学校代码：10286
分类号：TP311
密　级：公开
UDC：004.4
学　号：153483

东南大学

SOUTHEAST UNIVERSITY

硕士学位论文

基于 RTSP 协议的 iOS 视频播放器的设计与实现

研究生姓名：**刘　佳　林**

校 内 导 师：**吉　逸　教授**

宛　斌　讲师

校 外 导 师：**彭　鹏　高工**

申请学位类别 工程硕士　学位授予单位 **东 南 大 学**

一级学科名称 软件工程　论文答辩日期 **2018 年 05 月 18 日**

二级学科名称 软件工程　学位授予日期 **2018 年　月　日**

答辩委员会主席 **徐立臻**　评　阅　人 **翟玉庆　徐学永**

2018 年　月　日

東南大學

硕士学位论文

基于 RTSP 协议的 iOS 视频播放器的设计与实现

专 业 名 称： 软件工程

研究生姓名： 刘 佳 林

校 内 导 师： 吉 逸 教授

宛 斌 讲师

校 外 导 师： 彭 鹏 高工

DESIGN AND IMPLEMENTATION OF IOS VIDEO PLAYER BASED ON RTSP PROTOCOL

A Thesis Submitted to

Southeast University

For the Academic Degree of Master of Engineering

BY

LIU Jialin

Supervised by

Prof. JI Yi

and

Lect. WAN Bin

and

S.E. PENG Peng

College of Software Engineering

Southeast University

May 2018

东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

研究生签名：_____日期：_____

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆、《中国学术期刊（光盘版）》电子杂志社有限公司、万方数据电子出版社、北京万方数据股份有限公司有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括以电子信息形式刊登）论文的全部内容或中、英文摘要等部分内容。论文的公布（包括以电子信息形式刊登）授权东南大学研究生院办理。

研究生签名：_____导师签名：_____日期：_____

摘要

随着经济的发展，人们对子女教育的要求也在逐渐提升，近些年中小学生的校园安全事件频繁发生，使得家长更加关注学生在校的生活情况，基于这些问题一些企业适时推出了家校通系统，更方便地联系了家长和学校。视频监控是家校通系统中的重要组成部分，随着移动互联网的高速发展，将监控视频画面转移到手机上已经是大势所趋。为了实现家校通系统的移动应用更高效地播放监控视频，本文通过对在线视频播放的研究，实现了一套在 iOS 设备上使用的轻量级视频播放框架，该框架能够对采用 RTSP 传输协议的流媒体进行流畅播放。

针对轻量级 iOS 视频播放器框架的实现，本文主要研究了以下几个问题：第一，精简视频框架 FFmpeg。传统播放器因为支持众多的流媒体传输协议和视音频编码格式，需要大量的协议解析模块和解码器。家校通系统使用的在线视频网络传输协议单一，视音频采用的编码格式已知，所以本文通过对 FFmpeg 的研究，剔除在项目没有使用的网络模块和解码模块，将 FFmpeg 精简到了原规模的 7.81%。第二，播放器模块的封装。为实现播放在线视频并提供简单方便的交互方式，播放器框架采用 iOS 系统中常用.framework 模式，.framework 模式只需暴露较少的头文件，就可以实现全部与播放相关的功能；播放事件传递采用了 iOS 系统通知的方法，对不同的事件设计特定的键值，外界通过键值监听系统通知就能获得对应事件的消息。第三，播放框架的性能优化。大量重复的解码工作和视图绘制渲染会过多地使用 CPU 等资源，这是造成播放器性能低下的主要原因。对于视频解码的优化，本文优先采用 VideoToolbox 框架进行硬件解码，转移了视频解码的对 CPU 压力；大量视图渲染的优化采用的是 OpenGL ES 框架，该框架将主要的渲染工作转移到了 GPU，减少了 CPU 的工作量。

论文的最后对视频播放器框架进行了测试，整个框架占用的 ROM 空间只有 9.8MB，完全符合预期；通过与其他实现方法对比，播放器在运行时的性能表现出很大优势，与业内其他产品对比也处于领先或者持平状态。目前该框架已运用到了对应的移动产品中，符合项目的需求。

关键词：iOS；RTSP；FFmpeg；硬件解码；OpenGL ES

ABSTRACT

With the development of the economy, people's requirements for their children's education are gradually increasing. In recent years, students' safety issues in school have frequently occurred, these events make parents pay more attention to students' life in schools. Based on these issues, some companies have launched home-school system which can contact parents and schools easily. Video surveillance is a very important part of home-school system. With the rapid development of the mobile network, moving video surveillance to mobile phones is a general trend. In order to play surveillance videos more efficiently in mobile devices, this paper researched online videos playback and achieved a set of lightweight video-playback-framework for iOS system platforms with RTSP streaming media protocol.

For the implementation of lightweight iOS player framework, we mainly studyed the following issues: Firstly, rightsizing the video framework FFmpeg. Traditional players need a large number of protocol parsing modules and decoders because they support numerous streaming media transmission protocols and encoding formats. The network transmission protocol for home-school system is single, and the encoding formats of multimedia are known. Through the research on FFmpeg, this paper eliminated the network module and decoding module that were not used in the project, and reduced FFmpeg to 7.81% of the original scale. Secondly, packaging of this video player. In order to play online videos of the home-school system and provide a better interactive way, the player framework adopts .framework mode. This mode is commonly used in iOS App development, because all playback-related functions can be used with less header files being exposed. The playback event transmission solution adopts iOS System Notification. With this method, specific key values are designed for different events. The external users can obtain the corresponding event messages by the monitoring System Notification with key values. Thirdly, optimizing performance of this player framework. Repeated decoding work and a large number of images rendering use a large amount of resources such as CPU, which are the major causes of low performance in video player. For the optimization of video decoding, the VideoToolbox framework is used for hardware decoding in the project, which shifts the CPU pressure for video decoding. The optimization of large amount of images rendering use the OpenGL ES framework, which transfers the main rendering work to the GPU, reduces the workload of the CPU.

Finally, the video player framework has been tested, it uses only 9.8 MB ROM space, which is in full compliance with the expectation. The performance of the runtime has been greatly improved by comparison with other methods, and it is also superior or equal to the products from other companies. At present, the framework has passed the acceptance of the project team and is applied to the home school system.

Key words: iOS；RTSP；FFmpeg；Hardware Decode；OpenGL ES

目　录

第一章 绪论

1.1 课题研究背景

本课题来源于国内某安防企业的移动端监控视频播放项目。该项目是要实现一套家校通系统，该系统能更加方便家长与学校的联系，让学生家长能通过手机应用了解到学生在学校的学习和生活情况。

随着经济提升生活水平不断提高，人们对子女的教育问题也投入越来越多的关注，尤其是近些年虐童事件、校园霸凌事件不断曝光，家长更加希望能了解到学生在学校的生活情况。基于以上原因很多公司适时地推出了家校通系统，旨在更好的保证学校的安防和提高家长与学校的联系，项目所在公司的家校通系统如图 1-1 所示。

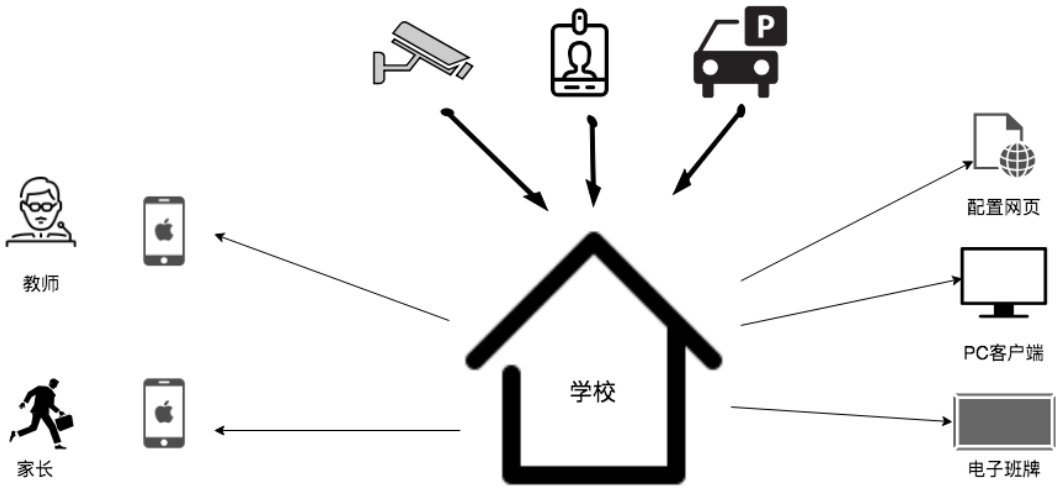

图 1-1 家校通系统示意图

根据图 1-1，家校通系统提供的客户端包含学校管理配置网页、PC 客户端、电子班牌、教师客户端和学生家长客户端。教师客户端和家长客户端是可以在 iOS 平台和安卓平台运行的移动应用。教师端主要是对班级的事务进行管理，包括考勤工作，查看学校开放的视频等，主要的角色是班主任，所以还有与家长沟通的功能。家长客户端主要功能是查看学生在学校的情况，包括学生的考勤状况、作业完成情况、考试情况、学校展开的活动以及观看学校开放的监控视频和联系班主任等。本文通过分析，确定两个 iOS 客户端应用都有视频播放模块，而且基于移动互联网蓬勃发展的现状考虑，今后肯定会有越来越多类似需要播放在线视频的应用，所以提出开发一个性能良好且规模适合的视频播放的静态框架，适应公司监控产品的 RTSP[1]协议和编码格式为 h.264[2]/MPEG-4、ACC 的在线流媒体播放，可以嵌入需要播放监控视频的应用中。

1.2 研究目的及意义

市面上有了很多的家校通系统，不过这些系统相对简单，只有联系老师、查看作业等的功能，与学校安全相关的产品还不是很多，作为一家安防企业，项目所在公司从这个角度出发，既结合了公司推出的摄像头等的硬件产品，也满足了市场的需求。传统的

视频监控，一般都是采用“定点采集定点播放”模式，采集端就是传统的视频摄像头，而播放端就是 PC。这种模式实用性很差，一般的 PC 机形体过大，观看监控视频必须到指定位置，而且成本比较高。移动互联网的蓬勃发展也促进了传统监控领域公司对移动业务的扩展，将监控视频播放画面转移到手机、平板电脑等移动设备上来已经是大势所趋。如何在竞争激烈的移动端获得市场认可，成了移动产品开发的关键。

通过研究发现，大多数企业为了应对繁杂多样的视频编码封装格式，在编解码方面采用的都是软件编程来实现的，这部分工作对 CPU 的消耗极其严重，而且随着视频清晰度的提升消耗增大更加明显。苹果公司在其移动系统 iOS 8 发布时，推出了可以实现硬件解码的框架 VideoToolbox，使用这种方案播放视频可以将大量重复的解码计算转移到 GPU[3]上，降低对 CPU 的消耗。本文在采纳各方意见并考虑公司的内部环境，认为对于项目所在公司流媒体服务器提供的流媒体协议和编码格式相对单一的流媒体播放，有提高播放性能并减小播放组件规模、提高用户体验的空间，可以提升产品的市场竞争力。基于这些原因，本项目计划实现的内容有以下几个方面：

(1) 实现可以对项目所在公司产品生成的 RTSP 流媒体视频进行播放的 iOS 视频播放静态库文件，在类似需要播放视频的 iOS 应用上都可以使用。

(2) 在实现播放的基础上，采用精简的解封装框架，尽量降低静态库文件所占用的 ROM 空间。现在随着移动应用功能的增多，很多应用软件的规模庞大占据过多空间，这与移动产品的轻量要求不相符合。所以减小播放组件的规模，给应用软件的其他部分提供更大的空间，也为应用的安全运行奠定了良好基础。

(3) 在视频解码和视频画面渲染方面，更多的采用专门的硬件模块实现，以减轻 CPU 的计算负担。尽管手机芯片已经发展到了新的高度，但是依然有用户反映在使用手机播放视频（尤其高清视频）时手机发热严重。事实上手机发热严重很大一部分原因就是在视频解码和图像渲染时 CPU 的高负荷使用造成的，如果能降低 CPU 的功耗，能够很好地改善这一现状，提升用户体验。

(4) 研究视频图像截图功能的实现算法，减少截图操作时的性能消耗。截图功能是视频播放时很常见的功能，添加这一功能会让安防产品更加完整，更能提升产品竞争力。

1.3 国内外研究现状

1.3.1 家校通系统的研究现状

2009 年董慧波和 2010 年陈中军分别提出基于 WEB 和基于 FLEX 的家校通系统[4][5]。两者都有创新性地提出了利用互联网连接学校和家长的想法，提出在学校、家长、老师之间建立一个家校互助平台，实现学校、家庭和教室之间的快捷、实时沟通；并且集成在线聊天、新闻管理、用户管理等功能，基本上实现了现代的家校通系统的雏形，为家校通系统的发展奠定了基础。

上述两个家校通系统只是针对 WEB 或者 PC 客户端实现，存在操作不便、不易普及的缺点。并且在移动互联网火爆发展的今天，上述系统在移动设备上不能很好的适配，完全不能满足现代用户的需求。

2011 年陈洁提出家校在线短信平台，构建学校和家庭之间的畅通的短信沟通渠道,方便学生家长与学校的沟通。平台分为 7 个子系统：家长缴费系统、基础信息管理系统、评语管理系统、成绩管理系统、通知管理系统、日常提醒系统和短信管理系统。该平台在实现的功能上进一步向现代的家校通发展，富有创新地使用了短信方式，让家长与学校的沟通更加方便快捷。尽管功能上有所增多，且将沟通方式部分转移到了移动设备；

但是此平台却未提供一个可视化的操作移动操作界面，只是通过短信联系，导致家长的参与度不够高。

1.3.2 iOS 视频监控客户端的研究现状

侯沛德在 2014 年提出了基于 iOS 的视频监控客户端的设计实现[6]，功能包含实时视频观看、信息推送、录像和截屏等功能。并且能实现对采用 h.264 编码和 RTSP 协议封装发送的视频进行播放，在一定程度上优化对监控视频的播放。但是在解码功能的实现上使用的是 FFmpeg 完成，该框架采用的解码方式为软件解码，利用 CPU 计算完成解码数据的转换，会过多地消耗 CPU 等资源，降低设备的性能。并且该客户端只支持 h.264 编码格式的视频解码，解码格式偏少，只适合在特定的场合使用。

1.3.3 安防移动产品研究现状

到目前为止几乎所有的安防行业和智能家居行业都已推出了移动端的产品，比如业内熟知的安防企业海康威视旗下子公司萤石、大华旗下的乐橙和小米等。其中萤石的萤石云视频 App 和乐橙的乐橙 App 是两个类似的产品，针对不同的互联网摄像头提供视频播放服务，这两个 App 也有相同的特点就是功能不多却占据了很大的 ROM 空间，其中萤石云视频 App 使用了高达 115.2MB，乐橙 App 也有 88.2MB。在两个产品中视频播放模块很明显占据过多的空间，其中乐橙 App 在播放视频时偶尔还会出现卡顿的现象。

小米的米家 App 是小米在智能家居方面的产品，可以连接到小米的多种智能家居设备，包括多款智能家居摄像头。但是这款应用的使用的 ROM 空间有 198.43MB，而且小米的摄像头设备多为家居类型，覆盖范围和拍摄的距离远比不上专业的监控摄像头，无法做到对校园的大面积监控。

1.4 研究内容

本文是来自国内某安防公司家校通项目，是为实现在 iOS 客户端播放采用 RTSP 协议传输、编码格式为 h.264/MPEG-4 和 AAC 的在线视频开发的视频播放软件框架。项目对 iOS 移动客户端的定位是轻量级产品，要求 App 产品不得占用过多的手机 ROM 空间。本文对 iOS 平台的流媒体播放进行研究，设计了一套在 iOS 平台可以播放相应的流媒体协议和编码格式视频的播放模块，采用 iOS 的.framework 模式对播放模块进行封装，在是实现播放的基础上对播放器的性能进行了优化，还精简了播放框架使用的 ROM 空间。主要研究内容包括以下 5 个部分：

(1) 对 FFmpeg 框架进行研究，对于在线流媒体的解析剔除除 RTSP 之外的流媒体协议支持，只需要针对 RTSP 码流视频进行解析。分析所有需要解析的视音频编码格式，对于不会遇到的视频编码格式进行精简，只支持项目所在公司的摄像头产品生成的少数几种视频编码格式[7]。

(2) 分析 FFmpeg 解封装之后的视频流和 VideoToolbox 硬件解码框架，对于可以采用硬件解码的视频流采用 VideoToolbox 实现，不支持硬件解码的视频流和音频流统一采用 FFmpeg 进行解码，尽可能提高播放器的性能。

(3) 分析 OpenGL（Open Graphics Library，开源图形库）和 OpenAL（Open Audio Library，开源音频库）框架在 iOS 平台的使用，尤其是在视频渲染方面如何能提升 GPU 的利用率，可以减轻 CPU 的功耗，而且在视频播画面的体验上更有优势，进一步提升

播放器的性能。

(4) 研究分析视频截图功能，对比系统截图和将解码后的 YUV 数据转化成图像数据的差异，并对 YUV 转 RGB[8][9]图像格式的算法进行探究，优化截图模块对 CPU 的使用。

(5) 研究分析在播放器静态库封装时采用的方式、公开头文件的多少和类型、播放事件的传递和响应方式，以减轻播放器在后续使用的时候复杂度，方便后续应用的开发。

1.5 论文组织结构

本文一共包含七章，文章的组织结构的安排是：

第一章绪论，首先是阐述了课题的研究背景，接下来分析了当前项目的研究目的及意义，然后根据国内外相关产品，分析了国内外对于监控系统所需要的播放器的研究现状，最后阐述了本文的组织安排。

第二章是介绍项目开发所需的关键技术，包含了流媒体传输技术、iOS 视频播放实现方法、视频解码和视频渲染。。

第三章从播放框架和视频播放两个方面对本文进行了需求分析，其中框架需求是针对开发者接入时的接口需求包括采用 Objective-C 设计、能处理紧急事务和采用最新的 iSO 系统接口；视频播放需求是指与视频播放相关的功能需求和性能的需求。

第四章是论文的概要设计，从课题所属项目到播放器框架架构设计；接着介绍了论文的分层思路，表示层指的是接入开发者能接触到的部分，业务层指的是播放框架的内部逻辑层，也就是核心模块层包括：RTSP 流媒体解析、视音频解码、视频截图、音画同步、视音频渲染和音频播放等功能。

第五章是详细设计与实现，对概要设计的内容一一展开实现，同样自顶向下展开，首先是播放器框架的表示层主要包含：视频的图像显示界面、框架的封装方式、公开的头文件以及播放相关的事件处理方法；核心模块包括：解封装模块、解码模块、图像渲染模块、YUV 转 RGB 格式图像模块、音频渲染模块、音画同步模块等。

第六章测试与分析，针对项目在对于播放器框架的需求、播放器的功能和性能需求逐步进行测试，分析测试结果可知播放器框架在视频播放功能、占用的 ROM 空间、解码渲染性能、流畅性、稳定性和可扩展性都符合项目需求。

第七章总结与展望，总结整个项目的实现和所得，并分析项目中遇到的问题和有待改进的地方，对未来的工作提出了展望。

第二章 相关技术

本章将针对项目的研究内容，对开发过程中用到关键技术进行介绍，包括流媒体传输技术、iOS 视频播放实现方法、视频解码和视频渲染。

2. 1 流媒体传输技术

视频采集和播放系统一般都采用的是“视频采集->编码->推流->拉流->解码->播放”流程实现的。目前视频推流所采取的流媒体协议主要有 RTMP、HLS、RTSP。

RTMP[10]（Real Time Messaging Protocol,实时消息传输协议）是奥多比公司推出的一套实时数据通信的协议，在当前直播和监控系统采用比较多[11]。但是 iOS 移动端浏览器和原生 API 由于不支持 Flash，所以需要通过第三方框架完成解析播放[12]，视频首次打开时会有 3-5 秒的延时，所以此协议没有被采用。

HLS[13](Http Live Streaming,动态码率自适应技术)是苹果推出的基于 http 的流媒体传输协议。因为 HLS 协议的切片传输的方式，导致播放视频的时候存在较大的延时，因此并不符合被需要实时性条件的直播和视频监控等产业使用。

RTSP[1][14]是由真实网络公司和网景共同提出的。由于该协议的延时很小，而且该协议可以扩展，能更好地使用在实时信息传播领域，正在被越来越多的使用。

2. 2 iOS 视频播放实现方法

移动端作为视频播放端现在在业界主要有两种解决方式：一是采用原生 App 方式播放，即每个服务提供商都定制一款 App，打开 App 后可以观看在线传输的视频流。二是采用 Html5[15][16]页面播放，没有原生的客户端，可以利用浏览器或者微信客户端播放视频。对于 App 播放的方式，能对视频播放提供更多的控制功能，但是成本相对较高。而 Html5 页面，成本不高且轻便易于开发和修改，缺点是受不同浏览器的限制、不容易适配，播放的视频不容易控制。

采用 App 播放在线视频的目前常见的方案有三种：

(1)早期使用的是 Live555[17]与 FFmpeg[18]（Fast Forward MPEG，快速视音频转换器）搭配完成在线视频播放的。Live555 是一个相当知名的专为流媒体传输的提供跨平台框架[19]，可以解析在线流媒体将流媒体解封装为视频流和音频流。解封装之后的解码播放由 FFmpeg 框架完成。此种方案比较因为太繁杂，现在几乎很少使用。

(2)FFmpeg 框架从 3.0 版本开始支持解析 RTSP 流媒体[20]，之后绝大部分的视频播放器都是选取此框架完成的。尽管 FFmpeg 提供了很多功能，几乎支持所有格式的视频的解码播放，但是因为它是一套跨平台播放器框架，为了适用多种软硬件平台，其对视音频的解码都是调用 CPU 完成的，在播放高分辨率的视频的时候偶尔会有掉帧、卡顿的情况。

(3)采用 MobileVLCKit 视频框架完成视频播放。MobileVLCKit 同样是跨平台视频解决方案，大量的多媒体封装格式和编码格式都可以支持。但是这款播放器框架内部是封装了 Live555 和 FFmpeg，是基于这两个 C 语言静态库完成的 Objective-C 语言的播放器组件。所以同样存在 FFmpeg 软解码的问题，而且封装完成后的静态库有 20MB 左右

大小，这在移动应用中略显庞大。

2.3 视频解码

为了便于存储和传播，市面上的视频几乎都做了编码压缩处理，h.264 和 MPEG-4 是两种视频编码格式，AAC 是音频编码格式。要播放压缩编码的视频，必须首先给编码的视频文件解码。目前在业界使用的解码方式包括软件解码和硬件解码[21]，软件解码指的是通过 CPU 运算完成视频数据转换解码；硬件解码指的是通过显卡的视频加速功能或者其他专门硬件对视频进行解码运算，因此 CPU 可以从繁重的视频解码中释放出来。而且显卡的 GPU[22]相对 CPU 更适合数据量庞大的、低难度的重复工作，所以与软件解码相比，硬件解码性能更有优势。苹果公司的移动操作系统从 iOS 8 推出了 VideoToolbox 框架，开始了对硬件解码的支持，而 VideToolbox 支持硬件解码的视频编码格式为 h.264，正好是项目需要支持的解码格式之一。考虑 iOS 系统的更新迅速，到项目启动时 iOS 8 及以后的系统，已经占据了超过 90%的苹果移动设备，所以对于 h.264 编码格式的视频，项目中采用的解码方式是硬件解码，MPEG-4 和 AAC 的解码方式则采用的是 FFmpeg 框架。针对节省 CPU 消耗的需求，项目在视频图像渲染方面采用 OpenGL ES 实现，为了便于 OpenGL ES 渲染图像画面，项目中将视频画面数据转化成了 YUV 格式的视频数据。下面将对 FFmpeg 框架的主要组成部分和数据结构与 YUV 数据编码方式进行介绍。

1、FFmpeg

FFmpeg 是知名的视音频的框架，集成了视音频的录制、多种编码格式转换、多种视音频格式编解码等功能，其最大的特点是跨平台性，几乎可以在各个常见的平台（Linux、Windows、macOS、BSDs、Solaris、Android、iOS 等）和架构（x86、ARM、MIPS 等）上编译运行。除了跨平台的可移植性，FFmpeg 还专门针对 x86，ARM，MIPS 等大多数主流的处理器提供了对应的汇编级别的优化实现，在提高性能上做了很多工作。此外 FFmpeg 是一个模块化的结构，具有高度的可扩展性，可以从官网下载源码，实现自定义的编解码器或者解封装等功能并注册到 FFmpeg 当中去。

FFmpeg 的主要组成部分包括 6 个部分，包括：libavformat、libavcodec、libavfilter、libswscale、libresample、libavutil。

(1) libavformat 中实现了目前多媒体领域中的几乎所有封装格式，可以进行封装和解封装操作。FFmpeg 支持的封装格式取决于编译后的这个库文件，例如 mp4、flv、mkv 等容器的封装与解封装和 RTMP、RTSP 等协议的封装与解封装；封装与解封装不涉及到复杂的计算，更多的是 I/O 操作；并且 FFmpeg 支持在 libavformat 中增加自定义的 format 模块定制的封装格式。

(2) libavcodec 中实现了包括音频和视频的几乎所有的编解码格式。不过有些专有的编码需要使用特定的编码器，例如 H.264（AVC）编码需要使用 x264 编码器，H.265[20]（HEVC）编码需要使用 x265 编码器，MP3（MP3LAME）编码需要使用 LIBMP3LAME 编码器等编码器。FFmpeg 本身同时也支持多种编码格式，例如 MPEG4、AAC、MJPEG 等编码，当然如果想用自定义的编码格式，或者硬件编解码，可以在 libavcodec 中增加自定义的编解码模块。

(3) libavfilter 库提供了一个通用的音频视频滤镜框架，处理一些滤镜相关的操作，不过目前在项目中用不到这个模块。

(4) libswscale 模块提供了高级别的图像转换接口，可以实现图像缩放和像素格式转换。常见于将图像从 1080p 转换成 720p 或者 480p 等的缩放，或者将图像数据从 yuv420p 转换成 yuyv，项目中在获取 YUV 数据的时候将会用到这个模块。

(5) libresample 模块提供了高级别的音频重采样接口。通过这个模块可以实现音频

采样、音频通道布局 rematrixing 和转换的音频格式和包装布局。通常在 iOS 设备上无法直接播放 FFmpeg 解码出来的 PCM 格式的音频，需要通过这个框架进行重新采样才能解决。

(6) libavutil 主要定义 FFmpeg 中的 AVFormats，AVCodecs，AVFilters 等所用到的公用的接口，也提供很多必要的工具函数比如时间相关的工具、设备相关的工具等，属于工具模块。

FFmpeg 的关键数据结构以及相互间的关系如图 2-1 所示。

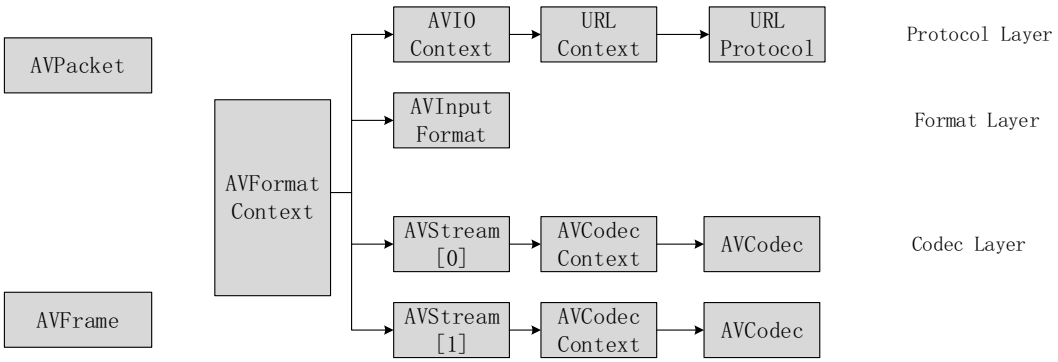

图 2-1 FFmpeg 关键数据结构

AVPacket：定义在 avcodec.h 中主要是用来充当容器，其功能是用来暂存解封装之后的视音频数据，它的数据成员其实是指向实际的数据缓冲区，在分别获取音频流和视频流的过程中,需要用到此模块。

AVFrame 结构体的主要功能是存储原始数据即项目中的视频 YUV 数据和音频 PCM 数据。此外还包含了一些相关的信息，比如，解码的时候存储了宏块类型表，运动矢量表等数据。

AVStream：结构主要是用来存放与流媒体相关的编解码器之类的信息，解码之后的数据就是由这个结构存储。它包含经常被用到的数据包括：将要提到的用来保存编码或者解码的结构 AVCodecContext，和与编解码相关的数据。

AVFormatContext：这是 FFmpeg 中的基本结构，其他结构都是由这个结构扩展而来。它描述的是一个多媒体文件或流媒体构成的基本信息。

AVInputFormat：存储输入视音频使用的封装格式，每种视音频封装格式都对应一个 AVInputFormat 结构。

AVCodecContext：这个结构指的是当前编解码器上下文环境，它包含了不同编解码器编解码时需要使用的参数。如果只是使用 FFmpeg 框架中的解码库 libavcodec，这个结构体需要调用之前开发者手动初始化，如果是需要使用到其他功能，需要在打开文件或者流媒体是完成初始化。

AVCodec 结构就是表示编码解码信息的结构体，包括解码编码的函数指针等信息。结构体中有一个数据成员是 CodecID，指的是解码器的类型，在项目是用解码器的类型来判断流媒体的编码类型。AVCodecContext 中有一个 AVCodec 类型的数据结构成员，编解码都是使用这个结构。

AVIOContext，URLProtocol，URLContext 主要存储视音频使用的协议的类型以及状态。URLProtocol 存储输入视音频使用的封装格式，每种协议都对应一个 URLProtocol 结构[23]，这部分在项目中没有直接用到。

2、YUV 数据编码方式

YUV 是一种电视系统使用的颜色编码格式，用此种方式来表示颜色编码比 RGB 采用极少的频宽，在数据量巨大的视频播放上是很有优势。YUV 三个数据值的范围都是

0~255，其中 Y 指的是当前点的明亮度；U 和 V 都指的是色度，用来描述当前点的色彩和饱和度。通常情况下，如果只有 Y 数据，渲染出来的画面就是黑白的，老式黑白电视机播放的电视就是只能显示了 Y 分量的信号。一个像素点的颜色值需要由三个分量值的结合才能完整体现，通常情况下 U 和 V 分量都是成对存在，但是一帧画面中 YUV 的数据量可能不一致，这与 YUV 的存储格式也就是采样格式有关，主流的采样方式有三种，yuv444，yuv422，yuv420。三种采样格式的数据分布方式如下所述。

- yuv444 采样，每一个 Y 与一组 UV 对应。
- yuv422 采样，每两个 Y 与一组 UV 对应。
- yuv420 采样，每四个 Y 与一组 UV 对应。

显而易见从 yuv444 到 yuv420 数据的存储量是递减的，鉴于节省内存空间的需求，项目中在播放器渲染时采用的采样格式就是 yuv420 的一种形式 yuv420p，在内存中的对齐方式大概如图 2-2 所示。

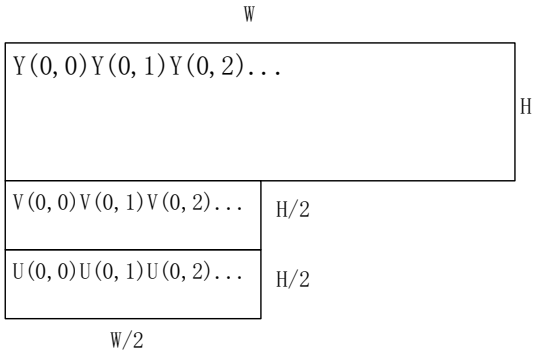

图 2-2 yuv420p 图像在内存中的存储

图 2-2 中 W 和 H 分别指这一帧图像的像素宽和高，也就是在水平方向和垂直方向上的像素数量。Y 分量分别与每一个像素点对应，而 U 和 V 分量分别与四个像素点对应。比如图中 Y(0,0)，Y(0,1)，Y(1,0)，Y(1,1)分别与 U(0,0)和 V(0,0)组成四组 YUV 数据。

2.4 视频渲染

视频画面渲染指的是将视频原始数据转化成图像帧序列并逐帧显示的过程。OpenGL[24]指的是一个专业的主要由 C++实现的图形程序接口，可以做到跨编程语言、跨平台编程，最常见的功能是对三维图像的绘制。OpenGL ES[25]是 OpenGL 在移动平台上的分支，属于简化版的 OpenGL，用于 OpenGL 在移动设备上的图像渲染工作。

为了改变单一渲染、不可操作的弊端，OpenGL 2.0 推出了 GLSL[26][27](OpenGL Shading Language，着色器语言)，引入着色器语言之后开发者可以编写简短的可以在显卡上执行的程序，替换了固定的渲染管线的一部分，让渲染过程中有了一定的可编程性。着色器语言采取的是类似 C 语言的语法，容易阅读和编写，项目中将要用到的是顶点着色器与片段着色器。顶点着色由顶点着色器负责，它能够得到当前图形框架中的各种状态，可以使用 GLSL 中定义的变量把值传到片段着色器或者 GPU 中。利用代码来控制纹理的提取和处理复杂的动画等操作则是在片段着色器中完成的，承载的更多图像效果的实现。

OpenGL ES 无法之间对整幅图像进行绘制渲染，绘制图像画面都是由一个个图元组成的，可以直接绘制的图元只有点、线段和三角形三种，显然对于图像显示最实用的方法是使用三角形完成。也就是将利用两个直角三角形拼接成一个矩形，矩形的四个顶点就是两个三角形的顶点。OpenGL ES 绘制三角形的方法有三种，分别为单个三角形绘制——GL_TRIANGLES、扇形三角形绘制——GL_TRIANGLE_FAN 和条形三角形绘制——GL_TRIANGLE_STRIP。第一种方法就是每一个三角形需要三个顶点数据来表示，而后两种方法中做到了对顶点数据的共用，在节省内存空间上有很大优势，三者的绘制三角形的方法如图 2-3 所示。

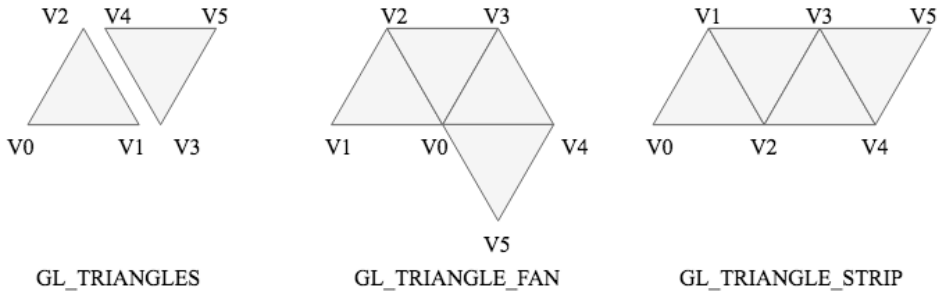

图 2-3 OpenGL 三角形绘制的三种方法

从图 2-3 GL_TRIANGLE_FAN 的绘制方式中可见，所有的三角形都共用了第一个顶点 V0，并且相邻两个三角形还共用另外一个顶点，此种方法在绘制扇形类的图形时比较方便，如果要在视频图像渲染时，需要经过多次的转换，不适合采用。关于第三种绘制方法 GL_TRIANGLE_STRIP，尽管相邻的两个三角形都共用了两个顶点，但是相邻两个三角形的绘制方向却略有差异；比如第 n（n>2）个顶点，n 如果为奇数，则三角形三个顶点为[n-1,n-2,n];n 如果为偶数，三个顶点为[n-2,n-1,n]。采用此种绘制方法的是为了保证所有的三角形都是按照同样的方向绘制的，比如都符合右手系或者都符合左手系，这样的三角形串才能正确地形成一个表面，项目中采用的绘制方法就是 GL_TRIANGLE_STRIP。

2.5 本章小结

本章从流媒体传输技术、iOS 视频播放实现方法、视频解码、视频渲染几个方面介绍了论文实现部分用到的关键技术，为后续项目实现提供了技术基础。

第三章 需求分析

论文前两章介绍了课题所在的实际项目、论文相关内容的国内外研究现状和论文实现所需要的一些基础知识。本章从播放器框架需求和视频播放两个方面对项目的需求进行分析。

根据前文的内容可知，本项目播放器的设计主要有三个目标：第一个是实现对 RTSP 流媒体协议的播放；第二个是在保证播放器性能的条件下尽量减小播放器静态库的规模，以方便在需要播放在线视频的应用中使用；第三点是提高播放器的性能，提高产品的市场竞争力。

3.1 播放框架需求分析

3.1.1 播放框架与外界交互

(1) 接口封装语言

因为项目中两个 iOS 客户端都是采用 Objective-C 语言开发的，为了方便后续播放框架的接入，要求在接口封装上使用 Objective-C 实现并且符合 Objective-C 的调用习惯。

(2) 对突发事件的处理

播放框架需要对突发事件进行处理，比如正在播放视频时用户按下 Home 键应用进入后台，框架需要对当前播放状态进行暂停处理。

(3) 采用最新的系统接口

iOS 系统基于安全性和与提高性能等方面考虑，总是在逐步增加一些系统接口。项目从安全性和兼容性考虑，要求在播放框架中需要使用 iOS 8 的最新接口实现。

3.2 视频播放需求分析

3.2.1 视频播放功能

本播放器的功能是播放在线流媒体视频，主要是直播或者点播公司摄像头采集的视频。在线播放器一般包含的功能包括：

(1)视频播放功能

需要实现对给定 RTSP 地址的流媒体进行流畅播放。通常 RTSP 流媒体视频地址是以“rtsp://”开头，对于非 RTSP 流媒体视频或者无法播放的视频地址给出对应的无法播放的提示。

(2)暂停播放功能

正常播放的视频暂停播放是播放器的必备功能，对点播视频尤为重要，所以本播放器必须能够提供暂停的功能。

(3)继续播放功能

已经暂停的视频需要有继续播放的功能，视频点播与视频直播的继续播放略有区别，点播的继续播放是按暂停时的进度播放，直播的继续播放是立即从当前时间刻度开始播

放，会跳过暂停时间的内容。

(4)点播视频定点播放

定点播放指的是用户在视频播放进度条上对播放位置进行操作，当移动到指定位置时播放器能够自动跳到相应的时间刻度进行播放。不过此功能只针对点播视频，直播视频不需要定点播放。

(5)点播视频快进/快退功能

与视频定点播放类似，快进/快退功能就是视频能从当前的播放时间刻度向前或者向后移动一个时间间隔，无论是在播放状态或者暂停状态，快进和快退功能都有效果，此功能也只针对点播视频，直播视频不需要实现该功能。

(6)支持特定编码格式的视音频解码

目前公司的摄像头等采集端的视频编码格式有两种一种是 h. 264[28]，另外一种是 MPEG-4[29]，音频的编码格式为 AAC。播放器需要针对上述编码格式的视音频流媒体进行解码播放。

(7)实时截图功能

实时截图功能是指在视频播放过程中可以实时获取播放画面中的任一时刻的图像，并可将图像作为照片文件保存在相册中。

3. 2. 2 视频播放性能

为了获得更好的市场认可，视频播放器在性能方面的用户体验也是着重考虑的问题。当前市面上移动设备播放器的主要的性能问题是解码和图像渲染过程中出现的机身过热、画面卡顿和异常掉帧等情况，以下分别对这些情况展开分析。

(1) 播放器占用 ROM 空间

在移动应用市场上应用软件的规模一直是厂商所看重的内容之一。播放器作为移动应用的一部分，希望将播放框架限制在 10MB 以内，占用较少的设备的 ROM 空间。

(2) 解码性能

由于科技的发展和网络带宽的提升，在线视频的清晰度都到了较高的水平，随之而来的是每一帧视频所需要解码的数据量的急剧上升。而很多播放器是通过 CPU 运算解码的，CPU 的过量使用导致手机机身发热和播放画面卡顿等情况出现。基于这些原因，项目中希望对于 h.264 格式视频解码解码效率要比传统方法提高 50%。

(3) 视频渲染性能

同解码性能一样，目前很多播放器采用系统自带控件实现视频图像的渲染，此种方式对内存和 CPU 造成巨大消耗，这也是造成了视频画面卡顿和异常掉帧等情况的原因之一。对于视频渲染性能，项目中希望能在传统的基础之上提升 30%。

(4) 截图性能

截图是与播放不相关的功能，并且有多种实现方式，为了在视频播放过程中不影响播放的性能并达到很好的截图效果，项目中希望通过算法优化，将截图时带来的性能消耗降低 50%。

(5) 视频播放流畅性

现如今移动互联网技术有了很大的发展，已经完全可以通过移动网络播放高清视频了，项目中要求播放框架至少能够播放清晰度为 720P 的视频流，播放过程中无卡顿、音画不同步的情况出现，图像的刷新率保持在 56 FPS 以上。

(6) 播放器的稳定性

软件系统的稳定是健全软件必备的目标，项目要求播放框架在播放视频时不可出现视频播放死机、黑屏等现象。

(7) 可扩展性

随着 VR[30]技术的发展，拍摄 3D 画面的摄像头的增多，后续可能会有 3D 视频播放的需求，所以播放器需要有针对 3D 视频播放的可扩展性，方便增加对 3D 视频播放的实现。

3.3 本章小结

本章从两个方面对本文进行了需求分析，一是播放器框架层面的需求，包括采用的开发语言、要求有完整的突发事件处理机制和采用最新 iOS 8 的系统接口完成开发；二是视频播放相关的功能需求和性能需求，功能需求包含一个完整视频播放器所需要实现的功能，性能需求是在实现播放的基础之上从视频解码和图像渲染方面提高产品的性能。

第四章 概要设计

第三章的内容是对项目需求从软件框架和视频播放两个方面进行了分析，本章按照播放器框架架构设计、表示层设计和业务层逐步介绍播放器框架的该要设计。

4.1 播放器框架架构设计

4.1.1 播放器所属项目架构

根据前文可知，本课题是属于家校通项目中的一部分，家校通项目的系统架构设计如图 4-1 所示。

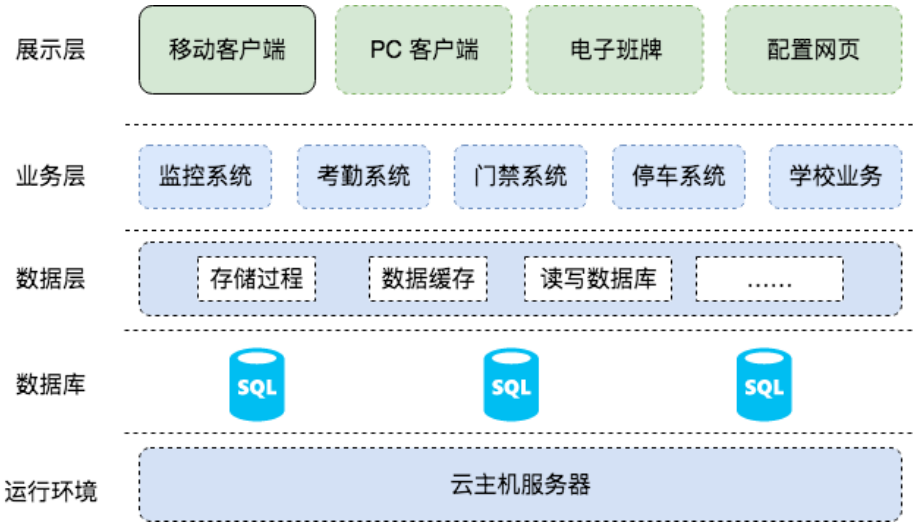

图 4-1 家校通系统架构

从图 4-1 可以看出，整个项目可以分为 5 层，第一层是展示层，主要的实现包括移动客户端、PC 客户端、电子班牌和配置网页等。移动客户端包括两个 iOS 端和两个 Android 端，论文是 iOS 端的一个模块。业务层包括监控系统、考勤系统、门禁系统、停车系统和学校业务，前四项是公司成熟的产品只需要进行相关的配置就可使用；最后的学校业务是项目中业务层开发的重点，需要根据学校的场景进行重新开发。整个业务层使用微服务架构实现，不同业务由不同的服务完成，不同的服务部署在不同的主机上，各服务之间没有依赖，更加方便开发调试和部署维护。下面三层是相对底层的设计实现，包括数据层、数据库和运行环境，数据层的功能配合业务层的数据读取实现存储过程、数据缓存、读写数据库等功能，系统的运行环境是子公司的云主机服务器。

4.1.2 播放器框架架构设计

本文是要实现一款适用于 iOS 平台的在线视频播放框架，框架的示意图如图 4-2 所示。

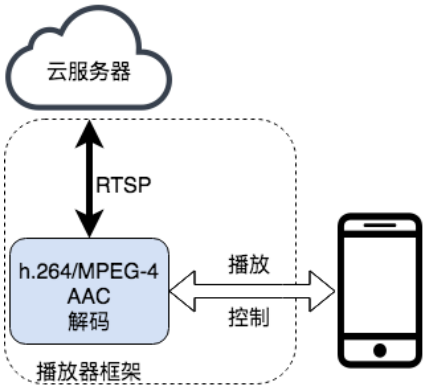

图 4-2 播放器框架示意图

从图 4-2 中可以看出，播放器需要实现对 RTSP 流媒体的获取解析，并完成对 h.264/MPEG-4 和 AAC 编码格式的视音频解码播放。对于播放器的架构项目中采用分层设计的结构将播放器分为两层，如图 4-3 所示。

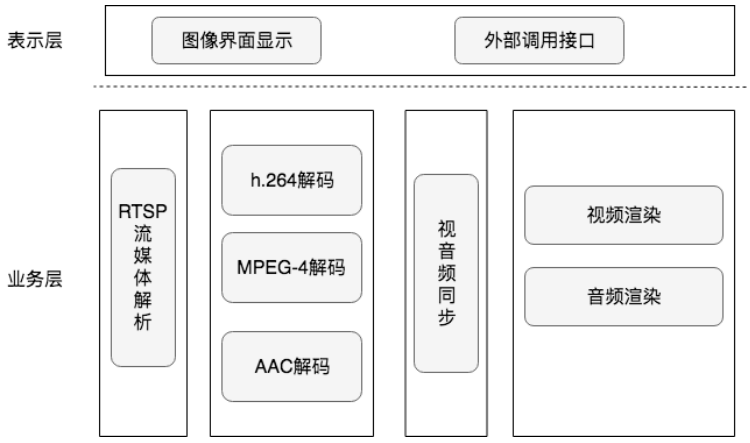

图 4-3 播放器分层设计

从图 4-3 中可以看出顶层为表示层，采用 Objective-C 语言设计，主要包含的是视频图像的显示和外部调用接口的相关内容，开发用户直接接触到的是表示层。底层为播放器的业务层，是播放器的内部实现，这一层对应用开发者是透明的，主要包括 RTSP 流媒体的解析、h.264/MPEG-4 和 AAC 格式的视音频解码、视音频同步和视音频渲染等模块。

4.1.3 播放器框架的使用流程

播放器最终会封装成 iOS 开发专用的静态库提供给需要实现对应流媒体协议和编码格式的移动应用使用，使用流程如图 4-4 所示。

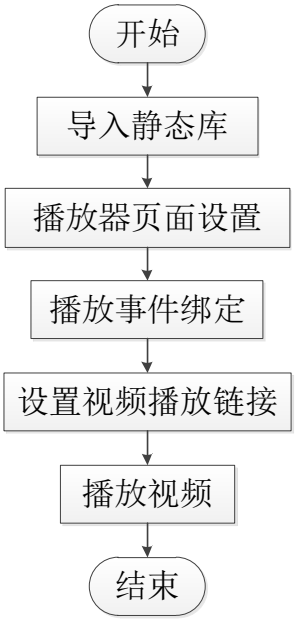

图 4-4 播放器框架的使用流程

从图 4-4 中可以看出，播放器框架简化了接入的开发者的工作流程，所有播放相关事务都是在播放器内部实现的，需要外部处理的事务（比如播放链接失效等）都是通过事件绑定通知到开发者用户。

4.2 表示层设计

为了提高播放器静态库框架的可用性和减少后续开发的复杂度，播放器需要对底层的处理进行了封装，只暴露少量的文件给二次开发的开发者用户，播放器表示层由四个部分组成如图 4-5 所示。

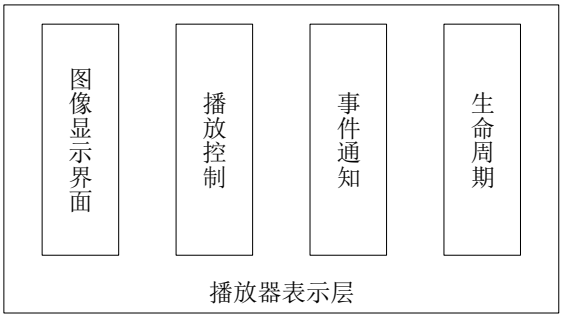

图 4-5 播放器表示层

从图 4-5 中可以看出，组成播放器表示层的四个部分包括图像显示界面、播放控制、事件通知和播放器的生命周期。图像显示界面是一个 iOS 界面控件，提供画面显示部分，这是播放器框架提供的唯一控件，其他部分统一提供相关事件的通知，界面显示部分需要开发者自行实现；播放控制主要是播放、暂停、播放定位、快进和快退相关的播放相关的控制事件；事件通知指的是在播放器内部产生的事件通知给开发者，比如播放器状

态改变、解码和渲染时出现问题或者播放链接错误等；播放器的生命周期指的是播放器对象的新建、初始化和回收。

4.3 业务层设计

4.3.1 RTSP 流媒体的解析

视频和音频的录制和压缩时都是分开进行的，压缩完成后会得到音频和视频两组数据流，为了存储和传输方便通常都会用某种方式将两者合并封装成一个文件或者文件流，这就有了多种封装格式。项目中的解封装模块的解析流程如图 4-6 所示。

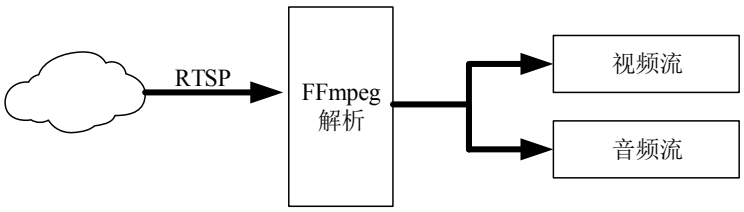

图 4-6 FFmpeg 解析流程

由图 4-6 可以看出，解封装的流程是首先从云服务器中获取 RTSP 流媒体资源，接下来通过 FFmpeg 框架对对 RTSP 流媒体进行解析，最终分别获得了视频流和音频流数据。

4.3.2 视音频解码

根据需求分析可知，播放器需要解码的视音频格式单一，视频有 h.264 和 MPEG-4 两种，音频为 AAC[31]格式。在视频播放器中解码是性能消耗最大也是最有提升空间的部分，所以在视频解码方面的优化是论文着重考虑的问题之一，项目中针对这几种编码格式的解码步骤如图 4-7 所示。

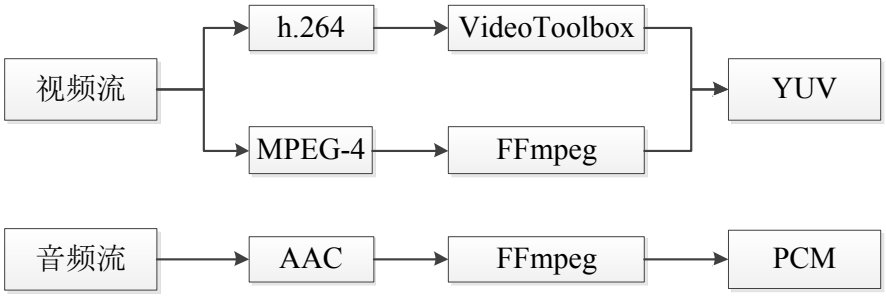

图 4-7 视音频解码步骤

从图 4-7 中可知，对于 h.264 格式编码的视频采用的解码方式 VideoToolbox，MPEG-4 编码的视频和 AAC 编码的音频采用的解码方式是 FFmpeg 框架。因为 h.264 在 iOS 8 系统已经得到了硬件解码的支持，在性能上可以得到很大的提高。而且苹果的移动新系统普及的速度非常快，iOS 8 及之后的系统所占用的设备已经超过了 90%，据此论文提出只针对 iOS 8 之后的设备进行适配，如果检测是老旧的系统，提示开发者播放框架不予支持。根据图像渲染方面的要求，最终视频解码生成 YUV[32]图像数据，音频解码形成 PCM 数据。

4.3.3 视频图像渲染

图像数据要显示在屏幕上，单帧画面采用最简单的方法是使用 iOS 提供的系统控件 UIImageView。但是因为 UIImageView 属于顶层控件，会消耗大量资源，视频本身就是大量连续图像组成，使用 UIImageView 显示会给设备带来更大的负担。近期 CPU 的性能提升缓慢，但是移动端的 GPU 却有了大幅更新，而且与 CPU 相比，GPU 的利用率却不高，在一个没有动画的页面上大部分时间是闲置状态。基于这些原因，本文在视频图像渲染上采用 GPU 的方式实现，提高 GPU 的利用率，同时减轻 CPU 的消耗。渲染过程如图 4-8 所示。

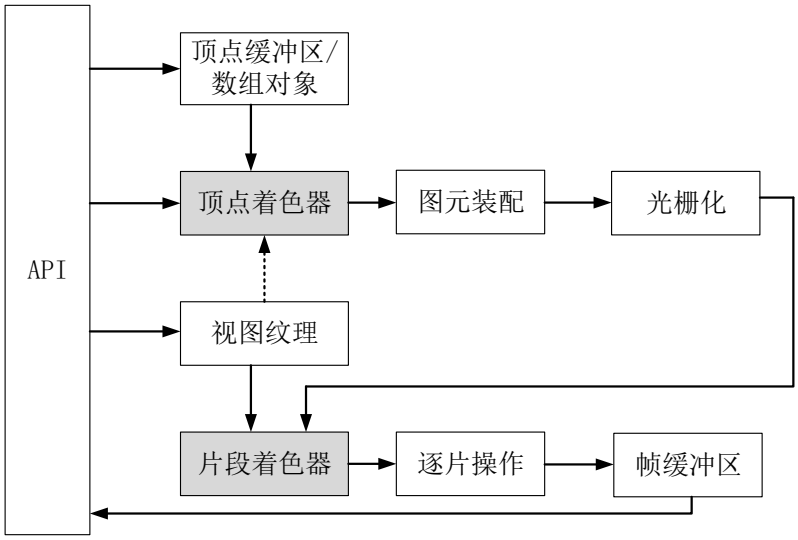

图 4-8 OpenGL ES 2.0 渲染过程

图 4-8 中，顶点着色器和片段着色器就是在着色器语言引入的对象，顶点数据是在代码中设定的，顶点与视图的纹理组合，完成视图的渲染。

4.3.4 截图功能

播放图像的截图功能的有两种实现方案，第一种是采用 iOS 系统提供的截图接口，对指定区域进行截图存储；第二种是将当前显示帧的图像数据转化成图片格式的文件进行存储。考虑到不同的屏幕尺寸的机型采用系统截图所得到的图片尺寸不一致，并且因为项目中选择了解码生成的图像数据格式为 YUV，所以对于此功能的实现采用的是第二种方案，即实时将当前显示帧的 YUV 数据直接转换成 RGB 图像格式存储在手机的相册中。

4.3.5 音频播放

为了与视频渲染对应，也基于播放器对 3D 场景播放扩展的考虑，项目中音频播放部分采用的是 OpenAL 框架，而且 FFmpeg 解码出来的 PCM 音频数据，采用 OpenAL 播放要比系统框架方便得多。OpenAL 原为 3D 场景设计的跨平台音频播放库，在游戏和 3D 视频中经常被采用，也可以在 2D 场景实现音频的播放。

4.3.6 音画同步

根据前文的内容可知，音频跟视频是两组不同的数据，但是在视音频播放时两者又必须是同步进行，否则会带来较差的用户体验。因为视频数据和音频数据中都有与时间相关的参数，所以关于音画同步的设计有三个方案：

第一种是以视频的时间为主，音频播放向视频同步；

第二种是以音频的时间为主，视频播放向音频同步；

第三种是按照系统时间为主，视音频的播放都向系统时间同步。

鉴于在正常情况下视频一秒钟内播放 24-60 帧画面人眼是无法区分的，但是人类的耳朵对声音的敏感性比眼睛对画面要高，而且解压后音频流中有 audio_clock 结构，用来指示声音的播放时长，所以项目中采取的音画同步的方案是上面提到的第二种——视频向音频同步，视频播放较快则延迟视频播放，视频播放较慢则主动丢帧到与音频对应的画面。不过很多监控摄像头不会对声音进行采集，没有声音的流媒体，不需要同步，直接按照视频的时钟进行播放。

4.4 本章小结

本章是论文的概要设计，首先介绍了顶层的播放器框架的所在项目的架构设计，接着介绍了播放器框架的架构设计，最后分别介绍了播放框架的表示层设计和业务层设计的主要内容，第五章将根据该要设计内容对整个播放器框架进行详细设计。

第五章 详细设计与实现

第四章的内容是阐述了播放器框架的概要设计，本章的内容是根据概要设计内容对播放器框架的详细设计与实现进行说明。

5.1 表示层

5.1.1 图像显示界面

视频画面显示的界面是本次播放器框架中提供的唯一控件，为了减轻接入开发者的复杂性，图像显示采用的是常见的 UIView，更符合 iOS 应用开发的习惯。采用 UIView 作为图像显示界面，开发者可以在 App 的控制器中的界面上直接添加就可以实现播放。前面提到采用的图像渲染的方式是 OpenGL ES，OpenGL ES 无法与 iOS 的原生控件交互，论文中采取的方法是把 UIView 的绘制层的 layer 转换成了支持 OpenGL ES 的 CAEAGLLayer，并将绘制层的_eaglLayer 设置为不透明的，这样不用绘制不可见的内容就会减轻计算的压力。

5.1.2 播放器框架的封装

iOS 开发中同时包含静态库和动态库，其中静态库的后缀包括.a 和.framework，动态库的后缀包括.tbd 和.framework。静态库链接时会被完整地复制到可执行文件中，被多次使用就有多份拷贝。而动态库链接时不复制，程序运行时由系统动态加载到内存，供程序调用。而且系统只加载一次，多个程序共用以节省内存。因为苹果官方应用商店审核很严格，使用动态库的应用很有可能无法在应用商店发布。因为播放器的使用场景很少需要在同一个界面上播放多个视频，所以很少出现同时多次调用静态库的情况，而且为何避免不必要的问题产生，项目中选择生成静态库文件，提供给公司内部所有要在 iOS 平台播放 RTSP 流媒体视频的客户端使用。后缀为.a 的静态库与后缀为.framework 的静态库的区别是.a 的静态库的不包含公开头文件，公开头文件需要另行提供，而.framework 静态库内部有一个名为 Headers 的目录，里面包含的是静态库公开的头文件。为了方便开发和发布，项目选择生成后缀为.framework 的静态库文件。

iOS 的静态库文件包括真机 Debug 版、真机 Release 版、模拟器 Debug 版。其中 Debug 版就是在调试时候使用，含完整的符号信息方便调试，Release 版是打包发行的版本，不会包含完整的符号信息，执行代码是进行过优化的，库文件大小会比 Debug 版本的略小，在执行速度方面会有优势。要分为模拟器版本和真机版本的原因是模拟器的处理器与真机设备处理器不一致；模拟器的处理器版本与当前所在的电脑的处理器架构版本相同一般为 i386 或者 x86_64，而真机设备处理器架构一般采用的为 ARM 系列。一般情况下在公司内部为了方便开发和调试，四种版本的静态文件都是公开的，所以这里会生成四种静态库文件。

5.1.3 公开的头文件

为了简化后续的开发工作，本文尽量减少了公开的头文件数量，选择公开的头文件包括 HKLivePlayer. h、HKLivePlayerImp. h、HKLivePlayerAction. h。其中 HKLivePlayer. h 属于整个播放器的外壳，内部包含所有需要引入的文件，本身没有播放的功能，真正播放相关的内容包含在 HKLivePlayerImp. h 中，开发的时候并不需要引入此文件，一次引入 HKLivePlayer. h 即可；HKLivePlayerAction. h 主要用来声明和处理与播放时相关的事件，比如播放状态修改通知，播放时间修改通知等，相关公开头文件如图 5-1 所示。

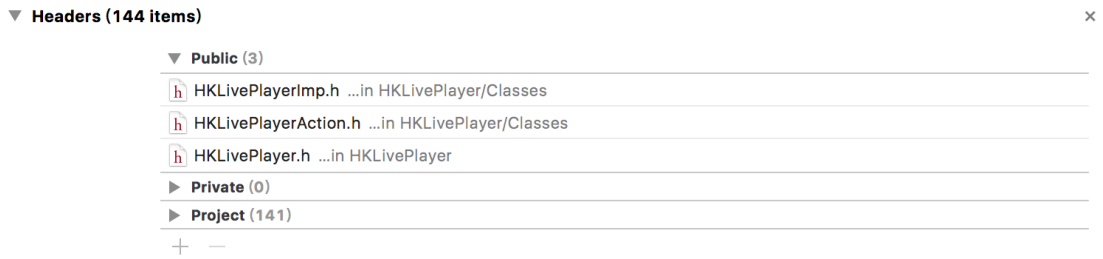

图 5-1 播放器公开的头文件

5.1.4 播放事件处理

iOS 开发过程中不同的对象进行通信的方法有代理、键值观察、系统通知和 Block。

代理是通过一个对象委托另一个对象为自己的代理，在代理对象中实现响应的方法后，委托对象中就可以调用这个方法。此种方法需要先声明代理方法，然后在代理中实现对应的方法，过程相对复杂。

键值观察也称为 KVO 是 iOS 提供的一种通知方法，可以为一个对象的某个属性设置观察者，只要被观察的属性值发生了变化，观察者就能就接受到事件产生的通知。

系统通知是 iOS 提供的另一种通知方法。就是在需要产生通知的地方通过系统发布事件通知，在应用中只需要通过特定的键值监听系统发出的消息，事件一旦发生，监听对象就能接收到通知。

Block 是 Objective-C 语言的特有机制之一，即声明一个代码块，在另外一处实现该代码块，等到被声明的对象调用代码块时，实现代码块之处就会响应。

上述方法看似相似，都是“一处声明另一处实现”的模式，但是每一种都采用了不同的机制。其中 Block 看起来简洁方便，但是在多文件交互时使用方法却很复杂，还有可能产生循环引用的问题。代理的声明和使用都需要很多的代码来实现，只适合在少量事件处理，在播放器项目中事件比较丰富，如果使用代理会显得臃肿。键值观察和系统通知都属于通知类型，两者的区别在于，键值观察只能在同一个环境中完成，属于局部通知；系统通知是向整个应用 App 发布事件产生的通知，属于全局通知，采用系统通知的缺点是在当前控制器退出之前需要移除对系统通知的监听，否则会引起应用崩溃。考虑到接入的开发者接触不到播放器框架的核心内容，为了让播放器框架的使用方法简洁明了，本文选用系统通知的方法统一对产生的事件进行通知，并提供统一绑定通知事件的方法和统一移除通知的方法如图 5-2 所示。

```
// 注册播放通知
[self.player registerPlayerNotificationTarget:self
                                  stateAction:@selector(stateAction:)
                               progressAction:@selector(progressAction:)
                               playableAction:@selector(playableAction:)
                                  errorAction:@selector(errorAction:)];
// 移除播放器通知
[self.player removePlayerNotificationTarget:self];
```

图 5-2 播放事件通知的注册和移除方法

图 5-2 中注册播知就是重新封装过的播放通知与响应方法绑定的方法，“@selector()”是 Objective-C 中定义的宏，只要在括号中添加对应方法名编译时就可以找到对应的方法。项目中的通知类型包括播放出错通知、播放状态通知、播放进程改变通知、播放时长改变通知，这些键值定义在 HKLivePlayerAction.h 头文件中。通过键值监听这些事件，一旦播放器向系统发布通知，监听之处就能做出响应，比如监听播放时长的改变来实现进度条的移动。表 5-1 是播放器事件系统通知的键值，只要通过这些键值就能了解播放器内部状态的改变，然后做出调整。

表 5-1 事件通知键值表

通知名称	键值
错误通知	HKPlayerErrorNotification
	HKPlayerErrorKey
状态通知	HKPlayerStateChangeNotification
	HKPlayerStatePreviousKey
	HKPlayerStateCurrentKey
播放进程通知	HKPlayerProgressChangeNotification
	HKPlayerProgressPercentKey
	HKPlayerProgressCurrentKey
	HKPlayerProgressTotalKey
可播放通知	HKPlayerPlayableChangeNotification
	HKPlayerPlayablePercentKey
	HKPlayerPlayableCurrentKey
	HKPlayerPlayableTotalKey

5.1.5 播放器的生命周期

为了简化播放器的使用，播放器的各个模块会高度封装到播放器对象中，播放器对象提供播放器的创建初始化和释放的方法。其中创建初始化提供两种方式，一种是半初始化，即先初始化播放器的组件，但是不设定播放地址，播放时直接添加地址播放；另一种是完全初始化，播放器通过给定的播放地址直接初始化到即将缓冲播放的状态。半初始化中播放器可以通过修改播放地址重复利用，适合在播放列表和播放页面在同一屏幕时使用。完全初始化的播放器只能适合当前的播放地址不能重复使用，适合在跳转到播放页面直接播放时使用。

因为苹果公司推出了自动引用计数机制 ARC 能够自动回收未被引用的对象，所以播放器对象在没有被引用时会自动释放。为了提高播放器的健壮性，本文在系统回收播

放器对象自动调用的方法 dealloc()中加入了移除系统通知和释放 FFmpeg 变量的方法，避免出现未移除系统通知和内存泄漏引发的程序崩溃。

5.2 核心模块

播放器框架的核心模块也就是业务层模块，包含了播放器接收流媒体和对流媒体数据解码展示的内容，由以下几个部分组成：RTSP 流媒体解封装模块、解码模块、图像渲染模块、YUV 转 RGB 图像的截图模块、音频渲染模块、和音画同步模块。

5.2.1 解封装模块

客户端在播放视频文件或者流媒体时都需要先将文件流解封装成视频流和音频流，再分别解码才能够显示和播放。项目中的解封装模块项目中采用的是 FFmpeg，尽管 FFmpeg 完整地支持众多格式编码和流媒体协议解析，但是因为完整版的框架占据了太多的 ROM 空间，不符合本文的需求，所以论文通过研究，对 FFmepg 剔除多余部分后重新编译，接下来介绍 FFmpeg 框架的编译使用过程。

1、FFmpeg 编译

FFmpeg 的编译需要根据框架所在的系统平台以及对应的处理器架构完成，否则编译后的文件可能会出现无法正常使用的现象，在 iOS 平台下一般使用 FFmpeg 的静态库. a 文件。表 5-2 是不同型号 iPhone 设备的处理器架构，其中模拟器因为是在电脑上实现的，其架构都与所在的电脑相同一般都是 i386 或者 x86_64 架构。硬件设备采用的处理器都是低功耗的 ARM 架构，同时期的 iPad 设备与 iPhone 设备基本是一致的。而且这些 ARM 指令集架构都向下兼容，比如 ARMV7s 指令集兼容 ARMV7 指令集。

表 5-2 不同型号 iPhone 设备处理器架构

型号	模拟器	iPhone4~iPhone4s	iPhone5~iPhone5c	iPhone5s 之后
处理器	i386/ x86_64	ARMV7	ARMV7s	ARM64

移动应用一般都要求规模尽量减小，为了使整个应用轻量化，项目中在调试运行编译生成的是同时包括 AR6M 和 Intel 指令集的 FFmpeg 版本，发布时编译生成的是只含有 ARM 架构的版本。

FFmpeg 默认启用全部的流媒体协议和多种编码格式的支持，项目中流媒体协议为 RTSP，要支持的编码格式是 MPEG-4 和 AAC，所以可以通过编译脚本关闭不需要的编码格式和协议的支持，配置编译选项时在编译脚本中加入表 5-3 所示的代码。

表 5-3 精简 FFmpeg 编译脚本关键代码

代码	作用
disable-decoders	取消所有解码器
enable-decoder=mpeg4	启用 MPEG-4 解码器
enable-decoder=aac	启用 AAC 解码器
disable-protocols	取消所有协议支持
enable-network	启动网络支持
enable-demuxer=rtsp	启动 RTSP 协议支持

从表 5-3 中可以看出，要使 FFmpeg 只支持某一种编码格式或者流媒体协议需要先将所有支持的编码格式或者流媒体协议全部取消，再分别激活启用。

编译的过程是从终端进入包含脚本的目录，再输入命令“./build-ffmpeg.sh”，脚本会自动先从 FFmpeg 仓库中下载源码，之后编译成同时包含架构 i386、x86_64 和 ARM 架构的.a 静态文件，这部分文件供调试的时候使用。发布版本输入命令“./build-ffmpeg-thin.sh arm64”会编译生成只包含 ARM 指令集结构的版本,编译过程如图 5-3 所示。

```
➜ FFmpeg-iOS-build-script git:(master) ✗ ./build-ffmpeg-thin.sh arm64
building arm64...
install prefix            /Users/Leo/Desktop/ffmpeg/FFmpeg-iOS-build-script/thin
/arm64
source path               /Users/Leo/Desktop/ffmpeg/FFmpeg-iOS-build-script/ffmp
eg-3.4
C compiler                xcrun -sdk iphoneos clang
C library
host C compiler           gcc
host C library
ARCH                      aarch64 (generic)
big-endian                no
runtime cpu detection     yes
NEON enabled              yes
VFP enabled               yes
debug symbols             no
strip symbols             yes
optimize for size         no
optimizations             yes
static                    yes
shared                    no
postprocessing support    no
network support           yes
threading support         pthreads
```

图 5-3 FFmpeg 编译

完整版的解码器的占用 ROM 空间高达 98.5MB，而精简版生成的静态库文件中解码器部分只有 7.7MB，总体规模缩小到完整版的 7.81%，可见剔除用不到的功能，能节省 90%多的 ROM 占用空间。编译完成之后会生成一个 include 目录和一个 lib-iOS 目录，include 目录中包含的是暴露出来的头文件，lib 目录下是生成的静态库文件。将这两个文件夹加入到工程中如图 5-4 所示。

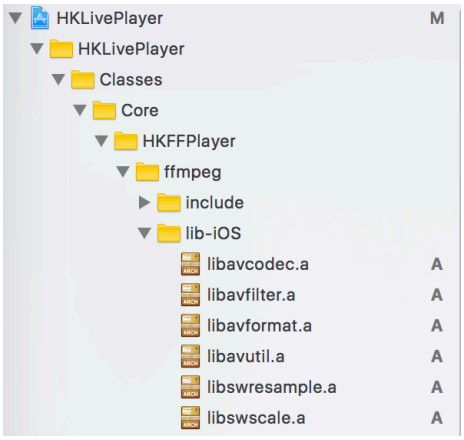

图 5-4 导入 FFmpeg

2、解封装流程

根据概要设计可知，项目中是通过视频流地址获进行取流播放，解码播放视音频首先要从网络数据流中拆分成视频流和音频流，这就是解封装的过程。因为 FFmpeg 中网络相关的技术已经非常成熟，获取数据流和解封装也非常方便快捷，所以在解封装这块项目中采用 FFmpeg 实现，图 5-5 是使用 FFmpeg 的解封装流程。

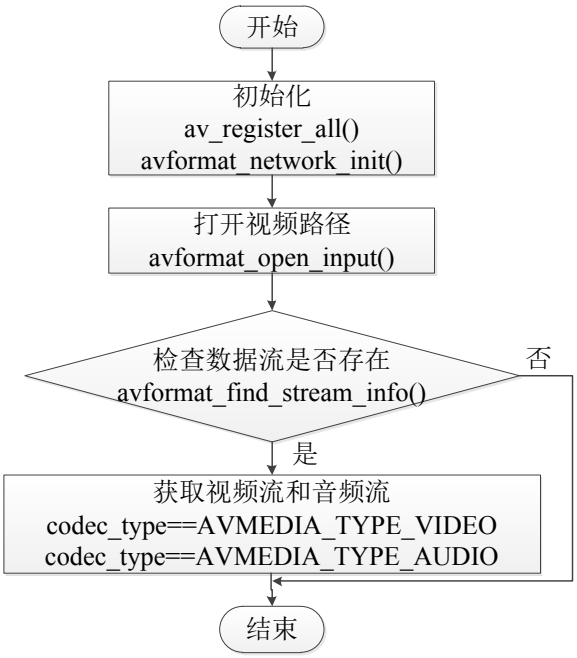

图 5-5 FFmpeg 解封装流程

第一步是相关的 FFmpeg 库的引入和基础配置。需要引入的头文件包括：avcodec.h、avformat.h、swscale.h、imgutils.h、swresample.h，之后在对象的初始化方法之中调用 av_register_all()方法注册 FFmpeg 相关的组件，网络支持模块需要调用初始化方法 avformat_network_init()。

第二步要输入播放的视频路径来初始化 AVFormatContext 结构，调用的函数是 avformat_open_input()，此函数需要传入的第一个参数是 AVFormatContext 结构的指针，第二个参数是将要解码播放的视频路径，因为此方法 C 为语言的函数，尽管 Objective-C 可以调用 C 语言的函数，但是 C 语言无法识别 Objective-C 的字符串对象，所以需要将 Objective-C 的 NSString 对象转化成一个 char *字符串。这个函数的返回值是 0，说明初始化成功。

第三步通常是调用 avformat_find_stream_info()方法，检查数据流是否存在问题，返回值大于 0 为正常的流媒体，如果返回的结果小于 0，说明不存在或者不是正常的流媒体，则退出解封装流程。

经研究发现，检查数据流函数基本上是在函数内部进行了一次部分数据的解码过程，对于某些格式的媒体这个过程耗时较长，在 iOS 设备上对高清视频甚至会报“time out”的错误。为了避免或者减少这个接口的延迟，项目中采取的方法是设置 AVFormatContext 结构的 probesize 成员来限制 avformat_find_stream_infoA()接口内部读取的最大数据量，也就是在调用这个函数之前先设置 probesize：“_fmtCtx->probesize＝4096;”，经过实验测试，采用此操作之后没有出现过“time out”错误。

第四步是获得的媒体流中视频流和音频流的索引。AVFormatContext 结构中有个数据成员 streams，这是 AVStream 结构类型的数组指针代表获取到的流媒体，里面可能包含多种视频流、音频流和字幕流等。通过 AVStream 结构的编解码器参数 codecpar 的编解码器类型 codec_type 是否为 AVMEDIA_TYPE_VIDEO 或者 AVMEDIA_TYPE_AUDIO，

确定码流是否未视频流或音频流。获取到了视频流和音频流就获取到了对应的AVCodecContext 结构，这是 AVStream 的一个成员结构 codec，所以正好跟每一个数据流对应。

5. 2. 2 解码模块

前文介绍过，本文的视频编码格式包含 h.264 和 MPEG-4 两种，对于 h.264 编码格式的视频，论文中采用的是 VideoToolbox 视频播放框架，该框架采用的是硬件解码，在解码方面大大减轻 CPU 的功耗，提高播放性能。到目前为止 iOS 还无法支持 MPEG-4 编码格式，所以对于这种格式的视频和 AAC 编码的音频，项目中采用的解码方法是 FFmpeg，解码流程如图 5-6 所示。

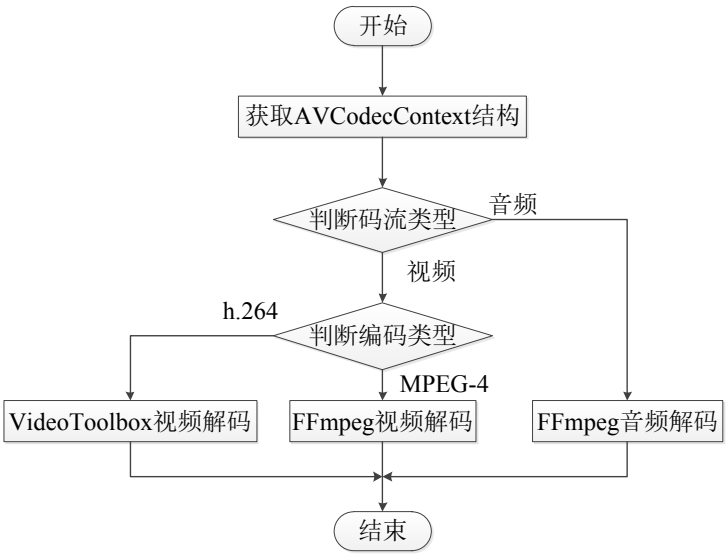

图 5-6 解码流程图

图 5-6 中在判断视频的编码类型时，是通过码流 AVCodecContext 的数据成员 codec_id 来确定码流的编码格式，codec_id 是枚举类型 AVCodecID，h.264 的为 AV_CODEC_ID_H264，表示 MPEG-4 的为 AV_CODEC_ID_MPEG4。

1、FFmpeg 视音频解码

采用 FFmpeg 解码视频和音频的两个过程类似，基本过程如图 5-7 所示。

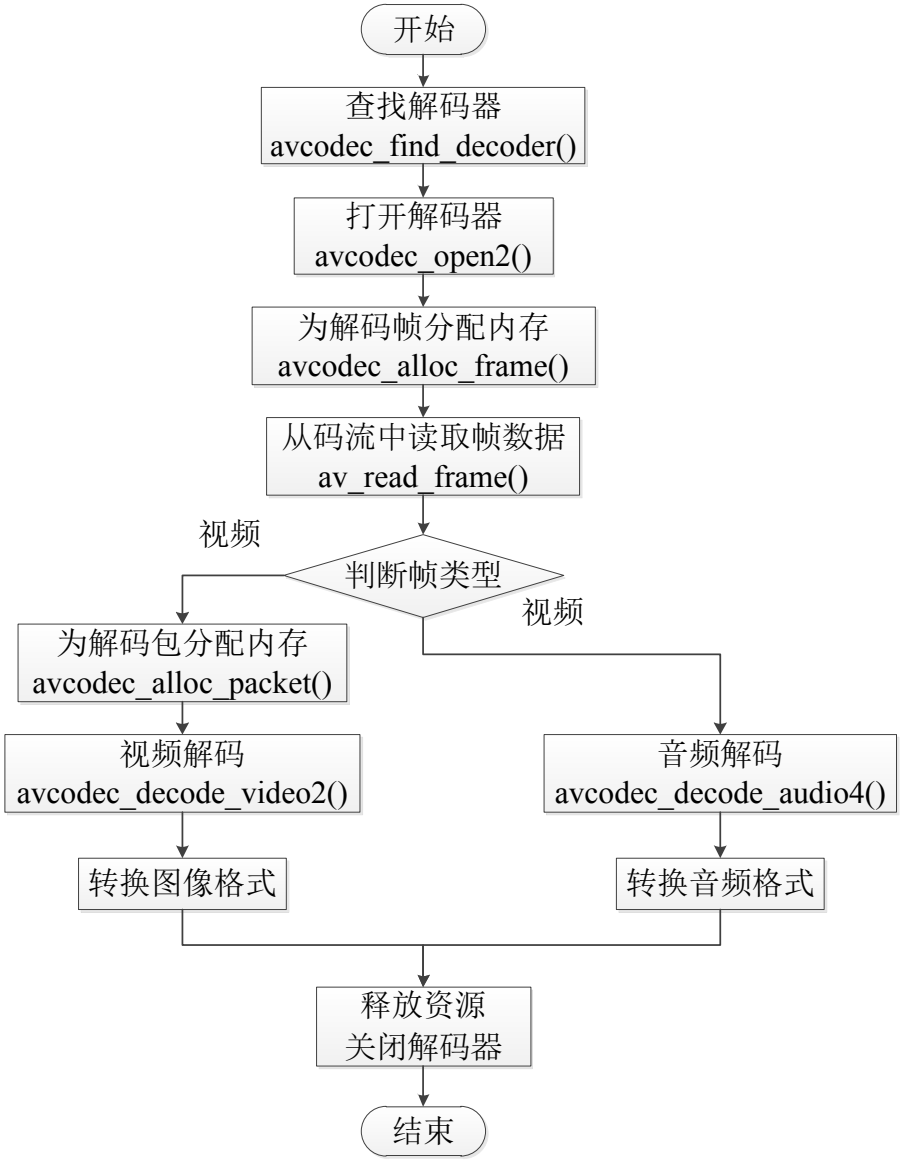

图 5-7 FFmpeg 解码音频视频流程

首先是查找解码器 AVCodec，调用的函数是 avcodec_find_decoder()这个函数的参数 AVCodecID 位于对应码流的 AVCodecContext 结构中。接下来是打开解码器 avcodec_open2()，参数是每个流对应的 AVCodecContext 结构和查找到的解码器结构。在解码之前先要对暂存一帧视频或音频的数据 AVFrame 进行内存分配，调用的函数是 av_frame_alloc()，解码完成之后输出的内容就是包含在这个结构中，对于视频流还需要对 AVPacket 结构调用 av_packet_alloc()进行内存分配。最后就是分别调用函数 av_read_frame()读取一帧数据进行解码，视频解码函数是 avcodec_decode_video2()，音频解码函数是 avcodec_decode_audio4()。

FFmpeg 视频解码后输出视频的内容有两种格式，一种是输出 RGB 格式图片数据，这个格式可以生成 iOS 的图片对象 UIImage 直接显示在屏幕上；另一种是 YUV 格式数据。要获得这两种数据，都需要调用 FFmpeg 的 libswscale 模块的方法进行图像格式转化，不同的操作是在调用 sws_getContext()获得图像格式转化上下文时所传入不一样的参数 AVPixelFormat，普通图像数据的参数是 AV_PIX_FMT_RGB24，YUV 数据的参数项目中是 AV_PIX_FMT_YUV420P。获得上下文之后调用图像转化的执行函数 sws_scale()

之后即可在 AVFrame 结构的成员 data 数组中获得，这个时候也可以释放掉获取的图像转换上下文供下次使用了。

关于音频解码，解码完成的数据是不适合 iOS 设备播放的，所以也有一个转化的过程，与视频的类似，采用的方法是 swr_convert()，所不同的是视频所输入的尺寸是图像画面的长和宽，音频是重新设置可以播放的采样率。

2、VideoToolbox 解码

VideoToolbox 框架解码前后相关数据结构之间的关系如图 5-8 所示。

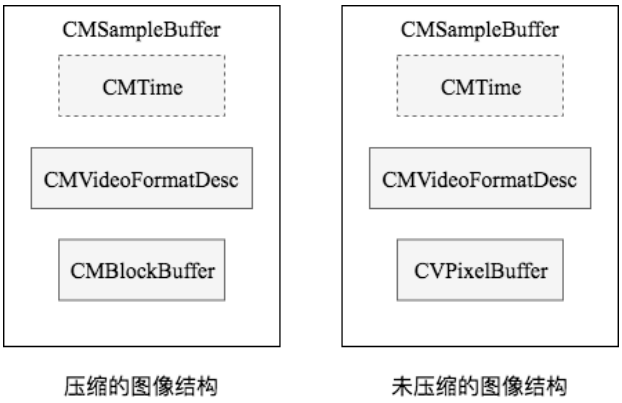

图 5-8 解码前后 VideoToolbox 对应的数据结构

从图 5-8 中可以看到压缩的图像结构和未压缩的图像结构很相似，只有 CMBlockBuffer 和 CVPixelBuffer 不一致；CMBlockBuffer 和 CVPixelBuffer 都指的是图像的数据结构，区别是 CMBlockBuffer 是指编码后的图像的数据结构，而 CVPixelBuffer 指的是编码前和解码后的图像数据结构。CMTime 是与时间关联的数据结构；CMVideoFormatDescription 是图像存储方式、编解码器等格式描述；CMSampleBuffer 是存放编解码前后的视频图像的容器数据结构。

根据图 5-8 显示，解码前的 CMSampleBuffer 由 CMTime、CMVideoFormatDescription 和 CMBlockBuffer 构成，但是这些数据结构都是在 VideoToolbox 中定义的，数据流中不会提供完整的 CMSampleBuffer，需要从数据流中把这些数据提取出来组合成 CMSampleBuffer。

在 h.264 的编码的数据流中，有一个最基础的层叫做 Network Abstraction Layer，简称 NAL。h.264 流数据正是由一系列的 NAL 单元(NAL Unit, 简称 NALU[33])组成的，如图 5-9 所示。

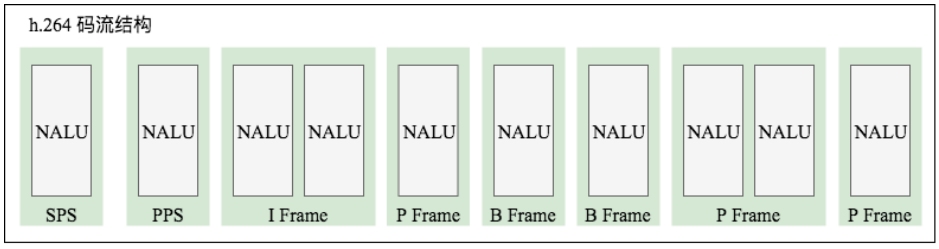

图 5-9 h. 264 码流结构

图 5-9 显示了一系列 NALU 构成的 h.264 码流的结构，这些 NALU 单元包含视频图像数据和 h.264 的参数信息。其中视频图像数据就是压缩的图像结构 CMBlockBuffer，也就是图中指示的 I Frame、B Frame、P Frame 中所包含的数据，其具体内容在音画同步模块有详细介绍；参数信息就是图 5-9 中前两个 NALU 中包含的内容 SPS（Sequence Parameter Set）和 PPS（Picture Parameter Set）。

要组合 CMVideoFormatDescription 需要先提取 h.264 码流中的 SPS 和 PPS。因为每个 NALU 都是以“0x00 00 00 01”为标志位开始，所以可以按照开始码定位 NALU，而开始码之后的第一个字节的低 5 位如果是 7 则代表是 SPS，如果是 8 则代表是 PPS，项目中可以按照类型依次提取 SPS 和 PPS 数据。最后将这两组数据组合，并分别将两者的字节大小组合然后调用 CMVideoFormatDescriptionCreateFromH264ParameterSets 就能生成 CMVideoFormatDescription 对象了。

提取 CMBlockBuffer 数据，还是通过开始码定位到每一个 NALU，查看类型之后如果不是 SPS 或 PPS 数据就可以提交解码了，不过在提交之前需要先将开始码“0x00 00 00 01”，替换成 NALU 的长度信息，这个长度信息需要自行计算。接下来就是将替换好的一帧文件提交到函数中生成 CMBlockBuffer 数据，调用的的函数是 CMBlockBufferCreateWithMemoryBlock。

与 FFmpeg 解码类似，iOS 的硬解码有两种方式，一种是直接解码展示图像，第二种是解码之后形成一帧图像数据。因为项目中在图像渲染方面采用的是 OpenGL ES，需要得到渲染的数据，所以解码方式在项目中选择的是后者。解码主要分为三步：首先采用 VTDecompressionSessionCreate 创建解码 session 对象。主要参数有三个，_decoderFormatDescription 就是前面创建的 CMVideoFormatDescription 对象；callBackRecord 是解码完的回调函数；attrs 是一个字典类型的对象，用来存储当前 session 的属性值，属性值的设置如表 5-4 所述。

表 5-4 解码 session 设置的属性值

解码 session 属性	属性所设置的值
kCVPixelBufferPixelFormatTypeKey	kCVPixelFormatType_420YpCbCr8Planar
kCVPixelBufferWidthKey	h264outputWidth
kCVPixelBufferHeightKey	h264outputHeight
kCVPixelBufferOpenGLCompatibilityKey	YES

从表 5-4 中可以看出，这些属性主要是用来设置图像的类型和宽高值等。因为渲染类型采用的 OpenGL ES 所以在 OpenGL 兼容的属性上使用了 Objective-C 语言的布尔值“YES”表示支持；OpenGL 在当前项目的实现中采用的渲染数据是 YUV420p 类型，所以当前 session 的图像类型设置的键值为 kCVPixelFormatType_420YpCbCr8Planar。

解码 session 的创建相当于解码器的初始化，接下来可以逐帧的解码视频，这就是循环调用 VTDecompressionSessionDecodeFrame 函数直到视频结束的过程。这个函数的主要参数包括前面生成的 CMBlockBuffer 数据、解码 session 对象和数据输出对象，这个输出的对象是图 5-8 所示的 CVPixelBuffer 解码之后的数据。

解码完成之后，为了防止应用的内存溢出，需要将创建的解码 session 对象销毁。销毁的方法是将 session 对象传入函数 VTDecompressionSessionInvalidate 中。

5. 2. 3 图像渲染模块

基于提升播放性能和减轻 CPU 的计算压力，在图像渲染方面采用的是 OpenGL ES 来实现。使用 OpenGL ES 渲染的操作分为以下 4 个内容。

1、根据 OpenGL ES 提供的图元来设计多边形，能将顶点与纹理一一对应。

2、在三维空间中将步骤 1 设计的多边形排列，并将观察的镜头位置设定好。如果只是 2D 渲染，将 z 轴设为 0，视角设在 z 轴正方向即可。

3、物体颜色的组合。OpenGL 在绘制时其颜色值指的是编码时设定的物体颜色值、

在所在条件下光照和阴影以及纹理表面得到的颜色值的组合。

4、计算屏幕上对应点位置的像素点的颜色值，这是由步骤 1 中的多边形和步骤 3 中计算的物体颜色值转换得到，这个过程称为光栅化。

前面提到 OpenGL 2.0 推出了着色器语言 GLSL，项目中需要用到的是顶点着色器和片段着色器，如图 5-10、图 5-11 所示。

```
//VertexShader.glsl
attribute vec4 position;
//uniform float translate;
attribute vec2 TexCoordIn;
varying vec2 TexCoordOut;

void main(void)
{
    gl_Position = position;
    TexCoordOut = TexCoordIn;
}
```

图 5-10 顶点着色器

```
// FragmentShader.glsl
varying lowp vec2 TexCoordOut;

uniform sampler2D SamplerY;
uniform sampler2D SamplerU;
uniform sampler2D SamplerV;

void main(void)
{
    mediump vec3 yuv;
    lowp vec3 rgb;

    yuv.x = texture2D(SamplerY, TexCoordOut).r;
    yuv.y = texture2D(SamplerU, TexCoordOut).r - 0.5;
    yuv.z = texture2D(SamplerV, TexCoordOut).r - 0.5;

    rgb = mat3( 1,          1,          1,
                0,          -0.3455,    1.799,
                1.4075,     -0.7169,    0) * yuv;

    gl_FragColor = vec4(rgb, 1);

}
```

图 5-11 片段着色器

从图 5-10 和图 5-11 可以看到两个着色文件都类似于一个 C 语言程序的代码，开始部分都是声明变量，都有一个 main 函数，这个 main 函数就是运行在 GPU 中。顶点着色器文件中有三个变量，第一个是顶点坐标，第二是纹理坐标，这两个两个变量会在编程时的进行数据进行绑定，第三个 TexCoordOut 是从顶点着色器传到片段着色器的。在片段着色器的四个变量中，第一个就是从顶点着色器传过来的 TexCoordOut，后面三个分别指代 YUV 数据中的 Y、U、V 样本，在 main 方法中通过一个颜色矩阵的运算将 YUV 数据转化成每个点的像素值，最后由 GPU 渲染到屏幕上。

基于对 3D 视频播放扩展的考虑，项目中使用的坐标系全部是三维坐标系，对于 2D 视频的处理是将 z 坐标轴的值都设为 0。并且对渲染时不同的模块进行拆分封装，拆封

方式如图 5-12。

图 5-12 渲染部分模块拆分

由图 5-12 可知，渲染方面主要由三部分组成：显示画面、显示控制、和显示模型。其中显示画面指的是用来显示图像的界面，实现图像数据到 OpenGL ES 的纹理转换，这个界面在 2D 或者 3D 画面都是相同的。显示控制指的是指的是对当前显示画面的控制和现实坐标系与 OpenGL ES 坐标系的转换，比如在 3D 球面显示时，手机屏幕可能无法正常实现 3D 场景，需要实现拖动屏幕时的画面变换。通常情况下，2D 视频是不需要画面控制，直接全部显示即可，所以在画面控制模块中只需要完成现实坐标到 OpenGL ES 坐标的转换。显示模型指的是指的是视频画面在屏幕中显示的形式，主要包括平面模型、球状模型和立体模型等，2D 视频显示采用的是平面模型，3D 视频采用的是球状模型。采用此种方式设计，为后续 3D 画面扩展提供了方便，添加对应的显示模型即可。

在 iOS 平台需要导入对应的框架 OpenGLES.framework 和 QuartzCore.framework。与 FFmpeg 类似，OpenGL ES 渲染也需要对应的上下文环境，而且我们选择的 OpenGL ES 的 2.0 版本，所以在创建上下文环境的时候需要指定 OpenGL ES 为对应的 2.0。可以认为 OpenGL 所有的工作都必须在这个上下文环境中完成。

OpenGL ES 需要创建两个缓冲区：渲染缓冲区和帧缓冲区。渲染缓冲区是一种 OpenGL ES 对象，用于存放渲染过的图像，本质上是对颜色值的存放。颜色缓冲区与前面定义的上下文环境和 OpenGL ES 绘制层进行了绑定，在最后将渲染缓冲区依附在了帧缓冲区的 GL_COLOR_ATTACHMENT0 所指示的位置上。

OpenGL ES 是高效地运用到了 GPU 进行渲染相关的运算，前文创建了两个着色器文件，为了让 GPU 运行这两个着色器文件中的代码，需要一个 OpenGL ES 能够识别的程序对象，连接两个着色器文件形成一个完整的 OpenGL ES 程序。以上步骤可以认为使用 OpenGL ES 渲染的初始化，而且 OpenGL ES 是状态机的工作方式，所以这些设置只需执行一次，到视频播放完毕之后清理即可。

初始化之后就可以进行图像绘制也就是贴纹理操作，即将 YUV 数据按照一定的格式传递给 OpenGL ES。贴纹理的过程有五步，首先分别生成与 YUV 对应的纹理，调用的方法是 glGenTextures()。所谓的生成纹理就是将 YUV 数据的不同部分拟合成一帧完整图像纹理。然后将 OpenGL ES 中的纹理对象激活 glActiveTexture()，只有激活的纹理才能够拷贝到 OpenGL ES 中；第三步是调用 glBindTexture()将生成的纹理与激活的 OpenGL ES 纹理对象绑定;第四步就是将 YUV 数据传递给 OpenGL ES 使用的方法为 glTexImage2D()；最后一步就是设置纹理的格式，调用的方法是 glTexParameteri()，此方法就是把纹理像素映射成显示像素。

在基础知识中提到项目中绘制图像采用的图元是三角形，也就是将矩形的右上角和左下角两个顶点连接形成的两个三角形。将两个三角形的四个顶点与 OpenGL ES 视口中的四个顶点对应，OpenGL ES 内部会把之前传递的 YUV 数据形成的纹理分成两个三角形图像与这个顶点对应，所以最后看到一帧完整的图像。这里有两个对应的坐标系如图 5-13 所示，一个是 OpenGL ES 内部的坐标系即顶点坐标系，另一个是纹理坐标系，其中顶点坐标的范围是(-1,1)，纹理坐标的范围是(0,1)。

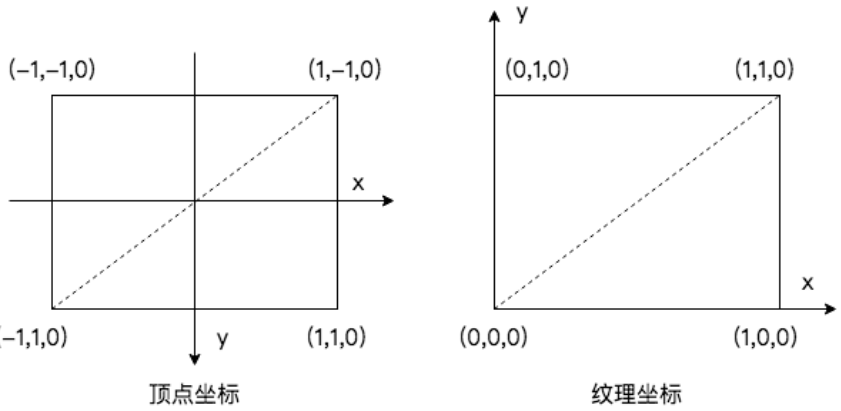

图 5-13 顶点坐标系与纹理坐标系

从图 5-13 中可以发现顶点坐标的 y 轴和纹理坐标的 y 轴方向刚好相反，而且顶点坐标系的原点在矩形的中间，也就是屏幕的中心，而纹理的原点在左下角，所以在设置两个坐标的时候一定要将各个点对应好，否则会得不到预设的纹理画面。将来个坐标与顶点着色器中的两个顶点变量进行绑定，之后调用 glDrawArrays()完成视频图像绘制，最后用绘制上下文环境对象_context 将渲染缓冲区展示在当前的 layer 层即可看到渲染的图像。

5. 2. 4 YUV 转 RGB 的截图模块

需求分析中提出播放器框架需要实现实时截图功能，根据播放器框架的架构设计可知，播放器中并没有实时生成图像文件，而是将解码之后的 YUV 数据提交给 OpenGL ES 实现图像渲染。因此项目中对于实时截图的功能采用的方法是将当前播放帧的 YUV 数据直接转成 RGB 图像格式，并最终保存在手机的相册中。式 5.1 为 YUV 数据转为 RGB 图像格式的公式，本节将根据此公式讨论 YUV 转为 RGB 的算法。

1、整型算法

根据式 5.1 可以分别得到 R、G、B 的转化公式。

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.4075 \\ 1 & -0.3455 & -0.7169 \\ 1 & 1.779 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U-128 \\ V-128 \end{bmatrix} \tag{5.1}$$

$$R = Y + 1.4075 * (V - 128) \tag{5.2}$$

$$G = Y - 0.3455 * (U - 128) - 0.7169 * (V - 128) \tag{5.3}$$

$$B = Y + 1.779 * (U - 128) \tag{5.4}$$

从式 5.2、5.3、5.4 中可以看到，三个公式都存在浮点数，相对浮点预算，整形运算更有优势。项目中首先考虑的是采用整型算法来实现数据转换。假设存储 YUV 的数组为 yuvData，数据 U 的起始位置为 uPos，V 的起始位置为 vPos，根据前文对 YUV 数据存储的介绍，Y 数据的起始位置为 0。假设用 y 表示公式中 Y 的值，u 和 v 分别表示公式中的 U-128 和 V-128 那么可以得到式 5.5、5.6 和 5.7。

$$y = yuvData[0] \tag{5.5}$$

$$u = yuvData[uPos] - 128 \tag{5.6}$$

$$v = yuvData[vPos] - 128 \tag{5.7}$$

对于式 5.2、5.3 和 5.4 中的浮点数，经过计算可以得到式 5.8~5.11。

$$0.4075 \times 256 \approx 104 \tag{5.8}$$

$$0.3455 \times 256 \approx 88 \tag{5.9}$$

$$0.7169\times256\approx183 \tag{5.10}$$

$$0.779\times256\approx199 \tag{5.11}$$

假设使用 rTemp、gTemp 和 bTemp 分别表示式 5.2、5.3 和 5.4 中除 Y 之外的内容，r、g、b 分别代表获得到的 R、G、B 数据，可以得到式 5.12-5.17。

$$rTemp = v + (104 * v) \gg 8 \tag{5.12}$$

$$gTemp = (88 * u) \gg 8 + (183 * v) \gg 8 \tag{5.13}$$

$$bTemp = u + (199 * u) \gg 8 \tag{5.14}$$

$$r = yuvData[0] + rTemp \tag{5.15}$$

$$g = yuvData[0] - gTemp \tag{5.16}$$

$$b = yuvData[0] + bTemp \tag{5.17}$$

以上公式通过多次整数运算最终得到了 RGB 对应的数据，因为用 RGB 表示颜色时其最大值为 255，所以在得到计算结果时，为了防止数据溢出，还需要进行判错计算。可以看出整型算法中最复杂的部分是 rTemp、gTemp 和 bTemp 的计算，尽管全是整数运算，并且对于除法运算采用了位移的方法，此种方法还有提升的空间。

2、半查表法

因为 YUV 数据中每一个分量都是范围在 0-255 的整数，本文考虑采用查表法，将 rTemp、gTemp 和 bTemp 的所有值都先计算好，存储在一张表中，按照 YUV 的分量值在表中获取可以提高速率。根据式 5.2、5.3 和 5.4，假设存储各浮点数与 U 和 V 数据乘积的数组为 table_R_V、table_G_U、table_G_V 和 table_B_U，那么通过查表获得的 rTemp、gTemp 和 bTemp 分别为。

$$rTemp = table\_R\_V[v] \tag{5.18}$$

$$gTemp = table\_G\_U[u] + table\_G\_V[v] \tag{5.19}$$

$$bTemp = table\_B\_U[u] \tag{5.20}$$

采用半查表法需要有 4 个一维数组，长度为 256 位，需要占用大约 1KB 内存空间，而且对于式 5.6 和 5.7 中的减 128 都可以在先行计算好，放入表中，减少使用时的计算量。此种方法采多使用了 1KB 内存空间，用查表代替了整型计算中的 2 减法、4 次乘法和 4 次位移运算，理论上是优于整型算法的，但是依然存在多次加减计算和赋值操作。

3、全查表法

经过半查表法的思考，可以认为通过每个 Y、U、V 分量直接获取到每个像素点 RGB 的值将会更加提高转化速率。从式 5.3 可知，G 分量的值是与 Y、U、V 三个分量的值都是相关的，假设实现一个三维数组 table_G_YUV，通过数组 g = table_G_YUV[y][u][v]获得 G 分量的值。但是因为三个下标都是 256 位，所以这个三维数组至少要使用 16MB 内存空间，为一个截图功能使用 16MB 内存空间是完全不值得的。

经过分析可知，对于 G 分量，Y 的值跟 U 和 V 的值并不相关，所以可以考虑使用两个二维数组通过两次查表的方式实现。具体的做法就是先计算生成两个二维数组，一个是关于 U 与 V 组合，另一个是关于 Y 与 UV 组合结果的组合。假设 U 与 V 组合的二维数组为 table_G_U_V，Y 与 UV 组合结果组合的二维数组为 table_G_Y_UV，通过全查表法获得的 G 分量的方法如式 5.21 所示。

$$g = table\_G\_Y\_UV[y][table\_G\_U\_V[u][v]] \tag{5.21}$$

对于 R 和 B 因为只与 YUV 中的两个分量相关，所以可以直接使用二维数组来实现。假设 R 分量中 Y 与 V 的二维数组为 table_R_Y_V，B 分量中 Y 与 U 的二维数组为 table_R_Y_U，那么 R 和 B 的获取方法分别如式 5.22、5.23 所示。

$$r = table\_R\_Y\_V[y][v] \tag{5.22}$$

$$b = table\_B\_Y\_U[y][u] \tag{5.23}$$

从式 5.21、5.22 和 5.23 可以看出，全查表法需要四个 256*256 的二维数组，也就是大约需要占用 256KB 内存空间，这完全是可以接受的，而且相对于半查表法，全查表法需要同样数据读取次数，但是省去了 4 次加减法的运算。

5. 2. 5 音频渲染模块

同 OpenGL 一样音频库 OpenAL 也是针对 3D 场景设计的跨平台方案，在大型游戏中经常使用。OpenAL 抽象出三种基本对象：Buffer（缓冲区）、Source（源）、Listener（听众）。Buffer 用来填充音频数据，然后附加到一个 Source 上，Source 可以被定位并播放。利用 OpenAL 实现音频播放必须先将 OpenAL. framework 框架导入工程中，导入 OpenAL 框架之后，实现音频播放的过程如图 5-14 所示。

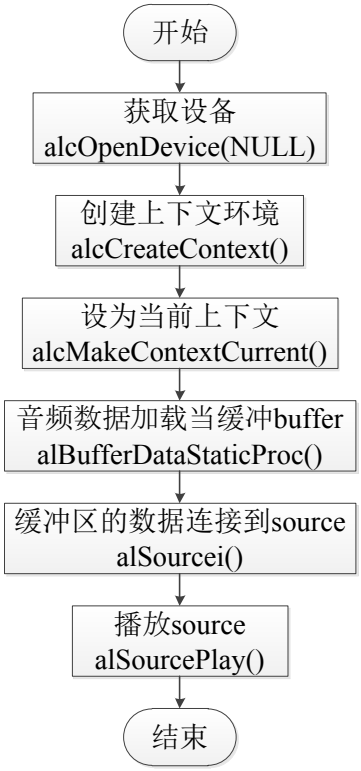

图 5-14 OpenAL 播放音频流程

首先是获取设备，在 iOS 设备中只需要获取一个全局的 OpenAL Device，调用函数 alcOpenDevice()，参数设为 NULL 即可获得系统默认的输出设备。获取设备之后调用 alcCreateContext()将刚刚获取的设备传入即新建一个与当前设备关联的上下文环境，之调用函数 alcMakeContextCurrent()将这个上下文环境设为当前上下文环境。项目中声明一个 AudioPlayer 的结构，包含 sourceId、bufferId 这指代前面所提到的基本对象，因为项目中只是实现播放功能，所以 Listener 就采用的是上下文中默认的 Listener。开发过程中用到了 OpenAL 的关于 alBufferData 的一个扩展 alBufferDataStatic，alBufferData 的作用是缓存数据，alBufferDataStatic 的功能是加载音频数据到内存并关联到 bufferId。alBufferData 会拷贝音频数据所以调用后，还需要释放掉音频数据, 而 alBufferDataStatic 并不会拷贝。

接下来是将解码后的音频数据加载到 OpenAL 中，因为解码得到的数据是 PCM 格式的，因为 PCM 格式的数据并不包含采样率的信息，没有这些信息是无法完成播放的所以还需要重新添加，首先是生成 OpenAL 的 Buffer 和 Source 并与前面 Player 结构中的 bufferId 和 sourceId 绑定，然后调用 alBufferDataStaticProc()绑定音频数据，最后进行

相关设置。至此数据加载到 OpenAL 已经完成，接下来调用 alSourcePlay()传入 sourceId 就能进行音频的播放了。

5. 2. 6 音画同步模块

在视频流和音频流中已包含了播放速率的相关数据，视频的帧率（Frame Rate）指示视频一秒显示的帧数；音频的采样率（Sample Rate）表示音频一秒播放的样本的个数。可以使用以上数据通过计算得到其在某一 Frame 的播放时间，以这样的速度音频和视频各自播放互不影响，在理想条件下，其应该是同步的，不会出现偏差。但是因为视音频解码速度以及硬件固有原因，总可能会出现视音频不同步的情况。这就需要一种随着时间会线性增长的量，视频和音频的播放速度都以该量为标准，播放超前了就减慢播放速度，播放落后了就加快播放速度。所以音画同步在播放器的设计中也是至关重要的一点。

音画同步的方法一般有以下三种：

1、将视频同步到音频上，就是以音频的播放时钟为基准来同步视频。视频比音频播放延迟，加速其播放；若是超前，则延迟播放。

2、将音频同步到视频上，就是以视频的播放时钟为基准来同步音频。

3、将视频和音频同步外部的时钟上，选择一个外部时钟为基准，视频和音频的播放速度都以该时钟为标准进行播放。

前面提到，视频和音频的同步就是一个等待与追赶的过程，某一方快了则等待，某一方延迟了就加速播放。但是如何来判定快慢就需要一个量来确定，在视音频流的包中都含有解码时间戳 DTS（Decoding Time Stamp）和显示时间戳 PTS（Presentation Time Stamp），就是这样的量。解码时间戳，告诉解码器 packet 的解码顺序；显示时间戳，指示从 packet 中解码出来的数据的显示顺序。

对于音频，DTS 和 PTS 是相同的，也就是其解码的顺序和解码的顺序相同，视频的编码要比音频复杂一些，特别的是预测编码是视频编码的基本工具，就会造成视频的 DTS 和 PTS 的不同，因此在项目中是将视频同步到音频的。

根据编码时对视频数据的压缩方法，一般情况下会将几帧画面放在一组称为 GOP（Group of Picture），而包含在 GOP 中的帧数据一般包含下面几种类型：

I 帧——帧内编码帧，包含了一帧的完整数据，也称为关键帧，解码时只需要本帧的数据，不需要参考其他帧。

P 帧——前向预测编码帧，该帧的数据不完全的，解码时需要参考其前一帧的数据。

B 帧——双向预测内插编码帧，这种类型的帧解码需要同时参考前后相邻两帧图像。

I 帧的解码是最简单的，只需要本帧的数据；P 帧也不是很复杂，只需要缓存上一帧的数据即可，总体都是线性，其解码顺序和显示顺序是一致的。B 帧比较复杂，需要前后两帧的顺序，并且不是线性的，在解码后有可能得不到完整 Frame，这是造成了 DTS 和 PTS 的不同的主要原因。

比如有一个视频序列，要显示的帧数据序列是 I-B-B-P，但是因为 B 帧是双向预测帧，所以在解码 B 帧之前得到 P 帧的信息，因此可能是按照 I-P-B-B 顺序来存储帧数据，他们的解码顺序也是如此，这样其解码顺序和显示的顺序就不同了，如图 5-15 所示，所以在编码的时候需要同时设定 DTS 和 PTS 两个数据了。一般的，只要视频流中含有 B 帧就会导致 DTS 和 PTS 不相同。

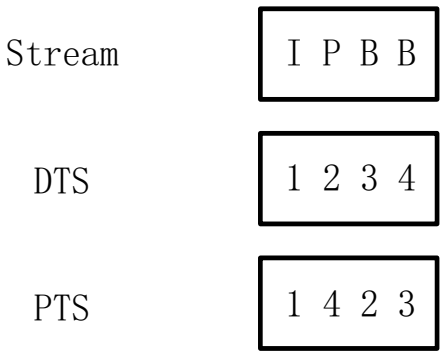

图 5-15 DTS 和 PTS 不同情况

综上原因，要播放视频内容需要获取到视频内容的 PTS，这里获取 PTS 所调用的函数是 av_frame_get_best_effort_timestamp()，如果获取不到 PTS 就先设为 0，等后续处理。之所以采取这个方法，是因为 FFmpeg 内部会通过计算返回最恰当的 PTS 的值。因为 PTS 本身为序数，必须乘上时间基数（time_base）才是用秒表示的时间。用到一个工具函数 av_q2d()，它作用是将时间基数转化为双精度浮点数，所以最终的 PTS 也被转化成了双精度浮点类型。av_frame_get_best_effort_timestamp()函数很有可能得不到一个正确的 PTS，如果没有项目中是用一个 video_clock 的值来近似，video_clock 表示视频播放到当前帧时的已播放的时间长度，就是上一个读取到地 PTS 的值。

前面提到项目是要将视频同步到音频的播放上来，所以需要获得音频的播放时间。在函数 audio_decode_frame()中解码新的 AVPacket 之后即可获得对应的时间属性 audio_clock，但是由于一个 AVPacket 中可以包含多个帧，AVPacket 中的 PTS 比真正的播放的 PTS 可能会早很多，所以在需要通过采样率来更新每个 AVPacket 中的播放时长。

有了视频帧的 PTS 和作为播放基准的音频播放时长 audio_clock, 就可以将视频同步到音频了。同步的步骤如下：

1、用当前帧的 PTS 减去前一播放帧的 PTS 得到一个延迟时间。

2、用当前帧的 PTS 和 audio_clock 进行比较，来判断视频的播放是超前还是慢了。

3、根据判断结果，设置播放下一帧视频的播放时间。

需要注意的事，项目是针对监控系统的产生的流媒体播放，有一大部分监控摄像头没有采集声音的功能，所以流媒体中很有可能没有音频，所以在视音频同步和播放的时候需要先判断数据流中是否存在音频，如果没有的直接按照视频解码后的 PTS 进行播放。

5.3 本章小结

这一章对项目的详细设计与实现进行了详细阐述，包括表示层、核心模块和使用方法等。表示层是针对接入开发者对播放框架的交互进行了介绍，核心模块主要对播放框架的解封装模块、解码模块、图像渲染模块、YUV 转 RGB 的截图模块、音频渲染模块、和音画同步模块展开了说明，下一章将介绍播放器的功能和性能进行测试。

第六章 测试与分析

第五章给出了播放器的详细设计与实现，本章将对本次项目进行测试分析，验证本次播放器框架是否符合项目需求。

本次测试利用论文中完成的播放器框架实现了一个只有视频播放功能的 Demo 应用，如图 6-1 所示。性能测试主要是通过开发工具 Xcode 的性能调试组件监听播放视频时 CPU 和运行内存的使用情况；因为无法通过 Xcode 监听其他厂商的产品，所以横向对比测试的方法是采用通过红外温度传感器检测手机的温度来体现。

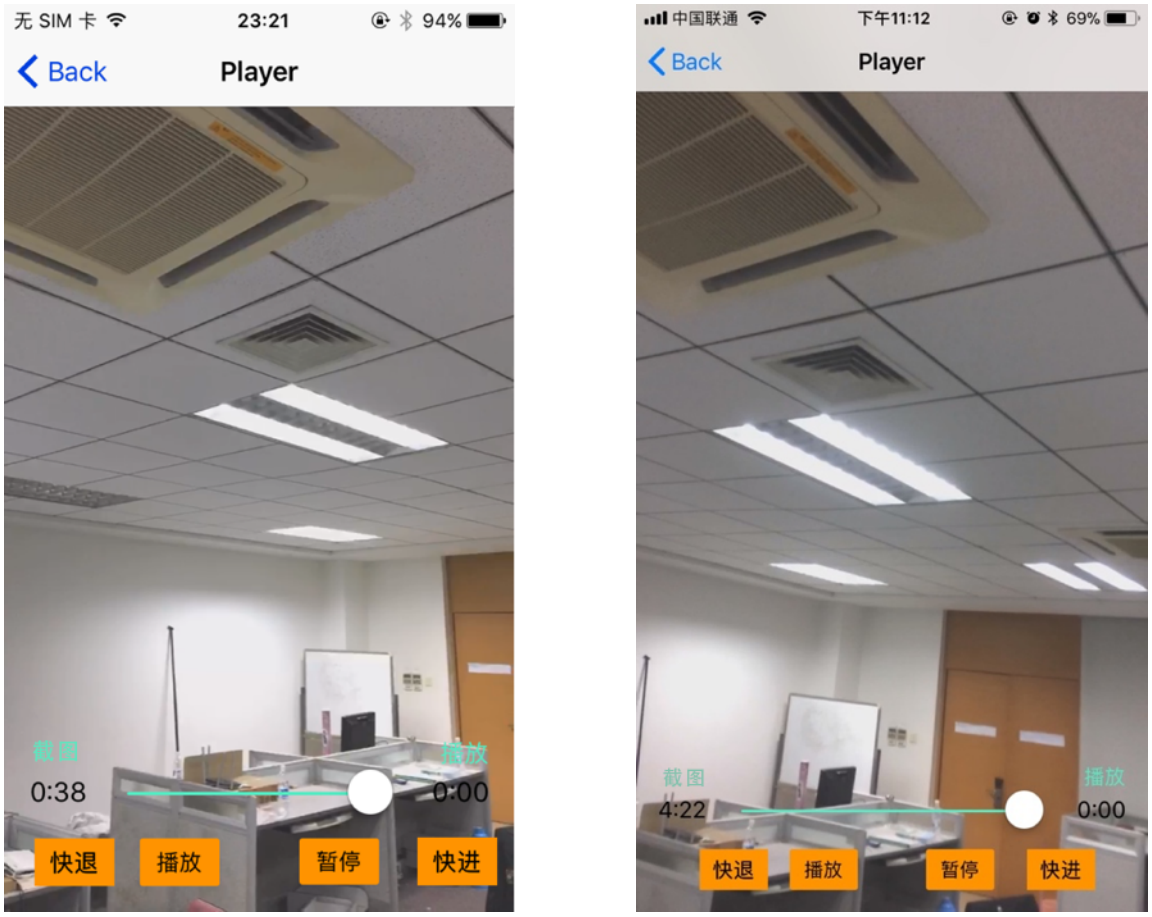

图 6-1 播放器在两台设备上的 Demo (左：iPhone5s，右：iPhone6s)

6.1 测试环境

服务器：项目所在公司的视频流服务器
视频码流：480p、720p、1080p RTSP 视频流
网络：8M WiFi 局域网
应用平台：iPhone 5s、iPhone 6s
两款测试手机具体参数如表 6-1 所示：

表 6-1 测试手机参数

	iPhone 6s	iPhone 5s
系统	iOS 11.1	iOS 10.1.1
RAM 内存	2GB	1GB
CPU	A9	A7
核心数	双核	双核

6. 2 播放框架测试

通过概要设计和详细设计可知，播放器框架的表示层是依据 Objective-C 语言和规范来设计的，在封装接口上完全符合 iOS 应用开发的习惯；且采用的都是 iOS 8 以来推出的新的系统接口；从开放的头文件 HKLivePlayerAction.h 可以找到所定义事件的键值和所有接口返回的状态代码，表 6-2 是对播放时出现的突发事的测试。

表 6-2 突发事件和错误测试

事件	测试步骤	测试结果	是否符合需求
(1)视频加载时点击了 Home 键	设定播放地址进入播放页面，点击播放按钮，开始加载时点击 Home 键	手机退回主页，视频停止缓冲，再次进入时播放按钮处于暂停状态	是
(2)视频播放时点击了 Home 键	视频正在播放时点击 Home 键	视频声音立即停止，手机退回主页，再次进入时播放已暂停，播放按钮处于暂停状态	是
(3)视频加载时有电话接入	设定播放地址进入播放页面，点击播放按钮，开始加载时通过另一部手机拨打测试手机	手机立即进入接听电话页面，退出通话页面后，播放按钮处于暂停状态	是
(4)视频播放时有电话接入	视频正在播放时通过另一部手机拨打测试手机	手机立即进入接听电话页面，视音频停止播放，退出通话页面后，播放按钮处于暂停状态	是
(5)视频加载时出现网络故障	设定播放地址进入播放页面，点击播放按钮，开始加载时关掉 WiFi 信号发射设备	缓冲画面消失，弹出“网络错误”提示框，提示：“请检查网络”	是
(6)视频播放时出现网络故障	视频正在播放时关掉 WiFi 信号发射设备	手机播放一段时间后播放停止，弹出“网络错误”提示框，提示：“请检查网络”	是
(7)视频播放时服务器关闭	视频正在播放时关闭流媒体服务器	手机播放一段时间后播放停止，弹出“网络错误”提示框，提示：“请检查网络”	是

通过表 6-2 测试结果和项目的详细设计可知播放器在框架设计上满足需求，能够完善处理突发事件，而且采用最新的系统接口，对将来的设备能够有更好的兼容性。

6.3 视频播放测试

6.2.1 视频播放功能测试

因为直播是将流媒体进行实时广播，而点播是播放存储在云端的流媒体资源，两者的区别在于直播无法实现对流媒体进行定点播放、快进和快退功能，所以视频播放功能地测试时需要将两者分开测试，测试结果如表 6-3 和表 6-4 所示。

表 6-3 视频点播功能测试

功能	测试步骤	测试结果	是否符合需求
(1)视频播放功能	设定 RTSP 点播播放地址进入播放页面，点击播放按钮	视频画面缓冲 1~2 秒钟之后开始播放，进度条开始向前移动	是
(2)暂停播放功能	执行功能(1)之后对正在播放的视频点击暂停按钮	视频立即暂停播放，进度条停止移动	是
(3)继续播放功能	执行功能(2)之后对暂停的视频点击播放按钮	视频立即开始播放，进度条继续移动	是
(4)定点播放功能	执行功能(1)之后拖动视频进度条	停止拖动进度条时当前播放状态立即停止，缓冲一秒之后立即在进度条指示位置开始播放	是
(5)快进功能	执行功能(1)之后点击快进按钮	当前播放状态立即停止，进度条也立即跳到对应位置，缓冲一秒后立即在前进 5 秒位置处播放	是
(6)快退功能	执行功能(1)之后点击快退按钮	当前播放状态立即停止，进度条也立即跳到对应位置，缓冲一秒后立即在后退 5 秒位置处播放	是
(7)截图功能	执行功能(1)之后点击截图功能按钮	播放页面上弹出“Player 需要您的同意才能添加照片”，点击“好的”按钮后截图保存到相册	是
(8)播放非 RTSP 地址的视频	设定 HTTP 点播播放地址进入播放页面，点击播放按钮	播放页面上弹出“播放地址错误”提示框，提示“只支持 RTSP 链接播放”	是
(9)播放错误的 RTSP 地址的视频	设定错误的 RTSP 点播播放地址进入播放页面，点击播放按钮	播放页面上弹出“播放地址错误”提示框，提示“请输入正确的播放地址”	是

表 6-4 视频直播功能测试

功能	测试步骤	测试结果	是否符合需求
(1)视频播放功能	设定 RTSP 直播播放地址进入播放页面，点击播放按钮	视频画面缓冲 1~2 秒钟之后开始播放，进度条直接跳到最后	是
(2)暂停播放功能	执行功能(1)之后对正在播放的视频点击暂停按钮	视频立即暂停播放	是
(3)继续播放功能	执行功能(2)之后对暂停的视频点击播放按钮	缓冲 1~2 秒之后，视频开始播放	是
(4)截图功能	执行功能(1)之后点击截图功能按钮	播放页面上弹出“Player 需要您的同意才能添加照片”，点击“好的”按钮后截图保存到相册	是
(5)播放非 RTSP 地址的视频	设定 HTTP 直播播放地址进入播放页面，点击播放按钮	播放页面上弹出“播放地址错误”提示框，提示“只支持 RTSP 链接播放”	是
(6)播放错误的 RTSP 地址的视频	设定错误的 RTSP 直播播放地址进入播放页面，点击播放按钮	播放页面上弹出“播放地址错误”提示框，提示“请输入正确的播放地址”	是

表 6-3 和表 6-4 中对播放框架在点播和直播方面进行了播放功能的测试，除网络环境导致缓冲时间不一致，基本上所有的功能都符合需求分析剔除的播放功能需求。

6. 2. 2 支持特定编码格式的视音频解码测试

本节主要对支持特定编码格式解码进行测试，测试的方法是使用播放器 Demo 分别对不同编码格式的视音频进行播放，测试结构如表 6-5 所示。

表 6-5 不同编码格式测试

功能	测试步骤	测试结果	是否符合需求
(1)h.264 视频测试	设定 h.264 格式视频播放地址进入播放页面，点击播放按钮	视频画面缓冲 1~2 秒钟之后开始播放，音频同步播放	是
(2)MPEG-4 视频格式测试	设定 MPEG-4 格式视频播放地址进入播放页面，点击播放按钮	视频画面缓冲 1~2 秒钟之后开始播放，音频同步播放	是

表 6-5 是播放器框架对 h.264 和 MPEG-4 格式编码视频流的测试结果，两者都符合功能需求的预期。需要注意的是，因为音频编码格式只有 AAC 一种，所以没有单独对音频编码格式进行测试，不过在视频测试过程中音频也能正常播放，所以对于 AAC 格式编码的音频，播放器也完全支持。

6.4 性能测试

6.4.1 播放器消耗 ROM 空间测试

播放框架所使用的 ROM 空间在封装之后是 9.8MB，使用播放框架完成的 Demo 应用安装包占用的 ROM 空间是 20.6MB；表 6-6 是业界同类产品 使用 ROM 空间的对比。

表 6-6 业界同类产品安装包使用 ROM 空间对比

应用名	测试 Demo	暴风影音	OPlayer	MoliPlayer	VLC	QQ 影音
占用 ROM 空间	20.60MB	70.34MB	51.6MB	32.00MB	48.32MB	35.76MB

上述播放器应用的主要功能都是视频播放，而测试 Demo 也只有视频播放的功能，尽管细节功能可能不一致，但是在 ROM 空间上也具有一定的参考作用，说明项目中对播放组件的精简达到的效果符合预期。

6.4.2 播放器 Demo 性能消耗测试

1、应用性能测试

在播放测试过程中在两台手机上都能正常运行，长时间播放也不会出现崩溃等现象。直播视频大概缓冲 3 秒左右开始播放，在网络状态稳定的情况下能正常播放。当网络情况不理想时播放器自动进入缓冲状态，缓冲完成后会自动继续播放。测试时发现在打开播放器的一瞬间 CPU 的使用率会突然上升到一定高度，很快回到正常播放时又会回到相对稳定状态，在持续播放过程中，内存的使用也是在一定范围内波动，没有随着播放的而持续增大，具体 CPU 和内存使用情况如表 6-7 所示。

表 6-7 播放 RTSP 流媒体时手机性能消耗

	CPU 消耗	未打开播放器 RAM 内存消耗	打开播放器 RAM 内存消耗
iPhone 6s	1%~4%	16.7MB	24.5MB
iPhone 5s	2%~6%	7.4MB	18.1MB

表 6-7 中未打开播放器 RAM 内存指的是在未进入播放页面应用所使用的内存，打开播放器 RAM 内存就是进入到了播放界面之后所使用的内存。所以简单估算播放器在播放时在 iPhone 6s 和 iPhone 5s 上所使用的内存分别是 13.8MB 和 10.6MB，在 CPU 和运行内存的使用都符合要求。不过在测试的时候发现，尽管 iPhone 6s 的内存和处理器都要优于 iPhone 5s，但是在使用内存上 iPhone 5s 却比 iPhone 6s 使用的要少。猜测原因可能有两个：一是两台手机的操作系统并不完全一致，可能是操作系统的的差异，导致 iPhone 5s 所使用的内存要小；第二个原因，可能是苹果公司针对内存尺寸小的手机进行了更多的优化，尽可能使内存较小的手机也能正常运行，具体原因需要进一步的研究。

2、横向对比测试

因为获取不到市面上其他厂商产品的开发版，无法利用 Xcode 等工具监听这些应用的 CPU 和运行内存的使用情况，所以在与各厂商产品的横向对比时论文采用的方法是利用红外测温机获取不同应用 App 播放时手机 CPU 的温度，通过 CPU 的温度对比来体现不同应用在播放视频时对 CPU 的消耗。

测试的方法是在稳定的室温 24 摄氏度下，播放 20 分钟的 1080p 的高清视频，播放结束后立即用红外探测器测试 CPU 的温度，表 6-8 和 6-9 是针对不同编码格式视频的测试的结果。

表 6-8 播放 h.264 格式视频后 CPU 的温度

	测试 Demo	米家	乐橙	MoliPlayer	VLC	QQ 影音
iPhone 6s	36.4℃	42.9℃	46.6℃	48.5℃	40.8℃	41.0℃
iPhone 5s	39.1℃	43.8℃	48.1℃	51.4℃	41.3℃	41.9℃

表 6-9 播放 MPEG-4 格式视频后 CPU 的温度

	测试 Demo	米家	乐橙	MoliPlayer	VLC	QQ 影音
iPhone 6s	40.8℃	47.9℃	49.4℃	50.5℃	41.6℃	42.9℃
iPhone 5s	42.9℃	48.6℃	50.4℃	52.0℃	42.8℃	43.5℃

通过表 6-8 和 6-9 可以看出，h.264 编码格式的视频播放后 CPU 的温度普遍低于 MPEG-4，可能原因是编码方式的差异导致。通过对比表 6-8 和 6-9 中数据可以明显地看出，测试 Demo 在播放高清视频后手机机身的温度都低于或持平于其他厂商的产品。尽管影响手机 CPU 温度的因素有很多，但是上述测试数据也从一定程度上说明了本项目在针对降低 CPU 的消耗和提升产品性能的效果。

6. 4. 3 播放器硬件解码性能测试

此处通过测试硬件解码播放和软解码播放编码格式为 h. 264 流媒体视频，使用硬件解码的效果在上一个测试中已经基本体现，这一节主要测试在相同条件下使用软件解码的效果，具体播放情况如表 6-10 所示。

表 6-10 软解码和硬解码手机功耗

	硬解码 CPU 消耗	软解码 CPU 消耗	硬解码 RAM 内存消耗	软解码 RAM 内存消耗
iPhone 6s	1%~4%	24%~26%	24.5MB	37.5MB
iPhone 5s	2%~6%	25%~30%	18.1MB	27.6MB

从 6-10 中可以看出采用硬件解码在 CPU 和 RAM 内存上的使用上更占有优势，因为硬件解码将大量重复的运算从 CPU 中转移，优化了播放器的性能。在两台手机上 CPU 的使用效率上硬解码比软件码提升了 2300%~3000%；RAM 使用效率提升了 52%~53%，采用硬件解码的方式完全符合项目需求。

6. 4. 4 视频渲染性能测试

在测试是否是 OpenGL ES 渲染时项目是在原有的播放库基础之上将图像渲染改为系统自带的图像控件 UIImageView 显示。尽管使用 UIImageView 更加方便快捷，但是在硬件局限性比较大的移动设备上，采用此种方式显然是不合适的。具体测试数据如表 6-11 所示。

表 6-11 不同图象渲染方式时手机功耗测试

	OpenGL ES 绘图 CPU 消耗	UIImageView 绘图 CPU 消耗	OpenGL ES 绘图 RAM 内存消耗	UIImageView 绘图 RAM 内存消耗
iPhone 6s	1%~4%	14%~19%	24.5MB	35.4MB
iPhone 5s	2%~6%	15%~21%	18.1MB	24.6MB

从表 6-11 数据中可知，相对使用系统控件显示画面，使用 OpenGL ES 框架绘制渲染降低了 35%~40%的 CPU 消耗；而且使用 UIImageView 显示图像需要频繁更新数据，对 RAM 内存的占用也比 OpenGL ES 高 36%~44%。并且在测试中可以看到，使用 OpenGL ES 渲染图像在播放时帧一直保持在苹果设备的最高刷新率 FPS 60，未会出现卡顿的情况。综上所述项目的视频渲染性能也完全符合预期。

6.4.5 截图性能测试

截图性能测试的方法是在播放器播放视频时进行截图操作，通过对比不使用算法、采用整型法、半查表法、全查表法和系统接口截图时的峰值 CPU 和运行内存的消耗来体现不同方法截图时性能。表 6-12 和表 6-13 是两台手机在不同截图方法下性能体现。

表 6-12 iPhone 6s 使用不同方法截图的性能体现

	不使用算法	系统方法	整型法	半查表法	全查表法
CPU 消耗	14%	6%	9.8%	8.0%	7.5%
RAM 内存消耗	30.0MB	26.4MB	27.2MB	27.5MB	28.0MB

表 6-13 iPhone 5s 使用不同方法截图的性能体现

	不使用算法	系统方法	整型法	半查表法	全查表法
CPU 消耗	16%	7%	11.0%	9.6%	8.2%
RAM 内存消耗	32.0MB	22.4MB	24.3MB	25.0MB	25.5MB

从表 6-12 和表 6-13 中可以看出，在多种方法中采用系统方法在性能上最有优势，但是系统方法获得的视频截图有多种弊端，不符合项目要求，只能作为参考。在两台手机中使用的优化算法后 CPU 消耗都比未使用任何优化算法时至少要提升 65%，RAM 内存消耗比未使用任何优化算法时至少提升了 57.14%，两者都符合项目预期。并且通过对比可以发现优化算法中 RAM 内存的消耗都很接近，但是全查表法在 CPU 消耗的上更接近于系统方法，所以可以认为采用全查表法实现截图功能内部的算法是完全符合预期的。

6.4.6 视频播放流畅性测试

视频播放的流畅性测试采用的测试方法是在稳定网络环境下分别针对不同分辨率的流媒体播放，测试结果如表 6-14 所示。

表 6-14 不同分辨率视频播放测试

	360P 视频播放	480P 视频播放	720P 视频播放	1080P 视频播放
iPhone 6s	缓冲 1 秒左右顺利开始播放，播放过程中无卡顿现象	缓冲 1 秒左右顺利开始播放，播放过程中无卡顿现象	缓冲 1~2 秒顺利开始播放，播放过程中无卡顿现象	缓冲 2 秒左右顺利开始播放，播放过程中无卡顿现象
iPhone 5s	缓冲 1 秒左右顺利开始播放，播放过程中无卡顿现象	缓冲 1 秒左右顺利开始播放，播放过程中无卡顿现象	缓冲 1~2 秒顺利开始播放，播放过程中无卡顿现象	缓冲 2 秒左右顺利开始播放，播放过程中无卡顿现象

从表 6-14 的测试结果可知，播放器框架对 720P 和 1080P 视频均可流畅播放，完全符合预期；需要注意的是，流畅播放需要良好的网络环境的支持，否则播放效果可能不理想。

6.4.7 稳定性测试

播放器框架稳定性测试采用方法是在稳定网络环境和充足的电量情况下长时间直播视频，测试结果如表 6-15 所示。

表 6-15 不同时长直播测试

	直播 20 分钟，播放 Demo 状态	直播 50 分钟，播放 Demo 状态	直播 100 分钟，播放 Demo 状态	直播 300 分钟，播放 Demo 状态
iPhone 6s	正常播放	正常播放	正常播放	正常播放
iPhone 5s	正常播放	正常播放	正常播放	正常播放

从表 6-15 中可知两台测试手机在不同时长直播测试中一直处于正常播放状态，未出现应用崩溃、手机死机等现象，说明播放器框架在稳定性上符合性能需求。

6.4.8 可扩展性

从项目的概要设计和详细设计可知，本播放器框架采用的是模块设计的方法，降低了功能之间的耦合性，同时增大播放器框架的可扩展性；视频画面采用 OpenGL ES 渲染时，特别针对 3D 视频扩展进行了模块拆分，而且公用的模块已经完成开发，对于 3D 画面的实现只需要添加球状显示模型，降低了扩展的难度，甚至对于其他常见画面变换也可以实现。综上所述，本视频播放框架在针对 3D 画面渲染是有一定的可扩展性。

6.5 本章小结

本章针对播放器框架在需求分析中提出需求进行了测试，测试包括两个方面内容：第一点是播放器框架的测试，第二点是播放相关功能和性能的测试；其中播放框架采用 Objective-C 语言和 iOS 最新的接口实现符合需求，对突发事件的测试也跟需求分析一致；因为直播和点播的差异，播放相关的功能分别针对与点播和直播展开测试，测试结

果符合预期；播放性能方面针对性能需求进行了多项测试，测试结果也懂符合需求。综上所述，本次项目已经全部实现了项目启动时提出的多种需求，能无缝嵌入家校通系统中，实现监控视频的播放。

第七章 总结与展望

7.1 总结

移动互联网的发展催生了很多产业也使得很多产业开始转型寻找新的出路。本文项目来自国内某安防公司推出的新产品中，项目实现了在 iOS 平台播放 RTSP 流媒体的功能，满足了公司内部的项目需求。

整个项目的研究工作主要包含以下几个方面：

(1)通过分析研究 FFmpeg 框架的组成，提出了在支持单一流媒体协议、少量视音频编码格式的情况下，针对该框架的一种精简方案。该方案使得项目封装之后体量更小，竞争优势更强。

(2)设计实现了一个基于 RTSP 协议的 iOS 视频播放器系统，该系统在视频解码上优先使用硬件解码框架，提高了播放器在播放视频的性能，减少了在播放视频时手机机身发热情况的产生。

(3)针对在截图模块中可以将 YUV 数据转化为 RGB 的一些常用算法进行了对比分析，最终采用全查表法，通过对算法的设计实现，将需要建立 YUV 转 RGB 的映射表，利用直接读取方式取代多次计算，从而可以降低对性能的消耗。

(4)采用 OpenGL 在移动设备上的分支 OpenGL ES 在 iOS 设备上高效利用 GPU 的实现渲染工作，减轻了 CPU 的功耗，提高了设备性能。

7.2 展望

本次项目的开发完成了项目的需求，而且在实现播放 RTSP 流媒体视频的基础之上提高播放器的性能，并且简化后续调用的复杂性，在开发过程中更加方便使用。但是因为开发周期限制等客观原因，产品还有很多可实现或者可优化的地方，以下是对项目提出的展望：

播放器没有实现手动切换视频分辨率的功能，尽管服务器端会根据当前码流情况修改视频的清晰度，但是在客户端手动切换清晰度已经是常见的在线视频播放器的功能，希望能在下一个版本中得到实现。

现在安防公司推出的摄像头种类众多，类似于鱼眼摄像头或者全景摄像头，拍摄的画面属于半球形或者球形，采用此种播放方式肯定是达不到拍摄时的效果，需要采用 3D 模型进行渲染。而且随着无人机和 VR 摄像头等产品的盛行，播放 3D 视频的需求肯定会越来越大，所以有必要在当前的视频播放器上添加 3D 视频播放的功能。前面提到过 OpenGL 和 OpenAL 主要是针对 3D 场景设计的，项目播放器中恰好用到了这两个框架，所以针对 3D 场景视频播放的扩展应当比较容易实现。

现在的视频播放器仅仅提供了播放视频的功能，并没有反馈或者对视频采样端的控制，比如有一种叫做云台的产品上面会集成摄像头，功能可以通过云台控制摄像头的方向。目前播放器无法做到对云台等产品发送控制信号，为了完善播放器对公司产品的支持，还是有加上对视频采集端的控制的功能。因为项目中的视频解封装模块采用的是 FFmpeg 框架实现的，所以除此之外项目中并没有自行实现网络方面的内容，所以在实现对采集端的控制需要对网络模块进行编写，并对相关产品的接口进行深入研究。

在线视频播放是在移动设备上是最消耗数据流量的，对数据流量的使用进行优化将会大大增加产品的竞争力。但是因为项目周期紧、视频云服务器是由其他部门提供，所以没有实现优化数据流量的使用，希望在今后版本中能够跨部门合作，降低视频播放的数据流量消耗。

致谢

似乎瞬息之间研究生生涯已经接近尾声，还没有享受完在学校学习生活的我将要再一次面临毕业。在东南大学的这段时间是我目前专业技能提高最多的一段时间，对我职业生涯乃至人生的产生了巨大影响。在我从另外一个专业前来学习计算机科学和软件工程知识的时候，得到了很多来自学校和老师的帮助，在论文的最后我想借此机会对学校和老师以及在求学过程中帮助我的同学表示由衷的感谢。

首先要感谢我在学校的两位导师吉逸教授和宛斌博士，他们在我的求学期间给了我很多生活和学习上的帮助，两位导师严谨的工作态度深深地感染了我，让我受益良多，尤其是在准备毕业设计开题和论文撰写时候，两位老师不遗余力地给我提出建议帮助我修改开题报告和论文，在开题答辩时候帮我提前做好准备，让我能顺利完成论文，真的非常感谢两位导师。

接下来要感谢的是在企业中的校外导师彭鹏老师以及很多帮助过我的同事，彭鹏老师在实习期间帮我解决了很多技术上的难题，让我在编程方面有了很大的提升。各位同事在不同领域的能力和丰富的经验在各个方面都能提供很大的支持，让我的工作能顺利进行。实习的时间虽然不长，但是这是我步入企业的第一步，非常幸运能碰到如此善良热心的同事。

开题答辩时候的张家凤、何洁月、周晓宇和崇志宏老师对我的论文的开展进行了指导，指出了我论文中问题并提出了改进意见，帮助我更好地完成了论文的开题。

还要感谢翟玉庆老师、方治老师和徐学永老师在我论文的完成时提供了宝贵意见，对我论文的完成给予了很多帮助，非常感谢三位老师。

最后要感谢百忙之中对我的论文进行评审的各位老师。

参考文献

[1] Akhlaq M, Sheltami T R. RTSP: An accurate and energy-efficient protocol for clock synchronization in WSNs[J]. IEEE Transactions on Instrumentation and Measurement, 2013, 62(3): 578-589.

[2] Seeling P, Reisslein M. Video Transport Evaluation With H.264 Video Traces[J]. IEEE Communications Surveys & Tutorials, 2012, 14(4):1142-1165.

[3] Bailey M. Combining GPU data-parallel computing with OpenGL[C]// ACM SIGGRAPH. ACM, 2013:1-65.

[4] 董慧波. 基于 WEB 的家校通系统[D].中国海洋大学,2009.

[5] 陈中军. 基于 Flex 的家校通系统[D].吉林大学,2010.

[6] 侯沛德. 基于 IOS 的视频监控客户端软件的设计与实现[D].兰州大学,2014.

[7] 张永芹. 适用于智能交通手机监控系统的流媒体技术方法研究与实现[D]. 南京邮电大学, 2013.

[8] AlTairi, Zaher Hamid,Rahmat, Rahmita Wirza,Saripan, M. Iqbal,Sulaiman, Puteri Suhaiza. Skin segmentation using YUV and RGB color spaces[J]. Journal of Information Processing Systems, 2014, 10(2):283-299.

[9] Wang A, Wu W, Chen J. Comparison and improvement for RGB and YUV color space based on K-means in image segmentation[J]. International Journal of Chemical Kinetics, 2013, 44(1):1-1.

[10] Zhao P, Li J, Xi J, et al. A Mobile Real-Time Video System Using RTMP[C]// Fourth International Conference on Computational Intelligence and Communication Networks. IEEE, 2012:61-64.

[11] 陈天骢. 基于 RTMP 与 SIP 的可视化交互系统研究与设计[D]. 上海交通大学, 2012.

[12] Aloman A, Ispas A I, Ciotirnae P, et al. Performance Evaluation of Video Streaming Using MPEG DASH, RTSP, and RTMP in Mobile Networks[C]// Ifip Wireless and Mobile NETWORKING Conference. IEEE, 2016:144-151.

[13] 刘峰. 基于 RED5 的 HLS 虚拟流媒体服务器部署方法研究与应用[D]. 西安电子科技大学, 2015.

[14] Pak D, Kim S, Lim K, et al. Low-delay stream switch method for real-time transfer protocol[C]// The, IEEE International Symposium on Consumer Electronics. IEEE, 2014:1-2.

[15] Andersson K, Dan J. Mobile e-services using HTML5[C]// Local Computer Networks Workshops. IEEE, 2012:814-819.

[16] 刘春华. 基于 HTML5 的移动互联网应用发展趋势[J]. 移动通信, 2013(9):64-68.

[17] 基于 Live555 的网络视频监控系统设计与实现[J]. 现代电信科技, 2012(12):38-42.

[18] Newmarch J. FFmpeg/Libav[M]// Linux Sound Programming. Apress, 2017.

[19] 吕少君, 周渊平. 基于 Live555 的实时流媒体传输系统[J]. 计算机系统应用, 2015, 24(1):56-59.

[20] Kim H T. Real-time Flame Detection Using Colour and Dynamic Features of Flame Based on FFmpeg[J]. 2014, 9(9):977-982.

[21] Molu M M, Goertz N. A comparison of soft‐coded and hard‐coded relaying[J]. Transactions on Emerging Telecommunications Technologies, 2014, 25(3):308-319.

[22] Li R, Saad Y. GPU-accelerated preconditioned iterative linear solvers[J]. Journal of Supercomputing, 2013, 63(2):443-466.

[23] Pan Z, Lei J, Zhang Y, et al. Fast Motion Estimation Based on Content Property for Low-Complexity H.265/HEVC Encoder[J]. IEEE Transactions on Broadcasting, 2016, PP(99):1-10.

[24] 姜博瀚. 移动互联网实时视频采集压缩技术研究[D]. 哈尔滨工程大学, 2014.

[25] Sellers G, Wright R S, Haemel N. OpenGL Superbible: Comprehensive Tutorial and Reference[M]. Addison-Wesley Professional, 2015.

[26] Marucchi-Foino R. Game and Graphics Programming for iOS and Android with OpenGL ES 2.0[M]. Wrox Press Ltd. 2012.

[27] 赵辉, 王晓玲. 计算机图形学:OpenGL 三维渲染[M]. 海洋出版社, 2016.

[28] Marroquim R. Introduction to GPU Programming with GLSL[C]// Tutorials of the Xxii Brazilian Symposium on Computer Graphics and Image Processing. IEEE Computer Society, 2009:3-16.

[29] Wang Y, Kim J G, Chang S F. Content-based utility function prediction for real-time MPEG-4 video transcoding[C]// International Conference on Image Processing, 2003. ICIP 2003. Proceedings. IEEE, 2013:I-189-92 vol.1.

[30] De Cuetos P, Guillotel P, Ross K W, et al. Implementation of adaptive streaming of stored MPEG-4 FGS video over TCP[C]// IEEE International Conference on Multimedia and Expo, 2002. ICME '02. Proceedings. IEEE, 2014:405-408 vol.1.

[31] Tamhankar A, Rao K R. An overview of H.264/MPEG-4 Part 10[C]// Video/image Processing and Multimedia Communications, 2003. Eurasip Conference Focused on. IEEE, 2015:1-51 vol.1.

[32] Wang F Y, Wang J C, Lu B. Study on the Application of Virtual Reality Technology in the Production of Industrial Products Based on OpenGL[J]. Applied Mechanics & Materials, 2013, 392:206-209.

[33] Draffan E A, Wald M, Halabi N, et al. A Voting System for AAC Symbol Acceptance[C]// ASSETS '15 Proceedings of the, International ACM Sigaccess Conference on Computers & Accessibility. ACM, 2015:371-372.