoPacket，然后将其放入队列之中，代码如下：

```
    VideoPacket* videoPacket = new VideoPacket();
    videoPacket->buffer = new byte[size];
    memcpy(videoPacket->buffer, outputData, size);
    videoPacket->size = size;
    videoPacket->timeMills = timeMills;
    packetPool->pushRecordingVideoPacketToQueue(videoPacket);
```

这样就将MediaCodec编码出来的H264数据按照与Mux模块约定好的格式填入到队列之中了。本节仅改动了硬件编码部分，软件编码部分不再进行讲解，但是代码示例中会有针对于软件编码分支的处理。

7.4.3 iOS平台画面编码后入队

在iOS平台中的实现也会基于第6章中iOS平台的硬件编码项目进行改动，将编码后的H264的Packet放入到视频队列中替换掉之前写文件的操作，以供后续的Muxer模块使用。基于之前的类H264HwEncoderHanlder进行修改时，首先要把初始化方法中写文件的代码全部去除掉，然后修改以下两个方法，分别是获取SPS和PPS的方法和获取一帧视频帧的方法。通过第6章我们已经知道，每一帧之前都必须要拼接上开始码（StartCode），实际上就是0000 0001，当解码器每一次碰到0000 0001这个开始码的时候，就知道这是一个数据帧的开始了，并且在解码器读取到下一个开始码之时，就知道这一帧已经结束了，所以我们在SPS、PPS以及普通的视频帧前面都要拼接上这样一个开始码来表示。但是在不同的封装格式里，对于视频帧的封装是不确定的，所以这里仅负责在视频帧前面拼接上开始码，然后发送到视频队列中，后续的封装处理就是Mux模块的事情了。由于SPS和PPS在整个编码过程中都是一样的，所以仅需要入队一次。我们可以在全局变量列表中新增一个布尔类型变量来标志是否做了SPS和PPS入队，然后将SPS和PPS前面都加上开始码，并将这个数组拼接到一起，封装成一个VideoPacket，最后放入队列之中。代码如下：

```
const char bytesHeader[] = "\x00\x00\x00\x01";
size_t headerLength = 4;
VideoPacket* spsPpsPacket = new VideoPacket();
size_t length = 2 * headerLength + sps.length + pps.length;
spsPpsPacket->buffer = new unsigned char[length];
spsPpsPacket->size = int(length);
memcpy(spsPpsPacket->buffer, bytesHeader, headerLength);
memcpy(spsPpsPacket->buffer + headerLength, (unsigned
        char*)[sps bytes], sps.length);
memcpy(spsPpsPacket->buffer + headerLength + sps.length,
        bytesHeader, headerLength);
memcpy(spsPpsPacket->buffer + headerLength*2 + sps.length,
         (unsigned char*)[pps bytes], pps.length);
spsPpsPacket->timeMills = 0;
LivePacketPool::GetInstance()->pushRecordingVideoPacketToQueue(spsPpsPacket);
```

在上述代码中可以看到，首先会在SPS前面拼接开始码，然后再拼接SPS的内容，接下来会再拼接一个开始码，最后再拼接PPS的内容，至此，这一帧特殊类型的帧就拼接完成了。接下来封装成为一个

VideoPacket，最后将构造好的这个packet推送到视频队列之中，一般情况下这会是视频队列中的首帧。紧接着来看一下在接受到一帧普通视频帧之后，应该如何做，代码如下：

```
const char bytesHeader[] = "\x00\x00\x00\x01";
size_t headerLength = 4;
VideoPacket* videoPacket = new VideoPacket();
int length = headerLength + data.length;
videoPacket->buffer = new unsigned char[length];
videoPacket->size = length;
memcpy(videoPacket->buffer, bytesHeader, headerLength);
memcpy(videoPacket->buffer + headerLength, (unsigned char*)[data bytes],
    data.length);
videoPacket->timeMills = miliseconds;
LivePacketPool::GetInstance()->pushRecordingVideoPacketToQueue(videoPacket);
```

可以看到，这里在视频帧原始数据的前面拼接上开始码作为视频帧的数据，然后再将数据长度以及时间戳共同封装到结构体对象之中，最终放入视频队列之中，而放入队列之后具体如何进行封装以及输出，将在接下来的7.5节中完成。

7.5 Mux模块

待音频帧和视频帧都编码完毕之后，接下来就是将它们封装到一个容器（如MP4、FLV、RMVB、AVI等）中，这样才可以形成一个完整的视频，在架构中这件事情是通过一个Mux模块来完成的。对于这个模块的输入，在前面各个模块的实现中都已经陆续展示出来了，这里做一个总结，7.3节把7.2节中的PCM数据编码成为了AAC，并且存放到了音频编码的队列之中了；7.4节已经把摄像头采集出的视频帧编码成了H264，并存放到了视频编码队列之中，而这两个队列就是Mux模块的输入，那么这个模块的输出又是什么？在本章的场景下就是磁盘上的一个MP4文件，当然也可以是网络流媒体服务器，这种情况下就属于直播场景了。

架构设计的合理性

弄清楚了输入和输出之后，就可以进一步扩展一下思路了，由于我们不想影响采集以及实时耳返和预览的过程，因此要在编码时单独抽取出一个线程来。那为什么我们又要为封装和文件流输出（对应于FFmpeg的Muxer层和Protocol层）单独抽取出一个线程来呢？仔细观察可以发现，编码其实是一个CPU密集型操作（即使是硬件编码，也是要占用CPU的时间片和编码的硬件设备进行内存数据交换的），而我们的封装和文件流输出却不会特别耗费CPU，尤其是当文件流输出到网络的时候，因此不应该由输出来影响整个编码过程。拆分开之后每个模块各司其职，统一接口，对整个系统的维护以及扩展都有了极大的好处，比如，由软件编码升级为硬件编码，由于接口不变，所以直接更改编码模块的实现就好了，或者我们的封装格式由MP4转换为FLV的话，也只需要改动封装模块。

7.5.1 初始化

接下来看一下如何用FFmpeg实现格式封装与文件流输出，经过之前章节对FFmpeg的了解，大家应该已经十分清楚，封装和输出其实就是FFmpeg里面的libavformat这个模块所承担的职责，首先来看一下这个模块的初始化方法：

```
int init(char* videoOutputURI, int videoWidth,
        int videoHeight,float videoFrameRate,int videoBitRate,
        int audioSampleRate, int audioChannels, int audioBitRate,
        char* audio_codec_name)
```

初始化方法的参数比较多，如果分为三部分来解释就会很清晰了。第一部分只有一个参数，就是输出的文件路径；第二部分就是视频流的参数，包括视频的宽、高、帧率、比特率以及视频编码格式（默认为H264格式）；第三部分就是音频流的参数，包括音频的采样率、声道数、比特率，以及音频编码器的名称。这个初始化方法的实现具体如下，构造一个Container（对应于FFmpeg中的结构体类型为AVFormatContext），根据上述的视频参数配置好一路视频流（AVStream）添加到这个Container中，然后根据上述的音频参数再配置一路音频流（AVStream）添加到Container中。下面再来看一下具体实现，第一步先注册FFmpeg里面的所有封装格式、编解码器以及网络配置开关（如果需要将视频流推送到网络上的话）：

```
avcodec_register_all();
av_register_all();
avformat_network_init();
```

然后根据输出目录来构造一个Container，即构造一个AVFormatContext类型的结构体，其实该结构体就是FFmpeg中使用libavformat模块的入口：

```
AVFormatContext* oc;
avformat_alloc_output_context2(&oc, NULL, "flv", videoOutputURI);
AVOutputFormat* fmt = oc->oformat;
```

接下来构造一路视频流，并加入到这个Container中。首先按照常量AV_CODEC_ID_H264找出H264的编码器，然后在Container中增加一路H264编码的视频流，并找出这路流的编码器上下文，对该上下文的属性依次赋值，即可构造好这路视频流。同时，这路流也已经被正确地添加到了Container中。代码如下：

```
AVCodec *video_codec = avcodec_find_encoder(AV_CODEC_ID_H264);
AVStream *st = avformat_new_stream(oc, video_codec);
st->id = oc->nb_streams - 1;
AVCodecContext *c = st->codec;
c->codec_id = AV_CODEC_ID_H264;
c->bit_rate = videoBitRate;
c->width = videoWidth;
c->height = videoHeight;
c->time_base.den = 30000;
c->time_base.num = (int) (30000 / videoFrameRate);
c->gop_size = videoFrameRate;
if (oc->oformat->flags & AVFMT_GLOBALHEADER)
    c->flags |= CODEC_FLAG_GLOBAL_HEADER;
```

接下来构造一路音频流，整个过程和视频流的构建非常类似，不同的是，编码器是通过传递进来的编码器名称来寻找的，代码如下：

```
AVCodec *audio_codec = avcodec_find_encoder_by_name(codec_name);
AVStream *st = avformat_new_stream(oc, audio_codec);
st->id = oc->nb_streams - 1;
AVCodecContext *c = st->codec;
c->sample_fmt = AV_SAMPLE_FMT_S16;
c->bit_rate = audioBitRate;
c->codec_type = AVMEDIA_TYPE_AUDIO;
c->sample_rate = audioSampleRate;
c->channel_layout = audioChannels == 1 ? AV_CH_LAYOUT_MONO :
AV_CH_LAYOUT_STEREO;
c->channels = av_get_channel_layout_nb_channels(c->channel_layout);
c->flags |= CODEC_FLAG_GLOBAL_HEADER;
```

和视频流不同的是，完成音频流的添加之后，这里需要为编码器上下文设置一下extradata变量，FFmpeg对于在编码端设置这个变量的目的是，为解码器提供原始数据，从而初始化解码器。还记得之前编码AAC的时候要在编码出来的数据前面加上ADTS的头吗？其实在ADTS头部信息里面可以提取出编码器的Profile、采样率以及声道数的信息，但是在FFmpeg中并不是每一个AAC的头部都能添加上这些信息，而是在全局的extradata中配置这些信息。那么，来看一下如何为FFmpeg的音频编

码器上下文来设置这个extradata：

```
int profile = 2;  // AAC LC
int freqIdx = 4;  // 44.1Khz
int chanCfg = 2;  // Stereo Channel
char dsi[2];
dsi[0] = (profile<<3) | freqIdx>>1);
dsi[1] = ((freqIdx&1)<<7) | (chanCfg<<3);
memcpy(c->extradata, dsi, 2);
```

读者可能会有疑问，视频流的这个extradata变量该如何设置呢？关于视频流编码器中的变量设置将会在7.5.2节中讲解，因为视频流中的这个变量存放的是SPS和PPS的信息，是由编码器在编码过程的第一步输出的，所以需要放在7.5.2节来讲解。同时我们也要配置一个音频格式转换的滤波器，就是ADTS到ASC格式的转换器，代码如下：

```
bsfc = av_bitstream_filter_init("aac_adtstoasc");
```

上述步骤完成之后说明我们的Container即封装格式已经初始化好了，然后就是打开文件的连接通道，可调用FFmpeg的Protocol层来完成操作，代码如下：

```
AVIOInterruptCB int_cb = { interrupt_cb, this };
oc->interrupt_callback = int_cb;
avio_open2(&oc->pb, videoOutputURI, AVIO_FLAG_WRITE,
        &oc->interrupt_callback, NULL);
```

如果上述代码中的avio_open2函数的返回值大于等于0，则将isConnected变量设置为true，代表其已经成功地打开了文件输出通道。唯一需要注意的一点是，需要配置一个超时回调函数进去，这个回调函数主要是给FFmpeg的协议层用的，在实现这个函数的时候，返回1则代表结束I/O操作，返回0则代表继续I/O操作，超时回调函数如下：

```
static int interrupt_cb(void* ctx) {
    if(getCurrentTimeMills() – latestFrameTime > 15 * 1000)
        return 1;
    return 0;
}
```

上述代码表示如果当前时间超过了封装上一帧的时间（15s），则终止协议层的I/O操作，当然，每次封装完一帧之后就要更新latestFrameTime这个变量。这个回调函数的配置是非常重要的，特别是在后续我们要和网络打交道的时候。初始化方法到这里就配置结束了，接下来看一下实际的封装和输出。

7.5.2 封装和输出

待构建好了这个Container之后，再不断地将音频帧和视频帧交错地封装进来，然后通过输出通道写出到文件或者网络之中。先来看一下主体的流程：

```
int ret = 0;
double video_time = getVideoStreamTimeInSecs();
double audio_time = getAudioStreamTimeInSecs();
if (audio_time < video_time){
    ret = write_audio_frame(oc, audio_st);
} else {
    ret = write_video_frame(oc, video_st);
}
latestFrameTime = getCurrentTimeMills();
duration = MIN(audio_time, video_time);
return ret;
```

因为音视频是交错存储的，即存储完一帧视频帧之后，再存储一段时间的音频（不一定是一帧音频，这要看视频的FPS是多少，因为代码中是按照时间进行比较的），之后再存储一帧视频帧，所以在某一个时间点是要封装音频还是封装视频，是由当前两路流上已经封装的时间戳来决定的。所以代码中首先要获取两路流上当前的时间戳信息，然后进行比较，将时间戳比较小的那一路流进行封装和输出，并且更新latestFrameTime变量用来辅助前面配置的超时回调函数，最后一步，是将两路流中较小的时间戳作为时长存储下来。接下来分别看一下封装和输出音频以及视频流的部分。

封装音频流时，首先从AAC的音频队列中取出一帧音频帧：

```
int ret = AUDIO_QUEUE_ABORT_ERR_CODE;
AudioPacket* audioPacket = NULL;
fillAACPacketCallback(&audioPacket, fillAACPacketContext);
audioStreamTimeInsecs= audioPacket->position;
```

然后取出该AAC音频帧的时间戳信息，存储到全局变量的audioStreamTimeInsecs中，作为要写入音频这路流的时间戳信息，以便于在编码之前取出音频流中编码到的时间信息。然后将该AAC的Packet

转换成一个AVPacket类型的结构体：

```
AVPacket pkt = { 0 };
av_init_packet(&pkt);
pkt.data = audioPacket->data;
pkt.size = audioPacket->size;
pkt.dts = pkt.pts = lastAudioPacketPresentationTimeMills / 1000.0f /
        av_q2d(st->time_base);
pkt.duration = 1024;
pkt.stream_index = st->index;
```

接着，将上述代码中的pkt作为我们调用bitStreamFilter转换的输入Packet，在调用完毕转换之后，将ADTS封装格式的AAC变成了一个MPEG4封装格式的AAC，之后就可以通过输出通道进行输出了，代码具体如下：

```
AVPacket newpacket;
av_init_packet(&newpacket);
ret = av_bitstream_filter_filter(bsfc, st->codec, NULL,
        &newpacket.data, &newpacket.size, pkt.data, pkt.size,
        pkt.flags & AV_PKT_FLAG_KEY);
if (ret >= 0) {
    newpacket.pts = pkt.pts;
    newpacket.dts = pkt.dts;
    newpacket.duration = pkt.duration;
    newpacket.stream_index = pkt.stream_index;
    ret = av_interleaved_write_frame(oc, &newpacket);
}
av_free_packet(&newpacket);
```

至此，就将从队列中取得的一帧ADTS封装格式的AAC写入到Container的音频流中了。下面来看一下如何将其封装和输出到视频流中，在这个方法的实现中，首先填充视频编码器上下文中的extradata，这一点非常重要。否则等这个视频进行播放的时候，会因为解码器无法正确初始化从而不能够正确地解码视频。首先取出H264队列中的视频帧，并且取出时间戳更新视频流封装到的时间，以便于Mux模块的主体流程可以判断下一帧应该编码哪一路流，代码如下：

```
VideoPacket *h264Packet = NULL;
fillH264PacketCallback(&h264Packet, fillH264PacketContext);
videoStreamTimeInSecs= h264Packet->timeMills / 1000.0;
```

接下来看一下如何正确填充extradata，从H264的队列中取出一帧H264的视频帧数据，因为在7.4节中已将SPS和PPS的信息拼接起来了，并且封装成为一帧H264数据并放入到了视频帧队列中了，所以在取得H264帧之后首先判定是否是SPS信息，判断规则是取出这一帧H264数据的index为4的下标，按位“与”上0x1F得到NALU Type，然后与H264中预定义的类型进行比较，代码如下：

```
uint8_t* outputData = (uint8_t *) ((h264Packet)->buffer);
int nalu_type = (outputData[4] & 0x1F);
```

至于NALU Type的类型定义，笔者找了几个比较重要的类型展示如下：

```
#define H264_NALU_TYPE_NON_IDR_PICTURE                          1
#define H264_NALU_TYPE_IDR_PICTURE                      5
#define H264_NALU_TYPE_SEQUENCE_PARAMETER_SET           7
#define H264_NALU_TYPE_PICTURE_PARAMETER_SET            8
#define H264_NALU_TYPE_SEI                              6
```

所以，判定帧类型是不是SPS类型即判定nalu_type是不是等于7，如果相等的话，则将这一帧H264数据拆分成SPS和PPS信息，由于在7.4节中已经将SPS和PPS拼接成了一帧放入到了视频队列之中。拆分过程的代码在这里就不再展示了，其实就是找出H264的StartCode，即以00 00 00 01开始的部分，第一个就是SPS，第二个就是PPS。把SPS和PPS分别放入到两个uint8_t的数组之中，一个是spsFrame，另外一个是ppsFrame，并且这个数组的长度也存放到了对应的变量之中。最后将spsFrame和ppsFrame封装到视频编码器上下文的extradata中，代码如下：

```
AVCodecContext *c = videoStream->codec;
int extradata_len = 8 + spsFrameLen - 4 + 1 + 2 + ppsFrameLen - 4;
c->extradata = (uint8_t*) av_mallocz(extradata_len);
c->extradata_size = extradata_len;
c->extradata[0] = 0x01;
c->extradata[1] = spsFrame[4 + 1];
c->extradata[2] = spsFrame[4 + 2];
c->extradata[3] = spsFrame[4 + 3];
c->extradata[4] = 0xFC | 3;
c->extradata[5] = 0xE0 | 1;
int tmp = spsFrameLen - 4;
c->extradata[6] = (tmp >> 8) & 0x00ff;
```

```
c->extradata[7] = tmp & 0x00ff;
int i = 0;
for (i = 0; i < tmp; i++)
    c->extradata[8 + i] = spsFrame[4 + i];
c->extradata[8 + tmp] = 0x01;
int tmp2 = ppsFrameLen - 4;
c->extradata[8 + tmp + 1] = (tmp2 >> 8) & 0x00ff;
c->extradata[8 + tmp + 2] = tmp2 & 0x00ff;
for (i = 0; i < tmp2; i++)
    c->extradata[8 + tmp + 3 + i] = ppsFrame[4 + i];
```

上述拼接规则是笔者在FFmpeg的源码中提取出来的（源码在libavformat目录中，avc.c这个文件里面的方法ff_isom_write_avcc中），分为以下几个部分：第一部分是元数据部分，即下标从0到5，代表了version、profile、profile compat、level以及两个保留位；第二部分是SPS，包括SPS的大小以及SPS的内部信息；第三部分是PPS，首先是PPS的数目，然后是PPS的大小和PPS的内部信息。这个拼接规则比较重要，请读者深刻理解一下，在后续使用硬件解码器加速视频播放器的项目中，还会用到这个拼接规则，只不过是通过extradata解析出SPS和PPS的信息。封装好视频流编码器的extradata之后，才表示这个Container完全封装好了，在这里必须调用write_header方法，将这些MetaData写出到文件或者网络流中，代码如下：

```
int ret = avformat_write_header(oc, NULL);
if (ret >= 0) {
    isWriteHeaderSuccess = true;
}
```

当write header成功之时，将变量isWriteHeaderSuccess设置为true，以方便后续在实现销毁操作时可以用来判断是否需要执行write trailer的操作。

接下来就是真正地封装并且输出视频帧了，最重要的是视频帧的封装，即把从H264队列中取出来的H264数据封装成FFmpeg认识的AVPacket，封装中最重要的一步类似于音频里的格式转换，即在音频的封装过程中使用ADTS到ASC的转换过滤器，而这里是没有这样的转换过滤器可以使用的，所以需要做手动转换，这个转换过程也很简单，把H264视频帧起始的StartCode部分替换为这一帧视频帧的大小即可，当然大小不包括这个StartCode部分，代码如下：

```
pkt.data = outputData;
if(pkt.data[0] == 0x00 && pkt.data[1] == 0x00 &&
        pkt.data[2] == 0x00 && pkt.data[3] == 0x01){
    bufferSize -= 4;
    pkt.data[0] = ((bufferSize) >> 24) & 0x00ff;
    pkt.data[1] = ((bufferSize) >> 16) & 0x00ff;
    pkt.data[2] = ((bufferSize) >> 8) & 0x00ff;
    pkt.data[3] = ((bufferSize)) & 0x00ff;
}
```

如上述代码所示，代表帧大小的这个bufferSize的字节顺序是很重要的，必须按照代码中的大尾端（big endian）字节序进行拼接才可以。接下来就是将该AVPacket的size设置为bufferSize，将pts和dts设置为从H264队列中取出来的pts和dts，此外还需要将视频编码器上下文的frame_number加1，代表又增加了一帧视频帧，最后还有一个对于AVPacket来说非常重要的属性——flags，即标识这个视频帧是否是关键帧，那么应该如何来确定取出来的这一帧H264视频帧是否是关键帧呢？还是得回到上面判断NALU Type的地方，如果NALU Type不是SPS，则判断其是否是关键帧，即nalu_type是否等于5；如果是关键帧，则将flags设置为1，可以使用FFmpeg中定义的宏AV_PKT_FLAG_KEY；如果不是关键帧则设置为0，因为解码器要按照是否是关键帧来构造解码过程中的参考队列。至此我们的封装工作就结束了。接下来就是输出部分，其实输出和音频流的输出是一样的，代码如下：

```
av_interleaved_write_frame(oc, &pkt);
```

封装和输出视频帧结束之后，就可以再回到Mux模块的主体流程了，主体流程会不断地进行循环，直到音频或者视频的封装和输出函数返回小于0的值然后结束，但是何时返回小于0的值呢？其实就是在获取AAC队列以及获取H264队列的时候，如果这个队列被abort掉了，那么就返回小于0的值，那么这两个队列又是何时被abort掉的呢？其实就是在停止整个Mux流程的时候。停止Mux模块如下，首先会abort掉这两个队列，然后等待主体Mux流程的线程停止，之后调用销毁资源的方法，销毁资源的方法将在7.5.3节进行介绍。

7.5.3 销毁资源

对于销毁资源，首先要做的是判断是否打开了输出通道，并且确定它是否做了write-Header的操作，如果做了的话，就要执行write_trailer的操作，并且设置好duration：

```
if (isConnected && isWriteHeaderSuccess) {
    av_write_trailer(oc);
    oc->duration = duration * AV_TIME_BASE;
}
```

这里有一点比较重要，如果我们没有write header而又在销毁的时候调用了write trailer，那么FFmpeg程序会直接崩溃，所以这里使用了一个布尔变量来保证write header和write trailer的成对出现。由于Mux模块不做编码工作，所以没有打开过任何编码器，也就无需关闭编码器，但是对于音频来讲，还使用了一个bitStreamFilter，所以需要关闭：

```
av_bitstream_filter_close(bsfc);
```

最后关闭输出通道，并释放整个AVFormatContext：

```
if (isConnected) {
    avio_close(oc->pb);
    isConnected = false;
}
avformat_free_context(oc);
```

操作完如上流程，就完成了销毁资源的操作。

7.6 中控系统串联起各个模块

最终我们来写一个控制器，将所有的模块都串联起来，从而完成整个项目。一旦进入录制视频页面，就已经有了视频的预览界面，即已经启动了7.4节的视频采集模块，只不过我们不会启动编码线程进行编码，然后，在控制器中初始化H264视频队列和PCM的音频队列，并调用7.5节的Mux模块的初始化方法。如果初始化成功了的话（因为有可能涉及输出通道建立不成功的情况，所以先初始化Mux模块，再初始化编码模块），则应该在启动音频的采集以及编码模块之后，再启动视频的采集以及编码模块；但是如果初始化失败的话，则销毁H264视频队列和PCM音频队列。代码如下：

```
PacketPool* packetPool = PacketPool::GetInstance();
packetPool->initRecordingVideoPacketQueue();
packetPool->initAudioPacketQueue(audioSampleRate);
packetPool->initAudioPacketQueue();
videoPacketConsumerThread = new VideoPacketConsumerThread();
int initCode = videoPacketConsumerThread->init(videoPath,
        videoWidth, videoheight, videoFrameRate, videoBitRate,
        audioSampleRate,audioChannels, audioBitRate,
        "libfdk_aac");
if(initCode >= 0){
    videoPacketConsumerThread->startAsync();
    // Start Producer
} else{
    packetPool->destoryRecordingVideoPacketQueue();
    packetPool->destoryAudioPacketQueue();
    packetPool->destoryAudioPacketQueue();
}
```

如果初始化成功，则启动音频的采集以及编码模块。首先启动7.2节的音频采集模块，然后启动7.3节的音频编码线程，接着启动7.4节中的视频编码模块，这样整个系统就运行起来了。来看一下具体的实现代码：

```
audioRecorder->start();
audioEncoder->start();
videoScheduler->startEncoding();
```

此时我们可以看到，音频采集线程将声音不断地采集到了PCM队列

之中，音频编码线程不断地从PCM队列中取出数据并进行编码，将编码之后的AAC数据送入到AAC队列之中；同时可以看到，视频采集线程不断地将预览画面采集下来，然后发送给视频编码线程进行编码，最终编码为H264数据并传入到H264的队列之中；而最开始启动的Mux模块，会不断地从这两个队列（AAC队列与H264队列）中取出AAC的音频帧和H264的视频帧，然后封装到MP4的Container中，最终输出到本地磁盘的文件中。

当停止录制的时候，首先会停止生产者部分，即停止视频的编码，然后停止音频的编码，接下来停止音频的采集，最后停止Mux模块，这样整个录制过程就结束了，代码如下：

```
videoScheduler->stopEncoding();
audioEncoder->stop();
audioRecorder->stop();
videoPacketConsumerThread->stop();
```

这个中控系统就是如此简单，读者可以体会一下一个复杂的系统经过优秀的设计，也会变得简单起来，所以笔者建议读者在日常的开发中可以采取设计先行的开发模式，等设计出来之后，找对应的开发人员做一个设计评审会，这样会较早暴露出问题，最终可以提高整个开发过程的效率。

7.7　本章小结

最终我们终于完成了整个项目。启动该程序之后，点击录制按钮，进入预览界面，我们可以寻找一个合适的画面，选择开始录制，大家可以先做个自我介绍，然后选择一个伴奏，唱一首歌曲，然后点击停止录制，最终导出我们生成的MP4文件，播放这个MP4文件，就可以看到我们刚才的表演了。在播放这个MP4文件的时候，有的读者可能会发现以下几个问题：

为什么我的声音听起来特别干，能不能给修饰一下呢？

为什么我脸上的痘痘好明显啊，能不能做个美颜呢？

是的，这也是后续章节的重点内容，我们会将声音进行美化，将视频进行美化，等下一个篇幅结束之后，再运行最新的项目生成新的视频，到那时再来观看这个视频的时候，你就会发现，这个视频中有一个更加漂亮（帅气）的你，声音也会更加动听。

第8章　音频效果器的介绍与实践

前7章不仅介绍了音视频的基础概念，还在Android和iOS平台上完成了两个比较完整的应用，一个是视频播放器的应用，一个是视频录制应用，因此可以把前7章称为基础篇或入门篇。从现在开始，将介绍一个新的篇章——提高篇，这部分内容旨在为基础篇中的两个应用添加一些必要的功能（比如添加音频滤镜、视频滤镜），做一些性能优化（比如硬件解码器的使用），实现一些公共基础库的抽象与构建（音频处理、视频处理的公共库）等。

我们已在第1章介绍过一些音频背景及其相关知识，本章会在此基础上进行更加深入的讲解。此外，本章还会介绍一些基本的乐理知识。让我们开始吧！

8.1　数字音频基础

第1章已经介绍了音频的模拟信号与数字信号的概念，而本章介绍的内容都属于数字音频，所以本节会以更加直观的方式来讲解数字音频。在开始本节之前建议读者先下载一个音频编辑工具，如Audacity、Audition、Cubase等，其中Audacity在我们的资源目录中提供了一个Mac版本的安装文件，如果读者使用的是Mac OS环境，则可以直接安装。由于本章执行的很多操作都基于Audacity工具的，所以先简单介绍Audacity。Audacity是一个集播放、编辑、转码为一体的工具软件，也是日常工作中必不可少的一个工具。大家可以在本章资源目录中找到对应的音频文件pass.wav并将其放到Audacity中，完成操作后，直接映入眼帘的就是下面要介绍的音频的第一种表示形式，即波形图的表示。

8.1.1　波形图

声音最直接的表示就是波形图，英文叫waveform。横轴是时间，纵轴根据意义的不同而有多种不同的格式，如有用dB表示的、有用相对值表示的等，但是可总体理解为强度的大小。下面先看看当笔者读出pass[pɑ：s]这个单词时所产生的波形图，如图8-1所示。

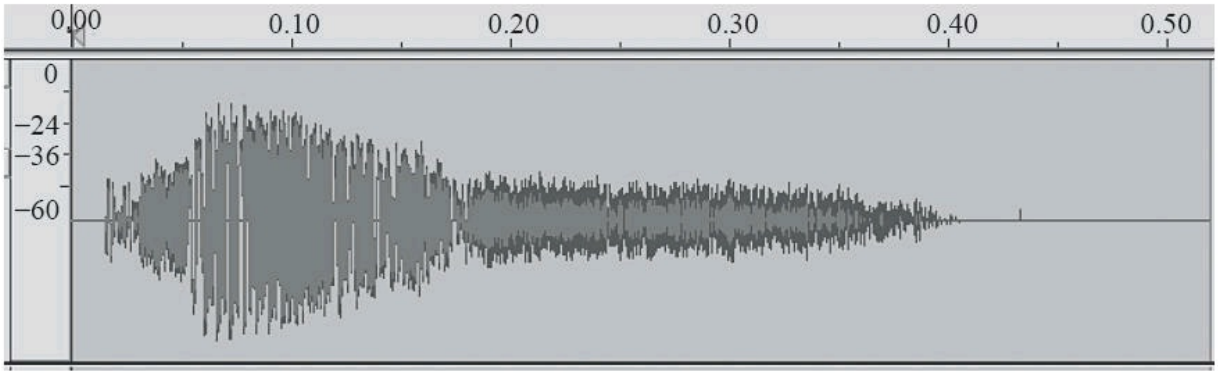


图　8-1

当横轴的分辨率不够高时，波形图看起来就像图8-1一样。如果不是一个单词，而是一段话，其波形图就会由多个这样的波形连接起来，而所有波形的轮廓可以称为整个声音在时域上的包络（envelope），包络整体形状描述了声音在整个时间范围内的响度。一般来说，每一个音节对应一个这样的三角形，因为每一个音节通常都会包含一个元音，而元音听起来比辅音更响亮（如图8-1中的0.05～0.18s）。但是也有例外，比如，类似/s/的唇齿音持续时间比较长，也会形成一个比较长的三角形（如图8-1中的0.18～0.4s）；类似/p/的爆破音会在瞬时聚集大量能量，在波形上体现为一个脉冲（如图8-1中的0.02～0.05s）。如果提高横轴时间单位的分辨率，比如只观察20ms的波形，则可以看到波形图更精细的结构，如图8-2所示。

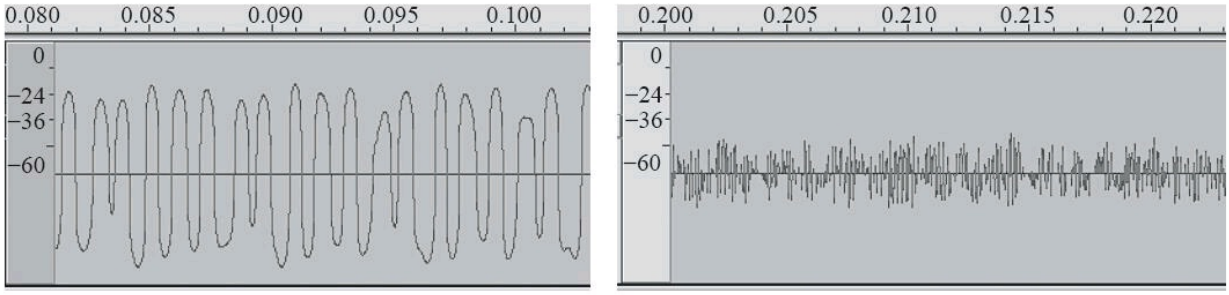


图　8-2

图8-2左边的图就是放大了0.08～0.10s部分的波形图情况，这部分是元音，我们也可以注意到这个波形是有周期性的，大约为3个周期多一点（每个周期约为7ms），这也是所有浊音在时域上的特性。而图8-2右边的图就是放大了0.2～0.22s部分的波形图情况，这部分是清音，是没有任何周期性的，并且频率（过零率，即在横轴精度一致情况下的波形的疏密程度）比元音高很多。

以上介绍的特性我们都可从波形图上直观看出来。我们知道，波形图表示的就是随着时间的推移声音强度变化的曲线，是最直观也是最容易理解的一种声音的表示形式，也就是通常所说的声音的时域表示。介绍完声音的时域表示，再从另外一个维度介绍声音是如何表现的，也就是它的频域表示——声音的频谱（spectrum）图的表示。

8.1.2　频谱图

使用图8-1中0.08～0.11s的这一段声音来做FFT，得到的频谱展示图如图8-3所示。

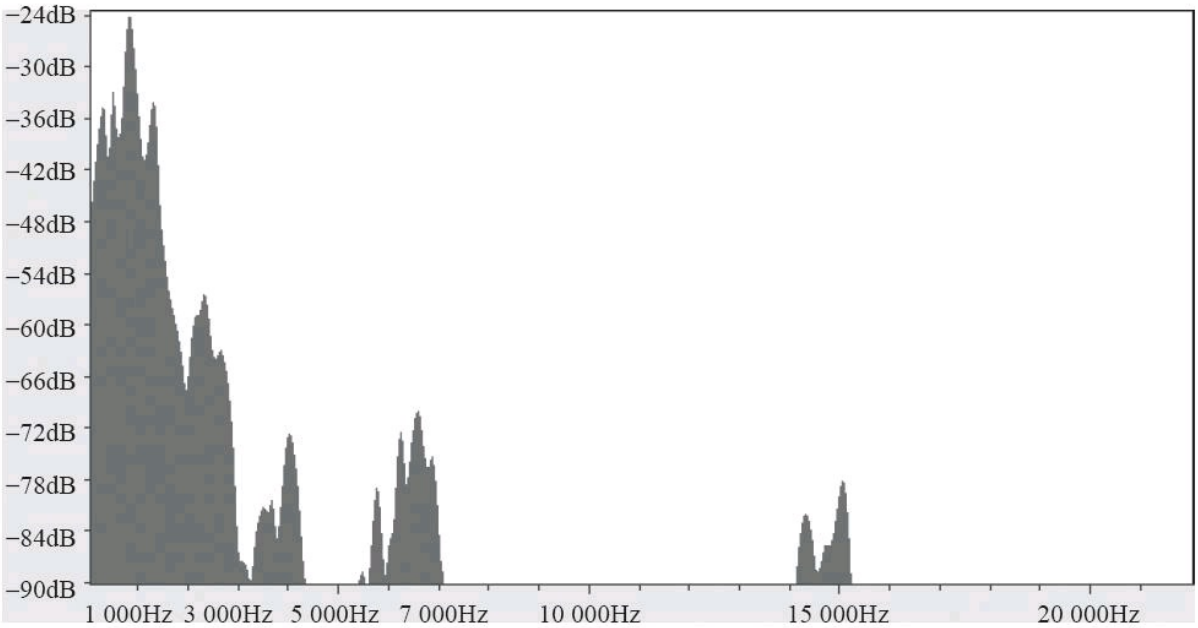


图　8-3

在解释频谱图之前先理解什么是FFT。FFT是离散傅立叶变换的快速算法，它可以将一个时域信号变换为频域表示的信号，有些信号在时域上很难看出是什么特征的，但是，如果变换到频域之后，就很容易看出了，这就是很多信号分析采用FFT变换的原因。因此，我们将一小段波形做FFT之后取模，注意这里必须是一小段波形（一般是20～50ms），如果这段波形表示的时间太长，就没有意义了。对音频信号做FFT的时候，是把虚部设置为0，所得到的FFT的结果是对称的，即音频采样频率为44100，那么0～22050的频率分布和22050～44100的频率分布是一致的。下面基于此来理解图8-3，横轴频率的表示范围为0～22050，而纵轴表示的就是当前频率能量的大小，我们能直接看到的就是频域的包络，如果把横轴表示的单位改为指数级（即把分布比较密集的地方使用更加精细的单位来表示），就可以显示出频域上能量分布的精细结构。图8-4表示频域分布的精细结构。

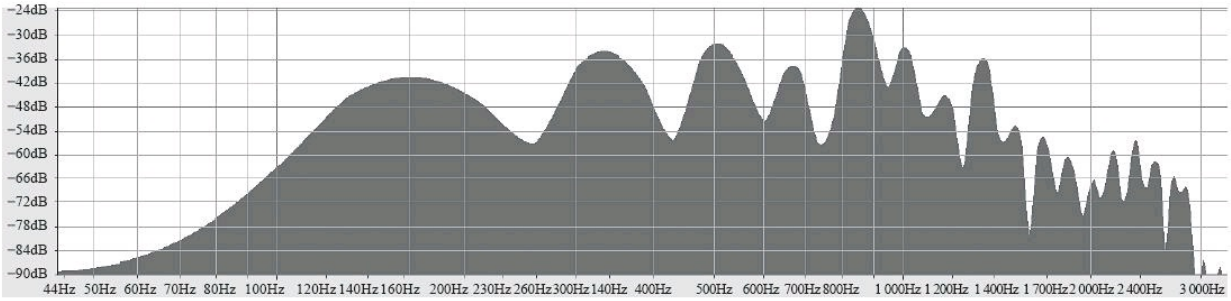


图　8-4

从图8-4中可以看出，每隔170Hz就会出现一个峰，而这恰恰是我们在图8-2左边的图中所看到的波形周期（约为6ms）所对应的频率。从图8-4中还可以看出语音不是一个单独的频率信号，而是由许多频率的信号经过简谐振动叠加而成的。图8-4中的每一个峰叫做共振峰，第一个峰叫基音，其余的峰叫泛音，第一个峰的频率（也是相邻峰的间隔）叫做基频（fundamental frequency），也叫音高（pitch），常记为f0。对于人声来讲，声带发声之后会经过我们的口腔、颅腔等进行反射，最终让别人听到，但是这里基频指的就是声带发出的最原始的声音所代表的频率。因此，如果声带不发出声音，比如唇齿音（/z//c//s/等）一般就无法检测出基频。

图8-4的频谱图有很多峰，每个峰的高度不一样，这些峰的高度之比决定其音色（timbre）。但对于语音的音色来说，一般没有必要精确描写每个峰的高度，而是用“共振峰”（formant）来描述的。共振峰指的是包络的峰，从图8-4可以看到，第一个共振峰的频率170Hz，第二个共振峰的频率为340Hz，第三个共振峰的频率为510Hz，第四个共振峰的频率为680Hz，第五个共振峰的频率为850Hz，第六个共振峰的频率为1020Hz，越往后共振峰的频率越弱，所以一般由前几个共振峰的形状决定声音的音色。接着再看0.2～0.22s波形的频谱图表示，如图8-5所示。

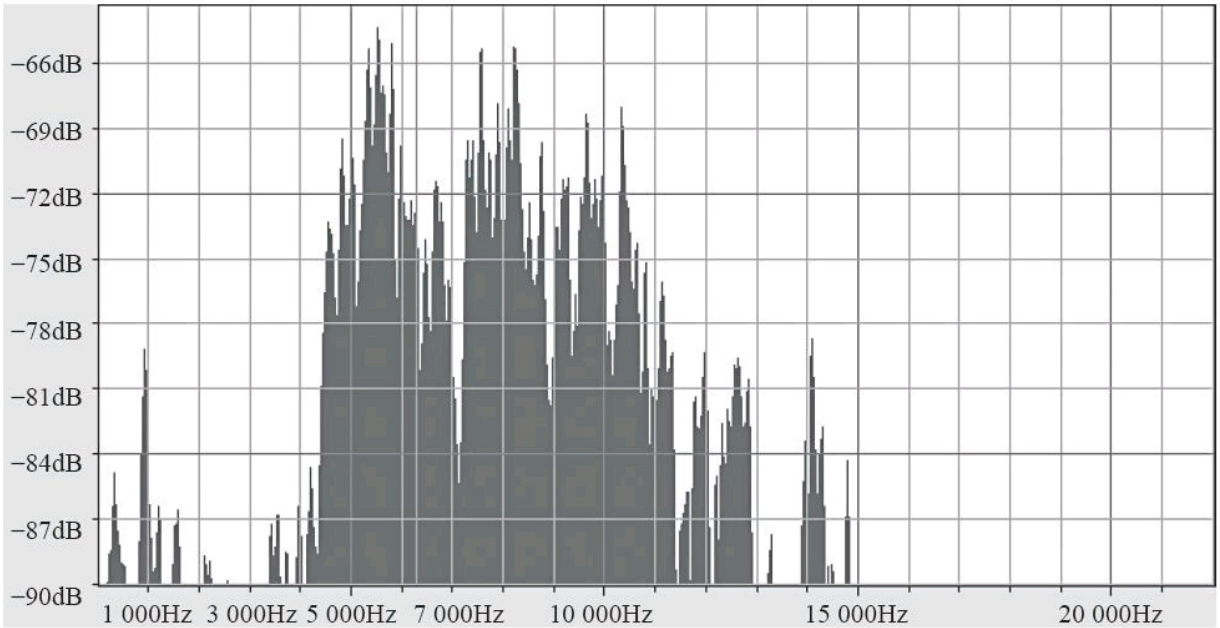


图　8-5

从图8-5可以发现，在低频率部分几乎没有峰（1000Hz处由于能量太小，可以忽略），第一个峰值出现在5000Hz以上，这种情况下也就无法计算基频，如果对应于人的发声部位，那就是我们的声带不发声，这一般称为清音。清音通常没有共振峰，也就没有基频和没有音高。

上面的频谱图只能表示一小段声音，而如果我们想观察一整段语音信号的频域特性，应该怎么做？这将涉及下一节介绍的语谱图，我们在第3章中讲解ffplay时在显示面板上绘制的就是语谱图。

8.1.3　语谱图

我们可以把一整段语音信号截成许多帧，并将它们各自的频谱“竖”起来（即用纵轴表示频率），用颜色的深浅来代替当前频率下的能量强度，再将所有帧的频谱横向并排起来（即用横轴表示时间），就得到了语谱图，可以用声音的时频域表示。语谱图可以理解为一个三维的概念，即横轴为x轴，表示时间；纵轴为y轴，表示频率；以及一个z轴，表示当前时间点，当前频率所代表的能量值（能量值越大，颜色越深）。使用Audacity软件打开pass.wav之后，在这一轨声音的左侧选择频谱图（在Audacity中语谱图称为频谱图）的视图模式，得到这段声音的语谱图，如图8-6所示。

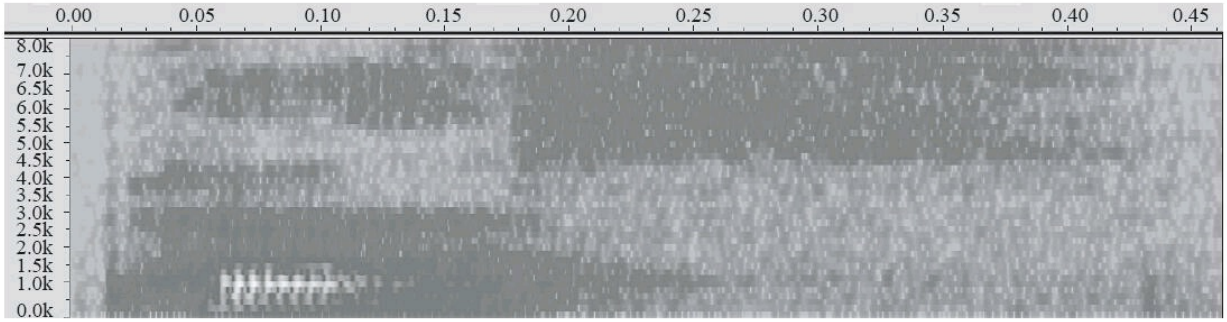


图　8-6

在图8-6中，横轴是时间，纵轴是频率，颜色越深的地方代表声音的能量越大。所以从图8-1的波形图可以看到，0.0～0.05s是在/p/这个爆破音的时候，其频率基本上都在1000Hz以下；而到0.05～0.15s，元音/ɑ：/的频率就非常明显，并且颜色已经非常深，是可以计算出基频的。随着时间的推移，到0.2s以后的/s/，其频率基本上都在5000Hz以上，这一段声音是无法计算基频的，属于清音部分。语谱图的好处是可以直观地看出共振峰频率的变化。

下面再介绍一下清音和浊音，因为这对于后续在基频检测以及对频域数据进行处理时会有很大帮助。在语音学中，将发音时声带振动的音称为浊音，将发音时声带不振动的音称为清音。辅音有清有浊，也就是大家常说的清辅音、浊辅音。而在大多数语言中，元音皆为浊音，鼻音、半元音也是浊音。我们可以尝试发出/a/这个音，同时用手触摸喉部，可以感觉喉咙是振动的；而发出b/p/、d/t/、g/k/等音的时候喉咙是

不振动的，这些音都是清辅音。还有一种是鼻音，如/m/、/n/、/l/等都是浊辅音。清音无法检测出基频，因此也就无法知道它所代表的音高；浊音一般可以检测出基频，所以可以计算出它表示的音高。

8.1.4 深入理解时域与频域

通过前面的介绍，读者应该已经比较清楚声音在时域和频域上的表示。但是，也许有些读者可能还是不太清楚声音的波形是如何产生的，又是如何跟频域联系起来的。下面先来生成一段单一频率的声音，然后逐步叠加不同频率的声音，以此作为我们的声源，从而逐步分析快速傅里叶变换（FFT）能为我们做什么。

首先编写一个函数来生成频率为440Hz，单声道，采样频率为44100Hz（注意采样频率代表的波形的平滑程度），时长为5s的声音，代码如下：

```
double sample_rate = 44100.0;
double duration = 5.0;
int nb_samples = sample_rate * duration;
short* samples = new short[nb_samples];
double tincr = 2 * M_PI * 440.0 / sample_rate;
double angle = 0;
short* tempSamples = samples;
for (int i = 0; i < nb_samples; i++) {
float amplitude = sin(angle);
    *tempSamples = (int)(amplitude * 32767);
    tempSamples += 1;
    angle += tincr;
}
// Write To PCM File
delete[] samples;
```

从以上代码中可以看到，生成的就是一个相位为零的正弦波，使用Audacity软件将生成的PCM文件以裸数据（raw data）的方式导入，放大横轴的刻度可以看到波形图，如图8-7所示。

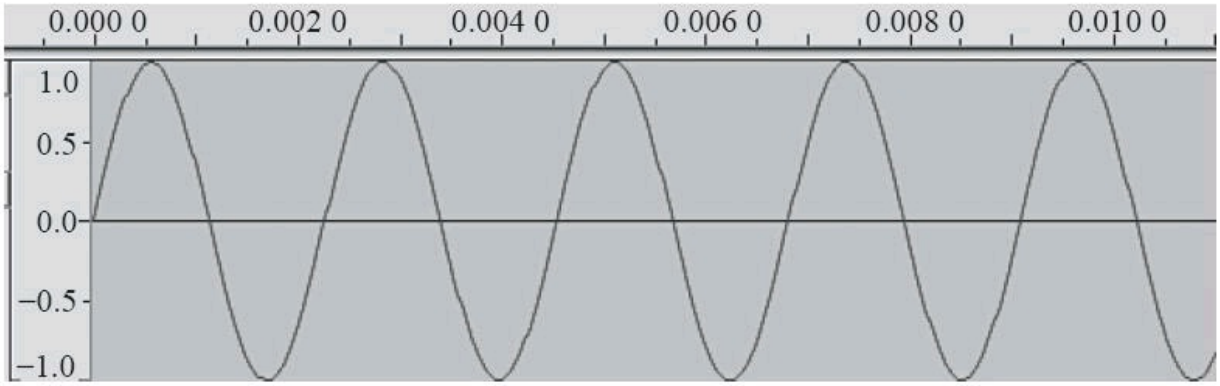


图　8-7

从图8-7中可以看出，时间为0的地方正处于正弦波幅度为0的地方，所以相位为0。从图中还可以看出，一个周期大约为2.27ms，这也正好代表了生成的这段声音的频率为440Hz。也许读者可能会问，那采样频率在波形图中又代表什么？采样频率在波形图中代表整个波形的平滑程度，采样点越多，波形就越平滑。紧接着选中20ms的音频来做傅里叶变换得到的频谱图，如图8-8所示。

从图8-8这个频谱图中可以看到，波峰就是440Hz。可见，傅里叶变化之后我们得到了这个波形在频域上的表示，并且是正确的。但是在日常生活中我们所听到的声音永远不会只是一个单调的正弦波，而是由很多波叠加而成的。因此，稍微改动生成波形的代码来生成一个更加复杂的声音，仅需修改生成幅度的那一行代码，如下：

```
float amplitude = (sin(angle) + sin(angle * 2 + M_PI / 3) +
        sin(angle * 3 + M_PI / 2) + sin(angle * 4 + M_PI / 4))
        * 0.25;
```

以上代码中使用了四个正弦波叠加，并且每个正弦波都有自己的相位，相位是随机给的。为什么后面乘以0.25，是因为后续要将这个值再转换为SInt16表示的值，所以将其转换为-1～+1的范围内。使用Audacity软件打开生成的这个PCM文件，将横轴的刻度拉大，波形图如图8-9所示。

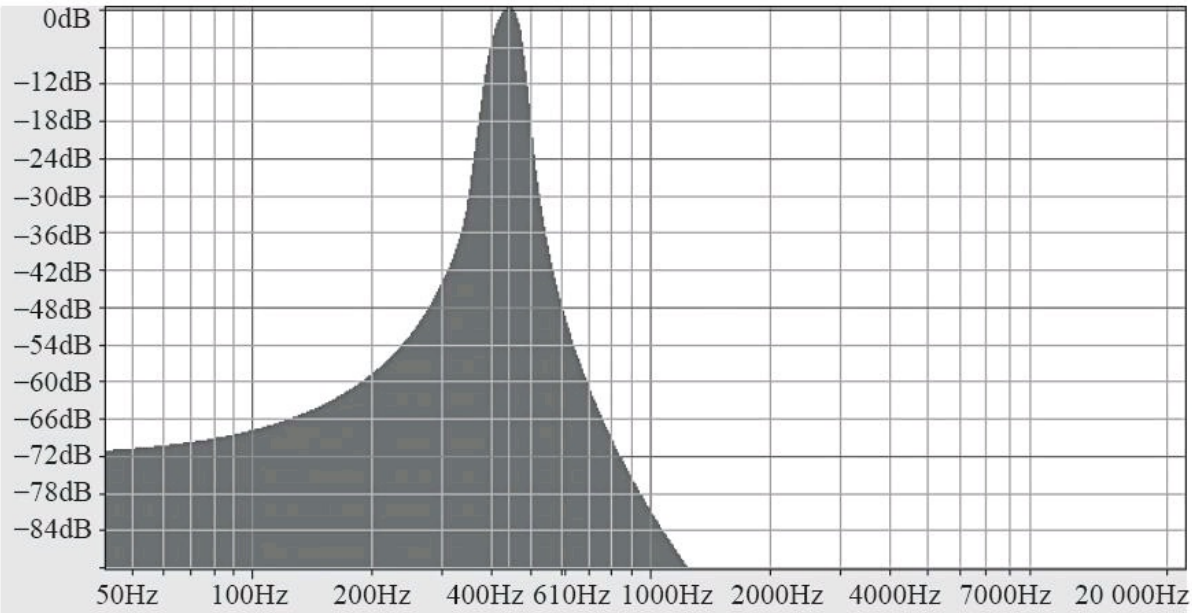


图　8-8

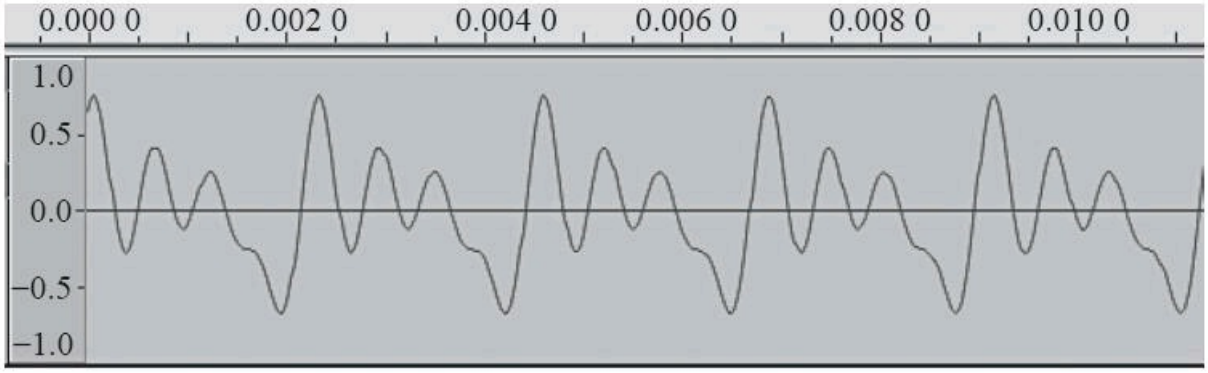


图　8-9

从图8-9可以看到，没有一个单一的正弦波（见图8-7）看起来那么规范，显然这是由很多个波叠加而成的，并且不同的波还有自己的相位，因为在时间为0的时候，能量不是从0开始的。再观察图8-9，还可以看出它具有周期特性，每个周期约为2.25ms。当然也可以由以上代码看出，最主要的频率还是440Hz所产生的正弦波频率，所以选中20ms的波形图来观察它的频谱图，如图8-10所示。

在图8-10中，第一个波峰在440Hz处，第二个波峰在880Hz处，第三个波峰在1320Hz处，第四个波峰在1760Hz处，由此可见，该频谱图类似于人所发出的非清音的频谱图，该频谱图的基频就是440Hz。

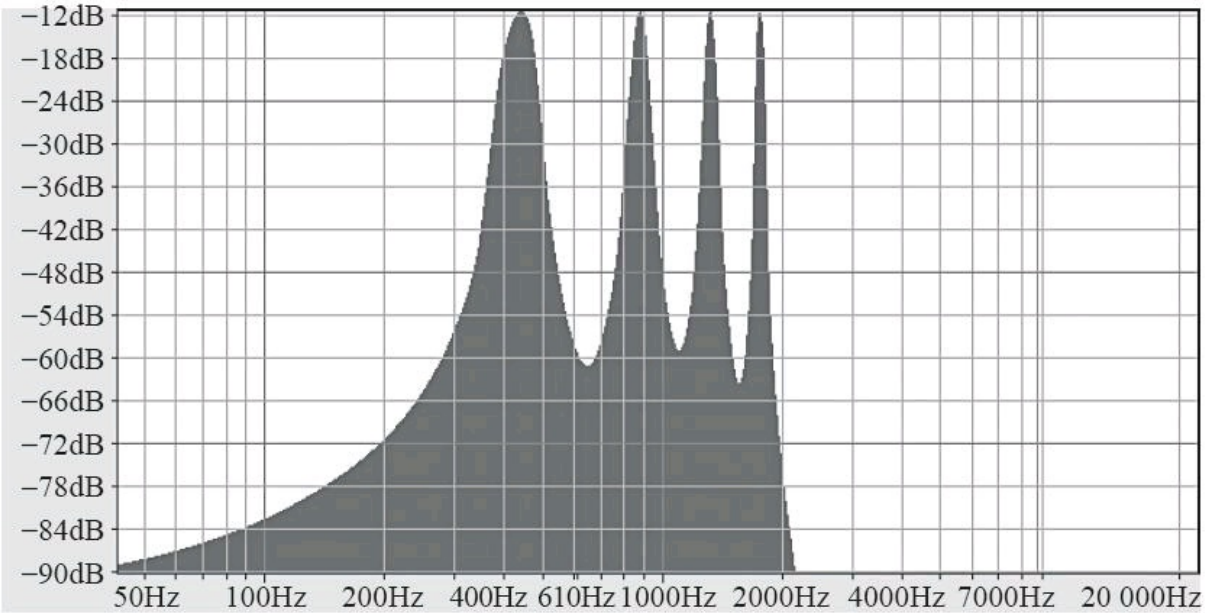


图　8-10

上面介绍了这么多波形图和频谱图，读者应该已经比较熟悉声音在时域上的表示，以及时域和频域的转换了。下面笔者会带领大家将声音的时域信号转换为频域信号，然后提取特征并做一些操作，让我们开始吧！

8.2　数字音频处理：快速傅里叶变换

本节介绍数字音频的处理，由8.1节介绍的内容可知，声音主要的表示形式就是时域和频域的表示，而音频处理就是针对声音分别在时域和频域上的处理。本节会详细介绍如何从时域和频域方面对音频做一些处理。对于时域方面的处理比较简单，不需要额外进行转换的操作，因为一般拿到的音频数据就是时域表示的音频数据。若要对音频的频域做处理，那么需将拿到的音频数据先转换为频域上的信号，再进行处理。那么如何转换为频域上的信号呢？在8.1节中提到过，使用FFT，即使用傅里叶变换，所以下面先学习傅里叶变换。

离散傅里叶变换简称为DFT，由于计算速度太慢，所以就演变出了快速傅里叶变换，即我们常说的FFT，在处理音频的过程中常使用MayerFFT来实现。本节不是讨论傅里叶变换的原理以及公式推导，而是讲解FFT的物理意义、如何使用FFT将时域信号转换为频域信号，以及如何利用逆FFT将频域信号重新转换为时域信号，同时在iOS平台会使用vDSP来提升效率，在Android平台的armv7的CPU架构以上，会使用neon指令集加速来提升性能，这样的安排相信会使读者更加深入地了解FFT，还可以迅速地将优化应用于自己的日常工作中。

1.FFT的物理意义

FFT是离散傅里叶变换的快速算法，可以将一个时域信号变换为频域信号。有些信号在时域上很难看出是什么特征的，但是，如果变换到频域之后，就很容易看出其特征了。这就是很多信号分析（声音只是众多信号中的一种）采用FFT变换的原因。虽然很多人都知道FFT是什么，可以用来做什么，以及怎么去做，但是却不知道做FFT变换之后的意义，下面就来和大家一块分析FFT的物理意义。

声音的时域信号可以直接用于FFT变换，假如N个采样点经过FFT之后，就可以得到N个点的FFT结果，为了方便进行FFT运算，通常N的取值为2的整数次幂，如512、1024、2048等。根据采样定理，采样频率要大于信号频率的两倍，所以假设采样频率为Fs，信号频率为F，采样点数为N，那么执行FFT之后的结果就是N个点的复数，每个复数可分为实部a和虚部b两部分，表示为：

```
z = a + b * i
```

每个点对应着一个频率点，而这个点的复数的模值就是这个频率点的幅度值，可以计算为：

```
amplitude = sqrt(a * a + b * b);
```

每个复数都会有一个相位，其在物理意义上代表的就是这个周期的波形的起始相位是多少，相位的计算如下：

```
phase = atan2(b, a);
```

由于输入的是声音信号，声音信号在时域上表示为一个一个独立的采样点，因此在做FFT变换之前，要先将其变换为一个复数，即将时域上某一个点的值作为实部，虚部则统一设置为0，由于所有输入的虚部都是0，因此导致FFT的结果是对称的，即前半部分和后半部分的结果是一致的。所以在对声音信号做FFT变换之后，只要使用前半部分就可以了，因为后半部分是对称的，无需使用。那么执行FFT得到的结果与真实的频率有什么关系呢？

还是用8.1.4节中生成音频文件的代码来说明，利用以下公式来生成一段采样频率为44100Hz的音频文件：

```
float amplitude = (sin(angle) + sin(angle * 2 + M_PI / 3) +
        sin(angle * 3 + M_PI / 2) + sin(angle * 4 + M_PI / 4))
        * 0.25;
```

拿到这个音频文件后，先去做一个FFT，具体如何操作，这里暂不讨论。先把FFT的计算当做一个黑盒子，给它输入音频的时域信号，得到频域信号。由于这个音频文件的采样频率为44100Hz，做FFT的窗口大小是8192，那么生成FFT结果的第一个点的频率就是0Hz，最后一个点的频率就是44100Hz，而共有8192个点，所以相邻两点之间表示的频率差值如下：

```
44100 / 8192 = 5.3833Hz
```

这就是我们通常所说的使用8192作为窗口大小来给采样频率44100Hz的声音样本做傅里叶变换，得到的结果分辨率是5.3833Hz。接下来，在FFT的结果数组中找出第一个峰值（即第一个最大的值），可以发现是Index位置为82的元素，计算出它代表的频率，如下：

```
5.3833 * 82 = 441.43Hz
```

由于声音源是由4个波叠加而成的，因此找到的第一个峰值是频率最低的峰值，也是440Hz所代表的峰值，那为什么我们得到的结果是441.43Hz呢？这就是前面所说的分辨率的问题，如果想准确地算出440Hz，就要增加窗口大小以提高频带分布的分辨率，才能使计算出来的频率更加准确。接着看第二个峰值，它是在Index为163的位置，计算出它代表的频率，如下：

```
5.3833 * 163 = 877.478Hz
```

得到了第二个波峰的频率信息后，接着可以计算出第三个波的频率为5.3833×245=1319Hz，第四个波的频率为5.3833×327=1760.3Hz，这与图8-10看到的波峰分布情况是一致的。每个点的峰值以及相位的计算也可用上述公式计算出来，这里不再赘述。其实FFT就是把多个波叠加后的时域信号，可以按照频率将各个波拆开，进行更清晰的展示。下面会更加详细讲解如何做FFT，以及如何在移动平台上进行优化。

2.MayerFFT的使用

在C++语言中进行FFT变换时，最常使用的就是MayerFFT的实现，下面就来看看如何使用MayerFFT将音频文件进行FFT转换。

先下载一个MayerFFT的实现，它的实现虽然比较复杂（这里不做讨论），但是已经比较好地封装在一个类中，这个类也可以在本章的代码仓库中找到，下面编写一个类文件FFT-Routine将具体的实现封装起来，然后提供接口给外界调用。

构造函数和析构函数的代码如下：

```
FFTRoutine(int nfft);
~FFTRoutine();
```

从以上代码中可以看到，构造函数中有一个参数nfft，这个参数代表FFT运算中一个窗口的大小，这也是做FFT最基本的设置，为避免频繁的内存开辟和释放操作，在构造函数的实现中，要开辟一个nfft大小的浮点类型的数组，以供做FFT运算时使用，在析构函数中销毁这个浮点类型数组。接下来看看从时域信号到频域信号的正向FFT变换的接口，代码如下：

```
void fft_forward(float* input, float* output_re, float* output_im);
```

从接口中也可以看出，第一个参数是输入的时域信号（浮点类型表示），第二个参数和第三个参数分别代表了转换为频域信号之后实部和虚部的两个数组，这个函数的具体实现如下。

首先要将输入数据复制到在构造函数开辟的数组中，然后调用MayerFFT进行FFT变换：

```
memcpy(m_fft_data, input, sizeof(float) * nfft);
MayerFFT::mayer_realfft(nfft, n_fft_data);
```

待MayerFFT执行完时域到频域转换之后，实部和虚部的数据也已经存放到n_fft_data中了，只不过是实部和虚部是对称存储的，我们需要按照顺序取出来，但是要先将直流分量（第一个元素）的虚部置为0，代码如下：

```
output_im[0] = 0;
for(int i = 0; i < nfft / 2; i++) {
    output_re[i] = n_fft_data[i];
    output_im[i + 1] = n_fft_data[nfft-1-i];
}
```

这样就可以利用开源的MayerFFT做了声音信号的时域到频域的转换。接下来再执行一个逆FFT操作，即把频域数据变换为时域数据，接口代码如下：

```
void fft_inverse(float* input_re, float* input_im, float* output)
```

从接口代码中可以看到，输入的是频域信号，分为实部和虚部，输出的是时域信号，即一个浮点型的数组。来看一下具体的实现，先将输入的实部和虚部按照MayerFFT中存储复数的存储格式还原回去，代码如下：

```
int hnfft = nfft/2;
for (int ti=0; ti<hnfft; ti++) {
    m_fft_data[ti] = input_re[ti];
    m_fft_data[nfft-1-ti] = input_im[ti+1];
}
m_fft_data[hnfft] = input_re[hnfft];
```

然后调用MayerFFT进行逆FFT运算，运算之后的结果还是存储到m_fft_data这个浮点数组中，而此时这个浮点数组中存储的就是时域信号了，最终，把时域信号拷贝到输出参数中，代码如下：

```
MayerFft::mayer_realifft(nfft, m_fft_data);
memcpy(output, nfft_data, sizeof(float) * nfft);
```

这样就可以将频域信号又转换回时域信号了，这种逆FFT运算会在一些音频处理中有特殊的作用，比如变调效果器（PitchShift）中就会用到这种操作。MayerFFT的运算都是在CPU上进行计算的，跨平台特性可以做得比较好（因为只有一个CPP的实现文件），但是性能是它的瓶颈，而在移动平台上最需要注意的就是性能问题，所以在下面会给出在各个平台上的优化处理。

3.iOS平台的vDSP加速

前面已经讲解了如何使用工具类MayerFFT来将时域信号的音频转换为频域的表示。而在移动平台上我们最关心的就是效率，所以下面要针对移动平台上给出相应的实现优化。在iOS平台上可使用iOS提供给开发者的vDSP来执行FFT的优化操作。苹果为开发者提供的vDSP无论是在iOS平台上还是在Mac OS平台上都可以使用，当然，在使用之前，必须将Accelerate这个framework引入我们的项目中，具体做法就是在工程文件的Build Phases的Link Binary with Libraries中添加

Accelerate.framework这个库，然后要使用到vDSP的类时用以下代码引入头文件：

```
#include <Accelerate/Accelerate.h>
```

现在就可以使用vDSP来加速运算了，vDSP中提供了很多函数来完成DSP运算，本节只介绍FFT的运算以及与FFT运算相关的函数。使用FFT时，需要先构造出一个指针类型的OpaqueFFTSetup的结构体，调用函数如下：

```
OpaqueFFTSetup *fftsetup;
int m_LOG_N = log2(nfft);
fftsetup = vDSP_create_fftsetup(m_LOG_N,kFFTRadix2);
```

第一个参数是在进行FFT转换时使用的窗口大小（为取log2之后的数值），在vDSP中，FFT的窗口一般是2的N次方，所以这里传入以2为底取对数的数值；第二个参数一般传递iOS提供的枚举类型kFFTRadix2，这样就构造出了fftSetup这个结构体类型。然后分配一个DSPSplitComplex的复数类型作为FFT的结果输出，代码如下：

```
size_t halfSize = nfft / 2;
splitComplex.realp = new float[halfSize + 1];
memset(splitComplex.realp, 0, sizeof(float) * (halfSize + 1));
splitComplex.imagp = new float[halfSize + 1];
memset(splitComplex.imagp, 0, sizeof(float) * (halfSize + 1));
```

如上述代码所示，由于用声音做FFT的结果是对称的，因此只需要去取半部分的数据。那么，为复数的实部和虚部分配空间时，也只要分配一半的大小就可以了。结构体中的realp代表了实部的部分，imagp代表了虚部的部分，当分配好这个结构体之后，就可以进行FFT运算了。首先，把要做FFT的时域信号放入上面构造好的复数的结构体中，代码如下：

```
vDSP_ctoz((DSPComplex*)input, 2, &splitComplex, 1, halfSize);
```

输入的时域信号是float类型的数值，首先强制类型应转换为复数类

型，然后利用vDSP_ctoz这个函数将复数的实部和虚部分开存储，即原始的复数结构体是交错（interleaved）存放的，而转换之后就是平铺（Plannar）存放的。将转换之后的结构体作为FFT运算的输入，代码如下：

```
vDSP_fft_zrip(fftsetup, &splitComplex, 1, m_LOG_N, kFFTDirection_Forward);
```

从上述代码可以看到，第一个参数是最开始构造的OpaqueFFTSetup指针类型的结构体，第二个参数是时域信号填充的复数结构体，后续的参数指定了行距和大小，最后一个参数代表做正向的FFT，FFT的结果还是会放入这个复数结构体中，我们可以转换为自己的float指针类型的输出，代码如下：

```
for (int i = 0; i < halfSize; i++) {
    outputRe[i] = splitComplex.realp[i];
    outputIm[i] = splitComplex.imagp[i];
}
```

待使用完FFT之后，要将分配的OpaqueFFTSetup指针类型的结构体销毁掉，并且还要将分配的复数结构体内部的实部和虚部部分的数组销毁掉，代码如下：

```
if (fftsetup) {
    vDSP_destroy_fftsetup(fftsetup);
    fftsetup = NULL;
    }
if (splitComplex.realp) {
    delete[] splitComplex.realp;
}
if (splitComplex.imagp) {
    delete[] splitComplex.imagp;
    splitComplex.imagp = NULL;
}
```

销毁完毕后，当然也可以利用vDSP来做逆FFT（IFFT）的运算，从名字上来看，这是FFT运算的一个逆过程（从频域信号转换为时域信号）。上面在讲解做FFT运算的时候，提到过最后一个参数代表了做正向的FFT，如果使用kFFTDirection_Inverse，则代表要做逆向的FFT。一个完整的做逆FFT的代码如下：

```
void fft_inverse(float* input_re, float* input_im, float* output) {
    DSPSplitComplex tsc;
    tsc.realp = input_re;
    tsc.imagp = input_im;
    vDSP_fft_zrip(fftsetup, &tsc, 1, m_LOG_N, kFFTDirection_Inverse);
    vDSP_ztoc(&tsc, 1, (DSPComplex*)output, 2, halfSize);
    float scale = 1.0 / m_nfft;
    vDSP_vsmul(output, 1, &scale, output, 1, m_nfft);
}
```

上述代码中，首先将实部和虚部放到复数的结构体中，然后调用FFT运算函数。注意，此时最后一个参数的传递代表要做逆FFT运算。接着调用vDSP_ztoc函数将平铺（Plannar）分布的复数转换为交错（interleaved）分布的output中，最终逆FFT的结果需要除以窗口大小才可以还原为原来的时域信号，其中当除以窗口大小时，使用了vDSP提供的vDSP_vsmul函数来提升性能。

4.Android平台的Ne10加速

在Android平台上，我们能做的优化就是使用Neon指令集来加速运算。Neon指令集是一种单指令多数据的计算模式，而开发者直接使用Neon指令集来实现运算的加速以及实现FFT，成本太高了，一则是FFT的实现过于复杂，二则是测试成本也比较繁琐，所以下面使用开源的Ne10这个库来实现Android平台的性能提升。

Ne10这个库的介绍与安装可以参见附录部分，而本节所介绍的仅仅是如何使用Ne10这个库来实现FFT与逆FFT的运算。首先引入Ne10的头文件，代码如下：

```
#include "NE10.h"
```

然后构造一个FFT运算的配置结构体。注意，这个配置项结构体我们选用的是实数到复数（r2c）的配置项，因为这种FFT配置项在做音频的FFT运算时更适合，代码如下：

```
ne10_fft_r2c_cfg_float32_t cfg;
cfg = ne10_fft_alloc_r2c_float32(nfft);
```

构造这个结构体需要传入的参数是使用FFT运算时的窗口大小，由于我们使用的是Ne10这个库，所以在进行FFT运算时需要将声音时域信号构造成Ne10需要的结构体来进行输入。由于这里输入的是float类型的数组，因此Ne10中定义的也是float类型的数据，代码如下：

```
ne10_float32_t* in;
in = (ne10_float32_t*) NE10_MALLOC (nfft * sizeof (ne10_float32_t));
```

为了获得FFT运算后的结果，也需要构造出输出结构体来接受FFT的运算结果，因为FFT的输出是复数的结构，分为实部和虚部，所以在Ne10中也采用了复数的结构，但它是由单独的结构体来表示的，代码如下：

```
ne10_fft_cpx_float32_t* out;
out = (ne10_fft_cpx_float32_t*) NE10_MALLOC (nfft * sizeof
         (ne10_fft_cpx_float32_t));
```

准备好以上内容后，就可以进行真正的FFT运算了，将输入的音频时域信号的float数组拷贝到上述定义的输入结构体in中，然后调用Ne10这个库提供的FFT运算函数进行FFT运算。注意，这里我们使用的运算函数是实数到复数的FFT运算，这种FFT运算更适合在音频场景下做FFT运算，最终得到的结果会放到上述分配的结构体out中，代码如下：

```
memcpy(in, input, sizeof(float) * m_nfft);
ne10_fft_r2c_1d_float32_neon(out, in, cfg);
for (int i = 0; i < m_nfft / 2; i++) {
    output_re[i] = out[i].r;
    output_im[i] = out[i].i;
}
```

经过FFT运算之后的结果就存在于结构体out中，取出前半部分数据赋值给输出的实部的float数组和虚部的float数组中。最终在做完所有的FFT运算之后，需要释放掉分配的资源，包括FFT配置项、输入结构体、输出结构体，代码如下：

```
NE10_FREE(in);
NE10_FREE(out);
NE10_FREE(cfg);
```

这类似于iOS平台的vDSP的优化方案，Ne10提供的FFT方案肯定也提供了逆FFT的运算操作，代码如下：

```
for (int i = 0; i < m_nfft / 2; i++) {
    out[i].r = input_re[i];
    out[i].i = input_im[i];
 }
ne10_fft_c2r_1d_float32_neon(in, out, cfg);
memcpy(output, in, sizeof(float) * m_nfft);
```

从以上代码可以看到，调用逆FFT运算时，调用的是c2r的FFT函数，即使用从复数到实数的转换函数来完成逆FFT的运算，最终再把in这个结构体中的实数复制到整个函数的output中，即时域信号的float数组中去。

8.3 基本乐理知识

本节来学习基本的乐理知识，为什么要学习乐理知识呢？因为在处理声音的时候，有一部分是与伴奏或者背景音乐有关的，或者与唱歌或听歌处理相关的，这就需要我们掌握一些基本的乐理知识了。

8.3.1　乐谱

为了能记录之前发生的事情，人们撰写出了历史，而不是口口相传，导致后世不知前世之事。类似的问题发生在各个领域，比如医学、数学、化学、物理等各个领域。同样，音乐界也不例外，人们为了能使美好的乐曲保留下来，并且便于学习和交流，于是创造出了各种各样的记谱方法，而这些记谱方法就是我们所说的乐谱。记谱的方法有很多种，像壁画一样，世界各地的人都会创造音乐，也都会有自己的记谱方法，比如在中国古代广为流传的《工尺谱》。但是现在被我们广为应用并且熟悉的有两种记谱方法，一种是用阿拉伯数字表示的《简谱》，一种是国际上流行通用的《五线谱》。

下面看看《我是一个粉刷匠》这首歌曲的第一句，使用简谱记谱方法和五线谱的记谱方法有何区别，简谱记谱方法如图8-11所示。

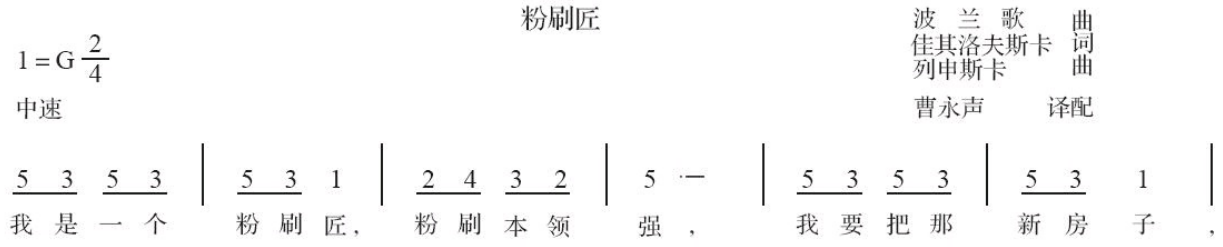


图　8-11

在图8-11中，左上角表示节拍信息和谱号。谱号G表示高音谱号；2/4代表拍号，每一拍（代表时间长度）是四分音符的时值，每一小节有两拍（代表节奏信息）。乐曲的第一行表示具体的音符排练顺序以及节奏信息，5 3 表示Sol Mi两个连起来算作一拍，而小节之间是使用|进行分割的，第二个小节的第二拍do自己占用了这一拍的时值，所以前两个小节连接起来就是sol mi sol mi sol mi do。这是一首儿歌，音符都在一个音组之内（一个八度之内），而有些歌曲可能要跨越多个音组，对于低音就在数字下面加点来表示，高音则在数字上面加点来表示，加的点越多，代表越低或越高。五线谱的记谱方法如图8-12所示。

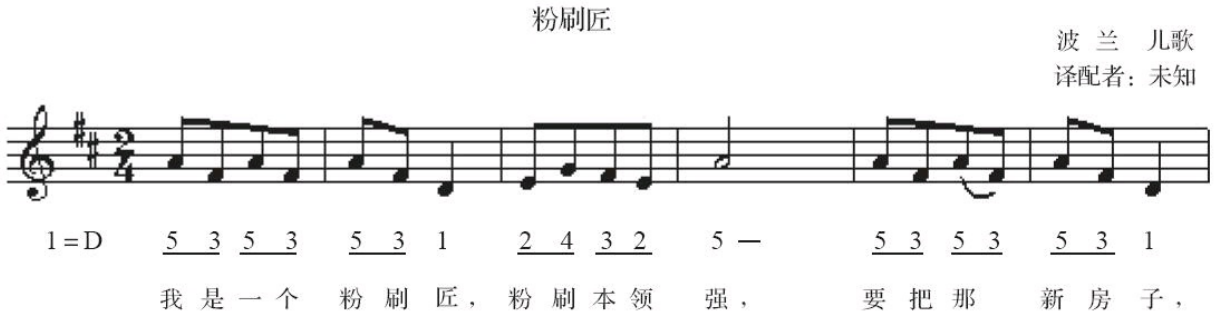


图 8-12

图8-12中同时有简谱和五线谱。下面介绍五线谱，第一个𝄞表示高音谱号；2/4代表拍号，每一拍都是四分音符，每一小节有两拍，其余的是谱表部分。制作五线谱时，首先应画出五根线，五根线画出来之后中间就形成四个空行，这在五线谱中称为间，所以五线谱由五条平行的“线”和四条平行的“间”组成。而线和间的命名从下向上依次是第一线、第一间、第二线、第二间、第三线、第三间、第四线、第四间、第五线。音符就画在这些线和间上，具体画在哪个线或哪个间上，所表达的音高也不一样。这九个音高必定不能满足我们想表达的音符，那应该怎么办？五线谱的上边和下边都可以再加线和间，向上即所谓的上加一间，上加一线，直至上加五线，向下即所谓的下加一间，下加一线，直至下加五线。但是，即使是这样，也只能够表示29个音符（本来可以表示9个音符，上边可以加五间五线，下边可以加五间五线），能表示的音符还不够。因此就有了谱号，即在五线谱的最开始要标记到底是高音谱号还是低音谱号或者中音谱号使用最多的是高音谱号和低音谱号。高音谱号如图8-13所示。

高音谱号也称G谱号。对于高音谱号，下加一线是do（中央C的do），三间就是高八度的do，上加两线是再高一个八度的do。关于度数，后面会有详细的介绍，大家可以理解为更高的一组音。低音谱号如图8-14所示。

图　8-13

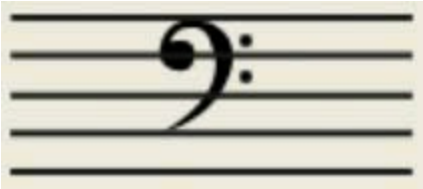


图　8-14

低音谱号也称F谱号。对于低音谱号，上加一线是do（中央C的do），二间是低八度的do，下加二线是再低八度的do。图8-15所示的是将简谱、五线谱、唱名与钢琴键盘画在了一个图中，大家可以对照着看一下，以便加深理解。

不同的谱号代表在五线谱中每个线或者每个间所表达的音高是不一样的。五线谱是世界范围内通用的记谱方法，因为它是最科学、最容易理解的记谱方法。

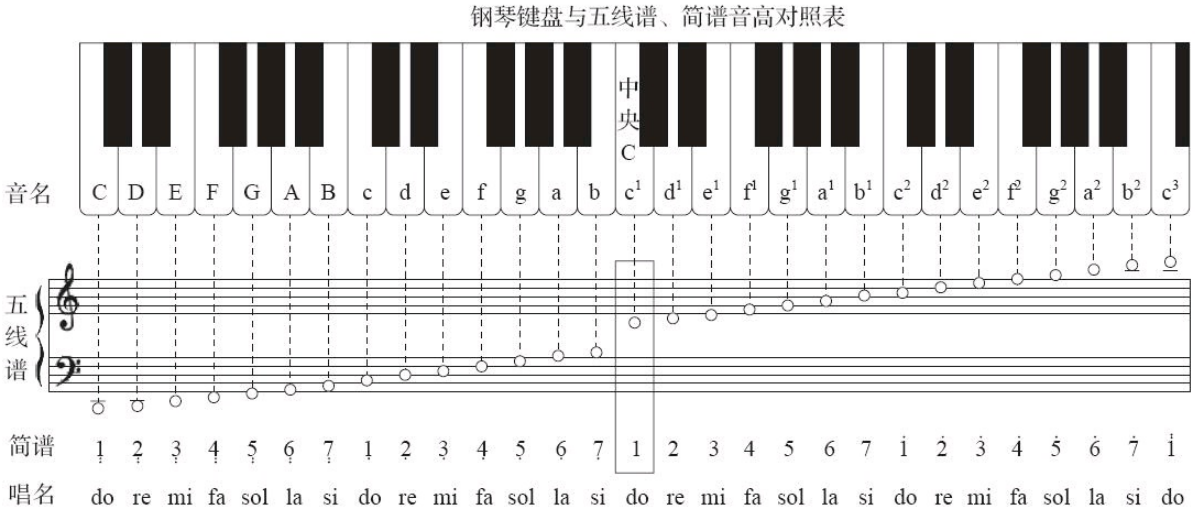


图　8-15

五线谱由三部分组成，分别是谱号、谱表和音符，其中前两部分已经介绍完成，剩下的就是音符。音符比较复杂，因为它涉及的概念比较多，所以下面分别从音符的音高和时值两个方面来进行介绍。

8.3.2　音符的音高与十二平均律

如何描述一个乐谱？常用的有五线谱、简谱等。无论是五线谱，还是简谱，都是表示音符的音高和音符的时值。时值是由节拍信息定义的，而本节讨论的是音符的音高部分。音符也有两种表述方式：第一种就是大家经常唱的do re mi fa sol la si，即唱名，也是大家最常使用的；第二种就是音名，即C D E F G A B。

基本乐理的概念比较多，下面我们逐一来理清楚。图8-16所示的键盘图可以看到一个中央C的白键，音名记为c1，从c1向右数，一直数到c2，这一串连续的音称为一个音阶。同时，c2这个白键所发出声音的频率恰好是c1这个白键发出声音频率的2倍。一个音阶同时也称为一个音组，在键盘上中央C所在的这个音组称为小字一组，向右数下去的每个组分别是c2所在的小字二组，c3所在的音组是小字三组，c4所在的音组是小字四组。那么，向相反的方向数的话，即c所在的音组是小字组，C所在的音组是大字组，c1所在的音组是大字一组。可这么多组如何记忆呢？其实很简单，大家只要记住音名就可以了，首先找到比中央C低八度的音名为c的这一组，所有的音名都是小写字母表示，所以称为小字组。从这一音组向右数每增加一个八度音名都会在小写字母后加1，同时音组就成为小字几组（比如中央C所在的音组成为小字一组）。再看比中央C低2个八度的音名是C的这一组，所有的音名都是大写字母表示，所以称为大字组，从这一组向左数，每低一个八度音名就在大写字母后边加1，而所在的音组就是大字几组，这样就能比较简单地记住音名及所有的音组了。

这里不得不再引入一个音程的概念。音程是指两个音符之间的音高关系，一般用度来表示。对照图8-16中的钢琴键盘来看，从c1到c1称为一度，从c1到d1称为两度，依此类推。我们常说从c1到c2之间差了八度，而八度实际上是指音程之间的关系。

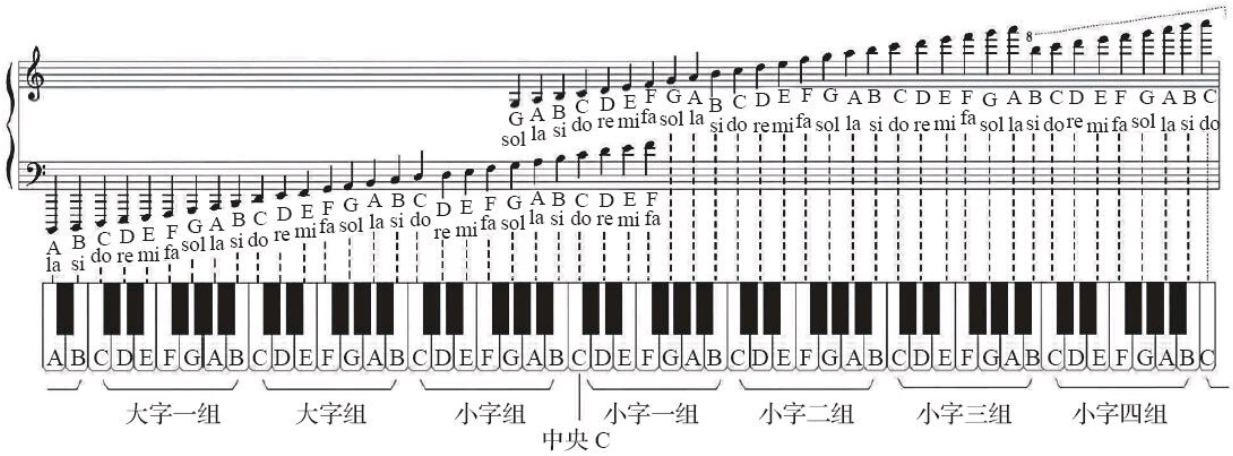


图　8-16

可以数一下，所有的键（包括黑键和白键）加起来恰好是十二个，而相邻键之间称为一个半音，即e1和f1之间是一个半音，c1和d1之间是两个半音即一个全音，而从c1到c2有12个半音，这也就引出了即将和大家介绍的概念——十二平均律。十二平均律的定义如下。

十二平均律亦称“十二等程律”，世界上把一组音分成十二个半音音程的律制，各相邻两律之间的振动数之比完全相等。

钢琴就是十二平均律制的乐器，国际标准音规定，钢琴的a1（小字组的A音，其实就是中央C这个音节的A音）的频率是440Hz，并且规定每相邻半音的频率比值为2^（1/12）≈1.059463。根据这两个规定，可以得出钢琴上每个琴键音的频率，比如a1的左边的黑键升g1的频率为：

440/1.059463=415.305Hz

a1右边的黑键升a1的频率为：

440×1.059463=466.16372Hz

依照这种运算，可计算出a的频率为220Hz，a2的频率为880Hz，恰好差了12个半音频率是一倍的关系。这种定音方式就是“十二平均律”。为什么钢琴称为乐器之王，是因为钢琴的音域范围为A2（27.5Hz）～c5（4186Hz），几乎囊括了乐音体系中的全部乐音。

而有的读者文言功底比较深厚，可能知道中国传统五声音阶为：

宫gōng、商shāng、角jué、徵zhǐ、羽yǔ

这是我国五声音阶中五个不同音的名称，对应唱名的话，宫等于Do，商等于Re，角等于Mi，徵等于Sol，羽等于La，这比现代乐谱中少了Fa和Xi这两个音。其实在我们的古音阶中，变宫与变徵分别对应于Fa和Xi。关于中国的五声音阶最早的记载出现在春秋时期，可见音乐是不分国界、不分时间的，每个国家都有自己的乐律，而中国音乐史上著名的“三分损益法”就是古代发明制定音律时所用的生律法。

8.3.3　音符的时值

音符的时值是指这个音符所持续的时间。平时大家衡量时间是有单位的，而音符是如何体现自己的单位的呢？其实就是长得不一样。音符一般由三部分组成，即符头、符干、符尾。下面我们来看一个简单的音符，如图8-17所示。

大家应该很熟悉图8-17中的这个音符，因为在很多地方都以此音符来表示音乐，这个音符是一个八分音符。音符共有多少种表示呢？请看图8-18。

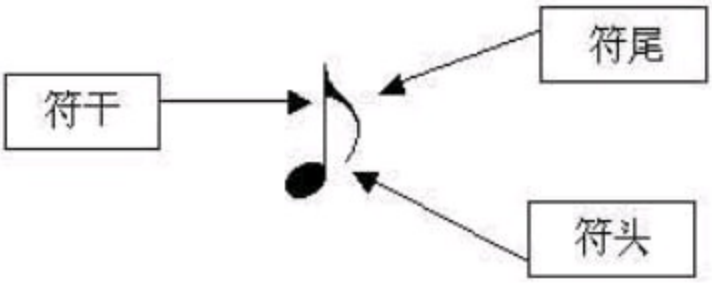


图　8-17

名　称	形　状	时　值 （以四分音符为一拍）	比　列
全音符	𝅝	4 拍	1
二分音符	𝅗𝅥（𝄼）	2 拍	$\frac{1}{2}$
四分音符	♩（𝄽）	1 拍	$\frac{1}{4}$
八分音符	♪（𝄾）	$\frac{1}{2}$拍	$\frac{1}{8}$
十六分音符	𝅘𝅥𝅯（𝄿）	$\frac{1}{4}$拍	$\frac{1}{16}$
三十二分音符	𝅘𝅥𝅰（𝅀）	$\frac{1}{8}$拍	$\frac{1}{32}$

图　8-18

在图8-18中，一拍的单位一般为一个四分音符，用一个实心符头和一个符干表示。为什么符干有的向上画，有的向下画？主要是为了在五线谱中更加容易被识谱者观察。在五线谱中规定符头要左低右高呈椭圆

形，在五线谱三线以上的符干要向下画并且在符头的左边，在三线以下的符干要向上画并且在符头的右边，符杆的长度一般以一个八度为单位。

大家可能觉得图8-12所展示的粉刷匠的五线谱中的音符并不存在于这个表中，不要着急，在五线谱中还有一种画法叫共用符尾，像粉刷匠中的第一个小节就有4个八分音符共用一个符尾。

还有一些休止符、变化音等比较特殊的音符标记方法，这里不一一讨论。

8.3.4　节拍

节拍是指强拍和弱拍的组合规律，有很多有强有弱的音，在长度时间内，按照一定的顺序反复出现，形成有规律的强弱变化，使得整个乐谱更有节奏感。根据强、弱的不同，可以组合成各种情绪、不同风格的乐曲。

音符除了记录音符的音高外，还要记录音符的时值。时值的表示就是通过节拍来表示的。

拍号是乐谱小节的书写标准，在乐谱中以一个分数的形式来表示。比如4/4，分母的4表示以一个四分音符为一拍，分子的4表示每小节有四拍。其实拍号是一个相对的时间单位，它只能表示每一个小节里有几个拍子以及每个拍子的时值。但是，具体占多长时间该怎么表示呢？这又要引入另外一个概念，即BPM。

BPM（Beat Per Minute），即每分钟节拍数的单位。最浅显的理解就是在一分钟时间内声音节拍的数量（相当于拿一个节拍器在一分钟之内发出节拍的数量），这个数量的单位便是BPM，也叫拍子数。BPM就是每分钟的节拍数，是全曲速度标记，是独立在曲谱外的速度标准，一般以一个四分音符为一拍，60BPM为一分钟演奏均匀60个四分音符（或等效的音符组合）。由于60BPM对应的曲目速度为一分钟均匀演奏60个四分音符（或等效音符组合），所以一个四分音符（或等效音符组合）的时值应为1秒，而对应的提供给演奏者显示的演奏速度。一般情况下，歌曲分为慢速（节奏）歌曲、中速歌曲、快速（节奏）歌曲，对应于节拍的话，慢速每分钟40～69拍；中速每分钟约90拍；快速每分钟108～208拍。

8.3.5 MIDI格式

MIDI（Musical Instrument Digital Interface，乐器数字接口），是20世纪80年代初为解决电声乐器之间的通信问题而提出的。MIDI是编曲界最广泛的音乐标准格式，可称为计算机能理解的乐谱。它用音符的数字控制信号来记录音乐，一首完整的MIDI音乐只有几十KB，而且能包含数十条音乐轨道。MIDI中存储的不是声音，而是音符、控制参数等指令，能解析MIDI的设备会根据MIDI文件中的指令来播放。具体音符在MIDI中是如何表示的呢？可以使用一张键盘图来了解音符对应的MIDI的名字以及MIDI值，如图8-19所示。

从图8-19可以看到，中央C的频率为261.63Hz，而所对应的MIDI名称是C4，对应的MIDI值是60。对于MIDI值，MIDI的名称该如何记忆呢？其实很简单，前面在讲音组时，已经知道键盘最低的音组是大字二组（当然大字二组只有A和B两个音），大字二组的第一个音A就是MIDI的A0，第二个音B就是MIDI的B0。在图8-19中，向下数就可以数出所有的MIDI名称。而对于MIDI值，我们只要记住中央C（c1）的MIDI值是60或者小字一组的A音（a1）是69，然后根据十二平均律就可以全部计算出来。

具体的MIDI制作就不在这里详细展开了，可以使用某一些电子钢琴，甚至现在有一些App都可以将乐者弹奏的音符（包括音高和时值）记录下来。那我们解析出了对应的时间上的音符能做什么呢？其实有很多种用处，如果我们知道一首歌曲对应的MIDI信息，在K歌应用中就可以给用户做打分，即评测用户唱的音高和MIDI中的音高是否匹配，从而作为打分的依据；也可以进行一些节奏修正和音高修正，因为MIDI中包含了时间信息和音高信息，我们可以对用户唱的歌曲进行对齐节奏和修正音高等操作。

下一节开始会介绍混音效果器，以让大家了解具体的混音过程，帮助大家对声音做出更好的处理。

MIDI number		Note name	Keyboard	Frequency Hz		Period ms	
21		A0		27.500		36.36	
	22				29.135		34.32
23		B0		30.868		32.40	
24		C1		32.703		30.58	
	25				34.648		28.86
26		D1		36.708		27.24	
	27				38.891		25.71
28		E1		41.203		24.27	
29		F1		43.654		22.91	
	30				46.249		21.62
31		G1		48.999		20.41	
	32				51.913		19.26
33		A1		55.000		18.18	
	34				58.270		17.16
35		B1		61.735		16.20	
36		C2		65.406		15.29	
	37				69.296		14.29
38		D2		73.416		13.62	
	39				77.782		12.86
40		E2		82.407		12.13	
41		F2		87.307		11.45	
	42				92.499		10.81
43		G2		97.999		10.20	
	44				103.83		9.631
45		A2		110.00		9.091	
	46				116.54		8.581
47		B2		123.47		8.099	
48		C3		130.81		7.645	
	49				138.59		7.216
50		D3		146.83		6.811	
	51				155.56		6.428
52		E3		164.81		6.068	
53		F3		174.61		5.727	
	54				185.00		5.405
55		G3		196.00		5.102	
	56				207.65		4.816
57		A3		220.00		4.545	
	58				233.08		4.290
59		B3		246.94		4.050	
60		**C4**		**261.63**		**3.822**	
	61				277.18		3.608
62		D4		293.67		3.405	
	63				311.13		3.214
64		E4		329.63		3.034	
65		F4		349.23		2.863	
	66				369.99		2.703
67		G4		392.00		2.551	
	68				415.30		2.408
69		**A4**		**440.00**		**2.273**	
	70				466.16		2.145
71		B4		493.88		2.025	
72		C5		523.25		1.910	
	73				554.37		1.804
74		D5		587.33		1.703	
	75				622.25		1.607
76		E5		659.26		1.517	
77		F5		698.46		1.432	
	78				739.99		1.351
79		G5		783.99		1.276	
	80				830.61		1.204
81		A5		880.00		1.136	
	82				932.33		1.073
83		B5		987.77		1.012	
84		C6		1046.5		0.955 6	
	85				1108.7		0.9020
86		D6		1174.7		0.851 3	
	87				1244.5		0.8034
88		E6		1318.5		0.758 4	
89		F6		1396.9		0.715 9	
	90				1480.0		0.6757
91		G6		1568.0		0.637 8	
	92				1661.2		0.6020
93		A6		1760.0		0.568 2	
	94				1864.7		0.5363
95		B6		1975.5		0.506 2	
96		C7		2093.0		0.477 8	
	97				2217.5		0.4510
98		D7		2349.3		0.425 7	
	99				2489.0		0.4018
100		E7		2637.0		0.379 2	
101		F7		2793.0		0.358 0	
	102				2960.0		0.3378
103		G7		3136.0		0.318 9	
	104				3322.4		0.3010
105		A7		3520.0		0.284 1	
	105				3729.3		0.2681
107		B7		3951.1		0.253 1	
108		C8	J. Wolfe, UNSW	4186.0		0.238 9	

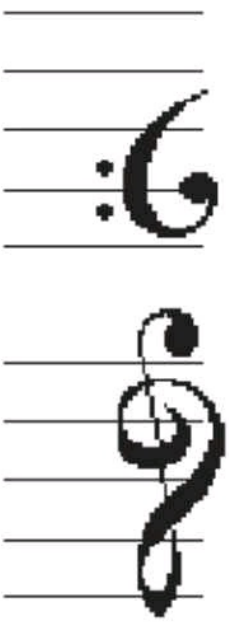


图　8-19

8.4 混音效果器

在音乐类App中，进行混音处理是必不可少的一项工作，而混音这门学科也是非常复杂的。作为一个优秀的混音师，不仅要有编曲经验，而且要会乐器懂乐理，还要能分辨什么样的声音是好声音，同时会使用混音的工具。本节会介绍一些混音中的基础知识，包括一些常用混音工具的使用。

8.4.1　均衡效果器

均衡效果器又称为均衡器（Equalizer），其最大的作用就是决定声音的远近层次。我们时常听到别人说这首歌曲是重金属风格的歌曲，或者说这首歌曲是舞曲风格等，其实就与声音的远近层次有关。不同歌曲风格的区别在于声音在不同频段的提升或衰减。

均衡效果器具有美化声音的作用，即调整音色，每个人由于自身声道、颅腔、口腔的形状不同，导致音色不同。如果这个用户所发出的声音在低频部分比较薄弱，就可以在低频部分予以增强，使得整个声音听起来更加温暖；那个用户所发出的声音在高频部分又过于强烈（薄弱），则可以在高频部分予以减弱（增强），可以使声音听起来不那么刺耳（更加嘹亮）。当然，专家级别的混音师在为歌手处理后期混音时，会有更复杂的调节方法，比如这个歌手的声音低频部分有瑕疵，可以提高中频部分来掩盖有瑕疵的低频段的声音。

均衡器最早是用来补偿频率缺陷的，因为那时音频设备的信号品质很差，在传输过程中损失非常严重，到最后除非进行信号补偿，否则信号就会变得极差。而现在均衡器更多的应用在掩盖歌手的某一个频段的声音缺陷，或者增强某一个频段的声音优势上。

接下来看一下声音的频率分布。

1）超低频。1～20Hz范围，大约是4个八度的范围。这个声音，人的耳朵是听不到的，如果音量很大，我们的耳朵能够感觉到一种压力感，比如地震就可以产生这种频率。一般来说，这个频段和音乐没有关系。

2）非常低频。20～40Hz范围，1个八度（频率差两倍）的范围。这个频率也是很低的，一般远距离的雷声以及风声在这个频段里。这个频段的音效，在音乐中还是会经常用到的。

3）低频。40～160Hz范围，2个八度的范围。电贝斯的声音属于这个频段，当然，低音提琴、钢琴也拥有这个音域。这就是音乐中常用的频段了。男低音也可以发出这个频段中的一部分声音。

4）低中频。160～315Hz范围，1个八度。这个八度音，男中音可以发出。单簧管、巴松管、长笛也拥有这个频段的声音。

5）中频。315～2500Hz范围，3个八度。这是人耳最容易接受的声音频段。我们从电话听筒里听到的声音一般就属于这个频段。如果没有低频和高频，单独听这个频段，是很干涩的。

6）中高频。2500～5000Hz范围，1个八度。人耳对这段音程是最敏感的。声音的清晰度和透明度都是通过这个频段来决定的。音乐的音量也主要由这个频段影响，人声的泛音也会在这个频段出现。如公共广播用的喇叭就是专门设计成3000Hz左右的频段。

7）高频。5000～10000Hz范围，1个八度。这个频段会使音乐更明亮。多种高音乐器都拥有这个频段的声音，人的唇齿音也在这个频段内。

8）超高频。10000～20000Hz范围，1个八度。这是可听频率范围内最高的音程，需要很高的泛音才可以达到这个范围，在音乐中很少见，而且人耳对这个频段很难辨别。但是，这个频段丰富的泛音可以作用于其他频段的声音，对音色有很大的影响。

了解了声音的分布之后，我们可以使用最简单的Audacity工具打开一段声音，在菜单中的特效选项下选择均衡（Equalizer），可以看到均衡器的调节菜单，如图8-20所示。

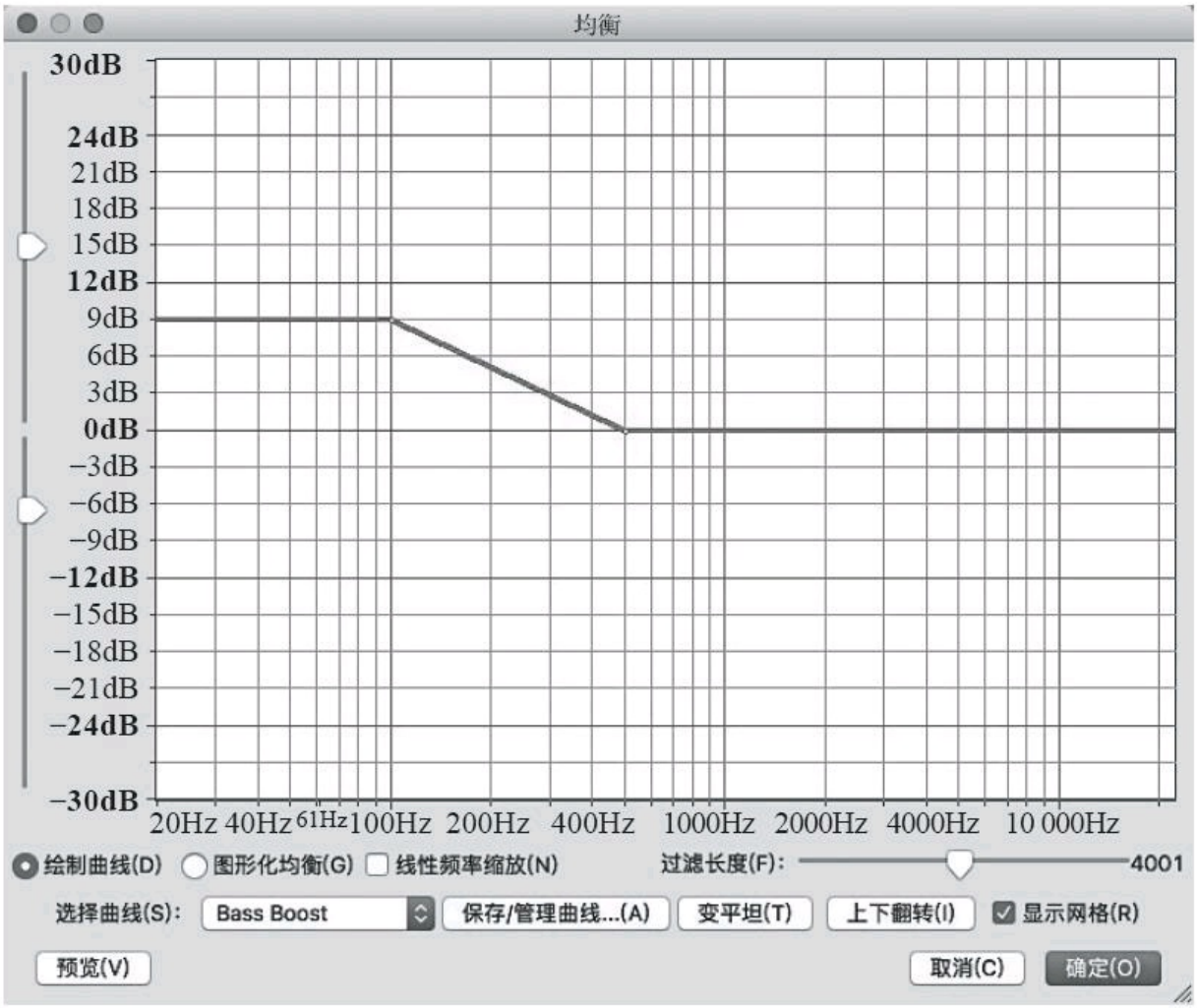


图　8-20

在图8-20中，中间部分有一个曲线，横轴是频率，纵轴是dB（0dB以上代表增强，0dB以下代表减弱），如果我们点击变平坦按钮，会看到这个曲线是在0dB上的一条水平的直线；如果我们打开选择曲线的菜单，则会看到一些默认的曲线，其中包括以下曲线。

·Bass Boost：低音增强；

·Bass Cut：低音截断（类似于高通滤波器）；

·Treble Boost：高音增强；

·Treble Cut：高音截断（类似于低通滤波器）；

·Telephone：代表电话音质（频率分布在400～3000Hz范围）；

·AM Radio：代表收音机音质（频率分布在50～400Hz范围）。

选择其中一个预制的效果器可以看到曲线的变化，点击预览按钮可以对加入这个效果器的音频效果进行预览。当然，可以拖曳曲线来对某一个频率进行增强或者减弱，然后进行预览。也可以选择图形化的均衡单选框，如图8-21所示。

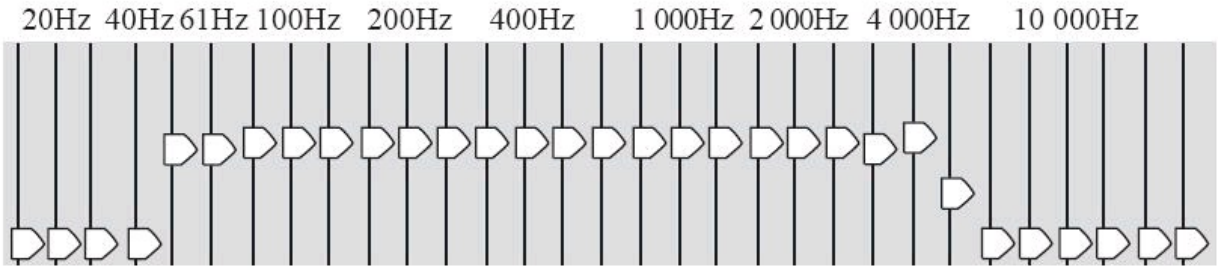


图　8-21

在图8-21中出现了各种频率上的滑动块，可以通过滑动块来将这个频率的声音增强或减弱。同时从图8-21所示的曲线上还可以看到，调整完毕之后点击预览按钮可以试听效果，如果最终确定了所有参数，点击确定按钮，就可以将这一组均衡效果器作用到声音上试听整个声音的效果。

上面描述了Audacity工具如何使用均衡器来修正声音，接下来重点分析我们需要对均衡器设置哪些参数。

最直观的参数就是频率，即修正（增强或者减弱）哪一个频率附近的声音，所以第一个最主要的参数就是frequency（代表哪一个频率）。如何修正呢？可以使用参数gain（代表增益是多少），除该参数外，还有一个参数可以使用，即bandWidth（代表频宽）。均衡器修正的不是某个单一频率上的声音而是一个频段的声音。所谓频段就是以一个频率作为中心点左右扩充一定的频率，就形成了一个频段，具体这个频段有多大，可用bandWidth来表示。bandWidth常用的表示单位有两个：一个是O，即Octave，代表一个音程即一个八度。基本乐理中我们描述过，一个八度体现在频率上就是2倍的频率，如果我们定义的中心频率为2kHz，频宽为1.0（单位为Octave），增益为3dB，那么对应到图中的曲线就是从1kHz开始上升，到2kHz上升到峰值3dB，到3kHz以后就不再上

升，这完全描述了这个频段的增强。另外一个是Q，即Quality Factor（质量系数），代表一个音程调整的有效影响斜率，也就是大家常说的Q值，其实这和前面第一种表示方法想达到的效果是一样的。实际上，Q和O有一定的换算关系的，因为我们在不同的平台或者开源算法中使用EQ的时候不一定知道要填入的bandWidth单位是什么，所以我们要知道两者是如何进行换算的。若我们知道O值（有多少个八度），那么如何计算出Q值，可如式（8-1）所示。

$$Q = \frac{\sqrt{2^N}}{2^N - 1} \tag{8-1}$$

如果知道Q值，那么如何计算出O值（有多少个八度），可如式（8-2）所示。

$$N = \frac{\log y}{\log 2} \frac{\log\left(1 + \frac{1}{2Q^2}\right) + \sqrt{\frac{\left(2 + (1/Q^2)\right)^2}{4}} - 1}{\log 2} \tag{8-2}$$

$$\log 2 = 0.30103$$

从式8-1和式8-2两个公式可以知道，只要给出任意一个值，就可以计算出另外一个值。

均衡效果器的作用以及应用场景大家已基本清楚了，本节只是均衡器的一个入门，混音师真正使用均衡器的时候，很少会对某个频段上的声音进行能量增强，反而会把其他频段上的声音能量进行衰减，所以在使用的时候，并不是一味地增加能量，而要根据具体情况具体分析。下面会讨论如何在Android和iOS平台上实现均衡效果器。

8.4.2　压缩效果器

压缩效果器又称为压缩器（Compressor），是指在时域上对声音强度所进行的一个处理。压缩器也可以简单地理解为：当音频的音量剧增的时候，自动将音量调小一点。压缩器就是改变输入信号和输出信号电平大小比率的效果器，如图8-22所示。

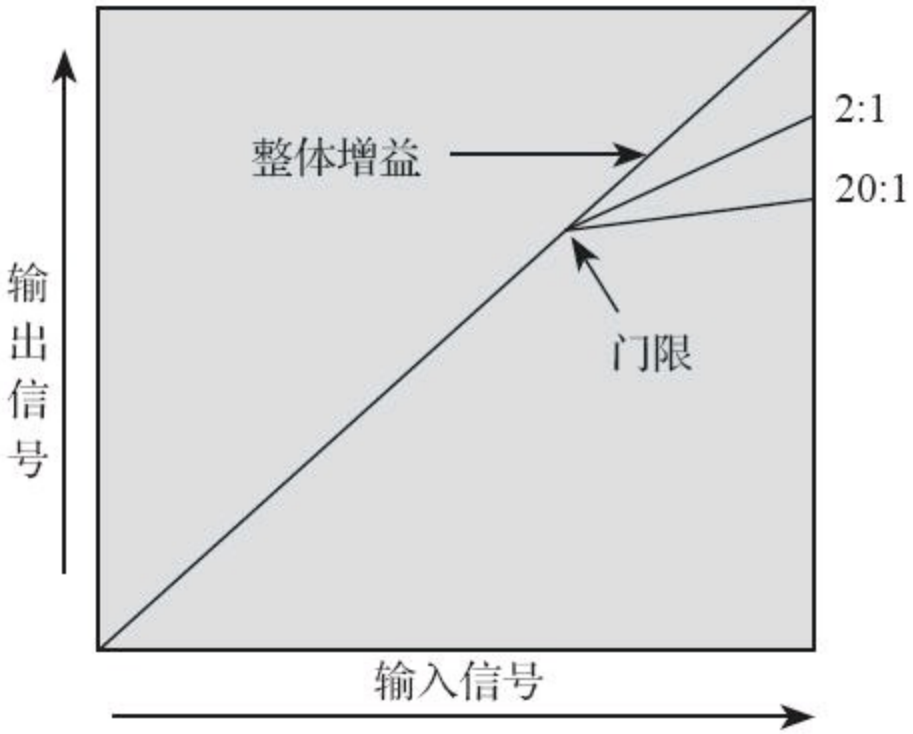


图　8-22

在图8-22中，最重要的一个概念是门限值（Threshold），即只有达到这个门限值才会进入压缩器的工作范围，整体增益（Unity Gain）就是输入信号和输出信号完全一样，也就是对输入信合不做任何改变，即压缩比为1：1。这就又提到了另外一个重要的概念，即压缩比。这个概念很简单，可以理解为图中直线的斜率。如果将压缩比调整为2：1，大于门限值的输入信号将以2：1的比例被压缩，比如2dB的输入信号在经过压缩器后就被压缩掉了1dB，这也就是图8-24中的2：1这条曲线；但如果将压缩比改为20：1的曲线，即比率设置成为20：1，这时压缩器就变成一个限制器，输入信号经过门限值之后每增加20dB的电平，输出只

能增加1dB。所谓限制器就是常说的峰值限制器（Peak Limiter）。在压缩器中还有两个非常重要的参数，一个是作用时间（Attack Time），另外一个是释放时间（Release Time）。下面分别介绍这两个参数的意义。

作用时间（Attack Time），又称起始时间，用于决定压缩器在超过门限值后多久会触发压缩器来工作。前面说过，超过门限值之后就到了压缩器的工作范围，但是不代表压缩器就会立马工作，而这里的起始时间实际上就是触发压缩器工作的时间。为什么要使用这个参数来触发压缩器工作？假设输入电平有一个瞬时峰值压缩器就进入工作，那么这就达不到压缩器的最初目的，因为压缩器的存在就是为了让整个音频作品平稳地在一定的能量范围内，所以设置一个起始时间，让压缩器躲过这些瞬时峰值而持续工作。

释放时间（Release Time）和起始时间正好相反，它决定了压缩器在低于门限信号多长时间之后停止工作，如果释放时间过短，那么在信号低于门限之后，压缩器会立即停止工作，就会导致抽泵现象，声音听起来会非常不舒服。

最后一个参数就是输出增益（Output Gain），即增益补偿，比如我们压缩了3dB的增益，然后使用增益补偿提升了整个输出信号，就可以将压缩掉的动态空间补偿回来。

明白了以上参数之后，下面使用Audacity工具打开一个音频文件，然后在特效的菜单中选择压缩器，如图8-23所示。

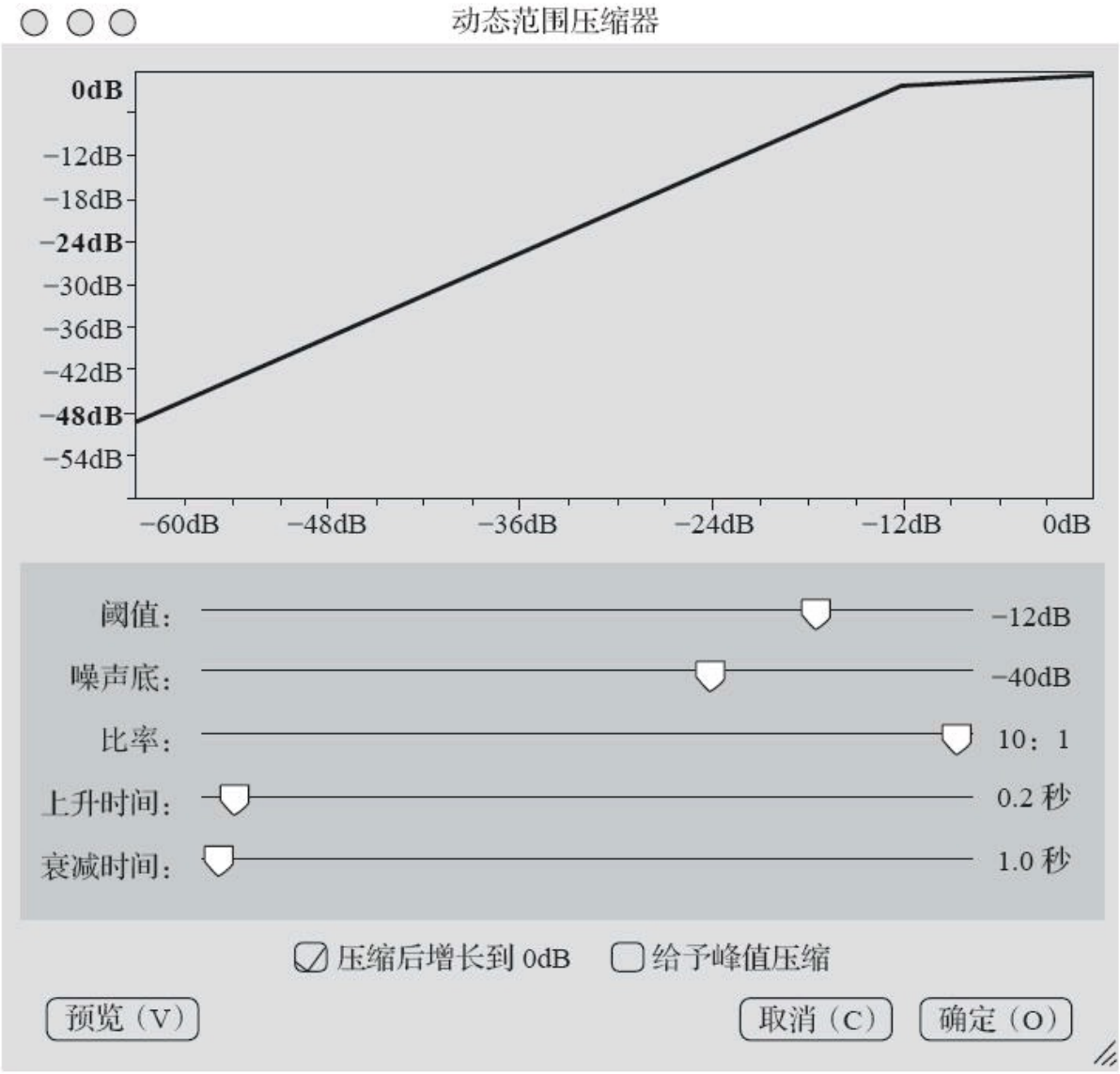


图　8-23

在图8-23中，参数噪声底可以不用去管，其余可以调整的参数包括阈值（上面所讲的门限值）、比率（压缩比）、上升时间（AttackTime）、衰减时间（releaseTime），而压缩后增长到0dB就是上面所讲的输出增益，我们可以自己调节这几个参数来观察曲线的变化，点击预览可以试听效果，如果效果满意，点击确定可以将压缩器作用到音频文件中。

在日常工作中，压缩器可以以多种其他效果器的形式出现，但是原理都是一样的，比如上面所提到的峰值限制器（Peak Limiter），将压缩

比调整到足够大（一般我们认为压缩比率在20：1以上的压缩器就是限制器）的压缩效果器就是一个峰值限制器。除此之外，唇齿音消除器（De-Esser）也是一种应用场景，因为唇齿音一般都是/ci/、/si/等高频的声音，频率范围一般为4～8kHz，做一个可调频率的压缩效果器来压缩特定的频率，可以将这些人声中的嘶嘶声衰减掉，从而达到悦耳的效果。另外还有噪声门（Nosie Gate）也是压缩器的一种应用场景，即我们规定一个门限值，门限以上的声音可以通过，门限以下的声音被视为噪音被完全切掉，这也是压缩效果器的另外一种特殊应用场景。

8.4.3　混响效果器

混响效果器又称为混响器（Reverb），但在介绍混响效果器之前先讨论一下什么是混响。大部分场景下都会产生混响，我们可以设想老师讲课的一个场景，老师的声音经过多次反射，假如有5条声音反射线（实际上有成千上万条）到达学生耳朵，老师每说1句话，学生实际上听到的就是6句话（1句话直接传到学生耳朵里，还有反射的5句话），但是由于这些反射声到达的时间间隔太近，所以学生实际上分辨不出6句话，而是1句带有混响感觉的话。混响效果器就是这样工作的，把很多路声音（由于经过不同的反射源反射，所以能量不同）进行很多次的叠加（因为反射的距离有长有短，所以到达听者耳朵的时间不同，叠加的时间也不同）。混响器就是接受一个输入的声音，然后进行某种计算，就可以达到6种声音（实际上是成千上万种声音）叠加的效果。这里所谓的某种计算在数学中叫做“卷积”计算，英文是“convolution”。如果把老师的声音看作一个单一的脉冲，通过计算之后得到一个完整的声波，如图8-24所示。

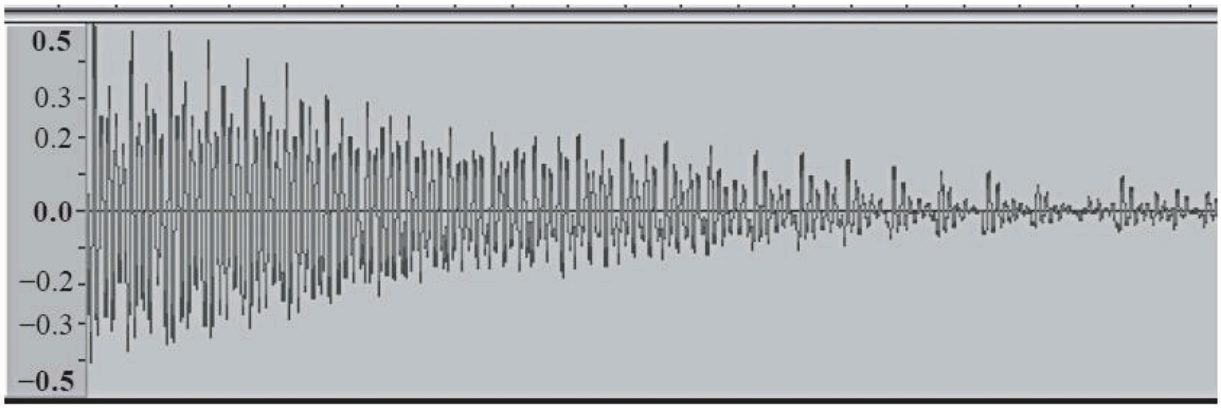


图　8-24

在图8-24所示的这个脉冲图中就含有6个脉冲（实际上是成千上万个脉冲）的声波，也就是在这个房间里，从老师到学生座位的混响特征。在声学上，由于这个混响特征是由脉冲得到的，所以称为脉冲反应，即impulse response，简称IR。很显然，在不同的空间里这个脉冲图并不相同，也就是说，不同空间里的混响特征不同，更进一步也说就是不同空间里的IR不同。我们将一个输入声音作为源声音，这个声音通过与IR卷积得到的结果就是这个声音源在这个混响空间内所产生的最终混

响结果。其实，几乎在任何场景下都会产生混响，只不过有一些混响效果在人的耳朵里不太容易分辨，像比较专业的录音棚里，墙壁上做了很多突出的吸音棉，可以最大限度地减小混响的影响，而在混音阶段再给作品增加混响。

在浴室中发出一个声音，最终我们听到的声音其实是代表浴室这个空间的混响特征。在小礼堂里，或者在一个非常开阔的大舞台上唱歌，听到的混响效果肯定不同。总之，每个空间都有自己独特的混响特征，也就是有自己独特的IR。为了模拟出各个空间的混响效果，也就是为了定制出不同的场景混响，我们可以制定不同场景下的IR，然后将声音源与特定场景下代表的IR进行卷积，这样就可以得到这个场景下的混响效果。那如何确定特定场景下的IR呢？

第一种就是采样IR混响，Sony、Yamaha都出过采样混响。采样混响全部是真实采样得来的wave文件，可以存放在任何存储器，采样混响的IR都由录音采样得来。在想要获得混响特征的地方，例如小礼堂、音乐厅舞台上安置音箱，座位席中安置立体声话筒，然后播放一系列测试信号，以脉冲信号为主，各种速度的全频段正弦波连续扫描为辅，录得声音，然后经过计算得到IR。用这种采样方法得到的IR，是最真实也是效果最好的一种，当然这种IR的制作也是极为昂贵的。

第二种就是算法混响，也是最常见的混响效果器，目前大多数数字混响效果器以及软件混响都是这种类型的。这类混响器虽然不带有真实的IR，但是提供了很多方法让你对它自带的原始脉冲序列进行修改，比如通过改变空间大小、早反射时间、衰减时间、阻尼等参数来修改IR，以达到控制混响效果的目的。为了性能的考虑，这种IR的脉冲个数是有限的，并不会像第一种采样混响中有无限的脉冲信号。

为了方便研究，声学上把混响分为几个部分，并规定了一些习惯用语。混响的第一个声音是直达声（Direct Sound），也就是源声音，在效果器里叫dry out（干声输出），随后的几个相隔比较开的声音叫“早反射声”（Early Reflected Sounds），它们都是只经过几次反射就到达的声音，声音比较大，比较明显，它们能够反映空间中的源声音、耳朵及墙壁之间的距离关系。后面的一堆连绵不绝的声音叫做reverberation。大多数混响效果器会有一些参数选项给你调节，现在就来讲讲这些参数的具体意义。

（1）空间大小（Room Size）

空间可以体现出声场的宽度和纵深度，不同的效果器在该参数上有不同的算法体现，且该参数非常重要。

（2）余响大小（Reverbrance）

如果早反射声可以决定空间的距离，那么余响可代表空间的构造，即空间里的物体多少、墙壁的材质、墙壁及室内物体的表面材质越松软，代表吸音的能力越强，余响越小。

（3）阻尼控制（Damping）

阻尼控制代表混响声音减弱的程度，对应到实际场景中就是场景里的物体多少。物体越多，且物体表面越不光滑，衰减就越厉害，可以根据我们想要得到的实际场景去设置这个参数。

（4）干湿比（Dry Wet Mix Ratio）

有的混响算法会有这个参数，干信号表示原始信号，湿信号表示混响信号，而干湿比就是代表最终输出信号的干声和湿声的比例。设置为100%，则意味着只要湿声不要干声。

（5）立体声宽度（Stereo Width）

有的混响效果器有这样的参数，如果把这个值设大，那么效果器在产生IR的时候会使左右声道差异变大，最终就会产生立体声的感觉。

和前面的操作一样，也是打开Audacity，在特效菜单中选择Reverb，如图8-25所示。

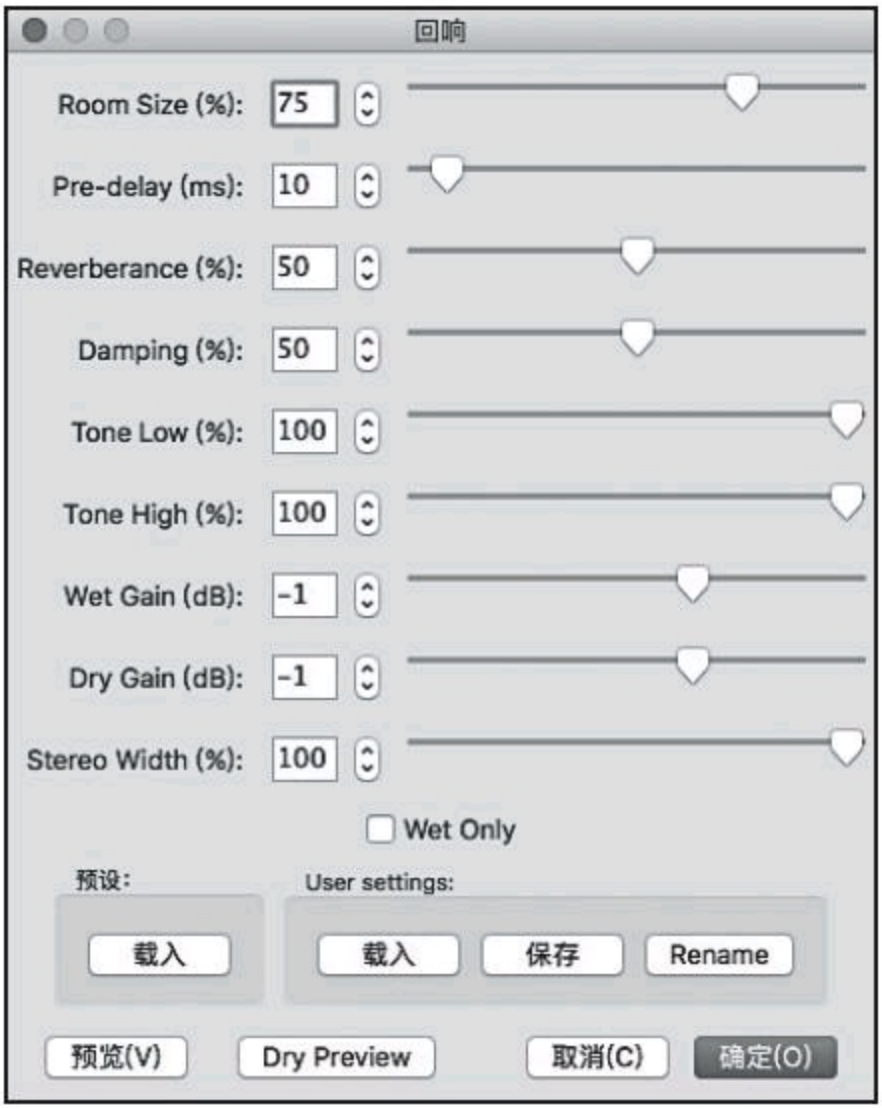


图　8-25

在图8-25中，可以看到一些可调节的参数，这些参数前面已经都一一介绍过了，大家可以自己调节参数进行预览，并且可以点击Dry Preview来预览干声，最后点击“确定”按钮，即可将这个混响效果器作用到音频文件上。

接下来讲解如何在这两个平台上实现这些效果器。

8.5 效果器实现

第8.4节已经介绍了各个混音效果器的作用，本节将讲解如何在Android平台和iOS平台上实现各效果器，并最终集成到我们的App中去。

8.5.1 Android平台实现效果器

在Android平台上实现效果器有很多种方法，如果我们从头开始一个一个去书写，显然不是一种合理的方案。我们应该寻找优秀的开源仓库实现这三种类型的效果器，比如sox开源库，它在音频处理界是一个非常优秀的框架，号称音频处理界的瑞士军刀。

1.sox编译与命令行使用

sox是最为著名的声音处理开源库，已经被广泛移植到Windows、Linux、Mac OS X等多个平台。sox项目是由Lance Norskog创立的，后被众多开发者逐步完善，现在已经能够支持很多种声音文件格式和处理声音效果。它默认支持的输入/输出是WAV文件，如果想要支持MP3等格式，需要预先安装libmp3lame库来支持这种格式的编码与解码。本节会先下载这个库的源码，并编译出二进制的命令行工具，然后再使用里面的三种效果器。sox的源码放在Source-Forge上，主页在如下的链接中：https://sourceforge.net/projects/sox/ 。

进入sox的主页之后，找到Code目录，下载整个源码目录，再使用git将整个目录clone下来。因此先建立一个sox目录，然后进入这个目录中，执行如下命令：

```
git clone https:// git.code.sf.net/p/sox/code sox-code
```

当上面这一行命令执行结束后，进入sox-code目录，可以看到仓库的源码已经全部被下载下来，接下来的工作就是将源码编译成为二进制命令行工具。先查看源码目录下的INSTALL文件，这个文件指明，如果要编译的源（即代码仓库）是使用git下载的源码，则要先执行如下命令：

```
autoreconf -i
```

这个命令执行完毕后，会在源码目录下生成configure、install-sh等文件。由于要编译最基本的sox的二进制命令行工具出来，所以应建立

一个Shell脚本config_pc.sh，键入以下代码：

```
#!/bin/bash
CWD=`pwd`
LOCAL=$CWD
./configure \
--prefix="$LOCAL/pc_lib" \
--enable-static \
--disable-shared \
--disable-openmp \
--without-libltdl \
--without-coreaudio
```

然后给config_pc.sh及configure添加执行权限，并在源码目录下面，新建pc_lib目录，最终执行这个Shell脚本文件：

```
./config_pc.sh
```

当shell脚本执行结束后，代表配置结束，接下来就可以执行安装命令了：

```
make && make install
```

执行成功后，进入pc_lib目录下，可以看到这个目录里被安装脚本生成了bin、lib、include等目录。进入bin目录，可以看到play、rec、sox等二进制文件，其中，sox就是我们要运行的二进制命令行工具，而play则可以在处理的同时直接播放一个音频文件，类似于FFmpeg中的ffplay工具。至于rec，则是录制声音的工具。由于我们在config_pc.sh中关闭了硬件设备的配置选项，所以play和record工具不能使用，只使用sox来处理音频文件，输入WAV格式的音频文件，输出WAV格式的音频文件。那我们使用sox二进制命令行工具对输入文件分别完成前面提到的三种效果器。

首先是均衡效果器。前面已介绍过均衡器的参数设置，整个参数可分为N组参数，每组参数代表对具体频率的增强或者减弱，每组参数包括频率、频带宽度和增益。sox的均衡器参数设置也一样，来看下面这条命令：

```
sox song.wav song_eq.wav equalizer 89.5 1.5q 5.8 equalizer 120 2.0q -5
```

上面这条命令的前两个参数分别代表输入文件和输出文件，它们后面有两个均衡器，第一个均衡器的中心频率为89.5Hz，频带宽度为1.5q，增加5.8dB的能量；第二个均衡器的中心频率为120Hz，频带宽度为2.0q，减少5dB的能量。如果想给声音多作用几个均衡器，在后面依次添加几组就可。待执行完命令之后，可以试听输出文件的效果，或者使用Audacity软件打开处理前和处理后的音频文件，使用频谱图观察处理前后的频谱分布的变化。

其次是压缩效果器。前面也介绍过压缩器的设置，整个参数包括门限值、压缩比、Attack Time、Decay Time等，sox中的压缩效果器使用库中的compand来实现，来看下面这条命令：

```
sox song.wav song_compressor.wav compand 0.3,1
        -100,-140,-85,-100,-70,-60,-55,-50,-40,-40,-25,-25,0,-20 0 -100dB 0.1
```

以上这条命令的前两个参数分别代表输入文件和输出文件；compand代表效果器的名称，在sox中使用compand效果器来实现压缩-扩展器（Compressor-Expander）。后面的参数以空格分开，首先是0.3和1，分别代表Attack Time和Decay Time；接下来的一组参数代表压缩器的转换函数表，每个数值的单位都是dB。继续来看0、-100、0.1这三个参数，第一个0代表增益，即压缩完毕后可以将整体增益作用到输出上，不再给任何增益了；第二个-100代表初始音量，可以设置成为-100dB，代表初始音量从一个几乎为静音的音量开始；第三个0.1代表延迟量。在实际的音频处理场景中，压缩器对于声音的忽然升高有很好的抑制作用。

现在来看由压缩器转换函数表绘制出的压缩曲线，如图8-26所示。

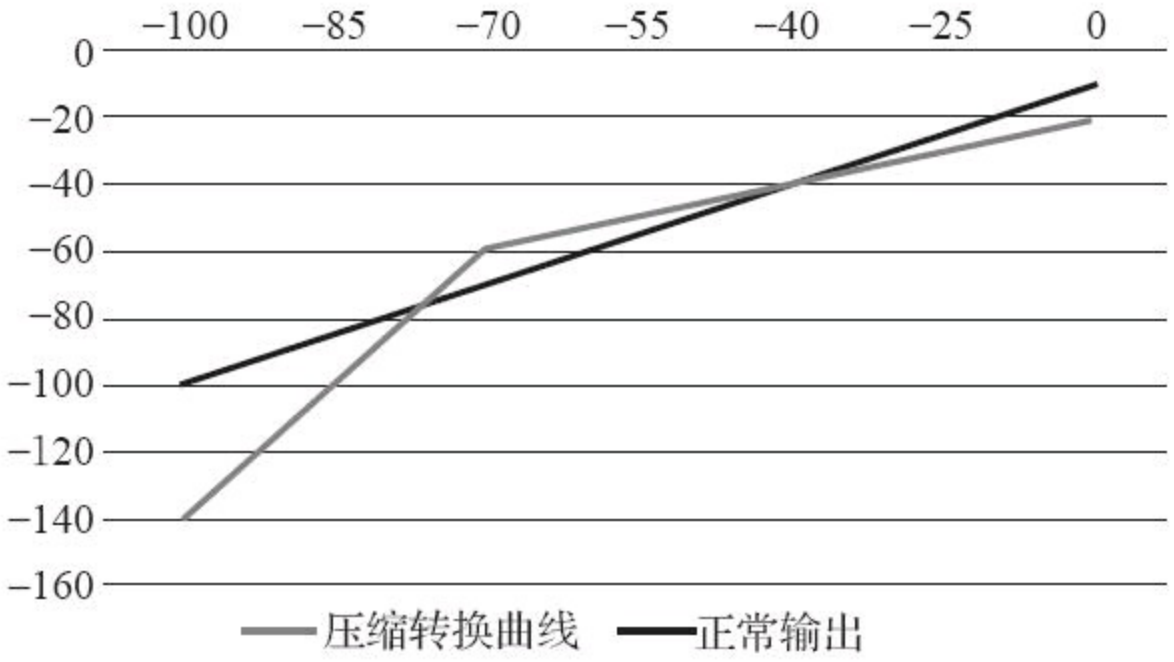


图　8-26

在图8-26（见彩插）中，红色直线为一条斜率为1的直线，实际上是不作任何处理的曲线；蓝色曲线就是我们的压缩曲线。蓝色曲线分为四部分，从中可以看到，在能量比较低的部分（-100～-80dB）可将输出能量降低，相当于将底部噪声部分压低；中间能量部分（-75～-45dB）有所提升；一部分（-40～-25dB）保持不变，即蓝色曲线和红色曲线重合；对比较高能量部分（-20～0dB）进行压缩处理，形成整个曲线。当然，输入/输出点数越多，曲线就会越平滑，得到的声音效果就会越好。大家可以试听经过压缩器处理完后的声音，是不是觉得整个音量的动态变化范围被压缩了呢？这就是压缩效果器的作用。

最后是混响效果器。混响器的参数前面已详细介绍过。下面来看如何使用sox给声音增加混响，命令如下：

```
sox song.wav song_reverb.wav reverb 50 50 90 50 30
```

其中，第一个参数是reverbrance，即余响的大小，可以设置为50试听效果；第二个参数是HF-damping，即高频阻尼，可以设置为50；第三个参数是room-scale，即房间大小，可以设置为90，代表一个比较大的房间；第四个参数代表立体声深度，设置越大，代表立体声效果越明

显，这里设置为50；最后一个参数是pre-delay，即早反射声的时间，单位为毫秒，这里设置为30ms。执行完以上命令后，读者再试听处理完的声音，会发现有比较明显的混响效果了。

这里介绍了如何编译sox，以及使用sox二进制命令行工具，下面将其交叉编译到Android平台，并且介绍其SDK的使用。

2.sox的交叉编译

这里会介绍将sox交叉编译到Android平台，并且介绍如何在Android平台使用sox的SDK来使库中的效果器工作。首先，将sox这个开源库交叉编译出一个静态库以及头文件，以方便我们在Android的NDK开发的编译阶段和链接阶段分别引用。新建立config_armv7a.sh，键入以下代码，以编译出静态库与头文件：

```
#!/bin/bash
NDK_BASE=/Users/apple/soft/android/android-ndk-r9b
NDK_SYSROOT=$NDK_BASE/platforms/android-8/arch-arm
NDK_TOOLCHAIN_BASE=$NDK_BASE/toolchains/arm-linux-androideabi-4.6/prebuilt/da
        in-x86_64
CC="$NDK_TOOLCHAIN_BASE/bin/arm-linux-androideabi-gcc --sysroot=$NDK_SYSROOT'
LD=$NDK_TOOLCHAIN_BASE/bin/arm-linux-androideabi-ld
CWD=`pwd`
PROJECT_ROOT=$CWD
./configure \
    --prefix="$PROJECT_ROOT/lib/armv7" \
    CFLAGS="-O2" \
    CC="$CC" \
    LD="$LD" \
    --target=armv7a \
    --host=arm-linux-androideabi \
    --with-sysroot="$NDK_SYSROOT" \
    --enable-static \
    --disable-shared \
    --disable-openmp \
    --without-libltdl
```

然后执行config_armv7a.sh（如果没有执行权限，则要加上执行权限），可以看到在当前目录的lib目录下有一个armv7目录，armv7目录中会有我们非常熟悉的include、lib目录，里面就有我们需要的头文件sox.h与静态库文件libsox.a，至此交叉编译工作完成。接下来看看如何使用sox库中提供的API在代码层面使用各种效果器。

3.SDK介绍

要使用sox库中提供的API，就要从它的官方实例中开始，从sox的根目录进入src目录，在src目录下有几个以example开头的C文件，这就是提供给开发者参考的Demo。打开example0.c这个文件可以看到，该文件的开头引用了sox.h头文件。然后看一下main函数，因为使用API的流程都在main函数中。在使用sox库之前，必须初始化整个库的一些全局参数，需要调用如下代码：

```
sox_init();
```

上述函数返回一个整数，如果返回的是SOX_SUCCESS这个枚举值，则代表初始化成功了。在整个应用程序中，如果没有调用sox_quit方法，那么不可以再一次调用sox_init，否则会造成Crash。接下来初始化输入文件，代码如下：

```
sox_format_t* in;
const char* input_path = "/Users/apple/input.wav";
in = sox_open_read(input_path, NULL, NULL, NULL);
```

初始化好了输入文件之后，再来初始化输出文件，代码如下：

```
sox_format_t* out;
const char* output_path = "/Users/apple/output.wav";
out = sox_open_write(output_path, &in->signal, NULL, NULL, NULL, NULL);
```

初始化好了输出文件后，就可以看到输入文件和输出文件都是WAV格式的，因为我们并没有集成其他编码格式的工具，所以是直接用的WAV格式。sox中提供的效果器种类较多，为了方便开发者使用，sox使用类似责任链设计模式的方式设计整个系统，所以使用时需要先构造一个效果器链，然后将要使用的效果器一个一个地加到这个效果链中，最终传入输入文件中的数据以及接受这个效果器链处理完的数据，就可以完成音效的处理工作。先来构造这个效果器链，代码如下：

```
sox_effects_chain_t* chain;
chain = sox_create_effects_chain(&in->encoding, &out->encoding);
```

上述代码构造出了一个效果器链，重点来看里面的两个参数，这两个参数实际上就是告诉效果器链输入音频的数据格式和输出音频的数据格式，比如声道、采样率、表示格式等。我们可以从最开始初始化的输入文件格式和输出文件格式中拿到数据格式，sox会存储到encoding属性中。接下来要向效果器链中增加效果器了，但是在增加实际的效果器之前，我们要先考虑一个问题，就是如何将输入音频数据提供给效果器链，以及如何将效果器链处理完的音频数据写入文件中。对于这个问题，sox已经帮我们提供了对应的API，为了方便开发者，sox的作者把输入和输出分别构造成了一个特殊的效果器，待我们创建出提供输入数据的效果器，就添加到效果器链的第一个位置；然后创建出输出数据的效果器，添加到效果器链的最后一个位置上。

先来看一下为效果器链提供输入数据的特殊效果器的构造，代码如下：

```
sox_effect_t* inputEffect;
inputEffect = sox_create_effect(sox_find_effect("input"));
```

上述代码构造出了一个用于给效果器链输入数据的特殊效果器，但是这个特殊效果器的数据从哪里来呢？答案就是上面初始化的输入文件，因此我们要将输入文件配置到这个效果器中，代码如下：

```
char* args[10];
args[0] = (char*) in;
sox_effect_options(inputEffect, 1, args);
```

可以看到上述代码将之前构造的输入文件格式的结构体强制转化为了char指针类型的参数，并配置给了效果器，而在sox中都是以char指针类型的参数来配置效果器的。配置好了后，要将这个效果器增加到效果器链中，并且将这个效果器释放掉，代码如下：

```
sox_add_effect(chain, inputEffect, &in->signal, &in->signal);
free(inputEffect);
```

至此给效果器链提供输入数据的特殊效果器就已经创建成功，并且进行了配置，最终成功添加到了效果器链中，这个过程也是任何一个效

果器从创建到配置到添加到销毁的整个过程。

接下来是我们想要使用的最核心的效果器部分，这里以一个非常简单的增加音量的效果器为例进行讲解，添加如下代码：

```
sox_effect_t* volEffect;
volEffect = sox_create_effect(sox_find_effect("vol"));
args[0] = "3dB";
sox_effect_options(volEffect, 1, args);
sox_add_effect(chain, volEffect, &in->signal, &in->signal);
free(volEffect);
```

从以上代码可以看到，音量效果器给整个音频文件增加了3个dB的音量。虽然整个过程比较简单，但是也有的读者可能会问，若想使用一个效果器，从哪里可以找到这个效果器的名称呢？其实所有效果器的名称都被定义在effects.h这个头文件中，读者可以自己去查阅。

接下来配置另外一个比较特殊的效果器，即接受效果器链处理完的数据，并将数据输出到文件中的效果器，代码如下：

```
sox_effect_t* outputEffect;
outputEffect = sox_create_effect(sox_find_effect("output"));
args0 = (char*)out;
sox_effect_options(outputEffect, 1, ags);
sox_add_effect(chain, outputEffect, &in->signal, &in->signal);
free(outputEffect);
```

上述代码也比较简单，主要是把前面所构造的输出文件配置给output这个特殊效果器，最终再将效果器添加到整个效果器链中。至此这个效果器链就已经构造好了，整个结构如图8-27所示。

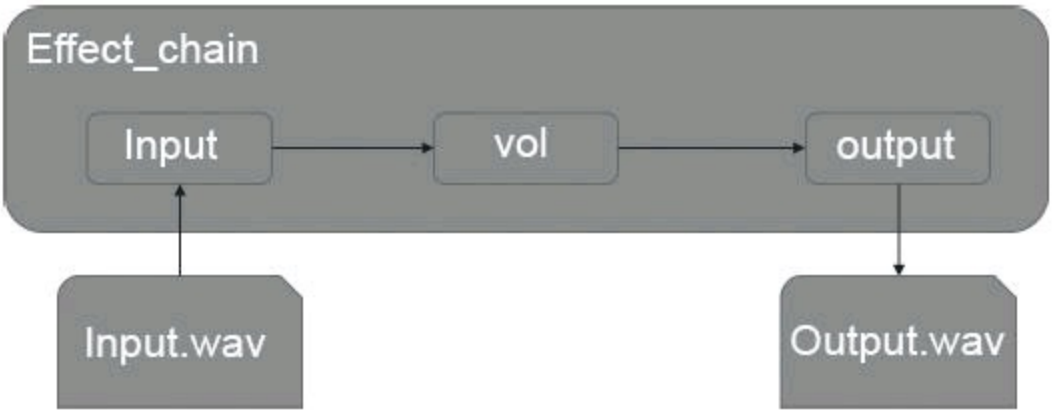


图　8-27

构造好了这个效果器链后，如何让整个效果器链运行起来呢？其实也很简单，只需要执行以下代码：

```
sox_flow_effects(chain, NULL, NULL);
```

这个方法执行结束，整个处理流程也就结束了，经过核心效果器——声音变化效果器处理之后的音频数据就被全部写入output.wav文件中了。当然，完成之后要销毁掉这个效果器链：

```
sox_delete_effects_chain(chain);
```

然后关闭输入文件和输出文件，代码如下：

```
sox_close(out);
sox_close(in);
```

最后释放sox库里的全局参数，代码如下：

```
sox_quit();
```

至此，可以使用sox提供的SDK来处理音频文件了。

4.均衡器的实现

下面介绍如何使用sox的均衡器。前面已经介绍过在命令行工具中如何使用均衡效果器，有了前面的基础，我们就可以知道如何编写出本节的代码，如下：

```
sox_effect_t* e;
e = sox_create_effect((sox_find_effect("equalizer")));
```

首先根据均衡器的名字创建出效果器，然后使用中心频率、频带宽度，以及增益来配置这个效果器，代码如下：

```
char* frequency = "300";
char* bandWidth = "1.25q";
char* gain = "3dB";
char* args[] = {frequency, bandWidth, gain};
int ret = sox_effect_options(e, 3, args);
```

由于均衡器的参数有3个，所以这里配置函数的第二个参数传递3，待执行完这个配置函数后，若返回的ret是SOX_SUCCESS，则代表配置成功。最后将这个效果器添加到效果器链中，代码如下：

```
sox_add_effect(chain, e, &in->signal, &in->signal);
free(e);
```

至此，已将一个均衡器加入效果器链中了，但是一般情况下会有多个均衡器同时作用到音频上，如果有多个均衡器，则创建多个均衡器，并依次添加到效果器链中。

一般情况下，我们在处理音频的时候，还会加上高通和低通，类似于均衡器，也属于滤波器。高通就是高频率的声音可以通过这个滤波器，低频率的声音就被过滤掉，有另外一种叫法就是低切。低通就是高通的逆过程，也称为高切。这里只展示高通滤波器，代码如下：

```
sox_effect_t* e;
e = sox_create_effect(sox_find_effect("highpass"));
char* frequency = "80";
```

```
char* width = "0.5q";
char* args[] = {frequency, width};
sox_effect_options(e, 2, args);
sox_add_effect(chain, e, &in->signal, &in->signal);
```

相比于普通的均衡器，高通滤波器不需要增益这个参数，所以只需要这两个参数就够了，低通效果器的名字为lowpass。

在sox库中，均衡器的具体实现是biquad（源码文件是biquads.c），而biquad也是大部分均衡器以及高通、低通等的实现方式。biquad又称为双二阶滤波器，双二阶滤波器是双二阶（两个极点和两个零点）的IIR滤波器，它可以不用将声音转换到频域而给声音做频域上的某些处理。至此均衡器的所有内容就讲解完毕，接下来讲解压缩效果器。

5.压缩器的实现

这里介绍如何使用sox中的压缩器。首先创建压缩效果器，代码如下：

```
sox_effect_t* e;
e = sox_create_effect(sox_find_effect("compand"));
```

其次给这个压缩器配置参数（作用时间与释放时间），代码如下：

```
char* attackRelease = "0.3,1.0";
```

然后在sox中采用更灵活的压缩曲线来控制压缩比，采用构造一个函数转换表的方式来实现，代码如下：

```
char* functionTransTable = “6:-90,-90,-70,-55,-31,-31,-21,-21,0,-20”;
```

最后是整体增益、初始化音量以及延迟时间，代码如下：

```
char* gain = "0";
char* initialVolume = "-90";
char* delay = "0.1";
```

构造完这些参数之后，利用这些参数配置这个压缩效果器，代码如下：

```
char* args[] = {attackRelease, functionTransTable, gain, initialVolume, delay
sox_effect_options(e, 5, args);
```

最终将这个效果器加入效果器链中，并销毁这个效果器，代码如下：

```
sox_add_effect(chain, e, &in->signal, &in->signal);
free(e);
```

大家尝试运行，最终拿出处理完毕的音频文件进行播放；感受在代码层使用SDK调用压缩效果器处理的音频是否和命令行工具处理出来的音频一致。

6.混响器的实现

下面介绍如何使用sox的混响器。先创建混响效果器，代码如下：

```
sox_effect_t* e;
e = sox_create_effect(sox_find_effect("reverb"));
```

其次给这个混响效果器配置参数，比如是否纯湿声，代码如下：

```
char* wetOnly = "-w";
```

然后是混响大小、高频阻尼以及房间大小，代码如下：

```
char* reverbrance = "50";
char* hfDamping = "50";
char* roomScale = "85";
```

最后是立体声深度、早反射声时间以及湿声增益，代码如下：

```
char* stereoDepth = "100";
char* preDelay = "30";
char* wetGain = "0";
```

将这些参数一块配置到效果器中，代码如下：

```
char* args[] = {wetOnly, reverrance, hfDamping, roomScale, stereoDepth, preDe
        wetGain};
sox_effect_options(e, 7, args);
```

最终将这个效果器加入效果器链中，并运行程序。大家可以试听处理完后的音频效果，其实在使用混响效果器的时候，一般在混响效果器之前增加一个echo效果器往往可以获得比较好的效果，由于篇幅的关系，这里就不再赘述了。

在sox库中，Reverb使用经典的施罗德（Schroeder）混响模型来实现，施罗德（Schroeder）混响模型使用4个并联的梳状滤波器和2个串联的全通滤波器来建立混响模型。梳状滤波器提供混响效果中延迟较长的回声，而延时较短的全通滤波器则起到增加反射声波密度的作用，如图8-28所示。

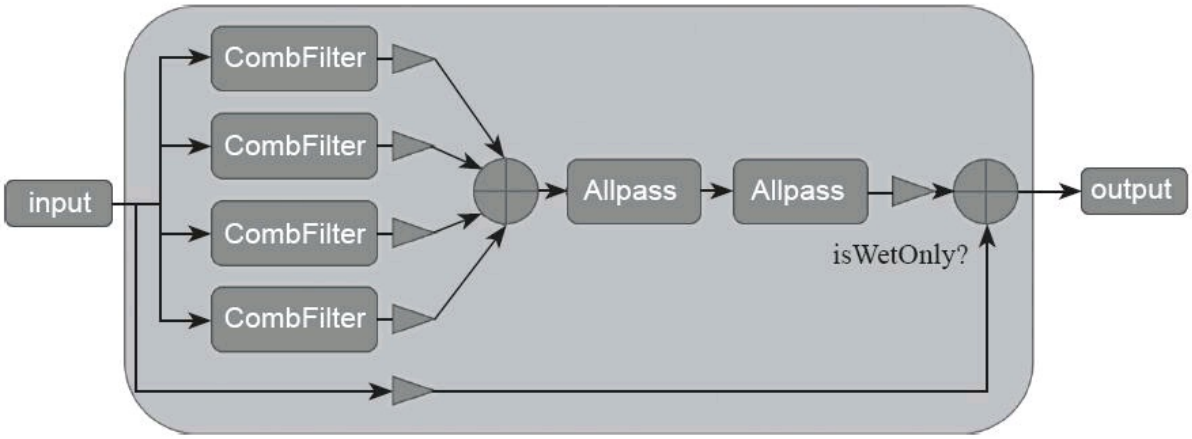


图　8-28

现在我们来分析施罗德混响模型的优缺点。优点是，通过设置6个滤波器的参数，可以模仿出前期反射和后期混响效果，全通滤波器可以在一定程度上减轻梳状滤波器引入的渲染成分；缺点是，产生的混响效果缺少早期反射声，这样会造成声音缺乏空间立体感而且不清晰。对于

缺点，我们可以进行改造和优化，其中最为常用的一种手段就是使用干声的echo来填充早期的反射声，改造后的结构如图8-29所示。

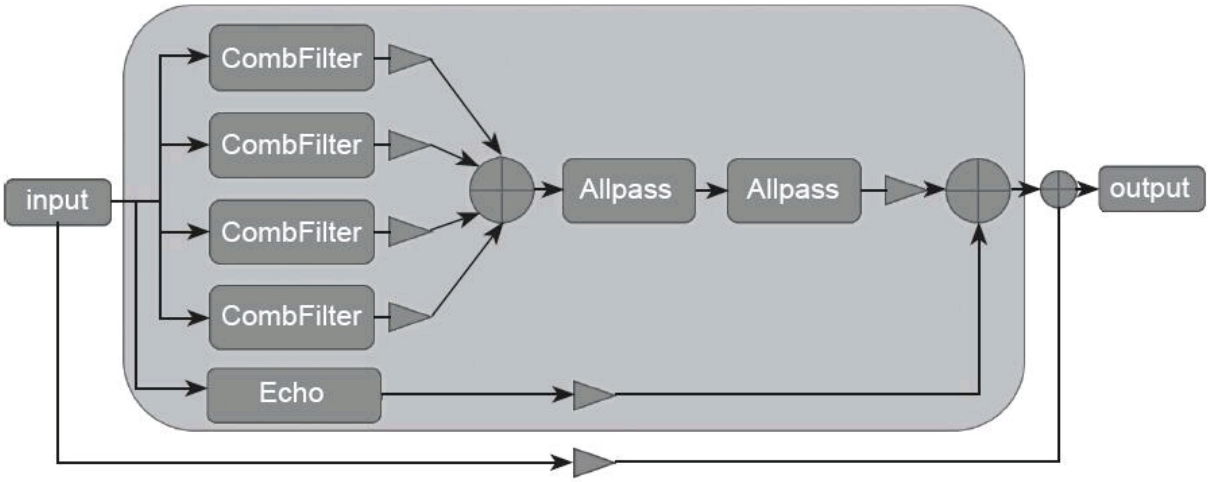


图　8-29

如图8-29所示，在混响效果器中一定要将isWetOnly属性设置为True，即只要湿声；再使用Echo来填充早起反射声部分，将湿声与Echo加起来之后作为混响的湿声部分；然后再与干声以一定的干湿比混合起来作为最终的输出结果。

至此Android平台上的效果器实现已经介绍完毕。特别说明，这些效果器都是采用C语言编写的，因此理论上可通过交叉编译运行在iOS平台上。但是，iOS平台的多媒体库非常强大，是否有更高效的处理方式呢？下面就介绍在iOS平台上如何使用更高效的方式来处理音频。

8.5.2 iOS平台实现效果器

相较于Android平台的开发者来说，iOS平台的开发者应该感到非常幸运，因为苹果公司为开发者提供了足够强大的多媒体方面的API，所以先来看看iOS平台本身的音频处理API能否满足我们实现效果器的要求。通过查阅开发者文档，可以看出AudioUnit可以用来处理音频，这个结论在第4章中有提到过。此外，第4章中还提到过AudioUnit包含多种类型，其中有一种类型是EffectType的AudioUnit。

1.均衡器的实现

下面使用AudioUnit来实现均衡器。在AudioUnit中，均衡器有两种实现类型：第一种类型称为ParametricEQ的实现，这种类型均衡器的设置类似于sox中的均衡效果器，如果想要给声音多个均衡器，则要增加多个此类型的效果器；第二种类型称为NBandEQ，从名字来看，这种均衡器可以同时对多个频带进行设置，不用为了对多频带进行调整而添加多个效果器。实际生产过程中，笔者常用的是NBandEQ，读者应该根据自己的场景来选择具体的均衡效果器。这里分别介绍这两种类型的AudioUnit的使用，且还是按照之前AUGraph的方式来使用。下面使用这两种方式实现同一个需求，给3个中心频率增加或减少能量。

（1）ParametricEQ

先来看它的描述，类型是Effect，子类型是ParametricEQ，厂商都是Apple公司，代码如下：

```
CAComponentDescription equalizer_desc(kAudioUnitType_Effect,
        kAudioUnitSubType_ParametricEQ,
        kAudioUnitManufacturer_Apple)
```

然后按照这个描述将均衡器AUNode加入AUGraph中，并且根据这个AUNode取出这个效果器对应的AudioUnit，代码如下：

```
for(int i = 0; i < 3; i++) {
    AUNode equalizerNode;
    AUGraphAddNode(playGraph, &equalizer_desc, &equalizerNode);
    equalizerNodes[i] = equalizerNode;
```

```
}
AUGraphOpen(playGraph)
for (int i = 0; i < EQUALIZER_COUNT; i++) {
    AUGraphNodeInfo(playGraph, equalizerNodes[i], NULL,
            &equalizerUnits[i]);
}
```

根据前面章节中讲述的AudioUnit配置过程来给这个效果器配置参数，代码如下：

```
for (int i = 0; i < 3; i++) {
    int frequency = [[frequencys objectAtIndex:i] integerValue];
    float band = [[bands objectAtIndex:i] floatValue];
    int gain = [[gains objectAtIndex:i] integerValue];
    AudioUnitSetParameter(equalizerUnits[i],
            kParametricEQParam_CenterFreq,
            kAudioUnitScope_Global, 0, frequency, 0);
    AudioUnitSetParameter(equalizerUnits[i],
            kParametricEQParam_Q,
            kAudioUnitScope_Global, 0, band, 0);
    AudioUnitSetParameter(equalizerUnits[i],
            kParametricEQParam_Gain,
            kAudioUnitScope_Global, 0, gain, 0);
}
```

注意，配置频带的参数不是Q值也不是以O（Octave八度）为单位的，而是以Hz为单位的，中心频率的设置以及增益的设置都是和之前一致的。配置好参数之后，就将这个效果器连接到数据源（RemoteIO或Audio File Player），然后试听效果或者将数据保存下来。

（2）NBandEQ

先来看它的描述，类型为Effect类型，子类型为NBandEQ，代码如下：

```
CAComponentDescription n_band_equalizer_desc(
        kAudioUnitType_Effect, kAudioUnitSubType_NBandEQ,
        kAudioUnitManufacturer_Apple);
```

从名字上可以看出，NBandEQ其实可以满足为多个频带增强或者减弱能量的需求，这里不再展示构造AUNode以及从具体的AUNode中获取AudioUnit的代码，而是直接把设置参数的代码展示如下：

```
for (int i = 0; i < 3; i++) {
    float frequency = [[frequencys objectAtIndex:i] floatValue];
    float band = [[bands objectAtIndex:i] floatValue];
    float gain = [[gains objectAtIndex:i] floatValue];
    AudioUnitSetParameter(nBandEqUnit, kAUNBandEQParam_FilterType + i,
            kAudioUnitScope_Global, 0, kAUNBandEQFilterType_Parametric,0);
    AudioUnitSetParameter(nBandEqUnit, kAUNBandEQParam_BypassBand + i,
            kAudioUnitScope_Global, 0, 1, 0);
    AudioUnitSetParameter(nBandEqUnit, kAUNBandEQParam_Frequency + i,
            kAudioUnitScope_Global, 0, frequency,0);
    AudioUnitSetParameter(nBandEqUnit, kAUNBandEQParam_Gain + i,
            kAudioUnitScope_Global, 0, gain, 0);
    AudioUnitSetParameter(nBandEqUnit, kAUNBandEQParam_Bandwidth + i,
            kAudioUnitScope_Global, 0, band, 0);
}
```

乍一看可能会觉得这里的参数怎么多。下面逐一解释各个参数的含义。NBandEQ表示想给哪个band设置就直接在某个参数后面加几即可，第一个参数设置是选择EQ类型，其中EQ类型包括高通、低通、带通等，这里选择Parametric类型的普通EQ；第二项参数Bypass的设置就是是否直接通过而不做任何处理，0代表不对这个频带进行处理，1代表对这个频带进行处理。剩余的三个参数前面已介绍过，但是要注意的是，BandWidth设置的单位是O（Octave，八度），因此，如果频宽单位是Q值，则要进行转换。

下面将这个效果器以AUNode的形式连接到数据源（RemoteIO或者Audio File Player）后，然后试听效果以及处理生成后的音频文件。

2.压缩器的实现

下面使用AudioUnit来实现压缩器。相比均衡器，压缩器的设置比较简单。先来看压缩器的描述，类型是Effect类型，子类型是DynamicProcessor，代码如下：

```
CAComponentDescription compressor_desc(kAudioUnitType_Effect,
        kAudioUnitSubType_DynamicsProcessor,
        kAudioUnitManufacturer_Apple);
```

其次利用这个描述构造AUNode，再找出对应的AudioUnit，代码如下：

```
AUNode compressorNode;
AudioUnit compressorUnit;
AUGraphAddNode(mGraph, &compressor_desc, &compressorNode);
AUGraphNodeInfo(mGraph, compressorNode, NULL, &compressorUnit);
```

然后给这个AudioUnit设置参数，代码如下：

```
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_Threshold,
        kAudioUnitScope_Global, 0, -20, 0);
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_HeadRoom,
        kAudioUnitScope_Global, 0, 12.937, 0);
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_ExpansionRatio,
        kAudioUnitScope_Global, 0, 1.3, 0);
AudioUnitSetParameter(compressorUnit,
        kDynamicsProcessorParam_ExpansionThreshold,
        kAudioUnitScope_Global, 0, -25, 0);
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_AttackTime,
        kAudioUnitScope_Global, 0, 0.001, 0);
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_ReleaseTime,
        kAudioUnitScope_Global, 0, 0.5, 0);
AudioUnitSetParameter(compressorUnit, kDynamicsProcessorParam_MasterGain,
        kAudioUnitScope_Global, 0, 1.83, 0);
```

这里的参数比较简单，和之前介绍的压缩器参数是一致的，主要有门限值、压缩比、作用时间和释放时间等。

最后将这个效果器连接到数据源AudioUnit的后面，再试听效果，如果满意，则可以试着生成一个目标音频文件。

3.混响器的实现

下面使用AudioUnit来实现混响器，代码如下：

```
CAComponentDescription reverb_desc(kAudioUnitType_Effect,
        kAudioUnitSubType_Reverb2, kAudioUnitManufacturer_Apple);
```

这里利用这个描述（compressor-desc）构造出AUNode，并且取出对应的AudioUnit，代码如下：

```
AUNode reverbNode;
AudioUnit reverbUnit;
AUGraphAddNode(mGraph, &reverb_desc, &reverbNode);
```

```
AUGraphNodeInfo(mGraph, reverbNode, NULL, &reverbUnit);
```

然后再设置参数，代码如下：

```
AudioUnitSetParameter(reverbUnit, kReverb2Param_DryWetMix,
        kAudioUnitScope_Global, 0, 15.65, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_Gain,
        kAudioUnitScope_Global, 0, 9.3, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_MinDelayTime,
        kAudioUnitScope_Global, 0, 0.02, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_MaxDelayTime,
        kAudioUnitScope_Global, 0, 0.25, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_DecayTimeAt0Hz,
        kAudioUnitScope_Global, 0, 1.945, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_DecayTimeAtNyquist,
        kAudioUnitScope_Global, 0, 10, 0);
AudioUnitSetParameter(reverbUnit, kReverb2Param_RandomizeReflections,
        kAudioUnitScope_Global, 0, 1, 0);
```

这里的参数设置和前面大部分的混响设置差不多，大家可以自己调试各项参数来试听效果。设置好参数后，可以将这个混响器连接到数据源之后并试听效果，如果满意，则可以试着生成一个目标音频文件。

8.6　本章小结

本章从声音的时域、频域表示开始讲解，并且讲解了FFT的物理意义，掌握这些基本的表示对于数字音频的理解是很有帮助的；然后讲解了一些基本的乐理知识，掌握这些乐理知识之后，相信读者对于声音的理解可以达到一个更高层次的理解；最后介绍了混音效果器，在8.4节和8.5节从各个效果器的原理以及实现进行了分析，并且对各自平台的优化策略也做了总结。读者可以依据自己工作中的需求逐一进行学习和应用。

第9章　视频效果器的介绍与实践

我们曾在第7章最后抛出了两个问题：第一个问题是声音方面的，即声音听起来有点干，能否修饰得更好听；第二个问题是视频方面的，即能否给图像调对比度、提亮、给皮肤磨皮等操作。其中第一个问题已在第8章解决，本章将通过给视频增加滤镜效果来解决第二个问题。

可能有读者会有疑问，视频的处理和普通的图像处理有什么区别吗？其本质是没有差别的，都是针对像素进行处理，只是视频的图像是一帧一帧的。在进行视频处理时，有几点需要注意：1）视频处理要求实时性高，因为视频的帧率最低应为15fps（1秒钟15帧）以上，所以每帧的处理速度应在66ms以下，这样才能保证视频的实时处理，否则会让视频卡顿，给用户带来不好的体验。2）视频有时间冗余性，所以在处理某些特殊场景时（如人脸特征点识别），可以利用这一特性来减少运算量。第1章我们已经介绍过视频以及图像的基础知识，下面让我们开始本章内容的学习吧！

9.1　图像处理的基本原理

数字图像处理是指利用计算机将图像的数字信号进行处理的过程。图像处理最早出现于20世纪50年代，当时的电子计算机已经发展到一定水平，人们开始利用计算机来处理图形和图像信息。数字图像处理作为一门学科大约形成于20世纪60年代初。早期图像处理的目的是改善图像的质量，它以人为对象，以改善人的视觉效果为目的。

下面先来看一下图像的基本属性。首先是亮度，也称灰度，它是大家常说的YUV格式的Y分量，如果使用RGB表示图像，那么可采用前面章节中提到的公式转换出亮度信息；其次是对比度（contrast），即画面黑与白的比值，也就是从黑到白的渐变层次，比值越大，说明从黑到白的渐变层次越多，色彩表现越丰富；最后是饱和度（saturation），是指色彩的鲜艳程度，也称为色彩的纯度。了解了图像的这几个基本特征后，下面看看如何改变一张图像里的这些特征，以及改变这些特征会对整张图像有什么影响。

9.1.1 亮度调节

亮度调节的实现有两种方法：一种方法是非线性亮度调节，另外一种方法是线性亮度调节，下面逐一进行介绍。

首先是非线性亮度调节。它的实现非常简单，即对于图像的RGB通道，每个通道增加相同的增量。其伪代码如下：

```
byte* image = loadImage();
byte* r,g,b = interlaceImage(image);
int brightness = 3;
r += brightness;
g += brightness;
b += brightness;
```

上述代码的实现非常简单：第一步调用loadImage方法将一张图片加载到内存；第二步将RGB通道分离开后，再对这三个通道分别增加相应的亮度值。这种亮度调节方法的优点是，代码简单，亮度调整速度快；缺点是图像信息损失比较大，调整过的图像平淡，无层次感。

其次是线性亮度调节，在介绍这种调节方式之前，先介绍HSL色彩模式。HSL是工业界的一种颜色标准，代表色相（Hue）、饱和度（Saturation）、明度（Lightness）三个通道的颜色，每个通道都可使用0～255的数值来表示。这种调节是通过对色相、饱和度、明度三个颜色通道的变化及其相互之间的叠加来得到各种颜色。线性亮度调节就是先将RGB表示的图像转换为HSL的颜色空间，然后对L通道进行调节，得到新的L值，再与HS通道合并为新的HSL，最终转换为RGB得到新的图像。下面用伪代码来实现上述的过程，第一步先用RGB计算出L值：

```
L = (max(r, max(g, b)) + min(r, min(g, b))) / 2;
```

L的取值范围是[0，255]，然后利用L值与RGB分别求出HS部分的值：

```
if(L > 128) {
    rHS = (r * 128 – (L - 128) * 256) / (256 - L);
```

```
        gHS = (g * 128 – (L - 128) * 256) / (256 - L);
        bHS = (b * 128 – (L - 128) * 256) / (256 - L);
    } else {
        rHS = r * 128 / L;
        gHS = g * 128 / L;
        bHS = b * 128 / L;
    }
```

再调整L值的亮度得到新的L值，并用新的L值和上面计算出的HS的值求出新的RGB，代码如下：

```
    int delta = 20;// [0-255]
    newL = L + delta – 128;
    if(newL > 0) {
        newR = rHS + (256 - rHS) * newL / 128;
        newG = gHS + (256 - gHS) * newL / 128;
        newB = bHS + (256 - bHS) * newL / 128;
    } else {
        newR = rHS + rHS * newL / 128;
        newG = gHS + gHS * newL / 128;
        newB = bHS + bHS * newL / 128;
    }
```

得到新的RGB像素点就是调节亮度之后的像素点。综上所述，线性亮度调节的优点是调节过的图像层次感很强；缺点是代码复杂，调节速度慢，而且当亮度增减量较大时图像有很大失真。

9.1.2　对比度调节

对比度调节要针对RGB三个通道同时调整，而不能对三个通道分别调整，因为分别调整会造成色偏的问题。对于这三个通道的调整，可以用一条函数曲线来将原始的RGB作为输入，对应这条函数曲线的输出就是调整后的结果。

设置对比度的函数如下：

```
y = (x – 0.5) * contrast + 0.5;
```

这条函数曲线的y代表输出，x代表输入，这条曲线会同时作用到RGB三个通道。如果contrast值为1，则输出等于输入，即曲线为一条斜率为1的直线。如果想增加对比度，即扩大整幅图像所占色彩的表现程度，则要将contrast设置为大于1的数值，例如要设置这个系数为1.2，曲线图如图9-1所示。

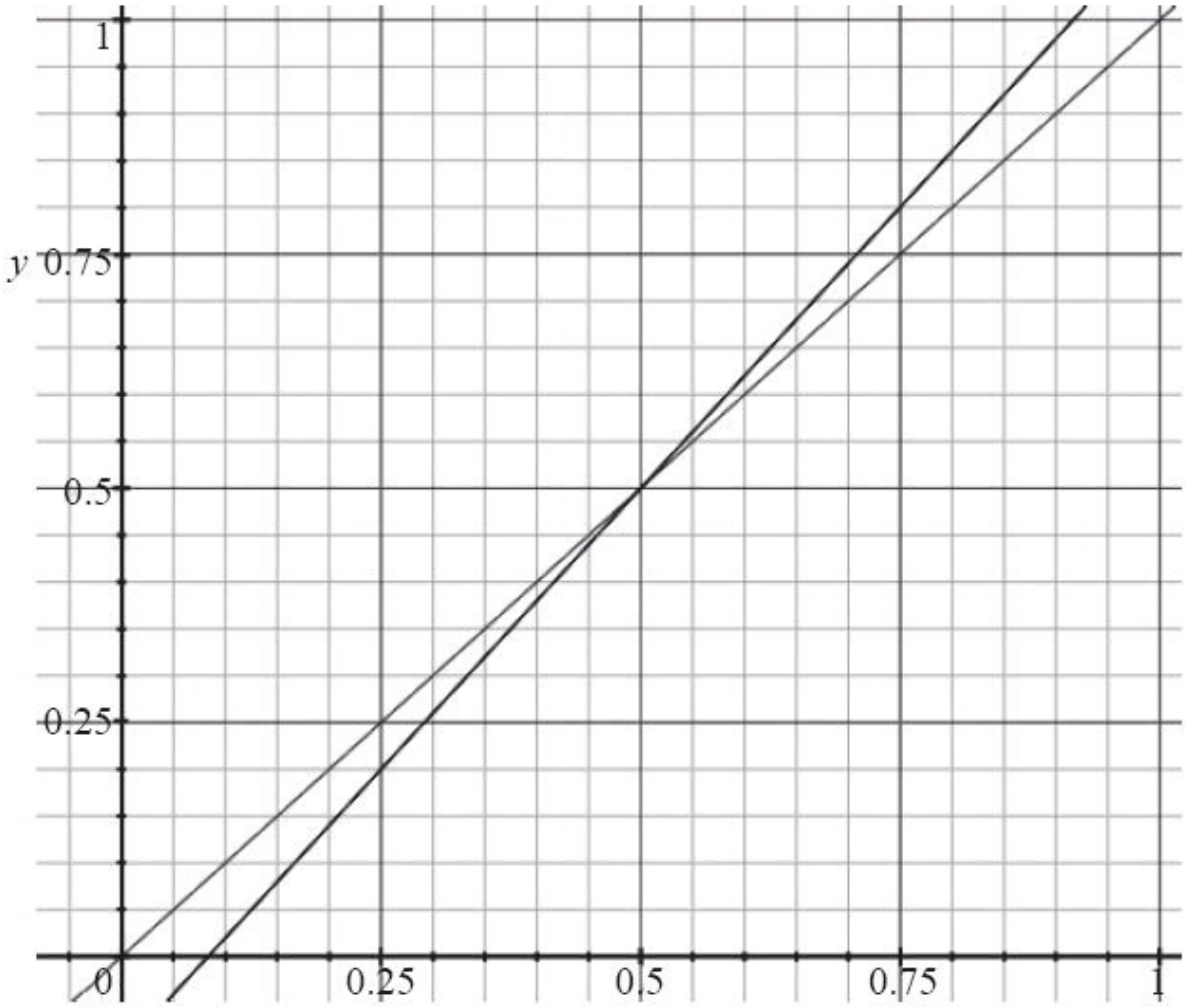


图　9-1

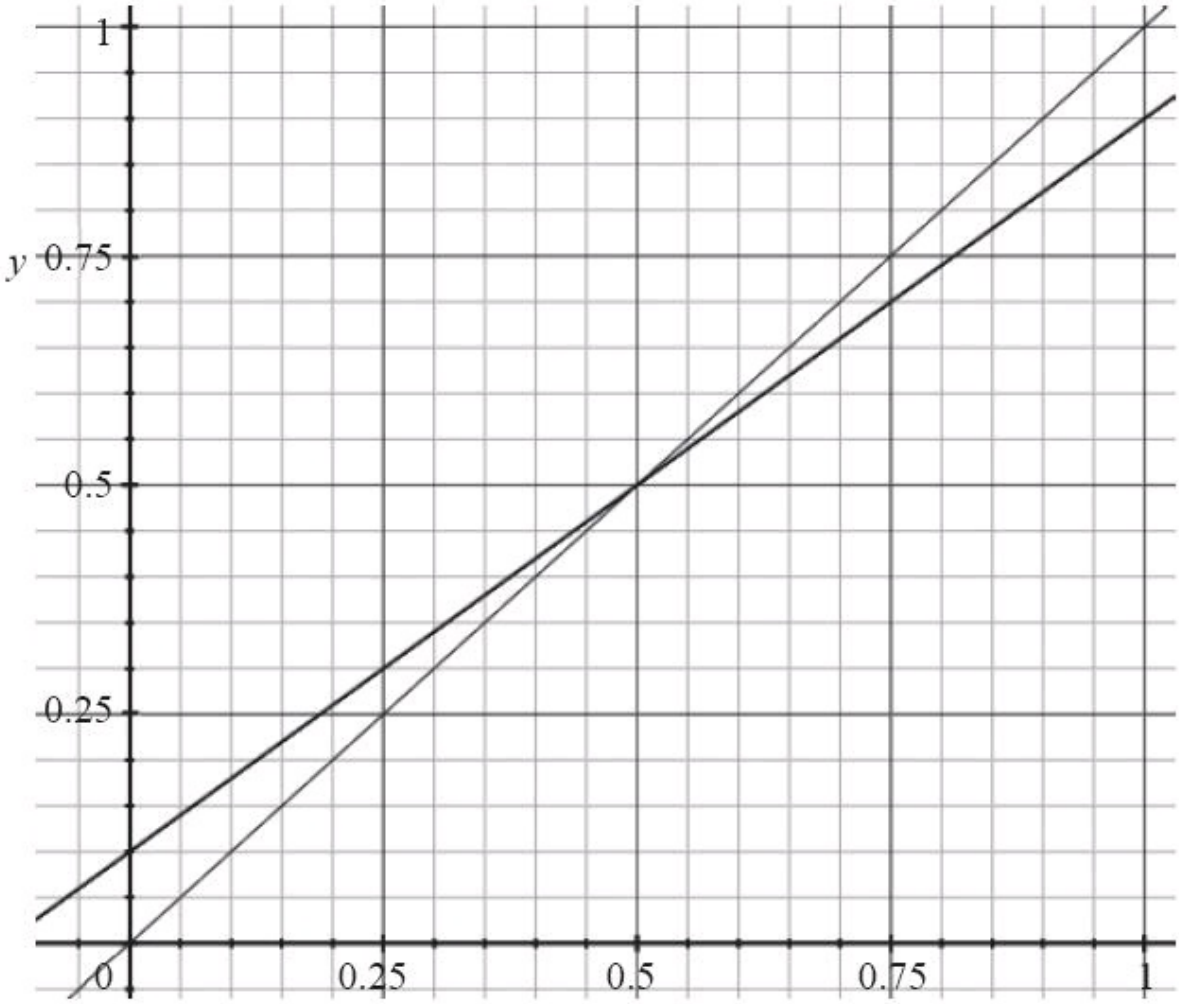


图　9-2

为了方便对比，图9-1中浅色的直线斜率为1，代表输出和输入是一样的，深色的直线代表加大对比度的函数曲线，大家可以从x轴的0.5这个点看起，左边深颜色曲线下降得更快一些，右边深颜色的曲线上升得更快一些，最终会导致整幅图像所表示的色彩更加丰富。从图9-1可以看到，这张图片从原来的0.25～0.75这段表示颜色的范围（即0.5）的输入最终扩展成为0.2～0.8这段表示颜色的范围（即0.6），也就是说，扩大了颜色表示范围。如果选取contrast的值为0.8，则代表缩小了色彩的表示范围。如图9-2所示，同样看横轴，输入范围从0.25～0.75就被缩小为0.3～0.7了，也就是颜色范围为0.5最终缩小到了0.4。大家可以通过一张图片去查看这种函数曲线的处理效果。

9.1.3 饱和度调节

图像的饱和度调节有很多种方法，最简单的方法就是判断每个像素的R、G、B值是否大于或小于128，若大于，则加上调节值，小于则减去调节值；也可将像素RGB转换为HSV，然后调整其S部分，从而达到线性调节图像饱和度的目的。这里介绍第一种比较简单的方法，首先通过RGB计算出本像素点的亮度值，计算公式如下：

```
luminance = 0.2125 * R + 0.7154 * G + 0.0721 * B;
```

然后设计一个可以调节饱和度大小的参数saturation，取值范围为[0.0，2.0]，默认值为1.0，代表输出图像就是输入图像，像素不做任何变化。如果取值为0.0，则代表为灰度图；若取值为2.0，则代表饱和度最大，公式如下：

```
output = (1.0 – saturation) * vec3(luminance) + saturation * vec3(R, G, B);
```

因为饱和度代表了色彩的纯度，所以增加饱和度时，首先要对RGB都乘以大于1.0的系数，然后减去一个亮度值，防止其亮度不一致。反之，降低饱和度原理也是一样的。

9.2　图像处理进阶

9.1节讲解了图像的基本特征，以及如何调节这些基本的特征来达到我们想要的效果。但是，仅仅凭借这些基本特征的修改还是很难达到实际生产环境的要求，因此本节给大家介绍一些更加复杂的图像处理。线性滤波可以说是图像处理中的一种常用方法，它允许我们对图像进行处理，产生多种不同的效果。做法很简单，假设有一个二维的滤波器矩阵（有一个高大上的名字叫卷积核）和一个要处理的二维图像，那么，对于图像的每个像素点，先计算它的邻域像素和滤波器矩阵的对应元素的乘积，然后加起来，以此作为该像素位置的值，即完成整个滤波过程。

9.2.1 图像的卷积过程

对图像和滤波矩阵进行逐个元素相乘再求和的操作，相当于将一个二维的函数移动到另一个二维函数的所有位置，这个操作就叫卷积。虽然卷积是图像最基本的操作，但却非常有用，这个操作有两个基本特点：一是这个操作是线性的，也就是我们用每个像素与其邻域的线性组合来代替这个像素；二是具有平移不变性，是指我们在图像的每个位置都执行相同的操作。正是因为具有这两个特性，后期才可以将图像处理的算法迁移到显卡（GPU）中去执行，这在后期做算法优化的时候会有更加深刻的认识。

对于平面图像的卷积，一般称为2D卷积，因为需要多个嵌套的循环，所以执行速度比较慢，除非我们使用比较小的卷积核。所谓卷积核，即选择这个像素点周围领域的多少，一般选用3×3或者5×5的卷积核。3×3的卷积核是由中间像素点和周围包围着它的8个像素点共同组成一个3×3的矩阵。在移动端，性能肯定是最重要的，所以我们一般选择上下左右相邻的四个领域像素点来组成卷积核来做卷积操作。但不论是何种卷积核，都会要求所有的像素点加起来都是奇数，这样才会以本像素作为中心点，然后以领域像素点（偶数）来和本像素点做线性叠加运算。规定了卷积核之后，就可以定义自己的滤波器矩阵了，滤波器矩阵一般有如下要求：

·滤波器矩阵的元素数目一般为奇数，这样才会有一个中心。

·滤波器矩阵所有的元素之和应该等于1，这是为了保持亮度不变。

·为了防止过载，滤波后的像素点的值一定要维持在0～255之间。

下面先定义一个不会做出任何影响的卷积操作的卷积矩阵，代码如下：

```
int matrix[3][3] = {
    0,0,0,
    0,1,0,
    0,0,0
}
```

然后拿一幅图片的所有像素点及其领域像素点来做卷积操作，你会看到，我们定义的卷积矩阵中心像素点的权重为1，领域像素点的权重为0。因此，叠加之后的像素点结果就是这个像素点的原始值。读者可以自己理解整个卷积过程。

9.2.2 锐化效果器

图像的锐化就是补偿图像的轮廓，增强图像的边缘及灰度跳变的部分，使图像变得更加清晰。在一般的磨皮效果器或者视频解码之后，图像往往会变得比较平滑。从频域的角度来考虑，图像模糊的实质是因为其高频部分的能量被衰减，而用户直接看到的现象就是图像中的边界（edge）、轮廓变得模糊，为了降低这种不利的效果，通常要使用扩展对比度效果器、去块滤波器，此外，还会使用到锐化效果器。

实现锐化效果器的方法很简单，也是通过图像卷积来完成的，因为锐化效果器主要就是为了增加图像细节部分的权重。定义如下卷积矩阵：

```
int matrix[3][3] = {
     0,-1, 0,
    -1, 5,-1,
     0,-1, 0
}
```

上面这个矩阵实际上是将要处理的像素点与其上下左右四个领域像素点按照矩阵描述来做线性叠加，从矩阵中对像素点的权重分配可以看出，这个卷积矩阵实际上就是计算当前点和领域像素点的Diff值，然后将所有的Diff值再加到这个像素点上。因此，当当前像素点和领域像素点差别不大的时候，卷积之后的结果是不变的；而当它正好是一个边缘或者细节点的时候，就会更加突出这个边缘或细节。上面的卷积操作实际上只做了5个像素点，如果卷积核确实要做3×3的话（即9个像素点），那么卷积矩阵如下：

```
int matrix[3][3] = {
    -1,-1,-1,
    -1, 9,-1,
    -1,-1,-1
}
```

如果我们想做一个可以动态调节锐化程度的矩阵，那么可以设定矩阵如下：

```
int matrix[3][3] = {
    0,       -k,        0,
    -k,      4k + 1,    -k,
    0,       -k,        0
}
```

其中，k的取值范围为[0.0，2.0]，如果取值为0.0，则代表什么都不做，输出图像和输入图像是一样的；如果取值为2.0，则代表锐化程度达到最大。这里只需要理解原理，后面会给出具体的实现代码。

边缘检测算法与锐化的类似，本质上也是做矩阵的卷积运算，只是矩阵所有的元素之和都是0，因为我们的目的是得到边缘信息，所以只需要以灰度图作为输入就可，并且在计算像素点的时候只需要取出一个子像素点就可（因为灰度图的r、g、b的值都一样），边缘检测算法的矩阵如下：

```
int horizontalMatrix[3][3] = {
    -1,-1,-1,
     0, 0, 0,
     1, 1, 1
}
```

这个矩阵可以检测出所有横向的边缘，同理，可以利用下面矩阵找出所有纵向的边缘：

```
int verticalMatrix[3][3] = {
    -1, 0, 1,
    -1, 0, 1,
    -1, 0, 1
}
```

找出这两个边缘点之后，可以做一个平方和来代替当前点的亮度值。大家可以设想一下，如果当前像素点的值和领域的8个像素点的值是一样的，那么出来的值必为0，也就是黑色的，但是，一旦有差别，就会是一个大于0的亮度值，差别越大，亮度值也越大。

9.2.3 高斯模糊算法

其实模糊滤波器就是对周围像素进行加权平均处理，对于均值模糊算法来讲，周围所有邻域像素点的权值都相同，所以不是很平滑，会显得糊糊的一片。高斯模糊就是用来解决这个问题的，它会把图像的模糊处理得很平滑，正因为这个优点，所以被广泛用在图像降噪上，特别是在边缘检测之前用来去除视频帧的噪点。下面先看高斯模糊的权值是如何分配的。

高斯模糊的权重是正态分布的权重，正态分布是一种可取的权重分配模式。正态分布在图形上表示为一种钟形曲线，越接近中心，取值越大，越远离中心，取值越小。大多数统计表明，生活中的很多特征都呈正态分布，包括人类的智力、身高、考试成绩等。图像处理也一样，只需要将中心像素点作为原点，以领域像素点距中心像素点的远近分配合适的高斯权重，就可以得到一个高斯加权平均值。

在图像处理领域，需要使用二维的高斯分布函数来实现高斯模糊的算法，二维高斯函数如下所示。

$$f(x,y)=\left(2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}\right)^{-1}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2}-\frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2}+\frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right] \quad (9\text{-}1)$$

其中，μ_1、μ_2、δ_1、δ_2 和ρ都是常数，我们称（x，y）服从参数为 μ_1、μ_2、δ_1、δ_2 和ρ的二维正态分布。如果以权重大小作为纵坐标，与二维图像就形成一个三维空间，这个函数在三维空间中的图像就是一个椭圆切面的钟倒扣在O（x，y）平面上，如图9-3所示，其中心在（μ_1，μ_2）点。

根据式（9-1），为了实现矩阵卷积滤波，给出一个5×5的二维矩阵，如下：

```
int martrix[5][5] = {
    1, 4, 7, 4,1,
    4,16,26,16,4,
    7,26,41,26,7,
    4,16,26,16,4,
    1, 4, 7, 4,1,
```

```
};
```

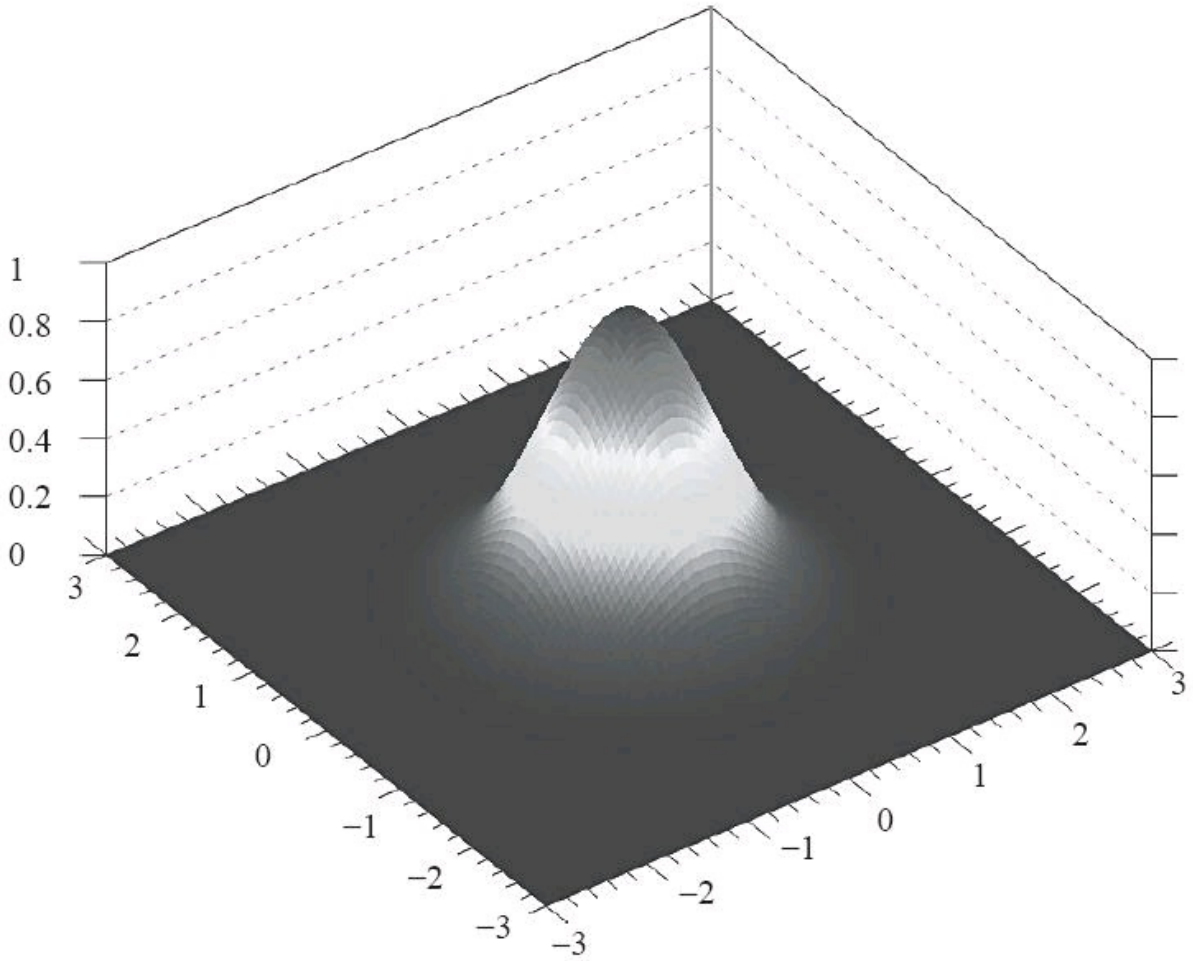


图　9-3

像素点与领域像素点对矩阵进行卷积之后，将其结果除以273（所有权重值之和），可得到亮度相同的像素点替换原始像素点。这里仅需理解原理，后面会给出具体的实现代码。

9.2.4　双边滤波算法

双边滤波（bilateral filter）是一种可以降噪保边的滤波器。之所以可以达到此效果，是因为滤波器是由两个因素共同影响的：一个是由几何空间距离决定滤波器系数；另一个是由像素差值决定滤波器系数。几何空间距离类似于高斯模糊算法，由距离中心像素点的远近来确定权重值，但是这个权重值到底能不能起作用还得看第二个因素，即像素差值，如果像素差值过大，那么就有可能不让参与最终的权重计算，以达到保边的效果；如果差值不太大，就可以达到降噪的效果。双边滤波在图像处理领域中有着广泛的应用，比如在磨皮、去噪点等场景下都有应用。

我们可以通过一张图片来了解整个双边滤波的过程，如图9-4所示。

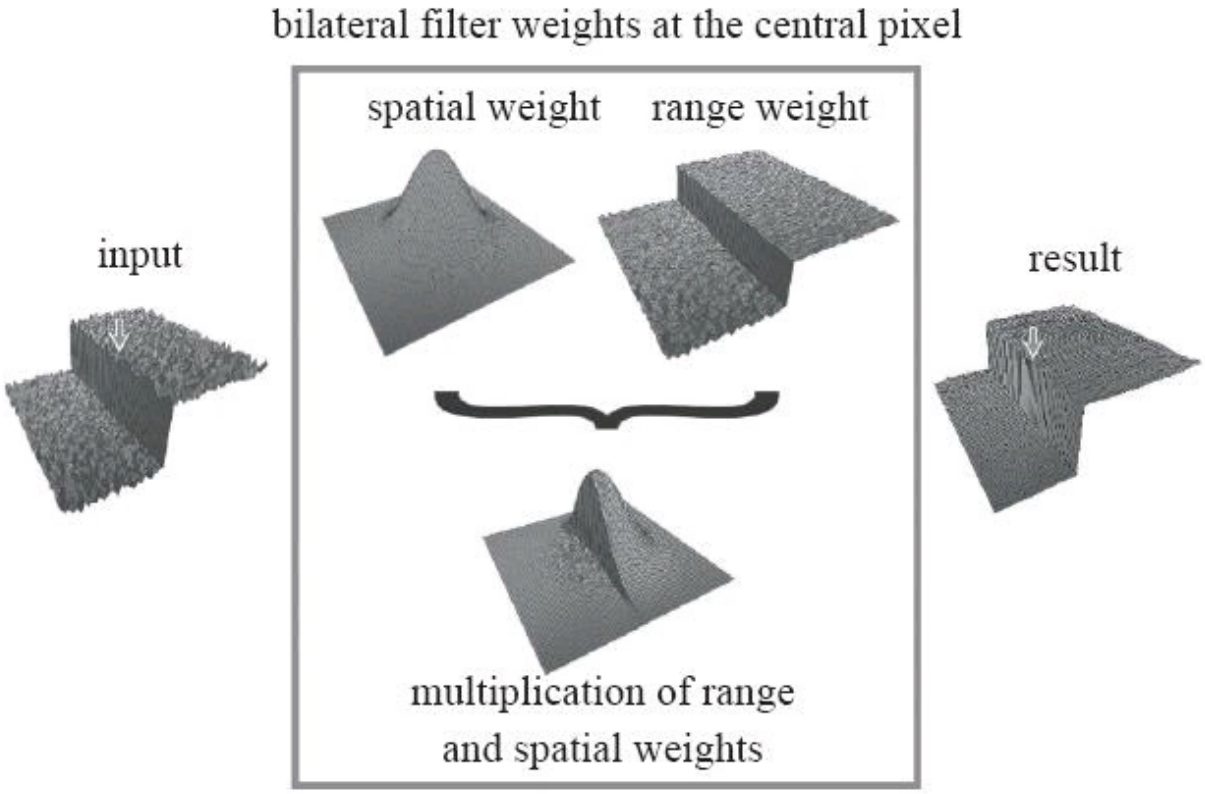


图　9-4

首先，假设左边的input所代表的是图片，其中箭头指向的边就是用来确定双边滤波权重的过程；右边的result所代表的就是图片最终经过双边滤波器处理之后的结果，其中箭头指向的位置的边虽然被保留了下来，但是在两个平面上的毛刺被平滑掉了，这也就是我们希望达到的两个目的——保边和降噪。

接下来分析这个权重具体是如何确定的。先是左边的空间权重（spatial weight）部分，这个权重就是一个高斯权重的分布；再看右边的像素差值权重（range weight）部分，这里的权重是根据像素点差异大小计算出来的权重大小，像素点差异越大，权重越小，所以这里在边缘部分的权重非常小；然后将两者相乘得到我们的最终权重信息；最后将该权重信息和领域像素点做加权平均，从而替换当前像素点，对这幅图片的所有像素点做相同的操作，最终得到结果图片。

9.2.5 图层混合介绍

图层混合是指两个图层需要混合在一起，可通过多种模式来实现这种混合。在Photoshop中有27种以上的混合模式，我们可以自己实现各种混合模式，本节挑选几个比较常见的图层混合模式进行讲解。不过，首先要确定几个术语，然后再分别来讲解几种常见的混合模式。

所有的图层混合都可以看成是两个图层的混合（即便是N个图层，也可以一一混合之后再与后面的图层进行混合）。我们规定：A代表上面图层的色彩值，B代表下面图层的色彩值，C代表混合之后的色彩值。所有色彩值的表示类型为浮点类型，取值范围是[0.0，1.0]。

1.正片叠底混合模式

将两个颜色的像素值相乘，得到的结果就是最终色的像素值。通常来说，执行正片叠底混合之后的颜色比原来两种颜色都深。任何颜色与黑色正片叠底混合之后得到的仍然是黑色，任何颜色与白色正片叠底混合之后仍保持原来的颜色不变；而与其他颜色执行正片叠底混合模式之后，会产生暗室中以该种颜色照明的效果。公式如下：

```
C = A * B;
```

这种混合模式常用于将上层图片的白色部分透过去，从而显示出下面图片区域的全部颜色，其他颜色加深的场景。因此有一个易于记忆的口诀：谁黑听谁的，白色彻底无视。

2.滤色混合模式

与正片叠底混合模式刚好相反，滤色混合是将两个颜色的互补色的像素值相乘，得到最终颜色的像素值。通常来说，执行滤色混合模式后的颜色都较浅。任何颜色与黑色执行混合滤色之后，原色不受影响；任何颜色与白色执行滤色混合之后得到的是白色；与其他颜色执行滤色混合后会产生漂白的效果。公式如下：

```
C = 1 – (1 - A) * (1 - B);
```

滤色混合模式使用的场景，比如有一张逆光的图片，看着比较黑暗，我们先复制一份，然后将复制的这张照片与原始图像进行滤色混合，就可以得到一种不错的效果。因此有一个易于记忆的口诀：谁白听谁的，黑色彻底无视。

3.叠加混合模式

在保留底色明暗变化的基础上使用“正片叠底”或“滤色”混合模式，虽然绘图的颜色被叠加到底色上，但会保留底色的高光和阴影部分。底色的颜色没有被取代，而是与绘图色混合来体现原图的亮部和暗部。使用叠加混合模式可使底色的图像饱和度及对比度得到相应提高，这会使图像看起来更加鲜亮。公式如下：

```
if(B <= 0.5) {
    C = 2 * A * B;
} else {
    C = 1 – 2 * (1 - A) * (1 - B);
}
```

上层决定下层中间色调偏移的强度。如果上层为50%灰，则结果完全为下层像素的值。如果上层比50%灰暗，则下层的中间色调将向暗地方偏移。如果上层比50%灰亮，则下层的中间色调将向亮地方偏移。对于上层比50%灰暗，下层中间色调以下的色带变窄（原来为0～2×0.4×0.5，现在为0～2×0.3×0.5），中间色调以上的色带变宽（原来为2×0.4×0.5～1，现在为2×0.3×0.5～1）；反之亦然。

4.柔光混合模式

根据绘图色的明暗程度来决定最终色是变亮还是变暗。当绘图色比50%的灰要亮时，底色图像变亮；当绘图色比50%的灰要暗时，底色图像就变暗。如果绘图色有纯黑色或纯白色，那么最终色不是黑色或白色，而是会稍微变暗或变亮。如果底色是纯白色或纯黑色，则不产生任何效果。这种效果与发散的聚光灯照在图像上的相似。公式如下：

```
if(A <= 0.5) {
    C = (2 * A - 1) * (B - B * B) + B;
} else {
    C = (2 * A - 1) * (sqrt(B) - B) + B;
```

```
}
```

5.强光混合模式

根据绘图色来决定是执行“正片叠底”还是“滤色”混合模式。当绘图色比50%的灰要亮时，底色变亮，这与执行滤色混合模式一样，对增加图像的高光非常有帮助；当绘图色比50%的灰要暗时，底色变暗，这与执行正片叠底混合模式一样，可增加图像的暗部。当绘图色是纯白色或黑色时，得到的是纯白色和黑色。这种效果与耀眼的聚光灯照在图像上的相似。公式如下：

```
if(A <= 0.5) {
    C = 2 * A * B;
} else {
    C = 1 – 2 * (1 - A) * (1 - B);
}
```

强光混合模式的效果完全等价于叠加混合模式中两个图层进行顺序交换的效果。如果上层的颜色高于50%灰，则下层越亮，反之越暗。

9.3 使用FFmpeg内部的视频滤镜

通过前面两节的介绍，大家应该已经了解了图像的基本处理以及比较复杂的处理。但是，如果在工作中遇到一些问题，是否需要我们自己去实现一些很基础的视频滤镜呢？答案当然是否定的，所以本节介绍如何利用FFmpeg的视频滤镜模块来解决应用场景下的问题。

9.3.1 FFmpeg视频滤镜介绍

第3章已经详细介绍了如何使用FFmpeg的命令行模式来完成视频滤镜的添加，因为在实际工作中，特别是在客户端开发中很难直接使用命令行去完成工作，所以本节会详细介绍如何在代码层使用FFmpeg提供的内置视频滤镜。

基于前面章节对FFmpeg的了解，不论是音频滤镜还是视频滤镜，都是针对原始格式进行的操作，而原始格式对应到FFmpeg中，封装的结构体就是AVFrame，所以我们进行滤镜处理的时机是确定的，即在编码之前或者解码之后。在录制视频的场景下，视频处理的时机如图9-5所示。

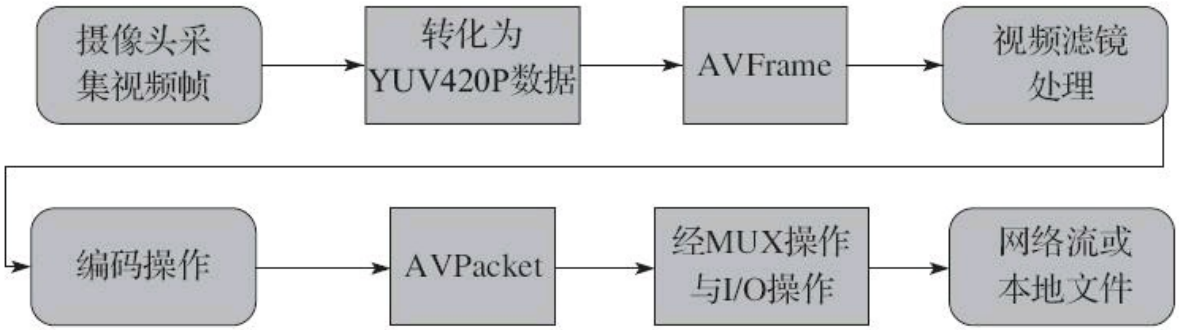


图 9-5

从图9-5中可以看到，视频滤镜是针对解码之后的AVFrame进行的处理，处理完毕之后的数据格式也是原始数据格式，最终再进行编码以及写到网络流或者本地文件中。在播放器场景下，视频处理的流程如图9-6所示。

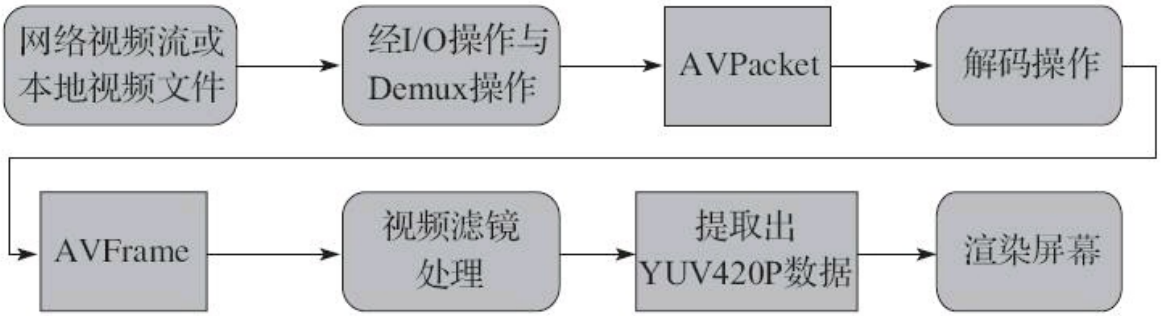


图　9-6

视频的播放过程恰好是录制的一个逆过程，当解码结束之后，就可以进行视频处理，处理完毕之后也是原始格式。现在我们已经明确了视频滤镜处理的时机或者FFmpeg视频滤镜的输入是什么了，并且了解了输出也是一个AVFrame的数据结构。接下来看如何使用FFmpeg的视频滤镜。在使用FFmpeg做视频滤镜处理之前，要确保在编译FFmpeg的配置阶段打开了我们想使用的视频滤镜，如果没有打开想使用的滤镜，那么在初始化的时候就会出错。

9.3.2 滤镜图的构建

在最开始注册所有封装格式和编码器的地方也要对过滤器进行注册，代码如下：

```
avfilter_register_all();
```

下面以播放器应用为例进行讲解。首先，打开资源文件，然后初始化视频滤镜处理器。由于在FFmpeg中的视频滤镜处理器是以一个图状的结构来完成复杂图像处理的，因此先要分配出一个处理器的图状结构来完成视频滤镜的操作，代码如下：

```
AVFilterGraph *filter_graph;
filter_graph = avfilter_graph_alloc();
```

接下来为_graph添加一个起始的Filter，作为视频帧数据的接受者，代码如下：

```
AVFilterContext *buffersrc_ctx;
AVFilter *buffersrc = avfilter_get_by_name("buffer");
char args[512];
snprintf(args, sizeof(args),
    "video_size=%dx%d:pix_fmt=%d:time_base=%d/%d:pixel_aspect=%d/%d",
    pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
    pCodecCtx->time_base.num, pCodecCtx->time_base.den,
    pCodecCtx->sample_aspect_ratio.num,
    pCodecCtx->sample_aspect_ratio.den);
int ret = avfilter_graph_create_filter(&buffersrc_ctx, buffersrc, "in", args,
    NULL, filter_graph);
if(ret < 0) {
    LOGI("Cannot create buffer source Filter :", av_err2str(ret));
    return ret;
}
```

从上述代码中可以看到，第一步要通过名称将“buffer”这个Filter找出来；第二步调用方法av filter_graph_create_filter来实例化这个Filter（由于这个Filter是源，所以要用args配置好所有的参数），并且加入上一阶段创建的图中，如果返回负数，则输出信息并且返回给客户端

代码。接下来再为_graph添加一个终点的Filter，作为提供给外界经过视频滤镜处理过的视频帧，代码如下：

```
AVFilterContext *buffersink_ctx;
enum PixelFormat pix_fmts[] = { PIX_FMT_GRAY8, PIX_FMT_NONE };
AVFilter *buffersink = avfilter_get_by_name("buffersink");
AVBufferSinkParams *buffersink_params = av_buffersink_params_alloc();
buffersink_params->pixel_fmts = pix_fmts;
ret = avfilter_graph_create_filter(&buffersink_ctx, buffersink, "out", NULL,
    buffersink_params, filter_graph);
av_free(buffersink_params);
if (ret < 0) {
    LOGI("Cannot create buffer sink :", av_err2str(ret));
    return ret;
}
```

上述代码会将名称为“buffersink”这个Filter按照配置的参数实例化并且加入图中，代表这个图状结构的最后一个Filter。现在这个图的架子基本搭建起来，接下来就是把实际要处理的图像的Filter加入图中。但在此之前，要先解决两个问题：第一个问题是这个Filter应如何表示；第二个问题是这个Filter应该放入图的哪个位置。先看如何描述一个Filter，在FFmpeg中使用字符串来描述Filter，比如给图片做镜像的效果器描述为：

```
char filters_descr[512];
snprintf(filters_descr, sizeof(filters_descr), "vflip");
```

如果需要多个效果器同时工作，则可以以逗号的形式将各个效果器连接起来。下面的代码是针对Android平台的摄像头采集出的一张640×480的图片，先做裁剪，然后做镜像，最后做一次270度或者90度的旋转，代码如下：

```
char filters_descr[512];
int videoWidth = 480;
int videoHeight = 480;
int cropLeftMargin = 180;
int cameraId = FACING_FRONT;// or FACING_BACK
snprintf(filters_descr, sizeof(filters_descr),
    "crop=%d:%d:%d:0,vflip,transpose=%d", videoWidth, videoHeight,
    cropLeftMargin, cameraId % 2 == 0  1 : 2);
```

下面来看这个Filter应该连接在图中哪一个节点的后面，代码如下：

```
AVFilterInOut *inputs = avfilter_inout_alloc();
inputs->name = av_strdup("out");
inputs->filter_ctx = buffersink_ctx;
inputs->pad_idx = 0;
inputs->next = NULL;
```

上述代码中的AVFilterInOut结构体代表效果器图中的一个具体节点，而这个节点实际上就是前面所声明的buffersink所代表的Filter。接着来看这个节点应该连接到哪一个节点上，代码如下：

```
AVFilterInOut *outputs = avfilter_inout_alloc();
outputs->name = av_strdup("in");
outputs->filter_ctx = buffersrc_ctx;
outputs->pad_idx = 0;
outputs->next = NULL;
```

上述代码中的节点实际上是效果器图的起始节点。效果器节点应连接到前面的buffersrc节点上。解决了上述问题之后，将这个Filter加入图中，完成操作之后，要释放掉我们构建出的输入和输出节点，代码如下：

```
ret = avfilter_graph_parse_ptr(filter_graph, filters_descr,
    &inputs, &outputs, NULL);
avfilter_inout_free(&outputs);
avfilter_inout_free(&inputs);
if (ret < 0){
    LOGI("avfilter_graph_parse_ptr failed : ", av_err2str(ret));
    return ret;
}
```

这里可能有读者会怀疑，我们是不是将这个Filter给连接反了，也就是将这个Filter节点的input和output搞错了。确实，这个地方比较绕，笔者读源码发现，这应该和FFmpeg内部是如何构建效果器图是有关系的，也正是因为FFmpeg内部实现的原因，所以这里就把节点的输入和输出写反了，读者可以通过源码理解一下。

现在已经把整个图构建起来了，接下来做最重要的一步，即配置这个图，代码如下：

```
if ((ret = avfilter_graph_config(filter_graph, NULL)) < 0){
    LOGI("avfilter_graph_config failed :", av_err2str(ret));
    return ret;
}
```

至此，整个效果器的图就配置好了。接下来学习如何真正使用刚刚配置好的这个视频滤镜图。

9.3.3 使用与销毁滤镜图

前面已经介绍过，这个滤镜图的输入是一个AVFrame结构体，输出同样也是一个AVFrame结构体。因此，先分配出一个输入视频帧对象和一个输出视频帧对象，代码如下：

```
AVFrame *inputFrame = avcodec_alloc_frame();
AVFrame *outputFrame = avcodec_alloc_frame();
```

接下来是如何使用这个滤镜图了。假设无论是录制视频的应用还是视频播放器的应用，都已经将inputFrame这个对象填充好，那么如何将inputFrame对象中的内容经过滤镜图处理成为一个outputFrame对象呢？代码如下：

```
if (av_buffersrc_add_frame_flags(buffersrc_ctx, inputFrame, 0) < 0) {
    LOGI("Error while feeding the filter graph");
    return -1;
}
while (1) {
    ret = av_buffersink_get_frame(buffersink_ctx, outputFrame);
    if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF){
        break;
    }
    //  Do Something
    av_frame_unref(outputFrame);
}
```

从以上代码中可以看出，第一步是将inputFrame填充到滤镜图中的第一个节点中去；第二步是进入一个循环，不断从滤镜图中的最后一个节点拉取出处理完毕的outputFrame，然后将outputFrame进行对应处理；如果是视频播放器，则抽取出YUV数据封装到VideoFrame结构体中；如果是视频录制器，则进行赋值pts后做编码操作。最终去掉对outputFrame的引用并重置这个结构体里所有的变量。

待最终使用完毕之后，需要销毁掉滤镜图与对应分配的视频帧资源，代码如下：

```
avfilter_graph_free(&filter_graph);
```

```
// 销毁其他资源
av_frame_free(&inputFrame);
av_frame_free(&outputFrame);
```

9.3.4 常用滤镜介绍

下面介绍FFmpeg中常用的视频滤镜。本节中的每一个最终输出都是一串字符，输出的这个字符串可以直接应用到9.3.2节中构建的Filter。

1.翻转、旋转、裁切

（1）翻转

在FFmpeg中提供了垂直翻转和水平翻转两种类型的效果器。其中，水平翻转是给图片做镜像操作的，名称为“hflip”；而垂直翻转的名称为“vflip”。在9.3.2节中可以直接使用的滤镜字符串表示为：

```
const char* filters_descr = "hflip";
```

（2）旋转

旋转在FFmpeg中使用transpose来表示，有0、1、2、3四种取值，它们分别表示的含义如下：

·0：逆时针旋转90度并做垂直翻转；

·1：顺时针旋转90度；

·2：逆时针旋转90度（即顺时针旋转270度）；

·3：顺时针旋转90度并做垂直翻转。

对应于9.3.2节，可以直接使用的滤镜字符串表示为：

```
const char* filters_descr = "transpose=1";
```

（3）裁切

裁切在FFmpeg中使用crop来表示，参数顺序依次为width：height：

top：left。其中width和height分别代表的含义是要裁切成的目标视频的宽和高，而top和left代表的意义是从原始视频的顶部位置和左边位置开始截取。对应于9.3.2节，可以直接使用的滤镜字符串表示为：

```
const char* filters_descr = "crop=480:480:180:0";
```

上述代码描述了将一个宽为480、长为640的视频从顶部距离180开始裁剪，最终裁剪为一个长和宽都是480的视频。

2.增加水印/消除水印

（1）消除水印

消除水印的做法比较简单，前提是要知道水印在视频的什么区域，视频滤镜的内部实现会将这一块区域做一个模糊，从而将水印部分模糊掉。在FFmpeg中用delogo来消除水印的滤镜，参数为x=x1：y=y1：w=w1：h=h1：band=band1。其中前四个参数的含义大家应该比较清楚，x代表距离图片左边缘的距离；y代表距离图片上边缘的距离；w与h分别代表水印区域的宽度和高度；最后一个参数band代表模糊程度（默认值是1），通常可以设置为5，也可以根据水印和背景的颜色差别来设置这个值。对应于9.3.2节，可以直接使用的滤镜字符串表示为：

```
const char* filters_descr = "delogo=x=0:y=0:w=100:h=77:band=10";
```

（2）增加水印

相较于消除水印，增加水印是工作中更加常用的一种视频滤镜。首先需要一张logo图片，利用movie将这张图片读入，并记为logo参数；然后对整个滤镜图结构，将输入视频帧记为in，输出记为out，利用overlay这个滤波器完成增加水印操作。对应于9.3.2节，可以直接使用的滤镜字符串表示如下：

```
const char* filters_descr = "movie=/Users/apple/logo.png[logo];
    [in][logo]overlay=main_w-overlay_w-10:10[out]";
```

如上述代码所示，首先利用movie将要添加的水印读入进来，并记为logo变量；然后和输入视频帧（记为in）一块组成输出视频帧（记为out）。组合的过程是通过overlay的参数来表示的，在overlay参数中有以下几个内置变量，main_w和main_h为输入视频帧的宽和高，overlay_w和overlay_h为读入进来的水印的宽和高。overlay的参数顺序与意义也比较简单，它的第一个参数就是距离原始视频帧左边缘的距离，第二个参数就是距离原始视频帧上边缘的距离。所以上述代码中是将水印放到了原始图像的右上角，如果读者的场景是放到左上角或者其他位置，那么可以根据内置变量表示出来。

3.对比度、饱和度、亮度调节

下面要介绍的是eq这个视频滤镜，在低版本的FFmpeg中是没有这个视频滤镜的，比如，笔者验证的2.1.1版本就没有这个效果器，但在2.8.5版本中有这个效果器。具体如何查看安装的FFmpeg是否包含这个效果器，可以执行以下命令：

```
ffmpeg -filters | grep eq
```

（1）对比度

在eq这个视频滤镜中可以调节对比度，参数是contrast，取值范围是-2.0～2.0，默认值是1.0。一般可将增加对比度设置为1.25或者1.5，其具体含义在9.1.2节中已经讲解过。

（2）饱和度

在eq这个视频滤镜中可以调节饱和度，参数是saturation，取值范围是0.0～3.0，默认值是1.0，具体含义可以参考9.1.3节。

（3）调节亮度

在eq这个视频滤镜中可以调节亮度，参数是brightness，取值范围是-1.0～1.0，默认值0.0，取值为正数，代表增加亮度。

下面给出一个增加对比度、增加饱和度，并且提亮的滤镜代码：

```
const char* filters_descr = "eq=contrast=1.25:brightness=0.05:saturation=1.05
```

4.添加面板

除基础效果器的使用，在日常工作中还可能会经常用到分辨率转换，这时就需要通过面板来处理。比如，有一个480×480的视频，因为某些平台的限制，现在需要将这个视频转换为一个16：9的宽屏视频，两侧填充黑边，如何实现？先来看滤镜代码如下：

```
const char* filters_descr = "pad=iw*16/9:iw:(ow-iw)/2:0:black";
```

上述代码会使用pad效果器建立一个面板，并把输入画面画到面板上。在这个效果器里有以下几个内置变量：iw和ih分别代表input画面的宽和高，ow和oh分别代表output画面的宽和高。pad的参数一共有四个，前两个参数分别代表面板的宽和高。在需求中，由于要做成比例为16：9的宽屏视频，所以计算出最终面板的宽度应该是当前宽度乘以16除以9，而高度维持不变。后面两个参数分别代表从距离面板的左边缘和上边缘多大距离处开始画。这里用到了内置变量ow，我们期望原始视频放在面板中间，所以距离面板左边缘的距离是ow-iw除以2，而距离上边缘的距离自然是0。

9.4 使用OpenGL ES实现视频滤镜

在移动平台上，大家关心的是性能，特别在图像或者视频处理方面，性能问题更是一个突出的问题。那在移动平台的图像处理方面如何提高性能呢？答案是通过OpenGL ES。我们要充分利用显卡并行工作的特点，这样可以极大地提高图像或视频的处理速度。

9.4.1 加水印

在第9.3节中已介绍过使用FFmpeg这个框架中自带的视频滤镜完成了一些操作，其中包括了加水印的操作。虽然不同的架构设计需要不同的技术实现，但如果是在已经架设过OpenGL ES环境的系统下，笔者还是强烈推荐使用OpenGL ES来完成添加水印的操作，因为这样会有更快的处理速度，更低的CPU消耗。本节来看如何使用OpenGL ES完成添加水印的操作。

水印的源一般是一张PNG的图片，所以需要为工程引入一个可以解码PNG的库（当然也可以让各个客户端各自实现解码，但是这不太符合跨平台系统的设计规则）。由于这个库需要同时运行在Android平台和iOS平台上，所以要求这个库是C或者C++语言实现的。最终，我们选择libpng库，libpng库的解码的输出一般是RGBA格式的byte类型的数组。接下来将这个数组上传到显卡中，使其成为一个纹理对象，最终将这个水印的纹理绘制到原始视频帧的指定位置，得到一个最终的带水印的视频帧。

1.引入libpng库

读者可以从SourceForge上下载最新的libpng的源码，也可以从本书的代码目录中找到笔者使用的libpng版本的源码。笔者没有使用编译静态库的形式来引用libpng库，因为这样需要编译多个平台，而libpng库里的源码文件并不多，直接以源码的形式引用也并不复杂，所以对libpng库的引用就采用源码的方式。拿到源码之后，以单独的一个目录放入工程目录中。

（1）libpng的API介绍

首先来看libpng库中的数据结构。

·png_structp变量：在libpng初始化的时候创建，由libpng库内部使用，代表libpng的调用上下文，开发者不应该对这个变量进行访问。调用libpng的API时，需要把这个参数作为第一个参数传入。

·png_infop变量：初始化完成libpng之后，可以从libpng中获得该类

型变量指针。这个变量保存了png图片数据的信息，开发者可以修改和查阅该变量。

接下来初始化libpng这个库，代码如下：

```
png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
    NULL, NULL, NULL);
```

初始化libpng的时候，用户可以指定自定义错误处理函数，如果无须指定，则传入NULL即可。png_create_read_struct函数返回一个png_structp变量，前面已经提到过，该变量不应该被用户访问，应该在以后调用libpng的函数时传递给libpng库。

然后调用获得图片信息的接口：

```
png_infop info_ptr = png_create_info_struct(png_ptr);
```

得到图片信息的结构体后，接下来就设置libpng的数据源了。使用自定义回调函数设置libpng数据源，代码如下：

```
ReadDataHandle png_data_handle = (ReadDataHandle) {
    {png_data, png_data_size}, 0
};
png_set_read_fn(png_ptr, &png_data_handle, read_png_data_callback);
```

实际上，我们把客户端代码读取出来的PNG图片的所有数据和数据的大小封装到结构体中，然后直接当做函数png_set_read_fn的第二个参数来传递给回调函数read_png_data_callback。对于这个回调函数的定义，具体如下：

```
static void read_png_data_callback(png_structp png_ptr,
        png_byte* raw_data, png_size_t read_length) {
    ReadDataHandle* handle = png_get_io_ptr(png_ptr);
    const png_byte* png_src = handle->data.data + handle->offset;
    memcpy(raw_data, png_src, read_length);
    handle->offset += read_length;
}
```

上述代码首先利用函数png_get_io_ptr取出设置在数据源函数中的指针，然后将read_length里面的数据按照要求的数量拷贝到raw_data这块内存区域中。

做好设置数据源工作之后，接下来就到了PNG图像处理部分，步骤如下。

1）读取输入png数据的图片信息，代码如下：

```
png_read_info(png_ptr, info_ptr);
```

该函数会把输入png数据的信息读入info_ptr数据结构中。

2）查询图像信息，代码如下：

```
png_get_IHDR(png_ptr, info_ptr, &width, &height, &bit_depth,
    &color_type, NULL, NULL, NULL);
```

前面提到png_read_info会把输入png数据的信息读入info_ptr数据结构中，接下来要调用API查询该信息。

3）设置png输出参数（转换参数），代码如下：

```
if (png_get_valid(png_ptr, info_ptr, PNG_INFO_tRNS))
    png_set_tRNS_to_alpha(png_ptr);
if (color_type == PNG_COLOR_TYPE_GRAY && bit_depth < 8)
    png_set_expand_gray_1_2_4_to_8(png_ptr);
if (color_type == PNG_COLOR_TYPE_PALETTE)
    png_set_palette_to_rgb(png_ptr);
if (color_type == PNG_COLOR_TYPE_PALETTE || color_type == PNG_COLOR_TYPE_RGB)
    png_set_add_alpha(png_ptr, 0xFF, PNG_FILLER_AFTER);
if (bit_depth < 8)
    png_set_packing(png_ptr);
else if (bit_depth == 16)
    png_set_scale_16(png_ptr);// 注意在高版本的库中应调用png_set_strip_16(png_ptr
```

这一步非常重要，开发者可以通过调用png_set_xxxxx函数指定输出数据的格式，比如RGB888、ARGB8888等输出数据格式。注意代码的最后一部分，如果位深度是16，在libpng库提供的高版本的API中应该调

用方法png_set_strip_16。这部分代码执行结束后，会将图像转换为RGB888的数据格式。当然，如果开发者想转换为YUV的格式或者其他格式，可以通过给libpng库设置转换函数来实现，这里就不介绍了，如果有需要，读者可以自己查阅libpng库的官方文档来设置。

4）更新png数据的详细信息。

通过前面设置png数据的图片信息，肯定会有一些变化，下面需要调用函数png_read_update_info更新图片的详细信息：

```
png_read_update_info(png_ptr, info_ptr);
```

5）读取png数据。

首先计算出每一行字节buffer的大小，然后乘以高度得到整个图片的大小，最终调用读取函数将数据全部读取到分配的内存区域中。

```
const png_size_t row_size = png_get_rowbytes(png_ptr, info_ptr);
const int data_length = row_size * height;
png_byte* raw_image = malloc(data_length);
png_byte* row_ptrs[height];
png_uint_32 i;
for (i = 0; i < height; i++) {
    row_ptrs[i] = raw_image + i * row_size;
}
png_read_image(png_ptr, &row_ptrs[0]);
```

6）结束读取数据。

通过png_read_end结束读取png数据，代码如下：

```
png_read_end(png_ptr, info_ptr);
```

7）释放libpng的内存，代码如下：

```
png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
```

8）将解码出来的数据封装到结构体中返回，代码如下：

```
return (RawImageData) {
    png_info.width,
    png_info.height,
    raw_image.size,
    get_gl_color_format(png_info.color_type),
    raw_image.data};
```

至此，libpng库的API调用就讨论结束了，大家可以参考代码仓库中的image.c文件，接下来会将它集成到Android和iOS客户端。

（2）Android平台的集成

在Android平台集成的重点是如何书写Android.mk文件，将这些源码直接集成到我们的NDK工程中。Android.mk的代码如下：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_C_INCLUDES += $(LOCAL_PATH)/libpng
LOCAL_EXPORT_LDLIBS := -lz
LOCAL_SRC_FILES := \
./libpng/png.c \
./libpng/pngerror.c \
./libpng/pngget.c \
./libpng/pngmem.c \
./libpng/pngpread.c \
./libpng/pngread.c \
./libpng/pngrio.c \
./libpng/pngrtran.c \
./libpng/pngrutil.c \
./libpng/pngset.c \
./libpng/pngtrans.c \
./libpng/pngwio.c \
./libpng/pngwrite.c \
./libpng/pngwtran.c \
./libpng/pngwutil.c
LOCAL_MODULE := libpng
include $(BUILD_STATIC_LIBRARY)
```

从以上代码可以注意到，除了正常地将源码都包含到整个文件中之外，还需要引入libz库，这样书写Android.mk文件之后，就可以编译出对应CPU平台下的静态库了，并且可利用第一步书写的调用客户端来完成解码操作。

（3）iOS平台的集成

iOS平台的集成操作，实际上只要把这个目录加到Xcode工程中就可，但是有一个问题需要处理，那就是引入libz库，需要我们在Build Settings选项里找到Other Link Flags，然后加入-lz，这样才可以使整个工程编译通过。

2.渲染水印，完成绘制

渲染水印其实很简单，就是先将原始视频画到一个FBO上，然后将水印图片画到相应的位置上去，此时这个FBO就相当于由两个图层组成，底层是原始视频帧画面，上层就是水印图片，整段代码结构如下：

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
    outputTextureID, 0);
// 1.将原始纹理作为底层绘制
drawInputTexture();
// 2.在合适的位置绘制水印图片
drawOverlayTexture();
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
    0, 0);
```

上述代码中首先将outputTextureID关联到FBO上，然后进行渲染操作，渲染到FBO上的内容就相当于绘制到了outputTextureID上，最终在绘制结束之后，将FBO关联的纹理ID设置为0。在绘制过程中，绘制原始纹理没有什么需要说明的，但是，为了让绘制水印图片的方法更通用，这里要详细讲解VertexShader中的旋转平移缩放矩阵的用法。旋转平移缩放矩阵在绘图过程中会经常用到，比如在Android平台的SurfaceView的自定义动画中，要将Bitmap画到画布上。在OpenGL ES中，旋转平移缩放矩阵用于在VertexShader中进行物体坐标的变换。物体坐标是一个四维向量，记为（x，y，z，w）。下面就介绍如何使用矩阵来变换物体的坐标。先来看矩阵和向量相乘的法则，如式（9-2）所示。

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} ax+by+cz+dw \\ ex+fy+gz+hw \\ ix+jy+kz+lw \\ mx+ny+oz+pw \end{pmatrix} \qquad (9\text{-}2)$$

对于GLSL中矩阵和向量的相乘，可以表示为：

```
mat4 myMatrix;
vec4 myVector;
vec4 transformedVector = myMatrix * myVector;
```

接下来看一个单位矩阵的表示，如式（9-3）所示。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 1*x+0*y+x*z+0*w \\ 0*x+1*y+0*z+0*w \\ 0*x+0*y+1*z+0*w \\ 0*x+0*y+0*z+1*w \end{pmatrix} = \begin{pmatrix} x+0+0+0 \\ 0+y+0+0 \\ 0+0+z+0 \\ 0+0+0+w \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \qquad (9\text{-}3)$$

由式（9-3）可见，单位矩阵实际上对向量是没有任何意义的，但是理解单位矩阵绝对是非常重要的，并且无论是平移、缩放还是旋转，都是基于单位矩阵进行变换的。所以我们来看平移矩阵是如何设置的，代码如下：

```
float translateMartrix[4][4] = {
    1, 0, 0, Tx
    0, 1, 0, Ty
    0, 0, 1, Tz
    0, 0, 0, 1
}
```

在二维图像中，一般都仅指定（x，y）的坐标，当仅指定了x和y坐标的时候，其他两个坐标值（z和w）将被自动指定为0和1。由于w的值默认为1，可以看到上述平移矩阵中的Tx、Ty、Tz就是针对于w是1，所以就可以在（x，y，z）三个方向上作用位移变量（Tx，Ty，Tz）了。而旋转矩阵比较复杂，这里只列举了一个绕z轴旋转角度a的矩阵，而绕z轴旋转也是二维世界里使用最多的旋转，如下所示：

```
float rotateMartrix[4][4] = {
    cos(a),  sin(a),  0, 0
    -sin(a), cos(a),  0, 0
    0,       0,       1, 0
    0,       0,       0, 1
}
```

从以上代码可以看到绕z轴旋转就是z轴不动，x轴和y轴去做相应角度的旋转。如果读者想知道绕x轴和绕y轴旋转矩阵的用法，可以参考代码仓库中示例代码的用法。缩放矩阵就会简单一些，如下所示：

```
float scaleMartrix[4][4] = {
    scaleX,   0,       0,       0
    0,        scaleY,  0,       0
    0,        0,       scaleZ,  0
    0,        0,       0,       1
}
```

从上述矩阵中可以看到，当这三个矩阵都确定之后，剩下的就是将三个矩阵合并成为一个矩阵，代码如下：

```
mat4 transformMatrix = TranslationMatrix * RotationMatrix * ScaleMatrix;
```

这里一定要注意顺序，先执行缩放，接着旋转，最后才是平移（矩阵的左乘和右乘得到的结果是不一样的）。只有这样，才可以先确定中心点，然后再绕中心点进行旋转，以及按照中心点进行平移。如果顺序搞乱了，就会得到不一样的结果，自然也不是我们预期的结果了。

下面将这个矩阵放入渲染水印的VertexShader中，并在Shader中将矩阵去乘以物体坐标，得到的新的物体坐标就是我们期望的水印放置的位置，以及达到旋转角度和缩放的程度。大家可以参考示例代码中的利用OpenGL ES添加水印效果器的实例。

9.4.2 添加自定义文字

给视频添加自定义文字也是工作中常碰到的场景，所以本节讨论如何添加自定义文字。首先需要向大家事先交代一个背景，就是OpenGL ES内部不可以直接进行文字的绘制，并且对绘制的文字有可能还有字体、阴影等需求，所以我们使用Android平台和iOS平台将文字绘制到一个Bitmap上；然后将Bitmap传递给OpenGL ES系统；最后由OpenGL ES进行处理与渲染。

1.平台绘制需要的文字

由于平台相关性，我们分两部分来介绍如何在Android平台和iOS平台上将文字制作成图片。一部分代码由底层的OpenGL ES系统回调客户端完成，该客户端会将文字的大小、文字的颜色、是否需要文字阴影，以及文字所占位置和对齐方式等参数传递给客户端代码，客户端会按照要求将文字绘制到一个黑色背景的图片上（最终利用黑色的alpha通道为0的特性仅显示出文字区域，所以背景使用黑色），最终将图片以RGBA的数据格式传递给OpenGL ES系统，OpenGL ES系统则会将这个图片再渲染到视频上，从而得到我们想要绘制的结果。

根据上述的设计，首先规定回调函数的接口，代码如下：

```
bool getTextPixels(int width, int height,
    int textLabelLeft, int textLabelTop,
    int textLabelWidth, int textLabelHeight,
    int textColor, int textSize, int textAlignment, char*  text,
    float shadowRadius, float textShadowXOffset,
    float textShadowYOffset, int shadowColor,
    byte[] buffer);
```

其中，第一行的参数代表这幅图片的宽和高；第二行和第三行的参数代表文字所在的位置以及文字的宽和高；第四行的参数代表文字颜色、大小、对齐方式以及文字内容；第五行和第六行的参数代表文字阴影的配置；第七行的参数就是生成这幅图片内容存放的内存区域。

（1）Android平台的实现

首先根据要求的宽、高以Bitmap的形式制作出一张图片，代码如下：

```
Bitmap textBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_888
```

制作的Bitmap的背景色默认为黑色，把这个Bitmap画到一个画布上，这样就得到了一个黑色画布，然后就可以在这个画布上画文字了。对这个画布进行操作，就相当于对这幅图片做了相同的操作。代码如下：

```
Canvas localCanvas = new Canvas(textBitmap);
```

要想在画布上画文字，首先得有一支画笔，并且要根据需求设置画笔的参数，代码如下：

```
Paint localPaint = new Paint();
localPaint.setColor(Color.argb(255, Color.red(textColor),
    Color.green(textColor), Color.blue(textColor)));
localPaint.setShadowLayer(shadowRadius, textShadowXOffset, textShadowYOffset,
    Color.argb(255, Color.red(shadowColor),
    Color.green(shadowColor), Color.blue(shadowColor)));
localPaint.setTextSize(textSize);
localPaint.setAntiAlias(true);
localPaint.setTextAlign(Paint.Align.CENTER);
```

从上述代码中可以看到，分别设置了画笔的颜色、阴影、文字大小、抗锯齿，以及文字的对齐方式。画笔设置完之后，在规定绘制的区域中绘制文字，代码如下：

```
Rect targetRect = new Rect(textLabelLeft, textLabelTop,
    textLabelLeft + textLabelWidth, textLabelTop + textLabelHeight);
FontMetricsInt fontMetrics = localPaint.getFontMetricsInt();
int baseline = (targetRect.bottom + targetRect.top –
    fontMetrics.bottom - fontMetrics.top) / 2;
localCanvas.drawText(text, targetRect.centerX(), baseline, localPaint);
```

其中，targetRect是文字要绘制的矩形区域；baseline的计算是为了将文字绘制在这个矩形区域竖直方向的正中间。最后将这个Bitmap的内

容复制到一个内存区域中，再返回给调用端，代码如下：

```
int capacity = width * height * 4;
ByteBuffer dst = ByteBuffer.allocate(capacity);
textBitmap.copyPixelsToBuffer(dst);
dst.position(0);
dst.get(buffer, 0, capacity);
```

（2）iOS平台的实现

在iOS平台上绘制文字以及图形时，使用CoreGraphics中的CGContext。首先通过宽、高以及表示格式创建出一个Bitmap，代码如下：

```
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
CGContextRef BitmapContext = CGBitmapContextCreate(buffer, width, height, 8,
    width * 4, colorSpace, kCGImageAlphaNoneSkipLast);
CGColorSpaceRelease(colorSpace);
```

然后设置当前图片的缩放和位移矩阵，代码如下：

```
CGContextTranslateCTM(BitmapContext, 0.0, height);
CGContextScaleCTM(BitmapContext, 1.0, -1.0);
```

在进行文字的绘制之前，要先将这个BitmapContext作为绘制的对象，类似于OpenGL ES中的绑定纹理ID，这里就绑定当前绘制的Bitmap，在绘制结束之后解除绑定操作，并将BitmapContext释放掉。代码如下：

```
UIGraphicsPushContext(BitmapContext);
drawTextPixels(param, context);
UIGraphicsPopContext();
CGContextRelease(BitmapContext);
```

最核心的drawTextPixels就是实际的绘制文字了，就是在指定的区域按照指定的参数绘制文字，代码如下：

```
NSTextAlignment alignment;
if (textAlignment == -1) {
    alignment = NSTextAlignmentLeft;
} else if (textAlignment == 0) {
    alignment = NSTextAlignmentCenter;
} else {
    alignment = NSTextAlignmentRight;
}
NSMutableParagraphStyle *paragraphStyle = [[NSParagraphStyle
    defaultParagraphStyle] mutableCopy];
paragraphStyle.alignment = alignment;
NSDictionary *attributes = @{NSFontAttributeName: [UIFont
    systemFontOfSize:textSize],NSForegroundColorAttributeName:
    UIColorFromRGB(textColor),NSParagraphStyleAttributeName:
    paragraphStyle};
CGRect rect = CGRectMake(textLabelLeft, textLabelTop, textLabelWidth,
    textLabelHeight);
[text drawInRect:rect withAttributes:attributes];
```

至此，将文字按照指定的大小、颜色以及区域绘制到了提供的内存区域上。注意，生成的图片除了文字外，其他的区域都是黑色的，理解这一点对于第二步“将这个图片进行渲染”有很重要的意义。

2.渲染文字，完成绘制

对于前面生成的文字图片，除了文字区域外，其他的区域都是黑色的，并且黑色区域中每一个像素的RGBA通道中A通道的数值都为0。但是，对于文字区域的像素，A通道都是有自己的透明度属性的，所以我们依赖于这一点将文字图片和视频图片进行混合，混合代码如下：

```
vec4 image = texture2D(imageTexture, v_texcoord);
vec4 textImage = texture2D(textTexture, v_texcoord);
float r = textImage.r + (1.0 – textImage.a)*image.r;
float g = textImage.g + (1.0 – textImage.a)*image.g;
float b = textImage.b + (1.0 – textImage.a)*image.b;
vec4 finalColor = vec4(r, g, b, 1.0);
```

从上述代码中可以看到，对于文字图片的黑色区域，若alpha通道为0，则使用的全是原始视频帧的色值；对于文字区域，若alpha通道不再为0，则会将两个颜色进行混合，得到最新的色值，并将得到的图片输出。

在文字的渲染部分常使用的是淡入淡出，即当文字出现的时候以淡入的效果进入，然后维持一段时间，等文字要消失之前要淡出的效果消

失。要实现相应的效果，需要引入一个progress的计算，并且将progress应用到上述公式中。首先，定义以下几个变量，使用sequenceIn表示文字效果器开始作用时间；使用sequenceOut表示结束作用时间；使用fadeInEndTime表示出现过程中的淡入效果完成时间点；使用fadeOutStartTime表示退出过程中淡出效果开始时间点。所以最终progress的计算公式如下：

```
float progress = 1.0;
int64_t currentTime = pos * 1000; // 换算为毫秒
if(currentTime <= fadeInEndTime){
    progress = (currentTime - sequenceIn) / (fadeInEndTime – sequenceIn);
} else if(currentTime >= fadeOutStartTime){
    float p = (currentTime - fadeOutStartTime) / (sequenceOut - fadeOutStartT
    progress = 1.0 - p;
}
```

以上代码中表示的意思是可以把时间轴分为三部分：第一部分是fadeIn部分，即代码中的第一个分支，progress的值会从0.0变化到1.0；第二部分是稳定显示部分，progress的默认值为1.0；第三部分是fadeOut部分，即代码中的第二个条件分支部分，progress的值会从1.0变化到0.0。

如何在两个混合图片中使用progress呢？代码如下：

```
vec4 image = texture2D(imageTexture, v_texcoord);
vec4 textImage = texture2D(textTexture, v_texcoord);
float r = textImage.r * progress + (1.0 – textImage.a * progress)*image.r;
float g = textImage.g * progress + (1.0 – textImage.a * progress)*image.g;
float b = textImage.b * progress + (1.0 – textImage.a * progress)*image.b;
vec4 finalColor = vec4(r, g, b, 1.0);
```

从以上代码中可以看出，当progress的值为0的时候，最终的色值全部都是原始图片帧的色值，随着向1.0变化，文字图片有文字的部分会慢慢显示出来，黑色区域永远不会显示，当progress的值达到1.0的时候就是最开始的公式，即将两幅图片按照文字图片的alpha值进行混合，从而作为最终OpenGL ES的渲染输出。

9.4.3 美颜效果器

对于摄像头采集出的图像，尤其是带有人脸的图像，是必须要做美颜处理的，而这才是本章中的重点，所以本节重点介绍基于OpenGL ES美颜算法的实现。美颜算法的实现有很多种，一般先通过磨皮算法将皮肤的一些纹理（痘痘、黑斑等）给磨掉，然后通过色相、提亮等效果器美化肤色。整个构成中，磨皮算法是最重要的一环，双边滤波算法是较为常用且效果也比较好的一种磨皮做法。9.2.4节已经讲解过双边滤波算法的原理，本节会给出如何在OpenGL ES的环境中实现双边滤波算法，以达到美颜的效果。

首先来看VertexShader，代码如下：

```
attribute vec4 position;
attribute vec4 inputTextureCoordinate;
uniform float texelWidthOffset;
uniform float texelHeightOffset;
const int GAUSSIAN_SAMPLES = 5;
varying vec4 blurCoordinates[GAUSSIAN_SAMPLES];
vec4 handleStepOffset(vec2 stepOffset)
{
    return vec4(inputTextureCoordinate.xy + stepOffset,
        inputTextureCoordinate.xy - stepOffset);
}

void main()
{
    gl_Position = position;
    vec2 singleStepOffset = vec2(texelWidthOffset, texelHeightOffset);
    blurCoordinates[0] = inputTextureCoordinate;
    blurCoordinates[1] = handleStepOffset(singleStepOffset);
    blurCoordinates[2] = handleStepOffset(singleStepOffset * 2.0);
    blurCoordinates[3] = handleStepOffset(singleStepOffset * 3.0);
    blurCoordinates[4] = handleStepOffset(singleStepOffset * 4.0);
}
```

上述代码就是找出当前像素点周围的像素点的坐标，而最终输出的blurCoordinates数组虽然仅有5个元素，但是代表的是当前像素点和周围的8个像素点，其中第1个元素为要计算的当前像素点，后面4个元素中每个元素实际上包含的是2个像素点，分别是正方向和反方向的像素点，所以共有8个像素点。它们会作为当前像素点的参考像素点，而计算这些参考像素点的步长由客户端传递过来的像素距离决定，计算公式

如下：

```
texelWidthOffset = 1.0 / width;
texelHeightOffset = 1.0 / height;
```

当然，如果想扩大参考像素点的范围，可以增大这两个步长值，最终VertexShader的输出结果就是blurCoordinates这个向量数组，这个向量数组将会传递到FragmentShader中。接下来的FragmentShader代码比较复杂，所以分开讲解。首先来看由客户端代码传递过来的uniform的值和VertexShader传递过来的像素坐标点数据：

```
uniform sampler2D inputImageTexture;
const lowp int GAUSSIAN_SAMPLES = 5;
varying highp vec4 blurCoordinates[GAUSSIAN_SAMPLES];
uniform mediump float distanceNormalizationFactor;
lowp vec4 sum;
lowp float gaussianWeightTotal;
```

如上述代码所示，distanceNormalizationFactor的值是客户端传递进来的，代表距离归一化的因子，默认值是4.0，取值范围是[1.0，8.0]。首先这个值用来确定当前参考像素点是否是边缘（edge）的参数，然后根据是否是边缘来确定作为一个有效的参考点的权重值。最后两个变量sum和gaussianWeightTotal是用来计算像素值相加总和和权重总和的。接下来进入main函数中，先把当前像素点的像素值取出来，代码如下：

```
lowp vec4 centralColor;
centralColor = texture2D(inputImageTexture, blurCoordinates[0].xy);
```

由于双边滤波是基于距离（高斯分布）与像素值差距双维度考虑权重的算法，所以每一个领域像素点的计算都要考虑这两个维度的因素。高斯权重的分布如下：

```
0.05,0.09,0.12,0.15,0.18,0.15,0.12,0.09,0.05
```

所有高斯权重加起来的总值为1.0，接下来通过中心像素点的权重计算当前像素点的值，并加到总像素值中去，同时也要把权重值加到总

的权重值中去，代码如下：

```
lowp float gaussianWeightTotal;
lowp vec4 sum;
gaussianWeightTotal = 0.18;
sum = centralColor * 0.18;
```

现在，按照高斯分布将剩余的四组领域像素点计算出来，先来看最相近的一组，代码如下：

```
vec4 sampleColor = texture2D(inputImageTexture, blurCoordinates[1].xy);
float distanceFromCentralColor = min(distance(centralColor, sampleColor) *
    distanceNormalizationFactor, 1.0);
float gaussianWeight = 0.15 * (1.0 - distanceFromCentralColor);
gaussianWeightTotal += gaussianWeight;
sum += sampleColor * gaussianWeight;
sampleColor = texture2D(inputImageTexture, blurCoordinates[1].zw);
distanceFromCentralColor = min(distance(centralColor, sampleColor) *
    distanceNormalizationFactor, 1.0);
gaussianWeight = 0.15 * (1.0 - distanceFromCentralColor);
gaussianWeightTotal += gaussianWeight;
sum += sampleColor * gaussianWeight;
```

在这段代码中，blurCoordinates的前两个x和y代表正向的一个像素点，后两个z和w代表反向的一个像素点。接着利用GLSL的内嵌函数distance计算出当前像素点和中心像素点的像素差值，再将差值乘以距离归一化因子，并与1.0比较，取较小的值，得到的这个值就代表当前像素点与中心像素点的颜色差距。值越小，代表差距越小，那么可以去做模糊处理。假设这个值是1.0，则代表不拿这个像素值参与中心像素点的计算，利用高斯权重值乘以1.0减去这个值，得到距离与像素值两个维度考虑的权重值。最后计算完毕剩余的三组值，得到最终像素值和权重的和。将像素值总和除以权重值总和得到最终这个像素点的像素值，代码如下：

```
gl_FragColor = sum / gaussianWeightTotal;
```

处理完所有像素点后，就得到了处理后的图像，这个Program只是做了一个横轴方向的工作，然后拿着这个Program的输出，再利用这个Program做一个纵轴方向的工作，等两遍都处理完毕之后，这幅图片的

磨皮工作就做好了。然后加上提亮、增加对比度、调整饱和度等细节就可达到一个整体美颜的效果。

9.4.4 动图贴纸效果器

对于一个成熟的录制视频软件来讲，除了美颜处理外，还需要给视频增加一些有趣的功能，最常见的就是卖萌、耍酷的动态贴纸。本节讨论如何实现动图贴纸效果器。

动图贴纸效果器也有很多种实现方法，其中一种实现方法是使用gif格式的图片来做动图。这种实现方法的优点是图片的压缩比大；缺点是gif格式的图片由于自身压缩算法的问题，在边缘部分会有白边，从而导致不好的用户体验。另外一种实现方法就是使用png序列图来做动图。虽然这种实现需要的图片容量比较大，但可以使用压缩工具在保证同等质量的前提下来提高压缩比，其优点是最终绘制出来的动图效果非常好。所以本书就是以png序列图的形式来实现动图贴纸效果器。

既然是png序列图，那么每一帧图片都是一张png图片，而将一张png图片如何进行解码并且渲染到视频上，在9.4.1节已经详细讲解过，而应用在动图贴纸上其实就是从单个渲染到多个渲染的过程，这可能没有什么技术含量，但是要想达到一个比较流畅以及快速的效果，可能不是那么简单的事情，所以本节会着重从如何优化整体体验的方面进行讲解。

但是，png序列图的渲染也不是来一帧视频帧就绘制一张png图片，而是依据不同动图贴纸的配置信息来安排的，动图贴纸也有自己的宽、高、fps等原始信息的配置。如之前所说的一样，在配置文件中应该描述这组png序列图的宽和高，因为一组序列图中的宽和高都是一致的，所以就有了前两个参数width和height。如果知道这组png序列图的名称（比如Say Hi）和png序列图的个数（比如6），那么png序列图的名称依次是Say Hi0，Say Hi1，…，Say Hi5，所以就有了中间的两个参数imageName和imageCnt。接下来就是fps的信息，即每张png图片持续的时间以imageIntervalInSec来表示（比如0.125就代表以125ms作为时间的间隔），最后一个参数就是这组序列图的持续时间，即序列图在视频上总共呈现的时间，使用durationInSec表示。一个整体的配置文件（JSON格式）如下：

```
{
    "width": 240,
```

```
    "height": 240,
    "durationInSec": 5,
    "imageName": "Say Hi",
    "imageCount": 6,
    "imageIntervalInSec": 0.125
}
```

动画设计人员可以使用AE设计好动画之后，导出为png序列图，然后在使用压缩工具（tinypng：https://tinypng.com/ ）压缩之后，再上传到服务器上。在上传过程中可以填写一些信息，上传之后，服务器端会将这些信息组装成为JSON信息并写入config.json文件中，并和png序列图打包到一起，最终压缩成为一个压缩包。客户端使用时先下载这个压缩包，然后进行解压缩，使其成为一个目录之后，找到config.json进行解析，就可以得到相应的原始信息。对于图片的完整路径，就是直接按照当前目录加上imageName和当前的png序列图的下标以及png的后缀名，然后根据fps进行解码，将相应的图片上传到显卡上，最终渲染到视频帧上。

说到优化体验，无非就是提升整体性能。在当前场景下，提升性能的方法就是缓存，即把解码之后上传到显存中所形成的纹理对象缓存起来，当下一次使用的时候，先判断是需要解码上传，还是可以直接从缓存池中取出来使用。所以缓存的构建就是本节的重点，下面详细介绍如何搭建一个纹理缓存系统。

首先封装出一个纹理对象的类来表示一个纹理，其中应该包括这个纹理的宽、高及纹理ID，以及当前对象的引用计数器。引用计数器大于0，代表当前纹理对象正在使用，不应该被外界看到并使用；引用计数器等于0，代表当前对象处于可用状态，可以交给外界使用。代码如下：

```
class GPUTexture {
private:
    int width;
    int height;
    GLuint texId;
    int referenceCount;
    GLuint createTexture(GLsizei width, GLsizei height);
public:
    GPUTexture();
    ~GPUTexture();
    int getWidth(){
        return width;
    };
```

```
    int getHeight(){
        return height;
    };
    GLuint getTexId(){
        return texId;
    };
    void init(int width, int height);
    void dealloc();
    void lock();
    void unLock();
    void clearAllLocks();
};
```

从以上代码中可以看到，提供给客户端代码的方法如下：初始化方法，用于创建出纹理对象；锁定与解锁方法，用于增加或者减少引用计数器；销毁方法，用于释放显卡中的纹理对象。由于篇幅的关系，就不再赘述了。

下面构造缓存系统。首先，以单例模式来构建缓存这个类，因为它在整个系统中只有一份，代码如下：

```
class GPUTextureCache {
private:
    GPUTextureCache();
    static GPUTextureCache* instance;
public:
    static GPUTextureCache* GetInstance();
    virtual ~GPUTextureCache();
};
```

然后应该有一个map类型的属性，用于盛放分配出来的所有GPUTexture对象。既然是map类型的属性，就必须有一个Key的分配规则，这个Key能唯一地标识纹理对象。前面在介绍纹理的章节中讲到过，宽、高和表示格式可以唯一地标识一个纹理对象，而在我们的系统中，表示格式使用的是默认的RGBA格式，所以仅使用宽和高就可以标识纹理对象，生成规则如下：

```
string GPUTextureCache::getQueueKey(int width, int height) {
    string queueKey = "tex_";
    char widthBuffer[8];
    sprintf(widthBuffer, "%d", width);
    queueKey.append(string(widthBuffer));
    queueKey.append("_");
    char heightBuffer[8];
```

```
    sprintf(heightBuffer, "%d", height);
    queueKey.append(string(heightBuffer));
    return queueKey;
}
```

生成了Key之后，那map的Value应该是什么类型的？应该是一个链表的形式。我们可以思考png序列图这个场景，使用第一张图片占用一个纹理对象之后，当遇到后续的图片还需要同样Key的纹理对象时，就需要一个链表来存放这个Key所代表的所有纹理对象，所以根据Key和Value就可构造出这个map对象，如下：

```
map<string, list<GPUTexture*> > textureQueueCache;
```

接下来是提供接口方法，用来从缓存系统中获取对应的纹理对象与将使用过后的纹理对象返还给缓存系统。先来看获取纹理对象，代码如下：

```
GPUTexture* GPUTextureCache::fetchTexture(int width, int height) {
    GPUTexture* texture = NULL;
    string queueKey = getQueueKey(width, height);
    map<string, list<GPUTexture*> >::iterator itor;
    itor = textureQueueCache.find(queueKey);
    if (itor != textureQueueCache.end()) {
        if ((itor->second).size() > 0) {
            texture = (itor->second).front();
            (itor->second).pop_front();
        } else {
            texture = new GPUTexture();
            texture->init(width, height);
        }
    } else {
        list<GPUTexture*> textureQueue;
        texture = new GPUTexture();
        texture->init(width, height);
        textureQueueCache[queueKey] = textureQueue;
    }
    return texture;
}
```

上述代码首先根据客户端传递过来的宽和高得到唯一标识的Key，然后拿这个值去map中寻找链表。如果找不到，则代表这个Key类型的纹理之前从未创建过，所以要创建一个链表，并以这个值为Key放入Map中，同时创建一个GPUTexture对象并返回。注意这里不要放到这个

链表中，因为如果放入到链表中，这个纹理对象就有可能会被外界所使用；如果按照这个Key找到对应的链表，就可以判断这个链表中是否有可用的元素，如果有则弹出，如果没有则创建并返回给客户端。接下来看如何将不再使用的纹理对象返还到缓存系统中，代码如下：

```
void GPUTextureCache::returnTextureToCache(GPUTexture* texture) {
    string queueKey = getQueueKey(texture->getWidth(), texture->getHeight());
    map<string, list<GPUTexture*> >::iterator itor;
    itor = textureQueueCache.find(queueKey);
    if (itor != textureQueueCache.end()) {
        (itor->second).push_back(texture);
    }
}
```

如上述代码所示，若客户端调用这个方法，则代表GPUTexture对象对于客户端来说无需再使用，所以在实现中也是根据这个纹理对象的宽和高构造出唯一标识的Key，然后在Map中找出对应的链表，并放到链表的末尾，而这个链表无形之中也是一个久未使用的数据结构的表示。如果对这个缓存系统所占用的显存有一个上限，那么就可以在这些链表的头部开始清理资源，即达到一个LRUCache的效果。这里就不再展示，读者可以自行完成。

这个缓存系统可以应用到png序列图效果器中，使得整个效果器不用频繁地分配和释放显存。同时，在效果器中可以再建立一层缓存，以避免频繁的解码操作，使用一个数组解码并上传到显卡中将GPUTexture对象分别存储起来，再按照顺序加入数组中，按照配置文件解析出fps信息，计算出当前要使用的Index后，就可以直接在数组中取出GPUTexture对象使用了，这就是一个整体的优化方案。

9.4.5　主题效果器

给一个视频增加主题，比如下雨天、老电影、Sunshine等主题，这在一些短视频社区App里是一项基本的功能。具体如何实现主题效果器，就是本节要讨论的内容。

对于主题效果器来讲，一般会分为以下处理流程：片头的布置、作者作品名的渲染、片中的主题效果、对视频源的处理，以及片尾的修饰预处理、这是一整个处理的过程。对于这个过程的描述，一般会使用配置文件实现，这样在App中就可以通过热更新来随意增加主题了。本节介绍主题效果器的协议配置，通过协议配置，会更加清楚整个主题需要哪些基本效果器，这其中最主要的一个效果器就是整个主题背景的展示（比如下雨天、飘雪或者阳光等）。下面先介绍主题背景效果器的实现。

背景效果器的实现分为两种。一种是使用粒子效果器实现。这种实现的优点是有比较好的性能；其缺点是设计人员和开发者的沟通成本比较高。一种是使用视频实现。这种实现的优点是实现简单，降低了设计人员和开发者的沟通成本；其缺点是对于性能来讲，并不是一种最好的实现方式。笔者本节会和大家一起讨论第二种实现，即使用视频实现主题效果器。

既然是一个视频，那么要有一个解码器来将这个主题视频一帧一帧地解码出来，然后按照这个视频的fps信息对应地渲染到原始视频帧上。解码的内容可以参看第4章，而渲染工作是这里介绍的重点，先看一帧主题视频的特征，如图9-7所示。

图9-7描述了视频主题的一帧图像，从图中可看到中间部分几乎接近黑色，而下雨天希望视频帧的中间部分（可能是人脸）可以全部展现出来，如何将这个视频的主题帧和原始视频帧进行叠加呢？可以使用前面介绍的滤色混合，这里使用查表的形式具体实现，即先使用Photoshop生成一个滤色混合的查找表，所有的混合模式都可避免直接计算，而通过查表的方式得到对应的值，这其实是在提升性能，以避免大量的计算，同时可以更加自由地控制从黑色到白色渐变的快速过程。根据不同的主题，设计人员可以直接调好这个滤色混合查找表，再把这个查找表的图片放入资源文件中，下雨天生成的滤色混合查找表图片如

图9-8所示。

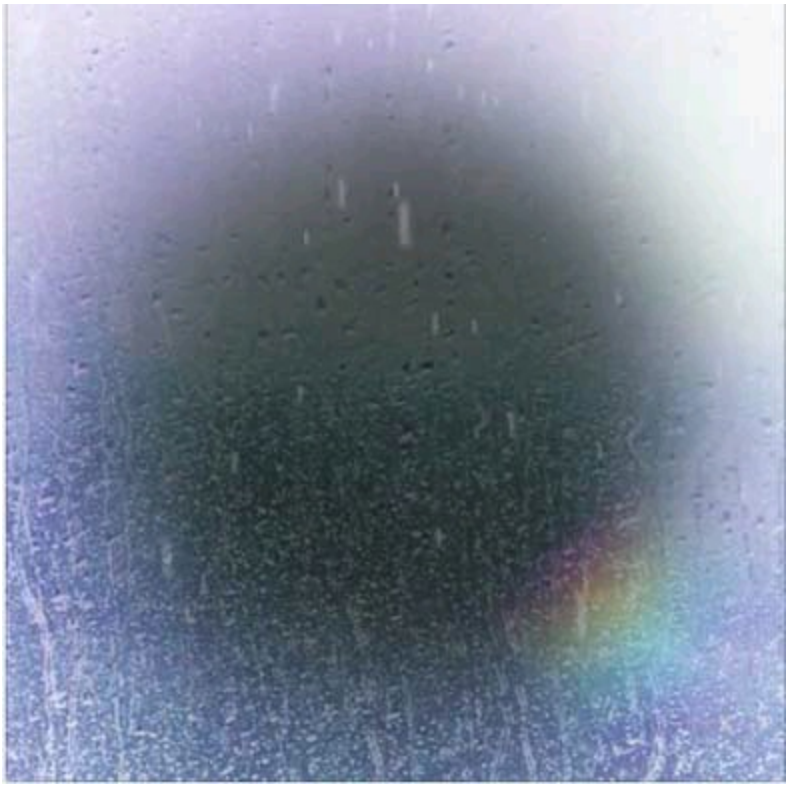


图　9-7

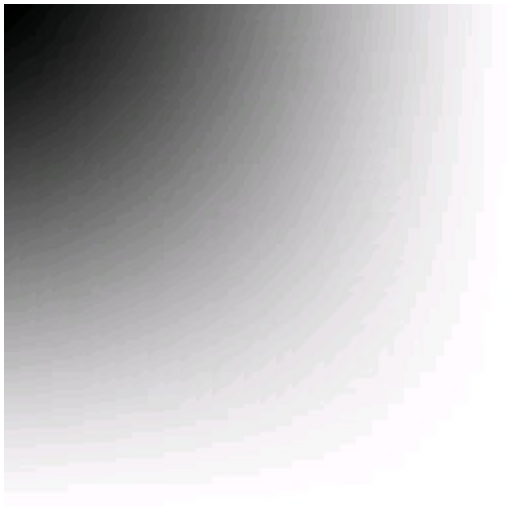


图　9-8

将图9-8的这个滤色混合的图片作为blendTexture传递给显卡，而FragmentShader中的具体处理代码如下：

```
precision lowp float;
varying highp vec2 textureCoordinate;
uniform sampler2D videoTexture;
uniform sampler2D themeTexture;
uniform sampler2D blendTexture;
void main()
{
    vec4 texel = texture2D(videoTexture, textureCoordinate);
    vec3 themeTexel = texture2D(themeTexture, textureCoordinate).rgb;
    texel.r = texture2D(blendTexture, vec2(themeTexel.r, texel.r)).r;
    texel.g = texture2D(blendTexture, vec2(themeTexel.g, texel.g)).g;
    texel.b = texture2D(blendTexture, vec2(themeTexel.b, texel.b)).b;
    gl_FragColor = vec4(texel.r, texel.r, texel.r, 1.0);
}
```

按照上述代码处理完毕之后，原始视频就和主题视频进行了混合，至此完成了视频主题效果器。接下来介绍主题文件的协议配置，开发者或者主题UI设计人员会将所用到的所有资源和这个配置文件打包到一个目录中，并压缩存储到服务器里。当客户端需要下载的时候，就将这个压缩包从服务器中下载下来，然后解压，解析配置文件，并根据配置文件中的描述构建出一个主题效果器的数据结构，以便进行后续渲染处理。

现在重点就是如何来描述主题协议。相较于上一小节的png序列图的配置文件，主题配置文件会更加复杂，所以可以使用XML文件实现。一般情况下，一个主题由多个效果器构成，从头到尾可以分为片头、作者作品名展示、片中主题效果、片尾处理等。而每一个效果器都会有一个开始作用时间和结束作用时间，以便于在渲染某一帧视频帧的时候，可以将视频帧的时间戳作为参考，从而决定这一帧视频帧到底要经过哪几个效果器的处理，这样就可以确定所有效果器都应该有两个参数了，即sequenceIn（代表开始作用时间）和sequenceOut（代表结束作用时间）。每一个效果器都应该有一个全局唯一的名字作为自己的标识，所以应该还有一个filterName来作为自己的效果器名称。此外，每个效果器都应该有自己独有的一些配置变量，比如文字效果器里需要标注好文字的大小、颜色及位置等信息。最终将各种效果器配合使用，从而生成一个主题效果器。先来看一个主题的配置文件，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<theme cover="icon.png" themeName="下雨天">
    <filterList>
        <filter filterName="header_scene" sequenceIn="0" sequenceOut="4500">
            <param id="video path" value="head_fade_scene.mp4" type="8"/>
            <param id="screen fade out in secs" value="3000" type="1"/>
```

```
        </filter>
        <filter filterName="text_scene" sequenceIn="600" sequenceOut="3500">
            <param id="scene width" value="480" type="1"/>
            <param id="scene height" value="480" type="1"/>
            <param id="scene text left" value="40" type="1"/>
            <param id="scene text width" value="400" type="1"/>
            <param id="scene text top" value="240" type="1"/>
            <param id="scene text height" value="28" type="1"/>
            <param id="scene text alignment" value="0" type="1"/>
            <param id="scene text color" value="25, 85, 92, 255" type="5"/>
            <param id="scene text shadow radius" value="1" type="2"/>
            <param id="scene text shadow x offset" value="1" type="2"/>
            <param id="scene text shadow y offset" value="1" type="2"/>
            <param id="text shadow color" value="102, 102, 102, 255" type="5'
            <param id="scene text size" value="24" type="1"/>
            <param id="scene text type" value="1" type="1"/>
            <param id="scene fade in end time" value="1410" type="1"/>
            <param id="scene fade out start time" value="3000" type="1"/>
        </filter>
        <filter filterName="blur_scene" sequenceIn="2000" sequenceOut="4500">
        </filter>
        <filter filterName="video_scene" sequenceIn="10" sequenceOut="-10">
            <param id="black board path" value="rain_overlay.mp4" type="8"/>
            <param id="map pic path" value="overlay_screen.png" type="8"/>
            <param id="amaro map pic path" value="amaroMap.png" type="8"/>
        </filter>
        <filter filterName="blur_scene" sequenceIn="-3000" sequenceOut="0">
            <param id="blur scene ascend flag" value="1" type="3"/>
        </filter>
        <filter filterName="trailer_scene" sequenceIn="-3000" sequenceOut="0'
            <param id="duration" value="1.5" type="2"/>
            <param id="scene width" value="480" type="1"/>
            <param id="scene height" value="480" type="1"/>
            <param id="scene text left" value="40" type="1"/>
            <param id="scene text top" value="270" type="1"/>
            <param id="scene text width" value="400" type="1"/>
            <param id="scene text height" value="40" type="1"/>
            <param id="scene text color" value="250, 247, 249, 255" type="5"/
            <param id="text shadow radius" value="1" type="2"/>
            <param id="text shadow x offset" value="1" type="2"/>
            <param id="text shadow y offset" value="1" type="2"/>
            <param id="text shadow color" value="102, 102, 102, 255" type="5'
            <param id="scene text size" value="26" type="1"/>
            <param id="scene text type" value="1" type="1"/>
            <param id="scene text alignment" value="0" type="1"/>
            <param id="scene overlay path" value="trailer.png" type="8"/>
        </filter>
    </filterList>
</theme>
```

我们看一下这个配置文件的意义，这个主题的名称叫下雨天，封面图是icon.png图片，具体主题里包含的效果器是filterlist这个标签里包含的。首先是片头效果器，使用的视频是将当前目录里的

head_fade_scene.mp4作为视频源，作用时间从0开始，到4500ms结束。然后，从600ms到3500ms这个时间段内会把作者名以及作品名以文字效果器的形式绘制到视频上，具体文字的颜色、大小、绘制区域都在参数中。接下来从2000ms到4500ms这个时间段内会有一个模糊的效果器产生作用，默认从非常模糊到逐渐清晰。以上3个效果器一起构造了这个片头效果器，再在整个视频中间使用视频附加效果器，使用的视频源是rain_overlay.mp4视频源。接着，从片尾减去3000ms（这就是配置里-3000的含义），虽然到片尾也会有一个模糊效果器产生作用，但是是从清晰逐渐过渡到模糊（这就是标签内部参数ascend flag的含义），并且在最后有一个片尾效果器，即在一个带有logo的图片上绘制作品的名字和作者的名字，具体的文字以及logo的图片配置在这个标签内部的参数中，这个效果器和之前的模糊效果器共同构造成为一个片尾效果器。这就是整个主题配置文件所描述的场景，读者可以参看本章代码目录中的下雨天视频主题。

9.5　本章小结

本章首先介绍了图像的基本处理，然后讨论了一些复杂的图像处理算法，最后两节从两种技术架构的角度分别介绍了如何使用图像处理的技术，读者可以根据自己的系统所使用的技术架构来选用技术实现。本章结束之后，在第7章最后留下的两个问题，其实就都解决了。接下来第10章会把第8章和第9章学到的内容应用到第7章的视频录制中，最终会构造出一款专业的视频录制应用。

第10章　专业的视频录制应用实践

本章会在第7章的基础上，再结合第8章和第9章的内容，完成一个专业视频的录制应用。一个视频录制的应用分为三部分：录制视频部分、编辑部分和离线保存部分。当然，如果是直播应用的推流端，那么只有第一部分的内容。其中第一部分的输入是摄像头和麦克风，而第二部分和第三部分的输入是解码器。解码器在第3章中已经讲解过，对于解码音频来说，这种解码的实现没有任何性能问题，但是对于解码视频（尤其是高清视频）来说，其性能就会成为瓶颈。

10.1 视频硬件解码器的使用

本节会介绍在Android平台和iOS平台上硬件解码器的使用，并基于第5章视频播放器的项目继续开发。待集成好硬件解码器之后，读者可以对比CPU和内存的使用情况，以及耗电量的情况。对于任何H264的解码器而言，都要将SPS和PPS信息传递给解码器。在第3章使用FFmpeg解码视频的时候，我们并没有显式设置SPS和PPS信息，是因为在FFmpeg内部自己做了设置，但是对于硬件解码器来讲，开发者必须手动设置。另外，使用FFmpeg解码出来的视频帧是以YUV格式的表示形式存储于内存中的，但是对于硬件解码器来讲，一般都是直接解码到显存中，便于后续的处理与渲染。所以下面先来看如何从视频流中解析SPS和PPS信息。

10.1.1 初始化信息准备

请读者回忆第7章是如何将编码H264之后的SPS和PPS信息封装到视频流中去的，就是将SPS和PPS以一定的格式写入视频流的编码器上下文的extradata这个属性中。而在解码中输入是一个视频流，然后要得到视频流里的SPS和PPS信息，这与编码过程正好是一个逆过程。将一个视频流（本地文件或者网络资源）最终显示出来，要经历协议解析、封装格式的解析、解码、音视频同步、渲染这一系列的步骤，第5章中已经完成了一个视频播放器，而本节的目的是将解码环节替换为硬件解码，以提升整个App的性能。

这里的协议解析和封装格式的解析还是使用FFmpeg框架来完成，所以要实现硬件解码器，就要先写一个子类来继承自原来的VideoDecoder类，然后重写openVideoStream方法。这个方法原来在父类的职责是找出第一个视频流，然后拿出视频流里的解码器上下文，进而打开软件解码器。在重写了之后，需要打开的就是硬件解码器，所以要先得到解码器上下文，然后根据解码器上下文中的extradata属性解析出SPS和PPS信息，代码如下：

```
-(void) parseH264SequenceHeader(uint8_t* extra_data,
            uint8_t** bufSPS, int* sizeSPS
            uint8_t** bufPPS, int* sizePPS) {
    int spsSize = (extra_data[6] << 8) + extra_data[7];
    *sizeSPS = spsSize;
    *bufSPS = &extra_data[8];
    int ppsSize = (extra_data[8+spsSize+1] << 8) + extra_data[8+spsSize+2];
    *bufPPS = &extra_data[8 + spsSize + 3];
    *sizePPS = ppsSize;
}
```

在上述代码中，传入参数就是解码器上下文中的extradata属性，这个方法会将解析出来的SPS部分的数据放入bufSPS中，数据的大小则放入sizeSPS变量中；将PPS部分的数据放入bufPPS中，大小则放入sizePPS变量中。具体实现代码恰好是7.5.2节将SPS和PPS封装到extradata属性中的一个逆过程，即取出下标为6和下标为7的元素并按照高低位组合为SPS的大小，记为sizeSPS，这样从下标为8的位置开始长度为sizeSPS的内存区域表示的就是SPS的信息。SPS结束之后的两个下标按照高低位组合就可以形成PPS的大小，记为sizePPS，而从这两个代表PPS的size的

下标开始向后数，长度为sizePPS的内存区域表示的就是PPS的信息。大家可以参考第7章中的逆过程以加深理解。

待SPS和PPS信息解析出来之后，进入iOS平台和Android平台所提供的硬件解码器的学习之前，先来回顾H264的两种封装格式：一种是Annexb的封装格式，另外一种是mp4（AVCC）的封装格式。在第7章最后的封装步骤中，我们曾经手动将Annexb格式的H264码流转换为MP4封装格式的码流，这两种格式的表示如图10-1所示。

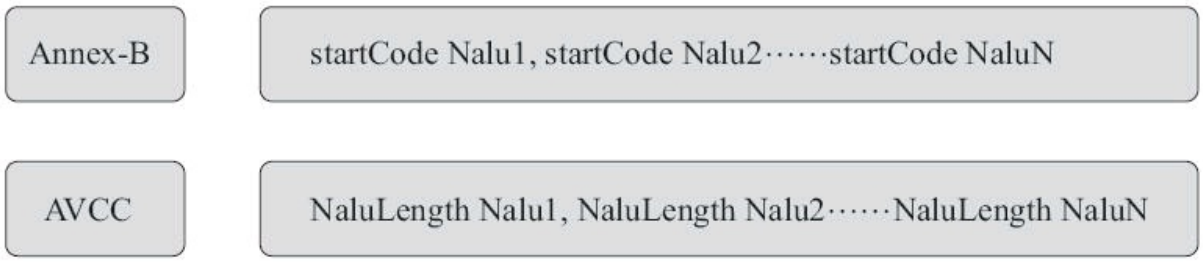


图　10-1

这里需要注意的是，AVCC格式中，Nalu的Length需要四个字节的大尾端（big endian）顺序，可以参考7.5.2节。在解码的过程，需知道解码器到底要传入哪一种格式，并且还要了解我们从FFmpeg中得到的H264的AVPacket是哪一种格式。对于第一个问题，不同的平台是不一致的，VideoToolbox要求的是AVCC格式的输入，而MediaCodec要求的却是Annexb格式的输入。但是FFmpeg解封装出来的AVPacket通常是AVCC格式的，对于不同平台开发者该如何处理？下面我们分别进行讲解。

10.1.2 VideoToolbox解码H264

SPS和PPS里记录着编码器编码视频所用的profile、level、图像的宽和高、deblock滤波器等信息，而解码器要想正确初始化且解码成功，必须知道编码器使用的这些原始信息。因此，下面分三部分来介绍硬件解码器的使用，首先根据SPS和PPS信息来初始化VideoToolbox，然后调用硬件解码器解码出原始格式的数据，最终销毁这个解码器。

1.VideoToolbox的初始化

前面多次提到，要想使用iOS平台提供给开发者的多媒体API，就必须初始化FormatDescription来描述编码器具体的格式信息，这里也一样。首先要利用SPS信息和PPS信息构造出一个CMVideoFormatDescriptionRef实例来描述编码器编码的格式信息，代码如下：

```
uint8_t* bufSPS = 0;
uint8_t* bufPPS = 0;
int sizeSPS = 0;
int sizePPS = 0;
this->parseH264SequenceHeader(videoCodecCtx->extradata,
    &bufSPS, &sizeSPS, &bufPPS, &sizePPS);
uint8_t*  parameterSetPointers[2] = {bufSPS, bufPPS};
size_t parameterSetSizes[2] = {sizeSPS, sizePPS};
CMVideoFormatDescriptionRef formatDesc;
OSStatus status = CMVideoFormatDescriptionCreateFromH264ParameterSets(
    kCFAllocatorDefault, 2, (const uint8_t *const*)parameterSetPointers,
    parameterSetSizes, 4, &formatDesc);
```

从以上代码中可以看到，会调用SPS和PPS的函数解析出SPS信息和PPS信息，然后根据SPS信息和PPS信息构造对应的数据结构，最终调用API构造出formatDesc变量。接下来利用这个变量就可以去初始化硬件解码器，代码如下：

```
// 1:解码完毕的回调函数
VTDecompressionOutputCallbackRecord callBackRecord;
callBackRecord.decompressionOutputCallback =
    decompressionSessionDecodeFrameCallback;
callBackRecord.decompressionOutputRefCon = (__bridge void *)self;
// 2:解码输出的CVPixelBuffer的目标格式
```

```
NSDictionary *destinationImageBufferAttributes =
    [NSDictionary dictionaryWithObjectsAndKeys:[NSNumber numberWithBool:YES],
        (id)kCVPixelBufferOpenGLESCompatibilityKey, nil];
// 3:创建解码器
OSStatus status =  VTDecompressionSessionCreate(NULL, _formatDesc, NULL,
    (__bridge CFDictionaryRef)(destinationImageBufferAttributes),
    &callBackRecord, &_decompressionSession);
if(status != noErr) {
    NSLog(@"Video Decompression Session Create: \t failed...");
}
```

上述代码的第一部分是构造一个解码器解码完毕之后的回调函数，用于接受解码之后的原始视频帧。由于解码器解码之后的视频帧原始数据是以CVPixelBuffer数据结构来存放的，所以代码的第二部分用来指定解码之后的输出格式。由于我们最终要使用OpenGL ES来做渲染，所以这里只设置OpenGL ES兼容性的属性为YES。当然也可以设置输出格式，例如，若解码器解码到UIImageView上显示，就需要设置PixelFormat为BGRA的格式；若在解码到UIImageView上显示的情况下，OpenGL ES的兼容性就必须设置为NO。代码的最后一部分就是根据前面的三个信息（FormatDesc、Callback、bufferAttr）来构造解码器，再判断status的状态来看解码器是否创建成功。

2.VideoToolbox解码过程

解码过程需要重写父类的decodeVideo方法，然后在这个方法中使用硬件解码器来解码视频帧。VideoToolbox需要开发者传入AVCC格式的H264视频帧，如果是一个裸的H264文件，封装格式就是Annexb格式的，这时就需要手动转换为AVCC的格式，再传递给解码器。但是，在FFmpeg中经过解封装（Demux）之后的AVPacket就是AVCC的封装格式，因此不需要进行转换。了解了格式之后，就来看一下VideoToolbox接受的具体结构体类型，即CMSampleBuffer、CMSampleBuffer是由CMBlockBuffer、FormatDesc、TimeInfo共同组成的，CMBlockBuffer中存储的就是实际解码之前的数据，可以从AVPacket的data变量中得到。接下来构造CMBlockBuffer结构体，代码如下：

```
CMBlockBufferRef blockBuffer = NULL;
uint8_t* data = packet.data;
int blockLength = packet.size;
OSStatus status = CMBlockBufferCreateWithMemoryBlock(NULL, data,
                                        blockLength,
                                        kCFAllocatorNull, NULL,
                                        0,
```

```
                                        blockLength,
                                        0, &blockBuffer);
if(status != kCMBlockBufferNoErr) {
    NSLog(@"BlockBufferCreation: failed...");
}
```

接下来继续构造CMSampleBuffer结构体，FormatDesc就是最开始通过SPS信息和PPS信息构造出来的结构体，而TimeInfo可以由AVPacket结构体中的PTS和DTS变换得到，所以构造结构体CMSampleBuffer的代码如下：

```
int64_t presentationTimeStamp = av_rescale_q(packet.pts,
    formatCtx->streams[videoStreamIndex]->time_base, AV_TIME_BASE_Q);
int64_t decompressionTimeStamp = av_rescale_q(packet.dts,
    formatCtx->streams[videoStreamIndex]->time_base, AV_TIME_BASE_Q);
int64_t duration = packet.duration;
if(!duration){
    duration = 1000 / [self getVideoFPS];
}
int32_t timeSpan = 1000;
CMSampleTimingInfo timingInfo;
timingInfo.presentationTimeStamp = CMTimeMake(presentationTimeStamp, timeSpar
timingInfo.decodeTimeStamp = CMTimeMake(decompressionTimeStamp, timeSpan);
timingInfo.duration = CMTimeMake(duration, timeSpan);
const size_t sampleSize = blockLength;
status = CMSampleBufferCreate(kCFAllocatorDefault,
                            blockBuffer, true, NULL, NULL,
                            formatDesc, 1, 0, &timingInfo, 1,
                            &sampleSize, &sampleBuffer);
if(status != noErr) {
    NSLog(@" SampleBufferCreate: failed...");
}
```

上述代码执行完毕之后，就顺利地将AVPacket转换成了CMSampleBuffer结构体。接着就可以调用解码方法了。对于解码方法，这里要着重说明一下，解码方法并不会直接将解码后的原始数据返回给开发者，而是通过第一步中设置的callback方法将解码之后的数据提取出来，并且共有两种解码模式：一种是同步方式，另一种是异步方式。这两种模式代表的意义是这个解码API的调用是否等待这一帧解码完毕之后再返回。先来看如何调用解码API，代码如下：

```
VTDecodeFrameFlags flags = kVTDecodeFrame_EnableAsynchronousDecompression;
VTDecodeInfoFlags flagOut;
status = VTDecompressionSessionDecodeFrame(_decompressionSession,
    sampleBuffer, flags, &sampleBuffer, &flagOut);
```

```
if (status == noErr) {
    VTDecompressionSessionWaitForAsynchronousFrames(_decompressionSession);
}
videoFrame.position = presentationTimeStamp / 1000000.0;
videoFrame.duration = (float)duration / 1000.0;
CFRelease(sampleBuffer);
if (NULL != blockBuffer) {
    CFRelease(blockBuffer);
    blockBuffer = NULL;
}
return videoFrame;
```

这里的代码比较简单，解码方法的第一个参数是解码器会话，第二个参数就是前面构造的sampleBuffer，第三个参数是解码模式，第四个参数是要传递到回调函数的变量，最后一个参数可以传空。唯一需要注意的是，这里使用的是异步解码模式，但是在解码函数结束之后，调用了WaitForAsync的方法等待解码过程的结束。当然，将解码之后的数据封装到videoFrame中是在回调函数中进行操作的，回调函数的实现稍后再展示，代码的最后会释放掉blockBuffer与sampleBuffer，并将videoFrame返回。回调函数代码如下：

```
void decompressionSessionDecodeFrameCallback(void *decompressionOutputRefCon,
            void *sourceFrameRefCon,
            OSStatus status,
            VTDecodeInfoFlags infoFlags,
            CVImageBufferRef imageBuffer,
            CMTime presentationTimeStamp,
            CMTime presentationDuration)
{
    if (status != noErr || !imageBuffer) {
        NSLog(@"Error decompressing frame …");
        return;
    }
    NSUInteger frameWidth = (NSUInteger)CVPixelBufferGetWidth(imageBuffer);
    NSUInteger frameHeight = (NSUInteger)CVPixelBufferGetHeight(imageBuffer);
    videoFrame = [[VideoFrame alloc] init];
    videoFrame.width = frameWidth;
    videoFrame.height = frameHeight;
    videoFrame.imageBuffer = (__bridge id)imageBuffer;
}
```

从以上代码可以看到，回调函数中第一个参数实际上就是构造硬件解码器传递进去的callback的上下文，也就是这个对象本身；第二个参数则是调用解码函数时传递进去的第四个参数；第三个参数status代表这一帧是否解码成功，其中最重要的参数是imageBuffer与时间戳信息，从以上代码中可以看到将它们封装到了结构体中，并返回给了客户端代

码。

这样就完成了解码的操作，读者可以与使用VideoToolbox做硬件编码的过程进行对比，体会编码和解码两个过程的相同点和不同点。

3.VideoToolbox的销毁

最后的销毁阶段需要重写父类的closeVideoStream方法，这里会销毁为硬件解码器分配的资源，代码如下：

```
if(_formatDesc){
    CFRelease(_formatDesc);
}
if(_decompressionSession){
    VTDecompressionSessionInvalidate(_decompressionSession);
    CFRelease(_decompressionSession);
}
```

至此，硬件解码器在iOS平台的实现就完成了，但是细心的读者可能会发现，硬件解码器解码出来的数据类型实际上是一个CVPixelBuffer，而这与之前软件解码器解码出来的YUV420P的数据是不一致的，这就导致了在后续的渲染阶段肯定也会出现不一致。的确，在后续的VideoOutput中，要改动代码以适配数据帧类型是CVPixelBuffer类型的渲染。具体的渲染方法与第6章中介绍的将摄像头采集出来的一帧CVPixelBuffer用OpenGL ES渲染到View上是一致的。但是可能也有读者会想到另外一种设计方案，即在解码结束之后，直接锁定这个PixelBuffer，根据宽、高及对齐方式将YUV数据读取到内存中，然后再封装到VideoFrame中，这样渲染端就不用去做任何改动，这也是面向对象编程的优雅实践。这种想法十分有道理，其实笔者最开始也是这么做的，但是锁定PixelBuffer后，将YUV数据读取到内存中的效率实在太低，整体的性能比软件解码的性能提升不了多少，所以最终还是去改动VideoOutput中的一个模块来完成整个流程，以达到最高性能的提升。

10.1.3 MediaCodec解码H264

在Android平台上提供给开发者的硬件解码器API接口是MediaCodec，调用硬件解码器分为三个阶段，分别是初始化阶段、解码阶段和销毁资源阶段。初始化阶段需要使用SPS、PPS及输出纹理ID来配置一个MediaCodec实例；解码阶段则需要将FFmpeg解封装出来的AVPacket结构体转换之后再给MediaCodec进行解码，解码之后的数据帧已经存放到了第一步的输出纹理ID上；当不再需要硬件解码器的时候，就销毁相关的资源。最终解码出来的数据帧就是一个纹理ID，而不再是YUV的内存数据，这就要求我们的视频帧队列不再是原始内存中的队列，而应该是一个可以存储纹理ID的显存队列，并且最好是一个循环队列，因为这样可以避免纹理ID频繁地开辟与销毁的代价，同时VideoOutput模块会去适配直接渲染一个纹理ID。

1.纹理循环队列

循环队列可以直接使用循环链表来实现。首先，规定链表中存放的元素应该是什么，通常包括纹理ID、视频帧的时间戳，以及视频帧的宽和高等信息。所以每个元素结构体的定义如下：

```
typedef struct FrameTexture {
    GLuint texId;
    float position;
    int width;
    int height;
} FrameTexture;
```

从以上代码可以看到，这个结构体很简单，但是要注意，创建这个结构体的地方要负责分配出一个纹理ID赋值给这个结构体的texId属性，当释放这个结构体对象的时候，也要主动删除掉这个纹理ID所占用的显存资源。基于这个元素，创建出循环链表中的节点，代码如下：

```
typedef struct FrameTextureNode {
    FrameTexture *texture;
    struct FrameTextureNode *next;
    FrameTextureNode(){
        texture = NULL;
        next = NULL;
```

```
    }
} FrameTextureNode;
```

定义好Node之后，接下来就可以构造这个循环链表了，在构造循环链表之前先明确对外界提供的接口方法。

初始化方法，由于这个方法要分配纹理对象，所以需要在一个OpenGL ES上下文线程中进行调用，代码如下：

```
void init(int width, int height, int queueSize);
```

在这个方法的实现中，初始化长度为length的循环链表，pushCursor指向第一个节点，pullCursor也指向第一个节点，使最后一个节点（tail）的Next指向第一个节点（head），这样就构造好了循环链表。当解码器解码出一帧视频帧后，要先锁定住pushCursor指向的元素，然后将这一帧视频帧拷贝到这个元素的纹理ID上，最后解锁pushCursor指向的这个元素，并让pushCursor指向自己的下一个节点，以实现push从一个元素到这个队列中的功能。其中上锁和解锁的接口方法如下：

```
FrameTexture* lockPushCursorFrameTexture();
void unLockPushCursorFrameTexture();
```

当从队列中取出元素的时候，AVSynchronizer组件会从队列中获取队头的一帧视频帧，若返回值大于0，则代表获取到正确的元素；若小于0，则代表队列处于丢弃状态。注意，这里不是弹出操作，代码如下：

```
int front(FrameTexture **frameTexture);
```

当确定要使用这一帧视频帧的时候，就要从队列中弹出这一帧视频帧，AVSynchronizer组件调用如下接口来完成pullCursor的移动操作：

```
int pop();
```

获取队列大小以及丢弃队列等接口的代码如下：

```
int getValidSize();
void abort();
```

最后在析构函数中遍历所有的元素，取出元素中的纹理ID并销毁，以及销毁元素本身。

了解了循环队列中存储的元素以及接口方法，在后续使用中就会比较清晰。接下来我们讲解MediaCodec这个硬件解码器的具体使用。

2.MediaCodec的初始化

从整个系统的扩展性考虑，下面编写一个解码器的子类，该子类继承自原VideoDecoder类，用于完成硬件解码的功能，在子类中重写父类的openVideoStream方法，用父类中的VideoCodecContext的extradata字段解析出SPS和PPS信息，并存储为全局变量作为备用。然后在OpenGL ES的线程中创建一个纹理ID，将这个纹理ID、SPS、PPS以及宽高传递给Java层，由Java层创建并配置MediaCodec实例。由于整个播放器业务逻辑都是在Native层开发的，所以这里比较复杂的就在于将Native层的数据对象传递到Java层，再由Java层调用MediaCodec的API来进行配置。我们来看Native层构造对象与调用Java的过程，代码如下：

```
int decodeTexId = textureFrameUploader->getDecodeTexId();
uint8_t* bufSPS = 0;
uint8_t* bufPPS = 0;
int sizeSPS = 0;
int sizePPS = 0;
this->parseH264SequenceHeader(extra_data, &bufSPS, &sizeSPS,
        &bufPPS, &sizePPS);
jbyteArray sps = env->NewByteArray(sizeSPS);
env->SetByteArrayRegion(sps, 0, sizeSPS, (jbyte*) bufSPS);
jbyteArray pps = env->NewByteArray(sizePPS);
env->SetByteArrayRegion(pps, 0, sizePPS, (jbyte*) bufPPS);
jmethodID createVideoDecoderFunc = env->GetMethodID(jcls,
        "createVideoDecoderFromNative", "(III[B[B]Z"];
bool suc = (bool) env->CallBooleanMethod(obj, createVideoDecoderFunc,
        width, height, decodeTexId, sps, pps);
if (!suc) {
    LOGE("Create MediaCodec decoder failed, use FFMPEG decoder instead");
}
env->DeleteLocalRef(sps);
env->DeleteLocalRef(pps);
```

其中，textureFrameUploader是一个维护OpenGL ES上下文线程的封装类创建的实例对象，由这个对象来维护输出纹理ID。要注意的是，这个纹理ID的格式不是普通的GL_TEXTURE_2D类型，而是特殊GL_TEXTURE_EXTERNAL_OES类型，这种纹理类型在前面讲解Camera采集图像的时候已经介绍过，此处不再赘述。

当然，这段代码必须放到一个JVM线程中去执行，否则从Native层是不可以调用Java层的代码的，具体如何附加（Attach）到一个JVM线程中去，有一种常用的写法，代码如下：

```
JNIEnv *env;
int status = 0;
bool needAttach = false;
status = g_jvm->GetEnv((void **) (&env), JNI_VERSION_1_4);
if (status < 0) {
    if (g_jvm->AttachCurrentThread(&env, NULL) != JNI_OK) {
        LOGE("%s: AttachCurrentThread() failed", __FUNCTION__);
        return false;
    }
    needAttach = true;
}
// Do Something
if (needAttach) {
    if (g_jvm->DetachCurrentThread() != JNI_OK) {
        LOGE("%s: DetachCurrentThread() failed", __FUNCTION__);
    }
}
```

上述代码的中间部分（注释的Do Something）就可以用来调用Java的代码。接下来看Java层如何构造与配置MediaCodec实例的。它会使用解码格式"video/avc"与宽和高来创建一种媒体格式，而且最重要的是，将媒体格式的csd-0和csd-1设置成为sps和pps，然后将传递过来的纹理ID构造成为一个SurfaceTexture，并最终构造为一个Surface，再用这两个对象来配置这个MediaCodec解码器。代码如下：

```
public boolean CreateVideoDecoder(int width, int height, int texId,
            byte[] sps, byte[] pps) {
    m_format = MediaFormat.createVideoFormat("video/avc", width, height);
    m_format.setByteBuffer("csd-0", ByteBuffer.wrap(sps));
    m_format.setByteBuffer("csd-1", ByteBuffer.wrap(pps));
    try {
        m_surfaceTexture = new SurfaceTexture(texId);
        m_surfaceTexture.setOnFrameAvailableListener(this);
```

```
            m_surface = new Surface(m_surfaceTexture);
            m_decoder = MediaCodec.createDecoderByType("video/avc");
            m_decoder.configure(m_format, m_surface, null, 0);
            m_decoder.start();
        } catch (Exception e) {
            Log.e(TAG, "" + e.getMessage());
            release();
            return false;
        }
        return true;
    }
```

从以上代码可以看到，如果配置过程中出现异常，则会调用release方法将所分配出的资源全部释放掉，并返回false，代表创建硬件解码器失败；如果成功，则返回true，代表配置好了MediaCodec实例。当配置好硬件解码器之后，接下来就是实际的解码过程，下面会进行讨论。

3.MediaCodec解码过程

实际的解码过程就是如何将输入转换为输出。首先来看MediaCodec允许如何输入，在6.3.2节讲解过MediaCodec的原理图（见图6-12），要想给MediaCodec输入，需要将解码器的变量inputBuffers取出，然后在每次将H264视频帧给解码器之前，取出对应的inputBuffer的Index，并将Annexb格式的H264数据放到这个buffer中。这部分代码主要是在Java层，代码如下：

```
    byte[] h264Data;
    int dataSize;
    final int inputBufIndex = m_decoder.dequeueInputBuffer(TIMEOUT_USEC);
    if (inputBufIndex >= 0) {
        ByteBuffer inputBuf = m_decoderInputBuffers[inputBufIndex];
        inputBuf.clear();
        inputBuf.put(h264Data, 0, dataSize);
        m_decoder.queueInputBuffer(inputBufIndex, 0, inputSize, timeStamp, 0);
    }
```

其中，h264Data和dataSize这两个参数由Native层传递过来，最终调用的queueInputBuffer方法代表将数据输入给了解码器。

明确了如何提供输入之后，还需要明确输入的数据是什么格式的，这里主要是C++层的逻辑，MediaCodec要求输入的H264封装格式是Annexb格式的，而FFmpeg解封装出来的AVPacket结构体是AVCC格式的，所以这里要做一个转换，使其符合MediaCodec要求的输入格式。

AVCC封装格式转换为Annexb封装格式的代码如下：

```
void MediaCodecVideoDecoder::convertPacket(AVPacket* packet) {
    uint8_t* data = 0;
    int pos = 0;
    long sum = 0;
    uint8_t header[4];
    header[0] = 0;
    header[1] = 0;
    header[2] = 0;
    header[3] = 1;
    while (pos < packet->size) {
        data = packet->data+pos;
        sum = data[0] << 24 + data[1] << 16 + data[2] << 8 + data[3];
        memcpy(data, header, 4);
        pos += (int)sum;
        pos += 4;
    }
}
```

由于一帧视频帧里可能有多个Nalu，所以这里有一个while循环，把所有Nalu中开始的length换为StartCode，这样就将MP4（AVCC）封装格式的H264数据转换为了Annexb封装格式，除了这种手动转换的方法以外，FFmpeg还提供了一种类型的Filter，即h264_mp4toannexb这个bit stream filter，使用这个Filter也可以达到同样的效果。代码如下：

```
// 1:初始化h264_mp4toannexb这个Filter,注意在编译FFmpeg的时候打开这个开关
AVBitStreamFilterContext* bsfc = av_bitstream_filter_init("h264_mp4toannexb")
// 2:转换格式
AVPacket newpacket;
av_init_packet(&newpacket);
int ret = av_bitstream_filter_filter(bsfc, videoCodecContext, NULL,
        &newpacket.data, &newpacket.size, pkt.data, pkt.size,
        pkt.flags & AV_PKT_FLAG_KEY);
if (ret >= 0) {
    // 转换成功,可以取出newpacket中的data数据
}
// 3:销毁这个Filter
av_bitstream_filter_close(bsfc);
```

接下来就是如何获得解码器解码出来的视频帧。还记得在第一阶段配置MediaCodec的时候，创建的SurfaceTexture设置了一个OnFrameAvailable的监听器吗？即待解码器解码出一帧视频帧之后，就会调用这个监听器中的onFrameAvailable方法，而这个方法中的处理就是最关键的地方。所以在解码过程中将H264数据输入给解码器（将

inputBuffer放入MediaCodec的输入队列中）之后，就要等待（wait）在这里，代码如下：

```
m_decoder.queueInputBuffer(inputBufIndex, 0, inputSize, timeStamp, 0);
synchronized (m_frameSyncObject) {
    try {
        m_frameSyncObject.wait(TIMEOUT_MS);
        if (!m_frameAvailable) {
            Log.e(TAG, "Frame wait timed out!");
            return false;
        }
    } catch (InterruptedException ie) {
        Log.e(TAG, "" + ie.getMessage());
        ie.printStackTrace();
        return false;
    }
}
```

当监听器的onFrameAvailable函数被调用的时候，就要发出signal信号，让解码线程继续运行，代码如下：

```
public void onFrameAvailable(SurfaceTexture st) {
    synchronized (m_frameSyncObject) {
        m_frameAvailable = true;
        m_frameSyncObject.notifyAll();
    }
}
```

从以上代码中可以看到，在这个方法中会发出signal信号，让解码线程继续运行，而在Native层会由解码线程转到textureUploader线程（这是一个OpenGL ES的渲染线程）进行调用Java层的代码来更新纹理，Java层更新纹理代码如下：

```
public long updateTexImage() {
    try {
        m_surfaceTexture.updateTexImage();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return m_timestampOfCurTexFrame;
}
```

这个方法执行完毕之后，硬件解码器解码成功的数据就被更新到

textureUploader中维护的输出纹理ID上，至此就完成了整个解码过程。解码过程就介绍到这里，由于涉及Native层代码和Java层代码的交互，所以整个流程比较复杂，读者可以参考代码仓库中的示例代码加深理解。

4.MediaCodec的销毁

这个阶段需要重写closeVideoFrame方法，调用父类方法之后，再调用Java层的销毁MediaCodec的方法，这里展示Java层的代码如下：

```
if(m_decoder) {
    if(m_inputBufferQueued){
        m_decoder.flush();
    }
    m_decoder.stop();
    m_decoder.release();
}
```

从以上代码可以看到，首先判断一个布尔型变量m_inputBufferQueued。这个变量的意义是，当解码线程将H264的数据输送给了解码器，但是解码器还没有解码结束时，这个变量就被设置为true。当这个变量被设置为true时，需要调用解码器的flush方法，剩下的就是调用stop方法来停止解码器，最终释放解码器资源。

至此硬件解码器就介绍完了，读者可以根据书中的介绍找到对应代码仓库中的源码并进行分析，这样才可以更加熟练地应用到自己的项目中去。

10.2　音频效果器的集成

本节会介绍如何把第8章中的音频处理算法集成到App中，在Android平台和iOS平台中的实现肯定不同，因为Android平台的算法都会在CPU上进行计算，而iOS平台使用的是AudioUnit的实现，所以本节分为两部分来讲解，分别对第8章中两个平台的音频处理算法给出结构调整，以适配我们的整个App的架构。

在开始设计之前，对于音效处理的理解是很关键的，下面将第8章中介绍的压缩效果器、均衡效果器和混响效果器作为基础的混音效果器，并将这三种效果器使用不同的参数组合起来，形成不同的音乐风格，比如摇滚、流行、舞曲等风格。还有，由于各个效果器的参数众多，最好以配置文件的形式来配置参数，这样就可以在不改动代码的情况下实现增加音乐风格的功能。我们基于以上两点来开始设计与实现。

10.2.1 Android音效处理系统的实现

由于音效处理是一个串行的数据处理过程，因此前一级节点的输出要作为后一级节点的输入，并把每级节点都当作是一个Filter，让所有的Filter继承自一个基类BaseFilter，这样就可以统一接口进行处理。通过上述分析，不难得出结论，可以使用责任链设计模式完成这种场景的需求。而在Filter的类型上，除了上述三种混音效果器外，应该还有一种最基础的音量调节效果器，或者是音量的自动增益控制效果器，总体结构如图10-2所示。

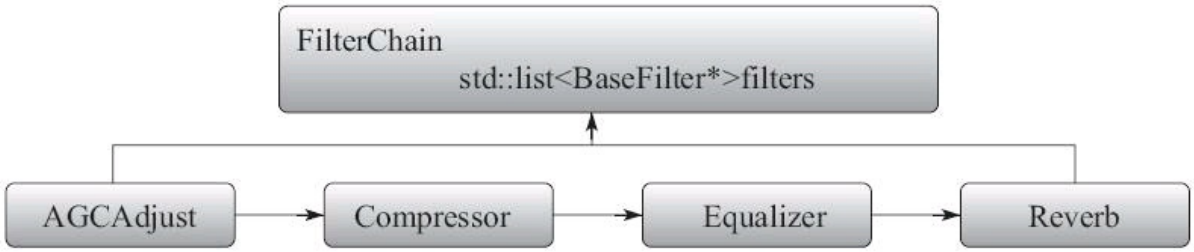


图 10-2

随着产品的功能迭代，有可能会继续加入新的效果器，比如淡出效果器，那么就新写一个FadeOutFilter继承自BaseFilter，然后重写生命周期的方法来完成数据处理，作为最后一级节点放入FilterChain中。这个Filter的构建规则以及如何添加到FilterChain中，后面会慢慢介绍到。

然后来看如何初始化这些Filter，由于不同的Filter需要不同的参数，所以为了满足当前Filter的参数组合，可抽象出一个AudioEffect类来将这些参数存放起来，并且新建一个工厂类AudioEffectBuilder来构建这个AudioEffect对象。从图10-3中可以看到，AudioEffect中有人声的处理链（Vocalchains）、伴奏的处理链（accompanyChains），以及后处理的处理链（mixpostChains），然后才是其他一些通用参数，比如压缩效果器的参数、均衡器的参数、混响器的参数等。这里使用工厂类AudioEffectBuilder来构建这个AudioEffect类型的对象。但如何获得这个工厂类呢？可新建一个类AudioEffectAdapter，这个类的职责就是根据效果器类型返回对应的工厂类。目前只有一个AudioEffectBuilder，但是后期是有可能进行扩展的。

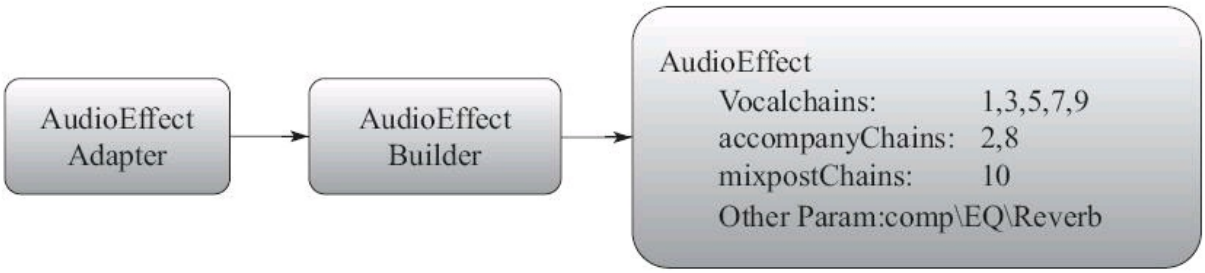


图　10-3

随着产品的迭代，后续也会增加更多的Filter，而这些Filter的参数也是未知的。那么如何才能让代码结构符合“开闭原则”，从而以只增加文件而不修改源代码的形式来添加一个AudioEffect呢？答案是使用抽象工厂的设计模式来完成，即新增一个AudioEffect的子类，比如增加电音效果器：AutoTuneAudioEffect，这个类中会包含很多新的参数变量，而这个类的实例化以及变量的赋值就由工厂类来完成，即AutoTuneAudioEffectBuilder来完成，这个类要继承自AudioEffectBuilder来统一向外界暴露接口。整体结构如图10-4所示。

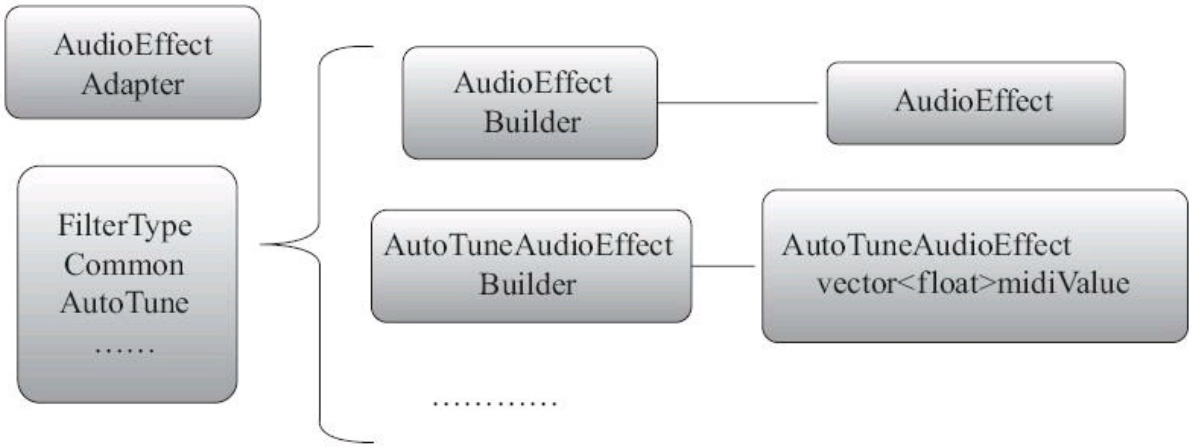


图　10-4

在各个Filter之间传递的数据结构，一般情况下是两轨声音的buffer（包含人声的buffer和伴奏的buffer）数据。但是，随着将来产品的迭代，有可能会有新的效果器加入，可能某些效果器在实时处理时也需要额外的输入参数，或者有实时的输出参数。所以这里定义了两个数据结构，如图10-5所示。

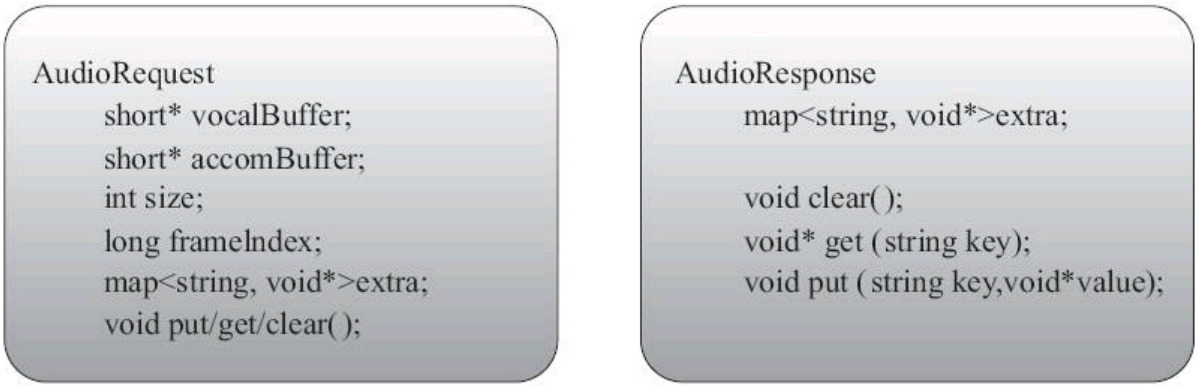


图　10-5

由图10-5可以看到，这里把输入抽象成为了一个AudioRequest类，其中包含了人声数据buffer、伴奏数据buffer，以及这段buffer的大小（size）和时间戳（frameIndex），最重要的是最后一个map类型的extra属性，这个属性就是负责提供实时数据处理的参数输入。同样，输出抽象类AudioResponse也会有这个map类型的extra属性，可以将实时处理数据的结果进行存储，并返回给客户端代码。明确了Filter之间传递的数据结构，下面来看BaseFilter的具体代码：

```
class AudioEffectFilter {
public:
    virtual int init(AudioEffect* audioEffect) = 0;
    virtual void doFilter(AudioRequest* request, AudioResponse* response) = 0
    virtual void destroy(AudioResponse* response) = 0;
};
```

随着产品的迭代，若要增加一个电音效果器，在满足“开闭原则”的条件下，要考虑如何在只新增几个类而不修改原有代码的情况下完成场景的需求。这里要分两部分来实现：第一部分是将新增的Filter的枚举类型值放入配置文件中，后续再根据枚举类型值来构造出AudioEffect中的元素vocalChain；第二部分是使用工厂类AudioEffectFilterFacoty来根据效果器的枚举类型构造出对应的效果器，整体结构如图10-6所示。

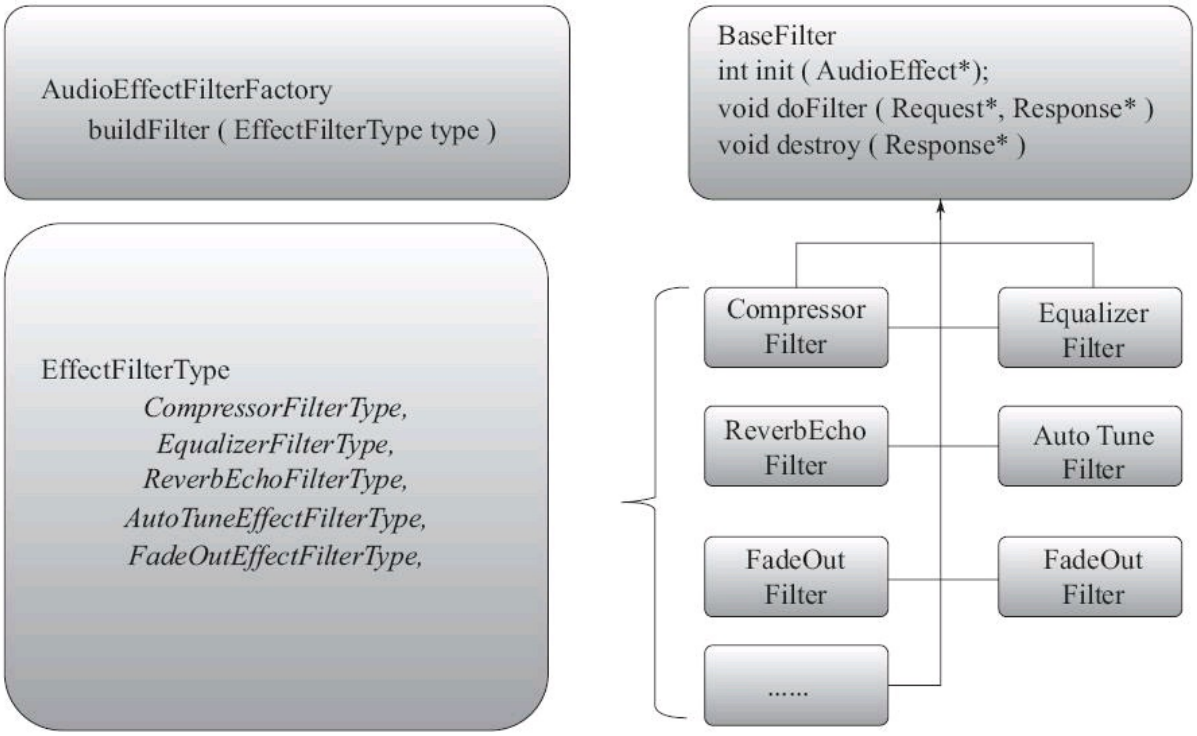


图　10-6

接下来封装一个总体的控制类AudioEffectProcessor，将整个系统串联起来，代码如下：

```
class AudioEffectProcessor {
protected:
    AudioRequest* request;
    AudioResponse* response;
    AudioEffect* audioEffect;
public:
    virtual void init(AudioEffect* audioEffect);
    virtual AudioResponse* process(short *vocalBuffer, short *accompanyBuffer
            int audioBufferSize, long frameIndex) = 0;
    virtual void setAudioEffect(AudioEffect* audioEffect) = 0;
    virtual void resetFilterChains() = 0;
    virtual void destroy();
};
```

最重要的process方法的处理结构如图10-7所示。其中，vocalEffectFilterChain是负责处理人声这一轨音频的，accompanyEffectFilterChain是负责处理伴奏这一轨音频的，当这两轨音频处理完之后，再利用MixEffectFilter将这两轨声音Mix为一轨声音，最终使用MixPostEffectFilterChain给这一轨声音做最后处理（比如淡出效

果器）。

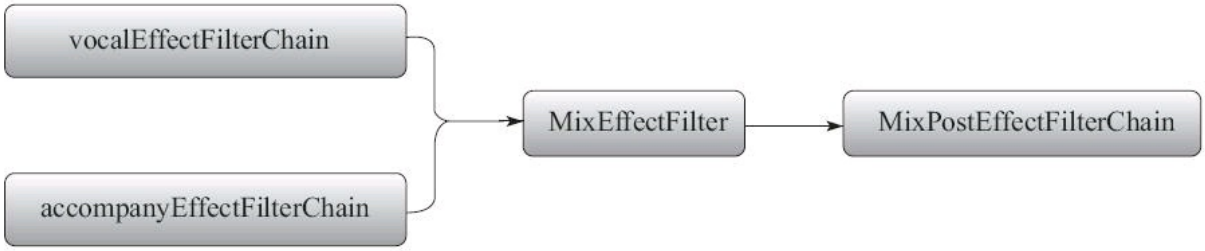


图　10-7

至此，这一套Android上的音频处理系统就搭建完毕。这套音频处理系统的设计充分体现了面向接口编程与尽量遵循了“开闭原则”。后续章节中，我们会将这套系统集成到我们的整个App中，所以在这里读者一定要理解这一套系统的设计，可以到代码仓库中将对应源码进行分析，便于深入理解。

10.2.2　iOS音效处理系统的实现

在iOS平台，我们基于AUGraph这个框架来实现音效处理系统。从名字就可以看出，这个框架是一个图状结构，而基于AUGraph将各个AudioUnit串联起来组成整个处理的图状结构是非常简单的。第8章也已经介绍了单独的压缩效果器、均衡器以及混响效果器。从更高层次来看，Input是用麦克风来收集人声的，或者通过解码器来解码伴奏的，对应的AudioUnit就是RemoteIO和AudioFilePlayer；而Output是Speaker播放Mix之后的人声和伴奏，或者将人声和伴奏的声音编码到本地磁盘上，对应的AudioUnit就是RemoteIO和Generic Output；至于Processor，对应的就是压缩效果器、均衡器以及混响器对于人声的处理以及将人声和伴奏的Mix操作。所以整体架构图如图10-8所示。

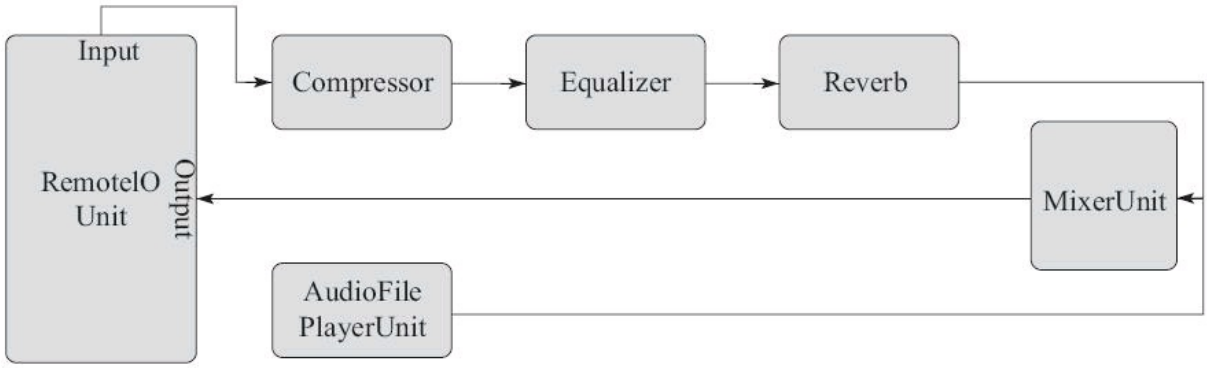


图　10-8

在图10-8中，除了RemoteIO同时作为Input和Output，以及AudioFilePlayer作为Input之外，其他所有节点共同组成音频的Processor部分，而其中压缩器、均衡器和混响器这三个效果器的参数有很多，不同的音乐风格（摇滚、流行、舞曲等）对应的这三个效果器中的参数是不同的。我们把这些参数放置到一个plist类型的配置文件中，会让代码有更强的可读性，并且有利于将来的扩展。

对于伴奏的解码角色AudioFilePlayerUnit，前面已经介绍过，iOS将这个AudioPlayerUnit提供给开发者用来解码伴奏，并且可以直接添加到AUGraph中。对于图10-8中的整个处理结构，驱动方是Output，即RemoteIO Unit，这在录制阶段或者在播放阶段都是可以的。但是，当处于离线渲染场景时，Output就不再是RemoteIO了，因此必须找一个可以

用来驱动整个Graph的节点，而这个节点就是GenericOutput类型的AudioUnit。这个GenericOutput的类型是kAudioUnitType_Output，子类型是kAudioUnitSubType_GenericOutput，待给它设置好属性后，将它连接到最终的一个节点上，然后启动一个while循环并一直渲染这个AudioUnit的数据，这样就可以将整个数据流转起来，最终获得音频数据之后并保存到文件中。

如果需要自己写一些算法来处理音频数据，应该如何把数据集成到这个AUGraph中去呢？答案是给对应的AudioUnit设置InputCallback的方式。在设置的这个回调函数中，首先会调用前一级节点渲染数据的方法，得到了AudioBufferList类型的ioData就可以进行CPU上的运算；最后将运算完毕的数据再放回到ioData变量中，就可以继续执行后续的效果器，从而满足这种场景的需求。

如果要将录制的人声在任何效果器起作用之前保存下来，就要给Compressor这个节点增加一个InputCallback，再在回调函数中调用RemoteIO这个AudioUnit产生数据，得到AudioBufferList类型ioData；然后利用ExtAudioFile写到本地磁盘文件中。保存下来的文件称为干声文件。ExtAudioFile这个API的使用在前面已经介绍过，不再赘述。

如果某些CPU算法的效果器要求SInt16的输入，就需要在这个AUGraph的处理结构适当的节点上插入AudioConvertUnit，以便将当前格式（如Float32）的数据转换为SInt16格式的数据，待这些算法的效果器执行完毕之后，再给AudioConvertUnit以将SInt16格式的数据转换为Float32格式，并发送给后面的效果器Node，以完成后续的数据处理操作。

总体来说，在iOS平台使用AUGraph完成音频处理是比较简单的，后面会将这个系统集成到视频录制的项目中，读者可以到代码仓库中查看源码，便于深刻理解。

10.3　一套跨平台的视频效果器的设计与实现

有的读者可能感觉跨平台的视频效果器一定是用C语言或者C++语言来编写并运行在CPU上面的，其实不然，由于本书的目标是移动平台，所以对性能的要求是极高的。鉴于之前的基础，不难得出结论，这套跨平台的效果器是基于OpenGL ES来实现的。下面让我们从分析应用场景入手，一块来设计和实现这套跨平台的视频效果器库。

先思考视频效果器库会在哪些场景下使用，比如有可能会在用户录制视频界面的预览中使用，因为用户在预览的时候可能需要美颜效果；也有可能在后处理阶段会使用，因为用户可能会给视频加上一些动图或者主题。下面介绍这两种典型的场景。

对于预览界面，可以回想第6章的摄像头预览项目。无论是iOS平台还是Android平台的摄像头预览，都是基于OpenGL ES渲染环境渲染摄像头采集出来的图像，而在OpenGL ES中传递的视频帧对象就是一个纹理ID，所以可以设计视频效果器库的处理视频帧接口如下：

```
void process(int inputTexID, float position, int outputTexID);
```

其中，参数inputTexID代表原始的视频帧；参数position代表这一帧视频帧的时间戳；参数outputTexID代表经过视频效果器处理完毕的输出纹理ID。其实一个接口就可以满足预览场景下的视频滤镜的处理。接下来请读者思考这个接口能否满足后处理场景下的需求？答案是肯定的，因为后处理场景下的渲染环境也是基于OpenGL ES来构建的，而在中间传递的也是纹理ID，所以这个接口也可以无缝地接入后处理场景中去。读者可以参考第7章中的图7-2，从更高层次来理解，无论无摄像头还是解码器都属于Input，而无论是View还是编码器都属于Output，至于中间要加入的视频效果器库则属于Processor，整体架构如图10-9所示。

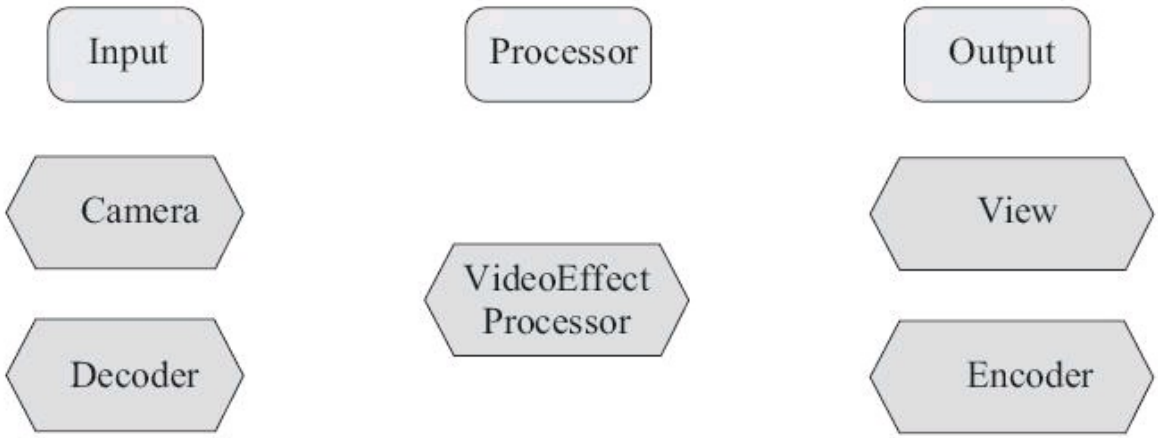


图　10-9

图10-9的分析如下：从Camera采集一帧视频帧，经过Processor处理，到View显示就是预览过程，再到Encoder编码就是录制视频的保存过程；从Decoder解码一帧视频帧，经过Processor处理，到View显示就是视频编辑界面，再到Encoder编码就是后处理的保存过程，所以这个架构可以满足整个视频录制和后处理的所有场景。

既然视频效果器库的处理接口已经确定好，如何与客户端代码对接就是当前要解决的问题。要在OpenGL ES中将一个纹理对象处理之后绘制到另外一个纹理对象上，肯定需要帧缓存对象，即FBO。对于FBO的用法，官方文档上有一句话是这样说的："频繁绑定FBO与解绑定FBO的效率远不如使用同一个FBO在不同的纹理ID上进行切换（Attach）。"所以视频效果器库需要和客户端代码制定以下协议：客户端代码需要在初始化特效处理器的时候同时生成FBO与输出纹理ID。注意这个初始化过程必须在OpenGL ES的线程中，代码如下：

```
VideoEffectProcessor *processor;
processor = new VideoEffectProcessor();
processor->init(width, height);
glGenFramebuffers(1, &mFBO);
glGenTextures(1, &outputTexId);
glBindTexture(GL_TEXTURE_2D, outputTexId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, 0);
glBindTexture(GL_TEXTURE_2D, 0);
```

然后在调用特效处理器的处理方法之前绑定FBO，之后解绑定FBO，代码如下：

```
glBindFramebuffer(GL_FRAMEBUFFER, mFBO);
processor->process(inputTexId, position, outputTexId);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

当用户切换视频特效的时候，调用特效处理器的切换特效的方法，比如从美颜特效切换到自然特效，代码如下：

```
processor->removeAllFilters();
int filterId = processor->addFilter(sequenceIn, sequenceOut, filterName);
processor->setFilterParamValue(filterId, filterKey, filterValue);
```

如上述代码所示，先移除掉所有的Filter。然后使用filterName增加一个Filter，并使用参数sequenceIn代表这个Filter的开始作用时间，使用参数sequenceOut代表这个Filter的结束作用时间。其次根据上一步得到的filterID给这个Filter设置对应的参数，这样就达到了切换效果器的目的，当然这种切换是把最细粒度的接口都公布给客户端代码。事实上，也可以封装一些接口，使其功能更加单一，从而更方便调用客户端，具体可以参照代码仓库中的示例代码。最后是销毁FBO、特效处理器以及纹理ID。注意这段代码也必须在OpenGL ES的线程中执行，代码如下：

```
if (mFBO) {
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glDeleteFramebuffers(1, &_mFBO);
}
if(processor) {
    processor->dealloc();
    delete processor;
    processor = NULL;
}
glDeleteTextures(1, &outputTexId);
```

将客户端代码的协议描述清楚之后，接下来看看如何设计效果器的内部实现。

这里要实现的这个框架包含9.4节的所有效果器，并且能根据产品

版本的迭代不断引进新的效果器。此外，用户看到的滤镜效果有可能包含多个效果器，它以一个效果器链的形式来完成一个滤镜效果，比如美颜滤镜，首先要磨皮效果器，然后要增加对比度的效果器，接着是提亮，最后可能还要一个色调曲线调整出清凉、复古等效果，如图10-10所示。

图　10-10

所以这里的设计要符合这种基本需求，首先为效果器建立一个Model类来存储效果器的参数以及作用时间。在效果器里，与渲染（OpenGL ES的绘制）无关的业务逻辑计算都放在这个类里来完成，定义ModelFitler的基类，代码如下：

```
class ModelFilter {
public:
    ModelFilter();
    ModelFilter(int index, int64_t sequenceIn, int64_t sequenceOut,
            char* filterName);
    virtual ~ModelFilter();
    virtual bool init();
    virtual void onRenderPre(float pos);
    virtual bool isAvailable(float pos);
    void setFilterParamValue(char * paramName, ParamVal value);
    bool getFilterParamValue(string paramName, ParamVal & value);
    int getId() {
        return id;
    };
private:
    int id;
    int64_t sequenceIn;
    int64_t sequenceOut;
    map<string, ParamVal>        mMapParamValue;
}
```

上述代码中，定义ModelFitler的基类将通用的属性和方法都封装起来，并向外暴露出几个接口：init方法是留给子类去做一些资源初始化的，比如视频主题效果器以及贴纸效果器的解码器初始化等行为；onRenderPre方法是为了某些效果器能在渲染本帧之前根据时间戳计算

当前效果器的进度，比如渐变模糊效果器；isAvailable方法则用于判定当前效果器是否在作用区间内，实现方法主要就是判定当前帧的时间戳和sequenceIn及sequenceOut的关系；getId方法就是获得当前效果器的唯一标识。另外，还有一个比较重要的参数设置方法和获取参数的方法，这里定义了一个结构体ParamVal，包含了所有的基础数据类型和一个Type类型来记录当前属性是什么数据类型，最终会将这个ParamVal作为一个Key的Value存储到map类型的属性中，以便后续访问，结构体ParamVal的展示如下：

```
struct ParamVal {
    union {
        void *arbData;
        int intVal;
        double fltVal;
        bool boolVal;
        Color colorVal;
        Position2D pos2dVal;
        Position3D pos3dVal;
    } u;
    string strVal;
    EffectParamType type;
};
```

这样就可以表示所有的属性了，而在子类中可以直接取出对应的参数进行计算。然后再建立一个类TimeLine把当前滤镜效果所用到的效果器列表包含进来，代码如下：

```
class ModelTimeLine {
public:
    ModelTimeLine();
    virtual ~ModelTimeLine();
    void clear();
    int addFilter(int64_t sequenceIn, int64_t sequenceOut, char* filterName);
    void deleteFilter(int sub);
    bool invokeFilterOnInit(int filterId);
    void setFilterParamValue(int filterId, char * paramName, ParamVal value);
    list<ModelFilter*> getAllFilters();
private:
    list<ModelFilter *>  filters;
    int filterCount;
}
```

如上述代码所示，当调用addFilter方法加入一个Filter的时候，首先会根据filterName找到对应的Filter，再将其实例化并加入filters这个列表

中，同时将filterCount加1，作为这个filter的ID并返回；然后客户端就可以根据这个filterId调用setFilterParamValue方法给filter设置参数；最后根据这个filterId调用invokeFilterOnInit方法，进而调用filter的初始化方法。下面创建所有效果器真正处理图像效果的Effect类，先定义一个BaseVideoEffect的基类，然后让所有的效果器都继承自这个类，并重写生命周期方法完成子类自己的行为。

下面分析基类BaseVideoEffect，既然是执行OpenGL ES的渲染操作的封装，那么必须要加载Shader以及Program的工具方法，同时也要把VertexShader和FragmentShader的内容，以及构造出来的顶点坐标和GLProgram作为成员变量封装起来，代码如下：

```
protected:
    char* mVertexShader;
    char* mFragmentShader;
    bool mIsInitialized;
    GLuint mGLProgId;
    GLuint mGLVertexCoords;
    GLuint mGLTextureCoords;
    GLint mGLUniformTexture;
    /** 工具方法 **/
    void checkGlError(const char* op);
    GLuint loadProgram(char* pVertexSource, char* pFragmentSource);
    GLuint loadShader(GLenum shaderType, const char* pSource);
```

接下来是最重要的生命周期方法，由于子类有可能要在初始化阶段初始化自己的Shader，找出Shader中的独有变量，并分配自己用到的中间纹理对象等，所以需要重写init方法；在渲染阶段，子类有可能要自己渲染操作，比如要传递Uniform变量给FragmentShader，所以要重写renderEffect方法；在销毁阶段，子类有可能要释放自己分配的一些纹理对象等资源，所以要重写destroy方法，具体代码如下：

```
class BaseVideoEffect {
public:
    BaseVideoEffect();
    virtual ~BaseVideoEffect();
    virtual bool init();
    virtual void renderEffect(int inputTexId, float pos, int outputTexId,
            EffectCallback * filterCallback);
    virtual void destroy();
};
```

每个子类会完成相应的效果渲染，比如磨皮效果器、提亮效果器、视频主题效果器等。我们来看真正渲染视频时是如何遍历所有的效果器并依次渲染的，从而完成最终效果渲染工作。首先来看方法签名，代码如下：

```
void VideoEffectProcessor::process(GLuint sourceTexId, float position,
    GLuint outputTexId);
```

接着根据当前帧的时间戳，过滤所有可能发生作用的效果器数量，代码如下：

```
list<ModelFilter *> filters = timeLine->getAllFilters(0);
list<ModelFilter *>::iterator itor = filters.begin();
int filterCount = 0;
for (; itor != filters.end(); itor++) {
    ModelFilter* filter = *itor;
    if (filter->isAvailable(position))
        filterCount++;
    }
}
```

然后判断若filterCount是0，则代表当前时间点没有效果器起作用，因此可直接调用一个直通的效果器将inputTexId渲染到outputTexId上，代码如下：

```
if(filterCount == 0){
    directPassEffect->renderEffect(sourceTexId, outputTexId, NULL);
}
```

如果filterCount不为0，则要依次执行所有的效果器，并要为每个效果器建立输出纹理ID，以及将前一级效果器的输出纹理ID作为当前效果器的输入纹理ID，将当前效果器的输出纹理ID作为后一级效果器的输入纹理ID，代码如下：

```
list<ModelFilter *>::iterator itor = filters.begin();
int sub = 0;
GLuint previousTexId = sourceTexId;
GPUTexture* previousTexture = NULL;
for (; itor != filters.end(); itor++) {
    ModelFilter* filter = *itor;
```

```
        if (filter->isAvailable(position)) {
            BaseVideoEffect* effect = effectCache->
                    getVideoEffectFromCache(string((*itor)->name));
            if (effect) {
                filter->onRenderPre(position);
                GLuint currentTexId = outputTexId;
                GPUTexture* texture = NULL;
                if(sub < (filterCount - 1)){
                    texture = GPUTextureCache::GetInstance()->
                            fetchTexture(width, height);
                    texture->lock();
                    currentTexId = texture->getTexId();
                }
                effect->renderEffect(previousTexId, position, currentTexId,
                    filter->getFilterCallback());
                previousTexId = currentTexId;
                if(NULL != previousTexture){
                    previousTexture->unLock();
                }
                previousTexture = texture;
            } else {
                LOGE("getVideoEffectFromCache failed");
            }
            sub++;
        }
    }
```

细心的读者从这段代码里可以看到有两个追求效率的缓存，其中一个是前面已介绍过的纹理缓存，可根据宽高到纹理缓存中取出对应的纹理对象，这大大降低了频繁开辟纹理与释放纹理的开销；另外一个就是effect的缓存，这个缓存存在的意义是降低频繁创建和销毁OpenGL的Program的开销。虽然这两个优化看起来不太起眼，但是这两个缓存在这里非常重要。一个App运行流畅与否，与这些细节的优化息息相关。

至此，视频效果器库就搭建好了。读者可以去查看代码仓库中的源码，后续章节会把这个效果器库集成到系统中。

10.4　将特效处理库集成到视频录制项目中

经过前面的学习，相信大部分读者已经掌握了各自平台的音频效果器系统和视频效果器系统，接下来要将这两个系统集成到我们的App中。一个录制视频的项目其核心流程有如下三个阶段。

第一阶段是录制视频。用户可以预览到摄像头中的实时图像，麦克风可以采集到外界声音，并有可能播放背景音乐或者伴奏（对于唱歌的App），这个阶段最后会生成一个视频文件。

第二阶段是编辑阶段。用户可以看到上一阶段录制的视频，并且可以给视频增加一些特效，包括音频特效与视频特效，最终在调节成一个满意的效果之后，点击“保存”按钮进入第三阶段。

第三阶段是离线保存阶段。自动按照用户选择的特效以第一阶段原始视频作为输入，进行处理并保存到最终文件中。

以上三个阶段就是整个视频录制项目的核心流程，在第二阶段与第三阶段中的视频解码部分可以使用10.1节讲解的硬件解码器来提升性能，每个阶段都会用到10.2节和10.3节的效果器。所以接下来介绍如何将音频与视频效果器库集成到每一个阶段。

10.4.1 Android平台特效集成

本节将介绍在Android平台上如何集成音频与视频特效，由于我们的实现都是在Native层，并且这两个特效库也都是使用C或者C++语言开发的，所以集成工作也主要发生在Native层。

1.集成音频特效处理库

下面集成音频特效处理库。10.2节已经详细介绍过音频特效库的设计，其中核心类就是AudioEffectProcessor。在第7章的视频录制项目中，对于音频模块编码器线程，在将人声PCM队列中的声音数据与伴奏PCM队列中的伴奏数据进行Mix之后，就会进行编码。如果读者对这两点都有印象，就可以继续下面的工作；如果觉得有点模糊，请查看7.3.2节和10.2节相关的内容。

大家还记得7.3.2节中编码器适配器类中的getAudioPacket方法吗？这个方法会调用一个processAudio方法。现在建立一个子类AudioProcessEncoderAdapter来重写这个方法，从而完成扩展的作用。在重写的processAudio方法中，就可以调用AudioEffectProcessor的process方法。子类还需要重写父类的init方法，实现伴奏队列与音频效果器的初始化，初始化方法代码如下：

```
void AudioProcessEncoderAdapter::init(LivePacketPool* pcmPacketPool, int
        audioSampleRate, int audioChannels, int audioBitRate,
        const char* audio_codec_name, AudioEffect *audioEffect) {
    this->accompanyPacketPool = LiveCommonPacketPool::GetInstance();
    AudioEffectProcessor*audioEffectProcessor = AudioEffectProcessorFactory::
            GetInstance()->buildAudioEffectProcessor();
    audioEffectProcessor->init(pcmPacketPool, audioSampleRate, audioChannels,
            audioBitRate, audio_codec_name);
    this->channelRatio = 2.0f;
}
```

以上代码中将channelRatio设置为2.0是因为Android平台录制出的人声是单声道的，而父类中计算每一次处理的音频帧大小（PacketBufferSize）时会用到该设置，因为父类是同时运行在Android平台和iOS平台的。在iOS平台上，取出的buffer多大就是多大；但是在Android平台上，由于需要AudioEffectProcessor将声音处理成双声道，因

此需要乘以channelRaio的参数。这里最主要的参数当属AudioEffect对象，该对象就是声音特效的模型对象，里面包含整个声音处理过程的音效参数。下面提供一个构造默认AudioEffect对象的方法，代码如下：

```
AudioEffect* AudioEffectAdapter::buildDefaultAudioEffect(int channels, int
            audioSampleRate, bool isUnAccom) {
    float accompanyVolume = 1.0f;
    float audioVolume = 1.0f;
    SOXFilterChainParam* filterChainParam = SOXFilterChainParam::
            buildDefaultParam();
    std::list<int>* vocalEffectFilters = new std::list<int>();
    vocalEffectFilters->push_back(VocalAGCVolumeAdjustEffectFilterType);
    vocalEffectFilters->push_back(CompressorFilterType);
    vocalEffectFilters->push_back(EqualizerFilterType);
    vocalEffectFilters->push_back(ReverbEchoFilterType);
    vocalEffectFilters->push_back(VocalVolumeAdjustFilterType);
    std::list<int>* accompanyEffectFilters = new std::list<int>();
    accompanyEffectFilters->push_back(AccompanyAGCVolumeAdjustEffectFilterTyp
    accompanyEffectFilters->push_back(AccompanyVolumeAdjustFilterType);
    std::list<int>* mixPostEffectFilters = new std::list<int>();
    mixPostEffectFilters ->push_back(FadeOutEffectFilterType);
    int recordedTimeMills = 0;
    int totalTimeMills = 0;
    float accompanyAGCVolume = 1.0f;
    float audioAGCVolume = 1.0f;
    float accompanyPitch = 1.0f;
    float outputGain = 1.0f;
    int pitchShiftLevel = 0;
    AudioInfo* audioInfo = new AudioInfo(channels, audioSampleRate,
            recordedTimeMills, totalTimeMills, accompanyAGCVolume,
            audioAGCVolume, accompanyPitch, pitchShiftLevel);
    return new AudioEffect(audioInfo, vocalEffectFilters,
            accompanyEffectFilters, mixPostEffectFilters, accompanyVolume,
            audioVolume, filterChainParam, outputGain);
}
```

上述代码展示了一个完整的构造AudioEffect对象的过程。首先，在人声和伴奏的处理器链的最前端添加了一个自动增益控制的音量调节效果器，在处理器链最后端添加了一个可以让用户自己调节音量的效果器。其中，类SOXFilterChainParam会将比较复杂的压缩效果器、均衡器和混响器的参数包含在里面，读者可以去代码仓库中找到源码，此处不再展示。使用上述构造的AudioEffect可以明显听到混响效果，因此我们暂时使用上述参数配置集成到视频录制项目中。

下面进入真正的处理过程，重写processAudio方法。首先从伴奏队列中取出PCM的数据，然后交给音频处理器去处理，并合并成为父类需要放置到packetBuffer的全局变量中，代码如下：

```
int AudioProcessEncoderAdapter::processAudio() {
    int ret = packetBufferSize;
    LiveAudioPacket *accompanyPacket = NULL;
    if (accompanyPacketPool->getAccompanyPacket(&accompanyPacket, true) < 0)
        return -1;
    }
    if (NULL != accompanyPacket) {
        int accompanySampleSize = accompanyPacket->size;
        short* accompanySamples = accompanyPacket->buffer;
        long frameNum = accompanyPacket->frameNum;
        audioEffectProcessor->process(packetBuffer, packetBufferSize,
            accompanySamples, accompanySampleSize, frameNum);
        delete accompanyPacket;
        accompanyPacket = NULL;
    }
    return ret;
}
```

使用这个方法的时候，父类已经把人声从人声队列中取出来，并放在分配了2倍大小空间的packetBuffer的前半部分，进入这个类之后，首先从伴奏的队列中取出伴奏PCM的数据，然后将这两轨音频交给音频效果处理器进行处理，处理之后的结果放入packetBuffer中，父类会继续完成接下来编码的操作。最终会在destroy方法中销毁相应的资源，代码如下：

```
void AudioProcessEncoderAdapter::destroy(){
    accompanyPacketPool->abortAccompanyPacketQueue();
    AudioEncoderAdapter::destroy();
    accompanyPacketPool->destoryAccompanyPacketQueue();
    if (NULL != audioEffectProcessor) {
        audioEffectProcessor->destroy();
        delete audioEffectProcessor;
        audioEffectProcessor = NULL;
    }
}
```

从以上代码中可以看到，首先丢弃了伴奏的队列，以免编码线程会被阻塞在processAudio方法中；然后调用父类的destroy方法，父类的这个方法会停止编码线程，再销毁伴奏的PCM队列，最终销毁我们自己创建的音频效果处理器。

这样就将音频效果处理器集成到我们的系统中了，是不是很简单？在编辑页面（及视频播放器中）如何集成音频效果处理器以及在离线保存中如何集成，读者可以结合代码仓库中的源码部分加深了解。

2.集成视频特效处理库

下面会将10.3节的视频特效库集成到第7章的视频录制项目中，而在视频录制项目中有一个模块是摄像头的预览与编码模块，结构图如图10-11所示。

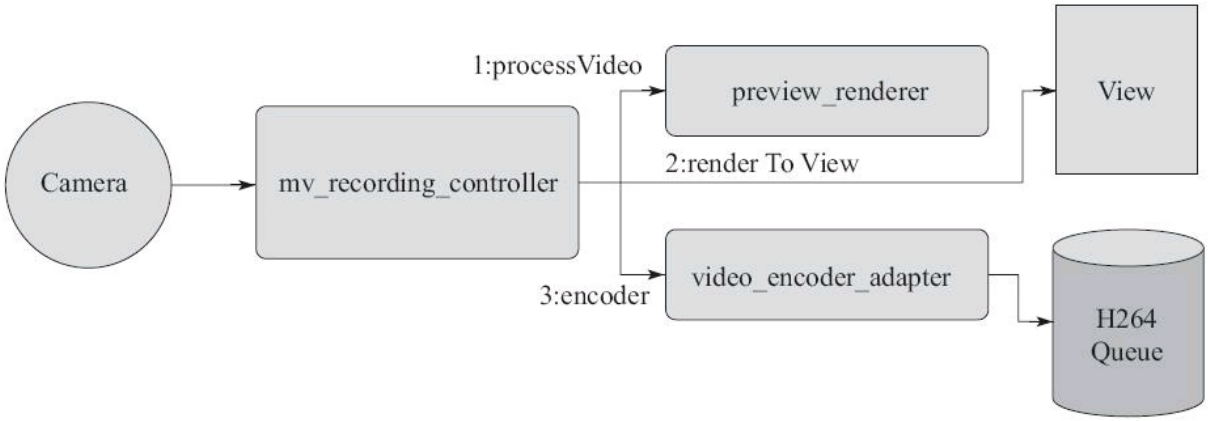


图　10-11

在图10-11中，摄像头采集了图像之后会传递到类MVRecordingController中。而在控制器中，第一步会交由PreviewRenderer处理（旋转、镜像等操作），并将这一帧图像处理成为一个RGBA格式的纹理ID；第二步则通过控制器将这个RGBA格式的纹理ID渲染到View上；最后一步将这个RGBA格式的纹理ID交由编码器进行编码，编码成功之后放入编码器队列中，这就是视频轨部分的处理。

下面要将视频效果处理器集成到PreviewRenderer这个类中，而这个集成过程对于外界的控制器类来讲是透明的。所以改进之后的PreviewRenderer处理这一帧图像的流程如图10-12所示。

图　10-12

如图10-12所示，由于摄像头采集的纹理ID是OES格式的，而这与第10.1节中接触到的硬件解码器解码的纹理格式是一致的，所以可以共用copier的渲染过程，将这个纹理做旋转和镜像，最终输出到RGBA格式的rotateTexID中。然后再做一个纹理图像的裁剪使其适配界面View的

宽高，因为摄像头采集的图像可能是640×480的，而我们需要的纹理可能是640×360（16∶9的矩形视频）或者是480×480（1∶1的方形视频）的，最终输出到inputTexId中。在学习本章之前已介绍过如何将这个inputTexId返回给控制器，以进行后续的渲染界面和执行编码流程。这里则是将inputTexId交由VideoEffectProcessor处理成为outputTexId，然后交由控制器执行后续的渲染过程。下面来看PreviewRenderer类中代码的具体改动。

因为PreviewRenderer的初始化方法在渲染线程中，所以可直接在初始化方法中调用视频特效处理器的初始化方法，代码如下：

```
void RecordingPreviewRenderer::init(int degress, bool isVFlip,
        int textureWidth, int textureHeight, int cameraWidth, int cameraHeigh
    // 原有代码逻辑不再展示
    mProcessor = new VideoEffectProcessor();
    if(mProcessor->init()) {
        mProcessor->removeAllFilters();
        filterId = mProcessor->addFilter(PREVIEW_FILTER_SEQUENCE_IN,
                PREVIEW_FILTER_SEQUENCE_OUT, IMAGE_BASE_EFFECT_NAME);
        if(filterId >= 0){
            mProcessor->invokeFilterOnReady(filterId);
        }
    }
}
```

从以上代码可以看到，成功初始化视频特效处理器之后，会先移除所有的效果器，然后增加一个BaseEffect类型的效果器，这个效果器仅完成拷贝工作。由于是在视频中预览界面，所以会将sequenceIn和sequenceOut分别设置为0和很大的值，最终执行这个Filter的Init方法。

接下来看看处理视频帧的改动，代码如下：

```
void RecordingPreviewRenderer::processFrame(float position) {
    glBindFramebuffer(GL_FRAMEBUFFER, FBO);
    // 原有代码逻辑不再展示
    glViewport(0, 0, textureWidth, textureHeight);
    mProcessor->process(inputTexId, 0.0, outputTexId);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

从以上代码可以看到，处理视频帧的调用十分简单，直接采用输入和输出调用处理器的处理方法就可。由于这里不需要使用时间戳信息做

额外的操作，所以第二个参数暂时使用0.0作为输入参数。接下来看看如何切换效果器。以美颜滤镜为例，代码如下：

```
mProcessor->removeAllFilters();
int filterId = mProcessor->addFilter(PREVIEW_FILTER_SEQUENCE_IN,
    PREVIEW_FILTER_SEQUENCE_OUT, BEAUTIFY_FACE_FILTER_NAME);
mProcessor->invokeFilterOnReady(filterId);
```

最终是销毁阶段，这个阶段也需要在OpenGL ES的线程中执行，所以放在PreviewRender的dealloc方法中，代码如下：

```
void RecordingPreviewRenderer::dealloc(){
    // 原有代码逻辑不再展示
    if(mProcessor){
        mProcessor->dealloc();
        delete mProcessor;
        mProcessor = NULL;
    }
}
```

至此，视频特效处理器就集成到了视频录制项目中，大家可以切换成为美颜滤镜查看预览中的效果，然后继续录制视频，并查看最终视频，这样就可以看到自己的皮肤确实被美化了。读者可以到代码仓库中查看整体工程的代码，或者直接运行本节对应的工程查看效果。

10.4.2　iOS平台特效集成

在iOS平台集成这两个特效处理库要简单很多，一个是因为OC开发语言允许直接与C/C++的混编，另一个是因为IDE提供的便利性不需要开发者自己去写makefile文件，这样我们的集成工作就变得很简单。

1.集成音频特效处理库

音频特效处理库的集成，在iOS平台上是很简单的，原因如下。在第7章的录制视频项目中，已将iOS平台的结构拆分得很清楚，AUGraph作为声音队列的生产者，编码线程作为声音队列的消费者，Mux模块负责将音频和视频封装到文件中，整体结构如图10-13所示。

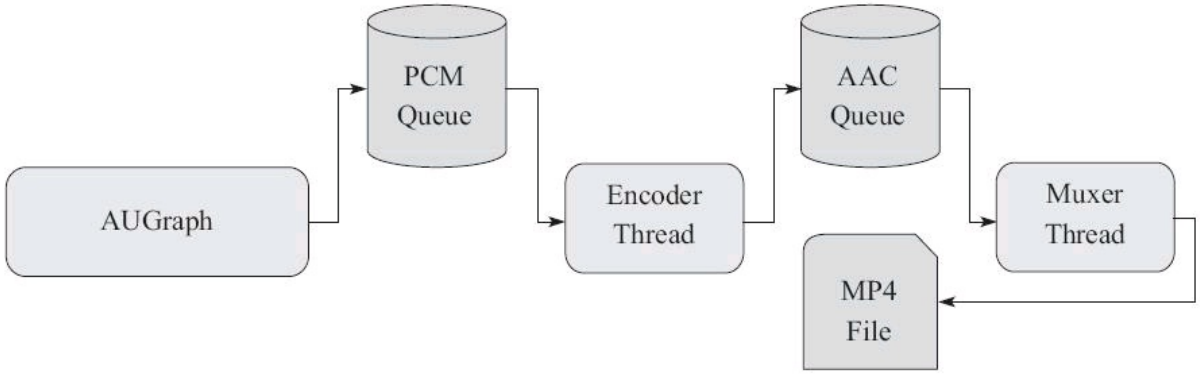


图　10-13

从图10-13中可以看到，首先将AUGraph中的ConvertNode（将Float32格式转换为SInt16格式的AudioUnit）设置为RenderCallback，在RenderCallback方法的实现中取出PCM数据；然后将数据封装为AudioPacket的数据结构放入PCM的队列中，再经由编码器线程从这个队列中取出PCM数据，编码之后封装为AAC的Packet，并存入AAC的队列中；最后由Muxer线程从AAC队列中取出进行封装，使其成为MP4并写入本地磁盘中。在10.2节中，集成入压缩效果器、均衡器和混响效果器之后，整个AUGraph已经成为图10-8所示的结构，所以集成音频效果器仅在AUGraph这个模块进行了变换，并没有影响之后所有的流程。因此，对于PCM队列来讲，也仅仅改变了生产者的结构；而对于消费者及其他模块来说，是没有任何侵入式影响的。也就是说，这个PCM队列作为一个接口将整个系统拆分开，达到了低耦合的目的。具体的集成代

码，其实就是将10.2节中的AUGraph替换过来。

在编辑（预览）阶段的流程，也更改了对应的AUGraph，即在中间增加压缩效果器、均衡器以及混响效果器，只不过这个阶段的AUGraph的人声输入不再是RemoteIO这个AudioUnit，而应该通过AudioFilePlayer这个AudioUnit来解码一个已经存在的人声文件，经过处理之后，最终和当前伴奏解码的AudioFilePlayer这个AudioUnit进行Mix，再输送给RemoteIO并播放给用户听，而RemoteIO就是这个AUGraph的驱动源。整个流程在离线保存阶段也是一样的，只不过驱动源不一样，这一点在10.2节有详细讲解，这里不再赘述。读者可以参考代码仓库中的源码进行分析，便于深入理解。

2.集成视频特效处理库

下面将视频特效处理器集成入视频录制项目中，先来回顾第7章中视频录制项目中的视频轨是如何处理的？

如图10-14所示，首先利用系统将开发者提供的Camera采集一帧YUV格式的图像，然后进入VideoCamera这个节点，这个节点会将CMSampleBuffer中的YUV格式的数据转换为一个RGBA的纹理ID。并将其分为两个分支，其中一条支路到ImageView这个节点，这个节点会把输入的纹理ID渲染到一个UIView上，最终让用户预览到图像；另外一条支路到VideoEncoder这个节点，这个节点会把输入的纹理ID编码成为H264的数据，再封装为我们自己的VideoPacket的结构体对象并放入视频编码队列中，之后再由Muxer模块取出进行封装，并最终写入磁盘文件。这个过程中重点来看中间处理纹理ID的三个节点，10.3节讲解的视频特效处理器的输入和输出都是一个RGBA格式的纹理ID，而将它封装为一个节点并插入VideoCamera节点之后、ImageView和VideoEncoder节点之前，是最合理的。所以集成入视频特效处理器之后，我们只展示中间的结构图，如图10-15所示。

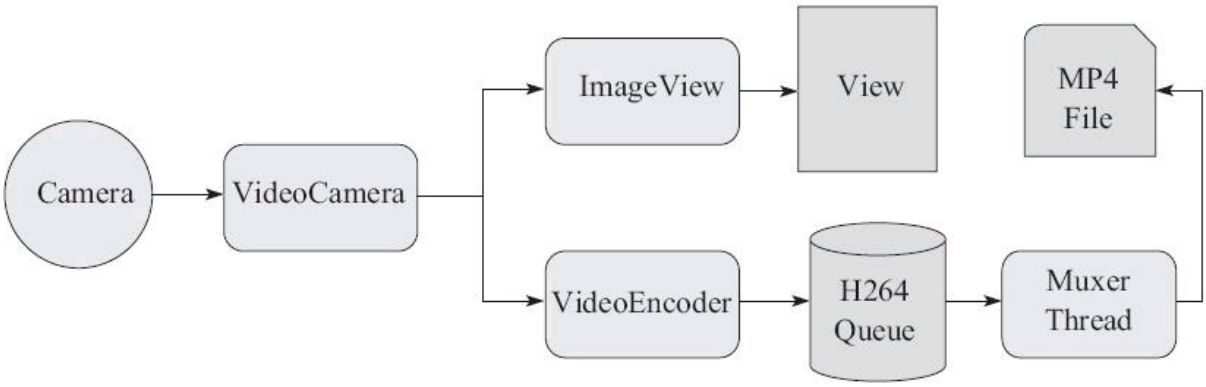


图　10-14

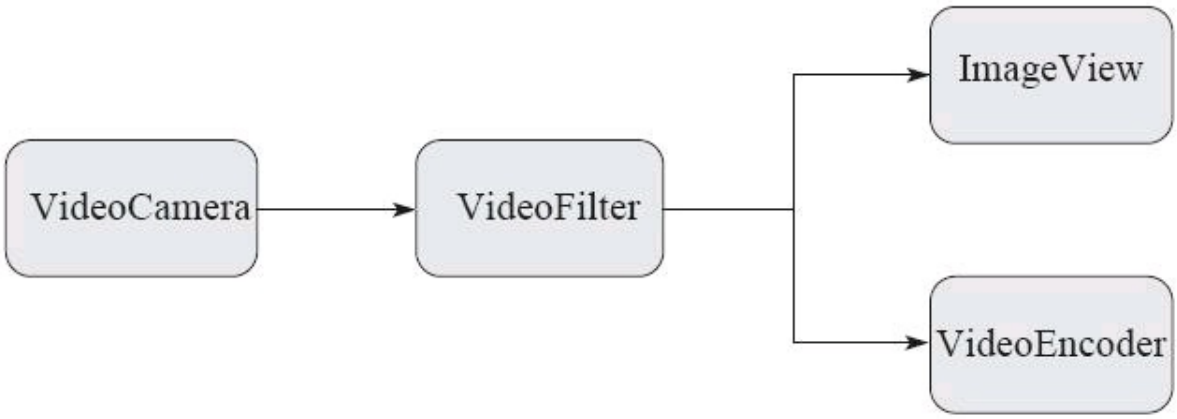


图　10-15

下面介绍如何封装一个节点，以及如何将视频特效处理器集成进来。首先，新建一个类ELImageVideoFilter，由于这个节点要输出一个outputTexId，所以要继承自ELImageOutput，并且这个节点还要依靠VideoCamera节点提供输入，因此要实现ELImageInput这个Protocol。最终，这个新建类的接口文件声明如下：

```
@interface ELImageVideoFilter : ELImageOutput<ELImageInput>

-(void)switchFilter:(ELVideoFiltersType)filterType;

@end
```

从以上代码可以看到，ELImageVideoFilter类还有一个公有的方法，这个方法是用来切换视频滤镜的。接下来将类的实现文件由“.m”的后缀名改为“.mm”的后缀名，因为视频特效处理库是一套用C++语言实

现的库，而在OC中要想调用C++的库，就得将实现文件的后缀名改为“.mm”。我们重写父类中生命周期方法，首先接受上一级节点处理完毕的纹理ID的方法，这个方法需要将输入纹理ID保存下来，作为整个渲染过程的输入纹理ID，代码如下：

```
- (void)setInputTexture:(ELImageTextureFrame *)textureFrame;
{
    _inputFrameTexture = textureFrame;
}
```

第二个生命周期方法就是渲染视频帧方法，这个方法就是当新的一帧需要渲染的时候由上一级节点进行调用，这个方法是在OpenGL ES的线程中执行的，因此它可以实例化并初始化视频特效处理器，代码如下：

```
- (void)newFrameReadyAtTime:(CMTime)frameTime timimgInfo:(CMSampleTimingInfo)
        timimgInfo;
{
    if(!_processor){
        _processor = new VideoEffectProcessor();
        _processor->init();
        _processor->removeAllFilters();
        int filterId = _processor->addFilter(PREVIEW_FILTER_SEQUENCE_IN,
                PREVIEW_FILTER_SEQUENCE_OUT, IMAGE_BASE_EFFECT_NAME);
        if(filterId >= 0){
            _processor->invokeFilterOnReady(filterId);
        }
    }
    [[self outputFrameTexture] activateFramebuffer];
    _processor->process([_inputFrameTexture texture], 0.0, [[self
                outputFrameTexture] texture] );
    for (id<ELImageInput> currentTarget in targets){
        [currentTarget setInputTexture:_processedFrameTexture];
        [currentTarget newFrameReadyAtTime:frameTime timimgInfo:timimgInfo];
    }
}
```

从以上代码中可以看到，首先会对Processor进行判断，如果没有初始化，就进行初始化；然后在初始化过程中默认为视频特效处理器加入一个直通的效果器，即BaseEffect，作用的开始时间为定义的一个宏，是0，结束时间也是定义的一个宏，约为10个小时，而在录制视频阶段不会根据时间戳做特殊处理，所以第二个参数传递的时间戳为0.0。下面绑定输出到纹理对象的FBO上，之后调用处理器将输入纹理对象渲染

到输出纹理对象上，最后将输出纹理对象设置为下一级节点输入纹理对象并调用下一级节点的渲染方法。

还记得在接口文件中声明了一个公有方法吗？是的，我们现在要来实现这个方法，即切换视频滤镜的方法，因为切换滤镜的方法有可能在主线程中调用。为了保证线程安全，在这个方法中仅设置一个变量标志，代表需要更改视频滤镜，并把要更改的滤镜类型保存到全局变量中，代码如下：

```
-(void)switchFilter:(ELVideoFiltersType)filterType
{
    if(filterType != curELVideoFiltersType){
        curELVideoFiltersType = filterType;
        isVideoFilterChanged = true;
    }
}
```

具体的更改滤镜的行为，需要等到下一帧渲染的时候再去执行，所以在newFrameAtTime函数的最后执行如下代码：

```
if(isVideoFilterChanged){
    self.processor->removeAllFilters();
    int filterId = -1;
    switch (filterType) {
        case PREVIEW_BEAUTY_FACE:
            filterId = self.processor->addFilter(PREVIEW_FILTER_SEQUENCE_IN,
                PREVIEW_FILTER_SEQUENCE_OUT, BEAUTIFY_FACE_FILTER_NAME);
        break;
        case PREVIEW_NONE:
        default:
            filterId = self.processor->addFilter(PREVIEW_FILTER_SEQUENCE_IN,
                PREVIEW_FILTER_SEQUENCE_OUT, IMAGE_BASE_EFFECT_NAME);
        break;
    }
    if(filterId >= 0){
        self.processor->invokeFilterOnReady(filterId);
    }
    isVideoFilterChanged = false;
}
```

从以上代码中可以看到，会先移除之前所有的效果器；然后按照效果器类型添加到对应的效果器；最后销毁资源，但不要把销毁资源的代码放入dealloc方法中，因为这个方法的调用也必须在OpenGL ES的线程中执行，而系统自动调用的dealloc方法却不是在OpenGL ES线程中调用

的，所以destroy方法的代码如下：

```
if(_processor) {
    _processor->dealloc();
    delete _processor;
    _processor = NULL;
}
```

至此，VideoFilter这个节点就构建完毕。这个节点完成的效果如下：接受上一级节点的输出纹理ID作为输入纹理ID；然后调用视频特效处理器按照设定的滤镜效果把输入纹理ID渲染到一个新的纹理ID上，再把新的纹理ID作为输出纹理ID分别设置给ImageView节点和VideoEncoder节点，这样用户就可以分别在预览界面以及最终生成的文件上看到添加了滤镜效果的视频轨。

为了添加视频滤镜功能，这里仅引入一个库，然后添加一个节点就完成。这比较符合[开闭原则]的设计原则。而这都是因为最初设计这一套系统时抽取了ELImageOutput这个父类，并建立了ELImageInput这个Protocol，这其实都是高度抽象的结果。也正因为这个抽象过程，才使得我们现在集成一个新的节点这么简单方便。有的读者可能会说，组织节点的地方是需要改动代码的，是的，目前我们的系统是需要在组织节点的地方改动代码，但是我们完全可以将组织节点的部分改为读取配置文件的方式来组织节点，这样又可以将这部分对代码的改动给剥离出去。因此，对于整个系统的架构与设计，笔者建议读者可以多思考，多总结，慢慢就可以将一个系统设计得越来越好。

至此，在iOS平台中，已成功将音频效果处理器和视频效果处理器集成到了视频录制项目中，而整个集成过程也比较简单，读者可以去代码仓库中找到本节对应的源码，然后尝试录制一个视频，再将这个视频与第7章的视频进行对比，看看第8章和第9章是否完成了当初我们想要的效果。

10.5　本章小结

至此，本书的第二部分就结束了，不得不说这一部分对于音视频的开发多么重要，因此用了很大篇幅介绍这些内容。笔者希望读者在接受新知识的同时，也要提高自己的系统架构能力。本章将音频与视频的特效处理库集成到系统中的操作之所以这么简单，是因为第7章中将模块拆分得比较合理，使用了面向接口编程（抽象基类ELImageOutput与协议ELImageInput），将整个系统向更高层次进行了抽象（抽象出Input、Processor、Output等各个模块）。相信随着本书的进行，读者也会慢慢理解一个好的系统架构给大家带来的好处。

前面所构建出的系统在音视频领域常被大家称为录播，而在接下来的章节中，笔者会继续带领大家进入直播领域，看看如何将系统集成到直播领域，并且会一步步分析直播领域与录播领域的差别是什么，以及应该做些什么就可以使得系统运行在直播领域。在一个直播App中，视频的录制端称为推流端，而播放器端称为拉流端，磁盘的I/O则变为网络的I/O。一般来说，在直播领域会使用第三方的CDN厂商作为中间的流媒体服务器，这样就可以以更快的速度和更便宜的带宽给用户带来更好的体验。但是一个直播App并不是只有了这三个角色就可以运行起来的，这三个角色只是最基础、最核心的部分。另外，还有聊天系统、礼物系统等，这些系统的构建在接下来的章节中都会有介绍，但是我们的重点还是会放在推流端和拉流端的网络适配上。让我们一起进入直播领域，看看一个直播App是如何构建起来的。

第11章　直播应用的构建

本章将会带领大家走进直播领域。直播的实现思路与大多数产品的实现思路一样，首先进行场景分析，然后在分析场景的过程中拆分直播系统的各个模块，并进行大致的技术选型，再依次介绍各模块的具体实现手段及其优缺点。

11.1　直播场景分析

在直播领域，大致可以分为两种类型的直播：一种是非交互式直播，另外一种是交互式直播。非交互式直播的典型场景有：2015年9月的反法西斯阅兵直播，某些体育直播等。这些直播因为其交互性不强，所以允许延迟（从视频中主体发生实际的行为到该行为被用户看到的时间）10s或者10s以上。非交互式直播的特点是：源（像阅兵直播、NBA直播、欧冠直播等）比较少，适合做多路转码（用户可以根据网络条件观看超清、高清、标清等多路视频）。交互式直播的典型场景有：秀场直播、游戏直播等。这些直播因为对主播和观众的互动性要求比较高，所以要求延迟在5s以内。交互式直播的特点是：源（像美女主播、游戏主播）比较多，不适合做多路转码，中间服务器只作为一个中转的角色。

直播中，传输的介质是网络，而网络中传播视频或者音频时需要使用对应的协议，目前适合直播场景的常用协议有如下几种。

·RTMP协议：长连接，低延时（3s左右），网络穿透性差。

·HLS协议：HTTP的流媒体协议，高延时（10s以上），跨平台性较好。

·HDL协议：RTMP协议的升级版，低延时（2s左右），网络穿透性好。

·RTP协议：低延时（1s以内），默认使用UDP作为传输协议。

因此，我们应该按照自己的场景来选择协议，如果是非交互式场景，则选择HLS协议更适合；如果是交互式场景，则选择HDL协议或者RTMP协议较合适。RTP协议常用于视频会议或直播的连麦场景中，不直接用于一对多的直播场景中。

接下来看看交互式直播场景下可以拆分为哪些模块。最基础、最核心的应该是推流系统、拉流系统和流媒体服务器（Live Server），并由这三部分共同组成整个直播系统的主播端和用户端之间在视频或者音频内容上的交互。整体流程是主播使用推流系统将采集的视频和音频进行编码，并最终发送到流媒体服务器上；用户端使用拉流系统将流媒体服

务器上的视频资源进行播放。整个过程是一种发布者/订阅者（Publisher/Subscriber）的模式，如图11-1所示。

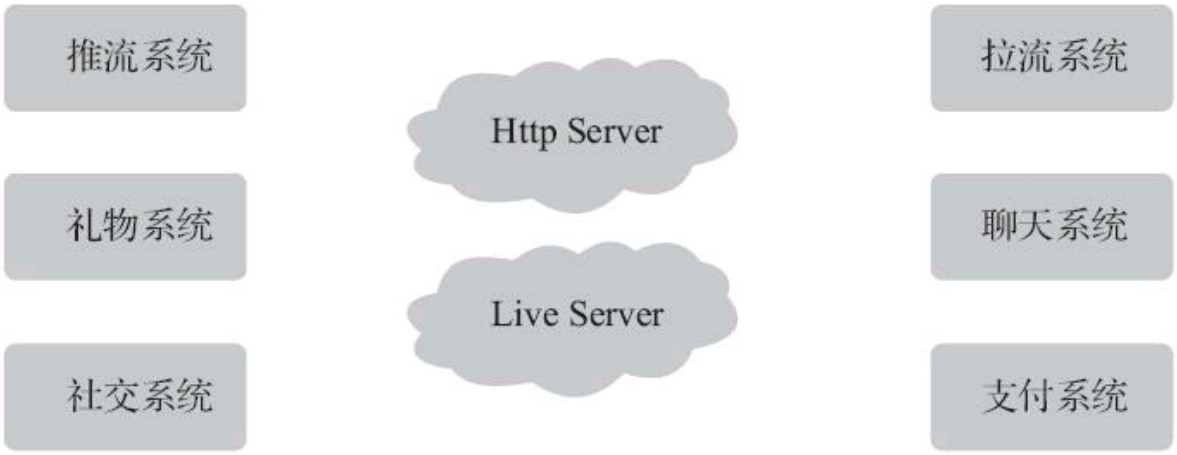


图　11-1

作为一个完整的产品，仅有这三个模块是远远不够的，最直观的，缺少礼物系统（可让观众给喜欢的主播送礼物），礼物系统可以为App提供礼物动效的展示等；既然观众能送礼物给用户，就要有充值功能，所以必须有支付系统来提供用户充值、主播提现等功能。在直播过程中，主播想和观众说话，观众在很短时间内就可以在视频中听到，但是观众想和主播进行交流，只能靠聊天系统来实现，聊天系统是用来建立观众到主播的反馈通道。直播这种行为实际上是一种社交行为，而任何一个直播产品都应该是一种社交产品，所以还需要社交系统，为观众和主播提供长期有效的社交行为（比如，根据关注关系适时推送“关注的主播”开播了，或者展示关注主播的开播列表以及视频回放列表等）。除了推流系统和拉流系统外的四个系统（见图11-1），都需要与服务器进行交互，我们称之为Http Server，即服务器模块。综上所述，最终整体结构如图11-1所示。

一般情况下社交系统包括但不限于以下的功能：

·第三方登录（包括微博、微信、QQ等）；

·第三方分享（包括微博、微信、QQ等）；

·手机号的登录与绑定；

·地理位置的使用；

·站内关系（关注与粉丝以及自己的Feed列表）；

·推送策略以及用户端收到推送之后的跳转行为；

·后台系统，用于提供给客服、运营人员操作榜单以及推荐用户等功能。

11.2　拉流播放器的构建

本节将基于视频播放器来构建用户端的拉流播放器。我们已在第5章详细讲解了播放器的结构，它是使用FFmpeg的libavformat模块来处理协议层与解封装层的细节，并使用FFmpeg的libavcodec模块来解码得到原始数据，最终使用OpenGL ES渲染视频以及使用对应的API来渲染音频。

相较于录播的实现，直播中的拉流播放器的使用时长（短则几十分钟，长则几个小时）会更长，所占用的CPU资源（观众界面还会有聊天的长连接、动画的展示等）也会更多，所以必须将第10章中讲解的硬件解码器集成进来。再者，网络直播中某些主播由于环境光的因素，或者网络带宽的因素，导致视频不是很清楚，所以要在拉流播放器中加入一个后处理过程，以增加视频的清晰度。这里以增加对比度来作为这个后处理过程的实现，当然在实际生产过程中，可以增加一些去块滤波器、锐化等效果器。

11.2.1 Android平台播放器增加后处理过程

要想打开网络连接，必须知道媒体源的协议，也要在编译FFmpeg的过程中打开对应的网络协议，比如HTTP、RTMP或者HLS等协议。解码器解码的目标是纹理ID，为此还建立了一个纹理对象的循环队列用于存储解码之后的纹理对象。整个解码器模块的运行流程如图11-2所示。

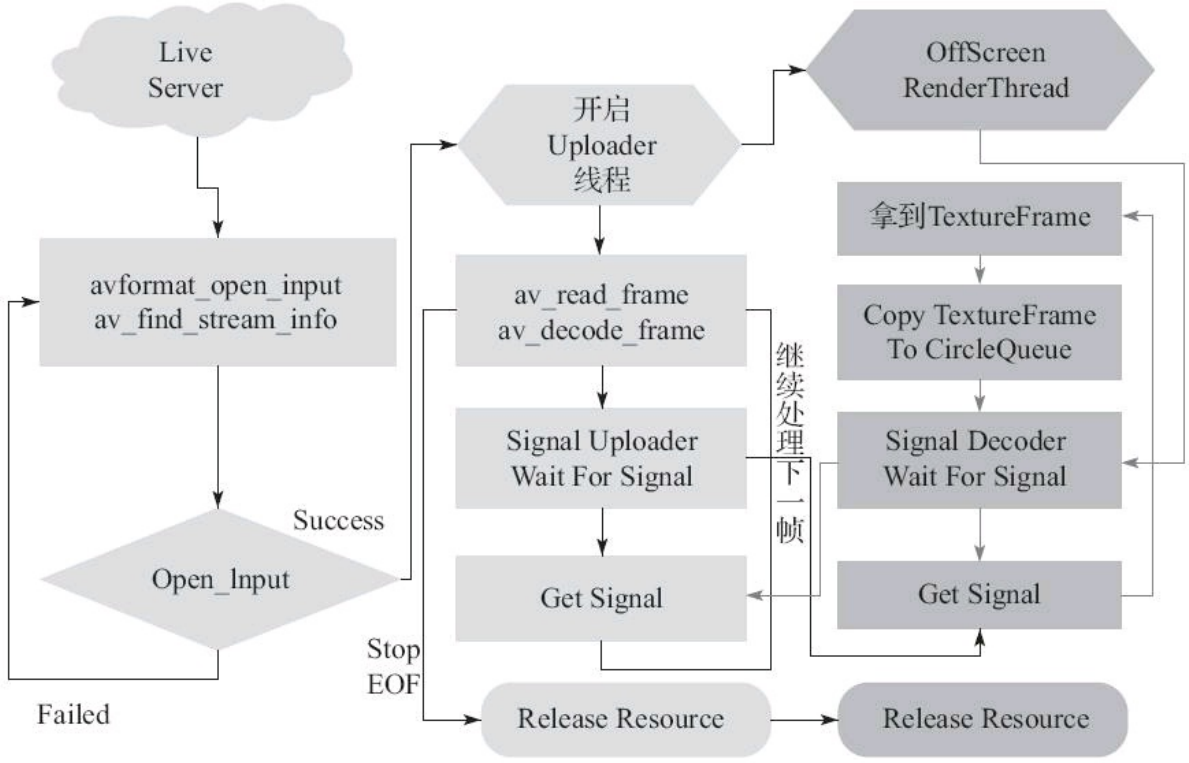


图 11-2

图11-2看起来有点复杂，下面逐一解释。首先是AVSync模块初始化解码器的过程，解码器会连接远程的流媒体服务器，如果打开连接失败，则启用重试策略重试几次；如果依然没能成功，则提示给用户；如果连接成功，则开启Uploader线程，即最右侧橙色部分。Uploader线程是一个OpenGL ES线程，当解码器是软件解码器实例的时候，这个线程的职责就是将YUV数据上传到显卡上并成为一个RGBA格式的纹理ID；当解码器是硬件解码器实例的时候，这个线程的职责就是将硬件解码器解码出来的OES格式的纹理ID转换为RGBA格式的纹理ID。在实现Uploader这个模块的过程中，要设定一个父类，然后对应于两个子类

（软件解码器与硬件解码器对应的Uploader）分别完成各自的职责。注意这里在为这个线程开启OpenGL ES上下文的时候，需要和渲染线程共享上下文环境，这样OpenGL ES的对象在这两个线程中才可以共同使用。当Uploader线程开始运行以后，会进入一个循环，循环一开始先阻塞（wait）住，等待解码线程发送Signal指令再去做上传纹理操作，之后进入下一次循环，也会先阻塞（wait）住，周而复始，完成整个纹理上传工作。

成功开启Uploader线程之后，初始化工作就结束了，等到AVSync模块中的解码线程开始工作后，解码器才会调用FFmpeg的libavformat模块进行解析协议与解封装（Demuxer），再调用具体实例（有可能是软件解码器，也有可能是硬件解码器）的解码方法。解码成功之后就会给Uploader线程发送一个Signal指令，之后这个解码线程就等待Uploader线程处理完这一帧视频帧之后，解码线程再继续运行，如图11-2中间部分所示。

待Uploader线程接收到Signal指令后，就会执行上传（转换）纹理的工作；而在转换成为一个RGBA格式的纹理对象后，就会调用回调函数（在初始化过程中，AVSync传递过来的回调函数）来处理这一帧视频帧，并将这一帧视频帧拷贝到循环纹理队列中。虽然将纹理对象拷贝到循环纹理队列中的行为是在Uploader线程中实现的，但是在AVSync的相关代码中，这也是为了降低各个模块的耦合度。所以这里要在AVSync模块中加入视频特效处理器，每当解码器中解码一帧纹理ID时，就交给视频特效处理器进行处理，待处理完毕之后再将这一帧纹理对象加入循环纹理队列中。当然，这里仅使用增加对比度作为特效处理器中的作用效果器，读者也可以将去块滤波器以及锐化效果器加入特效处理器中，以增加后处理的效果。

要实现上述功能，需要新建一个LiveShowAVSync的类，继承自类AVSync，并重写父类中的三个回调方法。第一个是当Uploader线程初始化成功之后的回调方法，代码如下：

```
void LiveShowAVSynchronizer::OnInitFromUploaderGLContext(EGLCore* eglCore,
            int videoFrameWidth, int videoFrameHeight) {
    videoEffectProcessor = new VideoEffectProcessor();
    videoEffectProcessor->init();
    int filterId = videoEffectProcessor->addFilter(0, 1000000 * 10 * 60 * 60,
            PLAYER_CONTRAST_FILTER_NAME);
    if(filterId >= 0){
        videoEffectProcessor->invokeFilterOnReady(filterId);
```

```
    }
    glGenTextures(1, &mOutputTexId);
    glBindTexture(GL_TEXTURE_2D, mOutputTexId);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, videoFrameWidth,
            videoFrameHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, 0);
    glBindTexture(GL_TEXTURE_2D, 0);
    AVSynchronizer::OnInitFromUploaderGLContext(eglCore,
            videoFrameWidth, videoFrameHeight);
}
```

如上述代码所示，由于这个回调函数的调用发生在Uploader线程中，并且Uploader线程已经准备好了OpenGL ES上下文，也绑定好了OpenGL ES上下文，所以这里就是初始化特效处理器的好时机。同时，还要初始化一个输出纹理ID，因为经过特效处理器处理后要有一个纹理ID作为输出，我们设定mOutputTexId来接受处理结束之后的纹理ID。

接着来看如何处理一帧视频帧，重写父类的处理视频帧的方法，代码如下：

```
void LiveShowAVSynchronizer::processVideoFrame(GLuint inputTexId, int width,
            int height, float position){
    GLuint outputTexId = inputTexId;
    if(videoEffectProcessor){
        videoEffectProcessor->process(inputTexId, position, mOutputTexId);
        outputTexId = mOutputTexId;
    }
    AVSynchronizer::processVideoFrame(outputTexId, width, height, position);
}
```

上述代码也很简单，如果视频特效处理器存在，就将处理器处理过的纹理ID（mOutputTexId）交由父类复制到循环纹理队列中去。最后一个需要重写的方法是销毁资源的方法，代码如下：

```
void LiveShowAVSynchronizer::onDestroyFromUploaderGLContext() {
    if (NULL != videoEffectProcessor) {
        videoEffectProcessor->dealloc();
        delete videoEffectProcessor;
        videoEffectProcessor = NULL;
    }
    if (-1 != outputTexId) {
        glDeleteTextures(1, &outputTexId);
```

```
    }
    AVSynchronizer::onDestroyFromUploaderGLContext();
}
```

因为这个方法的调用也是发生在Uploader线程中的，所以也符合视频特效处理器的销毁方法需求，同时，还要删除分配的这个输出纹理ID，最后调用父类的销毁资源方法。

至此，后处理过程就集成到了播放器中，在网络状态下这对视频的清晰度也会有一个增强的效果。虽然这里只新增了类而没有修改旧的代码，但也达到了增加功能的效果，可见，最初有一个合理的架构设计多么重要。

除了新增这个功能外，我们还有比较重要的适配工作要做。读者可否还记得在AVSync模块里定义了三个宏用来控制解码线程的启动和暂停，以及音视频对齐策略？之前定义的宏如下：

```
#define LOCAL_MIN_BUFFERED_DURATION                 0.5
#define LOCAL_MAX_BUFFERED_DURATION                 0.8
#define LOCAL_AV_SYNC_MAX_TIME_DIFF                 0.05
```

这三个值放在网络环境中就需要做适配，否则会出现视频的卡顿，并且会因为对齐而影响视频的整体播放，更改之后的三个宏定义如下：

```
#define NETWORK_MIN_BUFFERED_DURATION               2.0
#define NETWORK_MAX_BUFFERED_DURATION               4.0
#define NETWORK_AV_SYNC_MAX_TIME_DIFF               0.3
```

使用这三个宏给全局变量赋值，需要重写父类的initMeta方法，代码如下：

```
void LiveShowAVSynchronizer::initMeta() {
    this->maxBufferedDuration = NETWORK_MAX_BUFFERED_DURATION;
    this->minBufferedDuration = NETWORK_MIN_BUFFERED_DURATION;
    this->syncMaxTimeDiff = NETWORK_AV_SYNC_MAX_TIME_DIFF;
}
```

这样，本地视频播放器制作好了播放网络视频的适配，拉流播放器

也可以使用，读者可以参考代码仓库中的源码，以便加深理解。

11.2.2　iOS平台播放器增加后处理过程

在iOS平台增加了硬件解码器的支持后，并没有像Android平台一样在解码器端进行非常大的改造，而是在渲染端进行了适配，改动之后的结构如图11-3所示。

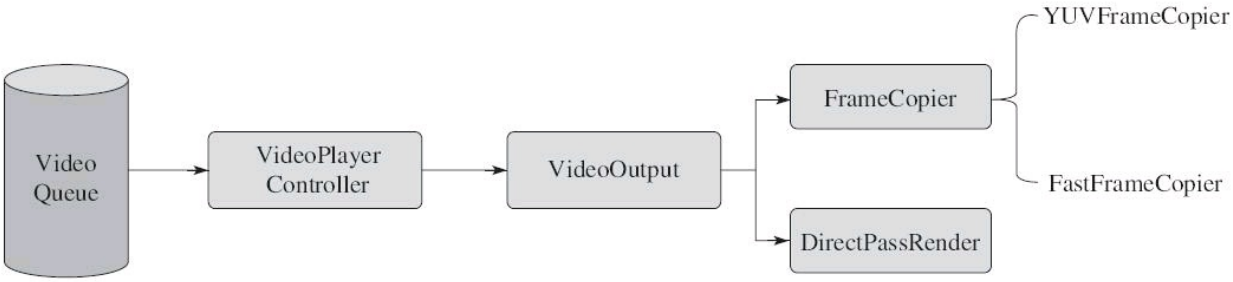


图　11-3

图11-3中，VideoPlayerController这个调度器会携带是否使用硬件解码器的参数来初始化VideoOutput，而VideoOutput中有一个属性为frameCopier，VideoOutput会根据是否使用硬件解码器而初始化不同类型的FrameCopier，如果使用硬件解码器，就实例化FastFrameCopier，如果没有使用硬件解码器，就实例化YUVFrameCopier。当VideoPlayerController从视频队列中取出一帧视频帧交给VideoOutput来渲染的时候，VideoOutput就会调用前面实例化好的FrameCopier并将VideoFrame类型的视频帧转化为一个纹理ID，然后VideoOutput就会绑定到displayFrameBuffer上，最后使用DirectPassRender将纹理ID渲染到RenderBuffer上，也就是渲染到Layer上，从而让用户可以看到。在上面的整个渲染过程中，我们要引入视频后处理效果，最好在FrameCopier之后插入，再将FrameCopier处理完的纹理ID交给新定义的VideoEffectFilter来做视频特效的处理，处理结束之后的outputTexId再由原来的DirectPassRender渲染到Layer上，最后将这个新的节点引入VideoOutput，整体结构如图11-4所示。

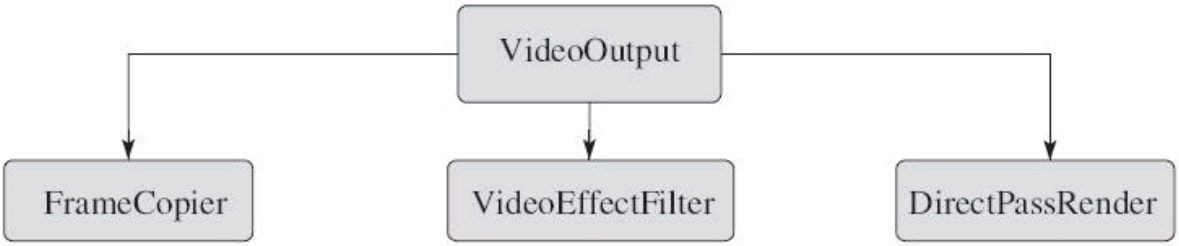


图　11-4

下面看看如何具体构建这个VideoEffectFilter。首先提供一个prepareRender方法，完成OpenGL ES相关资源的初始化，要求VideoOutput在OpenGL ES线程中调用这个方法，代码如下：

```
- (BOOL) prepareRender:(NSInteger) frameWidth height:(NSInteger) frameHeight;
{
    _frameWidth = frameWidth;
    _frameHeight = frameHeight;
    [self genOutputFrame];
    _processor = new VideoEffectProcessor();
    _processor->init();
    int filterId = _processor->addFilter(0, 1000000 * 10 * 60 * 60,
            PLAYER_CONTRAST_FILTER_NAME);
    if(filterId >= 0){
        _processor->invokeFilterOnReady(filterId);
    }
    return YES;
}
```

其中，genOutputFrame方法是生成这个类的输出纹理ID与FBO。至于如何生成纹理ID和FBO，并把这个纹理ID与FBO绑定起来，前面已介绍过，这里就不再展示具体的代码了。接下来就是真正的渲染过程，代码如下：

```
- (void) renderWithWidth:(NSInteger) width height:(NSInteger) height
        position:(float)position;
{
    glBindFramebuffer(GL_FRAMEBUFFER, _FrameBuffer);
    self.processor->process(_inputTexId, position, _outputTextureID);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

最后就是释放为后处理而分配的资源，代码如下：

```
- (void) releaseRender;
{
    if(_outputTextureID){
        glDeleteTextures(1, &_outputTextureID);
        _outputTextureID = 0;
    }
    if (_FrameBuffer) {
        glDeleteFramebuffers(1, &_FrameBuffer);
        _FrameBuffer = 0;
    }
    if (_processor) {
        _processor->dealloc();
        delete _processor;
        _processor = NULL;
    }
}
```

至此，VideoEffectFilter类就构建完毕。接着在VideoOutput类中将FrameCopier输出给VideoEffectFilter作为输入，而把VideoEffectFilter输出给DirectPassRender作为输入，这样渲染过程就集成进了后处理过程。当然，目前的后处理过程只是增强对比度的一个效果器，读者可以将去块滤波效果器、锐化效果器加入后处理过程中，这样会让整体播放效果有很大提升。

另外，针对视频源是网络流的特殊处理，要在VideoDecoder这个模块下给FFmpeg加入超时回调的方法，虽然在本地磁盘文件的读取过程中基本不会出现超时场景，但是由于网络环境过于复杂，所以这里要加入超时方法。在VideoDecoder类中，调用libavformat模块的打开链接之前，可给AVFormatContext设置超时回调函数，代码如下：

```
AVFormatContext *formatCtx = avformat_alloc_context();
AVIOInterruptCB int_cb  = {interrupt_callback, (__bridge void *)(self)};
formatCtx->interrupt_callback = int_cb;
```

其中interrupt_callback静态方法的回调函数实现如下：

```
static int interrupt_callback(void *ctx)
{
    __unsafe_unretained VideoDecoder *p = (__bridge VideoDecoder *)ctx;
    const BOOL isInterrupted = [p detectInterrupted];
    return isInterrupted;
}
```

在这个静态函数中调用detectInterrupted方法的实现很简单，就是判断当前时间戳和上一次接收到数据或者开始连接的时间间隔，如果大于超时时间（比如15s），则返回YES，代表已经超时，否则返回NO。若返回YES，则代表FFmpeg中阻塞的调用（比如read_frame、find_stream_info等）会立即返回。

二是对网络流的适配为更改缓冲区大小，由于在本地播放器中设置了minBuffer和maxBuffer作为控制解码线程的暂停和继续的条件，所以之前定义的宏如下：

```
#define LOCAL_MIN_BUFFERED_DURATION                    0.5
#define LOCAL_MAX_BUFFERED_DURATION                    1.0
```

在网络中，为了避免频繁卡顿，还要将控制解码线程暂停和运行的Buffer长度进行网络适配，新增宏定义如下：

```
#define NETWORK_MIN_BUFFERED_DURATION                  2.0
#define NETWORK_MAX_BUFFERED_DURATION                  4.0
```

在拉流播放器中就是使用以上两个宏定义来确定AVSync模块的缓冲区大小的。这样我们就完成了由本地播放器到网络拉流播放器的适配，读者可以参考代码仓库中的源码，以便于深入理解。

11.3　推流器的构建

本节构建主播端使用的推流工具，当然，也是根据前面章节中的视频录制应用进行改动适配。第7章已经构建了一个视频录制器，第8章和第9章为这个视频录制器增添了音频效果处理器和视频效果处理器。对于一个录播应用来说，已经比较完整了，但是对于直播应用，还需要做一些适配工作。下面看看整体结构，如图11-5所示。

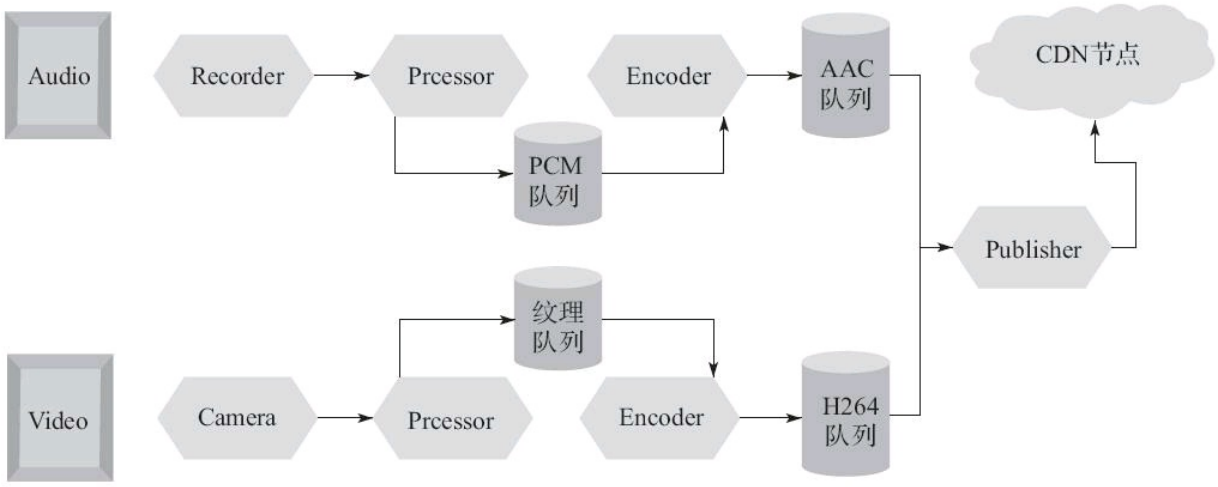


图　11-5

从整体结构来看，录播和直播的区别仅在于最终Muxer（图11-5中的Publisher）模块的输出不同，录播是向本地磁盘输出，而直播是向网络输出。网络不同于磁盘的是，有可能会出现网络波动甚至网络阻塞，所以要针对网络这种场景做对应的适配工作。

接下来分析复杂的网络环境。读者可以参考图11-6，主播端第一步请求Http Server的开播接口后会得到域名形式的推流地址；得到推流地址后，主播端将域名解析为实际的IP地址和端口号；将这个IP地址经过公有网络的各级路由器和交换机，最终找到实际的CDN厂商节点。从获得IP地址开始，影响整个连接通道的因素有很多，其中包括主播自己的出口网络、中间的链路状态，以及CDN厂商机房的节点链路情况等。影响连接通道的因素很多，如果都由开发者自己来解决，显然不合理。CDN厂商可以帮助开发者解决除主播自己出口网速以外的其他部分，当然，这也是CDN厂商存在的意义。但是，任何一家CDN厂商也没有100%的服务保证性，并且，主播自己的出口网络可能是小运营商，也

可能是教育网络，或者其他设备占用网络出口带宽。读者可以通过图11-6来分析整个网络情况。

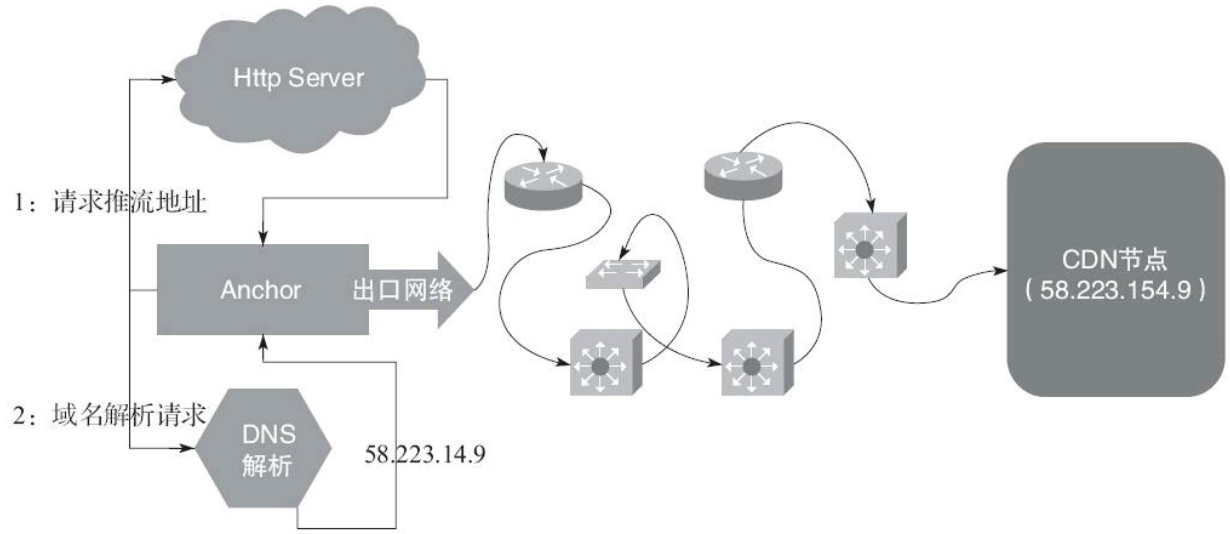


图　11-6

综上所述，必须做两件事情，才可以将视频录制器转换为可用的主播推流工具：第一件事情是如果网络超时，要通知给用户重新开播或者切换网络环境；第二件事情是当网络出现抖动的时候，我们要做丢帧来保证整个视频直播的延迟时间，以维持整场直播的交互性。

先来看第一件事情，即网络超时的设置。超时设置应该在Muxer模块中完成，由于Muxer模块使用的是FFmpeg中libavformat模块的封装层和协议层，所以这里设置的超时代码与我们在拉流端设置的超时代码类似，即在分配AVFormatContext之后，并在打开连接之前，设置超时回调，代码如下：

```
AVFormatContext* oc;
AVIOInterruptCB int_cb = { interrupt_cb, this };
oc->interrupt_callback = int_cb;
```

静态函数interrupt_cb的实现如下：

```
int RecordingPublisher::interrupt_cb(void *ctx) {
    RecordingPublisher* publisher = (RecordingPublisher*) ctx;
    return publisher->detectTimeout();
}
```

这个静态函数interrupt_cb中调用了RecordingPublisher类中的detectTimeout方法，而在detectTimeout方法中会针对当前时间戳的值减去上一次发送数据包的时间戳的值进行判断：如果大于我们设定的值（一般设置为5～15s），就返回1，代表超时，应停止发送数据包；否则返回0，代表没有超时，可以由FFmpeg的协议层继续发送数据包。在我们的发送线程中，如果发生了超时，则可以回调客户端代码，让客户端代码给用户弹出提示，让用户重新开播或者切换网络重新开播。

下面来看网络出现抖动，或者在弱网环境下的丢帧策略。读者可以先参考图11-5，图中有两种队列，分别是编码之前的原始数据队列和编码之后的编码队列。弱网丢帧策略常见的实现有两种：一种是丢弃原始数据队列中未编码的数据帧，另外一种是丢弃编码队列中的数据帧。这两种实现各有优缺点，无论采用哪种实现方式都以“不影响音视频的对齐”为第一准则。接下来分析两种丢帧策略的优缺点。丢弃原始数据帧的丢帧策略的优点是，节省了这部分丢弃帧占用编码器的资源，并且，由于是丢弃的原始数据帧，所以可以在任意时刻丢弃任意的音频视频帧。其缺点是，增加了直播的延迟时间，因为要保持中间队列有一个阈值。丢弃编码之后，数据帧策略的优点是减少了直播的延迟时间；缺点是丢弃的帧白白消耗了编码器资源，并且对于视频帧，只要丢帧，就丢掉一个GOP或者整个GOP的后半部分，否则会造成观众端不能正常观看视频。

不同的丢帧策略应用在不同的直播场景中，读者可以依据自己产品的场景来选择丢帧策略。笔者在实际开发过程中使用的丢帧策略是：视频丢弃是编码之后的视频帧，音频丢弃是编码之前的原始格式的音频帧。下面来看具体的实现。

对编码后的视频帧进行丢帧，只能丢弃一个完整的GOP（或者这个GOP后半部分非参考视频帧），或者这个GOP中剩余的视频帧，因为P帧需要参考前面的I帧与P帧才能被解码。对于B帧，需要双向参考（直播中一般不使用B帧，仅用I帧和P帧）。某些策略会保留GOP中的I帧，但是I帧是GOP中容量最大的视频帧；而某些策略是丢弃GOP中后半部分的P帧，直到这个GOP中仅剩余I帧的时候，再把I帧丢弃。第二种策略是一种可取的策略，但是为了简单考虑，最终的丢帧策略是：要丢弃就丢弃整个GOP（如果这个GOP已经发送出去了部分I帧和P帧，则丢掉这个GOP中剩余的视频帧）。如果读者想只丢弃GOP中后半部分的P帧

策略，在后续代码中进行更改也很简单。

丢弃了视频帧后，为了不影响音画的对齐效果，也应该丢弃同等时间的音频数据。但是，丢弃的那些视频帧总时长是多少呢？我们不可以只通过fps计算这些视频帧所代表的时长，而应该计算出视频帧每一帧持续的时间是多长，必须精确计算。因为fps对于Camera的影响是在一定时间范围内，所以在一定时间内连续的一段视频帧数目是可以被限定在这个fps之内的，但是，对于某一小段时间却不能保证满足fps要求的限制。那如何计算每一帧视频帧的精确时长呢？只能将Camera采集的视频帧都打上一个相对时间戳，编码时，要在编码成功第二帧时才赋值第一帧的duration信息，并把第一帧放入编码后的视频队列中，代码如下：

```
bool LivePacketPool::pushVideoPacketToQueue(LiveVideoPacket* videoPacket) {
    if (NULL != videoPacketQueue) {
        // 为了计算当前帧的Duration, 所以延迟一帧放入Queue中
        if(NULL != tempVideoPacket){
            int packetDuration = videoPacket->timeMills –
                        tempVideoPacket->timeMills;
            tempVideoPacket->duration = packetDuration;
            videoPacketQueue->put(tempVideoPacket);
        }
        tempVideoPacket = videoPacket;
    }
    return dropFrame;
}
```

其中，tempVideoPacket是一个全局变量，代表durtion属性还没有被赋值的视频帧，当被赋值之后，它会被加入视频队列中。接下来看看如何丢弃整个GOP或者GOP中没有发送出去的视频帧，并计算丢弃的视频帧所占用的总时长。在丢弃之前要给整个队列上锁，执行完操作之后解锁，代码主体如下：

```
int LiveVideoPacketQueue::discardGOP() {
    int discardVideoFrameDuration = 0;
    LiveVideoPacketList *pktList = 0;
    pthread_mutex_lock(&mLock);
    // 执行丢帧操作
    pthread_mutex_unlock(&mLock);
    return discardVideoFrameDuration;
}
```

执行丢帧操作的逻辑也比较简单，会先判断当前第一个元素是否是关键帧，如果是关键帧，则将布尔型变量isFirstFrameIDR设置为true，代码如下：

```
bool isFirstFrameIDR = false;
if(mFirst){
    LiveVideoPacket * pkt = mFirst->pkt;
    if (pkt) {
        int nalu_type = pkt->getNALUType();
        if (nalu_type == H264_NALU_TYPE_IDR_PICTURE){
            isFirstFrameIDR = true;
        }
    }
}
```

然后循环队列中所有的视频帧，并判断视频帧类型。如果帧类型不是关键帧，则丢弃这一帧，并把这一帧的时间长度加到丢弃帧时间长度的变量上；如果是关键帧，就先判断isFirstFrameID变量是否为true，如果是true，则先置为false，然后丢弃这一帧并将帧长度加到丢弃帧时间长度的变量上；如果是false，则代表已经删除了一个GOP，应该退出。代码如下：

```
LiveVideoPacketList *pktList = 0;
for (;;) {
    if (mAbortRequest) {
        discardVideoFrameDuration = 0;
        break;
    }
    pktList = mFirst;
    if (pktList) {
        LiveVideoPacket * pkt = pktList->pkt;
        int nalu_type = pkt->getNALUType();
        if (nalu_type == H264_NALU_TYPE_IDR_PICTURE){
            if(isFirstFrameIDR){
                isFirstFrameIDR = false;
                discardVideoFrameDuration += pkt->duration;
                relesse(pktList);
                continue;
            } else {
                break;
            }
        } else if (nalu_type == H264_NALU_TYPE_NON_IDR_PICTURE) {
            discardVideoFrameDuration += pkt->duration;
            relesse(pktList);
        }
    }
}
```

上述方法的调用端，就是指当编码之后的视频帧要放入队列中之前，要判断当前视频队列的大小和设置Threshold（阈值）的关系，如果超过阈值，则说明当前网络发生抖动或者处于弱网环境下，应执行丢帧逻辑。待丢弃完一个GOP之后，再以这个丢弃掉视频帧的时间长度参数去丢弃音频数据。因为丢弃的音频帧是原始数据帧，而对于PCM队列中每个元素都是固定长度（暂时设置为40ms）的一个buffer，所以代码如下：

```
bool LivePacketPool::discardAudioPacket() {
    bool ret = false;
    LiveAudioPacket *tempAudioPacket = NULL;
    int resultCode = audioPacketQueue->get(&tempAudioPacket, true);
    if (resultCode > 0) {
        delete tempAudioPacket;
        tempAudioPacket = NULL;
        pthread_rwlock_wrlock(&mRwlock);
        totalDiscardVideoPacketDuration -= (40.0 * 1000.0f);
        pthread_rwlock_unlock(&mRwlock);
        ret = true;
    }
    return ret;
}
```

上述函数的实现比较简单，调用的地方就在原来的音频编码适配器（AudioEncodeAdapter）的getAudioPacket方法中。至此，完成了丢帧策略的实现，主播端的推流工具由视频录制器改造而成。

11.4　第三方云服务介绍

在开发音视频的App的过程中，不得不和第三方的云服务打交道，因为这些CDN厂商对于视频的带宽可以提供更便宜的价格（相较于IDC的带宽），而对于视频的访问速度也可以提供更快速的访问通道。无论是在录播场景下还是在直播场景下，使用CDN厂商都要优于自己搭建一套存储服务与流媒体服务。

这里首先解释一下CDN，CDN的全称是Content Delivery Network，其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈与环节，使内容传输得更快、更稳定。一般实现手段是在各处放置节点服务器，节点服务器呈树形结构，用户能直接访问到的是边缘（叶子）节点服务器，如果边缘节点服务器没有用户所要访问的资源，就向它的上一级节点服务器获取数据；如果还没有（不同厂商提供的级数不同），就向开发者配置的文件实际地址（回源地址）获取；如果边缘节点有用户所要访问的资源，就直接给用户，并在用户选择边缘节点的策略上，CDN厂商会按照负载均衡、链路调度等服务安排用户就近获取资源，降低网络拥塞，提高用户访问的响应速度。而CDN去源站服务器获取源文件的这个过程称为回源，对于流媒体资源，回源率越小越好，否则源站服务器的I/O将不堪重负。在日常工作中，不只最终用户的视频作品会存储在CDN上，甚至我们的图片文件、音频文件，甚至前端工程师的js文件和css文件都可以存储在CDN上。当然，在录播场景下，用户视频作品的存储以及访问，开发者可能会用到某些CDN厂商。最终开发者或者公司和CDN厂商一般会按照几部分服务计算资费，最主要的就是网络带宽的费用，这部分的费用要比使用我们自己机房的带宽费用便宜很多，至于存储、转码等服务，则依据开发者自己的使用情况进行收费。

在直播过程中，第三方的CDN厂商又可以给我们提供哪些服务呢？笔者根据自己接触的CDN厂商，总结了以下几个主要服务。

·直播转发服务：提供快速、稳定的直播转发服务，接受主播端推上来的视频流，并可以转发给所有的订阅者（观众端），一般CDN厂商对于视频会做一些关键帧缓存等优化，可以让观众端以最快的速度看到视频。

·直播存档服务：在整个直播过程中可以存储视频，便于客户的产品可以沉淀视频内容，后续可以继续观看这个视频。

·直播转码服务：提供多协议、多分辨率、多码率的多路转码服务，产品的多个终端可能需求的拉流协议是不同的，比如网页需要HLS协议，而客户端需要HDL协议，或者在一些大型直播以及一些专有的体育赛事直播中需要给用户提供超清、高清、标清等多路视频流。

·视频抽帧服务：可以提供在可配置时间内（1～10s）抽取一帧图像进行存储，可以给客户提供内容审核、实时预览等功能。

·推流、拉流客户端SDK：可以提供推流端和拉流端的视频基础服务，可以让客户花更多的时间在自己的产品和社交功能上。缺点就是当与系统中的动画以及其他页面跳转等细节出现不兼容性问题时会比较麻烦。因此，是否采用SDK，读者可以根据自己的业务场景以及产品阶段进行选择。

除了上述基础的服务外，某些CDN厂商也会提供其他可编程接口、离线处理接口等服务。我们最常用的服务以及应用场景主要包括以下几方面。

·直播转发服务：在直播过程中作为中转服务器，开发者的Http服务器会分配这个中间地址，并发送给推流端，推流端将视频流推送到这个中转服务器上，而观众端也会到中转服务器（与推流的服务器不一定相同，CDN厂商会提供最优链路解决方案）上获取直播流。另外，在自己的Http服务器返回推流地址的时候，可以加入防盗链机制（推流地址会使用时间戳加密后进行验证）增加安全性。

·直播存档服务：在直播过程中，我们的App会将主播端直播出来的视频实时转码为一个MP4文件存储到CDN上，以便后续提供视频回放服务。

·直播转码服务：使用这个服务将视频实时转码为HLS的视频流，供网页播放服务使用，一般情况下不会使用到多分辨率以及多码率的服务，如果有一些特殊活动，可以提前申请这样的服务。

·视频抽帧服务：使用CDN厂商提供的这个服务，每隔5s获取一帧图像进行展示，以供内容审核人员针对一些违规视频进行处理。

合理使用CDN厂商提供给我们的服务，可以提升开发效率，缩短开发周期，可以把我们有限的精力投入到产品的打磨上，让我们在自己的细分市场或者垂直领域快速地进行迭代。但是使用CDN厂商也有一定的弊端，比如CDN厂商提供了服务的稳定性，如果这家CDN厂商死掉了，那很有可能导致我们的产品处于不可用状态，所以在实际的开发中，要有多家CDN厂商备选，可以进行热切换，最好的方案就是我们自己再搭建一套系统，以便在所有第三方服务都挂掉的时候，也可以保证产品的可用性。

11.5　礼物系统的实现

对于一个直播App，礼物展示系统的实现是非常重要的，它实现的好坏直接影响整个产品的收入情况。因此，我们在考虑礼物系统实现的时候，应该思考以下几个方面的问题。

·礼物系统的性能怎么样。

·礼物系统将来的扩展性如何。

·当开发一个新动画的时候，开发成本是多少，其中包括开发时间多长，参与人员由哪几部分组成等。

下面列举几种常用的实现手段。

第一种手段是使用各个平台自身提供的API来实现动画，比如iOS平台使用CALayer动画SpriteKit来实现，Android平台使用自己的Canvas来实现。针对这种实现手段，我们来分析以上几个问题，礼物系统的性能在iOS上没问题，在Android上可能要差一些；将来的扩展性并不会太好，将来可能会出现复杂的动画，比如类似碰撞检测的动画就很难实现；当开发一个新动画的时候，开发成本比较大，因为需要两个客户端开发人员和设计人员共同开发。对于设计人员输出的图片尺寸可能也不一样，需要不断地和两端开发人员进行调试，以及适配各种机型。

第二种手段是使用OpenGL ES来开发一套自己的动画引擎，前期开发成本很高，需要兼容粒子系统（使用Particle Designer设计的配置文件可以直接运行到系统中），甚至能兼容设计人员使用AE（After Effect，是Adobe的一款专门设计视频特效的图形处理软件）产出的动画特效。礼物系统的性能没有问题，将来的扩展性主要看最初自己的设计，但是遇到特殊情况，比如需要路径和碰撞检测的场景很难实现；对于开发成本，由于是跨平台的系统，需要的开发人员不多，但是需要精通OpenGL ES，也就是说，对开发人员的要求比较高，而与设计人员的沟通成本比较小。

第三种手段是使用现有的一些游戏引擎来实现动画，比如Cocos2dX、libGDX等。这里以最为流行的Cocos2dX为例来看上面提到的几个问题，Cocos2dX使用OpenGL ES作为绘制引擎，所以效率方面没

有问题。Cocos2dX是一款游戏引擎，在扩展性方面肯定是最强的，不论是碰撞检测，还是其他场景都比较容易实现；至于开发成本，由于是使用C++语言开发的，所以比较简单，但是需要开发人员学习Cocos2dX的API，因为这是一项跨平台的技术，所以整体的开发成本并不高。缺点就是引入Cocos2dX引擎会增加App的体积。

其他手段，如Airbnb的工程师发布的Lottie项目，这个项目可更简单地为原生项目添加动画效果，直接支持AE的动画特效，并支持动画的热更新操作，可以有效减小App的体积，且支持Android、iOS等平台。但是，由于这些技术业界使用的并不是太多，所以笔者不在本书中详细介绍。

下面将介绍Cocos2dX项目在Android和iOS设备上的运行原理，然后介绍Cocos2dX的关键API，最后利用这些API实现一个动画。

11.5.1 Cocos2dX项目的运行原理

如何构建项目这里就不做过多介绍了，大家可以根据官方文档进行构建。本节重点介绍Cocos2dX项目在Android平台和iOS平台上如何运行，如果读者面前有开发环境，可以打开代码仓库中的Android工程或者iOS工程跟随笔者进行分析。

1.Android项目的运行

下面来看Android工程，这里要使用到Cocos2dX项目提供的jar包以及我们自己编写的so库。首先配置jar包，可在build.gradle中进行，至于so库，可暂且假设已经编译出来。前面笔者提到过Cocos2dX也是基于OpenGL ES引擎绘制的，所以在Android上需要使用GLSurfaceView作为Cocos2dX的绘制目标，而jar包里提供的类Cocos2dxGLSurfaceView就是我们要使用的GLSurfaceView。先创建这个View对象：

```
Cocos2dxGLSurfaceView glSurfaceView = new Cocos2dxGLSurfaceView(this);
```

然后给这个GLSurfaceView设置EGL的显示属性，并设置Renderer为jar包里提供的专有类Cocos2dxRenderer：

```
mGLSurfaceView.setCocos2dxRenderer(new Cocos2dxRenderer());
```

再来看Cocos2dxRenderer内部具体的关键生命周期方法onSurfaceCreated，该方法里的第一行代码就调用了nativeInit方法，而Renderer的nativeInit方法最终会调用Native层，Cocos2dxRenderer类对应的Native层的源码文件是javaactivity-android.cpp，虽然不同版本的实现不同，但不论是在JNI_OnLoad方法中，还是在nativeInit方法中，都可以调用到以下这个方法：

```
cocos_android_app_init
```

这个方法会和so库中的main.cpp文件的实现连接起来，代码如下：

```
void cocos_android_app_init (JNIEnv* env, jobject thiz){
    LOGD("cocos_android_app_init");
    AppDelegate *pAppDelegate = new AppDelegate();
}
```

这样就可以让类Application的单例引用指向我们自己写的APPDelegate（继承自Application类）了。而在接下来的nativeInit方法中，会给Director设置GLView，代码如下：

```
director->setOpenGLView(glview);
```

GLView是一个接口，Cocos2dX在Android平台和iOS平台有各自的实现，分别完成一些平台相关的操作，比如viewPort、getSize、SwapBuffer等操作，面向接口编程的好处显而易见，正是因为这种设计才可以让Cocos2dX可以跨平台运行。在nativeInit方法中有一个最关键的调用，它能让整个引擎运行起来，方法如下：

```
cocos2d::Application::getInstance()->run();
```

上述流程会让整个引擎委托给我们书写的类来完成操作。而在APPDelegate中是如何实现的，会在11.5.2节继续讲解。在Renderer的生命周期方法onDrawFrame中会调用到nativeRender方法，而nativeRenderer方法也会调用到Native层，在Native层中可以看到如下调用：

```
cocos2d::Director::getInstance()->mainLoop();
```

mainLoop方法是Director类中要绘制内部所有场景（Scene）的地方，这样就可以不断地绘制整个动画。

综上所述，可依靠GLSurfaceView内部的渲染线程调用Renderer的onDrawFrame方法将整个渲染过程跑起来。至于上面提到的Cocos2dX的入口类Director以及关键API，后续章节会继续介绍。

2.iOS项目的运行

运行iOS项目之后，可以先找到源码文件main.m，这个文件是整个App的入口类，这里可以将AppController这个类作为整个App的生命周期方法的代理类。在这个类中，首先声明了一个变量，如下：

```
static AppDelegate s_sharedApplication;
```

声明这个变量的目的是让Cocos2dX的Application入口交由APPDelegate这个类，由于Application是单例模式设计的，而APPDelegate又继承自Application类，从而达到了委托给APPDelegate这个类的目的。下面看这个类中的启动方法：

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptior
    (NSDictionary *)launchOptions;
```

在这个启动的方法中，首先取出Application，然后设置OpenGL上下文的属性，接下来利用提供的CCEAGLView构造一个UIView，并将这个View以subView的方式加入ViewController中，最终给Director设置这个构造出来的GLView，以及调用Application的run方法，代码如下：

```
cocos2d::GLView *glview = cocos2d::GLViewImpl::createWithEAGLView(eaglView);
cocos2d::Director::getInstance()->setOpenGLView(glview);
cocos2d::Application::getInstance()->run();
```

run方法会调用到Cocos2dX引擎定义在AppDelegate中的生命周期方法，然后在源码中可以发现run方法最终会调用Director的startMainLoop：

```
[[CCDirectorCaller sharedDirectorCaller] startMainLoop];
```

run方法还会使用CADisplayLink来做一个定时器，按照设置的fps信息调用OpenGL ES的渲染操作，而关键的OpenGL ES操作都在CCEAGLView中实现，在Director中的关键操作（比如SwapBuffer）则会委托给GLView来完成，这样它们就又到达了CCEAGLView类中，从而让整个Cocos2dX引擎实现了在iOS平台的运行。相较于Android平台，iOS平台的运行原理比较简单，大家可以理解Cocos2dX是如何通过面向

接口编程达到实现跨平台特性的。

11.5.2 关键API详解

本节将介绍Cocos2dX里的关键API。不过，这里不再区分平台，而是使用一套跨平台的代码。对于Cocos2dX引擎来讲，最重要的是Director类，就像它的名字一样，它是整个游戏或者动画的导演，控制着内部所有场景（Scene）的渲染，可以进行显示、切换等操作。对于场景，大家可以将其理解为一个界面，类似于Android里的Activity，或者iOS里的ViewController，在这个场景中可以有很多图层（Layer），每一个图层就类似于Photoshop中的图层，所有上述这些对象共同构成了Cocos2dX引擎。

接下来读者可以看到在上面提到的AppDelegate，这个AppDelegate就是Cocos2dX引擎委托给开发者的程序入口，AppDelegate也必须继承自cocos2d：：Application，并重写这里的生命周期方法，如下：

·当应用程序启动的时候会调用方法applicationDidFinishLaunching。

·当应用程序进入后台的时候会调用方法applicationDidEnterBackground。

·当应用程序回到前台的时候会调用方法applicationWillEnterForeground。

由于iOS平台不允许App进入后台之后还使用OpenGL ES渲染，并且进入后台之后也没必要为用户展示动画，所以当应用程序进入后台的时候，应该调用停止动画的方法：

```
Director::getInstance()->stopAnimation();
```

调用了上述方法之后，Director中的渲染行为就不会再触发，内部实现会把invalid的变量设置为true。在mainLoop方法中，待判断出这个变量是true之后，Director就不会去渲染内部的场景了。而当App又重新回到前台的时候，则应该继续启用动画：

```
Director::getInstance()->startAnimation();
```

这个方法的内部实现又会把invalid变量设置为false，而在Director内部的mainLoop，就会继续渲染内部的场景，用户就可以继续看到动画了。接下来看看最重要的生命周期方法applicationDidFinishLaunching，这个方法是Cocos2dX引擎留给开发者设置参数与绘制操作等程序入口的地方。先来看设置Director的代码部分：

```
auto director = Director::getInstance();
director->setDisplayStats(false);
director->setAnimationInterval(1.0 / 45);
director->setClearColor(Color4F(0, 0, 0, 0));
```

首先获得Director的实例，然后将显示fps状态的开关关闭，接下来设置fps，这里设置为一秒钟45帧的帧率，最后一行代码设置为背景颜色，其实这个颜色是每次绘制最开始使用glClearColor时所用的颜色。由于设备分辨率具有多样性，设计人员在调整动画效果或者游戏效果的时候也只会设计一个标准分辨率，因此，适配不同分辨率的机器在Cocos2dX中的实现如下：

```
static cocos2d::Size designResolutionSize = cocos2d::Size(480, 320);
static cocos2d::Size smallResolutionSize = cocos2d::Size(480, 320);
static cocos2d::Size mediumResolutionSize = cocos2d::Size(1024, 768);
auto glview = director->getOpenGLView();
if(!glview) {
    glview = GLViewImpl::create("changba-cocos");
    director->setOpenGLView(glview);
}
glview->setDesignResolutionSize(designResolutionSize.width,
            designResolutionSize.height, ResolutionPolicy::NO_BORDER);
Size frameSize = glview->getFrameSize();
if (frameSize.height > smallResolutionSize.height){
    director->setContentScaleFactor(MIN(mediumResolutionSize.height/
            designResolutionSize.height, mediumResolutionSize.width/
            designResolutionSize.width));
}else{
    director->setContentScaleFactor(MIN(smallResolutionSize.height/
            designResolutionSize.height, smallResolutionSize.width/
            designResolutionSize.width));
}
```

从以上代码可以看到，首先会取出Director的绘制目标GLView，然后会给这个GLView设置进入原始的分辨率，并对比GLView的大小，给Director设置一个Scale系数，而Cocos2dX在实际绘制过程中会使用Scale

系数进行屏幕分辨率的适配。

完成Director的设置后，接下来就来实例化一个场景，让Director来显示这个场景。

```
auto scene = AnimatinScene::create();
director->runWithScene(scene);
```

可以看到这里所有的类型都是auto类型的，代表自动回收对象，而最后一行代码是告诉Director运行HellWord这个场景。

对AppDelegate的介绍就到这里。下面来看HelloWord这个场景是如何构建的。要创建一个场景，必须继承自cocos2d：：Scene，然后重写init方法，因为Scene里默认的create方法是调用了init方法，所以只有重写init方法才可以在场景的创建过程中完成我们的逻辑。

```
bool AnimationScene::init() {
    if (!Scene::init()) {
        return false;
    }
    auto keyBoardLayer = KeyBoardLayer::create();
    addChild(keyBoardLayer, 1, 1);
    return true;
}
```

第一行代码调用了父类的init方法，然后创建KeyBoardLayer，并调用addChild方法将Layer加入我们的场景中，而KeyBoardLayer就是设置的一个菜单Layer，上面可以增加动画按钮与退出按钮，点击不同的按钮有不同的行为。

接下来看KeyBoardLayer的内部实现。首先Layer要继承自cocos2d：：Layer，然后要重写init方法，在init方法中需要调用父类的初始化方法，代码如下：

```
bool KeyBoardLayer::init() {
    // 1:调用父类的初始化
    if (!Layer::init()){
        return false;
    }
    Size visibleSize = Director::getInstance()->getVisibleSize();
```

```
    Vec2 origin = Director::getInstance()->getVisibleOrigin();
    // 2:增加关闭按钮
    // 3:增加动画按钮
}
```

可以看到以上代码分为了三部分，第一部分是调用父类的初始化方法，然后取出屏幕的宽度与起始点位置，便于后续添加按钮来计算位置；第二部分是给Layer增加一个关闭按钮，代码如下：

```
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png'
            CC_CALLBACK_1(KeyBoardLayer::menuCloseCallback, this));
closeItem->setPosition(Vec2(origin.x + visibleSize.width –
            closeItem->getContentSize().width/2 ,
            origin.y + closeItem->getContentSize().height/2));
auto menu = Menu::create(closeItem, NULL);
menu->setPosition(Vec2::ZERO);
this->addChild(menu, 1);
```

从以上代码可以看到，这里选择了两张图片分别作为这个菜单项的普通状态和选中状态，点击的监听方法是本类的menuCloseCallback方法，然后设置位置，并且加入到创建的Menu里去，最终将这个Menu加入到Layer中，而menuCloseCallback方法的实现如下：

```
void KeyboardLayer::menuCloseCallback(Ref* pSender) {
    Director::getInstance()->end();
}
```

以上代码直接调用Director的end方法可结束整个Cocos2dX的绘制。接下来看Layer的init方法中的第三部分——添加一个动画按钮，代码如下：

```
auto animationBtn = Label::createWithTTF("show", "fonts/Marker Felt.ttf", 10)
animationBtn->setAnchorPoint(Point(0, 0));
auto listener = EventListenerTouchOneByOne::create();
listener->setSwallowTouches(true);
listener->onTouchBegan = [] (Touch *touch, Event *event) {
    if (event->getCurrentTarget()->getBoundingBox().
            containsPoint(touch->getLocation())) {
        Scene* scene = Director::getInstance()->getRunningScene();
        AnimationScene* animation = (AnimationScene*) scene;
        animation->showAnimation();
        return true;
    }
```

```
        return false;
    };
    Director::getInstance()->getEventDispatcher()->
                addEventListenerWithSceneGraphPriority(listener, animationBtn);
    animationBtn->setPosition(Vec2(origin.x + 0, origin.y));
    this->addChild(animationBtn, 1);
```

虽然这里称为增加了一个按钮，实际上使用的是Cocos2dX提供的Label控件，首先加载一个字体，然后按照文字（show）与字体大小（10）创建一个label，并创建一个监听事件，这个监听事件被触发的时候会取出当前Director运行的场景，并调用showAnimation方法。接下来将label绑定这个监听事件，最后给label设置位置，并加入Layer中。

至此关键的API也已经介绍完成。

11.5.3　实现一款动画

本节会带领大家实现一个亲吻的动画展示，首先来看由所有帧组成的一张大图片，如图11-7所示。

图　11-7

图11-7是将序列帧图像合并在一起得到的，接着来看将整张图片裁剪成为动画帧序列的plist配置文件，如图11-8所示。

plist配置文件描述了每一帧图片应该在整张图片的位置以及旋转角度，Cocos2dX引擎可以解析这个plist配置文件并结合原始图片最终形成序列帧。设计人员如何生成plist配置文件以及合并整张图片呢？答案是使用TexturePacker工具，plist配置文件也是可以被大部分游戏引擎解析的，其中包括Cocos2dX、libGDX、Unity3D等。当设计人员利用AE开发完动画之后，然后导出png序列图，之后使用TexturePacker制作成为plist配置文件与大图的形式，然后提供给开发者使用。接下来看看如何利用整张图片与这个配置文件完成动画的展示。

▼frames	Dictionary	(11 items)
▼el_kiss00.png	Dictionary	(5 items)
frame	String	{{442,2},{190,278}}
offset	String	{44,-49}
rotated	Boolean	NO
sourceColorRect	String	{{99,110},{190,278}}
sourceSize	String	{300,400}
▶el_kiss01.png	Dictionary	(5 items)
▶el_kiss02.png	Dictionary	(5 items)
▶el_kiss03.png	Dictionary	(5 items)
▶el_kiss04.png	Dictionary	(5 items)
▶el_kiss05.png	Dictionary	(5 items)
▶el_kiss06.png	Dictionary	(5 items)
▶el_kiss07.png	Dictionary	(5 items)
▶el_kiss08.png	Dictionary	(5 items)
▶el_kiss09.png	Dictionary	(5 items)
▶el_kiss10.png	Dictionary	(5 items)
▼metadata	Dictionary	(5 items)
format	Number	2
realTextureFileName	String	el_kiss.png
size	String	{1024,1024}
smartupdate	String	$TexturePacker:SmartUpdate:491988aa9bacfaee86f3477ef53c707e:1/1$
textureFileName	String	el_kiss.png

图　11-8

在Cocos2dX中，每一个能运动的物体都可以理解为一个精灵，那我们先使用Cocos2dX提供的精灵缓存类来解析出所有的序列帧，代码如下：

```
SpriteFrameCache * cache = SpriteFrameCache::getInstance();
cache->addSpriteFramesWithFile("el_kiss.plist");
```

将plist配置文件解析完成后，所有的帧序列都存在于缓存中了。接下来利用名字创建一个精灵对象，代码如下：

```
auto kissSprite = Sprite::createWithSpriteFrameName("el_kiss00.png");
int randomY = random(170.f, screenHeight - 80.f);
kissSprite->setPosition(screenWidth + 30, randomY);
kissSprite->setOpacity(255);
this->addChild(kissSprite);
```

以上代码中先使用第一张图片创建了一个精灵对象，然后设置了位置与透明属性，由于这个位置是画在屏幕的右侧还要再加30个像素的地方，所以暂时看不到。最后将这个精灵加入这个场景中。接下来为这个精灵安排动画，动画分为三部分，第一部分是从屏幕右侧移动到屏幕中间，第二部分是重复3次整个序列帧动画，第三部分是重新移出到屏幕右侧。下面来看第一部分从屏幕右侧移动到屏幕中看得见的位置，代码如下：

```
auto kissFirstStepMoveAction = MoveTo::create(0.3,
            Vec2(screenWidth - 200, randomY));
auto kissFirstStepEaseOutAction = EaseOut::create(kissFirstStepMoveAction, 2)
auto kissFirstMoveTargetAction = TargetedAction::create(kissSprite,
            kissFirstStepEaseOutAction);
```

代码中的第一行定义了一个移动的动画，相当于从原来的位置历经0.3s时间移动到屏幕宽度减去200的位置（纵坐标不变，依然是randomY），然后使用EaseOut进行封装，主要是为了将匀速运动变成非匀速运动，最终使用TargetedAction在这个精灵上创建出这个动作。下面来看第二部分的动画，代码如下：

```
// 1:在缓存中拿出所有精灵帧
Vector<SpriteFrame *>animFrames(11);
char str[100] = {0};
for(int i = 0;i < 11 ;i++) {
    sprintf(str, "el_kiss%02d.png",i);
    SpriteFrame *frame = cache->getSpriteFrameByName(str);
    animFrames.pushBack(frame);
}
// 2:创建Animation
auto animation = Animation::createWithSpriteFrames(animFrames, 1.0 / 11, 3);
// 3:构建Action
auto kissAnimationAction = Animate::create(animation);
auto kissAnimationTargetAction = TargetedAction::create(kissSprite,
            kissAnimationAction);
```

可以看到以上代码段分为三部分，第一部分在精灵缓存池中拿出所

有的精灵帧放入一个数组中，第二部分利用这个数组中的精灵帧和延迟时间以及循环次数创建出Animation对象，第三部分利用这个Animation对象构造出动作。下面来看最后一部分的动画，代码如下：

```
auto kissLastStepMoveAction = MoveTo::create(0.2,
            Vec2(screenWidth + 30, randomY));
auto kissLastStepEaseInAction = EaseIn::create(kissLastStepMoveAction, 2);
auto kissLastMoveTargetAction = TargetedAction::create(kissSprite,
            kissLastStepEaseInAction);
```

这与第一部分的动画正好相反，是向屏幕的右侧移动，使用EaseIn将匀速运动封装为非匀速运动。最后将这三部分动画封装到一个序列动作中，并在结束的时候将这个精灵对象移除，之后将这个序列动作放入场景中执行，代码如下：

```
auto sequence = Sequence::create(kissFirstMoveTargetAction,
    kissAnimationTargetAction, kissLastMoveTargetAction,
    CallFunc::create(CC_CALLBACK_0(Node::removeFromParent, kissSprite)),
            NULL);
this->runAction(sequence);
```

至此，showAnimation方法就实现好了，这个方法完成了动画的展示，读者可以参考代码仓库中的源码进行分析，以便于深入理解。

11.6 聊天系统的实现

聊天系统也有很多手段可以实现，在直播App中使用的聊天系统不只用于聊天，还会作为这个直播房间的指令控制系统。那指令控制系统都包含哪些指令呢？比如主播要踢出某一个观众，或者要禁言某一个观众，观众给主播赠送了一个礼物等，都属于一条条的控制指令，其实真正的聊天内容也可以看作一个指令，就是聊天指令。实现手段也有很多种，其中最常用的就是WebSocket协议。本节将详细介绍如何利用WebSocket来实现指令控制（或者聊天系统）。

WebSocket API是下一代客户端-服务器的异步通信方法，是HTML5规范中替代AJAX的一种新技术。现在，很多网站为了实现推送技术，使用的技术都是轮询。轮询是在特定的时间间隔（如每1秒），由浏览器对服务器发出HTTP request，然后由服务器返回最新的数据给客户端的浏览器。这种传统的模式有很明显的缺点，即浏览器需要不断地向服务器发出请求。然而，HTTP request的header是非常长的，里面包含的数据可能只是一个很小的值，这样会占用很多的带宽和服务器资源。新的轮询技术是Comet，使用了AJAX。这种技术虽然可达到双向通信，但依然要发出请求，而在Comet中，普遍采用了长链接，这也会大量消耗服务器带宽和资源。面对这种状况，HTML5定义了WebSocket协议，能更好地节省服务器资源和带宽并能实时通信，且实现了浏览器与服务器全双工通信（full-duplex）。在WebSocket API中，浏览器和服务器只需要做一个握手的动作，浏览器和服务器之间就形成了一条快速通道。两者之间可直接进行数据互相传送。但是它不单单仅适用于Web端，在客户端使用起来也非常方便，一般使用在客户端会维护一个WebSocket对象，该对象可以调用发起连接、发送消息、关闭连接的方法，同时要为这个对象绑定以下四个事件。

·onOpen：连接建立时触发。

·onMessage：收到服务端消息时触发。

·onError：连接出错时触发。

·onClose：连接关闭时触发。

在这四个事件中，开发者可以执行自己的操作，下面分别介绍如何

在Android和iOS客户端使用WebSocket技术实现简单的聊天系统。

11.6.1 Android客户端的WebSocket实现

Android客户端比较常用的WebSocketClient有autobahn、AndroidAsync、Java-WebSocket，笔者使用的是autobahn，读者可以根据自己的使用场景来选择。现在来看如何使用autobahn实现一个聊天系统。

首先来看实例化WebSocket对象与发起连接的实现，代码如下：

```
WebSocket mConnection = new WebSocketConnection();
String wsuri  = "ws://echo.websocket.org";
mConnection.connect(wsuri, new WebSocketConnectionHandler() {
    @Override
    public void onOpen() {
    }
    @Override
    public void onClose(int code, String reason) {
    }
    @Override
    public void onTextMessage(String payload) {
    }
    @Override
    public void onBinaryMessage(byte[] payload) {
    }
    @Override
    public void onRawTextMessage(byte[] payload) {
    }
});
```

从以上代码可以看到，在对地址"ws：//echo.websocket.org"发起连接时，需要传递一个回调接口，当打开连接成功的时候会回调onOpen方法，用来提示用户已经连接成功，并且开发者也会开始发送Ping命令，以保持心跳连接；如果打开连接失败，回调onClose方法，开发者可以在这里尝试重试策略，或者提示用户连接失败；当收到消息的时候，如果是字符串类型的消息，则回调onTextMessage方法；如果是二进制类型的消息，则回调onBinaryMessage方法。

接下来发送Ping消息，直接调用WebSocket对象的sendPing方法，如果发送实际的消息，就调用sendTextMessage方法，而在最终关闭连接的时候调用disconnect方法就可。

11.6.2 iOS客户端的WebSocket实现

iOS客户端实现WebSocket使用最多的就是SocketRocket这个第三方库，大家可以在GitHub上下载这个库，当然也可以直接在代码仓库中拿到源码。把这个库的目录拖到项目中后，还要为项目添加一个libicucore.tbd库，然后引入SRWebSocket的头文件，代码如下：

```
#import "SRWebSocket.h"
```

之后让ViewController实现头文件中的SRWebSocketDelegate协议，这时需要重写以下方法：

```
- (void)webSocketDidOpen:(SRWebSocket *)webSocket {
}
- (void)webSocket:(SRWebSocket *)webSocket didFailWithError:(NSError *)error
}
- (void)webSocket:(SRWebSocket *)webSocket didReceiveMessage:(id)message {
}
- (void)webSocket:(SRWebSocket *)webSocket didCloseWithCode:(NSInteger)code
    reason:(NSString *)reason wasClean:(BOOL)wasClean {
}
```

协议中定义的这四个方法，分别对应前面介绍的四个事件，它们分别在连接成功的时候回调webSocketDidOpen方法，可以提示给用户连接成功以及取消掉Loading框，并同时启动一个定时器，定时给服务器发送ping的消息，以保证自己处理存活状态；连接失败的时候回调webSocket：didFaileWithError方法，开发者可以重连策略，如果不重连，应该提示给用户连接失败；接收到消息的时候回调webSocket：didReceiveMessage方法。由于可以传递二进制的消息，所以开发者可以判断参数中的message类型，再去做自己的处理；并且可在连接最终关闭的时候回调webSocket：didCloseWithCode：reason：wasClean方法，开发者可以根据关闭原因去尝试重连，并记录错误原因。

而真正的实例化WebSocket对象以及发起连接操作也很简单，代码如下：

```
NSString *url = "ws://echo.websocket.org";
SRWebSocket *webSocket = [[SRWebSocket alloc] initWithURLRequest:
    [NSURLRequest requestWithURL:[NSURL URLWithString:url]]];
webSocket.delegate = self;
[webSocket open];
```

如果要发送消息，则调用WebSocket的send方法，最终退出界面的时候可以调用close方法来关闭掉连接。

当然，要想真正运行一个聊天室，还需要有一个服务器的支持。可以使用WebSocket开源网站提供的服务器地址：ws：//echo.websocket.org。若想要自己搭建一个WebSocket服务器，可使用Java-WebSocket库写一个JavaSE的程序，用于将发布者的消息转发给所有订阅者。如果想要真正部署到WebSever上，可以使用Tomcat容器来运行一个JavaEE的Servlet，这个Servlet可以使用javax.websocket包中WebSocket相关的类来构建一个转发程序，从而将发布者的消息转发给所有订阅者。

11.7 本章小结

在实现了上述所有模块之后，最终构建的直播系统如图11-9所示。

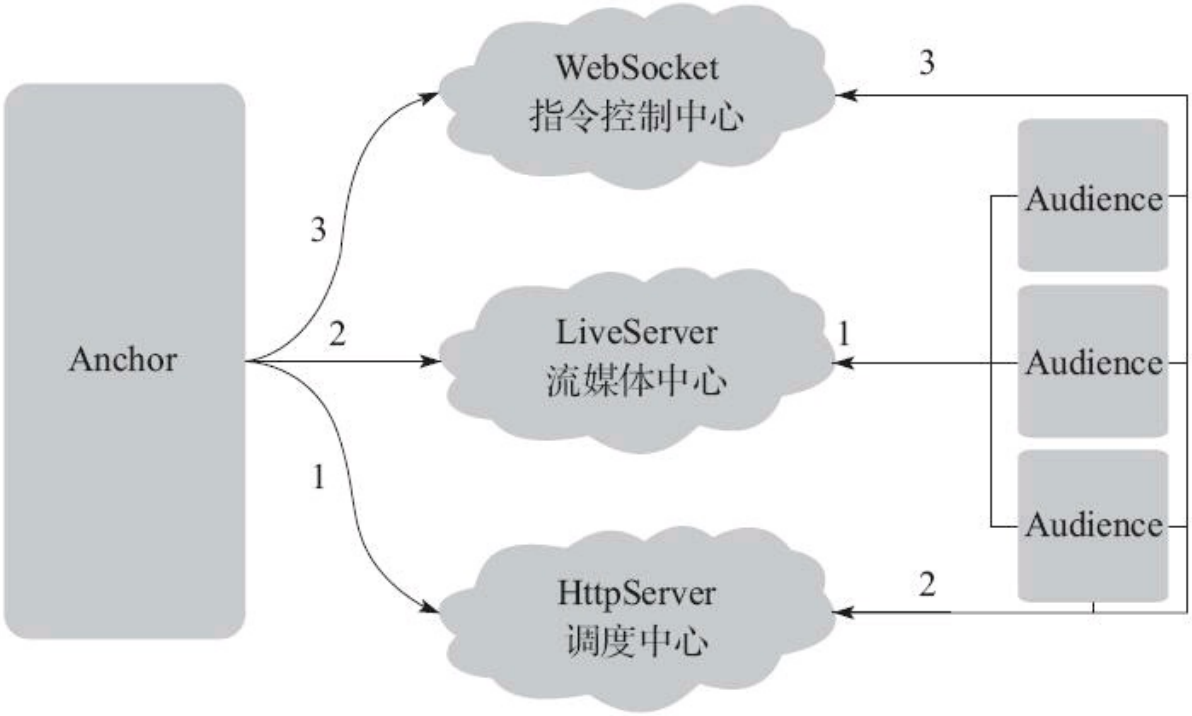


图 11-9

如图11-9所示，首先来看主播端（Anchor），主播要想开播，应先去请求调度中心的开播接口，调度中心会返回两个地址，一个是流媒体中心的地址，一个是指令控制中心的地址；然后主播会拿着流媒体中心地址去连接Live服务器，连接成功之后就可以推流了；接着拿着指令控制中心的地址去连接WebSocket服务器，连接成功之后就可以接受观众进入房间、聊天、礼物等指令，也可以发送禁言、踢人等指令。再来看看观众端（Audience），为了加快用户的首屏时间（从点击进入某个主播的房间开始，到看到主播视频的时间称为首屏时间），系统中所有展示房间的位置都会冗余这个主播的流媒体地址字段，所以不用请求HttpServer就可以直接观看这个主播的直播。但是，为了继续和主播发生聊天、送礼物等交互行为，并且验证自己的身份是否合法（可能被主播禁言或者踢掉），还要请求HttpServer拿到指令控制中心地址，然后去连接WebSocket服务器，如果连接成功就可以接收到指令以及发送对

应的指令。这样，通过这些模块的共同交互就构建出了一个可用的直播系统。但是，要想让这个直播系统表现得更好，还有一些细节需要处理，比如对弱网环境下主播端的处理，加快拉流端的首屏时间，两端利用统计数据来帮助我们完善整个系统的迭代更新等。

第12章　直播应用中的关键处理

第11章中已经将第5章的播放器项目改造成了一个可用的拉流端播放器，并将第10章中的视频录制器改造成了一个可用的推流工具。本章会基于第11章可用状态下的推流工具和拉流播放器进行改进，做一些关键处理，以达到一个企业级直播产品的要求。

12.1　直播应用的细节分析

本节会分两个部分来分析推流端和拉流端在实际生产过程中遇到的问题，然后会给出实现思路，而这些问题应该是直播产品都会遇到的，并且有可能是读者现在最头疼的问题。所以，下面逐一分析并提出解决问题的方法。

12.1.1　推流端细节分析

1.自适应码率

网络环境不稳定一直困扰着部分主播，特别是在直播场景中，这其中既包括一些小运营商的上行带宽分配不合理、域名解析错误等因素导致的问题，也包括网络抖动、暂时的网络链路拥塞等问题，这使得观众端无法流畅地观看主播的表演，也让直播无法继续。基于用户在直播中的这个痛点，我们采用自动升降码率技术来解决这个问题。解决方案就是：实时根据主播所处的网络环境，调整主播的视频码率，以达到观众端可以流畅观看主播以及与主播互动的效果。

2.数据统计

对于主播端的数据统计是非常重要的，因为它能反映出平台内主播的网络状况，直播时长等行为，也是所有直播产品必须做的事情。统计的数据包括开始直播时间、连接CDN推流节点服务器的连接时长、推流时长、丢帧率、推流的平均码率、自动升降码率的变动表等。通过这些统计数据，产品部门和运营部门可以针对性地举办一些活动，引导主播增加直播时间，也有助于我们推动CDN厂商给主播更优的链路节点，还可以优化自动升降码率策略。

12.1.2　拉流端细节分析

1.重试机制

由于拉流播放器的媒体资源在网络上，所以有可能会出现建立连接失败的情况，或者寻找流信息失败的情况。基于此，要建立重试机制，若拉流播放器在上述过程中失败，就可以重新连接或重新寻找流信息。

2.秒开视频

在拉流播放器中，最重要的体验就是观众点击了一个主播的房间之后，可以在最短的时间内听到主播的声音并看到主播的画面，这在直播领域称为秒开视频。这也是评判一个直播App体验最重要的一点，我们要做的就是尽量缩短首屏时间。实现思路就是，当用户点击行为发生之后，就直接启动播放器，然后再跳转到直播房间页面，在页面跳转过程中，拉流播放器就已经将一段音频和视频解码成功，当真正进入这个主播页面时就已经加载了这个直播流。虽然这是客户端能做到的，但是需要CDN厂商的配合，包括视频关键帧的缓存、CDN节点的部署、网络链路的优化等。

3.数据统计

对于拉流播放器来说，数据统计也是非常重要的，其中包括开始打开流的时间、打开流花费的时间、打开流失败花费的时间、打开流失败的类型、重试次数、首屏时间、拉流时长、卡顿次数等。可通过这些统计参数来具体优化拉流播放器的流程，比如通过卡顿次数来优化缓存大小，通过打开流花费的时间来推动CDN厂商去做优化等。统计数据是最基本的，只有这些统计数据的存在，我们才有策略支撑的依据。

上述需求其实是一个成熟的直播产品都应该具有的功能，现在带着这些问题与实现思路继续学习后面的章节吧。

12.2　推流端的关键处理

本节将针对推流端来解决12.1节提出的问题，主要还是针对复杂的网络环境来优化我们的推流策略，以便让整个直播可以流畅进行，同时也要将整个直播过程中主播的直播状态以统计数据的形式上报给服务器，从而方便后续的迭代改进。所以本节分为两部分：第一部分是将自适应码率策略集成入我们的系统，第二部分是采集具体直播过程中的数据，并分析这些数据将来可能产生的意义。请读者带着以上两个问题开始本节的学习。

12.2.1　自适应码率的实践

自适应码率策略解决的问题或者适用的场景已经在12.1.1节中介绍了，如果对这个策略所要解决的问题还不是很清楚，建议再看看相应的分析。下面来看实现思路：在推流过程中，我们可以根据当时主播的网络情况，测算出网络出口带宽是多少，进而和编码器产生的数据量进行比较：如果网络出口带宽大于编码器产生的数据量，则可以提高视频质量（增大编码器的码率设置，同时增大视频的fps以达到更加流畅的效果），可以让主播发布出更加优质的视频；如果网络出口带宽和编码器产生的数据量相近，那么视频质量可不做任何变化；如果网络出口带宽小于编码器产生的数据量，那就要降低视频质量（降低编码器的码率设置，同时减小视频的fps以免视频质量太差）以使主播可以流畅地进行直播。由于整个流的码率中视频轨码率达90%以上（视频轨码率默认设置为600Kbps，音频轨码率默认设置为64Kbps），所以只需处理视频轨的设置就能满足大部分场景，整体实现结构如图12-1所示。

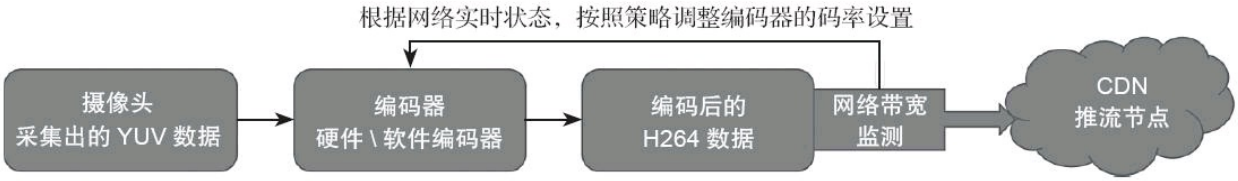


图　12-1

图12-1中所示的流程为，摄像头采集出的YUV数据经由编码器进行编码，成为H264数据，然后Muxer模块将H264数据进行封装，并发送到流媒体服务器上（这里的实现是使用CDN的推流节点），在发送之后由网络带宽监测模块来监测实际的网络上行带宽，然后经过一定的策略调整反馈给编码器，再进行码率设置。因此，整个系统中有三个核心模块：第一个是网络带宽监测模块；第二个是码率调整策略；第三个是实时改变编码器的码率。

1.网络带宽监测模块

这个模块不仅要能够监测网络上行带宽，也要能够监测编码器编码出来的码率是多少，这样才可以进行对比，从而确定当前网络上行带宽是否足以将编码器编码出来的视频帧发布到CDN节点服务器上。为了方

便下文中的讨论，首先统一一下术语，网络上行带宽称为发送码率，编码器编码出来的码率称为压缩码率。而在监测的过程中，仅看某一时刻的两个码率不足以反映问题，应该要监测一个窗口时间段的平均码率，而这个窗口的时间可以根据自己的场景进行配置（3～10s都可以）。首先定义一个窗口码率的结构体，代码如下：

```
typedef struct WindowBitRate {
    int startTimeInSecs;
    int endTimeInSecs;
    int bitRate;
    WindowBitRate() {
        this->startTimeInSecs = 0;
        this->endTimeInSecs = 0;
        this->bitRate = 0;
    }
    void update(int startTimeInSecs, int endTimeInSecs, int bitRate) {
        this->startTimeInSecs = startTimeInSecs;
        this->endTimeInSecs = endTimeInSecs;
        this->bitRate = bitRate;
    }
    void clone(WindowBitRate *windowBitrate) {
        this->startTimeInSecs = windowBitrate->startTimeInSecs;
        this->endTimeInSecs = windowBitrate->endTimeInSecs;
        this->bitRate = windowBitrate->bitRate;
    }
} WindowBitRate;
```

这个窗口的结构体将开始时间、结束时间以及平均码率都封装了起来，而BitRate-Monitor中为了记录发送码率和压缩码率，并且为了让这两种类型的码率可以对应上，声明了四个变量，分别代表上一个窗口的压缩码率和发送码率以及当前窗口的压缩码率与发送码率，代码如下：

```
WindowBitRate *sendLastWindowAVGBitrate;
WindowBitRate *sendCurWindowAVGBitrate;
WindowBitRate *compressLastWindowAVGBitrate;
WindowBitRate *compressCurWindowAVGBitrate;
```

统计过程结构如图12-2所示。

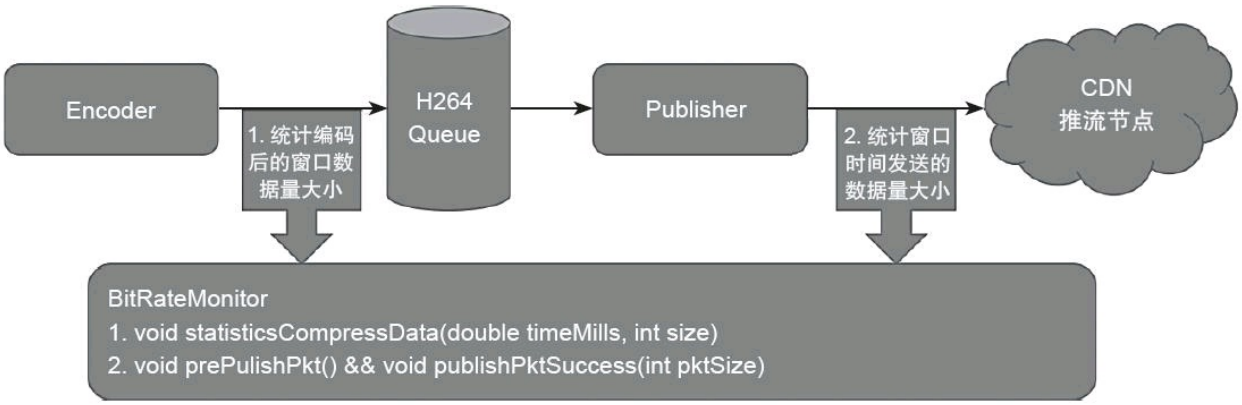


图　12-2

在图12-2中，统计码率的模块用BitRateMonitor类来表示，待编码器编码出一帧之后，可以将这一帧的时间戳和这一帧的大小来调用statisticsCompressData方法，这个方法会将这一帧放到合适的窗口中，如果满足这个窗口的统计，则会计算出这个窗口的平均码率是多少，每当积攒出一个压缩码率的窗口，就克隆到compressAVGBitRate的Last中，并且更新当前的开始时间、结束时间以及平均码率。

Publisher模块也一样，在发送一个AVPacket结构体之前，要调用这个类的prePublishPkt方法，待发送结束之后，再将这个AVPacket的大小调用publishPktSuccess方法，这个方法把发送出去的大小都累积起来，等到积攒够了一个窗口之后就克隆到sendAVGBitRate的Last中，并且更新当前的开始时间、结束时间以及平均码率。注意Publisher中的时间戳是发送的相对时间戳，用于衡量当前一个窗口时间内发送出去了多大容量的数据，以此来记录当前网络上行带宽是多少。

2.码率调整策略

上面统计出来的窗口码率、压缩窗口码率和发送窗口码率的时间不一定是对应的。也就是说，有可能会出现压缩窗口码率记录的是时间范围为[105，110]秒内的压缩码率，发送窗口码率记录的却是[100，105]秒内的发送码率，这样就没有参考的依据了。所以，上面为每一种类型的窗口码率都做了一个Last和一个Current这两个窗口码率。在比较的时候，应该先选取正确的对应顺序，然后再取出码率，这才是期待的发送码率和压缩码率，代码如下：

```
int compressAVGBitrate = compressCurWindowAVGBitrate->bitRate;
```

```
int sendAVGBitrate = sendCurWindowAVGBitrate->bitRate;
int curCurDiff = abs(compressCurWindowAVGBitrate->startTimeInSecs -
        sendCurWindowAVGBitrate->startTimeInSecs);
int curLastDiff = abs(compressCurWindowAVGBitrate->startTimeInSecs -
        sendLastWindowAVGBitrate->startTimeInSecs);
int lastCurDiff = abs(compressLastWindowAVGBitrate->startTimeInSecs -
        sendCurWindowAVGBitrate->startTimeInSecs);
int lastLastDiff = abs(compressLastWindowAVGBitrate->startTimeInSecs -
        sendLastWindowAVGBitrate->startTimeInSecs);
if (curCurDiff <= curLastDiff && curCurDiff <= lastCurDiff
        && curCurDiff <= lastLastDiff) {
    compressAVGBitrate = compressCurWindowAVGBitrate->bitRate;
    sendAVGBitrate = sendCurWindowAVGBitrate->bitRate;
} else if (curLastDiff < curCurDiff && curLastDiff <= lastCurDiff &&
        curLastDiff <= lastLastDiff) {
    compressAVGBitrate = compressCurWindowAVGBitrate->bitRate;
    sendAVGBitrate = sendLastWindowAVGBitrate->bitRate;
} else if (lastCurDiff < curCurDiff && lastCurDiff <= curLastDiff &&
        lastCurDiff <= lastLastDiff) {
    compressAVGBitrate = compressLastWindowAVGBitrate->bitRate;
    sendAVGBitrate = sendCurWindowAVGBitrate->bitRate;
} else if (lastLastDiff < curCurDiff && lastLastDiff <= curLastDiff &&
        lastLastDiff <= lastCurDiff) {
    compressAVGBitrate = compressLastWindowAVGBitrate->bitRate;
    sendAVGBitrate = sendLastWindowAVGBitrate->bitRate;
}
```

这样取出来的compressAVGBitrate和sendAVGBitrate就代表了同一个时间段内的压缩码率和发送码率。我们的目的就是按照发送码率反馈给编码器来调整压缩码率，而得到的这两个值只是一个时间窗口内的值，有可能这段时间内发生了网络抖动，不能仅使用这个值去盲目地设置编码器的码率。比较稳妥的策略是设置一个周期，比如5个窗口的时间长度，然后使用几个窗口的发送平均码率去设置直到编码器改变码率。但是这种策略只能作为降码率的条件，因为这种统计方式统计出来的发送码率不可能会大于压缩码率，发送模块发送的数据量不可能多于编码器编码出来的H264数据的数据量。因此，如果想要做到升码率，就需要在策略中加上当前H264队列大小的变化趋势，比如队列大小一直在0和1之间进行变化，则代表编码器编码出来一个视频帧，Publisher模块就立马将它发送出去，这时我们就可以尝试去上升码率；如果队列的变化趋势在队列大小的10%～70%范围内徘徊，则代表有可能网络发生抖动，用户端可能会出现卡顿，延迟在慢慢增大，可以暂时不做任何处理；如果队列趋势一直在70%以上，说明Publisher模块不足以将编码器编码出来的视频帧发送出去，这时就应该将前面得出的这个周期的平均发送码率作用到编码器。在大部分场景中，升码率一般都会比较保守，可以称之为“慢慢升”，降码率相对于升码率来讲可以快速，可以称

为“快速降”，这两种策略结合称为自适应码率策略，这个策略可增加整个直播过程的流畅度。

3.实时改变编码器的码率

系统中使用的编码器可以有三种，即iOS平台使用VideoToolbox来编码视频，Android平台使用MediaCodec或FFmpeg（使用libx264）来编码视频。

（1）VideoToolbox动态码率设置

使用VideoToolbox硬件编码器API，对系统的要求必须是iOS 8.0以上。对于Video-Toolbox的动态码率设置比较简单，直接对编码器会话设置码率与fps就可，代码如下：

```
- (void) settingMaxBitRate:(int)maxBitRate avgBitRate:(int)avgBitRate
        fps:(int)fps; {
    VTSessionSetProperty(EncodingSession,
        kVTCompressionPropertyKey_MaxKeyFrameInterval,
        (__bridge CFTypeRef)(@(fps)));
    VTSessionSetProperty(EncodingSession,
        kVTCompressionPropertyKey_ExpectedFrameRate,
        (__bridge CFTypeRef)(@(fps)));
    VTSessionSetProperty(EncodingSession,
        kVTCompressionPropertyKey_DataRateLimits,
        (__bridge CFArrayRef)@[@(maxBitRate / 8), @1.0]);
    VTSessionSetProperty(EncodingSession,
        kVTCompressionPropertyKey_AverageBitRate,
        (__bridge CFTypeRef)@(avgBitRate));
}
```

（2）MediaCodec动态码率设置

使用MediaCodec硬件编码器API，对于系统的要求必须是Android 4.3以上。在Android系统4.4以上提供了使用Bundle形式动态配置编码器内部参数的API，代码如下：

```
Bundle bitRateBundle = new Bundle();
bitRateBundle.putInt(MediaCodec.PARAMETER_KEY_VIDEO_BITRATE, targetBitRate);
mEncoder.setParameters(bitRateBundle)
```

这种方法在笔者测试的过程中，效果非常差，很多设备只要一改变码率，视频质量就会立即下降，所以不建议大家使用。笔者给出的建议是这样的，如果设备是5.0系统以上，系统为MediaCodec提供了一个reset方法，调用这个方法之后就可以去重新配置各个参数，最后把重新创建出来的Surface类型的MediaCodec的InputSurface传到底层，创建出EGLDisplay，然后进行编码操作，代码如下：

```
public void reConfig(int width, int height, int bitRate, int frameRate)
                throws Exception {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        try {
            if (mEncoder != null) {
                mEncoder.reset();
            }
            MediaFormat format = MediaFormat.createVideoFormat(MIME_TYPE,
                width, height);
            format.setInteger(MediaFormat.KEY_COLOR_FORMAT,
                MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface);
            format.setInteger(MediaFormat.KEY_BIT_RATE,
                bitRate - MEDIA_CODEC_NOSIE_DELTA);
            format.setInteger(MediaFormat.KEY_FRAME_RATE, frameRate);
            format.setInteger(MediaFormat.KEY_CAPTURE_RATE, frameRate);
            format.setInteger(MediaFormat.KEY_I_FRAME_INTERVAL, IFRAME_INTER\
            mEncoder.configure(format, null, null,
                MediaCodec.CONFIGURE_FLAG_ENCODE);
            mInputSurface = mEncoder.createInputSurface();
            mEncoder.start();
        } catch (Exception e) {
            throw e;
        }
    } else {
        throw new UnMatchedException("系统版本不匹配");
    }
}
```

上述代码比较简单，首先要判断设备的系统Android的版本代号在Lollipop（5.0系统）之上，然后调用reset方法，之后重新配置编码器，一定要重新获取这个编码器的InputSurface，最终调用start方法。如果设备的系统版本不在5.0以上，那么就先销毁整个编码器，然后以重新建立编码器的方式来达到动态设置码率与fps的需求。相比较而言，第二种方法的开销会比较大，但是对于版本在5.0系统以下的设备来说，这也是唯一的一种方法。

（3）FFmpeg动态码率设置

虽然使用了FFmpeg来进行软件编码，但是其内部还是使用libx264来实现的。所以设置动态码率时，最终还是会调用到libx264库中的动态改变码率，通过查看FFmpeg的源码文件libx264.c可以知道，要想调用到libx264库的reconfig方法，需要改变rc_buffer_size变量，代码展示如下：

```
if (x4->params.rc.i_vbv_buffer_size != ctx->rc_buffer_size / 1000 ||
    x4->params.rc.i_vbv_max_bitrate != ctx->rc_max_rate    / 1000) {
    x4->params.rc.i_vbv_buffer_size = ctx->rc_buffer_size / 1000;
    x4->params.rc.i_vbv_max_bitrate = ctx->rc_max_rate    / 1000;
    x264_encoder_reconfig(x4->enc, &x4->params);
}
```

这对FFmpeg的版本是有一定要求的，在FFmpeg的2.1版本中是不支持libx264的动态改变码率的，而上述这段代码摘自FFmpeg的2.8.5版本的libx264.c源码文件中。所以客户端的代码如下：

```
void VideoX264Encoder::reConfigure(int bitRate) {
    pCodecCtx->rc_max_rate = bitRate;
    pCodecCtx->rc_min_rate = bitRate;
    pCodecCtx->rc_buffer_size = bitRate * 3;
    pCodecCtx->bit_rate = bitRate;
}
```

如果调用上述代码，就可以达到使用软件编码器动态改变码率的需求。

至此，分析完了最后一个模块。再回顾整个流程，首先使用码率监测模块，将编码器编码出来的视频平均码率和Publisher发送的平均码率统计出来；然后根据这两个码率再加上这一段时间内的队列变化趋势得出一个合适的编码器应该编码的码率（有可能比当前编码器的码率设置要高一些，也有可能低一些）；最后将这个码率设置给编码器。如何让编码器在运行过程中动态设置码率，我们也给出了详尽的解释。按照以上思路，读者可以去完成自己产品中的自适应码率策略，也可以参考代码仓库中的源码，以便于深入理解。

12.2.2 统计数据保证后续的应对策略

本节整理推流端应该统计哪些数据，以便给开发部门和产品部门提供后续迭代的方向，给运营部门提供数据的支持。在真正开始统计推流端业务场景下的数据之前，首先要把主播端的IP地址、CDN推流节点服务器的地址以及主播的ID作为统计数据的基本信息。在12.1节中总结了以下要统计的数据，其中包括开始直播时间、连接CDN推流节点服务器的连接时长、推流时长与推流平均码率、丢帧率、自动升降码率的变动表等。下面介绍如何统计这些参数，以及这些参数有什么作用。首先，定义PublisherStatistics类将这些参数以及这些参数的操作封装起来，代码如下：

```
class PublisherStatistics {
private:
    long long startTimeMills;
    int connectTimeMills;
    int publishDurationInSec;
    int       totalPushVideoFrameCnt;
    int       discardVideoFrameCnt;
    float       publishAVGBitRate;
public:
    PublisherStatistics();
    ~PublisherStatistics();
    void connectSuccess();
    void discardVideoFrame(int discardVideoPacketSize);
    void pushVideoFrame();
    void stopPublish();
    char* getAdaptiveBitrateChart();
    long long getStartTimeMills(){
        return startTimeMills;
    };
    int getConnectTimeMills(){
        return connectTimeMills;
    };
    int getPublishDurationInSec(){
        return publishDurationInSec;
    };
    float getDiscardFrameRatio() {
        if(totalPushVideoFrameCnt > 0){
            return (float) discardVideoFrameCnt / (float) totalPushVideoFrame
        } else {
            return 0;
        }
    };
    float getPublishAVGBitRate(){
       return publishAVGBitRate;
    };
```

```
    float getExpectedBitRate(){
        return expectedBitRate;
    };
};
```

这个类的方法比较简单，属于对属性的操作。至于每个参数的意义以及如何赋值，下面会依次给出解释。

1.开始直播时间

当主播点击开播的那个时刻，我们就记录从1970年1月1日起以毫秒为单位的绝对时间戳。记录这个时间戳的含义在于，可以在后台画出主播开播时间点的分布曲线，针对主播的开播时间点，运营和客服可以更合理地分配时间去响应主播的一些需求，开发人员也可以更大力度地在主播直播分布最多的时间段内测试CDN厂商的链路分配以及响应速度。要说明的是，在初始化这个类的构造函数中就对变量startTimeMills进行赋值了。

2.连接服务器时长

由于协议层（Protocol Layer）使用的是FFmpeg的libavformat模块，所以统计连接时长就在调用RecordingPublisher类的init方法执行结束之后调用connectSuccess方法即可。如果想统计得更加精确，就在init方法调用FFmpeg的avio_open2方法之前初始化PublisherStatistics类，之后调用connectSuccess方法来统计以毫秒为单位的时间差，并以此作为连接时长。这个连接时长包括主播端的网络进行DNS解析的时间和拿到IP地址之后连接CDN厂商的推流节点服务器的时间，有助于开发人员分析当前主播的网络和CDN厂商分配的链路节点是否合理。

3.推流时长与推流平均码率

推流时长代表主播本场直播的时间，我们可以在整个Muxer（Publisher）模块的stop方法中调用stopPublish方法，这样就可以取出当前时间戳，并减去开始推流时间戳，从而得到推流时长。当然，在stopPublish方法中也可以计算出推流的平均码率信息。而推流时长和平均码率的信息也是非常有意义的，能从一定程度上说明这个主播的网络情况。针对平均码率，我们可以建议主播更换网络（如果是小运营商的话），或者找CDN厂商分配更合理的推流节点；针对推流时长，运营人员可以做一些活动来刺激优质主播，以提升推流时长，从而增加社区

的内容，增加社区的活跃。

4.丢帧率

丢弃的视频帧的数目占整场直播总视频帧数目的比例，每次有一帧视频帧被编码出来之后就调用pushVideoFrame方法，里面的totalFrameCnt则加一，待丢弃一个GOP（也有可能是GOP的后半部分）之后，就以丢掉的帧数目作为参数调用discardVideoFrame方法，这个方法内部会将discardFameCnt加上这个丢帧的数目，最终使用discardFrameCnt除以totalFrameCnt计算出丢帧率。丢帧率也是反应主播网络与链路选择以及CDN厂商推流节点的一个指标，拿这个指标去和CDN厂商交涉，以减小丢帧率，达到流畅播放的目的。当然，这和自动升降码率系统也有关系，这个指标也有助于开发人员优化升降码率系统的策略，以降低丢帧率这一指标。

5.自动升降码率变动表

由于我们为系统加入了自适应码率模块，为了能对这个自适应码率模块有一个评估，便于后续开发人员继续迭代优化这个模块，因此需要把作用到编码器的码率变化情况统计下来，上报给服务器。在后台可以使用图表展示编码器编码的平均码率与Publisher发送的平均码率的变化趋势，便于开发人员从时间调整周期、码率调整范围等方面优化自适应码率模块。

至此，分析完了所有的统计参数，读者可以依据自己的产品场景加入与自己业务相关的统计参数，以增强系统的流畅性与稳定性。

12.3 拉流端的关键处理

上一节中针对推流端在实际生产过程中遇到的问题进行了修复，并且为了给后续的迭代以及运营活动提供数据支持，增加了推流端的数据统计。本节针对拉流端的实际问题提出具体的解决方案，并且也会增加数据的统计，请读者带着12.1节中对于拉流端提出的问题开始本节的学习吧。

12.3.1 重试机制的实践

由于拉流播放器的媒体资源在网络上，因此有可能会出现在调用find_stream_info函数之后，这个视频中的Stream还是会解析不出正确的格式。命令行工具的ffmpeg或者ffplay去下载或者播放网络流的时候，如果不能解析出正确的格式，展示的错误提示如下：

```
Could not find codec parameters for stream 0 : reason xxx
Consider increasing the value for the 'analyzeduration' and 'probesize'
    options.
```

这句错误提示的含义是说，不能够找出第一（二）个流，原因可能有很多种；它提示我们应该增加analyzeduration和probesize选项的值。在命令行工具中可以使用以下命令行来查看这几个参数代表的意义：

```
ffmpeg -h full | grep probe
```

在终端键入上述这行命令之后，则可以看到如下关键信息：

```
-probesize : set probing size (from 32 to INT_MAX) (default 5e+06)
-analyzeduration : specify how many microseconds are analyzed to probe the
    input (from 0 to INT_MAX) (default 5e+06)
-fpsprobesize : number of frames used to probe fps (from -1 to 2.14748e+09)
    (default -1)
```

以上代码中有三个主要参数可以留给我们设置，且这三个参数共同用来控制FFmpeg的libavformat模块中的av_find_stream_info方法的执行时长。第一个参数是probesize，它从连接通道（当前场景就是网络流）中读出多少个字节层面来控制函数是否结束，即如果从通道中读出的AVPacket字节量大于设置的probesize这个值，就结束find_stream_info这个函数；第二个参数是analyzeduration，它从解码出来的帧的时间长度层面来控制函数是否结束，即如果从通道中读出来的AVPacket解码之后的时间长度大于analyzeduration，则结束这个函数，单位是微秒；第三个参数是fpsprobesize，它从解码出来的帧数目角度来控制函数是否结束，默认值是-1。如果开发人员不进行设置，函数内部会赋值默认值为

20。所以在命令行工具（无论是ffmpeg还是ffplay）中都可以设置这三个参数来控制寻找流信息的时间，而在开发人员编写代码的过程中，同样也可以进行设置，代码如下：

```
AVFormatContext *pFormatCtx;
pFormatCtx->max_analyze_duration = 20 * 1024;
pFormatCtx->probesize = 2048;
pFormatCtx->fps_probe_size = 3;
```

一般可以拿上述参数去初始化自己的AVFormatContext，这样既可以保证能解析出流，还能让时间消耗得比较少。但是，如果碰到一些比较高码率的流或者这些参数设置得过小，就会导致解析不到正确的流信息。那如何来判断是否正确地解析到流信息了呢？笔者在FFmpeg的libavformat模块的utils.c文件中找到了一份代码，可判断流信息是否解析成功，它的实现就是要遍历出这个Container中的音频流和视频流。首先来看音频流，代码如下：

```
AVCodecContext *audioCodecCtx = audioStream->codec;
if (!audioCodecCtx->frame_size && determinable_frame_size(audioCodecCtx))
    FAIL("unspecified frame size");
if (!audioCodecCtx->sample_rate)
    FAIL("unspecified sample rate");
if (!audioCodecCtx->channels)
    FAIL("unspecified number of channels");
if (audioCodecCtx->sample_fmt == AV_SAMPLE_FMT_NONE)
    FAIL("unspecified sample format");
if (audioCodecCtx->codec_id == AV_CODEC_ID_DTS)
    FAIL("no decodable DTS frames");
if (audioCodecCtx->codec_id == AV_CODEC_ID_NONE)
    FAIL("unknown codec");
```

上述代码分别对音频编码器上下文中的各个信息给出了判断，如果没有解析成功，则会调用FAIL方法。其中FAIL是定义的一个宏，宏定义如下：

```
#define FAIL(errmsg) do {       \
    printf("%s", errmsg);        \
    return 0;                       \
} while (0)
```

这个宏中会输出这条错误信息，并且返回0，代表解析流信息失

败。接着来看视频流信息的判定，代码如下：

```
if (!avctx->width)
    FAIL("unspecified size");
if (avctx->pix_fmt == AV_PIX_FMT_NONE)
    FAIL("unspecified pixel format");
if (st->codec->codec_id == AV_CODEC_ID_RV30 ||
        st->codec->codec_id == AV_CODEC_ID_RV40)
    if (!st->sample_aspect_ratio.num && !st->codec->sample_aspect_ratio.num
        && !st->codec_info_nb_frames)
    FAIL("no frame in rv30/40 and no sar");

if (audioCodecCtx->codec_id == AV_CODEC_ID_NONE)
    FAIL("unknown codec");
```

当openInput函数返回0的时候，代表可能由于这三个参数的值设置过小而导致解析流信息出现了错误，所以需要重试。过程如下：首先关闭连接通道，并释放AVFormatContext，然后重新执行openInput函数。在重试的过程中，可以增加probesize的值，然后执行find_stream_info函数。我们可以对重试的次数做一个限制，比如，若重试3次还是无法找到正确的流信息，那么就提示客户端寻找流信息失败。读者可以参考代码仓库中的源码，然后应用到自己的实际产品中。

12.3.2 首屏时间的保证

在一个直播App中，观众端最直观的体验就是首屏时间的时长。首屏时间指的是从观众点击一个房间开始，到看到这个房间内主播的画面以及听到主播声音的时间。首屏时间越短，产品越流畅，体验也越好。因此，如何缩短首屏时间成为一个直播App必须持续优化的问题，而这也是笔者在本节要和大家讨论的问题。

首先来看拉流播放器的整体架构图，如图12-3所示。

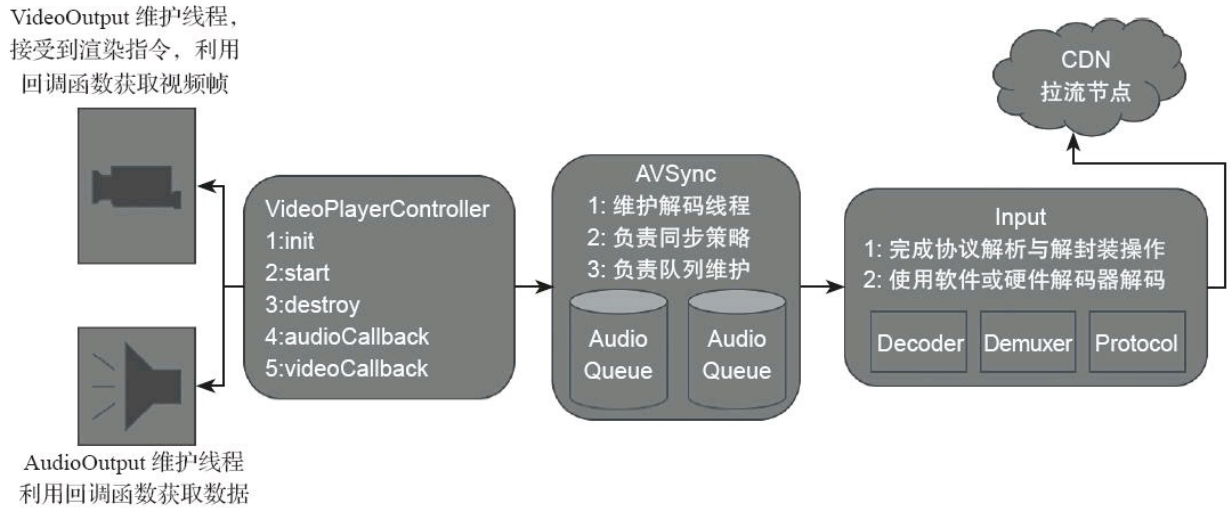


图 12-3

在图12-3中，最左边是VideoOutput和AudioOutput模块，其内部由自己维护线程来渲染视频画面与音频数据，整个播放流程是由AudioOutput来驱动的，当AudioOutput播放一帧音频帧时，就会向VideoPlayerController中audioCallback方法发出请求，让它来填充音频数据。待填充好音频数据之后，就发送一个指令给VideoOutput模块，让它来渲染视频画面；当VideoOutput接收到这个指令之后，会向VideoPlayerController中的videoCallback方法发出请求，拿一帧与当前播放的音频帧对齐的画面，然后进行渲染。最右侧是Input模块，负责协议解析，解封装，最终使用软件或者硬件解码器将音视频流解析为原始的格式，而它的客户端代码就是AVSync模块。AVSync模块是为了做音视频同步的，但是，首先它得维护一个线程和音视频队列，并负责调用Input模块将视频流最终解码成为音视频的原始数据，然后放入两个队列

中，以便向外提供获取音频数据的方法和获取视频帧的方法，其中音视频同步策略放在获取视频帧的方法中；而VideoPlayerController就是核心控制器类，会控制AudioOutput模块、VideoOutput模块、AVSync模块等所有组件的生命周期。这就是整个播放器的流程，下面就对照这个架构图来分析如何在整个流程中加快首屏时间。

首先，要尽快让Input模块完成find_stream_info过程，否则永远不会进入后续的解码过程甚至后续的渲染过程，也就不可能让用户听到声音和看到画面了。所以要对FFmpeg的AVFormatContext设置如下参数：

```
AVFormatContext *pFormatCtx;
pFormatCtx->max_analyze_duration = 20 * 1024;
pFormatCtx->probesize = 2048;
pFormatCtx->fps_probe_size = 3;
```

当然，设置这些参数加快了find_stream_info方法的执行时间，但是有可能（概率比较低）导致解析出来的流没有正确的信息，但12.3.1节中已经给出了重试机制，可以保证这种异常情况正确解决。这个设置也是直播播放器与录播播放器不同的地方，由于录播视频进行播放时肯定从第一帧视频帧展示给用户，而第一帧又是关键帧，所以可以很快解码出视频帧；而对于直播的视频，拉流播放器不一定在哪一帧连接上来，所以需要客户端和CDN厂商做优化，CDN厂商会缓存关键帧，尽量快地把关键帧给到拉流播放器。

然后要尽快启动播放器，当观众端点击某个主播的房间时就调用播放器的init方法，而页面的跳转也需要100ms到500ms的时间（实际场景中可能会是一个动画）。当跳转页面完成的时候，播放器早已完成了初始化和解码前几帧视频帧的工作，直接可以进行渲染了。但是在这种场景下，需要播放器具备一种特性，即没有页面也可以播放视频中的音频，如果播放器现在不支持，则需要改进成这种结构，因为这种结构对于后续的扩展（比如小窗播放器等功能）是非常有益的。由于目前VideoOutput组件的初始化被耦合到VideoPlayerController的Init方法中，所以要将VideoOutput组件的初始化与整个播放器的初始化分开。

1.Android平台播放器的预加载

在Android平台显示部分使用的是SurfaceView，依靠SurfaceView设置的Callback中的生命周期方法，将Surface传递到Native层构造出

ANativeWindow，然后构建成为EGLDisplay，最终由OpenGL ES渲染上去。之前的播放器初始化也是由SurfaceView设置的Callback中的生命周期方法onSurfaceCreate来触发的，而此次要改造播放器的最终结构，并脱离SurfaceView的生命周期控制。

首先在VideoPlayerController公布init接口方法：

```
bool init(char *URL, JavaVM *g_jvm, jobject obj);
```

这个方法的实现开辟了一个新线程，并在这个线程中实例化AVSync后调用初始化方法来打开流，如果成功，则实例化AudioOutput并调用start方法让声音开始播放，然后回调客户端代码表示已经成功打开了流。

声音播放自然会回调设置给AudioOutput的回调函数来填充音频数据，而这个填充音频数据的方法就会调用AVSync模块来填充音频数据，同时判断如果videoOutput还没有被设置的话，则不调用videoOutput的signalFrameAvailable方法，这时虽然还完全没有SurfaceView的参与，但播放器已经可以播放视频中的音频部分，代码如下：

```
int VideoPlayerController::fillData(byte* outData, size_t bufferSize) {
    int ret = bufferSize;
    if(this->isPlaying && synchronizer) {
        ret = synchronizer->fillAudioData(outData, bufferSize);
        if (NULL != videoOutput){
            videoOutput->signalFrameAvailable();
        }
    } else {
        memset(outData, 0, bufferSize);
    }
    return ret;
}
```

当客户端发生了页面跳转，并且SurfaceView已经显示出来的时候，Callback的生命周期方法onSurfaceCreate就会被触发，此时客户端调用VideoPlayerController中的onSurface-Create方法，方法实现如下：

```
void VideoPlayerController::onSurfaceCreated(ANativeWindow* window,
            int width, int height) {
    if (!videoOutput) {
```

```
        videoOutput = new VideoOutput();
        videoOutput->initOutput(window, screenWidth, screenHeight,
            videoCallbackGetTex, this);
    }else{
        videoOutput->onSurfaceCreated(window, screenWidth, screenHeight);
    }
}
```

如果是第一次进入，则会构建出VideoOutput类型的实例。VideoOutput也会分为两部分：第一部分是构造VideoOutput的OpenGL ES上下文与渲染线程，当然创建OpenGL上下文的时候一定要与Input模块的Uploader共享OpenGL ES上下文；第二部分是根据ANative-Window构造出要渲染的EGLDisplay类型的目标。在VideoOutput中，onSurfaceCreated方法就是用来创建渲染目标EGLDisplay的。

如果Activity跳转到了别的子页面（比如充值页面），SurfaceView也自然会消失，这时Callback的生命周期方法onSurfaceDestroyed会被触发，客户端应该调用VideoPlayer-Controller中的onSurfaceDestroyed方法，方法实现如下：

```
void VideoPlayerController::onSurfaceDestroyed() {
    if (videoOutput) {
        videoOutput->onSurfaceDestroyed();
    }
}
```

这里VideoOutput公布的onSurfaceDestroyed方法是销毁在onSurfaceCreated方法中构造的Renderer、EGLDisplay和根据SurfaceView中的Surface构建的ANativeWindow。注意这个方法并不会销毁OpenGL ES上下文与渲染线程，而VideoOutput提供的stop方法才是彻底销毁VideoOutput这个实例，也只有在这个播放器完全销毁的时候，才会调用VideoOutput的stop方法。

最终我们将播放器改造成了一个预加载的播放器，同时也脱离了显示界面的控制，而由自己的生命周期方法来控制播放器状态，使得客户端调用更加方便，同时也为后续小窗口播放器或者后台播放器提供了基础架构。

2.iOS平台播放器的预加载

在iOS平台显示部分是自定义一个UIView，在VideoPlayerViewController中可实例化这个UIView，并且将这个UIView的实例以subView的形式加到这个ViewController中。一般情况下，都是使用VideoPlayerViewController中的初始化方法来实例化AVSync并且调用openFile方法的，如果成功打开流，就拿出视频的宽高来实例化VideoOutput，并加到这个ViewController上。在这个流程中，VideoPlayerViewController就是一个控制器，当这个播放器界面退出的时候，就要把整个ViewController中分配的资源销毁掉，这就会造成整个播放器也不能继续播放这个视频，所以我们应该单独抽取一个VideoPlayerController作为控制器，用来管理AVSync模块和AudioOutput模块，声明一个Protocol用来执行VideoOutput的渲染操作。当有界面可以用来显示视频画面的时候，实现Protocol里面的方法来做画面渲染工作，Protocol声明如下：

```
@protocol PlayerVideoOutputDelegate <NSObject>
- (void) openInputSuccess:(NSInteger) textureFrameWidth textureFrameHeight:
    (NSInteger)textureFrameHeight usingHWCodec:(BOOL)usingHWCodec;
- (void) signalRenderFrame:(VideoFrame*) videoFrame;
@end
```

在VideoPlayerController类中声明这个Protocol类型的delegate，然后开放出一个方法可以用来设置delegate，代码如下：

```
@property (nonatomic, readwrite, copy) id<PlayerVideoOutputDelegate>
    videoOutputDelegate;
- (void) setVideoOutputDelegate:(id<PlayerVideoOutputDelegate>)
    videoOutputDelegate;
```

当然，VideoPlayerController必须是一个单例模式设计的类，因为它要在全局都可以访问到，代码如下：

```
+ (VideoPlayerController *) sharedPlayerController;
{
    static dispatch_once_t pred;
    static VideoPlayerController *sharedPlayerController = nil;
    dispatch_once(&pred, ^{
        sharedPlayerController = [[[self class] alloc] init];
    });
    return sharedPlayerController;
}
```

当然，最重要的方法是playWithURL，这个方法的实现就是实例化AVSync，并且调用openFile方法打开流，如果打开流成功，就调用videoOutputDelegate的openInputSuccess方法（如果videoOutputDelegate已经被设置过），接着实例化AudioOutput组件，并调用AudioOutput的开始播放方法。同时VideoPlayerController要实现AudioOutput组件中的FillDataDelegate协议，在fillAudioData方法中可调用AVSync模块来填充音频数据，同时向videoOutputDelegate发送一个信号要求VideoOutput渲染画面（如果videoOutputDelegate已经被设置过）。这样就完成了整个视频的播放。如果videoOutputDelegate没有被设置过，依然可以播放视频中的声音部分，而当有一个界面可以用来显示视频的画面部分时，那么就设置videoOutputDelegate，并完成渲染工作。如果后续要做小窗口播放器，甚至是后台播放的时候，这个架构依然可以直接使用。

为了达到秒开首屏的极致体验，还有一点要说明，在Input模块，尽量使用CDN厂商提供的HDL协议，因为某些CDN厂商提供的HDL协议对于推流端仍然向一个RTMP的地址去做推流，而分发给观众端的地址为http加flv的格式，这种格式的find_stream_info执行的时间要比分发的rtmp协议的流的时间短，所以尽量采用更加先进的协议，并且大部分的CDN厂商也做了关键帧的缓存，推流CDN节点到边缘CDN节点使用Push的方式等优化，所以使用CDN厂商要比自建机房好很多。

读者可以结合代码仓库中的源码进行分析，并应用到自己的产品中。

12.3.3　统计数据保证后续的应对策略

在12.2节已经分析了推流端应该要做的数据统计，以及数据统计对产品后续迭代以及运营活动的作用。本节将带着读者来完成拉流播放器的数据统计工作。首先要把观众观看的直播场次、观众端的IP地址以及观众端实际去拉流的CDN的边缘节点地址作为基本信息。

1.开始拉流时间

当观众点击开始观看一场直播的时候，系统获取当前的时刻作为开始拉流的时间，以毫秒为单位，这个数据可以在后台绘制出观众观看直播时间的分布图，也有助于运营推广活动的时候掌握应该在哪一段时间段内进行推广。此外，还可以调整客服和后台视频审查的人员安排。

2.连接时间

用户端发起一次Connect，如果失败，直接上报，否则记录Connect时间，单位为毫秒。这个数据有助于开发人员去推动CDN厂商分配更优质的节点。此外，还可以与前面的观众IP地址一块分析是否是某些小运营商导致的DNS劫持等事件。统计方式可以在VideoDecoder类的openInput方法前后增加时间统计，并计算时间差得到连接时间的值。

3.首屏时间

从用户点击某个主播头像开始到展示出第一帧主播视频画面的时间，这其实是观众端体验的一个重要指标。首屏时间越短，观众的体验就越好。可以根据这个时间来帮助开发人员进一步优化观众端的体验，并可以针对首屏时间长的观众，进一步分析到底是主播端的问题还是观众端网络的问题。统计方法就是当第一帧视频帧被渲染出来的时候，将当前系统时间减去开始连接的时间，得到首屏时间。

4.观看时长

从用户开始观看直播开始，到用户退出本次观看，会有一个观看时长，单位是秒。当然，这个观看时长跟主播的内容有很大关系，同时也跟主播推流的稳定度有很大关系，我们可以统计同一个主播在不同网络环境下推流的观众平均观看时长来提示主播选择优质网络的重要性，也

可以换一个维度去统计观众观看的热门直播中，平均时长最长的主播有哪些，还可以帮助运营人员筛选出平台的优质红人。

5.卡顿次数

当用户界面出现一次Loading框或者声音卡顿时，说明用户会卡顿一次。用户卡顿越少，体验越好，这个次数也应该上报给服务器。根据用户的卡顿次数，可以帮助开发人员分析本场直播中的瓶颈到底在哪里。如果本场直播中的观众端都出现了10次以上的卡顿，那么说明有可能是主播推流的问题；如果是观众端的网络链路问题，则可以分析能否去优化播放器的策略。统计方法就是在AVSync模块的showLoading方法中增加次数统计。

至此拉流端的关键策略介绍完毕，读者可以去代码仓库中分析源码，便于深入理解。

12.4　本章小结

本章主要介绍了一个企业级直播产品应该做出的优化与关键处理。首先进行了场景分析并提出了基本的解决方案；其次介绍了推流端（主播端）自适应码率的优化方案，并针对统计数据给出了收集方案与后续处理策略；最后针对拉流端（用户端）的秒开工作制定很多策略，同时也介绍了统计数据的收集以及后续处理策略。

第13章　工欲善其事，必先利其器

工欲善其事，必先利其器。这句话用在开发人员的工作中就是要把自己的开发工具打磨好，这样才可以在工作中游刃有余。前面章节中提到过的ffmpeg、ffplay就属于音视频开发中的辅助工具，而本章和大家一起讨论音视频开发中常用的工具，包括内存泄漏的检测工具、Crash收集的工具以及常用的ADB等工具。其中，有些工具并不仅仅是针对开发人员的，对于测试人员也非常有用。完成本章的学习之后，只要读者善于使用这些工具，在日常工作中肯定可以提高开发以及调试的效率。

13.1 Android平台工具详解

本节主要介绍Android平台下的常用工具，大致分为以下几个方面：如何使用ADB的各个命令完成电脑和开发设备的交互；如何使用MAT工具检查Java端的内存泄漏；如何使用工具检查Native层的内存泄漏；如何使用NDK的各个工具解决编译以及运行中的问题；如何使用Google开源的breakpad来收集Native层的Crash。其中，熟练掌握ADB工具与MAT工具的使用，对于测试人员来讲可以提高定位问题的准确度，对提升整个团队的效率是非常实用的。

13.1.1 ADB工具的熟练使用

ADB的全称是Android Debug Bridge，是Google给开发者提供的一个调试桥工具。借助这个工具，开发者可以在电脑上对Android设备进行很多操作，包括安装/卸载App、查看日志、文件操作、运行Shell命令等。其实ADB就是连接电脑和Android设备的桥梁。ADB工具是Android的SDK目录的platform-tools目录下的一个命令行工具。读者若想要在命令行下随意使用这个工具，则需要将所在的路径配置到path环境变量中。下面以Mac系统为例，展示配置环境变量的过程。

Mac的环境变量配置文件是用户主目录下的.bash_profile文件，如果读者之前没有配置过环境变量，那么系统默认是没有这个文件的，需要读者自己创建；如果已经有了这个文件，那么就键入以下内容：

```
ANDROIDSDKROOT=sdkpath
export PATH=$ANDROIDSDKROOT/platform-tools:$PATH
```

其中，sdkpath是Android SDK所在的路径，输入完毕之后保存并退出，然后执行以下命令：

```
source ~/.bash_profile
```

这行命令是为了让刚才的配置生效，如果不执行这个命令，就需要重启电脑，让系统重新加载这个配置文件。此时尝试输入adb，会出现ADB这个命令行工具的众多提示。

下面看看工作中常用的命令。首先我们要了解在电脑上是否运行着一个adb server，它是用来连接电脑和Android设备的，读者可以执行以下命令：

```
ps –A | grep adb
```

这行命令的作用是列出adb名字的进程，接着可以看到类似如下的

提示：

```
2947 ttys004    0:00.04 adb -P 5037 fork-server server
```

这就告诉我们现在adb server已经运行起来了，如果没有这一行提示，就可以运行以下命令来启动adb server：

```
adb start-server
```

这时再查看adb名字的进程，可以看到adb server正在运行。当然，也可以执行以下命令来关闭adb server：

```
adb kill-server
```

既然讲到这里，就需要提到一个经常让开发者疑惑的问题，即有一些Android设备即使打开了开发者模式也连接不上我们的电脑，这是因为adb server不认识这个Android设备的USB厂商，所以需要把USB的厂商ID加入配置文件中。接下来笔者就以Mac系统为例讲解如何获得USB的厂商ID，打开“关于本机”，然后选择“系统报告”，在硬件选项中点击“USB”，在里面可以找到自己的设备，然后点击找到厂商ID，如图13-1所示。

在图13-1中，厂商ID为0x18d1，我们需要将这个厂商ID放到adb server启动读取的配置文件中，而配置文件是哪一个文件呢？答案就是用户主目录的.android目录下名为adb_usb.ini的文件。配置好之后，需要在命令行重新启动adb server，即执行如下命令：

图　13-1

```
adb kill-server
adb start-server
```

然后利用以下命令来查看这个设备是否连接上了电脑：

```
adb devices -l
```

上述命令用于查看连接到电脑上的设备列表，也就是被adb server识别的设备列表。如果读者是在开发电视上的应用，那么就不可能使用USB连接线将电视设备与电脑连接起来，而应该使用connect命令进行连接：

```
adb connect 192.168.xx.xx:5555
```

这个命令仅能连接同一个局域网内的设备，默认端口号是5555，该命令执行成功后，则和使用USB有线连接一样，只不过是通过TCP协议进行操作的。Android设备被成功连接到电脑上之后，就可以使用ADB工具来操作Android设备了。常用的操作如下。

1.安装/卸载App

如果在电脑上有一个apk文件，想要安装到Android设备中去，可以执行如下命令：

```
adb install test_audio.apk
```

执行这个命令就会将test_audio.apk安装到Android设备上，如果设备上已经有了这个包名的应用，操作就会失败，如果想直接覆盖安装，那么应该在install后加上-r参数，命令如下：

```
adb install –r test_audio.apk
```

而一般的IDE在打包之后也会执行这个命令，并将打包好的apk安装到设备上。如果要卸载一个应用，则必须知道这个应用的包名，因为包名是Android设备上App的唯一标识，命令如下：

```
adb uninstall com.test.audio
```

2.文件操作

如果要将应用程序生成的文件从Android设备中放到电脑上，或者将资源文件从电脑上放到Android设备中去，就需要使用提供的文件操作工具。上述分为两种场景，第一种场景是将文件从Android设备里取出来，我们称为拉取文件，命令如下：

```
adb pull /mnt/sdcard/output.wav ./output.wav
```

这行命令是将Android设备的SD卡根目录下的output.wav文件取出，放到电脑的当前目录下，其中/mnt/sdcard代表SD卡的根目录。而将资源文件从电脑上放到Android设备中去，则称为推送文件，命令如下：

```
adb push input.wav /mnt/sdcard/input.wav
```

上述命令是将当前目录下的input.wav文件放到Android设备的SD卡

根目录下。注意，无论是拉取文件还是推送文件，都可以操作SD卡任意目录下的文件，但不能操作其他系统目录下的文件。

3.查看日志

应用程序运行过程中，在IDE中可以使用logcat来查看日志，如果不使用IDE，那么应该使用ADB工具提供的logcat命令。实际上，IDE中的logcat也是使用ADB工具中的logcat命令来实现的。那我们就对应着IDE中的常用操作，来看一下使用ADB工具如何实现。查看当前Android设备的所有日志信息，直接使用如下命令：

```
adb logcat
```

如果要清空现在所有的日志缓冲区，则执行如下命令：

```
adb logcat -c
```

其中，参数-c代表clear。而最常用到的就是过滤某个标签下的日志，命令如下：

```
adb logcat -s "AudioEncoder"
```

如果要同时过滤多个标签，则使用如下命令：

```
adb logcat -s "AudioEncoder | AudioDecoder"
```

其中，中间的竖线代表或者的意思。在IDE中还有一个比较重要的功能，就是根据日志等级进行过滤，假设AudioEncoder与AudioDecoder中都有Info级别、Debug级别以及Error级别的信息，那么想过滤出Debug级别的信息，可使用如下命令：

```
adb logcat AudioEncoder:D AudioDecoder:D *:S
```

如果要过滤某个进程的所有日志信息，那我们必须知道这个进程的ID（即pid），使用的命令如下：

```
adb logcat --pid 20410
```

其中，数字20410代表要查询的App的进程ID号，具体如何获得这个进程ID号，下面会有介绍。

ADB工具中最常用的命令就是这些，基本也覆盖了IDE中所有的功能。当然，logcat命令里还有输出为文件等功能，这里就不做介绍了。

4.运行Shell命令

使用adb shell命令，可以直接登录到Android设备中执行操作，但是在大部分情况下，可能并不是太方便，因此可以直接将执行的命令放到后面执行，模式如下：

```
adb shell [command line]
```

上面曾留下一个小问题，如果知道一个App的包名，想获取这个App的进程ID，如何处理？事实上，可以使用如下命令来查看：

```
adb shell "ps | grep com.test.audio"
```

命令ps是列出所有的进程，而后面的竖线是管道的意思，即把前面ps命令的输出作为后面命令的输入，而后面的grep命令是过滤操作，将com.test.audio这个进程名字过滤出来，执行命令之后可以看到如下结果：

```
u0_a107   20410 513 1073192 … … com.test.audio
```

可能不同设备得到的结果不尽相同，但是第二列就是我们想要的进程号，即20410。

下面再来介绍常用的命令，比如推送文件结束之后，看一下是否推送成功，可以执行以下命令：

```
adb shell "cd /mnt/sdcard; ls -l | grep wav"
```

这行命令的意思是，先进入SD卡的根目录，然后列出所有的wav文件，并且有修改时间的信息显示。如果要查看一个App的内存占用情况，可以执行如下命令：

```
adb shell dumpsys meminfo com.test.audio
```

其中，com.test.audio是要查看App的包名，执行这个命令的意义就是得到com.test.audio这个App的内存占用信息，下面选取其中比较重要的信息来看一下：

```
                   Pss  Private  Private  SwapPss     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
   Native Heap    8709     8704        0        0    19456    10809     8646
   Dalvik Heap    8034     8004        0        0    17620    10572     7048
    EGL mtrack   27012    27012        0        0
          TOTAL   67242    47920    13192       18    37076    21381    15694
```

从上面的信息中可以看到，Native Heap代表Native层的堆的大小，因为在Android引擎中，虽然对每一个App占用的内存大小有定义，但是对于App使用Native代码进行内存的分配和销毁却是不进行约束的，所以这里可以理解为Native Heap，即Native层所占用内存的大小，如果一直在增长，则需要检查程序是否有内存泄漏，而对应的Dalvik Heap则是对应的Java层占用的内存大小，最后一项EGL mtrack则代表OpenGL ES所占用的显存大小，因为在手机设备上都是集成显卡，所以没有单独的显存，而是和内存进行共享的。所以，我们查看内存信息时，也可以得到显存的信息，如果这一行的内存在无限增大，则应该注意在应用程序中是否有未释放的纹理对象或者帧缓存对象等OpenGL ES对象。

Android 4.4以上版本提供了一个Shell命令进行录屏，对于测试人员来说，可以比较方便地录制自己的操作（可以在设置中打开显示点按操作，会在视频中有鼠标点），命令如下：

```
adb shell screenrecord --size 720x1280 --bit-rate 500000 /mnt/sdcard/case-1.m
```

命令中的size是宽乘以高，所以要注意期望的是横板的视频还是纵版的视频，比特率可以依据自己的场景来设置，最终文件放置到SD卡的根目录下，以case-1.mp4作为存储文件名。在设备上执行完操作后，可使用快捷键Ctrl+C来停止整个视频的继续录制，然后使用adb pull命令将case-1.mp4拉取到电脑上，以进行播放操作，这时可以看到我们刚才执行的所有操作。

在工作中有可能写出的一些线程会在后台一直跑，它们有可能是去队列里取数据，虽然这个队列不是阻塞队列，但也有可能是开发不小心写出的空转线程，可以使用以下命令来查看有没有空转的线程：

```
adb shell top -t -m 5
```

top命令是查看进程占用CPU情况的指令，在Android设备中，top指令后加上参数-t代表查看线程占用CPU的情况，-m代表仅查看CPU占用最高的5个线程。如果看到某个线程占用CPU比较高，开发人员就可以注意是否需要优化或者更改实现方式。

至此，ADB工具的常用命令就介绍完了，读者可以多加练习，如果运用到日常工作中，一定可以提高工作效率。

13.1.2　MAT工具检测Java端的内存泄漏

平常在开发Android应用的时候，稍有不慎就有可能产生OOM（Out of Memory）异常。虽然Java语言有垃圾回收机制，但也不能杜绝内存泄漏、内存溢出等问题。Android系统会分配给每个应用程序一个Dalvik虚拟机，这样的分配策略避免了因系统中的一个App崩溃而导致其他App甚至是系统的崩溃。所以Android系统对于每一个Dalvik虚拟机也分配了一定的内存。当开发的App占用的内存超过了这个内存上限时，就会产生OOM异常。为了避免OOM，开发人员必须为App做一定的内存分配策略。在一般的Android程序中，占用内存最大的部分就是图片的缓存，因为Bitmap是存储在内存中的，通常使用LRUCache来作为图片的缓存池。本节介绍的内容是如何定位以及修复内存泄漏。内存泄漏是指有些对象是从虚拟机的根对象通过引用链可达的，但是这些对象在App中再也不会被用到了。由于这些对象是从虚拟机的根对象可达的，导致这些对象不会被垃圾回收器回收，而系统对这些对象又不再使用，从而导致它们成为内存垃圾，永远不被使用，又永远不被回收掉。造成内存泄漏的原因一般是开发人员的不良编码习惯或者对Android引擎不熟悉。本节使用DDMS（Dalvik Debug Monitor Server）与MAT（Memory Analyzer Tool）工具来定位内存泄漏，并最终解决内存泄漏问题。

为了深入理解内存泄漏，需要先来看看Dalvik虚拟机对于内存的分配情况，如图13-2所示。

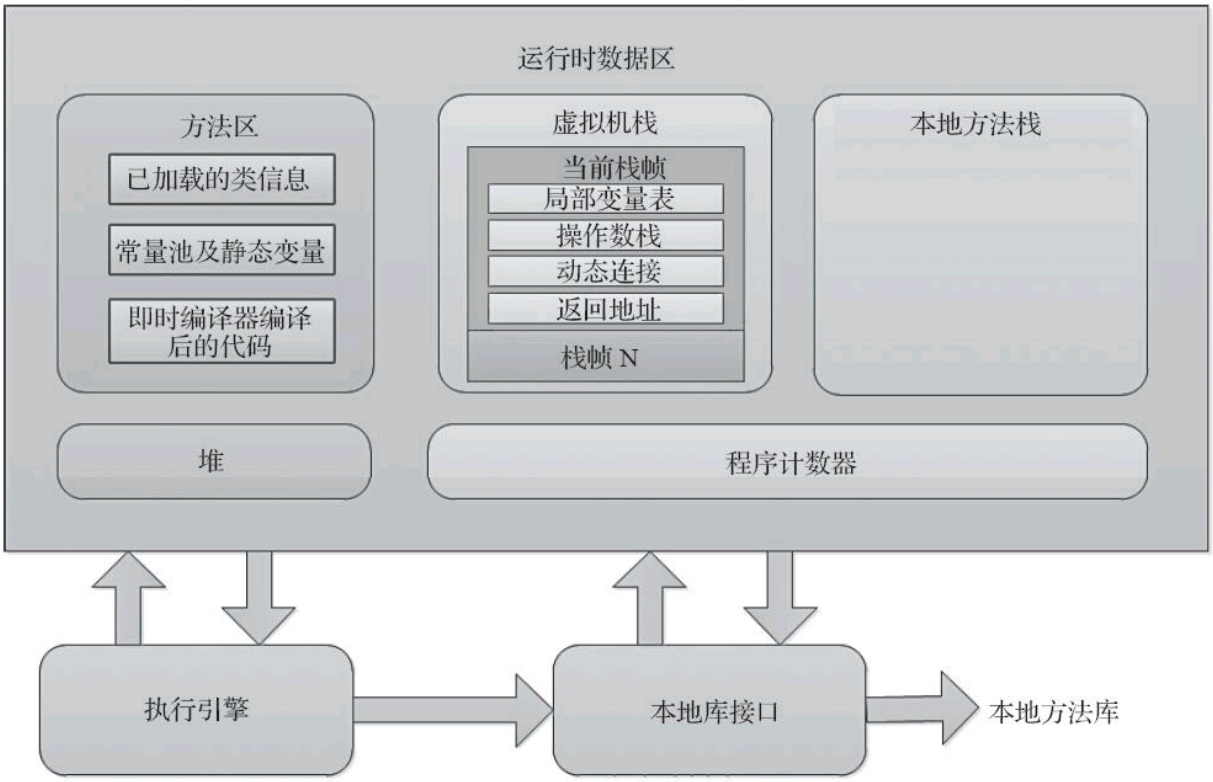


图　13-2

如图13-2所示，当类加载器（Class Loader）将一个类的字节码加载到虚拟机中的时候，首先会将这个类的信息存入方法区，这个类的静态变量也在方法区中；当基于这个类创建一个对象时，这个对象就存储于堆内存中。方法区内存区域和堆内存区域都是线程共享的数据区域。图13-2中的虚拟机栈是指当前线程运行的现场信息，属于线程间隔离的数据；本地方法栈是指对于Native方法的调用现场，对于线程也是独立的；程序计数器就是记录程序运行的下一条指令的地址，也是线程之间相互独立的区域。

了解了虚拟机内存分配方面的知识，再来看一下虚拟机的垃圾回收机制。垃圾回收机制主要针对堆内存区域进行垃圾回收。这里所谓的垃圾就是指以根对象为起点，从这个起点向下搜索，搜索走过的路径称为引用链，当一个对象不在任何引用链上时，则说明这个对象是不可能再被使用的，则标记为垃圾，垃圾回收器在下一次回收的时候就会把这块内存释放掉。这种垃圾回收方法充分解决了循环引用等问题，是比较先进的垃圾回收机制。标记是否为垃圾的过程如图13-3所示。

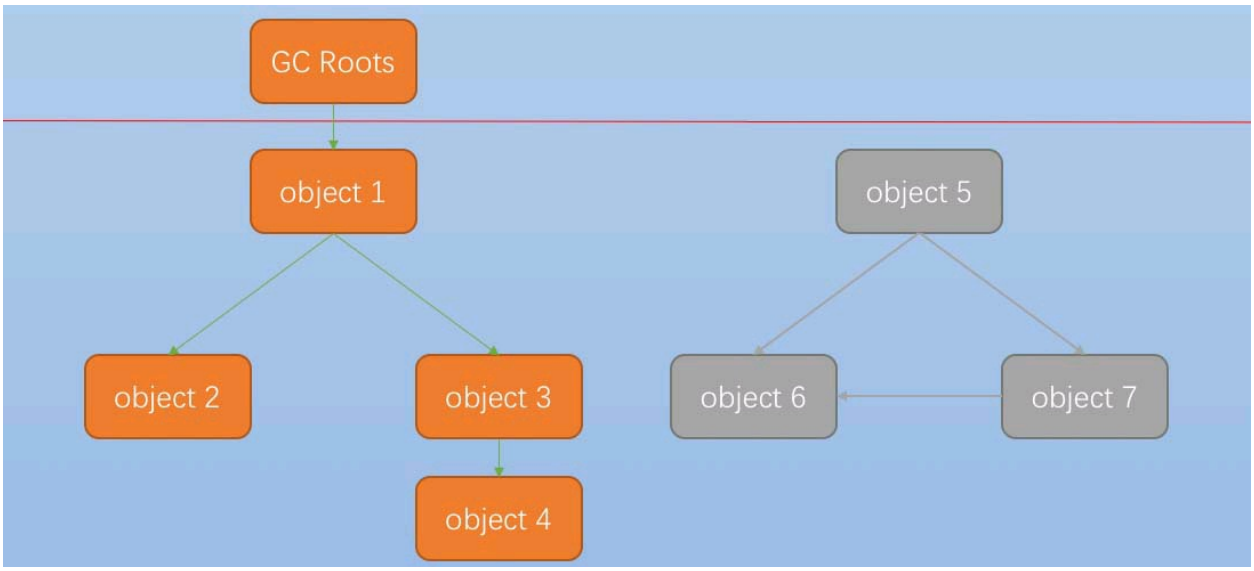


图　13-3

在图13-3中，object 1、object 2、object 3、object 4都存在于根对象（GC Roots）的引用链上，或者说是从根对象可达的对象。而object 5、object 6、object 7尽管有循环引用，但形成的只是一座孤岛，并不存在于根对象的引用链上，或者说从根对象是不可达的对象。那上面所讲的根对象（GC Roots）又有哪些呢？大致可分为以下几类。

·虚拟机栈中引用的对象，一般是当前使用中的局部变量。

·方法区中类静态属性引用的对象，就是静态变量对应的对象。

·方法区中常量引用的对象。

·本地方法栈中JNI（即一般所说的Native方法）引用的对象。

当由于开发人员不好的编码习惯或者对某些框架（Android引擎）不熟悉导致一些对象不再使用，但是还处于从根对象（GC Roots）可达的状态时，就造成了内存泄漏。本节将从三个层面来定位内存泄漏并最终给出解决方案，这三个层面分别是方法区内静态变量引起的内存泄漏、匿名内部类引起的内存泄漏（会以Android引擎提供的Handler的使用作为实例进行介绍）以及本地方法栈引起的内存泄漏。

1.方法区内静态变量引起的内存泄漏

一个类的静态变量处于虚拟机内存的方法区，属于根对象集合的一部分，开发中应该尽量避免给静态变量赋值Activity类型的实例。但是，有的开发者为了快速完成产品的需求，常以最快的方式（比如静态变量引用了Activity）来实现，这对于整个开发过程来讲是糟糕的。这种做法最终并不能让产品快速上线，即使产品上线了，用户使用之后也会遇到各种问题。下面展示一段代码示例：

```
public class AudioProcessorService {
    private static List<Context> mContexts = new ArrayList<Context>();
    public AudioProcessorService(Context context) {
        AudioProcessorService.mContexts.add(context);
    }
}
```

在主界面的MainActivity中点击按钮会跳转到TestJavaMemoryLeakActivity，在Activity的onCreate方法里创建一个AudioProcessorService类型的对象，接着在Activity界面中有一个按钮，点击该按钮就可以调用对象中的其他方法，最后点击返回退出主界面MainActivity。为了方便演示内存泄漏，在TestJavaMemoryLeakActivity中加入一个5MB大小的字节数组类型的成员变量，在实际生产环境下，每个界面有大量的图片与View，同样也会占用很大的内存，代码如下：

```
private byte[] buffers = new byte[5 * 1024 * 1024];
```

重复上述过程3～5次，肯定会造成TestJavaMemoryLeakActivity的内存泄漏。下面就来看看到底有没有内存泄漏，以及是什么操作导致的内存泄漏。首先切换到DDMS视图界面，如图13-4所示。

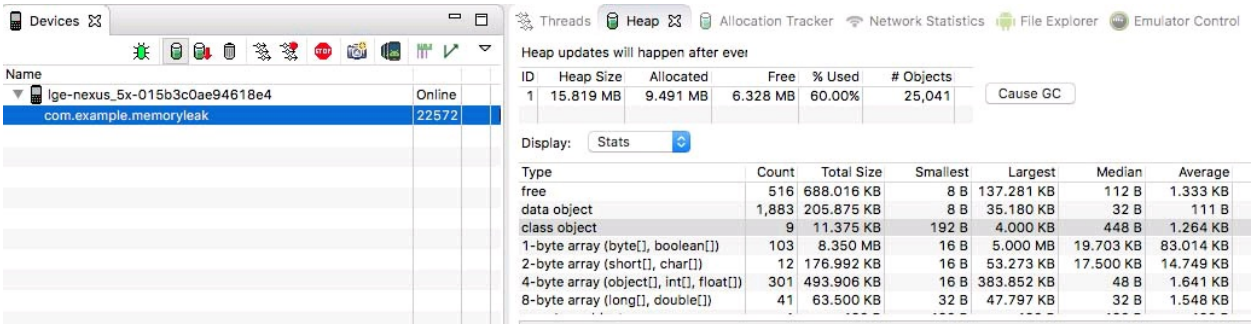


图　13-4

可以多次点击图13-4中右边的Cause GC按钮，主动让虚拟机多执行几次垃圾回收，然后点击左上角的Dump HPROF file按钮，将当前com.example.memoryleak这个应用的虚拟机内存分配情况以文件的形式下载下来。如果Eclipse中安装了MAT插件，就默认使用该插件打开这个文件；如果没有插件，就会以文件的形式保存到用户指定的路径下，然后使用单独的MAT工具打开这个文件。无论是单独的MAT工具还是Eclipse中的MAT插件，打开之后的结果如图13-5所示。

从图13-5中可以看到，饼形图是App内存的总览图，内存占用大小为25.8MB，有三个5MB的大对象存在，图中下方还有三个比较重要的按钮，分别是Histogram、Dominator Tree和Leak Suspects。利用MAT分析内存泄漏也是依靠这三个工具，我们可以先打开视图Leak Suspects，如图13-6所示。

图13-6所示的饼形图展示可能存在内存泄漏的对象，可以在这个视图的下半部分找到第一个泄漏5MB内存的对象，如图13-7所示。

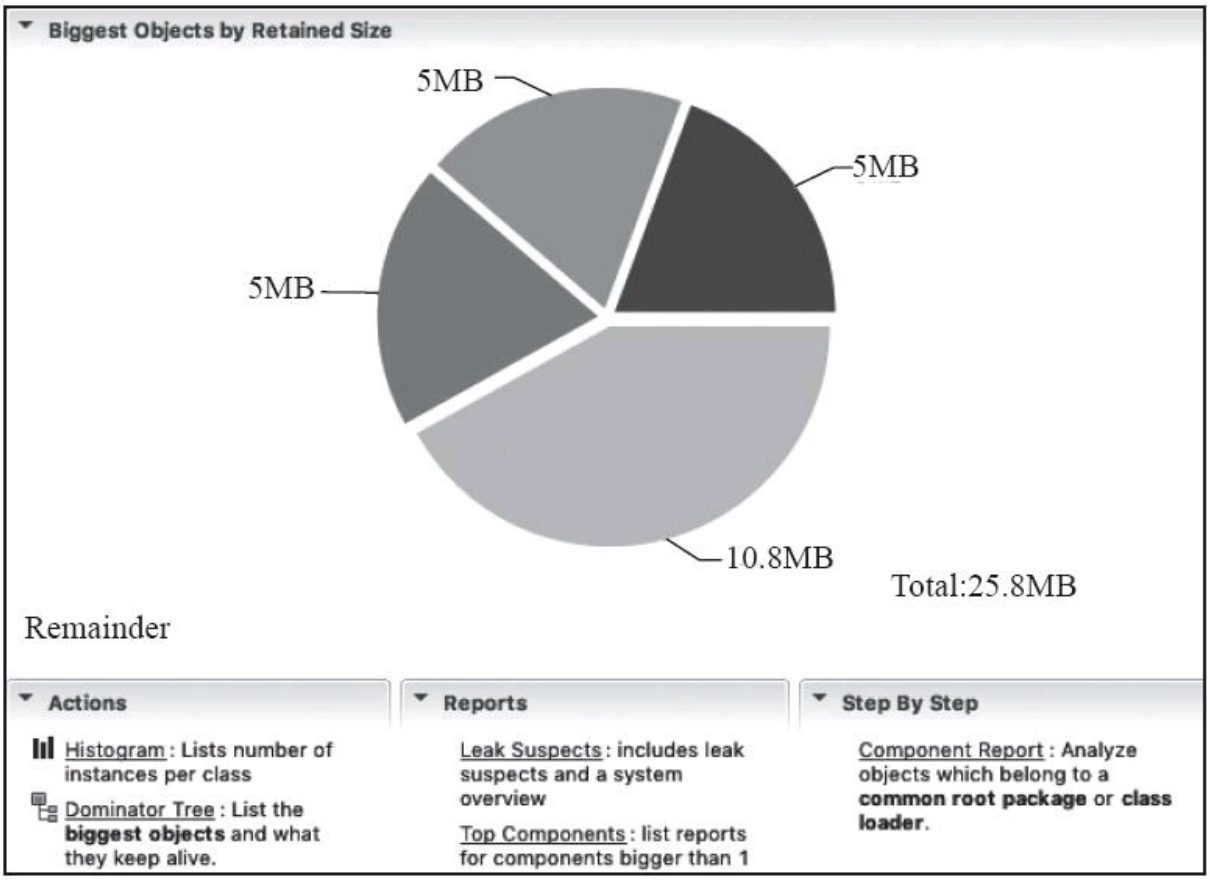


图　13-5

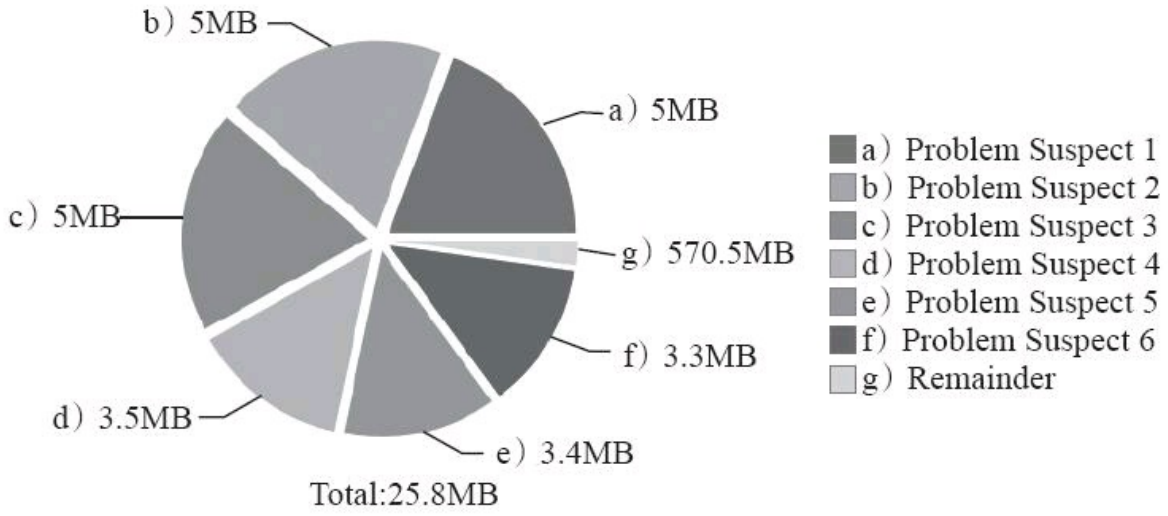


图　13-6

图13-7中显示了哪一个对象被哪一个类加载器加载进来，占用了多大内存，以及主要是由哪一个成员变量占用的内存。可以点击Details进入下一级界面来查看这个对象的引用细节，如图13-8所示。

Problem Suspect 1

One instance of **"com.example.java.layer.TestJavaMemoryLeakActivity"** loaded by **"dalvik.system.PathClassLoader @ 0x12c1a430"** occupies **5,246,160 (19.36%)** bytes. The memory is accumulated in one instance of **"byte[]"** loaded by **"<system class loader>"**.

Keywords
com.example.java.layer.TestJavaMemoryLeakActivity
byte[]
dalvik.system.PathClassLoader @ 0x12c1a430

Details »

图　13-7

图13-8中展示了一个5MB大小的字节数组，这个数组被TestJavaMemory-LeakActivity类的一个实例里的成员变量buffers所引用，而这个实例是被一个ArrayList里的Index为2的元素所引用，这个ArrayList实例又被AudioProcessorService类的实例变量mContexts所引用；这个数组同时被另外一个成员变量mContext所引用，mContext是Button对象中的成员变量。而Button就是这个Activity里的一个按钮组件，所以造成内存泄漏的就是AudioProcessorService中的静态变量

mContexts。如果再向下看，另外两个泄漏的Activity对象分别被ArrayList中Index为0和Index为1的元素所引用。

Class Name	Shallow Heap	Retained Heap
byte[5242880] @ 0xca6b4000	5,242,896	5,242,896
buffers com.example.java.layer.TestJavaMemoryLeakActivity @ 0x12c464c0	232	5,246,160
[2] java.lang.Object[10] @ 0x12c283f8	56	56
elementData java.util.ArrayList @ 0x12c02538	24	80
mContexts class com.example.java.layer.AudioProcessorService @ 0x12c4d900 System Class	8	416
mContext android.widget.Button @ 0x12c70c00 »	720	2,536
∑ Total: 2 entries		

图　13-8

结合上述代码，可以看到确实是因为这个静态变量作为根对象（GC Roots）一直引用着Activity实例，致使退出Activity的时候，Activity对象还是不能够被虚拟机的垃圾回收器回收掉。显然，在实际开发过程中，不太可能使用一个ArrayList存放Context类型的变量。但是有可能会使用一个静态变量引用。所以在开发过程中，不要使用静态变量或者常量来引用Activity的实例，否则会造成内存泄漏。另外，在日常开发中，Activity中很少会直接有这么大的byte数组，更多的情况下以Bitmap的形式存在，比如一个宽为360、高为640并且表示格式为RGBA_8888的Bitmap对象所占用的内存大小为：

```
360 * 640 * 4 = 900KB
```

如果有8张图片，那么占用7MB的空间，所以一般Activity的实例都属于内存对象，实际开发中可以优先检查Activity的内存泄漏。

2.匿名内部类引起的内存泄漏

下面从匿名内部类角度来分析造成内存泄漏的案例。在Java中，无论是匿名内部类还是内部类，都可以引用所在类中的实例变量或者方法，这是为什么？实际上，内部类对所在类的实例对象有一个引用，并且这个引用是一个强引用的关系，即如果这个内部类构建的对象没有被释放掉，那么内部类所引用的外部类创建的对象也永远不会被释放。而开发者在开发Android程序的时候，最常使用的是Android SDK提供的Handler来进行线程之间的通信。在创建Handler的时候，为了简单方便，开发者一般都会直接写一个匿名内部类或者内部类继承自Handler，并重写方法handleMessage在主线程中处理界面的渲染以及绘

制。下面展示一段大家在开发中最常使用的代码：

```
public class TestJavaMemoryLeakActivity extends Activity {
    private static final int ON_HOUR_KEY = 100;
    private byte[] buffers = new byte[5 * 1024 * 1024];
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_java_leak);
        Message msg = new Message();
        msg.what = ON_HOUR_KEY;
        handler.sendMessageDelayed(msg, 1000 * 60 * 60);
    }
    private Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case ON_HOUR_KEY:
                    displayTip();
                break;
                default:
                break;
            }
        }
    }
    public void displayTip() {
        Log.i("problem", "开播一个小时了。。。");
    }
}
```

上面这段代码完成的功能很简单，就是进入某界面就开始计时，一个小时之后输出一行日志“开播一个小时了…”。这其实就是在模拟一个直播App的主播端，待主播开播一个小时之后提示主播。这是一个非常典型的场景。我们可以模拟主播，在主界面点击按钮进入这个界面之后，等待几秒钟，然后退回主界面，重复以上操作3～5次。再进入DDMS界面，点击左上角的Dump HPROF file按钮，从而查看到内存泄漏的MAT界面，如图13-9所示。

Class Name	Shallow Heap	Retained Heap
byte[5242880] @ 0xcb0b6000	5,242,896	5,242,896
buffers com.example.java.layer.TestJavaMemoryLeakActivity @ 0x12c826a0	232	5,246,128
this$0 com.example.java.layer.TestJavaMemoryLeakActivity$MyHandler @ 0x12ca70a0	32	32
target android.os.Message @ 0x12c457c0	64	320
mMessages android.os.MessageQueue @ 0x12c04c18	40	472
<Java Local, Native Stack> java.lang.Thread @ 0x74487000 Thread	136	1,344
mQueue android.os.Looper @ 0x12c419a0 »	32	32
mQueue android.app.ActivityThread$H @ 0x12c419e0 »	32	32
mQueue com.android.internal.view.IInputConnectionWrapper$MyHandler @ 0x12c884c0 »	32	32
mQueue android.view.accessibility.AccessibilityManager$MyHandler @ 0x12c67c00 »	32	32
mQueue android.hardware.display.DisplayManagerGlobal$DisplayListenerDelegate @ 0x12c88680 »	32	32
mQueue android.view.ViewRootImpl$ViewRootHandler @ 0x12c88400 »	32	32
mQueue android.os.Handler @ 0x12c566c0 »	32	32
∑ Total: 8 entries		

图　13-9

从图13-9可以看到，这个Activity实例是被它内部类MyHandler的一个实例引用的，而MyHandler实例是被一个Message实例中的成员变量target引用的，Message对象又是被MessageQueue实例中的成员变量mMessages所引用的，MessageQueue对象则是被整个Android引擎中的本地方法栈、Looper等完全可以作为根对象集合的对象引用着的，所以这个Activity对象一直无法被释放。上述引用关系链是通过内存泄漏工具分析出来的，下面从另外一个角度即对应着Android框架的源码分析为何这个界面退出了却无法被垃圾回收器回收掉。下面我们看看Handler、MessageQueue、Looper的结构关系图，如图13-10所示。

图　13-10

从图13-10中可以看到，Handler向外暴露两种类型的接口：第一种是sendMessage接口，即开发者构造一个Message对象，然后调用sendMessage或者sendMessageDelayed利用Handler将消息发送出去；第二种是postRunnable接口，即开发者构造一个Runnable类型的对象，然后调用post方法或者postDelayed方法。在Handler内部，无论是对于哪一种接口，都会构造一个Message对象。对于post类型的接口，会将Runnable对象赋值给Message的成员变量callback，以备后用。然后将这个Handler赋值给Message的成员变量target，并按照当前时间戳加上Delayed的时间，作为执行时间调用消息队列（MessageQueue）的enqueueMessage方法，且将Message对象放入消息队列（MessageQueue）。而Looper的loop方法在Android引擎启动应用的时候就在主线程中不断地调用消息队列（MessgeQueue）的next方法，它会

取出消息（Message），并调用消息的target（开发者自定义的Handler）的dispatchMessage方法，在dispatchMessage方法中会先判定消息有没有被设置callback。如果有，说明是post接口推送进来的消息，则调用callback的run方法；如果没有被设置callback，则调用handleMessage方法，一般开发者会重写这个方法来处理自己的消息回调，这样就完成了子线程和主线程的交互操作。

上述分析中，核心的实现其实是消息队列（MessageQueue）的enqueueMessage方法和next方法，这两个方法会按照BlockingQueue的方式响应外界的调用，只不过这个队列是按照时间顺序组建起来的消息链表。从源码进行分析，读者应该会更加清晰为何Activity的实例不会被Dalvik垃圾回收器回收掉了，因为在调用了Handler对象的发送延迟消息代码后，这个Handler实例就被Message对象所引用，而Message由于还没有到达执行时间，所以一直存储在MessageQueue中，这个引用关系也一直存在，Activity实例也一直在根对象（GC Roots）可达路径上，所以就不可能被回收掉。虽然只是一段时间内不被回收掉，但是在这段时间内会影响到用户的使用，甚至有可能造成OOM异常的抛出，最终使整个应用崩溃。既然发现了问题，解决问题也就变得非常简单了。一般来说，解决这类问题有两种方法。

方法一：在Activity的生命周期方法onDestroy中将Handler中的所有Message都清空，这样就不会有任何引用了，代码如下：

```
@Override
protected void onDestroy() {
    super.onDestroy();
    handler.removeCallbacksAndMessages(null);
}
```

方法二：将这个内部类改为一个静态的内部类，既然是静态的，那么就属于类，而不属于类的实例，这样就不会对所在类的对象有一个强引用的关系了。而在Handler中，又需要调用所在类的方法执行一些页面绘制操作，那么就需要给Handler增加一个所在类的虚引用类型的实例变量，而虚引用会在垃圾回收器进行FullGC的时候回收掉，这就相当于切断了Handler这个类实例的引用，就不会再产生内存泄漏了，代码如下：

```
static class MyHandler extends Handler {
```

```
    private WeakReference<TestJavaMemoryLeakActivity> activity;
    public MyHandler(TestJavaMemoryLeakActivity activity) {
        this.activity = new WeakReference<TestJavaMemoryLeakActivity>(activit
    }
    @Override
    public void handleMessage(Message msg) {
        TestJavaMemoryLeakActivity context = activity.get();
        if(null == context) {
            return;
        }
        switch (msg.what) {
            case ON_HOUR_KEY:
                context.displayTip();
                break;
            default:
                break;
        }
    }
}
```

构造Handler的时候，代码变为如下所示的形式：

```
private MyHandler handler = new MyHandler(this);
```

上述两种方法都可以解决使用Handler情况下的内存泄漏，其他情况下，对于内部类的使用，大家一定要考虑内部类的生命周期不应该长于所在的外部类，否则应该回到这个模块的设计阶段，再来理清整个模块中类的关系。

3.本地方法栈引起的内存泄漏

下面从本地方法栈的角度分析造成内存泄漏的案例。在开发中，如果要在Native层调用Java层的方法，一般都会在JNI层建立一个全局的对象引用并存储到全局变量中。代码如下：

```
jobject globalObj = 0;
JNIEXPORT void JNICALL Java_com_example_java_layer_JNILayer_init(JNIEnv * env
        jobject obj) {
    JavaVM *g_jvm = NULL;
    env->GetJavaVM(&g_jvm);
    globalObj = env->NewGlobalRef(obj);
}
JNIEXPORT void JNICALL Java_com_example_java_layer_JNILayer_process
        (JNIEnv * env, jobject obj) {
    LOGI("Enter Java_com_example_java_layer_JNILayer_process...");
}
```

```
JNIEXPORT void JNICALL Java_com_example_java_layer_JNILayer_destroy
        (JNIEnv * env, jobject obj) {
    LOGI("Enter Java_com_example_java_layer_JNILayer_destroy...");
    //env->DeleteGlobalRef(globalObj);
    //globalObj = 0;
}
```

以上代码中，Init方法利用JNIEnv创建了一个GlobalRef全局对象，我们可以依靠这个全局对象在Native层的任何地方去调用Java层的代码。使用JNIEnv创建全局对象后，就会在本地方法栈中保留对Java层该对象的一个引用，就相当于在根对象（GC Roots）集合的本地方法栈中加入了GlobalRef。当虚拟机的垃圾回收器进行垃圾回收的时候，Java层的这个对象就不会被回收掉，所以在最终结束了调用或者生命周期的时候一定要删除GlobalRef。如果忘记删掉，就会造成内存泄漏。这里将上述代码中destroy方法中删除全局对象的方法注释掉，这样就伪造了相应的案例来分析内存泄漏。为了有效观察到内存泄漏，在有native方法的Java类JNILayer中引用AudioProcessorService的实例。代码如下：

```
public class JNILayer {
    private AudioProcessorService mService;
    public JNILayer(AudioProcessorService service) {
        mService = service;
    }
    public native void init();
    public native void process();
    public native void destroy();
}
```

当然，在AudioProcessorService类中要初始化JNILayer，并且在AudioProcessorService类中引用自己创建的Activity类型的实例，代码如下：

```
public class AudioProcessorService {
    private JNILayer jniLayer = new JNILayer(this);
    private Context mContext;
    public AudioProcessorService(Context context) {
        mContext = context;
        jniLayer.init();
    }
    public void doProcess() {
        jniLayer.process();
    }
    public void destroy() {
        jniLayer.destroy();
```

```
        }
    }
```

在Activity的生命周期方法onCreate中会初始化AudioProcessorService实例，然后在生命周期方法onDestroy中会调用AudioProcessorService的Destroy方法。在主界面中，点击按钮进入Activity，然后退出，反复执行上述操作几次。之后进入DDMS界面，并点击左上角的Dump HPROF file按钮，即可查看到内存泄漏的MAT界面，如图13-11所示。

Class Name	Shallow Heap	Retained Heap
byte[5242880] @ 0xca1b3000	5,242,896	5,242,896
buffers com.example.java.layer.TestJavaMemoryLeakActivity @ 0x12c783d0	232	5,246,128
mContext com.example.java.layer.AudioProcessorService @ 0x12c3a3f0	16	16
mService com.example.java.layer.JNILayer @ 0x12c3a3e0 Native Stack	16	16
mContext android.widget.Button @ 0x12c17400 »	720	2,536
∑ Total: 2 entries		

图　13-11

从图13-11中可以看到，选中的一行显示这个Activity对象被根对象（GC Roots）引用的是JNILayer类中的本地方法栈（Native Stack），对应到代码中，就是JNI层的类中Destroy方法没有把GlobalRef删除掉。放开注释掉的代码，然后以同样的流程进行操作，就不会再有内存泄漏。因此，一旦在JNI中创建了一个GlobalRef，一定要注意在最终不需要的时候删除掉，否则会造成内存泄漏。

上述内存泄漏是广大开发者常会犯的错误，当然还有一些其他内存泄漏的情况。比如使用BroadcastReceiver的时候调用了register，但是在不需要的时候没有调用unregister；又比如访问ContentProvider用的Cursor没有关闭或者I/O没有关闭等。所以开发者在日常开发中应保持良好的编码习惯，并在团队内做好代码评审等工作，以确保产品的顺利上线与良好体验。

13.1.3 NDK工具详解

NDK路径下的模块在前面章节中已经介绍过，本节重点介绍NDK提供的gcc工具所在的路径，如下：

```
NDK_GCC_TOOL_DIRECTORY=$NDKROOT/toolchains/arm-linux-androideabi-4.9/prebuilt
    /darwin-x86_64/bin/
```

在这个路径下有很多gcc的工具，熟练使用这里的工具可以大大提高我们的工作效率。下面逐一介绍日常工作中用到的这些工具。

1.利用readelf命令输出动态so库中的所有函数

命令如下：

```
./arm-linux-androideabi-readelf -a libMemoryLeak.so > func.txt
```

打开func.txt可以看到so中的所有函数，如果运行Android设备上的应用报错说找不到某个JNI方法，就可以使用这行命令将so库中的方法全部导出来，然后搜索对应的JNI方法，看看到底有没有被编译到动态库中，在最后有一个参数Tag_CPU_name写着ARM v7，代表以armv7的架构进行编译。注意这里的输入so包一定要是obj目录下带symbol file的so库，而不应该是libs目录下的so库。

2.利用objdump命令将so包反编译为实际的代码

指令如下：

```
./arm-linux-androideabi-objdump  -dx libMemoryLeak.so > stacktrace.txt
```

打开stacktrace.txt，是这个动态so库的符号表信息，可以看到编译进来的所有方法以及调用堆栈的地址。后面介绍的动态检测内存泄漏中，获取方法调用堆栈的地址信息就是这个地址。

3.利用nm指令查看静态库中的符号表

指令如下：

```
./arm-linux-androideabi-nm libaudio.a > symbol.file
```

打开symbol.file，可以看到静态库中所有的方法声明，如果在编译so动态库的过程中碰到undefined reference类型的错误，或者duplicated reference，可以使用这条指令将对应静态库的所有方法都导出来，然后看一下到底有没有或者重复定义的方法。

4.利用g++指令编译程序

指令如下：

```
SYS_ROOT=$NDKROOT/platforms/android-21/arch-arm/
arm-linux-androideabi-g++ -O2 -DNDEBUG --sysroot=$SYS_ROOT –o libMemoryLeak.s
    jni/test_memory.cpp -lm –lstdc++
```

不论是.a的静态库还是.so的动态库，甚至是可执行的命令行工具，都可以使用g++编译，而常使用的ndk-build命令实际上是对Android.mk以及g++的一种封装，这种封装将g++的细节封装起来，让开发者更加方便。附录中介绍NE10的交叉编译到Android平台，使用的就是g++的编译方式，只不过用CMake封装了一次。

5.利用addr2line将调用地址转化成代码行数

指令如下：

```
./arm-linux-androideabi-addr2line  -e libMemoryLeak.so 0xcf9c > file.line
```

文件file.line里就是调用地址0xcf9c对应的代码文件和对应的行数，注意这里输入的so必须是obj目录下的带有symbol file的so，否则解析代码文件与行数不成功。

6.利用ndk-stack还原堆栈信息

指令如下：

```
ndk-stack -sym libMemoryLeak.so –dump tombstone_01 > log.txt
```

当程序出现Native层的Crash时，系统会拦截并将Crash的堆栈信息放到/data/tomb-stones目录下，存储成为一个文件，系统会自动循环覆盖，并且只会保留最近的10个文件。如何将这里的信息转换为实际的代码文件可以使用ndk-stack工具，这个工具和ndk-build在同一个目录下。注意，-sym后面的so文件必须是obj目录下的带有symbol file的动态库，-dump后面的就是从Android设备中取出来的Crash文件。

13.1.4 Native层的内存泄漏检测

上一节中完成了Java层内存泄漏的检查与修复，本节将带着大家进行Native层的内存泄漏检查和修复。本节会从两个层面进行介绍：第一个层面是静态检测Native层的内存泄漏，可以作为svn或者git提交代码的检测条件；第二个层面是在程序运行中检测内存泄漏，相较于静态检查，这个层次的内存泄漏检查更为准确。

1.Native代码的静态检测

CppCheck是一个用于检查C/C++代码缺陷的静态检查工具。不同于C/C++编译器及其他分析工具，CppCheck只检查编译器检查不出来的bug，不检查语法错误。静态代码检查就是使用一个工具检查我们写的代码是否安全和健壮，是否有隐藏的问题。比如下面的代码：

```
int n = 10;
char* buffer = new char[n];
buffer[n] = 0;
```

这完全符合语法规范，但是静态代码检查工具会提示此处有溢出。也就是说，它相当于一个更加严格的编译器。目前使用比较广泛的C/C++静态代码检查工具有Cppcheck和pc-lint等。pc-lint是资格最老、最强有力的代码检查工具，但是是收费软件，并且配置起来有一点麻烦。而Cppcheck是一个免费开源的软件，所以本节就以Cppcheck工具为例来讲解如何对Native代码进行静态检测，以及如何针对检测出来的问题进行修复与更正。

首先是Cppcheck的安装。Cppcheck有两种方式可以供开发者使用：一种是GUI的方式，另外一种是命令行的方式。本书中所有的使用都采用命令行的方式来介绍。可以下载cppcheck的源码，然后自己进行配置、编译和安装，也可以使用包管理工具（Mac上的brew、linux上的apt-get等）直接安装cppcheck。如果在Linux系统下，可以通过下面的链接来下载源码：

```
http://sourceforge.NET/projects/cppcheck/files/cppcheck/
```

打开上述链接，选择对应的版本，解压。接下来进入目录，然后使用下述命令进行编译和安装：

```
g++ -o cppcheck -Ilib cli/*.cpp lib/*.cpp
make install
```

笔者使用的是Mac系统，最方便的方式是使用brew工具进行安装，可使用下述命令来安装cppcheck：

```
sudo brew install cppcheck
```

安装成功之后，可以输入cppcheck命令，若看到它的帮助文档，则代表安装成功。这个工具的使用方法非常简单，下面来看如何使用cppcheck。

如果仅有一个文件需要检查，可以直接将这个文件写到cppcheck后面，命令如下：

```
cppcheck memory_leak/native_mem_case/png_decoder.cpp
```

如果目录下所有的文件都需要检查，可以直接将要检查的目录放在cppcheck后面，这样它就可以递归检查这个目录下所有的文件了，并且会将进度和检测结果直接输出到屏幕上，如下：

```
cppcheck ./memory_leak/
```

cppcheck的检测结果是根据我们设置给它的检查规则来生成的，cppcheck提供的检查规则定义如下：

·error：出现的错误，如果不写，则是默认的选项。

·warning：为了预防bug防御性编程建议信息。

·style：编码格式问题（没有使用的函数、多余的代码等）。

·portablity：移植性警告。该部分如果移植到其他平台上，可能出现兼容性问题。

·performance：建议优化该部分代码的性能。

·information：一些有趣的信息，可以忽略不看。

其中，error规则是默认的选项，一般可以开启warning级别的检测，代码如下：

```
cppcheck --enable=all ./memory_leak/
```

如果要想把进度输出到屏幕上，并把最终检测的结果输出到文件中，可以使用如下命令：

```
cppcheck --enable=all ./memory_leak/ 2> err.txt
```

命令行中的2代表Shell命令的内置描述符中的stderr，即标准的错误输出，上述命令即将标准的错误输出到err.txt，接着打开err.txt就可以看到错误信息了。对于某些不想检测的文件或者路径，可以使用config-exclude参数指出，这在引用一些第三方库源码的时候是一个典型的应用场景。同时在cppcheck中也可以指定输出不确定的信息，这时仅需要把inconclusive参数加上，同时也可以指定标准（比如std标准、c99标准、c11标准），所以最终命令如下：

```
cppcheck --enable=warning --inconclusive --std=posix ./memory_leak/ 2> err.t
```

接下来伪造一些平常在开发中偶尔遇到的问题，并使用上述的cppcheck命令进行检测。在代码仓库的PngPicDecoder类的头文件中声明一个实例变量，代码如下：

```
int bufferCursor;
```

在构造函数中没有初始化这个实例变量，此时cppcheck会报出错误

警告，代码如下：

```
[png_decoder.cpp 5]: (warning) Member variable 'PngPicDecoder::bufferCursor'
```

如果分配一个数组类型的buffer，在其中一个条件分支内释放，但在另外一个条件分支内没有释放，cppcheck也可以检测出来，代码如下：

```
short* buffer = new short[1024];
if(condition) {
    //Do Something
    delete[] buffer;
} else {
    //Do AnotherThing
}
return;
```

cppcheck对上述代码进行检测之后，会报出如下错误：

```
[png_decoder.cpp 34]: (error) Memory leak: buffer
```

如果我们在代码中的局部变量声明了一个数组，但是还没有初始化或者没有赋值而直接访问了，cppcheck也可以直接检查出来，代码如下：

```
short* tmpBuffer = NULL;
if(condition) {
    tempBuffer = buffer;
}
tempBuffer[0] = 0;
```

cppcheck对这种场景会以错误的形式报告出来，代码如下：

```
[png_decoder.cpp:31]: (error) Possible null pointer dereference: tmpBuffer
```

如果代码中的局部变量没有初始化就进行使用，cppcheck也会报出

错误，代码如下：

```
int bufferCursor;
if(bufferCursor) {
    //Do Something
}
```

cppcheck对这种场景下的错误也会报出来，代码如下：

```
[png_decoder.cpp:22]: (error) Uninitialized variable: bufferCursor
```

如果对标准库中的一些API理解有问题，可能会写出如下代码：

```
const char* PngPicDecoder::getStatisticsData() {
    string buriedPointsStr;
    buriedPointsStr.clear();
    string str = "B_0.000";
    string comma = ",";
    buriedPointsStr += str;
    buriedPointsStr += comma;
    char temp[256];
    memset(temp,0,256);
    snprintf(temp, 256, "%.3f", 0.1358);
    str = temp;
    buriedPointsStr += str;
    return buriedPointsStr.c_str();
}
```

上述代码利用string的拼接功能，将很多统计变量按照一定的格式拼接起来，最终调用c_str函数返回C类型的字符串，但是会向外界暴露接口肯定是有问题的。cppcheck可以很好地检查出这种类型的错误，cppcheck给出的错误如下：

```
[png_decoder.cpp:56]: (error) Dangerous usage of c_str(). The value returned
```

上面列举的这些案例，都是我们在工作中不小心导致的问题，如果使用cppcheck工具在提交代码之前检查一遍，会对整个工作效率有很大提升。

2.Native代码运行中的检测

前面使用cppcheck对Native层的代码进行了静态检测。但是，仅仅靠对代码的静态检测是远远不够的，静态检测仅能解决开发人员的编码风格或者编码漏洞问题。真正在运行过程中出现的内存泄漏就需要靠动态的内存检测工具了。所以这里会介绍如何为Native层的代码进行动态检测内存泄漏。在Android平台上，比较好用又可靠的一种解决内存问题的办法：把Android NDK的C/C++代码移植到其他平台上并运行起来，然后使用该平台下的工具（如valgrind等非常强大的工具）进行检测。但是这种解决办法对于一个持续更新、快速迭代的产品来讲不是特别合适，因为需要不断地去写测试用例，持续集成需要花费很大精力。当然，还有另外一种解决问题的方法，那就是将其他平台的一些工具移植到Android上，比如valgrind就可以用在Android上，但由于效率太低，也不太方便。本书会将LeakTracer这一Linux平台上常用的memory leak检测工具移植到Android平台上，作为动态检测内存泄漏的工具。

LeakTracer分为两个主要部分：第一部分是对应用程序的检查，将LeakTracker集成到应用程序中，然后运行应用程序到Android平台，执行自己的测试Case，最终LeakTracer会将内存泄漏文件放到指定的路径下；第二部分是解析程序，解析程序将第一步生成的内存泄漏文件作为输入，并利用带有符号表（symbol file）的动态库生成肉眼可读的内存泄漏的调用堆栈信息。

LeakTracer的原理比较简单，第一部分是重写了new\delete\new[]\delete[]\malloc\free等操作符，等开发者使用new\new[]\malloc方法的时候，LeakTracer利用系统函数取出对应的调用堆栈的内存地址进行存储，等程序使用delete\delete[]\free方法的时候，再取出调用堆栈的内存地址和之前分配的调用堆栈的内存地址进行配对，如果没有配对成功，即为内存泄漏的地方，最终将所有内存泄漏的调用堆栈地址存储到内存泄漏文件中。

要完成第一部分的功能，需要将LeakTracer集成到App的Native层。读者在集成的时候，一定要使用代码仓库中的LeakTracer的源码，因为LeakTracer要针对Android平台做一些特殊的改动，其中有两个最重要的改动。

·第一个改动：在MemoryTrace.cpp的init_no_alloc_allowed方法中，使用dlsym加载动态链接库时，传入的常量由RTLD_NEXT改为

RTLD_DEFAULT。

·第二个改动：在MemoryTrace.hpp的storeAllocationStack方法中，将获取函数调用堆栈的方法由__builtin_frame_address和__builtin_return_address改为Android平台支持的_Unwind_Backtrace。

将整个LeakTracer的目录放入Native代码中，然后在Android.mk中添加以下代码：

```
LOCAL_C_INCLUDES += $(LOCAL_PATH)/libleaktracer/include
LOCAL_SRC_FILES += \
./libleaktracer/src/AllocationHandlers.cpp \
./libleaktracer/src/MemoryTrace.cpp
```

上述makefile文件会将对应的源码文件编译进来，然后就是集成阶段了。在JNI层的一个文件中，加入以下代码：

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    // leak tracer start recording
    leaktracer::MemoryTrace::GetInstance().startMonitoringAllThreads();
    return JNI_VERSION_1_4;
}
```

上述代码会在应用程序加载这个so包，之后调用LeakTracer的启动监测的方法，将整个LeakTracer启动起来，从而监测所有的内存分配与释放。在程序退出之前，增加以下代码的调用，就可将有内存泄漏的地方写入文件中：

```
JNIEXPORT void JNICALL Java_com_example_c_layer_NativeProcessor_destroy
    (JNIEnv * env, jobject obj) {
        const char *fPath = "/mnt/sdcard/mem_leak.log";
        leaktracer::MemoryTrace::GetInstance().writeLeaksToFile(fPath);
}
```

接下来模拟一个内存泄漏，该泄漏就在这个JNI层的代码调用的png_decoder类中，代码如下：

```
int PngPicDecoder::openFile() {
    short* buffer = new short[1024];
    return 1;
}
```

很显然，只要执行这段代码，就会产生内存泄漏。最后需要在文件Application.mk中加入以下这行代码：

```
APP_OPTIM := debug
```

上述这行代码要设置编译出来的so包是Debug类型，否则不能取出正确的调用信息。如果不想在Application.mk中设置这个参数，那么就需要在执行ndk-build的时候在后面加上参数NDK_DEBUG，即执行如下指令：

```
ndk-build NDK_DEBUG=1
```

如果开发环境是在Eclipse下，同时又依靠Eclipse配置的NDK来编译so包，那么就需要右击工程进入Properties，然后找到C/C++Build选项，将Builder中的复选框去掉，在Build command中输入上面的参数即可。

将整个App运行到手机上，然后点击自己的测试用例，待退出后，利用adb pull命令将SD卡路径下的mem_leak.log拿出来就是LeakTracer为生成的内存泄漏文件了。

接下来进入第二阶段，即解析内存泄漏文件成为肉眼可读的内存分配堆栈信息。使用代码仓库中的leakAnalysis.py脚本来分析堆栈信息，这个脚本需要的第一个输入是编译so包的时候在obj目录下带有symbol file的libMemoryLeak.so，第二个输入是上一步生成的内存泄漏文件，然后运行如下命令：

```
python leakAnalysis.py ./libMemoryLeak.so mem_leak.log
```

要想正确执行上述脚本，需要正确配置NDKROOT这个环境变量，因为脚本中需要用NDK里提供的工具进行解析符号表。脚本执行完毕之

后，会将内存分配的堆栈信息输出到屏幕上。如果读者想把结果放到文件中，可以直接执行如下命令，这个脚本会接受第三个参数为输出内存泄漏信息的文件：

```
python leakAnalysis.py ./libMemoryLeak.so mem_leak.log leak.txt
```

打开leak.txt，可以看到内存分配的堆栈信息以及一些主要信息，如下：

```
[
    {
        "leaked_bytes": 2048,
        "occurred_time": "2017-06-14 17:32:40",
        "stack_list": [
            "memory_leak/native_mem_case/png_decoder.cpp:18",
            "memory_leak/native_mem_case/NativeMemoryLeak.cpp:33"
        ],
        "unique_hash": "484FB1D0FFA75AA70F9FC51D7D453DCE"
    },
]
```

以上代码中包含的几个重要信息为泄漏的字节大小、泄漏的时间，以及内存分配的堆栈信息等，还有一个哈希值作为唯一标识。那在Python脚本中是如何实现解析内存分配堆栈信息的呢？实际上是使用前面讲解的NDK提供的addr2line工具将调用地址解析为代码的调用堆栈信息的，可以先打开第一步产生的mem_leak.log，如下：

```
# LeakTracer report diff_utc_mono=1497335780.019908
leak, time=96980.220532, stack=0xcf9c 0xd10c 0xd480 0xbe34 0xc654, size=2048,
data=..................................................
```

这个信息中记录了时间（time以秒为单位）、调用堆栈的地址信息、内存泄漏的大小等。其实数据（data）没有任何意义。脚本中需要做的就是将这些调用堆栈信息解析成为真正的肉眼可读的信息，如果使用前面讲解的addr2line工具，则代码如下：

```
NDK_GCC_TOOL_DIRECTORY=$NDKROOT/toolchains/arm-linux-androideabi-4.9/prebuilt
    /darwin-x86_64/bin/
$NDK_GCC_TOOL_DIRECTORY/arm-linux-androideabi-addr2line -e libMemoryLeak.so
```

```
0xcf9c 0xd10c 0xd480 0xbe34 0xc654
```

首先根据NDKROOT的路径构造出GCC工具存在的路径，然后使用addr2line工具将这些调用地址在带有Symbol file的so中找出真正的调用堆栈信息，上述命令执行结果如下：

```
./native_mem_case/libleaktracer/include/MemoryTrace.hpp:379
./native_mem_case/libleaktracer/include/MemoryTrace.hpp:429
./native_mem_case/libleaktracer/src/AllocationHandlers.cpp:40
./native_mem_case/png_decoder.cpp:18
./native_mem_case/NativeMemoryLeak.cpp:33
```

由于篇幅的关系，这里的路径都以相对路径标识，可以看到，几个内存地址就会解析出几个源文件的代码行数的调用关系。这里可以把libleaktracer的前缀都去掉，因为其中一些重载了操作符，也就是说，前面才是真正业务代码的泄漏堆栈信息，而在Python脚本中就把带有leaktracer前缀的调用堆栈去掉了。第二步的原理解释完毕了，所以最方便的解析方式还是使用上述的Python脚本。

在Android平台使用LeakTracer作为内存泄漏的检测工具是比较成熟的做法，读者可以参考代码仓库中的源码深入理解。

13.1.5 breakpad收集线上Crash

breakpad是Google提供的一套包含客户端和服务端的开源组件，用于收集客户端Native层的Crash。读者可以从Google Code上下载最新代码，也可以使用本书代码目录中的breakpad源码，其中Google Code上的代码地址为：

```
http://google-breakpad.googlecode.com/svn/trunk/
```

既然breakpad由客户端和服务器端两部分组件组成，下面就分为客户端和服务端两部分进行介绍。

（1）客户端部分

首先将breakpad客户端部分的代码集成入客户端的Native代码中；然后当Native层发生Crash行为的时候，breakpad会将Crash的信息以二进制形式写入minidump文件；最后客户端要将这个minidump文件上传到服务器上。

（2）服务端部分

首先将breakpad服务端工具编译出来，放入合适的目录下；然后利用dump_syms工具将客户端构建的带有symbol File的so包的symbol File导出到指定目录；最后把客户端上传的minidump文件和symbol File目录作为输入，利用minidump_stackwalk工具生成Crash现场。

上面描述了整个breakpad的工作流程，接下详细讲解整个流程的细节。首先来看如何将breakpad的代码集成入我们的客户端，一般集成第三方库到客户端中使用的方法都是将第三方库编译为静态库的方式放入prebuilt目录下，然后在Android.mk中进行引用，而这里使用另外一种方式，即将breakpad中的源码都放到一个Android.mk中进行编译，要编译breakpad，必须使用的ndk版本在r10c以上，因此需要更改Application.mk，如下：

```
APP_ABI := armeabi-v7a
```

```
APP_STL := gnustl_static
APP_CPPFLAGS := -std=gnu++11 -fexceptions -D__STDC_LIMIT_MACROS
NDK_TOOLCHAIN_VERSION = 4.8
APP_PLATFORM := android-9
```

在breakpad目录的android目录的google_breakpad目录下，将Android.mk这个makefile文件拿出来放到一个我们自己建立的libbreakpad目录下，然后将breakpad目录下的src目录也拷贝到我们新建的这个目录下，这样libbreakpad目录作为一个Module就建立好了。接下来需要将这个Module包含到整个构建脚本中。打开最外层的Android.mk，加入以下内容：

```
LOCAL_STATIC_LIBRARIES += libbreakpad_client
LOCAL_C_INCLUDES += \
    $(LOCAL_PATH)/libbreakpad/src \
    $(LOCAL_PATH)/libbreakpad/common/android/include
```

接着在入口的cpp文件中引入breakpad，引入头文件如下：

```
#include"client/linux/handler/exception_handler.h"
#include"client/linux/handler/minidump_descriptor.h"
```

然后，在加载so的回调方法中将breakpad的Crash拦截器启动起来：

```
static google_breakpad::ExceptionHandler *handler = NULL;
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved){
    const char* nativeLogPath = "/mnt/sdcard/appname/tombstones";
    google_breakpad::MinidumpDescriptor descriptor(nativeLogPath);
    handler = new google_breakpad::ExceptionHandler(descriptor,
        NULL, NULL, NULL, true, -1);
    return JNI_VERSION_1_6;
}
```

之后执行ndk-build，并构建so包。注意，这个ndk-build必须是r10c版本以上，最终构建so包成功之后，可以在libs目录下找到so包，而在obj目录下可以找到名称相同的另外一个so包，并且这个so包比libs目录下的so包要大不少，这是为什么呢？先来看图13-12。

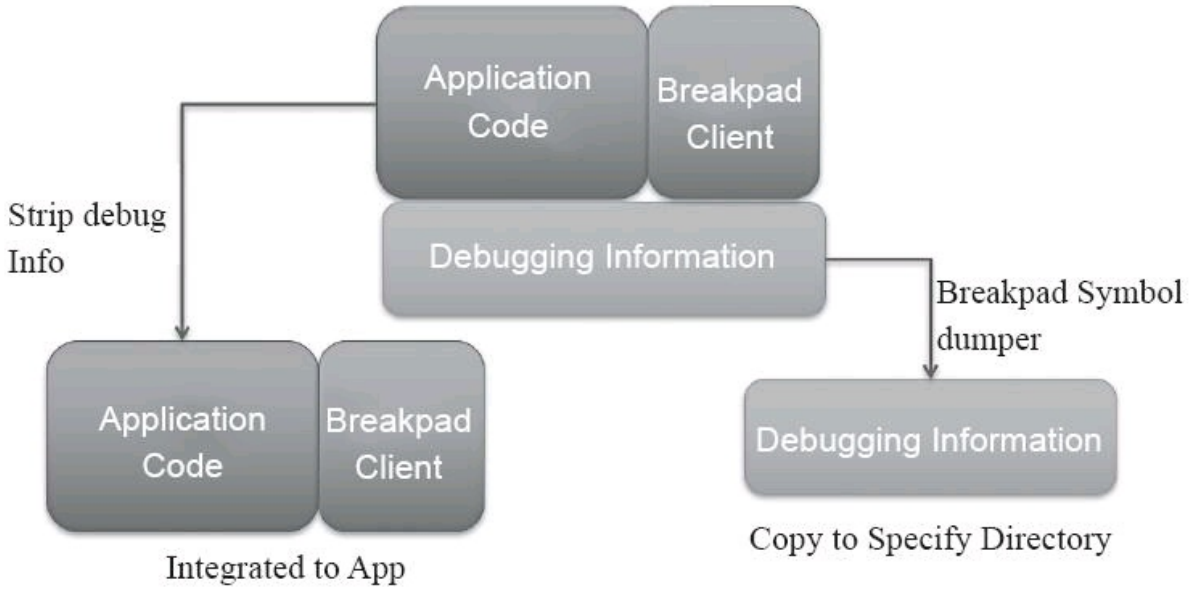


图　13-12

如图13-12所示，obj目录下的so包其实是图的中间部分，包括应用的中间文件、breakpad的中间文件以及代码调试信息，而libs目录下的so包是将所有的代码调试信息消除掉（strip）的包，是最终要集成到App中分发给用户使用的。使用breakpad服务器端编译出来的dump_syms工具将obj目录下的so包中的代码调试信息部分解析出来，并放入合适的路径下。注意，这里集成到App里的so必须要和服务器端解析symbol File的调试信息对应起来，否则解析不出正确的方法调用堆栈。可能有读者会想，这里的dump_syms是如何编译出来的呢？其实很简单，首先找一台Linux机器，在breakpad目录下执行以下命令：

```
./configure && make
```

之后在breakpad的src/tools/linux/目录下就可以看到编译出来的dump_syms工具了，同时在src/processor/目录下可以看到即将用到的minidump_stackwalk工具。接下来看看当用户手中的客户端发生了Native层的崩溃时，客户端的行为是什么，如图13-13所示。

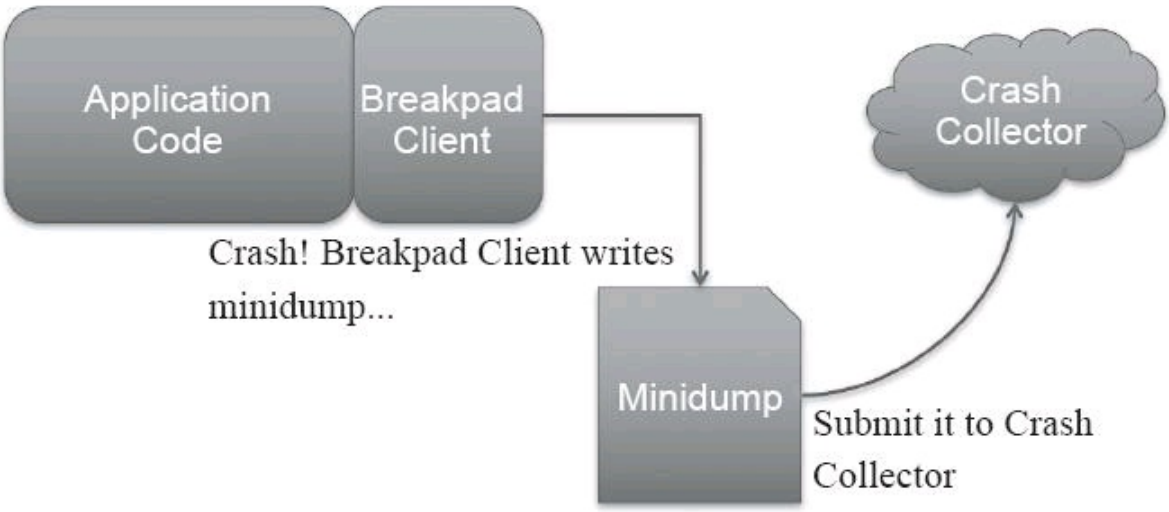


图　13-13

在图13-13中，当客户端的Native层（Application Code）发生崩溃的时候，配置的breakpad会将崩溃信息拦截下来，并生成一个二进制mini_dump文件，最后客户端将这个文件上传到专门收集与处理崩溃信息的服务器上，即图中的Crash Collector。那接下来服务器会如何做呢？如图13-14所示。

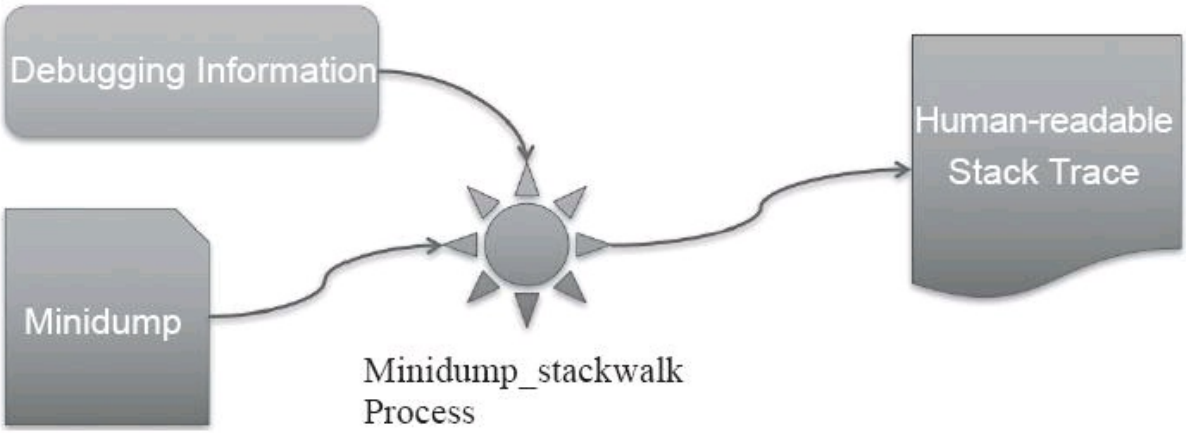


图　13-14

在图13-14中，服务器端会将第一步产生的代码调试信息和第二步客户端上传的mini_dump二进制文件作为minidump_stackwalk工具的输入，最终输出肉眼可读的崩溃现场。

至此，如何利用breakpad收集Native层的崩溃就介绍完毕了，使用工具的同时，最好也要理解清楚原理，这样可以在遇到其他类似问题的时候触类旁通。当然，使用第三方收集Native层崩溃的SDK也可以，比如CrashLytics等，但是理解了原理之后，会发现它们的本质其实都是一样的。使用第三方收集Native层的崩溃当然有好处也有坏处，读者可以根据自己的应用场景来决定如何收集Native层的崩溃。

13.2 iOS使用Instruments诊断应用

Instruments是一个很灵活的、强大的工具，是性能分析、动态跟踪和分析OS X以及iOS代码的测试工具。使用它可以极为方便地收集一个或多个系统进程的性能和行为的数据，并能及时跟随时间产生的数据，而且可以检查所收集的数据，还可以广泛收集不同类型的数据。此外，还可以追踪程序运行的过程，这样Instruments就可以帮助我们了解用户的应用程序和操作系统的行为。本节以视频播放器项目为例，使用Instruments提供的工具分别从CPU占用、内存分配以及内存泄漏等方面进行分析，最后会在Instruments的帮助下，比较播放器中的解码器模块使用硬件解码器和软件解码器的各项性能指标。

13.2.1 Debug Navigator

在介绍Instruments工具之前，先看Xcode

中左侧的选项，其中有一个Show the Debug navigator，选中它，会显示当前调试的这个App的CPU占用情况、内存占用情况（Memory）、电量消耗情况（Energey Impact）等，如图13-15所示。

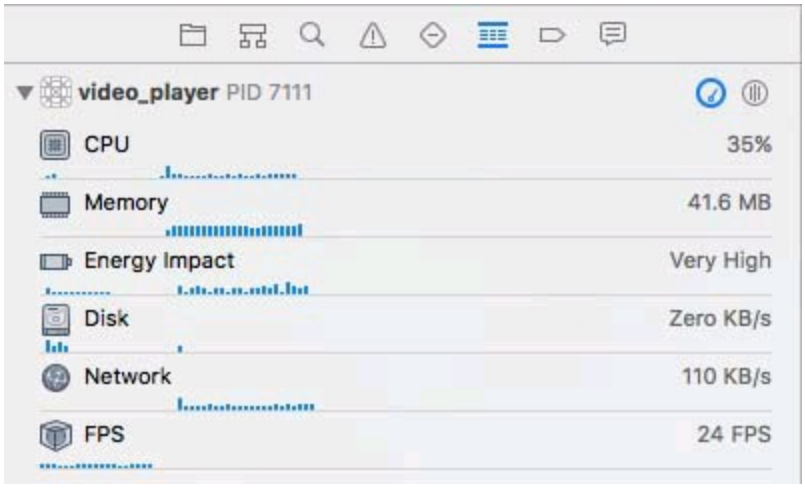


图 13-15

点开每个维度都会有具体的实时数据和一个随着时间变化的动态时间线图。下面看看软件解码和硬件解码在CPU维度的占用情况，如图13-16所示。

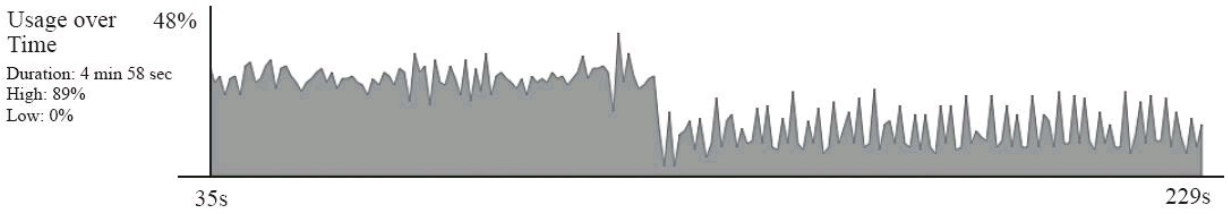


图 13-16

在图13-16中，中间的CPU占用突然下降就是由软件解码器切换到硬件解码器的时间点，前半部分是使用软件解码器的CPU占用变化图，后半部分是使用硬件解码器的CPU占用变化图。使用软件解码器的时候，CPU的占用为30%左右，切换到硬件解码器之后CPU的占用下降到

了20%左右。笔者用来测试的视频分辨率为640×360，fps为24，测试设备是iPhone 6S。大家不要小看CPU上这10%的占用，如果在一个直播场景下，还会有动画、聊天等CPU消耗比较大的线程，而这10%的CPU就很有可能影响到用户端的流畅度了。一旦观看的视频分辨率更高、fps更大，CPU的对比会更加明显。细心的读者可能会发现，CPU的占用图一直是波形的结构，这是为什么呢？大家还记得AVSync模块中解码线程的运行策略吗？当队列中的数据到达MaxDuration的时候，解码线程暂停解码，而消费者线程会消费队列中的数据。当队列中的数据小于MinDuration的时候，解码线程会继续解码，而解码线程所耗费的CPU是巨大的，所以当解码线程运行的时候，CPU的消耗就达到了波峰的位置；当解码线程暂停的时候，CPU的消耗则达到了波谷的位置，若将时间作为横轴来看CPU占用的变化情况，就形成了如图13-16所示的这种波形图。下面对比硬件解码和软件解码在内存中的占用情况，这个维度的变化如图13-17所示。

图　13-17

在图13-17中，内存突然从50MB以上下降到10MB左右就是从软件解码器切换到硬件解码器，前半部分是使用软件解码器时占用的内存，后半部分是使用硬件解码器时占用的内存。为什么内存的占用会突然下降了这么多？原因是构造的VideoFrame结构体中不再使用内存中存储的YUV数据，而使用iOS的“主存储”中的CVImageBufferRef。切换为硬件解码器后，内存占用的下降对于整个App的流畅运行也是一件好的事情，因为内存可以用来存储更多的图片缓存等数据。

在Debug Navigator中还有一些其他指标，比如电量的消耗、硬盘I/O的占用、网络I/O的占用以及界面刷新频率（FPS）等。若要粗略地了解一个App的运行情况，可以从这里快速得到结果。但是，如果想发

现更加细节的问题或者要解决某个具体的问题，那么就需要使用Instruments工具来定位问题。接下来继续使用Instruments来分析我们的视频播放器项目。

13.2.2 Time Profiler

Time Profiler工具是用来分析方法的执行时间的，它可以找出哪些方法以及哪些线程执行的时间比较长，一般用作性能分析的工具。使用这个工具时，有一点需要注意，即测试的App一定要运行在真机设备上，因为模拟器运行在Mac电脑上，而电脑的CPU运行速度要比iOS设备的快，相反，Mac上的GPU和iOS设备的完全不一样，模拟器不得已要在软件层面（CPU）模拟设备的GPU，这意味着GPU相关的操作在模拟器上运行得更慢，典型的场景如视频播放器项目、视频录制项目，都与CAEAGLayer的绘制相关。

在Xcode中的左上方Run的地方长击，然后在弹出的菜单中选择Profile，Xcode会编译程序并启动Instruments，然后在Instruments界面中选择Time Profiler工具。进入Time Profiler界面之后点击左上角的Recording来启动应用。待应用启动之后，在Call Tree中的工具已经默认选择了Separate by Thread选项，这个选项的意思是会按照各个线程来显示CPU的占用情况。我们可以选择Invert Call Tree选项，这个选项的意义是可以直接看到方法调用路径最深的方法CPU的占用情况；我们还会选择Hide System Libraries选项，这个选项的意义是隐藏掉系统代码的耗时，因为一般情况下我们仅关心自己的业务代码的CPU耗时，所以勾选这个选项非常有利于分析业务代码的CPU消耗。勾选之后并运行，可看到如图13-18所示的界面。

Weight		Self Weight	Symbol Name
12.96 s	100.0%	0 s	▼video_player (7331)
3.26 s	25.1%	0 s	▶frame_worker_thread 0x44aede
2.91 s	22.4%	0 s	▶frame_worker_thread 0x44aedd
2.83 s	21.8%	0 s	▶frame_worker_thread 0x44aedf
1.38 s	10.6%	0 s	▶-[AVSynchronizer decodeFrames] 0x44aee0
641.00 ms	4.9%	0 s	▶_dispatch_worker_thread3 0x44aed3
606.00 ms	4.6%	0 s	▶_dispatch_worker_thread3 0x44aed2
573.00 ms	4.4%	0 s	▶_dispatch_worker_thread3 0x44aed0
549.00 ms	4.2%	0 s	▶AURemoteIO::IOThread::Run 0x44aef4
170.00 ms	1.3%	0 s	▶Main Thread 0x44aec2
35.00 ms	0.2%	0 s	▶_dispatch_worker_thread3 0x44aecf
12.00 ms	0.0%	0 s	▶-[AVSynchronizer decodeFramesWithDuration:] 0x44aee1
3.00 ms	0.0%	0 s	▶GenericRunLoopThread::Entry 0x44aee2

图 13-18

图13-18是使用软件解码的视频播放器播放80s的视频。各个线程的

CPU占用情况，前三个可能看起来比较奇怪，因为我们并没有启动这样的线程，而且还占用了这么多的CPU。其实这是FFmpeg为解码开辟的三个异步解码线程，并且这三个线程占用了最多的CPU时间片。接着来看AVSync模块的decodeFrames方法，它就是我们自己开启的解码线程，相较而言，这个占用的CPU时间片也非常多，所以可以看到整个解码共占用了整个App中的82.5%的CPU时间片。至于接下来的两个dispatch worker thread，是VideoOutput模块中的渲染线程，由于渲染线程使用的是NSOperationQueue，所以也可以看到这个线程模型的内部实现开启了多个worker来轮询Queue中的Operation。最后是AURemoteIO这个音频播放的线程所占用的CPU时间片，由于AudioOutput模块使用的是AUGraph，而AUGraph是靠RemoteIO Unit来驱动的，所以展示在这里的线程名字就是AURemoteIO：：IOThread：Run。后面还有其他线程，比如MainThread等，占用CPU的时间片就比较少了。基于此，可以分析出，时间的消耗大部分都花费在了解码线程上，那么如何优化呢？使用硬件解码代替软件解码，可以减少对解码线程的CPU时间片的分配，下面来看将视频播放器中的解码模块替换为硬件解码方案的Time Profiler图，如图13-19所示。

图13-19展示的是将解码器模块的实现由软件解码器替换为硬件解码器播放80s后各个线程所占用CPU时间片的情况。最明显的是整个App被分配到的时间片只有6.18s，比使用软件解码器少了一半的时间，并且解码线程所占用的CPU时间片也大大减少了。同时也完全可以印证了13.2.1节中CPU的占用从30%下降到20%的原理。这只是通过Time Profiler检查与验证这种大模块的替换与优化，对于一些小的细节性优化，比如普通的Queue更改为Blocking类型的Queue，也可以让CPU占用降低。

Weight		Self Weight	Symbol Name
6.18 s	100.0%	0 s	▼video_player (7332)
1.70 s	27.4%	0 s	▶-[AVSynchronizer decodeFrames] 0x44b030
855.00 ms	13.8%	0 s	▶_dispatch_worker_thread3 0x44b003
819.00 ms	13.2%	0 s	▶_dispatch_kevent_worker_thread 0x44b002
687.00 ms	11.1%	0 s	▶_dispatch_kevent_worker_thread 0x44b000
652.00 ms	10.5%	0 s	▶_dispatch_worker_thread3 0x44b001
649.00 ms	10.5%	0 s	▶AURemoteIO::IOThread::Run 0x44b042
643.00 ms	10.4%	0 s	▶_dispatch_worker_thread3 0x44afff
166.00 ms	2.6%	0 s	▶Main Thread 0x44afee
6.00 ms	0.0%	0 s	▶-[AVSynchronizer decodeFramesWithDuration:] 0x44b031
6.00 ms	0.0%	0 s	▶GenericRunLoopThread::Entry 0x44b032

图　13-19

Time Profiler工具可以用来检测系统的性能瓶颈，根据线程以及方法的耗时情况，开发人员可以合理优化自己的App，提升用户体验。但是一定要注意的是，优化代码是在代码运行正确的基础之上的，所以一定要对修改代码的影响部分进行充分测试。

13.2.3 Allocations

管理内存是App开发中最重要的一个方面，在音视频的开发中，内存管理更是非常重要。相较于电脑，移动设备内存是更紧缺的资源，在iOS的开发中，开发者通常使用Instruments里的Allocations工具来定位和找出减少内存使用的方式，比如可能通过改进程序架构和算法来实现。但是，再好的App设计都会被不同的内存问题困扰。

打开Allocations界面，运行应用程序，然后选择Generrations快照工具来完成本节的学习。在进入播放界面之前，先来做一个快照（点击Mark Generation按钮），然后进入播放界面播放视频，在播放视频过程中再做一个快照，最后退出播放界面，并做一个快照，如图13-20所示。

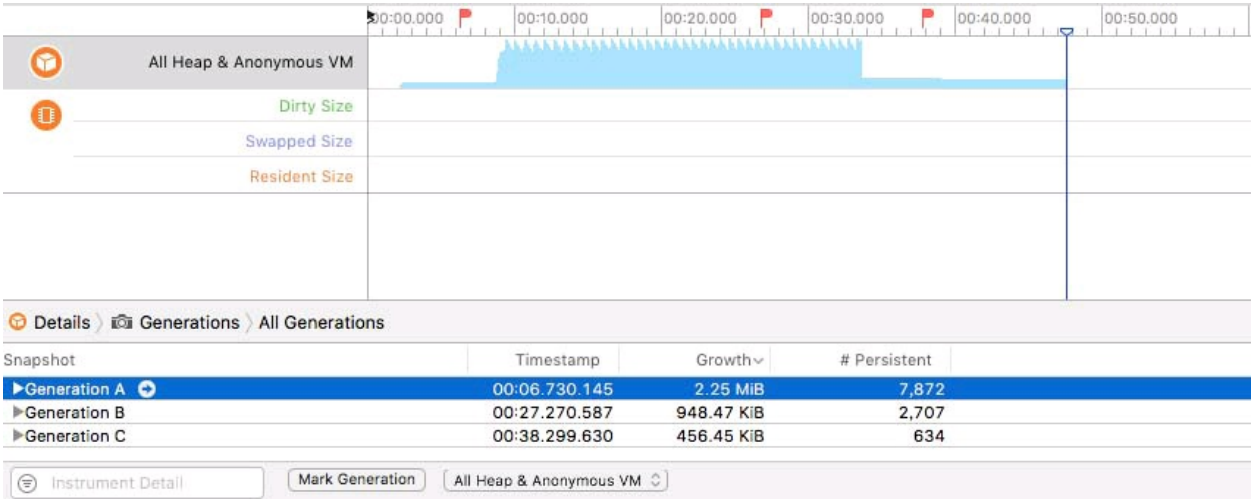


图 13-20

在图13-20中，从下方区域可以看到生成的三个快照，在每个快照的Growth这一列，可以点击每一行使其展开，这样就可以看到这个快照增长的内存有哪些，对比快照A和快照C可以发现有没有内存泄漏，因为A和C恰好是进入播放页面和退出播放页面，如果在Growth这一列中看到有自己分配的对象还没有被释放，那么就可能是内存泄漏，可以进一步分析。

当然，内存占用较大也不一定是坏事，这就是所谓的空间和时间的置换问题，对于每一个App，应该根据自己的场景来合理使用内存。像

拉流播放器项目中，如果MaxBufferDuration设置的大一些，就会使得整个程序占用内存大一些，不会因为网络的抖动产生卡顿现象，但是，如果当用户看了几秒钟就退出了，那么就浪费了用户的网络带宽，所以对于这种情况，就应该根据自己的产品场景来设置视频缓存长度的大小。而在普通的iOS开发中常常会遇到类似问题，比如图片缓存的场景，如果缓存设置太小，就有可能使得图片重新下载率比较高；如果设置过大，重新下载率降低，但是占用内存又升高，所以也应该根据产品场景来决定如何设置图片缓存的大小。总之，App占用内存不是越小越好，而是应合理为好，读者可以回顾一下自己产品对于内存使用的策略，是否有该调整的地方。

13.2.4 Leaks

内存泄漏（Memory Leak）是指程序在申请内存之后，无法释放已申请的内存空间，一次内存泄漏危害可以忽略，但内存泄漏堆积后果是很严重的，最终会造成内存溢出，致使整个程序崩溃。在音视频开发过程中，一旦音频帧数据或者视频帧数据有泄漏，会使得整个内存飙升严重。内存泄漏一般分为以下四种情况。

（1）常发性内存泄漏

发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

（2）偶发性内存泄漏

发生内存泄漏的代码只有在某些特定环境或操作下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

（3）一次性内存泄漏

发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，会导致有一块且仅一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。

（4）隐式内存泄漏

程序在运行过程中不停地分配内存，但是直到结束的时候才释放内存。严格来说，这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是，对于一个服务器程序，需要运行几天、几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害则非常大，

因为较之于常发性内存泄漏和偶发性内存泄漏，它更难被检测到。

下面介绍Instruments里的Leaked的用法。运行视频播放器项目，然后看到如图13-21所示的界面。

图　13-21

在图13-21中，上面那一行是内存和匿名虚拟内存的占用情况，下面这一行是泄漏的检查，工具默认每10s检查一次，若这里看到的都是对号，则代表没有任何内存泄漏，证明这个App运行良好。但是，为了演示如何利用这个工具来分析音视频中的内存泄漏，就需要模拟一个内存泄漏，然后使用工具进行检测。更改VideoDecoder.m中的closeAudioStream方法，注释掉其中释放的AVCodecContext操作，代码如下：

```
- (void) closeAudioStream {
    //1:释放重采样相关的资源
    //2:释放AudioFrame
    //if (_audioCodecCtx) {
        //avcodec_close(_audioCodecCtx);
        //_audioCodecCtx = NULL;
    //}
}
```

closeAudioStream方法的第三步是释放掉音频流的Codec上下文，现在注释掉这四行代码，然后重新进行Profile，并进入检查内存泄漏的界面，运行程序，点击观看视频按钮，观看20s后退出视频播放界面，可以看到内存泄漏，如图13-22所示。

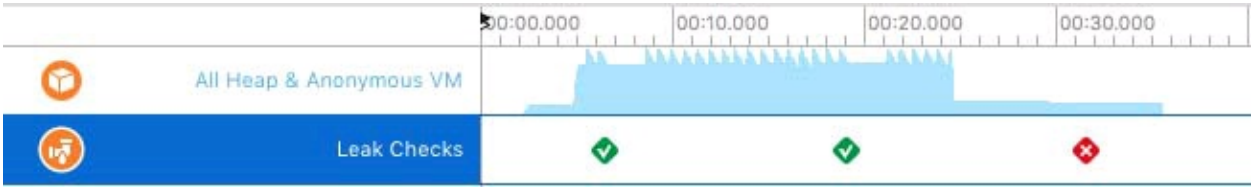
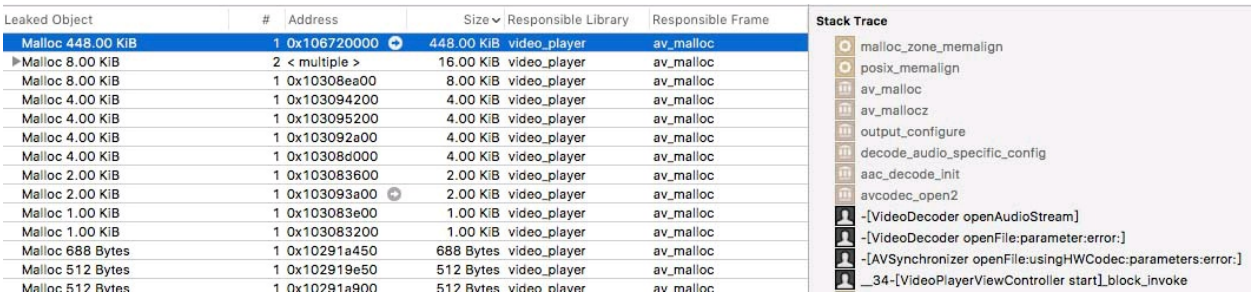


图　13-22

待退出播放器界面之后，发生了内存泄漏（出现了错号），点击这个按钮，会自动将泄漏的调用堆栈显示出来，如图13-23所示。

点击每个泄漏的内存，在右侧都会显示它的调用堆栈。从图13-23中可以看到，很多内存块的泄漏都与VideoDecoder类中的openAudioStream方法有关，若点击这个方法，会直接跳转到代码处，如下所示：



Leaked Object	#	Address	Size	Responsible Library	Responsible Frame
Malloc 448.00 KiB	1	0x106720000	448.00 KiB	video_player	av_malloc
Malloc 8.00 KiB	2	< multiple >	16.00 KiB	video_player	av_malloc
Malloc 8.00 KiB	1	0x10308ea00	8.00 KiB	video_player	av_malloc
Malloc 4.00 KiB	1	0x103094200	4.00 KiB	video_player	av_malloc
Malloc 4.00 KiB	1	0x103095200	4.00 KiB	video_player	av_malloc
Malloc 4.00 KiB	1	0x103092a00	4.00 KiB	video_player	av_malloc
Malloc 4.00 KiB	1	0x10308d000	4.00 KiB	video_player	av_malloc
Malloc 2.00 KiB	1	0x103083600	2.00 KiB	video_player	av_malloc
Malloc 2.00 KiB	1	0x103093a00	2.00 KiB	video_player	av_malloc
Malloc 1.00 KiB	1	0x103083e00	1.00 KiB	video_player	av_malloc
Malloc 1.00 KiB	1	0x103083200	1.00 KiB	video_player	av_malloc
Malloc 688 Bytes	1	0x10291a450	688 Bytes	video_player	av_malloc
Malloc 512 Bytes	1	0x102919e50	512 Bytes	video_player	av_malloc
Malloc 512 Bytes	1	0x10291a900	512 Bytes	video_player	av_malloc

Stack Trace

malloc_zone_memalign
posix_memalign
av_malloc
av_mallocz
output_configure
decode_audio_specific_config
aac_decode_init
avcodec_open2
-[VideoDecoder openAudioStream]
-[VideoDecoder openFile:parameter:error:]
-[AVSynchronizer openFile:usingHWCodec:parameters:error:]
__34-[VideoPlayerViewController start]_block_invoke

图　13-23

```
int openCodecErrCode = 0;
if ((openCodecErrCode = avcodec_open2(codecCtx, codec, NULL)) < 0){
    NSLog(@"Open Audio Codec Failed %s", av_err2str(openCodecErrCode));
    return NO;
}
```

从上述代码中可以看到，是因为这里打开了音频Codec的上下文，而最后我们没有释放掉而导致的内存泄漏。试着将代码恢复回来，再进行内存泄漏的检查，就没有问题了。对于内存泄漏，原因有很多种，笔者也只是以FFmpeg的API为例进行了讲解。由于ARC的内存释放机制使用的是自动引用计数的方式，所以，一旦存在循环引用，就肯定会导致内存泄漏。一般情况下，将其中的一个对象中的变量设为weak，不让它出现在保留周期中即可解决问题。

13.3　本章小结

本章主要介绍了日常工作中用到的工具，首先介绍了Android平台下常用的工具，重点介绍了ADB工具的使用以及Java层和Native层内存泄漏的检测，当然还有NDK工具的使用，读者可以利用NDK的工具方便地排查一些开发中的问题。其次，介绍了iOS平台下强大的Instruments工具，分别从内存、CPU负载等方面进行了介绍与分析。其实某一些工具根据工作场景的不同，本章也并没有介绍得很详细，比如ADB中的pm与am，但是读者掌握了本章的内容之后，可以很快地使用更多的工具，不论是开发人员还是测试人员，在工作中利用好这些工具都可以提升自己的工作效率，希望大家深入学习，好好理解并运用到工作中。

附录A　通过Ne10的交叉编译输入理解ndk-build

A.1　Ne10简介

目前大部分智能手机已经配备了高清摄像头、高保真麦克风，由此而带来的声音类应用与图像类应用越来越普遍，而本书所讲解的就是基于移动平台的音视频类应用的开发。但是，在处理这些音视频数据的时候，单纯依靠现有CPU的计算能力是远远不够的，所以对于一个音视频应用来讲，提高性能是永无止境的。在视频处理方面，多会使用OpenGL ES技术，利用显卡的并行计算能力来提高处理图像或视频的速度，但是在音频处理方面却很少有比较成熟的技术供开发者使用。ARM NEON技术采用SIMD（单指令，多数据）体系结构，可以有效提升多媒体和信号处理应用程序的性能，从而增强用户体验。同时，NEON技术与ARM处理器紧密结合，提供单指令流和内存的统一视图，从而能够提供一个具有更简单工具流的开发平台。由于目前市面上的Android设备和iOS设备使用的都是ARM架构的芯片，所以在音视频处理过程中使用ARM提供的NEON指令集来加速运算是一件非常有意义的事情。但是，开发者要从头开始编写很多基础的Math功能以及信号处理的FFT、FIR、IIR等滤波器，这基本上是不现实的事情，所以Ne10应运而生。

Ne10是由ARM主导开发的一个开源软件库。该库旨在提供一系列通用的、基于ARM NEON架构并且经过深度优化的函数集合。通过调用这些库函数，可以让软件开发人员免于编写重复的底层汇编代码，同时也能充分利用ARM NEON SIMD指令的并行运算能力。Ne10的主要目录结构包括：doc（文档）、inc（头文件目录）、samples（示例目录）、android（安卓平台下的动态库）、common（基础库）和modules（模块目录）。部分目录会在后续章节中逐一进行介绍，这里我们来看一下modules目录下的结构。在modules目录下有三个目录，功能如下：

·math数学模块：主要包含矢量/矩阵数学运算。

·dsp数字信号处理模块：主要包含FFT快速傅里叶变换，以及部分FIR/IIR滤波函数。

·imgproc图像处理模块：主要包含图像缩放、旋转等后处理函数。

A.2 编译和运行官方Demo

A.2.1 安装cmake

Ne10的编译是基于cmake编译的。首先，开发者需要在开发环境中安装cmake，注意一定要安装命令行方式的cmake，不要安装图形化界面的cmake。cmake的全称是cross platform make，从其全称上可以看出它是一个跨平台的编译工具。由于各个平台的编译环境与构建语法不同，所以开发者开发一个跨平台的公共库或者软件是一件很困难的事情。而cmake可以使用统一的语法编译出多个平台的makefile文件或project文件，再执行make命令就可以编译出目标包。如何安装cmake呢？在Mac OS X系统下，开发者直接使用brew进行安装cmake即可：

```
brew install cmake
```

A.2.2 编译NE10

安装好cmake之后，就可以编译Ne10这个库了。首先进入Ne10的根目录，然后进入doc目录，打开building.md文件，在该文件中可以找到对于在Android平台下编译NE10的帮助。编译步骤如下：

1）建立build目录，并进入这个目录。

```
mkdir build && cd build
```

2）将NDK目录配置到ANDROID_NDK变量中。

```
export ANDROID_NDK=/absolute/path/of/android-ndk
```

3）指定编译的目标平台是armv7，当然，也可以指定为aarch64。

```
export NE10_ANDROID_TARGET_ARCH=armv7 #Can Also be "aarch64"
```

4）指定cmake的配置文件路径，使用cmake生成对应平台的makefile文件。

```
cmake DCMAKE_TOOLCHAIN_FILE=../android/android_config.cmake ..
```

5）执行make指令，编译对应平台下的包。

```
make
```

make命令运行完毕后，如果没有出现错误日志，就代表Ne10编译成功。

A.2.3 运行官方Demo

开发者可以进入build目录下，该目录就是编译的目标目录。该目录下有一个android目录，这个android目录下有一个NE10Demo，它就是运行在Android设备上的应用程序。开发者可以将NE10Demo中的jni目录下的libNE10_test_demo.so取出，放到工程下的libs中的armeabi-v7a目录下，然后将工程导入IDE中，并运行。

官方的这个Demo工程是使用WebView来展示界面的，从jni里的源码中可以看到，Ne10会利用自己的单元测试用例来测试Ne10的性能，并与CPU的运行的速度做对比。我们可以在Java层代码中打一个断点来看底层返回的string类型的结果，默认的测试函数是abs这个函数，测试结果如下：

```
{ "name" : "test_abs_case0 1", "time_c" : 11441, "time_neon" : 1448 },?
{ "name" : "test_abs_case0 2", "time_c" : 8289, "time_neon" : 1858 },
{ "name" : "test_abs_case0 3", "time_c" : 11193, "time_neon" : 2781 },
{ "name" : "test_abs_case0 4", "time_c" : 13575, "time_neon" : 2229 }
```

从以上代码中可以看到，abs这个函数利用neon加速之后的结果是纯使用CPU计算速度的5倍以上。如果想得到更多其他函数的测试结果，读者可以自己去改动jni层的测试用例调用，然后使用cmake以及make指令编译出so包，运行就可以得到结果了。

A.3 通过Ne10的编译来看ndk-build的执行过程

A.3.1 如何构建基于Ne10的应用

Ne10的官方Demo运行成功之后，如果开发者想开发基于Ne10的应用，那么该如何做呢？依据之前的开发经验，最简单的方式就是获取include文件与静态库文件，然后放入Native代码中，分别在编译和链接阶段找到这两个文件就可。具体做法如下。

先进入build目录中，找到modules目录下的libNE10.a，这就是我们想要的静态库文件。那头文件在什么位置呢？就在主目录下的inc（include）目录中，这个目录里就有我们编译阶段需要的头文件。按照之前的目录构建方式，我们将头文件和静态库文件放入Native代码中，再开始开发基于Ne10的应用即可。

A.3.2 深入理解Ne10的交叉编译

笔者曾在GitHub上就Ne10在Android平台的编译和使用与Ne10的作者之一Joe Savage有过比较多的交流。Joe Savage是一个非常nice的人，通过与他的多次讨论，笔者发现了Ne10的更多内部细节。按照A.2.2小节中的步骤将Ne10编译成功之后，进入build目录，可以看到该目录下有三个比较重要的目录，分别是android目录、samples目录和modules目录。

1.android目录

android目录下是编译好的安卓平台下的动态so库，以及编译过程中由cmake产生的中间文件。通过A.2.3节中的描述，将动态库libNe10_test_demo.so放入官方Demo工程中并运行，可以得到对比测试的结果。这个动态库的源码实际上是Ne10根目录中的android目录下jni中的代码，而jni中的代码使用的是Ne10根目录下的test目录下的测试case。大家可以想一下，这个编译动态库的过程其实与绝大多数安卓工

程编译动态库的过程是不一样的，因为大部分开发者正常编译一个安卓工程的动态库使用的是ndk-build命令。

但是Ne10的构建方式不是使用ndk-build这个脚本，而是使用了更加通用的cmake。cmake是一个跨平台的构建工具，用自己的描述语言或者语法可以生成对应平台的Makefile（make脚本）文件。开发者不建议在安卓平台的每个项目都这样做，因为Ne10项目本身不仅是安卓平台的项目，而且是所有ARM架构的系统都可以使用的开源库。所以不同的应用场景选择不同的解决方案，而选择cmake是NE10的最佳解决方案。而对于Android工程的Native代码的构建场景，使用ndk-build才是最佳解决方案。如果开发者想更改NE10Demo的测试程序，再自己集成Ne10之后的性能测试或者正确性测试（当然，Ne10的作者已经做过了这些测试，但是，由于构建方式的不同，以及开发者的工程可能会有比较复杂的业务逻辑，所以自己做测试在所难免），开发者可以修改Ne10项目目录中的android→NE10Demo→jni目录下的源码文件NE10_test_demo.c，然后到build目录下执行make命令，最后把so文件拷贝到对应目录中编译并运行安卓程序。当然，如果增加jni函数也要相应地在java文件中增加native方法的声明。以上介绍完了andorid目录下的结构，这个目录主要是针对Android工程编译的动态so库，开发者可以更改对应的源码文件，然后使用so库进行测试。

ndk-build命令

ndk-build命令是一个脚本，在指定的NDK-ROOT下面。ndk-build这个脚本实际上会检查工程当前Application.mk文件里的配置选项，包括交叉编译的gcc版本、APP-CFLAGS、APP-CPPFLAGS、是否开启异常及Rtti等参数。当然，如果没有Application.mk这个配置文件，ndk-build命令会使用一系列默认的配置。首先，要确定使用的gcc版本以及要编译的目标平台，例如，我们使用gcc4.8编译armv7-a平台上的包。然后，ndk-build才会读取Android.mk里的配置，包括CFLAGS、LDFLAGS、源码文件以及include的预编译mk。最后，进行编译和链接。无论是Application.mk还是Android.mk，配置文件中指定的CFLAGS、LDFLAGS这些参数的默认值都在NDK目录下的哪一个文件中指定呢？它们都存在于NDK→toolchains目录下对应的gcc版本目录中的setup.mk这个文件中，例如，当使用的gcc版本为4.9的时候，setup.mk所在的目录为：$NDK_ROOT/build/core/toolchains/arm-linux-androideabi-4.9或者$NDK_ROOT/toolchains/arm-linux-androideabi-4.9。

由于不同的NDK版本所在的位置不同，当ndk-build脚本被执行的时候，可以根据默认配置以及Android.mk里的配置构建出对应的动态库或者静态库或者二进制的可执行程序。

2.samples目录

从samples目录下可以看到一个二进制的可执行文件NE10_samples_static，我们可以将这个文件放入Android手机上，以命令行工具的方式运行它。首先找一台root了的手机（只有获得root权限，才能方便直接登入系统去执行这个命令），再利用adb push命令将这个二进制文件推送到sdcard上：

```
adb push NE10_samples_static /mnt/sdcard/NE10_samples_static
```

接着使用adb shell命令登入这台安卓设备：

```
adb shell
```

然后将上一步从电脑推入的二进制程序拷贝到/data/目录下，并增加执行权限：

```
su
cp /mnt/sdcard/NE10_samples_static /data/
cd /data/
chmod 777 NE10_samples_static
```

最后运行以下这个二进制命令：

```
./NE10_test_static
```

如果没有错误，应该可以看到官方二进制Demo的测试结果。开发者可以修改对应的源码文件，然后执行make命令，生成新的二进制文件，将新的二进制文件放入Android设备中，再做一些快速测试。那如何修改二进制命令对应的源码文件呢？首先切换到Ne10的根目录下，然后进入samples目录，就可以修改对应的主文件或者单元测试的文件。

修改完毕后，再回到build目录执行make命令，然后找到最新的二进制可执行程序推送到安卓系统内部，再次执行就可以看到修改后的效果了。

在平时的开发过程中，开发者并不是总需要安卓的开发环境（IDE）才能测试Native层的代码，也可以编译二进制的可执行文件来做快速测试。秘诀在于Android.mk配置文件的最后一行，如果包含的是shared library，就是构建动态库；而包含的是static library，就是构建静态库。但是，如果包含的是execute library，就是构建二进制可执行文件。而包含任何一个变量，都是一个预定义存在于NDK-ROOT目录下的一个文件，路径为：

```
$NDK_ROOT/build/core
```

因此如果想用ndk-build编译一个二进制程序，就在Android.mk文件的最后一行包含execute library就可。

3.modules目录

modules目录中有一个静态库文件libNE10.a，该文件就是编译好的armv7-a平台下的静态库。开发者可以将这个静态库作为一个prebuilt的静态库链接到自己的程序中，然后直接执行ndk-build命令，就可以编译出动态so库了。但是这里有可能会出现一个编译失败的问题，错误如下：

```
ld: error: jni/prebuilt/libNE10.a(NE10_fft_generic_float32.c.o) uses
    VFP register arguments, output does not
```

根本原因在于Ne10默认的编译选项里开启了硬浮点运算，即

```
-mfloat-abi=hard -mfpu=vfp3
```

但是，ndk-build这个脚本使用的是gcc版本对应的setup.mk文件里的编译选项和链接选项，而setup.mk文件中默认的选项使用的是如下指令：

```
-mfloat-abi=softfp -mfpu=vfpv3-d16
```

因此，使用默认的编译选项是不对的，所以需要在Application.mk里重新指定编译选项和链接选项来覆盖掉默认的选项配置：

```
APP_CPPFLAGS := -pie -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=har
               -mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
APP_CFLAGS := -pie -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=hard
               –mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
APP_LDFLAGS := -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=hard
               –mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
```

而使用ndk-build脚本编译Native代码时，编译选项和链接选项如何查看呢？开发者仅需在使用ndk-build的时候在后面加上V=1就可以看到编译选项和链接选项，如下：

```
ndk-build V=1
```

执行以上这个命令后，就可以查看编译选项和链接选项是否是我们更改之后的选项了。

4.软浮点和硬浮点

上面我们曾说过Ne10的编译是使用cmake工具来编译的，所以我们必须正确安装cmake。而在Ne10的默认编译选项里包含了对浮点数运算的选项配置：

```
-mfloat-abi=hard -mfpu=vfp3
```

这个编译选项是指什么？其实是这样子的，在gcc的编译选项中，mfloat-abi参数可选值有三个，分别是soft、softfp和hard，解释如下。

·soft是指所有浮点运算全部在软件层实现，效率不高，适合早期没有浮点计算单元的ARM处理器。

·softfp是armel的默认设置，它将浮点计算交给FPU处理，但函数参

数的传递使用通用的整型寄存器而不是FPU寄存器。

·hard则使用FPU浮点寄存器将函数参数传递给FPU处理。

需要注意的是，在兼容性方面，soft模式与后两者是兼容的，但softfp和hard两种模式是不兼容的。默认情况下，在Application.mk里会有如下配置：

```
APP_ABI := armeabi-v7a
NDK_TOOLCHAIN_VERSION = 4.9
```

配置会使用NDK中4.9版本的gcc编译armv7-a平台下的包。当直接使用Ne10的静态库进行链接的时候，链接阶段就会出现错误：

```
ld: error: jni/prebuilt/libNE10.a(NE10_fft_generic_float32.c.o) uses
    VFP register arguments, output does not
```

这里就是libNE10.a这个静态库文件在编译阶段使用的编译选项中开启了硬浮点运算，而现在使用ndk-build脚本进行构建的时候使用的是软浮点，链接遇到了不同的输入文件类型就报送上述错误。要想解决这个问题，可以将APP_ABI配置成为armeabi-v7a-hard：

```
APP_ABI := armeabi-v7a-hard
```

这样gcc在寻找编译选项（setup.mk）的时候就会寻找硬浮点，setup.mk文件配置如下：

```
ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
    TARGET_CFLAGS += -mfloat-abi=softfp
else
    TARGET_CFLAGS += -mhard-float \
                     -D_NDK_MATH_NO_SOFTFP=1
    TARGET_LDFLAGS += -Wl,--no-warn-mismatch \
                     -lm_hard
endif
```

在最新版本中，NDK已经不支持armeabi-v7a-hard的模式，所以我

们需要配置编译选型和链接选项，配置编译选项里的将浮点预算使用硬浮点的fpu是为了确保正确性，链接选项里的-Wl，--no-warn-mismatch是为了告诉链接器忽略警告，确保链接成功，不要再检查输入文件的格式不同。所以最终的Application.mk如下：

```
APP_ABI := armeabi-v7a
APP_CPPFLAGS := -pie -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=har
               -mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
APP_CFLAGS := -pie -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=hard
               –mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
APP_LDFLAGS := -mthumb-interwork -mthumb -march=armv7-a -mfloat-abi=hard
               –mfpu=vfp3 -Wl,--no-warn-mismatch -std=gnu99 -fPIC
NDK_TOOLCHAIN_VERSION = 4.9
```

运行ndk-build脚本，执行完毕后若没有错误，就完全编译好了这个动态库。有的读者可能会问，是如何找到这些参数的呢？其实，前面所讲解的三个目录下都有一个CMakeFiles目录，每个目录下会有一个XXXXX.dir目录，这个目录里有一个flags.make及link.txt，这两个文件中就是编译和链接命令及其携带的参数。cmake是根据自己的脚本文件中的配置生成这些参数，但在这里找到的编译选项以及链接选项才是最终的选项，就像使用ndk-build脚本编译，我们写上V=1才可以看到中间过程编译和链接的选项。

还有一种方法就是配置Ne10的编译选项，使用软浮点，就需要我们在执行cmake的时候带上参数，即

```
cmake -DNE10_ARM_HARD_FLOAT=OFF
    -DCMAKE_TOOLCHAIN_FILE = ../android/android_config.cmake ..
```

这时编译出来的静态库就是使用软浮点运算的了，但是Ne10的作者不推荐这样做，因为这样一方面会降低效率，另一方面有一些汇编文件里的代码可能依赖于硬浮点的运算，这有可能会造成错误。

A.4 Ne10提供的Math函数列表

Ne10提供的Math函数列表如下：

1）浮点数组加（减、乘、除）一个float常量数值放入目标浮点数组中。

2）向量数组（二维、三维、四维）加（减、乘、除）一个向量常量放入目标向量数组中。

3）浮点数组加（减、乘、除）另外一个浮点数组（按照相同的index进行运算）放入目标浮点数组中。

4）向量数组（二维、三维、四维）加（减、乘、除）另外一个向量数组（按照相同index进行运算）放入目标向量数组中。

5）矩阵（二维、三维、四维）与矩阵的加、减、乘、除。

6）矩阵与向量相乘。

7）浮点数组都设置为一个常量。

8）浮点数组绝对值。

9）一个常量减去一个浮点数组中的每一个元素放置到目标浮点数组中。

10）向量常量减去一个向量数组中的每一个元素放置到目标向量数组中。

11）浮点数组乘以常量再加上另外一个浮点数组（按照index进行）放入目标浮点数组中。

12）向量数组乘以向量常量再加上另外一个向量数组（按照index进行）放入目标向量数组中。

13）浮点数组乘以另外一个浮点数组（按照index相乘）再加上一个浮点常量放入目标浮点数组中。

14）向量数组乘以另外一个向量数组（按照index相乘）再加上一个向量常量放入目标向量数组中。

15）矩阵的转置、单位矩阵运算等数学运算。

A.5　FFT性能测试

在不同的Android平台手机上做测试，FFT的结果要快3倍左右，有的甚至能达到5倍，所以效果还是很明显的。测试样本的时间长度为10s，采样频率为44100Hz，双声道的的PCM先做FFT，比较结果是否与MayerFFT一致（平方后相加进行浮点数比较，相差在0.0001以内），然后做逆FFT，听声音是否正常，如果没有问题，代表结果是正确的。最终结果的正确性与性能对比结果如表A-1所示。

表　A-1

机型	正确性	速度（DSP/CPU）	机型	正确性	速度（DSP/CPU）
Nexus 5X	正确	3.x 倍	魅蓝 2	正确	3 倍左右
Oppo R9	正确	3.x 倍	小米 3	正确	3.x 倍
三星 Note4	正确	3.x 倍	Oppo R9 Plus	正确	2.x 倍
VIVO X7	正确	2.x 倍	Lenovo K8	正确	2.x 倍
红米 Note3	正确	3.x 倍	小米 5	正确	3.x
三星 Note2	正确	2.x 倍	三星 S6	正确	3 倍左右
红米	正确	2 倍左右	坚果手机	正确	3 倍左右
小辣椒	正确	2.x 倍	小米 4	正确	4 倍左右

附录B　编码器的使用细节

B.1　AAC编码器的使用细节

音频的编码方式有很多种，目前最为流行的就是AAC的编码格式。这种编码格式可以在中低码率的限制下编码出较高质量的音频流。目前，AAC的规格（Profile）有以下三种。

·LC-AAC编码规格。

·HE-AAC v1编码规格。

·HE-AAC v2编码规格。

这三种规格的关系如图B-1所示。

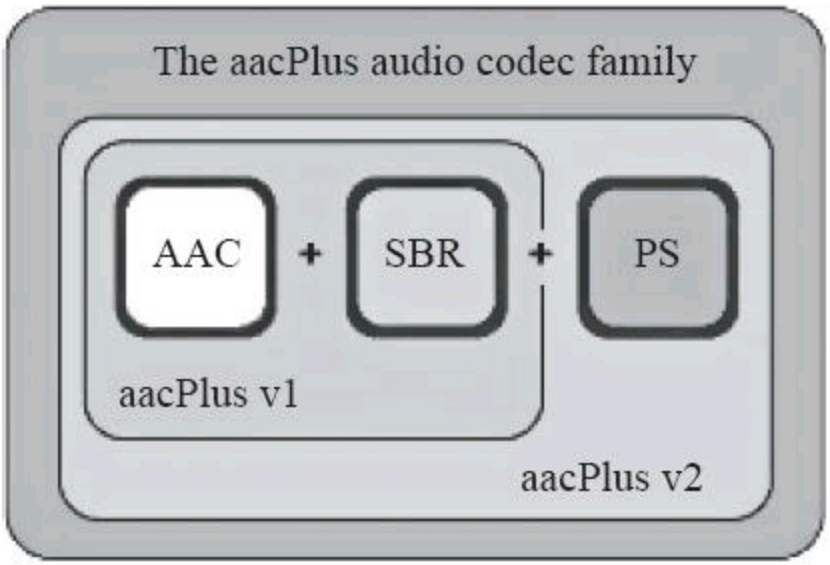


图　B-1

LC-AAC的Profile是最基础的AAC的编码规格。另外两种较为高级的编码规格中都带有HE字样，HE的全称是High Efficiency，翻译为中文就是高效性的意思。

HE-AAC v1（又称AACPlusV1，SBR）使用容器的方法实现了AAC（LC）和SBR技术，SBR代表Spectral Band Replication（频段复制）技术。音乐的主要频谱虽然集中在低频段，但是高频部分也是很重要的，因为高频段决定了整个音乐的音质。对全频段编码，若为了保护高频段，就会造成低频段编码过细而导致文件巨大；若保存了低频的主要成分而失去高频成分，就会丧失音质。这也是LC编码规格的最大问题，所以SBR技术应运而生。SBR把频谱切割开来，低频单独编码以保留主要的频谱部分，高频单独放大编码以保留音质，这样就能保证在减小文件大小的情况下还保存了音质，完美化解了这一矛盾。

HE-AAC v2编码规格也使用容器的方法包含HE-AAC v1和PS技术。PS全称为parametric stereo（参数立体声），而一个立体声文件的大小是一个单声道文件的两倍。但是，两个声道的声音存在某种相似性，根据香农信息熵编码定理，相关性应该被去掉才能减小文件大小。所以PS技术存储了一个声道的全部信息，然后，花很少的字节用参数描述另一个声道和它不同的地方。这样就去除了两个声道中的冗余信息，在音质损失很小的情况下，进一步减小了文件大小。

LC-AAC、HE-AAC v1、HE-AAC v2之间比特率和主观质量的关系是：在低码率的情况下，HE-AAC v1、HE-AAC v2编码后的音质要明显好于LC-AAC的。

在FFmpeg文档中设置AAC编码器的Profile有如下描述：支持LC-AAC这个Profile的编码器有libfdk_aac、libfaac、libvo_aac以及aac实现，但是支持HE-AAC以及HE-AAC v2的仅有libfdk_aac与libaacplus。所以开发者要想在FFmpeg中使用Profile为HE-AAC以上的AAC编码器，只能使用libfdk_aac或者libaacplus；否则，当调用FFmpeg的API打开编码器的时候，会收到“invalid AAC profile.”的错误返回值。

以代码的方式调用FFmpeg编码AAC的Profile设置如下：

```
AVCodecContext *encoder_ctx;
encoder_ctx->codec_id = AV_CODEC_ID_AAC;
encoder_ctx->sample_fmt = AV_SAMPLE_FMT_S16;
encoder_ctx->profile = FF_PROFILE_AAC_HE;
encoder = avcodec_find_encoder_by_name("libfdk_aac");
avcodec_open2(encoder_ctx, encoder, NULL);
```

使用ffmpeg的命令行工具正常编码一个AAC格式的文件命令如下：

```
ffmpeg -i source.wav -acodec libfdk_aac -b:a 64K target.m4a;
```

上述命令默认使用的编码规格就是LC的编码规格。那如何在命令行模式下调用FFmpeg编码AAC的Profile设置呢？命令如下：

```
ffmpeg -i source.wav
    -acodec libfdk_aac -profile:a aac_he -b:a 64K target.m4a;
```

上述命令使用libfdk_aac编码器，使用HE-AAC v1的编码规格将source.wav编码为码率是64K的target.m4a文件。

```
ffmpeg -i source.wav
    -acodec libfdk_aac -profile:a aac_he_v2  -b:a 48K target.m4a;
```

上述命令使用libfdk_aac编码器，使用HE-AAC v2的编码规格将source.wav编码为码率是48K的target.m4a文件。

将编码的target.m4a文件导入Praat软件，用频谱图来分析音质。下面通过码率为48K，分别以LC、HE-AAC以及HE-AAC2这三种不同的编码规格编码音频文件，然后导入Praat软件，通过频域图对比来分析音质的损失，如图B-2所示。

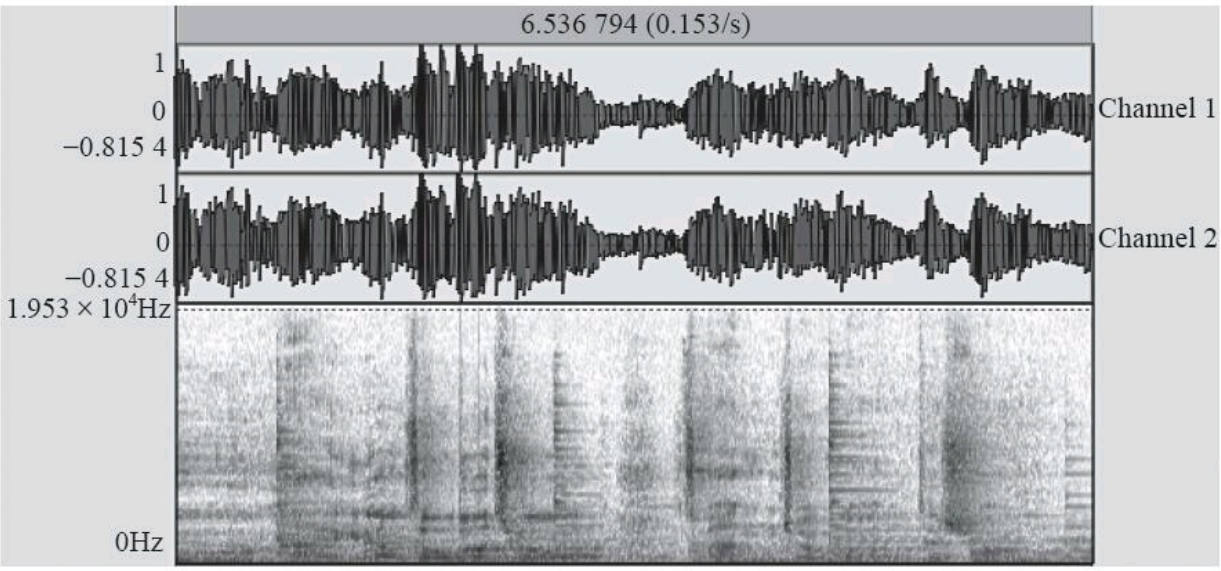


图　B-2

图B-2是原始声音的语谱图，原始声音为44100的采样频率，双声道的声音。根据奈奎斯特采样定律，频带分布到22050，所以全频带分布的截止频率就是22050。对于这个声音，我们使用比特率为48Kbps，再分别使用LC、HE-AAC以及HE-AAC v2来编码，然后观察编码之后的频带分布。LC编码规格下的语谱图如图B-3所示。

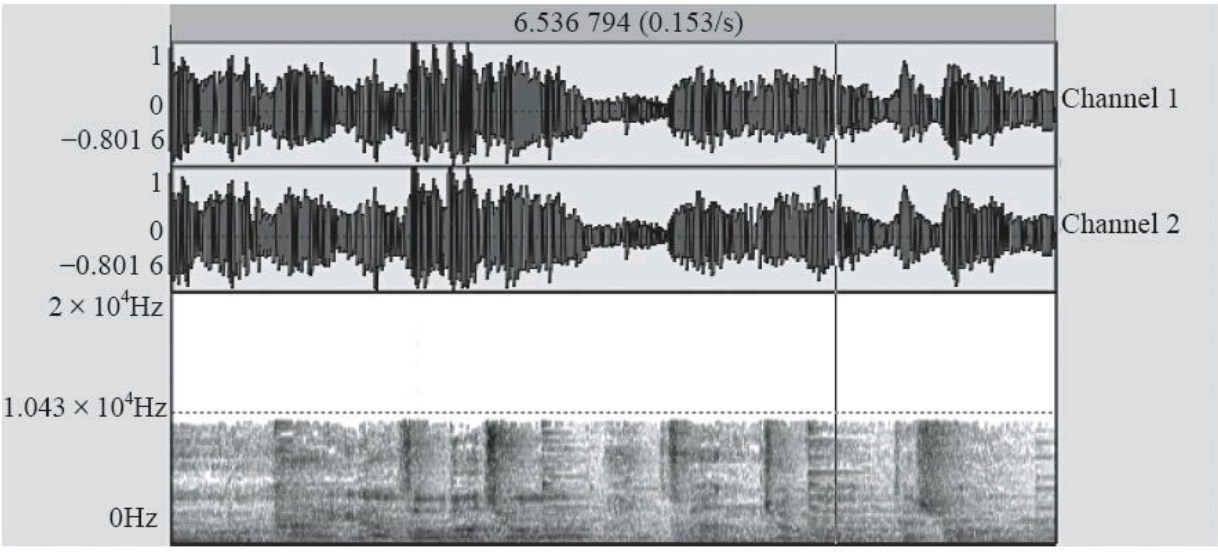


图　B-3

图B-3所示的就是LC Profile编码之后的语谱图，从图中可以看到它的频带分布到了10kHz就被截断，对于高频部分影响比较大。接着看以

HE-AAC编码规格编码出来的文件的语谱图，如图B-4所示。

从图B-4中可以看到，HE-AAC编码出来的文件效果要比LC的好，因为它的截止频率约到了16kHz以上。接着来看以HE-AAC v2编码规格编码出来的文件的语谱图，如图B-5所示。

从图B-5可以看到，HE-AAC v2编码出来的文件效果最好，因为几乎达到了全频带覆盖。

我们以FDK_AAC为例来看看各个Profile下推荐的码率跟采样率以及声道数的关系，如图B-6所示。

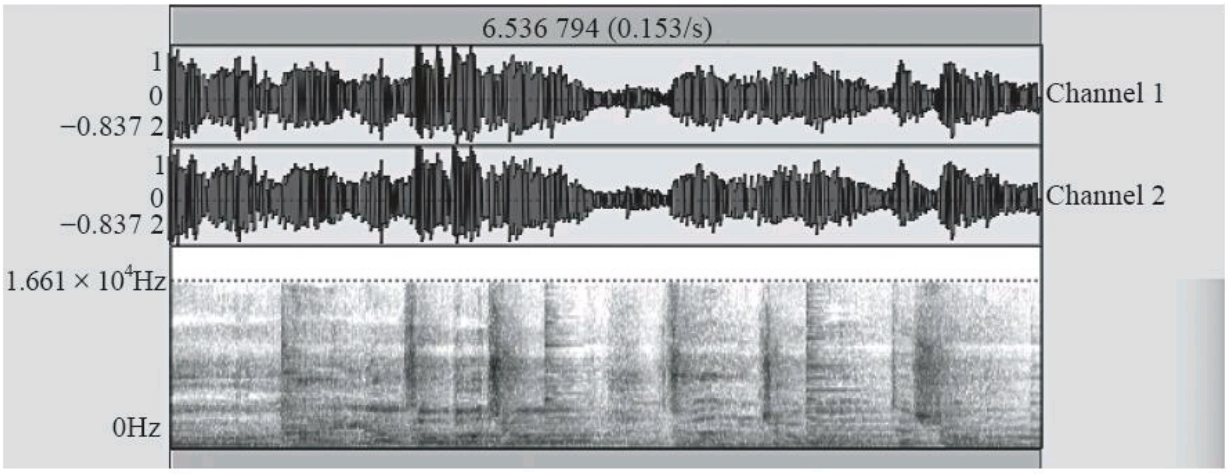


图　B-4

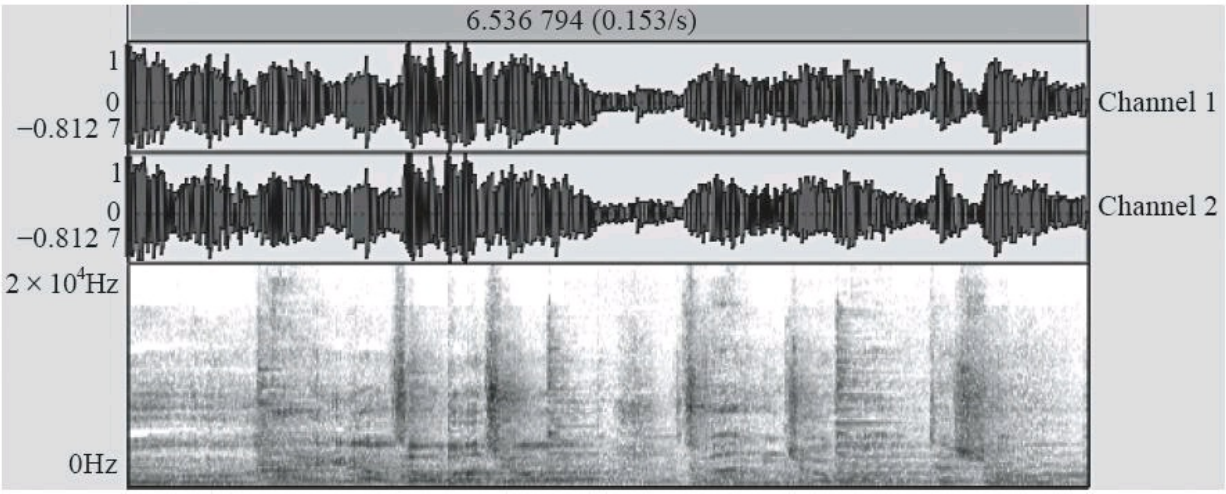


图　B-5

Audio Object Type	Bit Rate Range [bit/s]	Supported Sampling Rates [kHz]	Recommended Sampling Rate [kHz]	Number of Channels
[29] HE-AAC v2 (AAC LC + SBR + PS)	8000 - 11999	22.05, 24.00	24.00	2
	12000 - 17999	32.00	32.00	2
	18000 - 39999	32.00, 44.10, 48.00	44.10	2
	40000 - 56000	32.00, 44.10, 48.00	48.00	2
[5] HE-AAC (AAC LC + SBR)	8000 - 11999	22.05, 24.00	24.00	1
	12000 - 17999	32.00	32.00	1
	18000 - 39999	32.00, 44.10, 48.00	44.10	1
	40000 - 56000	32.00, 44.10, 48.00	48.00	1
	16000 - 27999	32.00, 44.10, 48.00	32.00	2
	28000 - 63999	32.00, 44.10, 48.00	44.10	2
	64000 - 128000	32.00, 44.10, 48.00	48.00	2
[5] HE-AAC (AAC LC + SBR)	64000 - 69999	32.00, 44.10, 48.00	32.00	5, 5.1
	70000 - 159999	32.00, 44.10, 48.00	44.10	5, 5.1
	160000 - 245999	32.00, 44.10, 48.00	48.00	5
	160000 - 265999	32.00, 44.10, 48.00	48.00	5.1
[2] AAC LC	8000 - 15999	11.025, 12.00, 16.00	12.00	1
	16000 - 23999	16.00	16.00	1
	24000 - 31999	16.00, 22.05, 24.00	24.00	1
	32000 - 55999	32.00	32.00	1
	56000 - 160000	32.00, 44.10, 48.00	44.10	1
	160001 - 288000	48.00	48.00	1
[2] AAC LC	16000 - 23999	11.025, 12.00, 16.00	12.00	2
	24000 - 31999	16.00	16.00	2
	32000 - 39999	16.00, 22.05, 24.00	22.05	2
	40000 - 95999	32.00	32.00	2
	96000 - 111999	32.00, 44.10, 48.00	32.00	2
	112000 - 320001	32.00, 44.10, 48.00	44.10	2
	320002 - 576000	48.00	48.00	2
[2] AAC LC	160000 - 239999	32.00	32.00	5, 5.1
	240000 - 279999	32.00, 44.10, 48.00	32.00	5, 5.1
	280000 - 800000	32.00, 44.10, 48.00	44.10	5, 5.1

图　B-6

AAC编码器的编码规格至此就讲解完毕了。对于解码端来讲，LC Profile是兼容性最好的编码规格。读者可以按照自己的应用场景去设置适合的码率与编码规格。

B.2　FFmpeg中使用libx264的码率控制

libx264是一个H.264/MPEG4 AVC编码器，对于普通用户，通常有两种码率控制模式：crf模式和ABR模式。码率控制就是一种决定为每个视频帧分配多少比特数的方法，它将决定文件大小和质量的分配。

B.2.1　crf模式

crf的全称是Constant Rate Factor，这种模式可以允许在输出的文件不太重要的时候，而达到特定的视频质量。这种编码模式的优点是，提供了最大的压缩效率，每一帧可以按照要求的视频质量去决定它需要的比特数；这种编码模式的缺点是，不能计算规定时间长度的视频文件的具体大小，或者准确控制输出码率。下面我们来看具体的使用步骤。

1.选择一个crf值

crf值是描述视频质量的一个量化值，取值范围为0～51，其中0为无损模式，23为默认值，51代表最差质量。该数字越小，图像质量越好。从主观上讲，18～28是一个合理的范围。18往往被认为从视觉上看是无损的，它的输出视频与输入视频几乎一样或者相差无几。但从技术角度来讲，它依然是有损压缩。若crf值加6，输出码率大概减少一半；若crf值减6，输出码率翻倍。通常，在保证可接受视频质量的前提下选择一个最大的crf值，如果输出视频质量很好，可以尝试一个更大的值来降低视频文件的大小；如果视频质量看起来很糟，可以尝试一个小一点的值，以提升视频质量，读者可以按照自己的应用场景来设置这个值。

2.选择一个预设

预设（preset）是一系列参数的集合，这个集合使得编码器能够在编码速度和压缩率之间做出权衡。同等视频质量下，一个编码速度稍慢的预设会提供更高的压缩率（压缩率是以文件大小来衡量的）。换言之，要想得到一个指定大小的文件或者采用恒定比特率编码模式，开发者可以采用一个较慢的预设来获得更好的质量。如果不需要考虑时间，x264建议开发者使用最慢的预设。目前，x264中所有的预设按照编码速度降序排列为：ultrafast、superfast、veryfast、faster、fast、medium、slow、slower、veryslow、placebo，默认预设为medium。开发者可以使用--preset来查看预设列表，也可以通过x264--fullhelp来查看预设所采用的参数配置。

开发者还可以基于输入内容的独特性通过使用--tune来改变参数设

置。当前的tune包括film、animation、grain、stillimage、psnr、ssim、fastdecode、zerolantency。假设你的输入内容为动画，则可以使用animation，或者你想保留纹理，就用grain。如果你不确定使用哪个选项或者你的输入与所有的tune皆不匹配，则可以忽略--tune选项。你可以使用--tune来查看tune列表，也可以通过x264--fullhelp来查看tune所采用的参数配置。

最后一个可选的参数是--profile，这个参数决定编码器到底使用哪一种H.264的编码规格（profile）来编码视频，当前所有profile包括baseline、main、high、high10、high422、high444，注意使用--profile选项和无损编码是不兼容的。

如下所示，作为一种快捷方式，你可以通过不声明preset和tune的内容来让ffmpeg罗列所有可能的内部preset和tune，如下：

```
ffmpeg -i input.flv -c:v libx264 -preset -tune output.mp4
```

执行上述指令，可以得到如下结果：

```
[libx264 @ 0x7f9ff8000600] Error setting preset/tune -tune/(null).
[libx264 @ 0x7f9ff8000600] Possible presets: ultrafast superfast veryfast
                                             faster fast medium slow slower
                                             veryslow placebo
[libx264 @ 0x7f9ff8000600] Possible tunes: film animation grain stillimage ps
                                          ssim fastdecode zerolatency
```

3.使用你的预设

一旦确定了预设，开发者就将这个预设设置给编码器，以便让编码器按照我们的预设编码对应的视频流。接下来将使用x264编码一个视频，我们使用一个比普通预设稍慢的预设，这样可以得到比默认设置稍好一点的视频质量。指令如下：

```
ffmpeg -i input.flv -c:v libx264 -preset slow -crf 22 -c:a copy output.mp4
```

在上述指令中，使用编码速度为slow、crf值为22的预设编码视频文件中的视频流，而音频流不做重新编码，直接重新Mux到新的输出视频

文件中。

B.2.2 ABR模式

ABR模式更注重码率的控制，适合在一段时间内生成固定大小的视频，而不太注重视频质量的场景。假如输入的视频时长是5分钟（300秒），要求转码后输出文件的大小为25MB，比特率的计算公式为文件大小除以时长，所以计算出比特率如下：

```
25MB * 1024 * 8 / 300s = 683Kbps
```

整体文件的比特率为683Kbps，如果要得到视频流的比特率，则需要使用整体文件的比特率减去音频流的比特率。音频的比特率计算如下：

```
683kbps - 128kbps(音频比特率) = 555kbps
```

根据上述计算可以得到视频流的比特率为555Kbps，然后使用命令行工具ffmpeg进行转码，命令如下：

```
ffmpeg -i input.flv -c:v libx264 -preset medium -b:v 555k
    -c:a libfdkaac -b:a 128k –f mp4 output.mp4
```

ABR编码模式提供了某种“运行均值”的目标，终极目标是最终文件大小匹配这个“全局平均”数字。因此，如果编码器遇到大量码率开销非常小的黑帧，它将以低于要求的比特率编码。但是，在接下来的几秒内，非黑帧它将以高质量的编码方式使码率回归均值。另外，可以与“max bit rate”配合使用来防止码率的波动。

开发者可以使用-x264opts参数来重写预设或者使用libx264的私有选项，比如设置关键帧的指令如下：

```
ffmpeg -i input.flv -c:v libx264
    -x264-params keyint=30:min-keyint=30:no-scenecut=1
    –acodec copy –f mp4 output.mp4
```

上述指令设置关键帧间隔为30帧一个关键帧，接下来设置编码器在编码过程中不适用B帧，指令如下：

```
ffmpeg -i input.flv -c:v libx264
    -x264-params keyint=30:min-keyint=30:no-scenecut=1:bframes=0
    –acodec copy –f mp4 output.mp4
```

还有一种编码模式是大家经常提及的，即CBR编码模式，全称是Constant Bit Rate。事实上，根本就没有CBR这种模式，但是开发者可以通过补充ABR参数“模拟”一个恒定比特率设置，比如：

```
ffmpeg -i input.flv -c:v libx264 -b:v 500k
    -minrate 500k -maxrate 500k -bufsize 1500k output.mp4
```

在上述代码中，-bufsize是一个“码率控制缓冲区”，它会在每一个有用的1500k视频数据内强制你所要求的均值（此处为4000k），所以我们会认为接收端/终端播放器会缓冲那么多的数据，因此在这个数据内部波动是没有问题的。当然，如果只有黑帧或者空白帧，它所花费的比特率将少于需求的比特率。

还有一种经常使用的最大比特率的crf模式，它是通过声明-crf和-maxrate参数来设置最大比特率的，比如：

```
ffmpeg -i input.flv -c:v libx264 -crf 20 -maxrate 600k -bufsize 1800k output.
```

这会有效地将crf值锁定在20，但是，如果输出码率超过600Kbps，这种情况下编码器会将质量降到低于crf 20。libx264中还有一种对于延迟的设置，即-tune zerolatency，这个设置可以有效降低编码的延迟输出，在直播以及VOIP场景中是必须设置的。

最后一点要考虑的就是兼容性问题。如果想让你的视频最大化和目标播放设备兼容（比如老版本的ios或者所有的Android设备），那么可以这做：-profile：v baseline会关闭很多高级特性，并提供很好的兼容性。常用的编码规格还有Main和High，与AAC的编码规格类似，规格越高，兼容性越差，同时在更低码率下编码出来的视频质量会更高。

下面看看在代码层面如何调用FFmpeg的API去设置libx264的预设以及参数，代码如下：

```
AVCodecContext* pCodecCtx;
pCodecCtx->max_b_frames = 0;
```

上述代码表明是否使用b帧，设置为0，代表不使用B帧；设置为3，代表1个gop之内使用3个B帧。

```
av_opt_set(pCodecCtx->priv_data, "preset", "superfast", 0);
av_opt_set(pCodecCtx->priv_data, "crf", "20", AV_OPT_SEARCH_CHILDREN);
```

上述代码相当于命令行模式下的设置预设和crf。

```
pCodecCtx->flags |= CODEC_FLAG_QSCALE;
pCodecCtx->qmin = 10;
pCodecCtx->qmax = 30;
```

上述代码利用qscale参数来设置视频质量，qscale是以<q>质量为基础的VBR编码模式，取值范围为0.01～255，值越小，质量越好，即-qscale 4和-qscale 6比较的话，4的质量比6的好。该参数使用次数较多，实际使用时发现，qscale是一个固定量化因子，设置qscale之后，前面设置的-b就无效了，而是自动调整了比特率。-qmin q表示最小视频量化标度（VBR）设定最小质量，与-qmax（设定最大质量）共用，-qmax q表示最大视频量化标度（VBR），使用该参数，就可以不使用qscale参数。

附录C　视频的表示与编码

真正的高手是不会因为语言的限制（C语言中没有类、接口、继承等特性）而写出一个不可维护的系统，而是会依据所使用语言的自身特性去设计并实现出一个可扩展性与可维护性良好的系统。所以语言不重要，对编程思想的认识才是最重要的。本附录会和读者分享视频帧表示格式的演进以及编码器是如何工作的。

C.1　视频帧的表示格式

如果一张图片使用RGBA格式来表示，并且每一个通道都使用一个字节来作为它的精度表示，那么一个像素就需要占用四个字节，一张宽度为720像素、长度为1280像素的图片所占用的空间大小为：

```
1280 * 720 * 4 = 3.515625MB
```

如果一个视频的帧率（fps）为24fps，长度为5分钟，那么这个视频用RGBA原始格式表示，所占用的大小为：

```
3.515625MB * 24 * 5 * 60 = 24.72G
```

一个时长为5分钟的720P的视频若占用这么大的空间，显然在存储以及网络传输方面会有非常大的问题。如果在没有任何压缩的情况下，使用这种格式表示视频，是肯定不能接受的。如果仅使用完全无损的数据压缩算法，比如DEFLATE（在PKZIP、Gzip和PNG中使用），是无法达到我们需要的存储与带宽需求的，所以我们要找到其他方法来压缩视频。

为了做到这一点，数字视频的开发者利用人类眼睛的视觉效果来达到要求，包括：

·人类眼睛对亮度的敏感度远远大于颜色的敏感度；

·一段视频中包含了大量的视频帧，而相邻的视频帧之间几乎没有任何变化；

·一帧图像内部包含了许多使用相同或相似颜色的区域。

第二点指的是当前视频帧和下一个视频帧以及下下个视频帧的差距几乎很小，所以可以公用一些相同的部分，即如果当前帧存储下来之后，下一帧和下下帧只需要存储增量信息或者变化的信息就可，这可以达到去除时间冗余的目的；第三点指的是，如果把一帧图像划分成很多个小区域，那么有许多区域是相同的，我们可以利用相同的区域仅存储一份的策略来压缩视频帧，以达到去除空间冗余的目的。而本节重点介绍第一点。第一点是利用人的眼睛对亮度的敏感度要明显高于对颜色的敏感度来压缩视频的。首先用一张图片来验证这个生理现象，如图C-1所示。

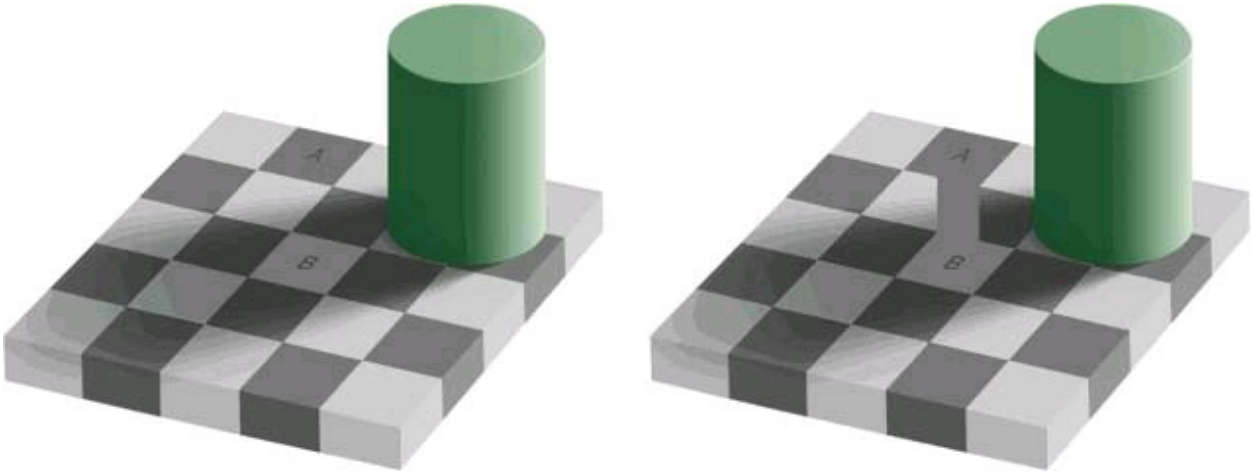


图　C-1

首先请读者看左边的图片，如果可以看出块A和块B的颜色是不相同的，那说明我们的眼睛是没有问题的。再看右边的图片，在块A和块B之间有一个连接器，其实这两个块的颜色是相同的。那为什么左边的图片看起来不一样呢？这是我们的大脑在捉弄我们，大脑让我们更多地关注明亮程度而不是颜色。所以我们一旦知道人类眼睛的这一生理特点（对图片中的亮度更加敏感），就可以尝试利用它。我们之前使用RGBA的表示格式来描述一帧视频帧，但是也有其他的表示格式，有一种表示格式将亮度（luminance）与色度（chrominance）分开来表示，这就是所谓YCbCr格式表示视频帧的方式。

YCbCr颜色模型使用Y通道表示亮度，使用两个颜色通道Cb（蓝色程度）与Cr（红色程度）表示色彩。YCbCr表示格式可以从RGB表示格式中派生出来，当然也可以转换回RGB格式。所以使用这种模型，也可以创建出完整的彩色图像，如图C-2所示。

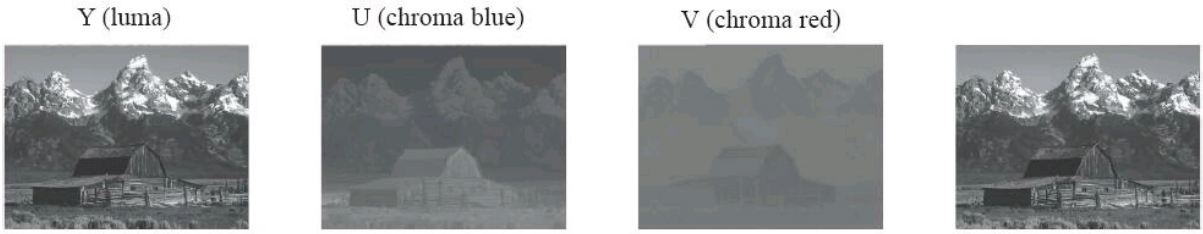


图　C-2

有些人可能会说，我们怎么能在不使用绿色的情况下生产出所有的颜色呢？为了回答这个问题，我们将讨论从RGB到YCbCr的转换。我们将使用标准的BT.601的系数，它是由ITU-R小组推荐的，第一步是计算luma，并替换RGB值。

```
Y = 0.299R + 0.587G + 0.114B
```

一旦有了亮度值，就可以计算出Cb和Cr的值了，计算公式如下：

```
Cb = 0.564(B - Y)
Cr = 0.713(R - Y)
```

我们也可以从YCbCr转换为RGB的格式，甚至可以得到绿色，计算公式如下：

```
R = Y + 1.402Cr
B = Y + 1.772Cb
G = Y - 0.344Cb - 0.714Cr
```

既然YCbCr这种表示格式天然地将亮度信息和色度信息分开通道存储了，那么我们可以利用人类眼睛对亮度信息更加敏感这一特性来对色度信息进行降采样处理，如图C-3所示。

图C-3这种表示格式也就是大家最常使用的YUV420P的表示格式，一般在YCbCr的表示格式中常分为三部分来表达，即a：x：y，这就定义了a*2个像素中的亮度与色度的关系。比如，YUV444代表色度信息不经过任何的降采样处理，也称为无压缩的YUV格式；而格式YUV420不是不需要V通道，而是每八个像素有两个U和两个V，如图C-4所示：

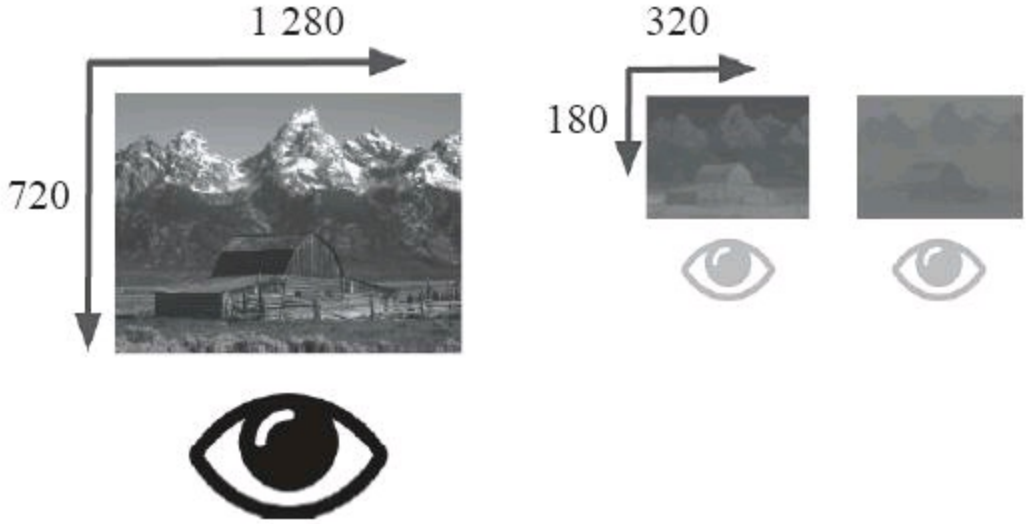


图　C-3

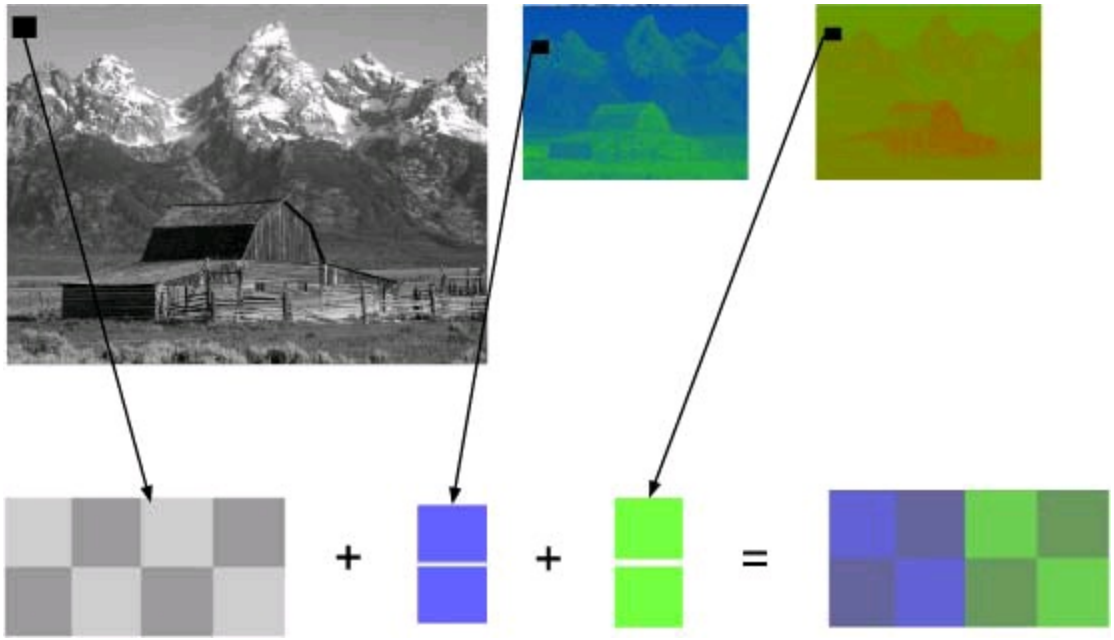


图　C-4

这样1280×720的一帧视频帧所占的存储容量计算如下：

```
1280 * 720 * 1.5 = 1.3184MB
```

而YUV的表示格式的出现也是为了彩色电视机可以兼容黑白电视机的一种实现方案，即黑白电视仅需要使用Y通道的数据，就可以显示出所需要的效果。

C.2　显卡上传中字节对齐

视频帧虽然使用YUV格式来表示，但是最终渲染到屏幕上的还是以RGB的形式渲染上去的，因为无论任何设备的屏幕都是由无数个RGB的子像素点来呈现图像的像素的，如图C-5所示。

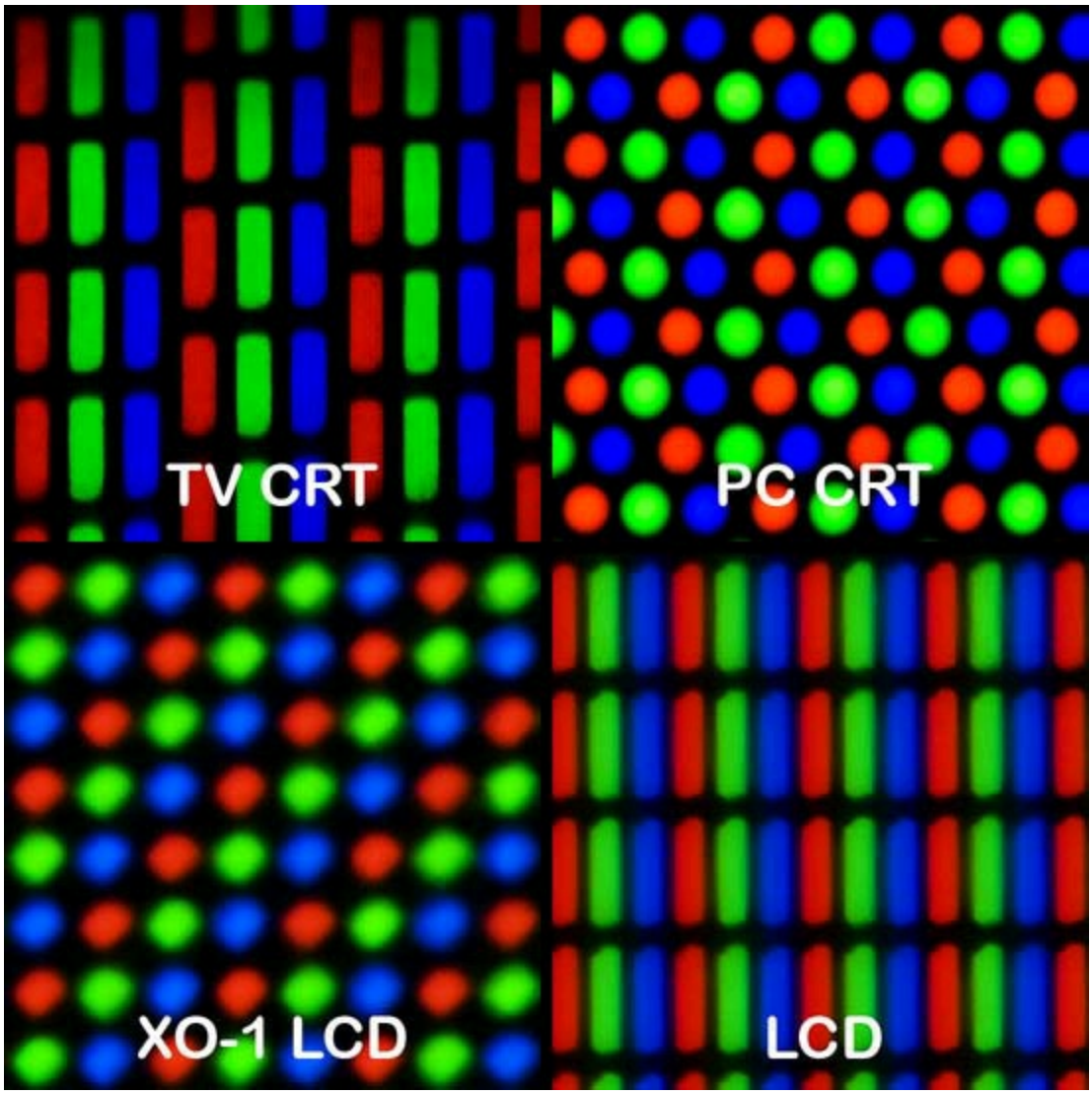


图　C-5

如果使用CPU将每一帧视频帧从YUV格式转换为RGB格式再去渲染到屏幕上，效率通常都会非常低。而使用OpenGL ES将YUV渲染成为RGB无疑是效率比较高的一种方式，要想完成使用OpenGL ES转换YUV，第一步就是要把YUV上传到显卡中的一个已知纹理上去。在上传之前，有一个对齐像素字节的函数需要调用，OpenGL ES中提供了方法原型，如下：

```
glPixelStorei(GLenum pname, GLint param);
```

上述函数的含义是设置像素存储模式，它包含有两个参数。

·pname：指定要被设置参数的符号名，以枚举的形式定义在OpenGL ES中，第一种枚举类型是GL_PACK_ALIGNMENT，它影响将像素数据写回到内存的打包操作，对glReadPixels函数的调用产生影响；第二种枚举类型是GL_UNPACK_ALIGNMENT，它影响显卡从内存读到的像素数据的解包操作，对glTexImage2D以及glTexSubImage2D函数的调用产生影响。

·param：指定为相应的pname类型设置的值。可选值有1、2、4或8，默认值为4，该值用于指定存储器中每个像素行有多少个字节对齐，对齐的字节数越高，系统优化的空间就会越大，即将内存中的多个字节一起上传到显卡中或者从显卡中下载到内存中。

在实际代码中，我们看到的调用可能如下：

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

比如最常见的YUV420P格式中YUV的比例为4：1：1，即每四个像素用4个Y、1个U和1个V来表示，而标清分辨率有320×240或者640×480（480P），高清分辨率一般是1280×720（720P）或者1920×1080（1080P）。而显卡在传输数据时默认为4字节对齐（param默认值为4），也就是对像素数据按4字节对齐进行存取。所以显卡更偏向于每一行的数据量是4的整数倍（按上述分辨率来看恰好是比较常见的）。所以为了更高的存取效率，OpenGL默认将像素数据按4字节的方式传输向显卡。这在常见的分辨率下都没有问题，问题在于对于非4字节对齐的像素数据，第一行的最后一次打包的4字节将包括第一行的结束数据部分和第二行开始的数据部分，当然致命的不是在这里，而是在最后一行，存取将很可能会越界。为了防止这样的情况发生，一是硬性把像素数据延展成4字节对齐的（就像BMP文件的存储方式一样）；二是选择绝对会造成4字节对齐的颜色格式或值格式（GL_RGBA，或者GL_INT、GL_FLOAT之类的表示格式）；三是以牺牲一些存取效率为代价，去更改OpenGL的字节对齐方式，将param值设置为1。所以，要想提高效率，尽量保证每一行的U或者V是4的整数倍。但是，如果出现奇葩的分辨率，比如420×314，每一行的Y通道是420个值，除以4是可以除尽的，但是每一行的U或者V仅有105个值，就不再是4的整数倍了，这种情况下，如果不去设置对齐方式，就有可能出现Crash或者出现颜色混乱的问题。解决办法是让字节对齐方式从默认的4字节对齐改

成1字节对齐（选择1，无论分辨率怎样，都是绝对不会出问题的，但是效率下降了），下面给出在YUV420P格式下的通用解决方法，如下所示：

```
int frameWidth = frame->width;
int frameHeight = frame->height;
if(frameWidth % 16 != 0){
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}
```

上述代码可以作为通用解决问题的方法，即当宽度是16的整数倍的时候，就不去做任何设置，以默认的4字节对齐作为对齐方式，为什么是16呢？因为像素数目除以4是U或者V的数目，再保证以4字节对齐，所以就是16了。当不是16的整数倍的时候，就需要设置1字节对齐的方式作为对齐方式。

C.3 编码器的工作编码原理

前面介绍了为了让视频可以存储到硬盘以及可以满足网络传输，不仅需要利用人类眼睛的生理特点将RGB表示格式换为YUV420P格式来表示（虽然这将存储大小减小了一半），还应该使用有损的压缩方式，将YUV420P的原始数据压缩到更小。前面还介绍过有两种方式可以压缩视频原始数据：一是去除时域上的冗余信息；二是去除单张视频帧的空间冗余信息。那这两部分工作也是编码器工作的重点，本节就来介绍这两部分的通用实现手段，但是在不同的编码器中实现的方式又有所不同，编码出来的视频质量以及性能消耗也是不同的，而性能和质量也是开发者选用编码器的衡量指标。

C.3.1 帧类型介绍

在尝试消除冗余信息（不论是时间的冗余信息还是空间的冗余信息）之前，我们先来统一一下术语。假设有一个帧率（fps）为30fps的电影，图C-6是前四帧的内容。

图　C-6

在这四帧视频帧中，我们可以看到大量的重复信息，比如蓝色的背景，它在第一帧到第四帧中并没有任何变化。为了消除这些冗余信息，我们可以将视频帧的类型抽象为三种类型。

·I帧

I帧是一个仅包含当前帧信息的视频帧类型，所以它又被称为参考帧、关键帧。在解码过程中，它不需要依赖任何帧就可以被解码出来，一个I帧看起来和一张静态图片非常类似，视频流或者视频文件中第一帧通常是I帧，并且会定期在其他帧类型中间插入I帧。

·P帧

P帧是前向参考帧，可以使用前面的I帧与P帧来呈现出当前的视频帧，这也是解码器的解码规则，例如图C-6中第二帧视频帧与第一帧视频帧的变化只是球向右前方移动了，所以我们可以基于第一帧视频帧来描述变化部分的内容作为第二帧视频帧的内容，这样第二帧视频帧所占用的空间就会大大减小。

·B帧

相较于P帧仅参考前面的视频帧内容，B帧又增加了对后边视频帧的参考，这样可以提供更好的压缩。但是，这对于编码器来说，计算量增大的同时，也增加了编码输出的延迟时间（因为需要参考后续过来的视频帧），所以在一些实时性要求较高的场景下（比如电话会议）通常不使用B帧。

这些帧类型共同组成了一个完整的视频，如图C-7所示。

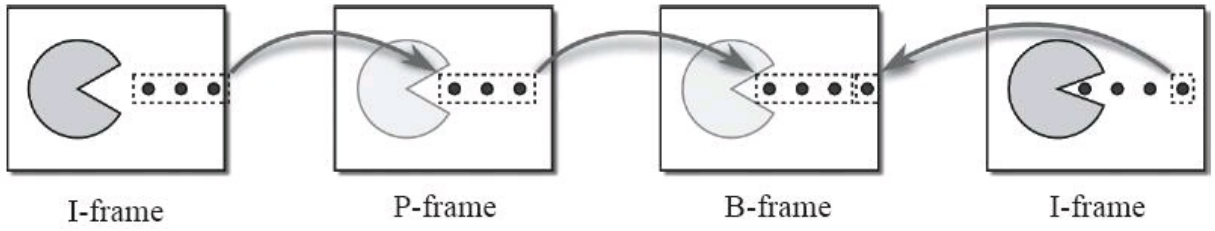


图　C-7

如果仅从存储或者网络带宽角度来衡量，我们可以认为I帧是最昂贵的，P帧会便宜一些，B帧则最便宜。

C.3.2　消除时间的冗余信息

本节我们一块来讨论如何消除视频帧在时间上的冗余信息，比较成熟的技术就是帧间预测技术（inter-frame prediction）。我们先来看图C-8的两帧视频帧。

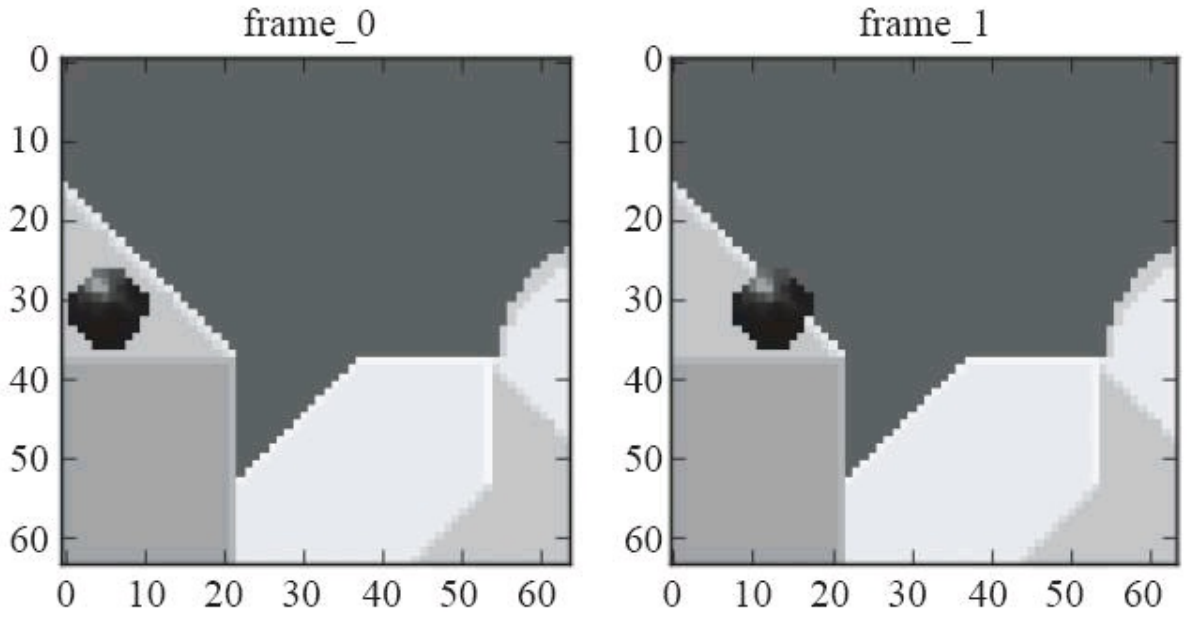


图　C-8

我们将去除时间上的冗余信息来达到使用更少的字节存储这两帧视频帧的目的，自然想到的就是将这两帧视频帧做减法，求出diff值，就是我们要进行编码的东西，如图C-9所示。

其实还有一种更好的方法可以使用更少的比特数来存储第二帧视频帧的内容。首先，我们将每一帧视频帧（frame_0）分为很多个部分，然后将这两帧视频帧中的每一部分进行匹配，这种算法我们可以看成是运动估计，如图C-10所示。

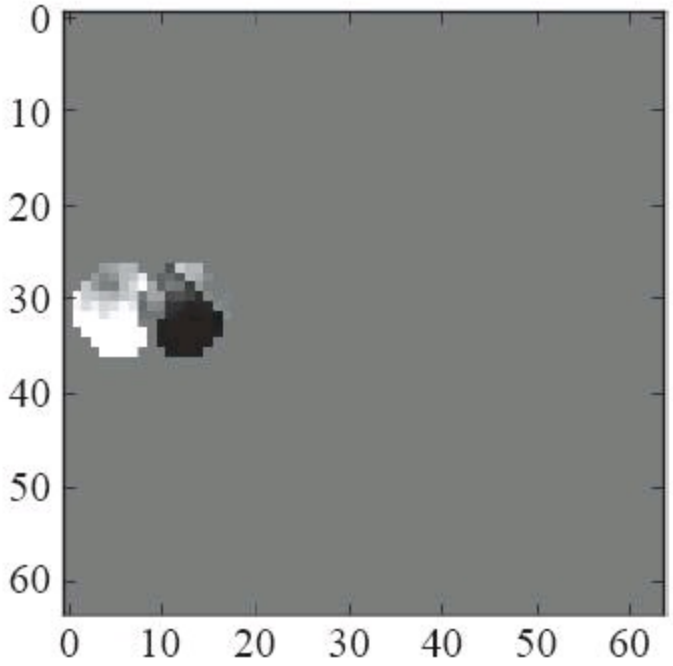


图　C-9

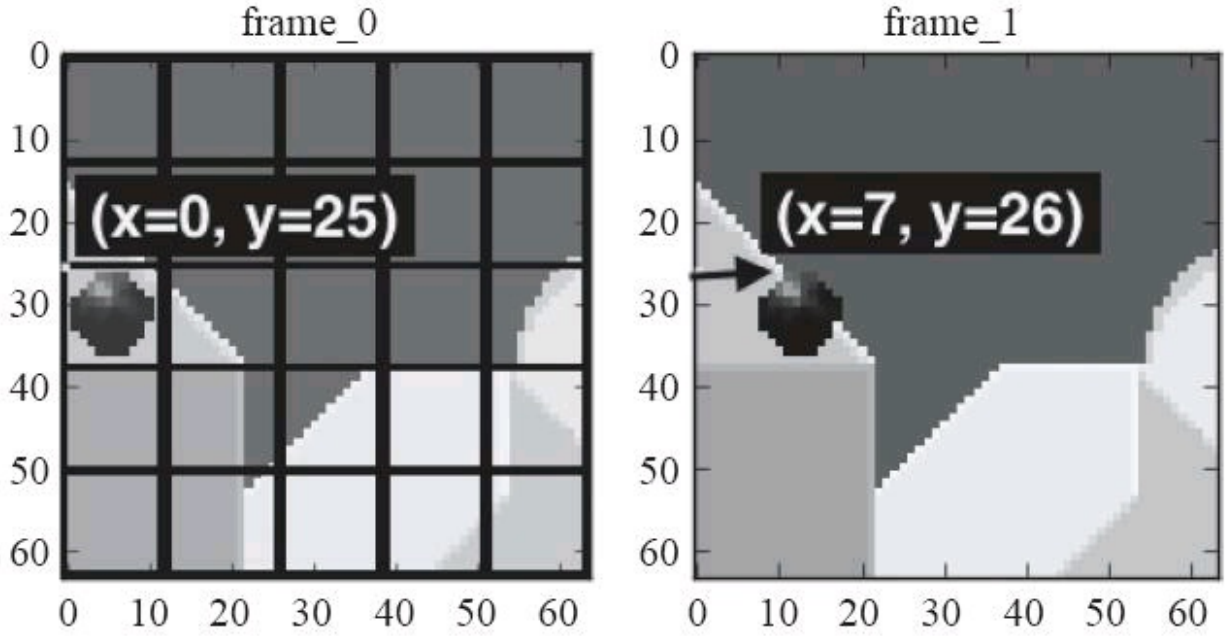


图　C-10

从第一帧变化到第二帧的过程中，我们可以估计第一帧视频帧中的黑色球从点（x=0，y=25）到点（x=7，y=26），而x和y组成的值的集合就称为运动向量。我们可以更进一步来节省编码内容，即仅对这两帧

之间的运动矢量差进行编码，所以最终运动矢量是x=6（6-0），y=1（26-25）。当然，在真实的编码过程中，图中的小球会划分为N个部分，我们把它划分为一部分只是为了更加方便理解。所以，当应用了运动估算算法之后，编码的数据要比简单地计算Diff值再进行编码的数据要少得多。

其实可以使用ffmpeg命令行工具来看到一个视频的分块情况，命令如下：

```
ffmpeg -debug vis_mb_type -i input.mp4 output.mp4
```

这样去播放output.mp4文件，就可以看到视频的分块情况。当然，也可以直接使用ffplay播放分块的视频，命令如下：

```
ffplay -debug vis_mb_type input.mp4
```

还可以使用ffmpeg工具来查看运动矢量的情况，命令如下：

```
ffplay -flags2 +export_mvs -vf codecview=mv=pf+bf+bb input.flv
```

至此，时间冗余信息可以利用运动估计算法给消除掉。之所以可以使用运动估计，需要一个帧作为参考帧，也就是I帧。那么I帧是如何进行压缩的呢？这就是接下来我们要讲解的空间冗余信息的消除部分。

C.3.3　空间的冗余信息消除

在一帧视频帧中，我们可以看到大量的重复信息，如图C-11所示。

图　C-11

图C-11中可以看到有大量的蓝色和白色，如果这是一帧I帧的话，就无法使用帧间预测技术来压缩它。因此，我们要采用其他办法来压缩这张图片，因为它的重复信息比较多，如图C-12所示。

如图C-12所示，我们将对标出来红色块部分进行编码，还可以根据红色块周围的颜色来预测当前部分的颜色值，比如可以预测帧将继续垂直传播颜色，这意味着未知像素的颜色将保持其邻居的值，如图C-13所示。

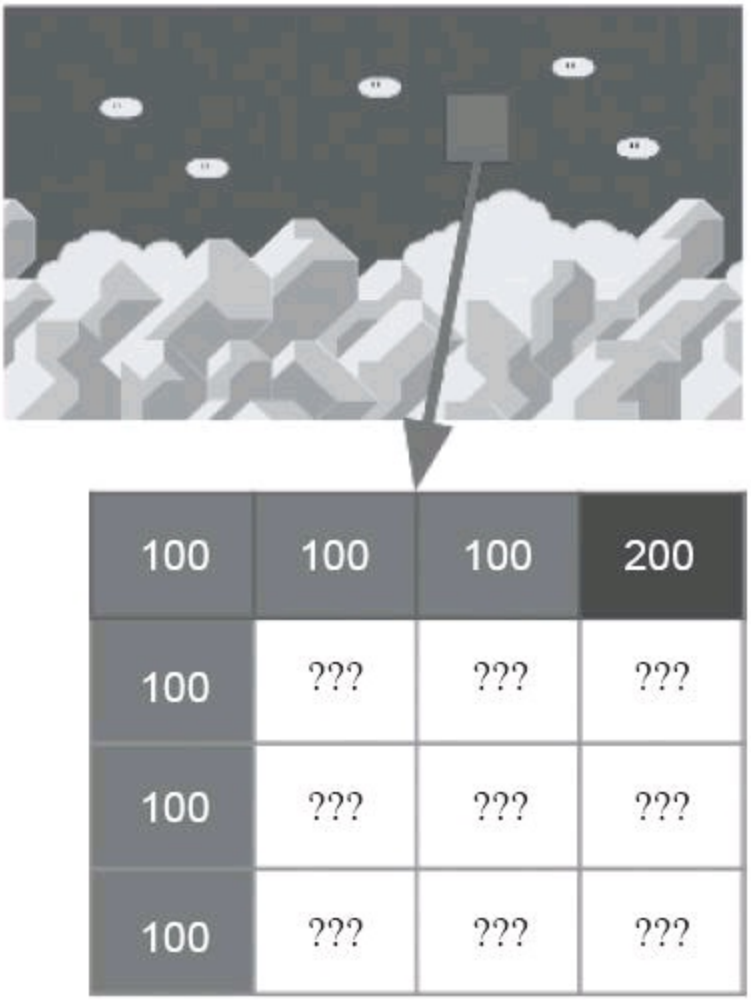


图　C-12

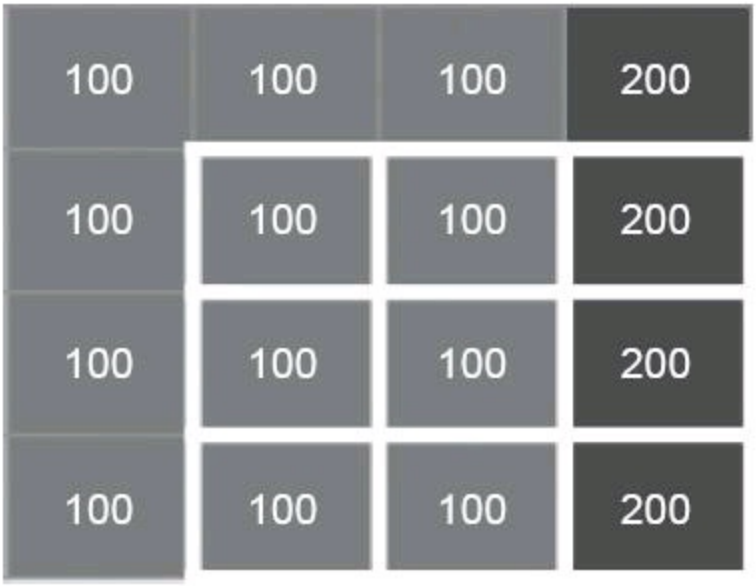


图　C-13

但是我们的预测也有可能是错误的，所以需要使用另外一项技术，即帧内预测技术。使用真正的值减去预测的值，可以得到一个更容易不压缩的矩阵，如图C-14所示。

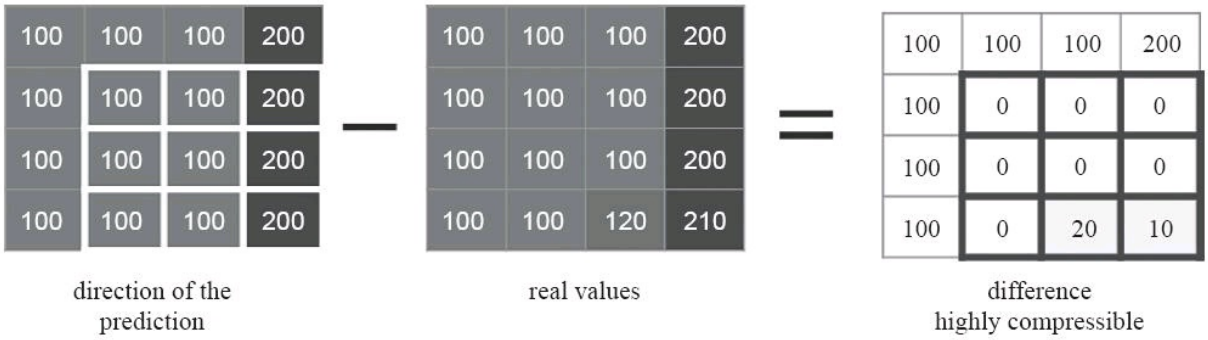


图　C-14

帧内预测技术使得空间压缩变成现实。编码器还需要使用量化、熵编码等步骤共同来编码出最终的视频。如果读者有兴趣了解更多的内部细节，可以参考libx264的官方文档。