工程硕士
学位论文

基于RTSP协议的多源视音频实时直播系统的设计与实现

吕坤轩

廣西大學

二〇一四 年 十一 月

分类号 TN919.8 密级 公开

UDC

工程硕士学位论文

基于RTSP协议的多源视音频实时直播系统的设计与实现

吕 坤 轩

学科专业 计算机技术

指导教师 莫林教授

论文答辩日期 2014-11-21 学位授予日期 2014-12-31

答辩委员会主席 陈友初 教授级高级工程师

广西大学学位论文原创性和使用授权声明

本人声明所呈交的论文，是本人在导师的指导下独立进行研究所取得的研究成果。除已特别加以标注和致谢的地方外，论文不包含任何其他个人或集体已经发表或撰写的研究成果，也不包含本人或他人为获得广西大学或其它单位的学位而使用过的材料。与我一同工作的同事对本论文的研究工作所做的贡献均已在论文中作了明确说明。

本人在导师指导下所完成的学位论文及相关的职务作品，知识产权归属广西大学。本人授权广西大学拥有学位论文的部分使用权，即：学校有权保存并向国家有关部门或机构送交学位论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或其它复制手段保存、汇编学位论文。

本学位论文属于：

□保密，在　　年解密后适用授权。

☑不保密。

(请在以上相应方框内打“√”)

论文作者签名：吕婷轩　　日期：2015.1.6

指导教师签名：黄林　　日期：2015.1.6

作者联系电话：　　电子邮箱：

基于RTSP协议的多源视音频实时直播系统的设计与实现

摘要

得益于网络技术与多媒体技术的发展，流媒体技术已经成为近年的一个研究热点，被广泛应用于多于生活中的众多领域，如视频监控、视频会议、远程协助等。为了解决对多个离散场景的视音频实时共享，实现多场景同步实时观摩的问题，结合实际的项目，本文对网络直播系统进行了深入研究，主要工作内容如下：

(1) 提出一种用于多源视音频实时直播的软件解决方案。利用软件将取自多源的多路视频和音频分别整合到一起，输出一路标准编码、封装的视频流和音频流，实现自由选择的多个场景的同步记录、存储、发布和交换，实现视音频的网络共享。

(2) 分析并设计了基于RTSP协议的多源视音频实时直播系统。对系统各方面的需求进行了细致的分析，并根据系统的需求对系统结构从整体到局部的设计进行了分析和说明。

(3) 实现了基于RTSP协议的多源视音频实时直播系统。本文利用采集设备实现了视音频数据的采集，并对采集速度的控制问题进行了分析和探讨。文中设计并实现了视频图像的分屏拼合方案，对音频的混合进行了论述与实现，解决了视频和音频的多流合一问题。借助于FFmpeg解决方案，本文定义并实现了视频的H.264编码类、音频的AAC编码类及视音频复用类，解决了实时编码与存储的问题。通过对 live555 方案的修改与重写，本文实现了H.264与AAC实时数据源，实现了基于RTSP协议的直播服务器，实

现了视音频数据的封装、发送，并对发送前视音频的同步控制进行了分析与探讨。

本文系统已经应用于某公司某科技法庭项目，试运行效果良好，达到设计目标。

关键词：多源视音频　实时直播　多流合一　H.264　AAC FFmpeg live555

DESIGN AND IMPLEMENTATION OF THE REAL-TIME NETWROK LIVE BROADCAST SYSTEM FOR MUL-SOURCE VIDEO AND AUDIO BASED ON RTSP

ABSTRACT

Benefiting from the development and improvement of network and multimedia technology, streaming media technology has recently become a research hotspot, and widely been applied to many fields, for instance, video conference, video surveillance, remote assistance. In accordance to a certain actual project, this thesis has done thorough research into the network broadcast system, which is armed at dealing with the problem of sharing video and audio from several separated scenes in real time, and enabling multiple scenes to be viewed both in synchronization and in real time. The followed is the main work in this thesis.

Firstly, a software solution is put forward for real-time network live broadcast of multi-source video and audio in the thesis. With the method of software technology, multi-channel video streams from multiple sources can be integrated together, and so can audio streams. Moreover, the output integrated streams, including both a video stream and an audio stream, should be encoded and packed according to relevant standards, which ensures several selected scenes to be recorded, stored and released synchronously, and makes them shared through the Internet.

Secondly, this thesis has made an analysis and designed the structure of a real-time network live broadcast system based on RTSP for multi-source video and audio. In order to complete the system, requirement is analyzed detaily, which also helps to analyze and illustrate the design of the system structure from whole to part.

Finally, the thesis illustrates the implementation of the system above. First of all, capturing both video and audio is completed with capture devices, and the issue of controlling the capture speed is analyzed and discussed. As far as the issue of integrating multiple streams into one, this thesis presents a method for stitching several video images, and describes the method of audio mixing as well. What's more, with FFmpeg project, this thesis implements some classes, used for H.264 and AAC encoding, and storage. When the RTSP server is talked about, detailed analysis has been made for live555 project. This thesis modifies the project in some degree, and implements a class for H.264 real-time source and another class for AAC real-time source. Besides, the RTSP server platform is implemented. Before sending H.264 and AAC data, the paper discusses how to deal with inter-media synchronization.

The system implemented in the thesis has now been being applied to the technology court project of a certain company. Besides, it performs well during the trial running, and reaches the design target.

Key Words: multi-source video and audio; real-time network live broadcast; integrating; H.264; AAC; FFmpeg; live555

目录

第一章　绪论

1.1 项目背景

伴随着网络技术与多媒体技术的快速发展，流媒体(Streaming Media)应运而生[1]-[3]，它以流的方式在网络中传输音频、视频等多媒体数据[4]，接收端只需要进行短暂的缓冲，就可以播放。如今，该技术越来越受到人们的关注，在社会生活中发挥着越来越重要的作用，被广泛应用于视频监控、视频会议、远程协助、远程教育等众多方面。

现阶段，各种流媒体系统已经日益成熟悉，但需求的多样性决定的应用的多样性，一种流媒体系统很满足人们各种各样的要求。类似于法庭审判这种复合场景，有法官、书记员、原告、被告、陪审员、观众等多个子场景的视频和音频信息需要同步实时记录、存储、发布和交换，以确保能真实地还原庭审实况。传统的视频监控系统能过多个摄像机真实地记录各子场景的实况，但不能保证其同步性，也无法通过网络同时实时观摩全局场景。传统的视频会议系统能利用硬件设备实现多场景的复合，但需要专用的设备与软件，成本较高，不易于维护，同进不方便与其它流媒体系统进行交互。

本文课题的研究工作是某公司某智能科技法庭项目的重要组成部分，主要研究目的是利用软件技术将多个场景的视频画面和音频分别整合在一起，采用标准的编码、封装格式与流媒体协议，易于与其它流媒体系统或软件进行交互与对接，输出一路标准的视频复合流和一路音频复合流，这一路视频复合流和音频复合流就包含了多个场景的视频和音频信息，实现自由选择的多个场景的同步记录、存储、发布和交换，实现视音频的网络共享，保证了对全局场景实况的再现。

1.2 研究现状

流媒体是一种新的媒体传输方式，而不是一种新的媒体形式，主要包括顺序的和实时的两种传输方式[5]。其中，前者是对多媒体数据的顺序下载，多用于多媒体文件的下载播放[6]，典型的应用像百度影音、快播、迅雷看看等；实时流式传输总是实时传送[7]，多媒体数据到达服务器后不做驻留即被传输出去，相对于顺序流，实时流需要适应带宽调整传输速率，以保证实时性，典型的应用如视频会议、QQ 和 YY 等视音频聊天工具。

流媒体技术是一种综合了音频、视频编解码以及传输等多方面的技术。下面将从视音频的编解码与传输等两方面简要介绍一下流媒体在国内外的研究现状。

(1) 在编解码方面

未经编码的多媒体数据含有大量冗余信息，需要进行压缩编码，去除掉原始视音频数据中的冗余信息，减少数据量，以便存储与传输。

目前，视音频的编码已有国际标准，制定发布这些标准的组织主要有 ISO(国际标准

化组织)和 IEC(国际电工委员会)的动态图像专家组 MPEG(Moving Picture Experts Group)、ITU-T(国际电信联盟电信标准化部门)的 VCEG 视频编码专家组(Video Coding Experts Group)，以及由 MPEG 与 VCEG 共同成立的联合视频工作组 JVT(Joint Video Team)。其中，MEPG 成立于 1988 年，致力于开发视、音频的压缩技术，至今已经制定了 MPEG-1、MPEG-2、MPEG-3、MPEG-4、MPEG-7[8]及 MPEG-21[9]等多个标准，这些标准被广泛用于有线电视网(CATV)、广播、电缆网络以及卫星直播等领域[10]。VCEG 到目前为止也开发制定了一系列视频通信协议和标准，包括 H.261 视频会议标准，和其后续本 H.263、H.263 +、H.263 ++、H.264[11]等，最新的标准是 H.265[12]。其中 H.264 与 H.265 是由 MPEG 与 VCEG 联合成立的联合视频组 JVT(Joint Video Team)制定发布。在这些视频编码标准中，MPEG4 和 H.264 标准是应用较广泛的两种，H.264 标准在同等条件下比 MPEG4 具有更好的网络适应性，因而在实时流媒体特别是视频监控中得到较多的应用。

除此之外，在国内，我国具备自主知识产权的第二代信源编码标准 AVS(Audio Video Coding Standard)[13]已经成为国家标准。

(2) 在流媒体服务器与传输协议方面

随着流媒体技术的快速发展与广泛普及，众多 IT 科技巨头都竞相推出自己的流媒体有关技术标准与产品，以在应用市场中占据一席之地[14]。其中，Apple 公司开发 QTSS(QuickTime 流媒体服务器)[15]，Microsoft 公司的 WMS（Windows 媒体服务器）[16]，Real Network 公司发布的 RSS（Real System 服务器）[17]，这三大流媒体服务器软件占据着市场的主导地位。

为使视频、音频等多媒体数据适应网络传输，互联网工程任务组 (Internet Engineering Task Force, IETF)于 1996 年在 RFC1889 中公布了 RTP 协议(Real-time Transport Protocol, 实时传输协议)[18]及其姊妹协议 RTCP 协议(Real-time Transport Control Protocol, 实时传输控制协议)[19]，后来在 RFC3550 中进行更新；同时，Real Network 公司在上世纪 90 年代发布自己的私有协议 RDT(Real Data Transport)[20]，用于视、音频数据，著名的播放器 Real Player 就是使用该协议。Adobe Systems 公司也为 Flash 播放器和服务器开发了视频、音频和数据的开放传输协议 RTMP(Real Time Messaging Protocol)[21]，通过 Flash Player 客户端，该协议也在市场中占稳脚根，成为流媒体技术发展的重要标杆[22]。

为了在娱乐交互系统中流媒体服务器，1996 年 Real Networks 公司、Netscape 公司以及 Columbia University 共同开发了 RTSP(Real Time Streaming Protocol)[23]协议，并提交给 IEFE，IEFE 的 Multiparty Multimedia Session Control Working Group (MMUSIC WG)于 1998 年在 RFC2326 中将该协议定为标准协议并正式发布。目前，RTSP2.0 正在紧锣密鼓地研发中，用于替换 RTSP1.0。

1.3 论文的结构和主要内容

本论文的主要工作是设计并实现一个基于RTSP协议的多源视音频实时直播系统，全文共分为六章，其主要内容如下：

第一章，绪论。简单介绍论文的研究的项目背景与意义，同时简要介绍流媒体有关的国内外的研究现状，最后简述本文的结构与主要内容安排。

第二章，流媒体相关技术概述。简要介绍本论文研究涉及到的视频、音频编解码技术，流媒体数据传输所使用的传输与控制协议。这些技术是本论文的工作基础，介绍这些技术能帮助对论文有关内容的理解。

第三章，基于RTSP协议的多源视音频实时直播系统的结构设计。分析系统的需求与设计，说明系统的组织结构，这些内容能帮助更好地理解本文整体研究内容。

第四章，视音频前端采集处理模块的设计与实现。该章说明系统输入模块的设计与实现，具体说明多源视频和音频的采集、多流合一处理、编码和解码、视音频复用存储实现方法和过程。

第五章，基于live555框架的视音频直播服务器的设计与实现。该章主要对live555进行较深入地分析研究，并在其基础进行改进，实现视音频的实时发布，具体说明此过程的设计与实现。

第六章，系统功能测试。该章主要对该多源实时发布系统的功能的完全性与稳定性进行测试。

第七章，总结与展望。该章总结本论文的工作、遇到的问题、有待解决的问题，以及具体的改进意见。

1.4 本章小结

本章是背景章节，简要说明了本论文的项目背景、有关技术的研究现状，并对论文的结构与主要内容的安排进行了简要的概括介绍，这些内容对了理解本文内容有一定的帮助。

第二章　相关技术概述

本文工作涉及了视频和音频处理、视音频的编码和解码、以及实时流媒体数据的传输，因此，本章将具体介绍一下视音频的编解码技术、图像与声音的处理技术以及视、音频等多媒体数据的传输协议。

2.1 流媒体传输与控制协议概述

多媒体数据流式传输的实现，以及传输质量的保证，都需要相应的协议提供支持[24,25]，常用的流媒体协议主要RTSP、RTP/RTCP、RDT等。除此之外，流媒体技术还包括会话描述协议(Session Description Protocol, SDP)。流媒体各种协议的层次关系如图2-1所示。

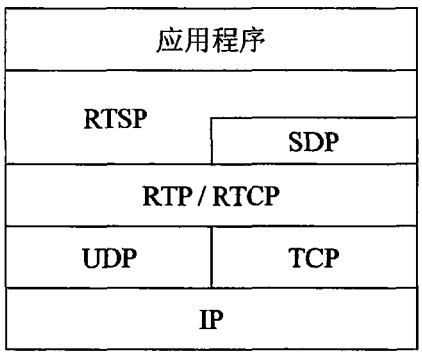


图 2-1 流媒体协议层次

Fig.2-1 Hierarchy of streaming protocol

2.1.1 RTSP协议简介

实时流协议RTSP是一种基于文本的应用层协议，为C/S模型，采用请求-应答的交互方式，用于对流媒体服务器进行远程控制。协议本身不提供流媒体数据的发送，流媒体数据的发送是通过使用RTSP协议下层的RTP协议，或其他传输协议如RDT等。RTSP请求通常使用TCP协议发送，也可以使用UDP协议发送。表2-1给出了RTSP协议常用的请求的具体描述[26]，其中C表示Client(客户端)，S表示Server(服务器)；P表示Presentation(描述)，S表示Stream(媒体流)。图2-2为以这五种请求的应答交互为例的时序图，能帮助我们更好地去理解RTSP协议客户端与服务器的交互过程。

表 2-1 RTSP 请求说明

Table 2-1 The instroduction of RTSP request

方法	方向	对象	需要	含义
DESCRIBE	C->S	P,S	推荐	请求媒体描述信息；
ANNOUNCE	C->S,S->C	P,S	可选	C->S:发送描述或媒体对象； S->C:更新会话描述；
GET_PARAMETER	C->S,S->C	P,S	可选	请求检索指定的表示或媒体流的参数值；
OPTIONS	C->S,S->C	P,S	必须 (S->C:可选)	请求服务器提供的可用方法
PAUSE	C->S	P,S	推荐	请求暂时媒体数据中断传输；
PLAY	C->S	P,S	必须	请求发送媒体数据；
RECORD	C->S	P,S	可选	记录媒体数据；
REDIRECT	C->S	P,S	可选	通知客户端连接到另一个服务器位置；
SETUP	C->S	S	必须	确定传输机制，建立视音频会话连接；
SET_PARAMETER	C->S,S->C	P,S	可选	给指定的表示或媒体流设置参数；
TEARDOWN	C->S	P,S	必须	请求终止传输，释放相关资源；

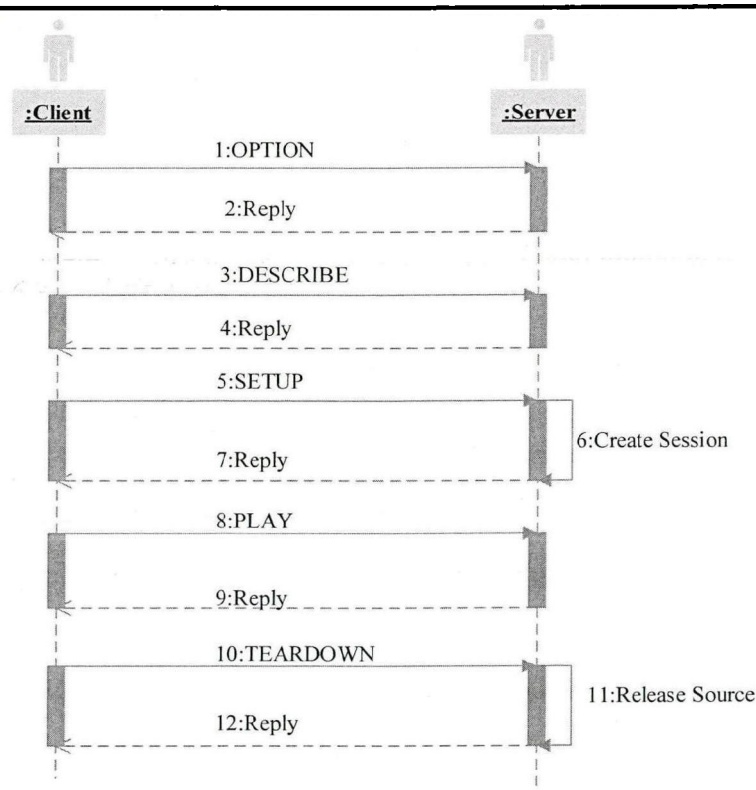


图 2-2 RTSP 协议客户端/服务器请求应答时序图

Fig 2-2 The sequence chart of interation between server and client

2.1.2 RTP / RTCP 协议介绍

1. 实时传输协议—RTP

RTP 是在网络上为实时的多媒体数据提供传输服务的传输层协议，位于 UDP 和 TCP 协议之上，可以使用 TCP、UDP 等下层协议，能够提供一对一或一对多的传输服务，也能通过协议的时间戳字段提供时间信息，并依靠时间信息实现流同步。如果下层网络支持，RTP 也支持使用多播分发机制将数据发送到多个目的地。需要注意的是 RTP 协议没有提供任何机制以确保传输的实时性和服务质量，这些都需要低层的服务来完成。同时，RTP 协议不保证数据的顺序传输，也不要求下层网络顺序传输数据。接收端需要根据 RTP 协议中的序列号字段对负载数据进行重组。

RTP 数据包包括协议头和负载两部分。RFC3550 文件中给出了 RTP 协议头字段的详细信息，其中前 12 个字节是固定的，CRSC identifiers 字段是可选的；负载部分为要传输的媒体数据。图 2-3 给出了 RTP 报文的格式[27]。

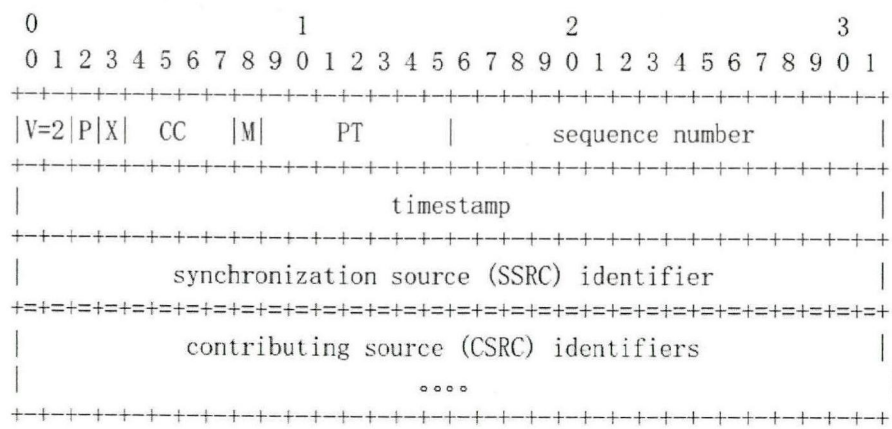


图 2-3 RTP 报文头格式

Fig. 2-3 The format of RTP packet header

各字段的含义如下：

1) V(Version)：2bit，RTP 版本标识；当前协议版本为 2。

2) P(Padding)：1bit，填充字段；该字段为 1 时，表示负载部分的尾部包含填充数据，其长度由填充部分最后的字节表示。

3) X(Extension)：1bit，扩展字段；若该字段为 1，表示 RTP 固定头后面有头部扩展字段。

4) CC(CSRC Count)：4bit,CSRC 计数；该字段表示跟在固定头后面的 CSRC 的数量，取值 0—15。

5) M(Marker)：1bit,标识字段；常用于确定帧边界。

6) PT(Payload Type)：7bit，负载类型；该表示 RTP 负载部分的多媒体数据的类型。

7) Sequence Number：16bit，序列号；该字段表示 RTP 分组的序号，每发送一个数

据包该值加1，初值随机产生，此字段用于丢包检测和数据重组。

8) Timestamp：32bit，时间戳；该字段表示负载数据的采样时刻，为相对时间，若多媒体数据帧被分割成若干分片，那么属于同一帧的分片的时间戳是一样的，此字段提供时间信息，配合RTCP中的绝对时间戳可进行流同步。

9) SSRC：32bit，同步源标识字段；该字段表示当前RTP会话的ID，每个会话的同步源标识都是随机的、唯一的。

10) CSRC：32bit,贡献源列表；该字段表示RTP包中负载部分的贡献源，数量由CC字段给出，最多16个。

2. 实时传输控制协议—RTCP

RTCP协议是与RTP一起工作的控制协议，协助RTP协议完成对流量与拥塞的控制。在RTP会话的过程中，每一个会话的成员都会周期性地向其他所有参与者发送RTCP数据包，如图2-4所示，这些RTCP控制信息包中包含已发送的和丢失的数据包的统计信息。服务器可以根据接收端反馈的RTCP数据包了解网络传输情况和丢包率，从而调整发送速率及其他媒体参数；接收端可以根据发送端发送的RTCP数据包了解网络抖动，进行流同步。

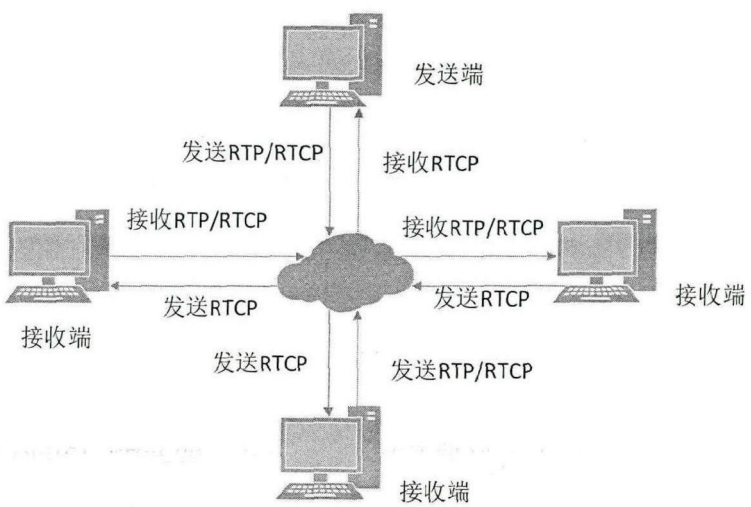


图2-4 RTCP报文收发示意图

Fig. 2-4 Reception and sending of RTCP packet

根据携带的控制信息不同，RTCP数据包共可分为5种，分别是

1) SR：发送者报告(Sender Report)，由发送端发出，提供NTP时间戳、RTP时间戳、发送的字节数等信息。

2) RR：接收者报告(Receiver Report)，由接收端发出，提供SSRC标识符、累计丢包数等统计信息，供发送端参考使用。

3) SDE：源描述报告S(Source Description)，提供站点的相关信息，包括CNAME等。

4) BYE：结束报告，用于报告站点结束会话。

5) APP(Application)：应用报告，用于报告特定的媒体类型和应用信息。

根据这五种不同数据包，RTCP主要提供以下4种功能：a,提供服务质量和拥塞控制信息；b,提供流同步信息；c,确定会话参与者规模；d,传送会话控制信息。RTP/RTCP协议具有简单、扩展性好、数据流和控制流分离等优点[27]，传输效率较高，非常适合在网络上传输实时数据[28]。

2.2 H.264与AAC 编码与其RTP封包规范概述

压缩编码，可以去除冗余信息，减少数据量，以用于存储、在网络中进行传输。流媒体数据的传输通常使用RTP协议，不同的媒体类型通常也有不同的RTP封包标准，如RFC3984文件中定义了H.264的封装规范，RFC3016文件中定义了MPEG-4音频的RTP封装规范。本论文工作的视频编码使用的是H.264编码，音频使用的是AAC编码，因此，将在接下来的小节中介绍一下H.264视频编码和AAC音频编码，以及这两种编码所对应的RTP封包规范。

2.2.1 H.264编码与AAC编码简单介绍

(1) H. 264编码简介

H.264视频编码标准是VCEG视频编码标准H.26x的一员，也是MPEG-4 AVC的第10部分，因此，有AVC/H.264、H.264/AVC等多种名称。为了在低比特率下具有良好的压缩效果并适应在不同的网络环境，H.264在结构上被设计分为VCL视频编码层和NAL网络抽象层等两层[29]，如图2-5所示。

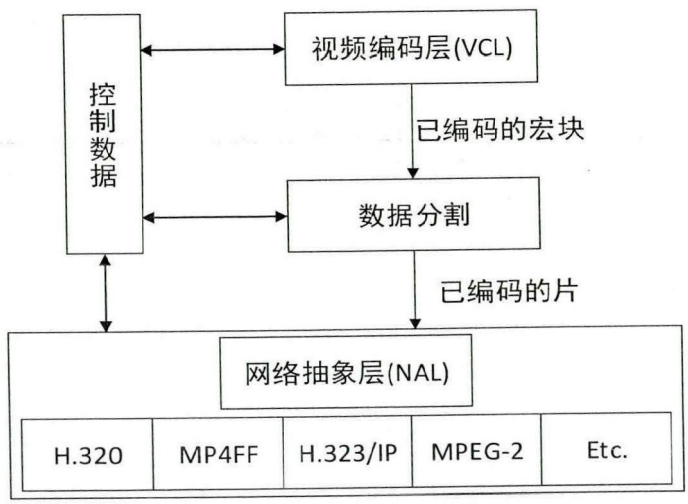


图2-5 H. 264编码器结构

Fig. 2-5 The structure of H.264 encoder

VCL被设计用来描述编码后的视频内容；NAL主要负责格式化视频的VCL描述，以及提供适用于不同传输层传输或存储媒体的头部信息，也就是说VCL数据在传输或存储之前，需要先封装进NAL单元中。NAL单元（NAL Unit）通常又被称为NALU[21]，

是组成H.264视频流的基本单元，要据打包的数据类型不同可以为分VCL单元和非VCL单元，VCL单元含有编码后的视频内容，而非VCL单元主要包含视频附加信息。NALU便是组成H.264视频流的基本单元，由一个字节的首部和若干字节的原始字节序列载荷（Raw Byte Sequence Payload, RBSP）组成[30]。图2-7给出了NALU首部和格式[31]：

0	1	2	3	4	5	6	7
F	NRI		TYPE				

图2-6 NALU首部格式

Fig. 2-6 The format of NALU header

其中各字段的含义如下：

1) F：Forbidden_zero_bit，1 bit。若该位为0，表示NAL单元类型八位字节和负载没有比特错误或语法违规；若为1，表示可能有比特错误或语法错误。

2) NRI：NAL_reference_idc，2 bit。该字段取值范围为00-11，00表示NAL单元内容不被用于为帧间预测重建参考图像，可被丢弃；大于00的值表示NAL单元内容需要用于保持参考图像的完整性。

3) TYPE: NAL_unit_type，5 bit。该字段值表示NAL单元的类型，取值范围为0-31表明NAL单元共有32种类型[31]，具体类型如表2-2所示：

表2-2 NALU类型

Table 2-2 The type of NALU

NALU类型	类型说明	NALU类型	类型说明
0	未指定	9	访问单元分隔符
1	非IDR图像的编码条带	10	序列结尾
2	编码条带数据分割块A	11	流结尾
3	编码条带数据分割块B	12	填充数据
4	编码条带数据分割块C	13	序列参数集扩展
5	IDR图像的编码条带	14-18	保留
6	辅助增强信息(SEI)	19	未分割的辅助编码图像的编码条带
7	序列参数集(SPS)	20-23	保留
8	图像参数集(PPS)	24-31	未指定

H.264编码以其码率低、图像质量高、容错能力强、网络适应性强等优点，被广泛用于蓝光光盘存储、网络流媒体、网络软件、各种高清电视广播、视频监控等诸多方面。

(2) AAC音频编码

AAC（Advanced Audio Coding），高级音频编码，是一种有损数字音频编码标准，在相同比特率下能获得比MP3更好的声音质量，被设计用于替代MP3。它由Fraunhofer IIS、杜比实验室、AT&T、Sony等公司共同开发，最初是MPEG-2音频编码标准，称为MPEG-2 AAC；后来被MPEG-4采用，称为MPEG-4 AAC。

AAC常见的封装格式有两种：ADTS(Audio Data Transport Stream)和

LATM(Low-overhead MPEG-4 Audio Transport Multiplex)。这两种封装在 MPEG-2 和 MPEG-4 中都有采用，都适用于网络传输，使用 RTP 协议传输 AAC 音频编码数据时多使用 LATM 格式。

ADTS 封装的 AAC 音频帧由两部分构成：ADTS 头和负载。负载部分是音频编码数据，通常是可变长的，而 ADTS 头通常是固定 7 个字节，又分为固定头和可变头部分，分别占 28bit。表 2-3 给出了 ADTS 头部参数的定义[32]：

表 2-3　ADTS 头部参数

Table 2-3 Parameters of ADTS header

参数/固定头	比特数	参数/可变头	比特数
Syncword	12	Copyright_identification_bit	1
ID		Copyright_identification_start	1
Layer	2	aac_frame_length	13
Protection absent	1	Adts buffer fullness	11
Profile_object_type	2	Number_of_raw_data_blocks_in_frame	2
Sampling_frequency_index	4	-	-
Private bit	1	-	-
Channel_configuration	3	-	-
Original_copy	1	-	-
home	1	-	-

其中，几个主要的参数与说如下：

- Syncword：同步字，表示一个 ADTS 帧的开始，通常为 0xFFF；
- ID：MPEG 版本，0 表示 MPEG-4，1 表示 MPEG-2；
- Layer：音频编码层，通常设为 00；
- Protection absent：误码校验；
- Profile：音频对象类型对应的 ID，表 2-4 为 MPEG-2 AAC 中音频对象类型及索引值[32]；
- Sampling_frequency_index：采样率索引，表 2-5 为 AAC 编码支持的采样率及索引值[33]；
- Channel_configuration：声道配置，单声道、双声道等；
- Aac_frame_length：AAC 帧的长度，包括 ADTS 头和负载；
- Number_of_raw_data_blocks_in_frame：帧中原始数据块的数量。

表 2-4 MPEG-2 AAC 中音频对象类型及索引

Table 2-4 The type and index of MPEG-2 AAC profile

Index	profile	Index	profile
0	Main profile	2	Scalable Sampling Rate profile(SSR)
1	Low Complexity profile(LC)	3	(reserved)

在表 2-4 中，Main 类型音质最好，但占用空间最多；LC 类型简单、编码效率高，无增益控制功能；SSR 类型和 LC 类型类似，多了增益控制功能；通常，LC 类型编码效率高，音质上与 Main 差别不大，使用的最多。

表 2-5 AAC 编码支持的采样率及索引

Table 2-5 the sampling rate and index supported by AAC

Index	Sampling Rate/Hz	Index	Sampling Rate/Hz	Index	Sampling Rate/Hz
0	96000	6	24000	12	7350
1	88200	7	22050	13	Reserved
2	64000	8	16000	14	Reserved
3	48000	9	12000	15	Reserved
4	44100	10	11025	-	-
5	32000	11	8000	-	-

LATM 封装的 AAC 音频帧也由两部分构成：AudioSpecificConfig(音频特定配置单元)和负载。AudioSpecificConfig 中包含音频的基本信息，负载由 PayloadLenghtInfo(负载长度信息)和 PlayloadMux（净负载）组成。AudioSpecificConfig 信息有两种传递方式：带内传输和带外传输。带内传输指的是每个 LATM 帧都有 AudioSpecificConfig 信息；带外传输指的每个 LATM 都不带 AudioSpecificConfig 信息，该信息通过其他方式如 SDP(Session Description Protocol)[34]，传递给解码端。

2.2.2 H.264 与 AAC 的 RTP 封包规范简介

数据在网络上传输时，通常都会受到 MTU(Maximum Transmission Unit)的影响，比如以太网的 MTU 为 1500 字节，那么数据在经过以太网传输时，大于 1500 字节的字节会被拆分。为了提高传输效率、降低丢包率、保证发送与接收的良好对接，H.264 视频数据与 AAC 音频数据需要按照一定的标准进行 RTP 封装，其中 RFC3984 文件定义了 H.264 的 NAL 单元的 RTP 封装，RFC3640 和 RFC3016 定义了 AAC 帧的 RTP 封装。

1. H. 264 的 RTP 封包规范

H.264 流的 NALU 在进行 RTP 包封装时，有三种不同的负载结构：单一 NAL 单元包、聚合包和分片单元包[31]。接收端可以通过 RTP 负载的第一个字节来判断其结构，该字节与图 2-7 的 NALU 首部格式类似，F、NRI 字段含义相同，TYPE 字段则代表负载的结构类型，表 2-6 为 NALU 在 RTP 封装的类型及含义[31]。

表 2-6 NALU 的 RTP 封装的类型及含义

Table 2-6 The type and description of RTP payload for NALU

类型	说明	类型	说明
0	未定义	26	MTAP16：多时间聚合包
1-23	NAL unit: 单一 NAL 单元包	27	MTAP24：多时间聚合包
24	STAP-A：单时间聚合包	28	FU-A：分片单元
25	STAP-B：单时间聚合包	29	FU-B：分片单元
26	MTAP16：多时间聚合包	30-31	未定义

下面介绍一下 NAL 单元的 RTP 封装结构：

✶ 单一 NALU 包：RTP 的负载即为一个完整的 NAL 单元，负载的首字节和原 NAL 单元的首字节相同。

✶ 聚合包：由于 NAL 单元较小，多个 NAL 单元聚合成一个 RTP 包，有 4 种情况，对应表 2-6 的 24、25、26、27。

✶ 分片单元包：将较大的 NALU 拆分到若干个 RTP 包中，属于同一 NAL 单元的分片具有相同的时间戳，有 2 种情况 FU-A、FU-B，分别对应表 2-6 中的 28 和 29。

本文主要使用单一 NALU 包和分片单元包中 FU-A 的封装结构，图 2-7 为 FU-A 的 RTP 负载格式[31]，FU-A 单元由 1B 的 FU Indicator、1B 的 FU Header 和 FU Payload 组成。

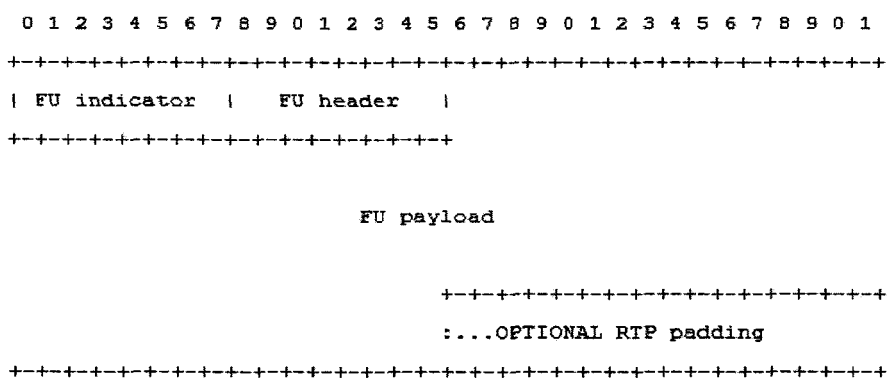


图 2-7 FU-A 的 RTP 负载格式

Fig. 2-7 The format of FU-A in RTP payload

FU Indicator 的格式如图 2-8，F、NRI 字段与原 NAL 单元首字节的相同，TYPE 字段为 RTP 封装结构，使用的是 FU-A 结构，故 TYPE 字段是 28。

0	1	2	3	4	5	6	7
F	NRI		TYPE				

图 2-8 FU Indicator 格式

Fig. 2-8 The format of FU Indicator

FU Header 的格式如图 2-9，如果采用 FU-A 封装方式，通常会有首分片、中间分片、

尾分片三部分（不一定有中间分片）。FU Header 的 S 字段是首分片标识符，当分片为首分片时，该字段为 1，否则为 0；E 字段为尾分片标识符，当分片为尾分片时，该字段为 1，否则为 0；R 字段是保留位，必须设为 0；TYPE 字段与原 NAL 单元首字节的 TYPE 字段相同。

0	1	2	3	4	5	6	7
S	E	R	TYPE				

图 2-9 FU Header 格式

Fig. 2-9 The format of FU Header

2. AAC 的 RTP 封包规范

RFC3640 和 RFC3016 都定义的 AAC 音频帧的 RTP 封装规范，这两种规范略有不同。RFC3016 里定义了 AAC 的 LATM 格式的 RTP 封包规范。RFC3640 定义的是 MPEG-4 裸流(Elementary Stream, ES)传输的 RTP 负载格式。本文使用的是 RFC3640 文件定义的格式，下面简单介绍一下。

按照 RFC3640 规范，RTP 负载中的 MPEG-4 流的类型信息是通过 MIME(Multipurpose Internet Mail Extensions)[35]格式参数传递的，如通过 SDP 信息或其他方式。这些 MIME 格式参数说明了负载了配置信息。考虑到简化和专用的接收端，一个 MIME 参数可用于说明使用该负载的特定模式。一个模式定义可能包括 MPEG-4 裸流的类型，以及应用配置，以避免接收端分析所有的 MIME 参数。

为了传输压缩的音视频数据，MPEG 定义了访问单元（Access Unit, AU）[35]。AU 是带有计时信息的最小数据单元。一个 AU 就代表一个音频帧或一个视频帧。一个 RTP 包中可以包含一个或多个完整的 AU，一个 AU 也可以被拆分成若干分片由多个 RTP 包负载，图 2-10 展示了 AU 的 RTP 封包结构[35]：

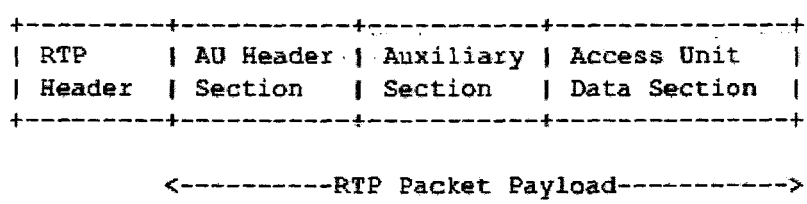


图 2-10　AU 的 RTP 封包结构

Fig. 2-10 The structure of AU in RTP packet

AU 单元包括三个部分：AU Header Section、Auxiliary Section、Access Unit Data Section。

- AU Header Section：AU 头字段，可以包含一个或多个 AU Header，当所有 Au Header 为空时，该字段为空；
- Auxiliary Section：辅助字段，包含辅助数据，该字段可以空；
- Access Unit Data Section：AU 数据区，该字段包含一个或多个完整的 AU，或某

一个 AU 的一个分片，不能为空。

AU Header Section 的结构包括 AU-Headers-Length、若干个 AU-Header、Padding Bit 三部分，如图 2-11 所示[35]。AU-headers 使用 MIME 格式参数配置，可能为空；如果为空，AU-headers-length 就不应展示，那么整个 AU Header Section 就为空；若不为，AU-headers-length 字段长度为 16bit，它表示其后的 AU-headers 以比特为单位的总长度，不包括最后的填充比特(padding bits)。每一个 AU-header 对应着同一个 RTP 包中 AU 数据区一个单独的 AU（分片）。对每一上 AU（分片），只有一个 AU-header。AU-header 以比特为单位，因此 AU Header Section 可能不是字节对齐的，这时就需要 padding bits 进行字节对齐。

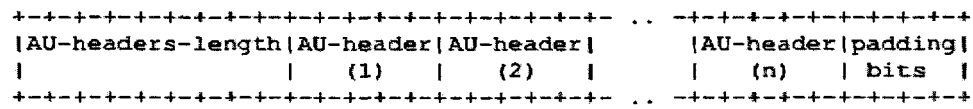


图 2-11　AU Header Section 结构

Fig. 2-11 The Structure of AU Header Section

AU-header 包含的域按顺序分别 AU-size、AU-Index/AU-Index-delta、CTS-flag、CTS-delta、DTS-flag、DTS-delta、RAP-flag、Stream-state。除 CTS-flag、

DTS-flag、 RAP-flag 外，其他域的比特长度由 MIME 格式参数设定。AU-size 指的是该 AU-header 在同一 RTP 包中的 AU 数据区的完整 AU 的长度（非 AU 分片的长度）；其他域的详细说明见 RFC3640 文件，对于一个 RTP 包中只有一个 AU 的情况，AU-header 只需要设置 AU-size，AU-Index/AU-Index-delta 设为 0 即可。

2.3 相关项目简介

本论文研究工作涉及到图像处理、音频采集、视音频的编码和解码，及 RTSP 直播服务器的实时，因此对所使用的一些开源项目进行简单介绍。

- OpenCV(Open Source Computer Vision Library)是一个开源的跨平台的计算机视觉库，于 1999 年由 Intel 公司建立，采用 C/C++语言编码，同时提供了 JAVA、C#、Python、Ruby、MATLAT 等其他语言接口，能够在 Linux/Windows/Mac 等操作系统上运行，广泛用于于图像处理、计算机视觉及模式识别等领域。
- OpenAL(Open Audio Library)是一个跨平台的音频库，主要用于音效处理。主要包括两个 API 分支：由实际 OpenAL 函数调用组成的核心和用于管理表现环境、资源使用和跨平台封装的 ALC(Audio Library Context) API。
- FFmpeg 是一个用以录制、转换及音视频流化的完整的、跨平台解决方案。它一个领先的多媒体框架，支持多种标准及非标准格式，可以解码、编码、转码、复用、解复用、流化、过滤以及播放等。Live555 简介
- Live555 是一个开源的、跨平台的流媒体解决方案，主要使用 C++编写，支持

Windows、Linux等平台，支持RTSP、RTP/RTCP、SDP、SIP等常用流媒体有关协议，能够对市面上多种视音频编码数据进行流化、封装、收发等处理。

2.4 本章小结

本章介绍了流媒体所涉及到的一些技术、标准和协议，如RTSP、RTP/RTCP协议、H.264视频编码标准、AAC音频编码标准，以及其码流结构和对应的RTP封包规范。在系统功能实现的时候，经过需要借助这些标准或规范检查实验结果，因此，这部分背景知识是开发实现的基础知识，对于了解本文也有了定的帮助。

第三章　基于RTSP协议的多源视音频实时直播系统的分析与设计

流媒体技术应用广泛，网络电话、视频对话、视频点播、网络直播、视频监控等，无一不是流媒体技术的应用实例。本文研究与实现的内容——基于RTSP协议的多源视音频实时直播系统，是网络直播与视频监控的结合与扩展，本章将分析介绍该多源视音频实时直播系统的需求和系统功能，提出系统的总体设计方案。

3.1 系统的设计目标

流媒体系统最终的目标是实现多媒体数据的存储与共享。本文系统设计的总体目标也围绕着这个大方向，主要是利用网络、多媒体等技术和软、硬件等产品对多场景视频和音频信息进行采集，利用软件将这多路视频流和音频流分别整合到一起，输出一路标准编码、封装的视频流和音频流，实现自由选择的多个场景的同步记录、存储、发布和交换，实现视音频的网络共享，达到对包含多个子场景的复合场景的真实呈现或再现。

3.2 系统需求分析

传统的视频监控与视频会议方案解决了跨地域观摩某一现场场景的问题，有一定的普遍适用性，也有其应用局限性，无法满足某些特殊场合或情景的应用需求，如某一场合有多个重要角色且位于离散的位置，而某远程用户需要同时观看每一个角色的状态。

本文系统来源于某智能科技法庭项目，下面将分析说明本文系统设计与实现的时各种需求。

3.2.1　业务需求

一个系统要满足一定的业务需求，本文系统主要的业务要求概括地讲就是声色俱备、多流合一、实时直播与存储、标准化，具体说明如下：

1. 声色俱备

实时场景直播不同于常规网络视频监控，因此，不仅需要记录实时的视频画面，也同样需要捕获实时的现场声音，以便通过实时的画面和声音再现现场情景，或实现对实时现场情况的观摩。

2. 多流合一

一个大的场景中通常有多个子场景，因此，一个摄像机很难即关注全局场景，又照顾到每个子场景的细节，虽然能通过多个摄像机+DVR（Digital Video Recoder，数字视

频录像机）或NVR(Network Video Recoder)，网络视频录像机)的方式将多路摄像机画面同时输出到一个屏幕上，但却是各路分开独立存储，很难同步呈现，无法通过一路视频流实现全局多子场景的同步再现。视频多流合一就是要将多路视频画面按特定的方法拼合成一幅图像，这样多路视频流就被整合成一路，实现“多流合一，以一盖全”，即以一路拼合视频呈现多个场景。音频也需要做类似的处理，混合多路音频，实现多流合一。

3. 实时直播与存储

获取视频和音频的主要也是用于存储和共享。存储可以用于后续查阅，再现当时的实况，便于取证等；直播是为了视音频等媒体的共享，如果将多源合一得到的实时视频和音频数据发送到网络中，实时视音频实时直播，就可以跨越地理的限制通过接收软件实现对整个现场实时情景的观摩。

4. 标准化

视频和音频数据的压缩编码采用较常用的编码标准，数据发送采用较常用的封装格式，网络协议采用主流的流媒体协议，方便与其他流媒体系统或软件交互与对接，不需要使用专业的硬件设备或软件，使用常规网络播放器等软件即可接收、播放。

3.2.2 功能需求

本文系统主要是为了让多客户能够同时观摩实时多场景视频、听到多场景音频。系统要求能够捕获到实时的视频画面（不限于现场画面，也可以是录制的媒体文件、电脑桌面的截屏等），捕获到实时的声音（不局限于现场声音，也可以是录制的媒体文件）通过捕获的图像和声音共同呈现原场景。系统要求能够将多个在时间上对齐的画面拼合到一张图像上，将多个在时间上对齐的声音混合到一起。系统要求能够分别对这个多合一形成的视频画面和音频进行压缩编码，以保证其能适应网络传输。

此外，为减轻主机的工作压力、节省网络带宽，系统要求多媒体数据采用组播发送方式。当客户端向系统发出观摩请求时，通过与系统完成RTSP会话交互，加入到组播组，并获取视音频媒体有关信息，随后接收多媒体数据进行解码播放或显示。

3.2.3 环境要求

有关技术并没有硬性要求，但制定技术标准还是很有必要的，可以避免后续开发实现的不兼容性，当然不同的场合有不同的技术要求，本文主要的技术要求如下所述：

1. 音视频数据编码采用国际流行的编码算法（H.264/AAC）。音频编码遵循：MPEG-4 AAC（ISO/IEC 14496-3），精度≥16bit，48kHz；视频编码遵循：H.264/MPEG-4 AVC（ISO/IEC 14496-10）。

2. 音视频网络传输遵循: RTP/RTSP 和ISMA标准。ISMA2.0包括：网络传输协议：RTP/RTSP/SDP；H.264 封包协议：RFC3984；AAC 封包协议精度≥16bit，48kHz：

RFC3640/RFC3016。

3. 音视频存储遵循：.MP-4 文件格式（ISO/IEC 14496-14）。
4. 视音频等多媒体数据使用组播方式发送。
5. 网络环境要求较好，最好是专网。

3.3 系统结构设计

3.3.1 总体结构分析与设计

在分析系统的结构设计前，先来看一来系统设备网络结构，如图 3-1 所示。客户端使用网络播放器（如 VLC Player、MPlayer、PotPlayer 等），即可接收播放服务端发送过来的多媒体数据，该部分不是本文系统的组成部分，因此，其设计与实现本文不做说明。系统的硬件设备部分主要由网络摄像机、麦克风、混音器、服务主机及一些网络设备等部分组成，其发挥的主要作用如表 3-1 所示。

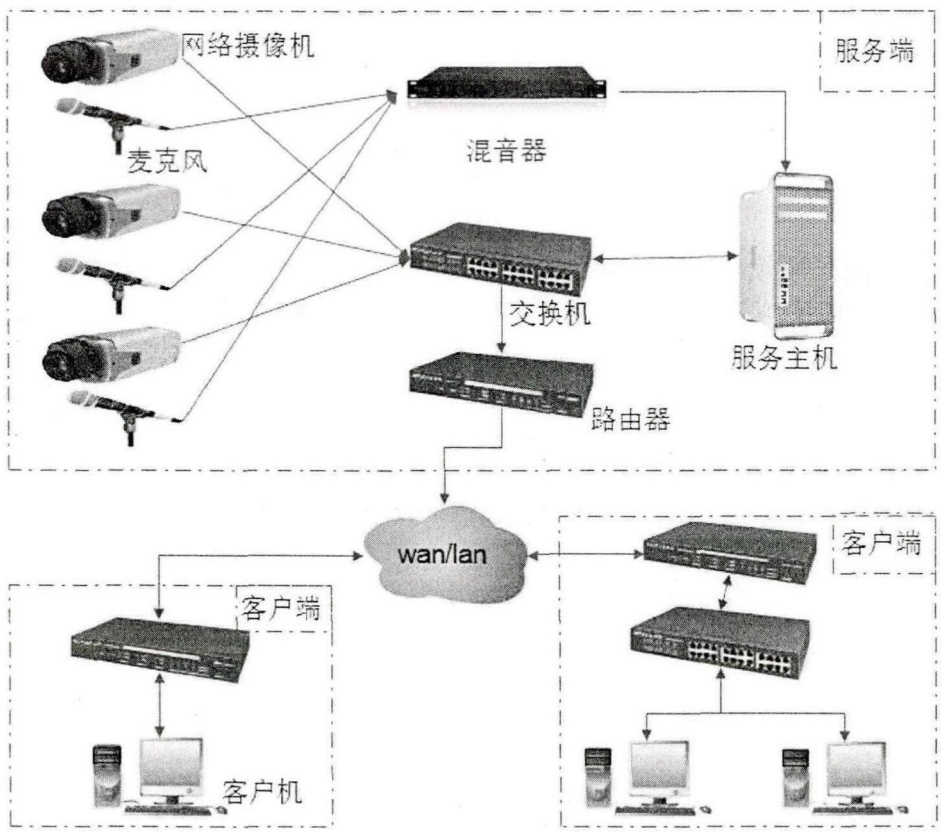


图 3-1 系统设备网络结构

Fig. 3-1 The network structure of system devices

表 3-1 设备说明

Table 3-1 The introduction of devices

设备	说明
网络摄像机	捕获视频画面，输出特定编码的视频流。
麦克风	采集声音，输出电信号。
混音器	输入麦克风采集的声音信号，混合后输出。
服务主机	运行系统软件，提供有关服务。

以一路视频为例，其内部的数据流向如图 3-2 所示，音频的数据流向与其大同小异。

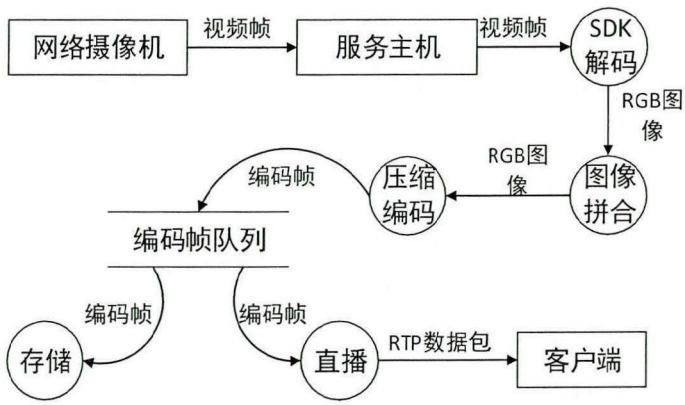


图 3-2 简化的 DFD

Fig. 3-2 DFD in brief

根据上述分析，本文系统主要由前端数据处理模块和直播服务器模块组成，如图 3-2 所示。前端数据处理模块负责数据的输入与输出，具体来说是通过各种采集设备捕获实时视频和音频数据，经过一些必要处理后，输出视频和音频的压缩编码数据，输出的数据可用于存储，也用于网络发送；直播服务器模块主要负责接收前端数据处理模块输出的视频和音频编码数据，等待并响应客户端的请求，经过分析、RTP 封装后，发送出去。

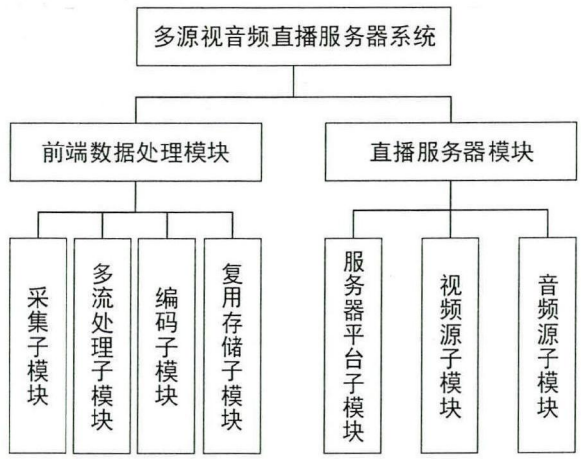


图 3-3 系统功能模块

Fig. 3-3 The function modules of system

3.3.2 视音频前端数据处理模块

根据图3-3，该模块又可以细化以下几个子模块，主要职能如下：

1. 采集子模块

采集子模块的主要职责是采集视频和音频数据，需要注意的视频和音频的采集方法是不同的，因此，其具体实现也不相同。视频采集需要借助摄像机：如果是网络摄像机的话，可以直接通过网络接口获取视频流；若是模拟摄像机，需要借助 DVR 完成A/D(Analog/Digital)转换，再通过DVR的网络接口获取视频流。音频需要使用麦克风获取声音的电信号，经过声卡完成A/D转换，进而获取声音的数据。

由于视频和音频采集方法不同，该模块实际是视频和音频采集模块的统称，具体来说是视频采集模块和音频采集模块。

2. 多流处理子模块

此模块负责完成视频和音频的多流合一处理，利用计算机图像拼接技术，将实时的多路视频画面拼接成一路画面，这路拼接生成的画面就包含了多路视频画面，可以达到用一路视频视频观看多个场景画面的效果。音频的多源合一处理类似，就是要将多路实时音频混合成一路。

视频和音频多流合一处理方法不同，该模块实际是视频和音频多流处理的统称，具体来说是视频拼合模块和音频混合模块。

3. 编码子模块

视频和音频有着不同的编码标准，因此，编码子模块应该算是视频编码模块与音频编码模块的统称。未经压缩编码的数据包含有大量的冗余信息，不适应直接用于网络传输，经过压缩编码，去除其中的冗余数据，可以极大地减少数据量，利于存储和网络传输。

视频和音频编码标准不同，该模块实际是视频和音频编码的统称，具体来说是视频编码模块块和音频编码模块。

4. 复用存储子模块

复用存储模块主要负责将一路复合而成的视频数据和一路混合的音频数据保存到一个本地媒体文件中，以供日后点播，或用于后续的调研取证等，主要的容器包括MP4、MKV、FLV等。

3.3.3 直播服务器模块

该模块需要实现 RSTP、RTP/RTCP 等有关协议，等待客户端的请求，并做相应的处理，将视频和音频等媒体数据打包后发送给客户端。利用开源的流媒体服务器项目live555，本文将直播服务器模块又细化为平台子模块、视频数据源子模块、音频数据源子模块。

1. 平台子模块

该模块主要负责设置和实现直播服务器的运动环境和程序框架，保证各流媒体协议处理部分能够正常运作。

2. 视频数据源子模块

该模块是直播服务器的视频编码数据输入接口，负责接收压缩编码生成的视频编码帧，交给 live555，经分析、打包处理后，发送到网络。

3. 音频数据源子模块

该模块类似视频数据源子模块，它是直播服务器的音频编码数据输入接口，负责接收实时生成的音频编码帧，需要注意的是编码格式不同的话，因其 RTP 封装方式不同，就需要实现对应的数据源。也就是说，一种编码格式对应一种数据源。

3.4 本章小结

本章分析阐述了系统的有关需求，介绍了基于 RTSP 协议的多源视音频实时发布系统的总体设计，并根据系统需求将系统功能模块进行了细化分解，具体分析了各模块的主要职能及解决的问题，各功能子模块的实现也是本文的工作重点，这部分内容将在后面的章节中具体说明。

第四章　视音频前端数据处理模块的设计与实现

本文第三章分析了系统的需要，并给出了总体设计，接下来将阐述各部分的具体实现，本章主要介绍前端数据处理模块的具体实现，包括采集、多流合一处理、编码与存储等。

4.1 视音频的采集模块的实现

采集模块是系统的眼睛和耳朵，主要负责信息的输入。该模块借助于一些软、硬件产品，获取视频和音频的具体数据，再将这些数据传递给后续模块，由后续模块对获取到的视频和音频进行下一步加工。

4.1.1　基于网络摄像机的视频采集

广义上来讲，能获取或采集到视频的方法都能称视频视频采集，因此，视频的采集就包括了通过摄像机采集、从录制文件中获取、从网络中获取、电脑桌面截屏等多种方式。本文视频的采集主要是通过摄像机采集，也可以从录制文件中获取视频流并解码，达到对本地视频文件采集的目的。

常用的摄像机包括模拟的和网络的等类型。其中，模拟摄像机采集到的模拟信号，不能被计算机识别，需要使用视频采集卡来完成模拟到数字的转换，再交给计算机处理；网络摄像机内置一个嵌入式 DSP 芯片，采用嵌入式实时操作系统，通过 CMOS 感光器完成捕获到的光信号到数字信号的转换，接着由 DSP 芯片对视频数据进行压缩编码，最后再通过网络发出经过编码的视频流数据。凭借着性能优良、易于安装维护和管理、视频画面清晰流畅、产品丰富、造价较低等优势，网络摄像机被广泛地应用于教育、商业、医疗、公共事业等众多领域。

本文开发实现时使用网络摄像机，市场上网络摄像机品牌与各类不胜枚举，通常生产商都会制定自己私有的网络通讯协议和码流封装格式，因而需要配合其提供的 SDK 才能实时获取和解码视频流，以用于显示查看或其他分析和处理。下面说明本文使用的海康威视 DS-2CD2210(D)-I5 高清网络摄像机实现视频采集的具体过程与方法。

海康威视提供的设备网络 SDK 主要包含网络通讯库、软解码库等功能组件。网络通讯库是设备网络 SDK 的主体，它提供了通过网络访问摄像机、与摄像机进行交互及获取视频流的接口；软解码库主要负责对从摄像机获取的实时的视频流进行解码处理，其获取视频实时流的 SDK 接口调用的主要流程如图 4-1 所示。

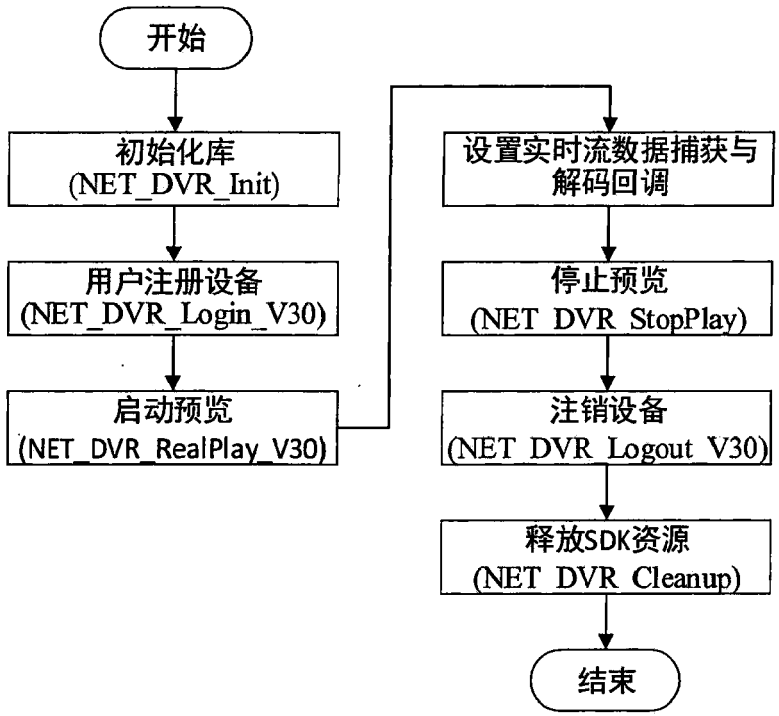


图 4-1 海康 SDK 获取视频流流程图

Fig. 4-1 The flow chart of getting video stream with Haikang SDK

其中，主要部分说明如下：

1) 初始化开发库：调用 NET_DVR_Init 函数对开发库进行初始化处理；

2) 注册设备：调用函数 NET_DVR_Login_V30 对用户进行注册操作；

3) 启动预览：调用函数 NET_DVR_RealPlay_V30 设计解码回调函数 从前端服务器取实时码流，解码显示以及播放控制等功能；

4) 设置实时流数据捕获与解码回调：设置实时流的捕获和解码回调函数，之后可以通过解码回调函数获取解码后的视频数据；

5) 其它，停止预览（NET_DVR_StopPlay）、注销设备(NET_DVR_Logout_V30)、释放 SDK 资源（NET_DVR_Cleanup）主要用于一些结束、回收、清理。

由于后面的工作需要多线程并发从多个网络摄像机获取视频图像，为提高代码利用率、方便调用，本文定义并实现了用于获取海康威视频网络摄像机的视频捕获类 CHKDvrVideoCapture，该类是基于海康威视 SDK 的，其部分主要成员与说明如表 4-1 所示。

当需要从网络摄像机中获取图像时，声明一个 CHKDvrVideoCapture 对象，完成各项初始化及注册工作，执行 StartCapture()函数，设置好解码回调函数，只要在对象的生存期内便可以通过函数回调不断地从摄像机获取视频流、进行解码，外部只需要调用 GetFrame()函数，便可以将解码后的 YUV420 格式数据转换成 RGB 格式数据，并拷贝到一个 IplImage 结构体中，接下来就可以利用第二章介绍的计算机视频库 OpenCV 对视频图像进行进一步的分析处理。

表 4-1 视频捕获类 CHKDvrVideoCapture 部分成员及描述

Table 4-1 The members of CHKDvrVideoCapture

	成员	描述
变量	m_buff_YV12	Char*，用于保存解码回调函数输出的图像的 YUV420 格式的数据。
	m_iBuffSize	Int，m_buff_YV12 中有效数据的长度。
	m_iHeight	int，图像的高。
	m_iWidth	int，图像的宽。
函数	HKRealDataCallBack()	友元函数，实时流捕获并解码回调函数，进行视频图像解码，通过 PlayM4_SetDecCallBackMend()设置解码回调函数。
	HKDecCBFun()	友元函数，解码回调函数，将解码后的视频图像 YUV420 数据拷贝到 m_buff_YV12，并设置 m_iBuffSize、m_iHeight、m_iWidth。
	Login()	调用 NET_DVR_Login_V30()注册设备。
	StartCapture()	调用 NET_DVR_RealPlay_V30()启动实时流捕获，并调用 NET_DVR_SetRealDataCallBack()设置实时流捕获回调函数。
	GetFrame()	将 m_buff_YV12 中的 YUV420 格式的图像数据转换成 RGB 数据，拷贝到 Opencv 图像结构体 IplImage 中，用于后续处理。
	StopCapture()	结束实时流捕获。
	Logout()	调用 NET_DVR_Logout_V30()注销设备。
	CHKDvrVideoCapture()	构建函数，调用 NET_DVR_Init()初始化 SDK，和一些其他参数。
	~CHKDvrVideoCapture()	资源回收和清理。

4.1.2 基于 OpenAL 方案的音频采集

目前，市场上缺少类似网络摄像机这样的网络音频采集设备，虽然某些品牌的某些型号的网络摄像机带有音频采集模块，但该模块并非其核心模块，仅用于辅助监控，其所支持的音频的采样率和量化精度较低，无法满足对采样率和量化精度有较高要求的应用场景。本文音频的采集使用电容式话筒捕获、声卡采样量化、软件编码的工作方式，其拓扑图如图 4-2 所示，如果有多支话筒，可以使用混音器，将多支话筒采集到的音频信号混合在一起，再输入到主机的声卡中。

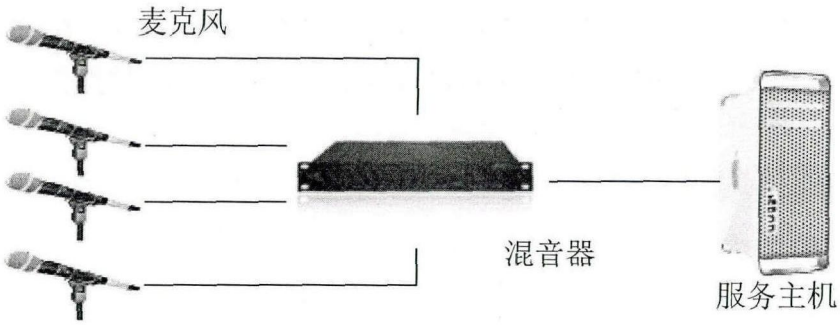


图 4-2 音频采集设备连接

Fig. 4-2 The connection of devices for capture audio

话筒采集到的音频信号传递给主机的声卡后，声卡不会主动对这些音频信号进行数字化处理，这些工作需要编写程序驱动声卡来完成。本文 2.3 章节简单介绍了 OpenAL 解决方案，该方案具有开源和跨平台等特性，本文基于该方案定义并实现了音频采集类 AudioCapture，用于音频数字化处理，其处理流程如图 4-3 所示。

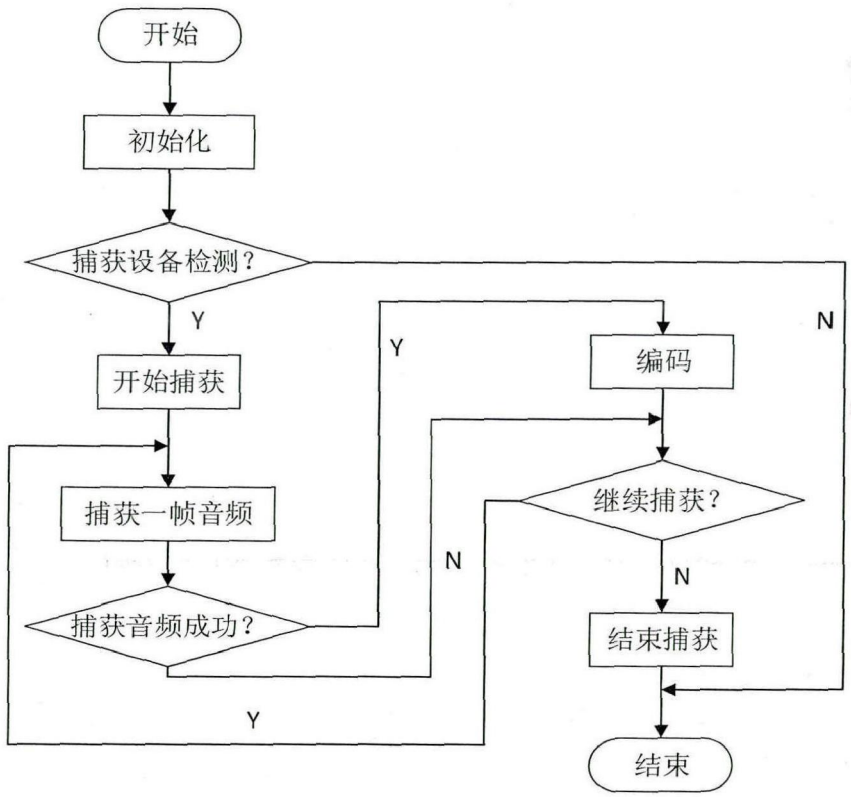


图 4-3 音频采集流程图

Fig. 4-3 The flow chart for capture audio

AudioCapture 类有几个主要成员变量：sample_rate（采样率）、channels（声道数）、format（OpenAL 格式，包括声道和量化精度信息）、defaultCaptureDevice（默认捕获设备）、defaultCaptureDeviceName（默认捕获设备名称）、samplesPerFrame（单帧采样数）。其中，format 在 OpenAL 中有四种定义：AL_FORMAT_MONO8、AL_FORMAT_MONO16、AL_FORMAT_STEREO8、AL_FORMAT_STEREO16，分别指

的是单声道8位、单声道16位、双声道8位、双声道16位，也就是说OpenAL的量化精度最高支持16位；samplesPerFrame指的是某些编码格式每一帧音频中的采样数，对于AAC编码每帧音频中有1024个采样[36]，而MP3每帧中有1152个采样。下面针对图4-3，描述下AudioCapture主要函数接口的实现：

(1) 初始化，主要由构建函数AudioCapture()来完成，需要输入音频采样率、声道数、格式及单帧采样数，同时该函数会根据这些参数初始化声音捕获设备，并读取其名称保存下来。

(2) 默认设备检测，检测是否获取到默认捕获设备，成员函数getNameOfDefaultCaptureDevice()返回默认捕获设备的名称，可以据此判断是否获取到默认捕获设备。

(3) 开始捕获，通过startCapture()成员函数启动捕获设备。

(4) 捕获一帧音频，captureAudioSamples(uchar* buffer)成员函数内部调用OpenAL的接口 alcGetIntegerv()来进行采样，但每次采集到的采样数并不一定都能达到要求(samplesPerFrame)，当采集到的样本数不小于 samplesPerFrame 时，就将采样的量化数据保存到buffer中，其长度计算根据下面的公式4-1求得。

$$Length = samplesPerFrame * channels * bits / 8 \quad (4\text{-}1)$$

其中，Length 指的是一帧音频采样量化数据的长度，单位为字节；samplesPerFrame 是一帧音频中的采样数；channels 是音频声道数；bits 是音频的量化精度。采样成功后会返回采样的个数，可以据此判断是否采样成功。

(5) 结束捕获，stopCapture()成员函数总是与startCapture()成员成对出现，捕获工作通常在这两个函数中间完成。

这样，就实现的声音的数字化，音频数字化的主要方法是PCM(Pulse Code Modulation，脉冲编码调制)编码，该方法主要包括采样、量化、编码三步[37]。得到的数据也是经过PCM编码的数据，接下来可以将这些PCM数据编码成AAC或MP3等音频格式。

还需要注意的是采样数据的排列方式。单声道每个采样只有一个数据，其排列方式是连续排列，即采样1、采样2、采样3……。双声道每个采样有左声道、右声道两部分数据，通常排列方式是连续交叉排列，即采样1左声道、采样1右声道、采样2左声道、采样2右声道、采样3左声道、采样3右声道……；另一种排列方式是Plannar排列，即采样1左声道、采样2左声道、采样3左声道……采样N左声道、采样1右声道、采样2右声道、采样3右声道……采样N右声道。一般采样到的双声道数据是连续交叉排列的。

4.2 采集速度控制的分析与探讨

4.2.1 控制采集速度的必要性

帧率和采集率分别是视频和音频的重要参数，它们都包括一定的时间信息，是播放器控制播放速度的重要依据。帧率是单位时间视频帧的数量，它直接反映了视频播放的速度；采样率是单位时间内音频采样的数量，与它相关的是单帧采样数，指的是某种编码格式一帧音频数据包含的采样的数量，不同编码格式，一帧中的采样数是不一样的，如前面章节提过 AAC 编码每帧中有 1024 个采样，而 MP3 编码每帧中有 1152 个采样。

假设有一个 MP4 格式的多媒体文件，视频帧率为 fps，音频为某编码格式，采样为 samplingRate，该编码每帧有 samplesPerFrame 个采样，当播放器打开这个文件时就能查询到这些信息，就会以每秒 fps 帧的速度播放视频，以每秒 sampleRate/samplesPerFrame 帧的速度播放音频。如果播放时不按这个预设的速度播放，会怎么样呢？显然，一个文件中音频帧或视频帧的数量是确定的，当播放速度人为提高时，每秒播放的帧数就会增加，视频画面变快，最终的播放时间短于标准时间；反之，降低速度时，每秒播放的帧数减少，视频画面变慢，播放时间会长于标准时间。

采集得到的视音频数据都是用于编码，如果采集的速度过快，每秒采集到的视频或音频帧数就会多于编码设定的单位时间内的帧数，存储下来后，播放器会按编码预定的帧率和采样率播放，由于总帧数变多了，那么总的时间就会长于标准时间；反之，若采集速度过慢，每秒采集到的视频或音频帧数少于编码设定的单位时间内的帧数，存储下来后，由于总帧数变少了，总的时间短于标准时间，且播放时会出跳变现象。

因此，应该根据编码设定的帧率、采样率，去控制采集的速度。

4.2.2 控制采集速度的方法

视频和音频的采集通常是在各自独立的线程中进行，需要根据视频和音频的编码参数进行控制，不宜过快也不宜过慢。具体的控制策略如下所述，处理流程如图 4-4 所示：

1. 根据编码设定的帧率和采样率确定理论速度

假设编码设定视频帧率为 fps，音频采样率为 samplingRate，设定的音频编码格式每帧采样数为 samplePerFrame，采集一帧视频图像所需时间的理论值 timePerVideoFrame，可根据公式 4-2 求出，单位为 ms。

$$timePerVideoFrame = 1000 / fps \tag{4-2}$$

采集一帧音频所需时间的理论值 timePerAudioFrame，可依据公式 4-3 求解，单位为 ms。

$$timePerVideoFrame = 1000 * samplesPerFrame / sample\_rate \tag{4-3}$$

2. 计算单次实际采集时间

每次的实际采集速度需要计算，记录采集开始时的时刻 t1，当采集工作完成时，记录当前时刻 t2，那么本次采集所用的时间 temp，要注意时间单位，最好换算成 ms。

3. 比较采集用时的理论值与实际值

以视频为例，根据其帧率计算出理论值为 timePerVideoFrame，某次采集的实际用时为 temp，那么它们的差 wait_time 计算方法如下所示。

$$\text{wait\_time} = \begin{cases} 0 & , \text{temp} \geq \text{interval} \\ |\text{temp} - \text{interval}| & , \text{temp} < \text{interval} \end{cases} \tag{4-4}$$

根据 wait_time 的值，做如下处理：

a) wait_time == 0：说明此次采集所用的时间比理论值要长，不需要做等待，可以直接进行下一次采集；

b) wait_time > 0：说明此次采集所用的时间比理论值要短，需要稍做等待，等待 wait_time 时长后，再开始下一次采集。

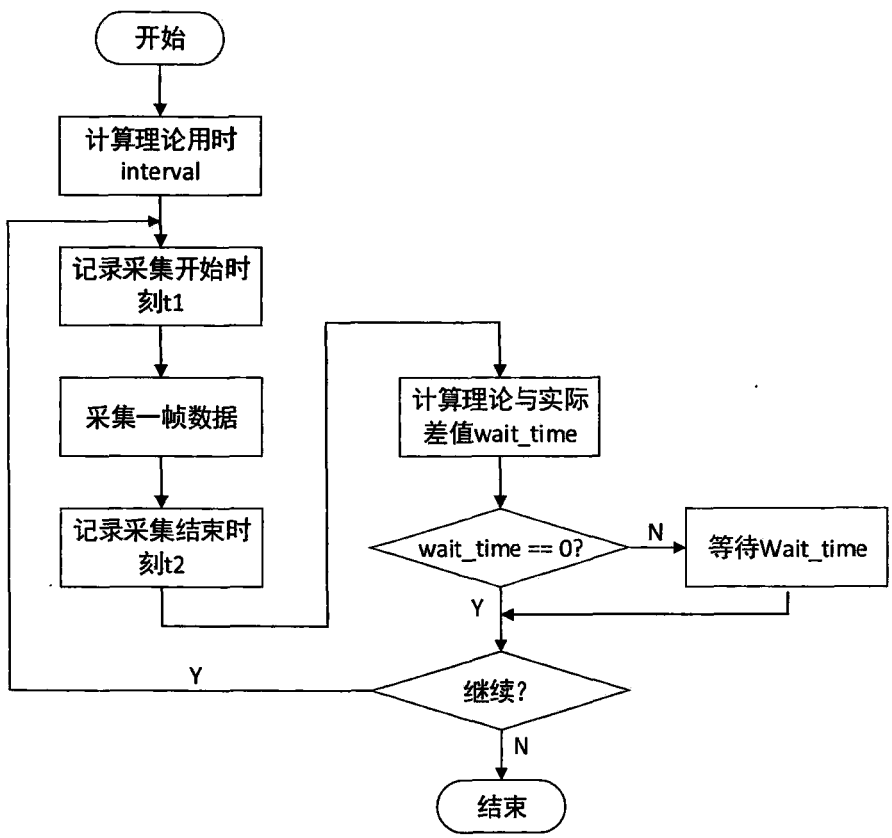


图 4-4 采集速度控制处理流程

Fig. 4-4 The flow chart for controlling speed of capturing

4.3 多流合一处理模块

网络摄像机提供通过网络访问实时视频的途径，使我们可以跨平台、跨地域地访问实时视频画面。NVR 实现了多路视频的集中显示，但这多路视频彼此独立，没有解决

多摄像机同时访问，以便全局地观摩某场景的问题。解决这一问题的一个途径是运用图像处理技术，将从多路网络摄像机中获取的视频画面经过缩放后，拼合在一张图像中，通过实时地获取、实时地拼合，每张拼合成的图像都包含这几个网络摄像机的实时图像；同时，多路音频也可以混合成一路。

4.3.1 多视频图像拼合

视频图像拼合的主要思路是创建一张较大的空白图像，再将这张空白图像按需要划分成若干子区域，然后利用图像处理有关技术，将从网络摄像机获取的视频图像进行相应比例缩放，然后将缩放后的图像拷贝到指定区域，其处理流程如图 4-5 所示，当然这只是从一路网络摄像机取视频图像并拼合的处理，如果有多路网络摄像机，需要多线程并发进行。

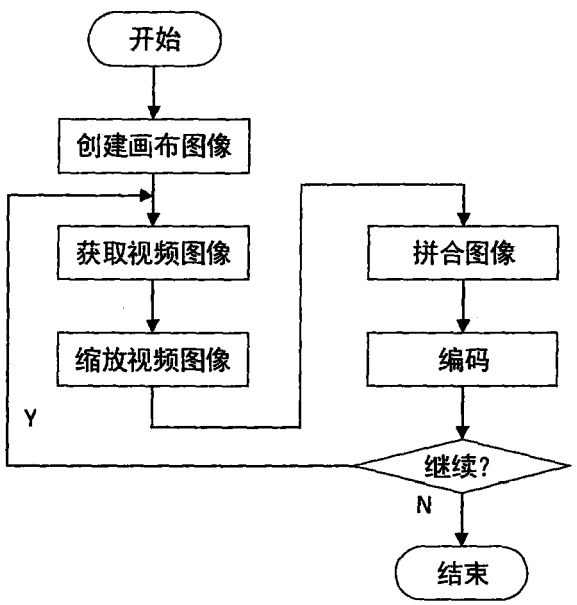


图 4-5 图像拼合流程

Fig. 4-5 The flow chart of image stitcing

1) 创建画布图像

这张较大的空白图像，可以形象地称为画布图像，它就像现实中的一片白色的画布一样，可以在上面随便作画。画布图像包含若干分割点，这些分割点中某两个点相连就构成一条分割线，这一条条的分割线就将整个画面图像分割成若干个矩形子区域，每个矩形子区域都可以填充一帧对应尺寸的视频图像，因此，一路视频图像经过相关处理后，就可以填充到画布图像的指定子区域中。

2) 确定画布图像的分割方式

创建画布图像首先要确定画布图像的大小，即宽 Width 和高 Height，再确定其子区域的分割方式，分割方式不宜过于奇特，主要有 4 分屏、5+1 分屏、7+1 分屏、9 分屏等，如图 4-5 所示，接下来根据其宽、高和分割方式计算出各分割点的坐标，如图 4-6

中的圆点。

通常情况下，我们可以根据一对对角顶点确定一个矩形，那么确定好分割方式后就可以计算出画布图像的分割点坐标，接着可以根据这些分割点将画布图像划分为若干矩形子区域，如图 4-5(b)所示的 5+1 分屏，我们设定画布图像的左上顶点为坐标原点，可以根据画布图像的尺寸 Height 和 Width，计算出其分割点 p1~p11 的坐标，如 p1(0,0)、p3(Width/3*2,Height/3)、p7(Width/3*2,Height/3*2)等，那么由 p1、p7 可以确定矩形 A，由 p2、p4 确定矩形 B，以此类推，最终可以确定一个包括 A、B、C、D、E、F 的矩形列表 rectangleList，该列表就保存着画布图像的子区域信息。为了让这些矩形子区域有明显的分割界线，一般会使用绿色等较亮的颜色画出矩形的内部边界线，如图 4-6 所示的绿色线条。

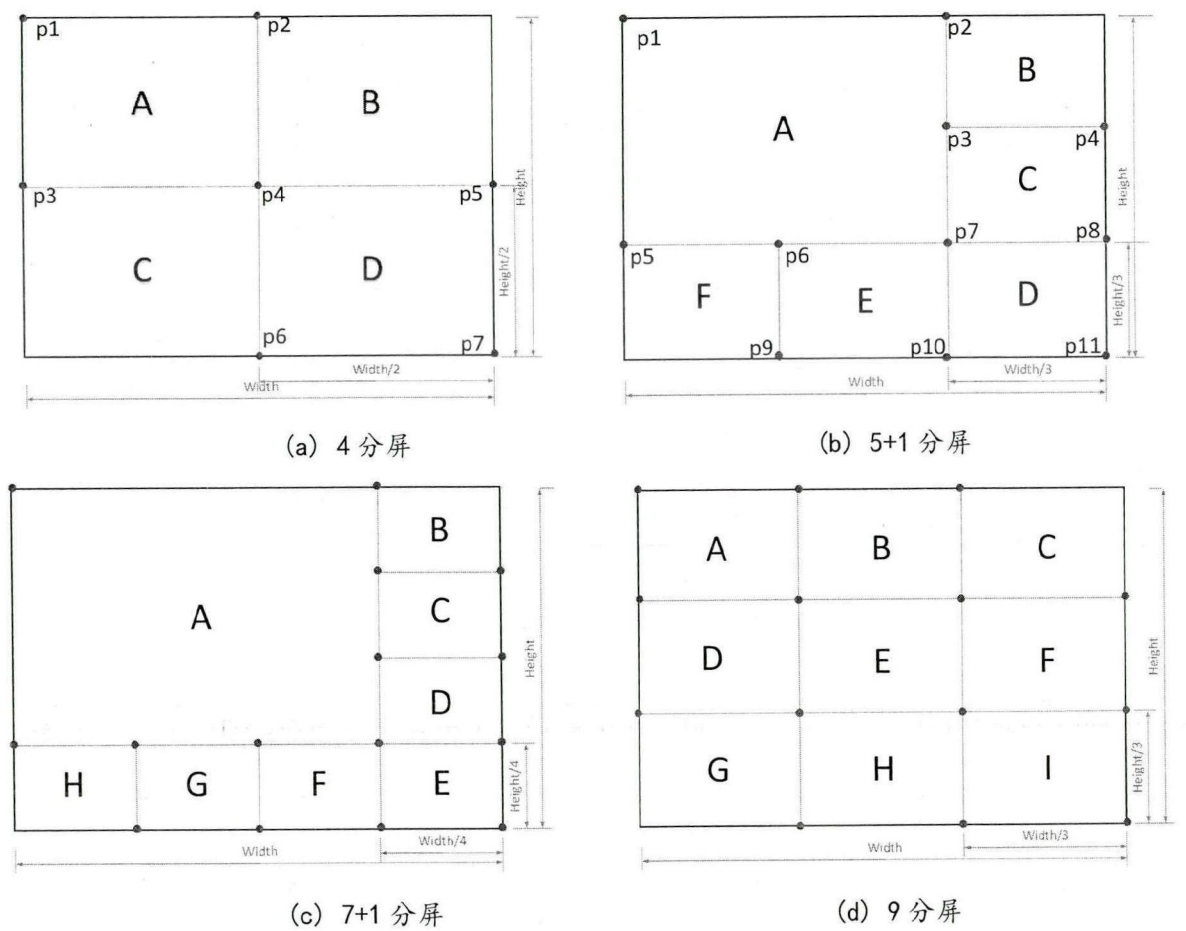


图 4-6 图像拼合方式

Fig. 4-6 The type of image stitching

3) 获取待拼合图像

完成画布图像的创建和分割后，接下来的工作是获取解码后的 RGB 图像数据了，这一步工作在前面 4.1.1 视频的采集已经完成，通过多线程并发访问，我们可以同时从多个网络摄像机获取解码后的 RGB 视频图像数据，转换成 OpenCV 图像结构体 IplImage 的实例后，就可以进行有关处理了。

4) 视频图像缩放与拼合

视频图像缩放就是对视频图像进行比例放大或缩小。OpenCV 提供了丰富的图像处理文宗，其中就包括图像的缩放处理，图像缩放主要用到 OpenCV 中的尺寸调整函数 cvResize()，该函数可以用来将源图像精确地转换为目标图像尺寸，目前 OpenCV 为该提供了五种插值方法，用于图像尺寸调整，如表 4-2 插值方法[38]所示。

表 4-2 cvResize()插值方法

Table 4-2 The interpolation method of cvResize()

插值方法	含义
CV_INTER_NN	最近邻插值
CV_INTER_LINEAR	线性插值
CV_INTER_AREA	区域插值
CV_INTER_CUBIC	三次样条插值
CV_INTER_lANCZOS4	8x8 像素领域 Lanczos 插值

在对某帧视频图像 srcImage 进行缩放之前，需要先确定它的在画布图像上的目的矩形子区域 T。接下来通常的做法是创建一个和目的矩形子区域相同尺寸的临时 IplImage 实例 tempImage，然后对 srcImage 缩放处理，得到的数据保存在 tempImage 中。然后将 tempImage 中的图像数据拷贝到画布图像的矩形子区域 T 中；另一种简单的做法是设置感兴趣区域[39]，将画布图像的 T 区域设置成感兴趣区域，这样对画布图像进行处理就等于是对其矩形子区域 T 进行处理，因此可以直接对 srcImage 缩放，将缩放后的数据保存到画布图像中，此时也等于是对 T 区域直接操作，最后再释放感兴趣区域即可。

到这里，就完成了将多路视频图像拼合成一路视频图像的图像，如果需要接收远程端发送过来的视频流，只需要先将其解码转换成 RGB 格式数据，其他处理类似，处理过程中需要注意线程间互斥。

4.3.2 多音频混合

众所周知，声音是由物体振动产生的，不同频率的声音是可以进行线性叠加的，而且叠加以后互不干扰。本文的音频混合也是不同来源音频的叠加，主要分为两种情况：模拟音频的叠加和数字音频的叠加。

1) 模拟音频的叠加。

模拟音频的叠加是音频模拟信号的叠加。电容式话筒捕获到声音的振动信号后，会将它转换成电信号，当有多支话筒采集并输出多路音频电信号时，只需要将这几路电信号合成一路，就实现了多路音频的混合。混音器可以用来完成多路音频电信号的合成工作，对于这种场景中有多支话筒的情况，就可以使用混音器这样的硬件设备来实现多路本地音频的混合，这路混合形成的音频信号就包含原来几支话筒所采集到的声音信息，接下来就可以对这路混合音频信号进行采样、量化、编码处理了。

2) 数字音频的叠加

数字音频的叠加是音频模拟信号数字化得到的 PCM 编码数据的叠加处理，多路音频的量化数据进行叠加也可以实现多路音频混合，这可以由软件来实现的。数字音频的叠加需要注意几点：非压缩编码数据、采样率要相同、量化精度要相同。如果需要两路不同采样率的音频数据叠加，需要对其中一路进行重采样处理，以使两路具有相同听采样率；同样的，如果量化精度不同，也需要转换其中某一路的量化精度。经过压缩编码的音频数据，如 AAC、MP3、ogg 等，是不能直接进行线性叠加处理的，需要进行解码处理，保证具有相同的采样率和量化精度后，才能进行叠加。

假设有两组未压缩的音频数据 A、B，它们具有相同的采样率 S、相同的量化精度 N（通常为 8bit、16bit、24bit、32bit 等），对应某时刻 t 的采样数据为 A_t、B_t，那么其混合后的结果 M_t，可以根据下列公式计算。

$$M_t = \begin{cases} A_t + B_t & ,-2^{N-1} < A_t + B_t < 2^{N-1} - 1 \\ 2^{N-1} - 1 & ,A_t + B_t \geq 2^{N-1} - 1 \\ -2^{N-1} & ,A_t + B_t \leq -2^{N-1} \end{cases} \tag{4-5}$$

4.4 基于 FFmpeg 方案的视音频编码和解码模块的实现

视频和音频信号经过数字化处理以后，有大量的空间、时间、编码、视觉和知识等冗余信息[40]，数据量庞大，不宜直接用于网络传输和存储。视频和音频编码可以有效地减少这些冗余信息、降低码率，在流媒体技术所起的重要作用不言而喻。

4.4.1 视音频编解码概述

目前，主流的 H.264 开源编码器有 JM、T264 和 x264[41]：JM 程序结构冗长，编码复杂度高，不适合实时视频传输[42]；T264 只能解 T264 编码输出的码流，目前已不再更新[43]；x264 在保证一定编码性能的同时，编码复杂度相对较低，实用性强，应用广泛 [44]。AAC 音频编码器种类繁多，主要的编码器有 Fraunhofer IIS 、FhG、Nero AAC、QuickTime/iTune、FAAC、Divx AAC。

第二章介绍过的开源跨平台的视频和音频解决方案 FFmpeg，提供了录制、转换以及流化视音频的完整解决方案，支持 MPEG、DivX、MPEG4、FLV 等 40 多种编码和 AVI、MPEG、OGG、Matroska 等 90 多种解码，同时也支持很多第三方的编解码库，音频编解码库如 libfaac、libfdk_aac、libmp3lame、libvo-aacenc、libopus、libvorbis 等，视频编解码库如 libx264/libx264rgb、libxvid、mpeg2、png 等。

本文借助FFmpeg完成h.264视频的编码和解码、AAC音频的编码和解码，以及h.264视频流与 AAC 音频流的复用与保存，因此，将 h.264 视频的编解码、AAC 音频的编解码与其复用存储的实现放到一起。

4.4.2 FFMPEG编解码流程分析

本文使用的FFmpeg SDK是国外网友Zeranoe针对windows系统编译的版本，编译时已经加入了一些外部库，如x264、VisualOn AAC等。FFmpeg项目主要包括[45,46]：libavcodec、libavformat、libavutil、libswscale等。libavcodec用于存放各个codec(编解码器)模块；libavformat用于存放muxer/demuxer(复用器/解复用器，复用器用于将视频流、音频流合并到一个容器文件中，解复用器用于将容器文件中的视频和音频流拆分)模块，libavutil用于内存操作等模块；libswscale用于视频的缩放与转换等。这几大模块中主要的几种常用数据结构及说明如表4-3所示。

表4-3 FFmpeg中常用的数据结构

Table 4-3 The common data structure in FFmpeg

数据结构	说明
AVFormatContext	FFmpeg中表示Format上下文的结构体，是主要的外部接口结构体，包含了多媒体文件或流的基本信息，用于文件输入与输出操作。
AVCodecContext	FFmpeg中表示Codec上下文结构体，其成员包含了与编码和解码有关的参数。
AVCodec	编解码结构体，包含了编解码器信息。
AVStream	描述媒体流的结构体，包含了媒体流的有关信息。
AVPacket	用于暂存解复用后、解码前媒体数据及附加信息的结构体。
AVFrame	用于暂存未压缩编码媒体数据及附加信息的结构体。
AVCodecID	编解码器ID。

下面以解码本地文件为例，简要分析一下使用FFmpeg编码或解码的处理流程，其流程图如图4-7所示。

1) 注册初始化，通过av_register_all()函数初始化FFmpeg的libavformat部分，并且注册所有的复用器、解复用器、协议等，也可以通过其他注册函数分别注册所需的模块。

2) 打开文件，通过avformat_open_input()函数打开一个流（可以是文件也可是网络流），读取其头部信息并解析，创建Format上下文结构体，并设置其部分主要成员变量。

3) 提取流信息，通过avformat_find_stream_info()函数读取媒体文件以获取其流信息，主要是编解码器参数，设置到AVFormatContext结构中streams[i]->codec中，各类媒体，如视频、音频，有自己对应的stream。

4) 查询媒体类型，根据获取到的流信息检查流中媒体的类型。

5) 查找解码器，根据媒体的信息，通过avcodec_find_decoder()函数，根据给定的解码器ID或解码器名称查找对应的解码器，并返回AVCodec指针。

6) 打开解码器，通过avcodec_open2()函数打开上一步查找到的解码器，初始化AVCodecContext结构体。

7) 读取媒体数据帧，通过 av_read_packet()函数从媒体流中读取一帧数据，读取的数据由 AVPacket 结构实例存放。

8) 解码媒体帧，根据上一步读取的帧的媒体类型，使用相应的解码函数调用相应的解码器进行解码处理，如视频解码 avcodec_decode_video2()，音频解码 avcodec_decode_audio4()，解码器由第 5 步查找获取。

9) 关闭解码器，通过 avcodec_close()函数关闭 6 中 avcodec_open2()打开的解码器。

10) 关闭文件，通过 avformat_close_input()函数关闭媒体流，释放资源，关闭 IO。

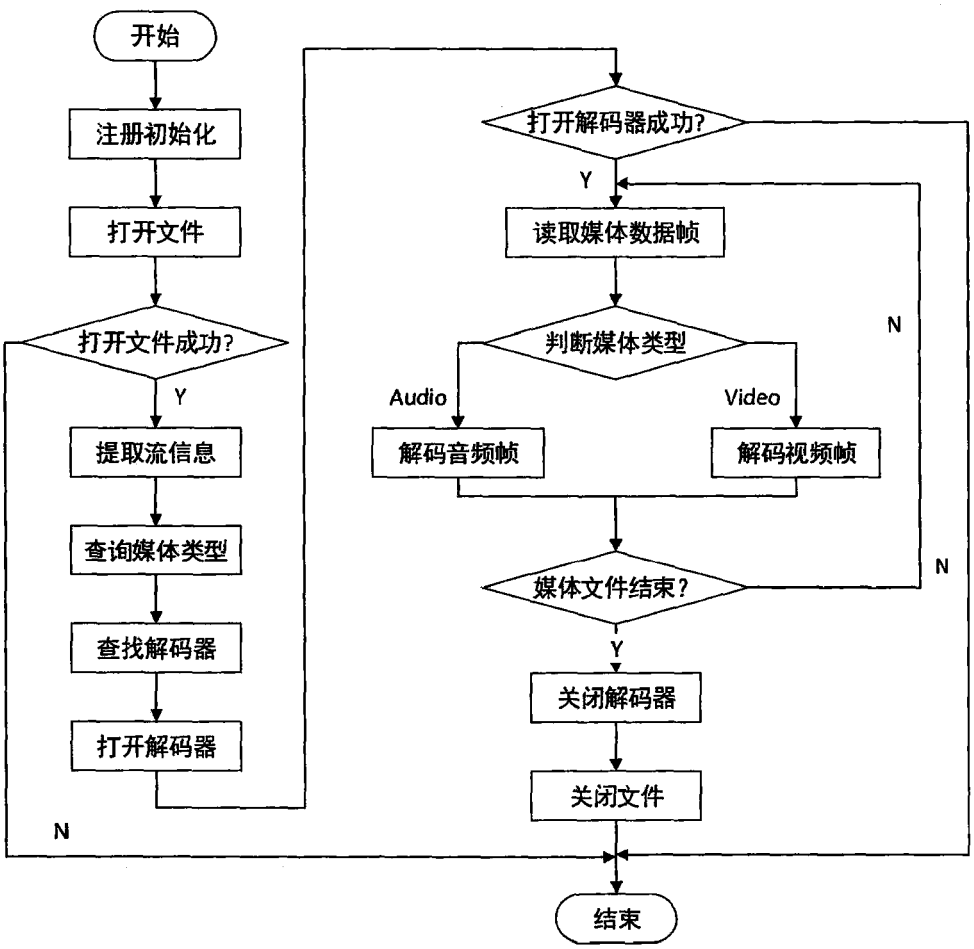


图 4-7 FFmpeg 解码本地文件流程图

Fig. 4-7 The flow chat for decoding local file with FFmpeg

编码过程与解码过程类似，不同的地方是解码时，可以从待解码数据中获取有问解码参数和信息；而编码时，待编码数据通常来源于内存，编码器的选择与编码参数的设置需要根据预设的编码格式及待编码数据有关信息来确定。

FFmpeg 项目由国外编码爱好者设计开发，版本更新跨度较大，较新的版本中往往接口和函数名都发生变化，并且不兼容较旧的版本，同时缺少说明文档，特别是中文文档，这些给国内初学者学习 FFmpeg 开发带来不小的困难。同时，在多线程环境中也有诸多问题，本文利用 FFmpeg 方案实现了自定义的 H.264 与 AAC 的编解码类，并解决了多线程冲突问题。

4.4.3 H.264 视频编码和解码的实现

拼合处理得到的图像需要进行压缩编码，本文视频编码采用 H.264 标准；开发实现时为了测试编码效果，需要解码 H264 流，因而本文也加入解码的实现。编码和解码所需的实时数据都来源于内存，因此，编码和解码有关参数都需要自己设置。下面说明一下本论文基于 FFmpeg 的 H.264 视频编码和解码的具体实现。

4.4.3.1 H.264 视频编码的实现

本论文研究内容在编程时常常需要在一个进程中同时开多个线程进行编码和解码工作，而 FFmpeg 是用 C 语言开发的、面向过程开发的项目，项目源代码中有很多全局的变量和全局的处理，在多线程并发环境下，这些变量就需要进行互斥访问，否则就会出现意想不到的错误，导致程序异常甚至崩溃，早期的开发也经常遇到这个问题，同时，有些全局的注册初始化工作也只需要进行一次就可以了。因此，本文自定义并实现了一个简单的编解码器基类 MyCoder，用于执行一些 FFmpeg 的全局化操作及管理一些全局变量，其它自定义的编解码器都继承 MyCoder，其主要成员变量及函数如表 4-4 所示。

表 4-4 MyCoder 主要成员及函数

Table 4-4 The main members of MyCoder

	成员	描述
变量	allRegistered	静态成员，注册初始化标识；
	subobjectNum	静态成员，已创建对象数量；
函数	MyCoder()	构造函数，执行注册初始化操作；
	FF_LockMGR_callback ()	友元函数，设置临界区，提供线程锁；
	~MyCoder()	析构函数；

由于 FFmpeg 注册初始化处理是全局的，当程序某线程第一个自定义的编解码器对象被创建时完成注册初始化处理就可以了，以后的对象不需要再次进行注册初始化。基类 MyCoder 中定义了静态成员 allRegistered 用于监测注册初始化处理是否已经完成，首次创建的对象会调用基类的构造函数检测 allRegistered，为 false 就执行 av_register_all() 操作，并设置 allRegistered 为 true。另外，FFmpeg 中有不少全局变量需要互斥访问，比如 avcodec_open2()/avcodec_close()函数在多线程环境下是不安全的，原因就是该函数内需要访问一些全局变量，FFmpeg 提供了互斥访问接口，但需要自己实现，锁管理回调函数 FF_LockMGR_callback(void **mutex, enum AVLockOp op)实现了根据条件 op 分别处理临界区 mutex 创建、进入、离开、释放等操作，调用 FFmpeg 锁管理注册函数 av_lockmgr_register()进行注册即可，同样的，锁管理注册也只需要注册一次，注册完成后，当一个线程进入 avcodec_open2()/avcodec_close()中时，FFmpeg 就会调用锁管理回调锁定全局变量处理代码。

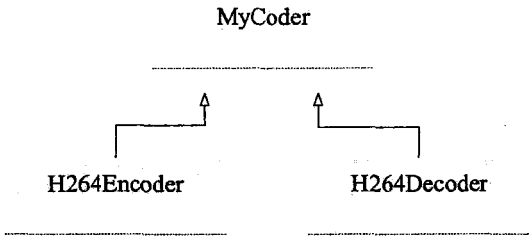


图 4-8 自定义 H.264 编解码器类图

Fig. 4-8 The class diagram of implemented H.264 codec

自定义的 H.264 编码器类 H264Encoder 派生自 MyCoder，该类实现了使用 FFmpeg 进行 h.264 编码的封装，可以很方便地对 4.3.1 中经过拼合处得到的 IplImage 图像进行色彩空间转换、编码等处理，其主要成员变量与函数如表 4-5 所示。

表 4-5 H264Encoder 部分成员

Table 4-5 The main members of H264Encoder

	成员	描述
变量	codecCtx	AVCodecContext*，用于存放编码上下文；
	codec	AVCodec*，用于存放编码器；
	codec_id	AVCodecID，FFmpeg 中指示编码器 ID；
	avframe	AVFrame*，用于存放未编码的数据帧；
	codecOpened	int，用于标识编码器是否打开；
	frame_count	int，已编码帧计数
函数	H264Encoder()	构造函数，设置编码格式、尺寸、帧率、码率，调用 initH264Encoder()完成编码器初始化；
	initH264Encoder()	调用 setCodecContext()完成编码器初始化，并根据编码上下文参数，为 avframe 分配空间；
	setCodecContext()	根据编码格式、尺寸、帧率、码率等，设置编码上下文，查找并打开对应编码器；
	IplImage2AVFrame()	将 opencv 中的图像结构转换成 FFmpeg 中的图像结构 AVFrame，等待编码处理；
	encodeIplImage2AVPacket()	将 opencv 图像编码存放到 AVPacket 结构中，其中需要先调用 IplImage2AVFrame()进行转换；
	resetGopSize()	重置一组图像中图像的帧数
	resetBitRate()	重置平均码率；

几个主要成员函数的具体说明与实现如下，编码流程可以简化为如图 4-9 所示：

- H264Encoder()：需要输入主要的编码参数，如编码格式(AC_CODEC_ID_H264)、宽、高、帧率、码率等，这些参数并不是类的成员变量，而是交由 initH264Encoder()，由该函数完成编码器的初始化，从而初始化成员变量。
- initH264Encoder()：接收用户设置的编码参数，将这些参数传给 setCodecContext()，由 setCodecContext()初始化编码器上下文 codecCtx、编码器 codec 等，并根据 codecCtx 有关参数，为 avframe 分配内存空间。

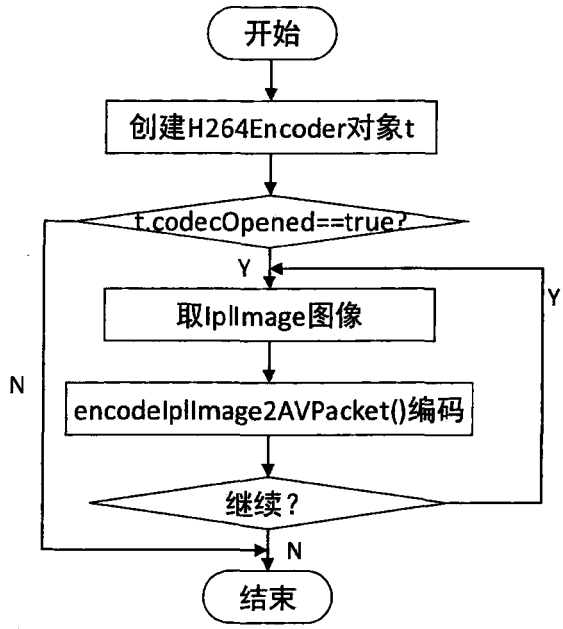


图 4-9 H264Encoder 编码流程

Fig. 4-9 The flow diagram for encoding with H264Encoder

- setCodecContext()：接收 initH264Encoder()传递的编码参数，根据编码格式查找编码器，为编码上下文设置码率、帧率、gop_size（组序列长度）、色彩空间格式、profile（档次）、level（等级）等参数，编码上下文的时间基 time_base（即每两帧的时间间隔）会根据帧率设置成 1/帧率，之后调用 avcodec_open2()打开编码器。
- IplImage2AVFrame()：拼合图像是 RGB 格式，无法直接交给 FFmpeg 进行编码，需要转换成 YUV420P 格式，FFmpeg 提供了转换接口，由 sws_getContext()创建 IplImage 到 AVFrame 的转换上下文(SwsContext*)img_convert_ctx，主要包括设置源图像的尺寸和色彩空间格式，以及目的 AVFrame 的尺寸和色彩空间格式（FFmpeg 中 YUV420P 格式 AV_PIX_FMT_YUV420P），再由 sws_scale()根据 img_convert_ctx 中的转换信息完成转，如果设置转换上下文时目的图像尺寸与源图像尺寸不同，还可以实现图像缩放。需要注意的是转换前要根据目的尺寸和目的色彩空间格式为 AVFrame 分配存储空间。
- encodeIplImage2AVPacket()：编码器初始化完成以后，就可以将源 IplImage 图像，和目的 AVPacket 传给该编码函数，该编码函数内部调用 IplImage2AVFrame()完成待编码数据格式转换，保存在 avframe 中，同时还要当前已编码帧计数 frame_count 为 avframe 设置时间戳 pts，记录当前帧的相对时间信息，单位是编码上下文 codecCtx 的时间基 time_base。之后将 avframe 传给 FFmpeg 视频编码函数 avcodec_encode_video2()，根据编码上下文 codecCtx 完成编码处理，编码后的数据保存在 AVPacket 结构中，带有与 avframe 相同的时间信息 pts，每编码成功一帧 frame_count 加 1。
- resetGopSize()：H.264 编码是一种参考编码，它以若干帧图像序列为单位进行组织，根据表 2-2 包含图像信息的 NALU 主要是 IDR 图像和非 IDR 图像。其中 IDR 图像采用帧内压缩算法，不依赖相邻帧，包含重构图像的完整信息，解码时可独立构成图像，

数据量较大；非 IDR 图像采用帧间压缩算法，无法独立成像，需要依赖 IDR 帧或其他相信帧，数据量较小。一个 GOP 就是由一帧 IDR 帧和若干非 IDR 帧组成，改变编码上下文中 codecCtx 的 gop_size，能改变数据量较大的 IDR 帧的出现频率，可以在一定程度上改变数据量，当然 gop_size 不宜过大。

- resetBitRate()：初始化编码器时需要设置 bit_rate（实际是平均码率），码率不宜过低，过低的话图像的质量会非常差，也不宜过高，过高图像质量没有多少提升，但会导致数据量增加，不利于存储和网络传输。实时直播时，可以要据接收端的反馈信息，适当调整码率。

4.4.3.2 H.264 视频解码的具体实现

解码同样需要对 FFmpeg 各模块进行注册初始化，因此，自定义的 H.264 解码器类 H264Decoder 同样派生自 MyCoder，如图 4-8。实现了 H.264 编码，解码处理就相对简单一些，自定义 H.264 解码类 H264Decoder 部分主要成员如表 4-6 所示。

解码与编码有诸多相似之处，自定义的 H264Decoder 在结构上与上一节中的 H264Encoder 有许多类似的地方，如由构造函数 H264Decoder()调用 initH264Decoder()函数，由 initH264Decoder()函数调用 setCodecContext()函数完成解码器、解码上下文的初始化处理，不同之处在于，解码时码流数据是即定的，帧率、比特率等都不需要再做设置，只需要设置编码器 ID、宽高即可。

表 4-6 H264Decoder 部分成员

Table 4-6 The main members of H264Decoder

	成员	描述
变量	codecCtx	AVCodecContext*，解码上下文
	Codec	AVCodec*，解码器
	Codec_id	AVCodecID，解码器 ID
	avframe	AVFrame*，保存未编码图像
	codecOpened	Int，解码器打开标识
函数	H264Decoder()	构造函数，接收解码器 ID、宽、高等参数，付给 initH264Decoder()进行初始化；
	initH264Decoder()	调用 setCodecContext()进行初始化处理，及 avframe，将解码器打开情况保存在 codecOpened;
	setCodecContext()	要据解码参数，设置解码上下文、查找对应解码器，并打开解码器；
	AVFrame2IplImage()	将 FFmpeg 未编码图像结构 AVFrame 转换成 opencv 图像结构 IplImage；
	decodeAVPacket2IplImage()	将压缩帧解码，转换成 opencv 图像结构 IplImage；

确定 codecOpened 为 true 即解码器正常打开后，就可以接收 h.264 数据帧交给 decodeAVPacket2IplImage()解码，该解码函数内部调用 FFmpeg 视频解码函数 avcodec_decode_video2()，根据解码上下文将 h.264 帧解码成 YUV420P 格式图像数据，并保存在 avframe 中，之后由 AVFrame2IplImage()完成 FFmpeg 图像结构到 OpenCV 图

像结构的转换，其实现类似 H264Encoder 中的 IplImage2AVFrame()，使用 H264Decoder 解码 H.264 数据帧流程可以简化为如图 4-10 所示。

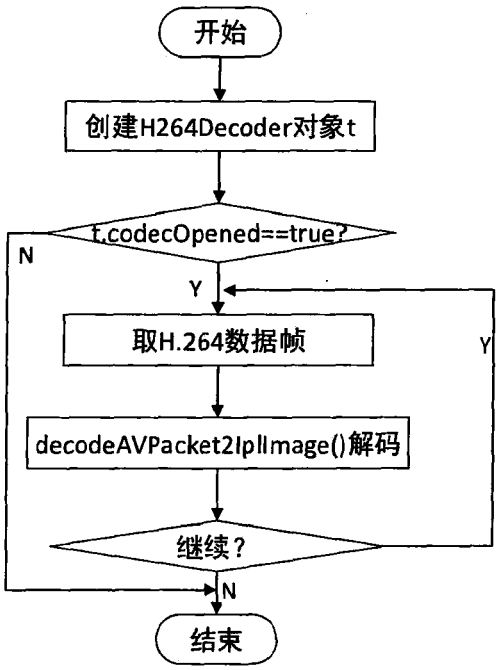


图 4-10 H264Decoder 解码流程

Fig. 4-10 The flow diagram for decoding with H264Decoder

4.4.4 AAC 音频编码和解码的实现

采用 FFmpeg 对采集到的数据进行编码、以及对经过压缩编码的数据进行解码，其基本处理流程与对视频的编解码处理类似。为测试编码效果需要解码 AAC 数据帧，本文实现了基于 FFmpeg 的 AAC 音频编码类 AACEncoder 与 AAC 音频解码类 AACDecoder，可以简化 AAC 编码与解码处理。

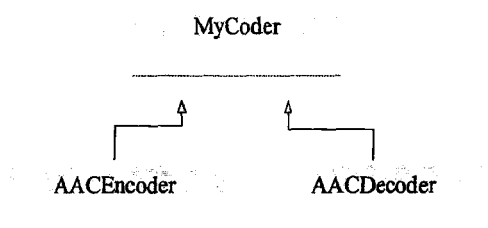


图 4-11 自定义 AAC 编解码类

Fig. 4-11 The class diagram of implemented AAC codec classes

4.4.4.1 AAC 音频编码的具体实现

自定义的 AAC 编码类 AACEncoder 同样继承了 MyCoder，这样就避免了多线程中 avcodec_open2()/avcodec_close()调用出错的问题。AACEncoder 音频编码的基本思路是设置编码器 ID、采样率、声道等编码参数，根据这些参数执行编码上下文、编码器的初始化，之后接收采样到的音频量化数据，编码成 AAC 格式，AACEncoder 类的部分

主要成员如表4-7所示。

表4-7 AACEncoder部分成员

Table 4-7 The main memebers of AACEncoder

	成员	说明
变量	sample_rate	int，音频采样率；
	channles	int，声道数；
	bit_rate	int，比特率；
	codecCtx	AVCodecContext*，编码上下文
	codec	AVCodec*，编码器
	codec_id	AVCodecID，编码器ID
	avframe	AVFrame*，保存未编码图像
	codecInitialized	int，编码器初始化标识
	frame_count	int，音频编码帧计数
函数	AACEncoder()	构造函数，接收输入的编码器ID、采样率、声道数、比特率；
	initAACEncoder()	调用setCodecContext()执行初始化处理；
	setCodecContext()	根据编码参数，设置编码上下文、查找对应编码器，并打开；
	encodeS16PCM2AAC()	音频量化数据编码成AAC格式
	setAVPacketSlient ()	生成一个静音帧

AACEncoder的构造函数AACEncoder()可以接收用户输入的编码器ID（AV_CODEC_ID_AAC，AAC编码在FFmpeg中的ID）、采样率、声道数、比特率等进行部分成员变量的初始化，但部分成员如编码上下文codecCtx、编码器codec需要依赖其他成员进行进一步处理才能完成初始化。initAACEncoder()将采样率、编码ID、声道数等参数传递给setCodecContext()，并将返回值传给编码器初始化标识codecInitializated。setCodecContext()接收到编码器ID、采样率、声道数、比特率后，根据编码器ID找到AAC编码器，并为编码上下文codecCtx分配好空间，设置好编码上下文中的采样率、声道器、比特率、线程计数，同时还要设置待编码音频量化数据的格式，如AV_SAMPLE_FMT_S16（量化精度为16位的交叉排列有符号数据），如果这里设置的量化数据格式与采集得到的待编码数据的格式不匹配就无法完成编码。设置好编码上下文各主要参数后，就可以打开编码器，完成编码上下文和编码器的初始化，编码上下文的时间基time_base会被设置成1/采样率，即每两个采样的时间间隔为1/采样率，单位为秒，那么如果某编码格式一帧中有samples个采样，该编码格式每帧的时长就是samples/采样率。

编码函数的原型是encodeS16PCM2AAC(uint8_t* pcm_data,int data_size,AVPacket* packet)，pcm_data是待编码PCM数据，data_size为pcm_data长度，packet为目的编码包。每种音频编码格式一帧中采样的数量sampleNum都是固定的，如AAC一帧中有1024个采样、MP3一帧中有1152个采样，如果确定声道数channels、量化精度bits，就可以

确定该格式每帧待编码量化数据的大小 data_size，单位为 byte，如公式 4-6。

$$data\_size = sampleNum * channles * (bits / 8) \quad (4\text{-}6)$$

因此，本文音频使用 AAC 编码，采集音频时每次也应该采集 1024 个采样，其大小 data_size 就是 1024*2*(16/8)=4096 字节。将量化数据 pcm_data 地址传给 avframe 的数据指针，根据编码帧计数 frame_count 为 avframe 设置时间戳 pts，由于 AAC 编码中每帧有 1024 个采样，那么两帧的时间戳相差 1024 个时间基，pts 的值就是 frame_count*1024。之后，根据编码上下文，调用 avcodec_encode_audio2()完成编码，编码后的数据保存在 packet 中，其也带有时间信息 pts，与 avframe 的 pts 相同，每次编码成功后 frame_count 加 1。使用 AACEncoder 进行 AAC 编码，其处理流程可以简化为如图 4-12 所示。

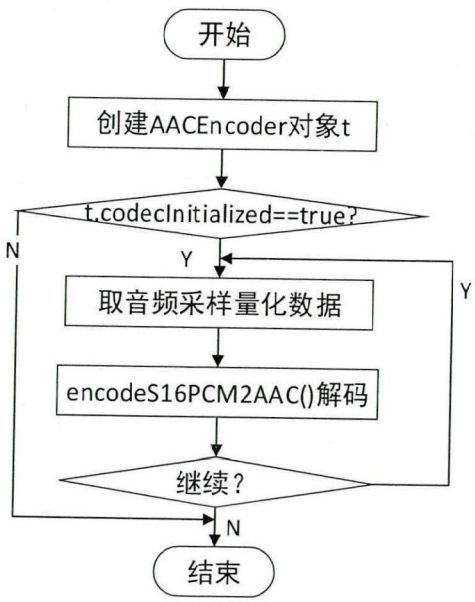


图 4-12　AACEncoder 编码流程

Fig. The flow diagram for encoding with AACEncoder

4.4.4.2 AAC 音频解码的具体实现

音频解码的主要作用是将音频编码数据解码成 PCM 数据，PCM 数据可以进行叠加或其它分析处理，也可以解码后送给声卡，经声卡转换成模拟电信号后由音响设备播放。本文自定义的 AAC 音频解码类，依靠 FFmpeg 实现解码功能，通过 SDL 库实现播放功能，其主要成员如表 4-8 所示。

AACDecoder 的初始化过程同 AACEncoder 类似，解码是处理即定的数据，因此不需要设置码率，只需要输入解码器 ID、采样率、声道数，就可以通过 AACDeocder()->initAACDecoder()->setCodecContext()一步步的调用，实现包括解码上下文、解码器等有关解码成员变量的初始化。有关播放的成员变量可以在需要播放时再进行初始化。

initAACDecoder()返回值用于初始化 codecInitialized，通过该成员可以判断必要有关解码成员的初始化状态，如果无异常就可以接收 AAC 编码数据帧，交由

decodeAVPacket2S16PCM()解码，通过调用FFmpeg的音频解码函数avcodec_decode_audio4()，根据初始化成功的解码上下文，就可以将AAC编码的数据解码成PCM数据，需要注意的是如果是双声道，较新版本的FFmpeg解码得到的PCM数据格式是Planar格式的，如FFmpeg中的AV_SAMPLE_FMT_S16P(16位量化精度Planar格式有符号数据)，Planar格式的PCM数据是不能直接播放的，需要转换成左右声道交叉排列才可以播放，为了方便播放及其他处理，解码完成后，decodeAVPacket2S16PCM()函数需要调用重采样转换函数PCMResample ()，使得解码输出的PCM数据左右交叉排列。

表4-8　AACDecoder部分成员列表

Table 4-8 The main members of AACDecoder

	成员	描述
变量	Sample_rate	int，音频采样率
	channels	int，声道数
	codecCtx	AVCodecContext*，解码上下文指针
	codec	AVCodec*，解码器指针
	codecID	AVCodecID，解码器ID
	Avframe	AVFrame*，未编码数据帧指针
	codecInitialized	int，解码器初始化标识
	wanted_spec	SDL_AudioSpec，SDL音频规格结构体，播放用。
	audio_chunk	uint8_t*，待播放PCM数据起始地址，播放用。
	audio_len	uint32_t*，待播放PCM数据长度，播放用。
	audio_pos	uint8_t*，待播放PCM数据地址，播放用。
	audioOpened	int，播放器打开标识，播放用。
函数	AACDecoder()	构造函数，接收输入的编码器ID、采样率、声道数；
	initAACDecoder ()	调用setCodecContext()执行初始化处理；
	setCodecContext()	根据解码参数，设置解码上下文、查找对应解码器，并打开；
	decodeAVPacket2S16PCM()	解码函数
	PCMResample ()	重采样及格式转换函数
	playAudio_callback()	友元函数，播放回调函数，需要访问audio_len、audio_pos等成员。
	playS16PCM()	播放函数
	AudioOpened()	SDL初始化函数

PCMResample()可用于音频重采样及格式转换，FFmpeg定义了重采样及转换上下文SwrContext用于记录源音频与目的音频的声道格式(单声道或立体声)、采样率、数据格式等，为SwrContext对象分配空间、设置好转换源、目的的信息，通常只是做数据格式转换，因此源、目的的采样率与声道格式是相同的，初始化以后，就可以使用swr_convert()函数完成转换。

如果需要播放解码后的音频的话，需要先调用SDLInit()初始化有关播放的成员变量。AudioOpened()主要的处理是调用SDL_Init()初始化SDL音频播放库，之后初始化

SDL 音频规格结构 wanted_spec，该结构的成员主要包括待播放音频数据的采样率 freq、数据格式 format（包含量化精度）、声道数 channels、单帧采样数 samples、播放回调函数指针 callback、用户数据 userdata 指针等，callback 指向播放回调 playAudio_callback()，userdata 指针指向当前 AACDeoder 对象并传递给 playAudio_callback()，以便其能访问 audio_pos、和 audio_len。最后，根据 wanted_spec 由 SDL_OpenAudio()打开音频播放设备。整个处理若无异常，设置 audioOpened 为非负值，表示播放器正常打开。

音频回调函数 playAudio_callback()的作用就是将数据送给声卡播放了，其原型是 playAudio_callback (void *userdata,Uint8 *stream,int len)，userdata 即 AudioOpened()设置指向当前 AACDecoder 对象，stream 为 SDL 播放缓冲区，len 为 stream 长度，由 SDL 根据 wanted_spec 中的 format、channels、samples 计算。其主要代码其说明如图 4-13 所示：

```
//获取当前对象指针
AACDecoder* dcr=(AACDecoder*)userdata;
//待播放数据长度为0
if(dcr->audio_len==0)
   return;
//len初始值是确定值，当有待播放数据传入时，它总是等于初始值与待播放数据长的较小者；
len=(len>dcr->audio_len?dcr->audio_len:len);
//将待播放数据输入SDL播放缓冲区
SDL_MixAudio(stream,dcr->audio_pos,len,SDL_MIX_MAXVOLUME);
//待播放数据地址前移
dcr->audio_pos += len;
//待播放数据长度相应减少
dcr->audio_len -= len;
```

图 4-13　playAudio_callback()主要代码

Fig. 4-13 The main code of playAudio_callback()

播放时，playS16PCM()需要检测 audioOpened，若正常，audio_chunk 指向解码得到的 PCM 数据，audio_len 等于数据长度，audio_pos 初始值指向 audio_chunk，调用 SDL_PauseAudio(0)启动音频播放，直到一帧解码后的数据播放完成。

4.5 视音频复用存储模块的实现

编码后的视频流和音频数据可以复用存储，通俗点儿说就是将视音频编码数据写入一个文件中，如 MP4 或 MKV 文件等。FFmpeg 支持将 H.264 和 AAC 编码的数据复用到MP4、MKV、FLV等格式的文件中。接下来说明自定义H.264与AAC复用器HA Muxer的实现，由于添加视音频流的时候需要打开编码器，为了防止多线程冲突，HA Muxer也继承了 MyCoder。

不同的多媒体容器格式有着不同的封装规则，不同的容器有不同的封装格式，如 MP4 将文件数据分为 Metadata 和 Media Data 两部分，并将两者分开存放[47]；FLV 文件

主要由 File header、File body 两部分组成[48]；MKV 容器整体可分为 EMBL(Extensible Binary Meta Language) header 和 Segment 单元[49]。HA_Muxer 借助 FFmpeg 对 MP4 及 MKV 等容器格式的复用进行了封装，使用时不必考虑特定容器的封装细节，主要处理过程包括初始化、添加音频流/添加视频流、写文件头、音频/视频数据写入容器、写文件尾等，如图 4-14 所示。

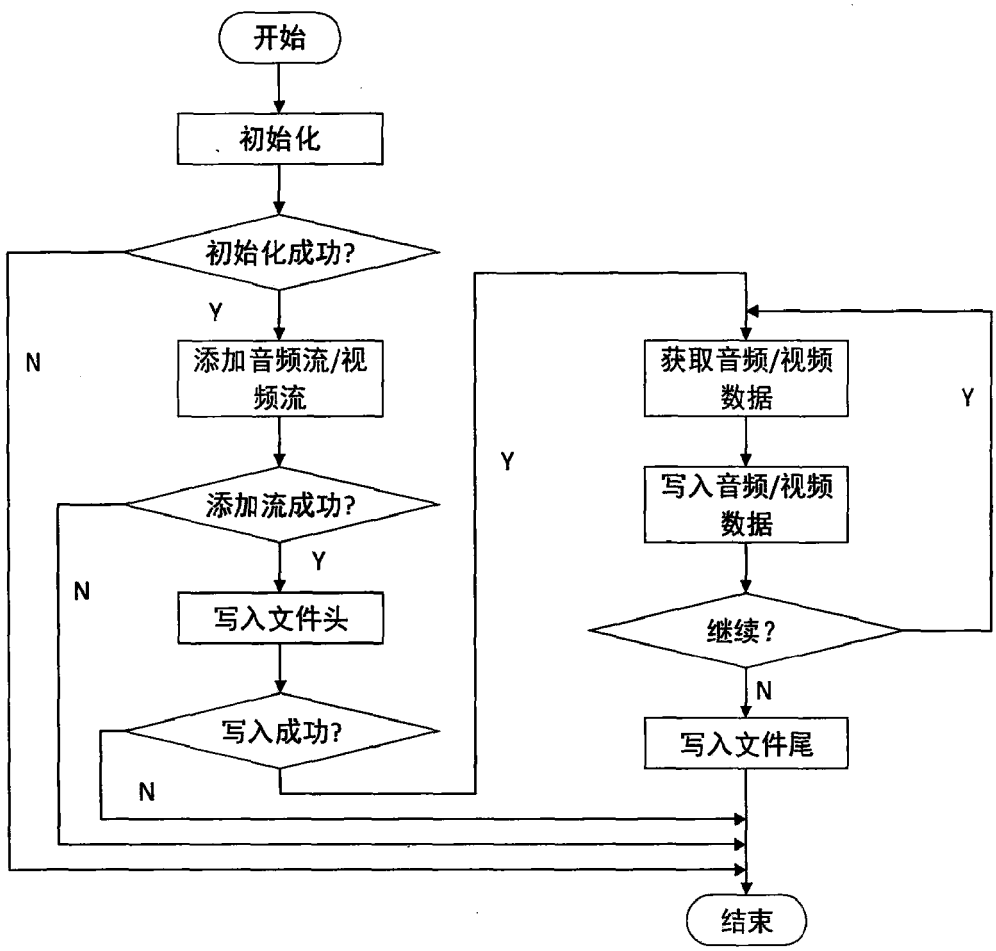


图 4-14　HA_Muxer 处理流程

Fig. 4-14 The flow diagram for storing with HA_Muxer

HA_Muxer 派生于 MyCoder，通过父类与子类的构造函数共同完成初始化，之后需要添加待存储的视频和音频流信息，再根据容器格式与待存储媒体流信息生成相应的文件头部信息，写入到文件中，然后就可以不断地获取视音频的编码数据，并写入媒体文件了，储存结束后需要写入文件尾，结尾文件写入，同时将一些必要信息回填到文件中，HA_Muxer 的部分主要成员如表 4-9 所示。

初始化主要是输出文件及容器的初始化处理，由构造函数完成，其原型为 HA_Muxer(char* filename)，传入带后缀的目的媒体文件名，如 file.mp4，由 FFmpeg 中为输出格式分配格式上下文的函数 avformat_alloc_output_context2()分析文件后缀，根据此后缀分析对应的容器，从而初始化相应的复用器及目的文件，并将结果保存在 fmtCtx 成员变量中。同时，还需要通过 avio_open()函数打开 FFmpeg 的 I/O，将打开结果保存

在avioOpened变量中，以便后续判断。

下一步就是添加视/音频流信息了。addVideoStream()成员函数负责添加视频流信息，需要传入视频编码ID、宽和高、帧率、码率、gop_size（组序列长度）等参数，根据编码ID找到相应的编码器，再根据编码器通过FFmpeg流添加函数avformat_new_stream()为fmtCtx添加视频流描述，并将videoStream指向新生成的视频流描述，然后就是为视频流描述videoStream初始化编码器上下文，用编码ID、宽和高、帧率、码率、gop_size等参数初始化videoStream编码器上下文相应的成员，再打开编码器即可，这时视频流描述videoStream会根据其编码器上下文的时间基time_base生成一个自己的时间基time_base，两者的值可能不一样。另外，也需要用传入的帧率来初始化类的成员fps。添加音频流信息类似，可以参考添加视频流信息与音频编码器初化过程。

表4-9 HA_Muxer部分主要成员

Table 4-9 The main members of HA_Muxer

	成员	描述
变量	fmtCtx	AVFormatContext*，描述输出媒体文件的构成和基本信息
	outputFmt	AVOutputFormat*，描述输出容器格式
	audioStream	AVStream*，描述音频流信息
	videoStream	AVStream*，描述视频流信息
	avioOpened	bool，I/O打开标识
	avioClosed	bool，I/O关闭标识
	fps	float，视频帧率
	sample_rate	int，音频采样率
	videoFrameNum	int，写入的视频帧数
	audioFrameNum	Int，写入的音频帧数
函数	HA_Muxer()	构造函数，做初始化处理
	addVideoStream()	添加视频流信息
	addAudioStream()	添加音频流信息
	writeHeader()	写对应容器头部
	writeAVPacket()	写入视/音频编码数据
	writeTrailer()	写文件尾
	getWrittenVideoFrameNum()	获取已写入视频帧的数量
	getWrittenAudioFrameNum()	获取已写入音频帧的数量
	getWrittenVideoDuration()	获取已写视频流的时间长度,单位ms
	getWrittenAudioDuration()	获取已写音频流的时间长度,单位ms

经过文件初始化及视音频流信息添加，就获得了存储必要的容器信息及所需的流信息了，这些信息都保存在fmtCtx中，接下来需要根据avioOpened判断I/O是否打开，如果已打开就可以调用FFmpeg函数avformat_write_header()，根据fmtCtx中的各种信息，为存储文件写入头部了，这一步处理由成员函数writeHeader()完成。

接下来就可以将一帧帧编码后的视音频数据写入文件了。4.2章节编码生成的H264

帧和 AAC 帧保存在 AVPacket 的对象中，并带有时间信息 pts，其单位是对应编码器上下文的时间基 time_base（视频是 1/帧率，音频是 1/采样率）；而 fmtCtx 中的流信息结构 AVStream 有自己的时间基，FFmpeg 要求将 AVPacket 对象的时间信息 pts 换算成以 fmtCtx 对应流信息结构 AVStream 的时间基为单位的新时间信息后，才能进行存储，其变换公式为如公式 4-7。

$$newPts = pts / codecCtxTimeBase * streamTimeBase \quad (4\text{-}7)$$

其中，codecCtxTimeBase 为编码器上下文中的时间基，streamTimeBase 为流信息描述中的时间基，pts 为原编码帧的时间信息，newPts 为新的时间信息。此外，编码生成的 H264 数据帧中带有起始码“00 00 00 01”或“00 00 01”，AAC 数据帧中带有 ADTS 头，存储写入文件时需要去掉起始码或 ADTS 头。经过些必要的处理，就可以利用 FFmpeg 写帧函数 av_interleaved_write_frame()将编码后的 H264 帧或 AAC 帧写入媒体文件了。writeAVPacket()成员函数可以针对 H264 数据和 AAC 数据进行上述相应处理，将视音频编码写入文件，每写入一帧 H264 视频帧 videoFrameNum 加 1，每写入一帧 AAC 音频帧 audioFrameNum 加 1，这样就可以随便获取已存储的视频和音频帧的数量。

由于类的成员 videoFrameNum 和 audioFrameNum 分别记录着已写入文件 的视频帧和音频帧的数量，就可以通过成员函数 getWrittenVideoFrameNum() 和 getWrittenAudioFrameNum()分别获取这两个值。同时，也可以根据 videoFrameNum 和视频帧率 fps、audioFrameNum 和音频采样率 sample_rate，获取已写入的视频时长 videoDuration、音频时间长audioDuration，如公式 4-8 及 4-9。

$$videoDuration = 1000 * videoFrameNum / fps \quad (4\text{-}8)$$

$$audioDuration = 1000 * audioFrameNum * SamplesPF / sample\_rate \quad (4\text{-}9)$$

其中，SamplePE 为每帧中采样的数量，AAC 为 1024，ideoDuration、audioDuraion 的单位是 ms。

最后，如果需要结束存储的话，需要调用 writeTrailer()成员函数，该函数通过调用 FFmpeg 中 av_write_trailer()往文件头部回调一些必要信息，并结束存储。

有关资源的释放和 I/O 的关闭等放在析构函数中。

4.6 本章小结

本章详细说明多源视音频实时直播系统前端模块的具体实现过程，主要包括视频和音频的采集，视频多画面的拼合、多路音频的混合等处理，接下来重点说明的是 H264 视频的编解码实现与 AAC 音频的编解码实现。此另，本章还对采集速度进行了分析和探讨。

第五章　基于Live555框架的流媒体直播服务器模块的设计与实现

获得包含多场景画面的 H.264 视频流和多路声音的 AAC 音频流，将这一路 H264 视频数据和一路 AAC 音频数据交给流媒体直播服务器，由服务器完成视音频数据流的分析、打包、发送处理，就实现了将实时视音频数据直播到网络中。当需要观摩时，接收端完成与直播服务器的 RTSP 交互，即可从网络中获得实时的视频流，经过解码后，就可以将实时的视频和音频呈现在观众面前。本章具体说明 RTSP 直播服务器模块的实现。

5.1 引言

市场上的流媒体服务器软件种类繁多，商业的功能强大、性能稳定，并能提供完善的技术支持服务，但往往采用私有的媒体封装格式及私有的传输协议，且价格高昂，并不开放核心模块的源代码。开源的也不容小觑，其开放源代码，允许开发人员自行修改完善以满足合特定的应用场景和需求，受到广大流媒体爱好者、开发人员及中小型企业用户的欢迎，同样有着不俗的发展潜力和应用前景。

Live555 设计模式合理、代码短小精悍，具有良好的跨平台、易于拓展等特性，支持 RTSP、RTP/RTCP、SDP、SIP 等标准流媒体协议，支持市场上众多视频编码格式数据（如 MPEG、H.263+、H.264、H.265、JPEG、DV 等）、音频编码格式数据（如 MP3、OGG、WAV、AAC 等），以及 MKV、AVI、FLV 等容器文件的流化、封装、发送、接收等处理，其易于拓展的特性也保证了对其他非主流编码格式及容器文件的支持，能为流媒体服务器提供基本的框架。本论文中流媒体直播服务器正是基于该框架实现的。

5.2 Live555 框架分析

Live555 代码庞大，无说明文档，给对 live555 的学习带来不小的困难。本节将对 live555 框架的协议结构、内容、消息处理、RTP 打包等方面进行分析，为直播服务器的实现奠定基础。

5.2.1　Live555 流媒体协议结构分析

Live555 框架实现了有关流媒体的几种主要协议，其中 RTP 协议主要用于传输多媒体数据，RTCP 协议主要用于传输控制，RTSP 协议主要用于流的控制，其协议结构图如图 5-1 所示。

H264 和 AAC 等视音频媒体数据通常经过 RTP 协议打包封装以后，再交由下层协议发送。Live555 不但支持"rtp over udp"，同样支持"rtp over tcp"。一般情况下，视音频媒体数据经过 RTP 协议封装以后通过 UDP 协议发送到网络中，UDP 协议采用无链接方式，不保证数据的可靠性，能很快地将数据量庞大的视音频媒体数据发送出去，又支持组播，发送效率较高，在进行视音频直播时有较小的延时，能保证视音频具有较好的实时性；但由于是采用无链接方式，在网络环境较差的广域网中，很难保证接收端接收视音频数据的完整性、流畅性。如果需要将视音频媒体数据发送到广域网中，可以选择通过 tcp 协议发送，这样可以保证视音频数据的完整性和流畅性，但网络延时可能会较大一些。

本论文研究内容是应用于网络情况较好的专用网络，同时接收观摩端的数量也具有不确定性，需要进行组播，因此，选择"rtp over udp"，即使用 UDP 传输层协议发送视音频媒体数据。

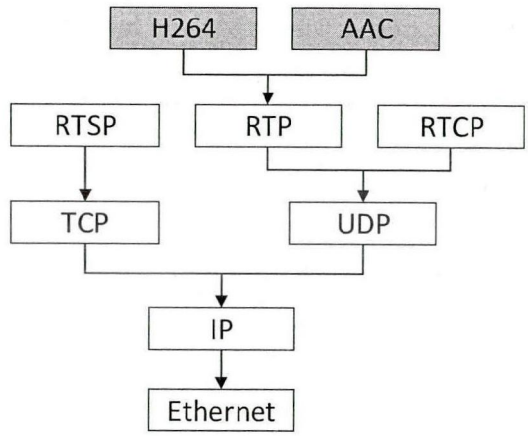


图 5-1 live555 流媒体协议结构

Fig. 5-1 The structure of streaming protocols in live555

5.2.2 Live555 结构分析

Live555 核心部分主要由四大模块组成，分别是 UsageEnvrionment、groupsock、liveMedia、BasicUsageEnvironment 等几个基本类库[50]，各类库又有自己的子目录，各模块间的关系图[51],如图 5-2 所示。

1) UsageEnvironment 类库模块

该模块代表整个系统的运行环境，由使用环境（UsageEnvironment）、任务调度表（TaskScheduler）等几个主要的抽象类组成。其中，UsageEnvironment 类主要负责处理各类消息，特别是错误消息[52]；TaskScheduler 类主要用于延时事件的调度和有关处理。这些类都被定义成抽象类，开发人员可以根据程序的实际运行环境（如 GUI 环境或脚本语言环境等）去实现一些子类。

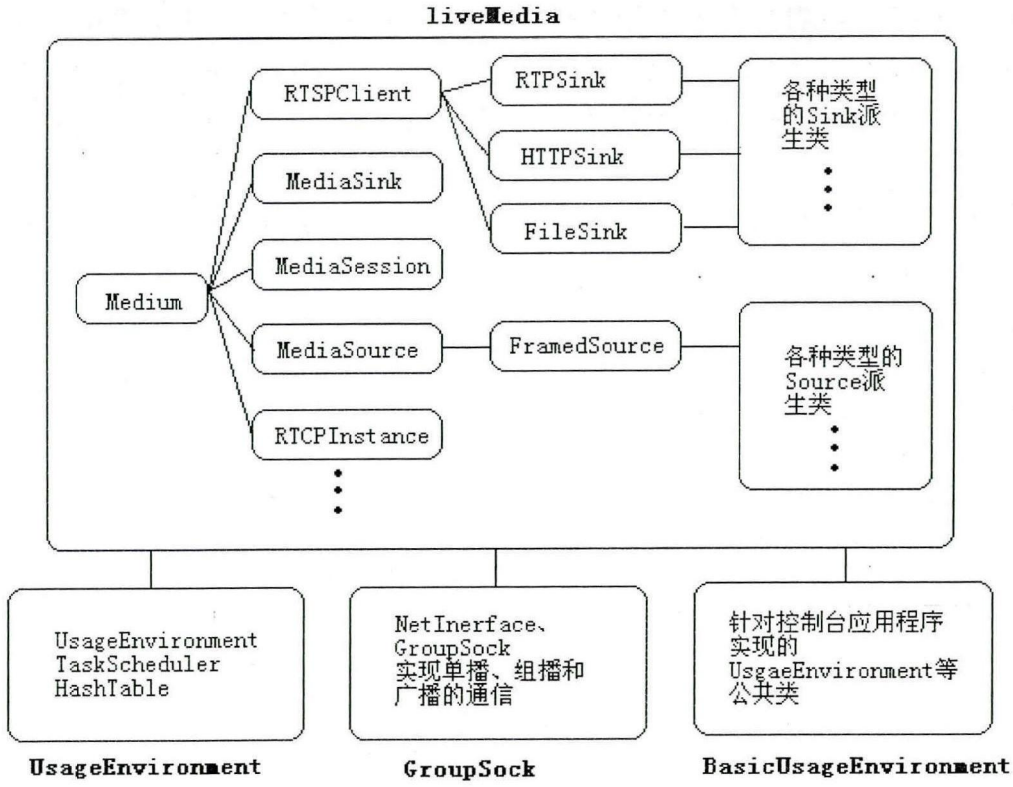


图 5-2 live555 各模块关系

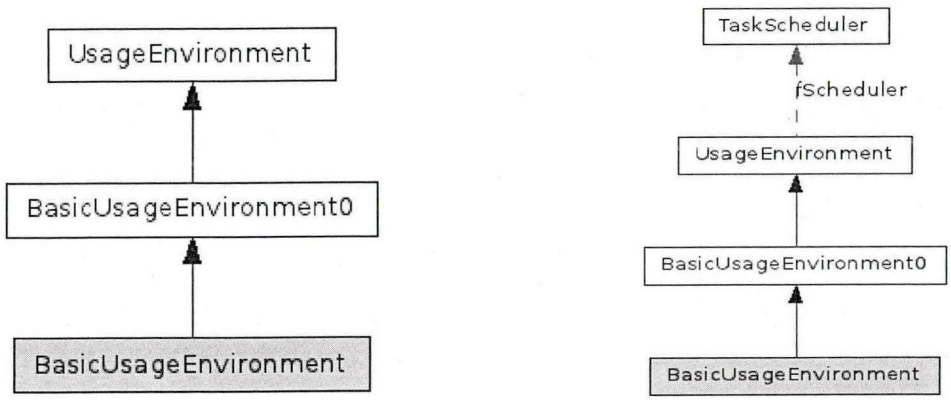
Fig. 5-2 The relationship of modules in live555

2) groupsock 类库模块

groupsock 类库实现了对网络接口的封装，负责与客户端的数据通信工作，可用于视音频等媒体数据的发送和接收，主要支持组播通信，但同样支持单播和广播等通信模式[53]。

3) BasicUsageEnvironment 类库模块

该模块是对 UsageEnvironment 模块的继承，BasicUsageEnvironment 类是 UsageEnrironment 的子类，代表简单的控制台运行环境，其关系如图 5-3 所示[54]。



a. 类图　　b. 协作图

图 5-3 BasicUsageEnvironment 与 UsageEnvironment 关系

Fig. 5-3 The relationship between BasicUsageEnvironment and UsageEnvironment

4) liveMedia类库模块

liveMedia类库是整个live555项目的核心部分，它实现RTSP交互、多种编码格式的视音频媒体数据的RTP打包、传输和解析等。该类库为各种流媒体类型和编解码器定义了一个类层次结构，根为抽象类Medium。Medium类派生出一系列用于实现流媒体服务的子类，部分重要子类及主要作用说明如表5-1所示，这些类又继续派生各种各样的子类，使得live555能够支持多种多样的编码格式及容器格式。

表5-1 liveMedia中部分重要类

Table 5-1 Several key classes in liveMedia

类	说明
RTSPServer	用于构建一个RTSP服务器，其内部定义的RTSPClientSession用于处理单独的客户会话.
RTSPClient	用于处理RTSP请求的发送和响应的解析，并负责创建RTP会话。
MediaSource	数据源抽象类，其不同的派生类Source负责获取不同编码格式或容器的数据。
MediaSink	数据消费者抽象类，其不同的派生类Sink负责不同数据的存储或发送等。
MediaSession	媒体会话类，其不同的子类（MediaSubSession）代表不同的不同的视频或音频会话。
MediaSubSession	子会话描述类，保存了流媒体的主要信息，其对象实例由MediaSession创建和管理，可以将Source与Sink联系起来。
RTCPInstance	用于RTCP协议通信，对RTP报文进行统计和分析。

5.2.3 Live555的RTSP请求处理与数据处理分析

第二章图2-2展示了RTSP协议服务器端与客户端的交互过程，为了能利用live555框架实现实时视音频直播，需要对其会话处理过程进行分析：

(1) 建立RTSP连接

通过RTSPServer类创建一个RTSP服务器，创建服务器的时候需要先建立一个Socket，用于监听TCP的端口；然后将RTSPServer类中用于处理客户端连接的函数句柄和Socket句柄传给任务调度器taskScheduler，taskScheduler会将Socket句柄放入Socket句柄集fReadSet中，与此同时，Socket句柄与连接处理句柄incomingConnectionHandler()会被关联在一起；接下来，程序进入到消息主循环doEventLoop()中，等待客户请求及待处理任务。如果有客户端发出RTSP请求，select函数会返回对应的socket句柄，并根据socket句柄与连接处理句柄的对应关系，找到相应的连接处理句柄并创建一个RTSPClientSession实例处理客户会话。

(2) OPTION请求处理

当服务器端收到客户端发出OPTION请求时，doEventLoop()主循环找到相应的处理句柄，对报文进行解析，如果解析是OPTION报文，则进入

RTSPClientSession::handleCmd_OPTION()函数进行相应处理，将可用的方法发送给客户端。

(3) DESCRIBE 请求处理

DESCRIBE 请求的解析与 OPTION 请求的解析过程类似，不同的是解析之后的处理，当服务器端解析当前请求为 DESCRIBE 时，进入 RTSPClientSession::handleCmd_DESCRIBE()函数，同时根据客户端请求的 URL 后缀（流名称，直播时通常可以自定义流名称，在直播服务器实现时会具体说明），查找对应的流媒体会话 ServerMediaSession。再由 ServerMediaSession 组装生成媒体流的 SDP 信息，并将该 SDP 信息发送给客户端。

(4) SETUP 请求处理

解析过程依然和前面两种请求类似，解析结果为 SETUP 后，调用 RTSPClientSession 类的成员函数 handleCmd_SETUP()进行处理，接着调用 parseTransportHeader()函数解析 SETUP 请求的传输头，下一步通过特定的媒体子会话 ServerMediaSubSession（如 PassiveServerMediaSubsession）的 getStreamParameters()成员获取流媒体传输参数，如通信协议、通讯方式、地址等[55]，将这些参数组装成消息再发送给接收端。客户端需要为每一个请求的流发送一次 SETUP 请求，如果待发送的媒体流同时视频和音频，客户端需要发送两次请求。

(5) PLAY 请求处理

服务器端解析 PLAY 请求的处理同上，解析确认是 PLAY 请求后，调用 RTSPClientSession::handleCmd_PLAY()成员处理客户端的播放请求。Live555 获取视音频媒体数据后，进行分析、RTP 封装、发送要经过一系列的函数调用。以下是是对某视频编码格式数据进行 RTP 打包和发送的大概过程。

先是调用媒体子会话 ServerMediaSubSession（不同的编码格式通常对应不同的子会话）的成员函数 startStream()，接下来的调用顺序如下：

->MediaSink::startPlaying()

->MultiFramedRTPSink::continuePlaying()

->MultiFramedRTPSink::buildAndSendPacket()

->MultiFramedRTPSink::packFrame()

->FramedSource::getNextFrame()

-> MPEGVideoStreamFramer::doGetNextFrame()

-> MPEGVideoStreamFramer::continueReadProcessing()

->FramedSource::afterGetting()

->MultiFramedRTPSink::afterGettingFrame()

->MultiFramedRTPSink::afterGettingFrame1()

->MultiFramedRTPSing::sendPacketIfNessary()

至此，一个RTP数据包就被发送出去了，接下来计算下一RTP数据包的发送时间，并将MultiFramedRTPSInk::sendNext()函数句柄传给任务调度器的待处理任务队列，当服务器doEventLoop主循环查找到该句柄时，会执行该函数，并再次调用MultiFramedRTPSink::buildAndSendPacket()发送新的RTP数据包，这样服务器就可以循环地发送视音频媒体数据了。

这个函数调用过程只是一个粗略的过程，其中的很多函数都是纯虚函数，实际调用时，会根据不同的子类实现，进行不同的调用处理。

(6) TEARDOWN请求处理

服务器端doEventLoop()主循环通过找到与之对应的处理句柄incomingRequestHandler，对报文进行解析，解析结果TEARDOWN报文，则调用RTSPClientSession::handleCmd_TEARDOWN()成员回应客户端请求，结束会话。

5.3 基于live555架构的RTSP直播服务器的实现

在经过对live555的结构、请求处理过程以及媒体数据处理过程进行具体分析后，就可以对现有的live555进行修改、继承实现，从而实现H264编码的视频和AAC编码的音频的实时直播了，主要包括服务器平台的实现、H264实时视频源与AAC实时音频源的实现。Live555内部实现了H.264按RFC3984进行RTP封装，AAC按RFC3640进行RTP封装。

5.3.1 RTSP服务器平台的实现

Live555项目主要用于本地视音频媒体文件的点播，在其源代码中实现了对部分编码格式的视频和音频的简单的点播功能，本论文要求实现实时视音频数据的直播，需要对项目源代码进行必要修改、继承和重新实现，本文实现了H264视频实时数据源H264RealTimeSource和AAC音频实时数据源AACRealTimeSource，具体实现过程见后续小节，这里先说明借助H264和AAC实时数据源实现RTSP服务器平台的具体过程。

(1) 创建任务调度器和运行环境

首先是创建任务调度器对像scheduler，再由scheduler及具体的环境创建运行环境实例liveEnv。Live555框架是事件驱动的，所有的事件都通过任务调度器进行统一管理，可以说任务调度器scheduler是整个服务器的发动机，它会从任务队列里找到待处理事件，进而执行。liveEnv是服务器的运行环境，重要性不必多说。

(2) 设置H264视频和AAC音频的通信参数

服务器在发送视频和音频时通常使用两个不同RTP的会话，而且某些通信参数和编码格式相关，因此，H264视频和AAC音频的通信参数需要分开设置。

在本章5.2.1中已经说明本文内容需要进行组播，因此，需要借助groupsock类库的

函数 chooseRandomIPv4SSMAddress()生成一个组播地址，然后分配两个连续的端口用于 RTP 和 RTCP 报文传输（偶数端口给 RTP），需要注意的是要将端口由主机字节形式转换成网络字节形式；再根据组播地址和端口号，分别创建 RTP 和 RTCP 的 Groupsock 实例发 vRtpGrpSock、vRtcpGrpSock，并根据 vRtpGrpSock 创建 H264 视频的 RTPSink 实例，用于 H264 视频的发送，代码如下

```
RTPSink* vSink = H264VideoRTPSink::createNew(*liveEnv, &vRtpGroupsock, 96)
```

其中 96 为 RTP 的负载类型，表示 RTP 负载为 H264 视频数据；而后，还需要为 vSink 创建 RTCPInstance 实例 vRTCP，用于处理 RTCP 报文的发送和接收。

AAC 音频的通信有关参数设置类似，与 H264 使用相同的组播地址，但使用另一对不同连续的端口，依然给偶数端口分配给 RTP，根据组播地址和端口号，分别创建 RTP 和 RTCP 的 Groupsock 实例 aRtpGrpSock、aRtcpGrpSock，接下来 AAC 音频创建 RTPSink 实例，用于 AAC 音频的发送，创建代码如下。

```
RTPSink* aSink = MPEG4GenericRTPSink::createNew(*liveEnv,  &aRtpGrpSock,
        97, samplingFrequency, "audio",  "AAC-hbr",  configStr, channels)
```

其中 97 为 RTP 负载类型，表示类型为音频；samplingFrequency 为音频采样率；"audio"为 sdp 中媒体类型字符串；"AAC-hbr"为 AAC 根据 RFC3640 的 RTP 封装模式，表示一个 RTP 包中可以负载一个或多个 AAC 音频帧；channels 为音频的声道数；configStr 为 AAC 的帧配置信息，由音频的采样率索引 samplingFrequencyIndex（见表 2-4）和 AAC 音频的规格 profileIndex（或者说是对象类型索引，见表 2-5）计算得到。计算方法如下：

```
unsigned char audioSpecificConfig[2];
 u_int8_t const audioObjectType = profileIndex + 1;
 audioSpecificConfig[0] = (audioObjectType<<3) | (samplingFrequencyIndex>>1);
 audioSpecificConfig[1]=(samplingFrequencyIndex<<7)|(channels<<3);
 sprintf(configStr, "%02X%02x", audioSpecificConfig[0], audioSpecificConfig[1]);
```

(3) 创建 RTSPServer 实例并添加媒体会话

```
RTSPServer* rtspServer = RTSPServer::createNew(*liveEnv,tcpPort)
```

创建 RTSPServer 实例，其内部会创建一个 Socket 连接，并侦听 tcpPort 端口（默认使用 554），以接收并响应客户端的请求。之后创建服务器媒体会话

```
ServerMediaSession*  sms  =  ServerMediaSession::createNew(*liveEnv,  streamName,
"H264-AAC","Session streamed by \"H264-AAC\"",True );
```

其中，streamName 为自定义的流名称，会被组合到 RTSP 的 URL 中，服务器收到客户端的 URL 请求时，正是根据该名称找到相应的会话；TRUE 代表 SSM(Source Specific Multicast)。接下来为 sms 添加视音频子会话，再将服务器媒体会话加入到 RTSPServer 对象 rtspServer 中，如下代码片断所示。

```
sms->addSubsession(PassiveServerMediaSubsession::createNew(*vSink, rtcp));
```

```
sms->addSubsession(PassiveServerMediaSubsession::createNew(*aSink, auRtcp));
rtspServer->addServerMediaSession(sms);
```

PassiveServerMediaSubsession 为被动媒体子会话，当服务器启动时，即使没有客户端请求，服务器也会不断地向外发送视音频媒体数据。

(4) 创建视音频实时数据源

这里就要用到前面提到 H264 和 AAC 的实时数据源：H264RealTimeSource 和 AACRealTimeSource。编码生成的 H264 帧和 AAC 帧会分别存入 H264 队列和 AAC 队列，H264RealTimeSource 根据 H264 帧队列创建实时视频源，AACRealTimeSource 根据 AAC 帧队列创建实时音频源，分别传给 vSink 和 aSink 的 startPlaying()成员函数，开始数据处理和发送。

(5) 进入 doEventLoop()主循环

进入主循环后，服务器程序就会不断地查询任务队列，找到第一个待执行的延时任务并执行，以响应客户端的 RTSP 请求和处理 RTP/RTCP 数据包的发送任务。

5.3.2 H264 视频数据源的设计与实现

H264 视频数据源的作用主要是获取实时编码生成的 H264 帧，传给 live555，经过内部一系列的分析处理以后，发送到网络。本文通过对 live555 源代码进行继承和重写，实现了 H264 视频实时数据源类 H264RealTimeSource，配合本文第四章 H264 视频的采集、拼合、编码处理工作，接收实时 H264 数据帧，再传给 H264VideoStreamDiscreteFramer 进行离散完全帧的简单分析，然后交给 H264FUAFragmenter 进行分片处理，并按 RFC3984 进行数据封装，最后交给 H264VideoRTPSink 完成数据发送，进行整体的数据流向如图 5-4 所示。

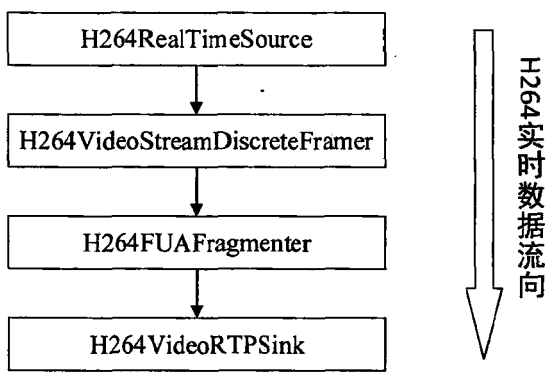


图 5-4　H264 数据流向

Fig. 5-4 The direction of H.264 data being past

H264RealTimeSource 是服务器获取实时 H264 视频帧的窗口，H264RealTimeSource 类继承了 FrameSource，其部分成员如表 5-2 所示。当 live555 向 FramedSource 请求 H264 数据时，就会转到 H264RealTimeSource 中相应的成员函数，由该成员函数获取数据，并传给 live555。

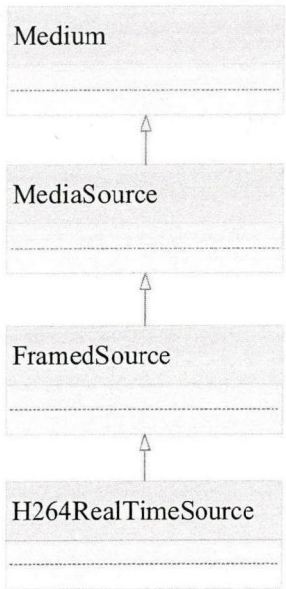


图 5-5　H264RealTimeSource 继承关系

Fig. 5-5 The class diagram of H264RealTimeSource

(1) 创建对象和初始化

秉承 live555 项目代码的书写风格，一般类的构造函数设置为私有，避免类外创建对象，对象的创建使用静态成员函数 createNew()完成，需要传入 5.3.1 创建的运行环境对象、H264 帧队列、帧率及临界区对象，之前调用构造函数执行成员变量的初始化处理。

(2) 从 H264 帧队列中取帧

前面已经提到了，live555 过一系列内部函数调用来取帧，通过 FramedSource::GetNextFrame()取帧时会调用子类 H264RealTimeSource 继承实现的 doGetNextFrame()成员函数来取数据，doGetNextFrame()再调用 deliverFrame()从队列中取数据，这样做的意义在于由于不能保证每次取帧时，队列中都一定有帧，当队列中没有帧时，可以将 deliverFrame0()句柄连同当前对象放入延时队列，以便延时一段时间后下次查询取帧。

H264 编码线程不断地编码并将编码生成的 H264 帧放入 H264 队列，如果直播服务器线程没有开启、直播发送没有进行的话，显然会导致内存溢出，因此当取帧发送未开始，开始取帧标识 fStartToGetFrame 为 false 时，H264 队列是不放入数据的。当 deliverFrame()函数被调用，取帧开始时，取帧标识 fStartToGetFrame 置为 true，那么编码线程就会不断地生成 H264 帧并放入队列。接下来进入临界区 fCS，查询 H264 帧队列，如果队列中无帧，需要创建查询取帧的延迟任务，并返回；如果队列中有帧，那就从 H264 帧的首部取出一帧 H264 数据，检查是否有起始码（00 00 00 01），有的话去掉，然后将数据拷贝到 FramedSource::fTo 成员，数据长度传给 FramedSource::fFrameSize，获取当前 GTM 时间传给 FramedSource:: fPresentationTime，用于 RTP 时间戳的生成；之

后，释放当前 H264 帧所占的内存空间，并从 H264 队列中删除，防止内存溢出，离开临界区 fCS，调用 FramedSource::afterGetting()进行后续处理。

需要注意的是，这里会用取到帧的时间来计算生成 RTP 时间，而不是帧生成的时间。

表 5-2　H264RealTimeSource 部分成员

Table 5-2 The main members of H264RealTimeSource

	成员	描述
变量	fAVPktList	CList<AVPacket*>，保存 H264 帧的队列
	fCS	CRITICAL_SECTION*，临界区，用于编码线程与直播线程互斥读写互斥
	fMicroSecsPerFrame	int，一帧的时间，单位毫秒
	frameNum	int，已读取的帧数
	fStartToGetFrame	bool，开始取帧标识
	fSourceStopped	bool，结束数据源
函数	createNew()	H264RealTimeStreamSource*，静态函数，用于创建新对象
	H264RealTimeSource()	构造函数，私有，执行一些初始化工作
	doGetNextFrame()	FramedSource 类纯虚函数的实现，调用 deliverFrame()以取帧
	deliverFrame0()	静态成员，让当前对象调用 deliverFrame()以从队列中取帧，同时也用于任务调度
	deliverFrame()	用于从队列中取帧

(3) 创建延时取帧任务

当 deliverFrame()查询 H264 队列中无数据帧时，就需要创建查询延时任务，将 deliverFrame0 句柄与当前对象放入延时任务队列，并返回，否则数据异常会导致程序崩溃，延时任务创建如下：

nextTask()=envir().taskScheduler().scheduleDelayedTask(fMicroSecsToDelay,(TaskFunc*)deliverFrame0, this);

此时还需要设置延时时间 fMicroSecsToDelay，表示延时 fMicroSecsToDelay 微秒后，主循环会查询到该处理任务，并执行。这个延时时间 fMicroSecsToDelay 设置不宜过大，也不宜过小；过大导致延时时间过长，取帧发送速度过慢，而编码线程编码速度相对稳定，造成 H264 队列堆叠；过小虽然能很快地进行下次查询，但查询速度过快，会给 CPU 带来额外的负担。可以根据视频的帧率，设置成前后两帧采集的时间差，与编码线程的编码速度保持相对一致。

这样主循环从任务列表中查询到 deliverFrame0()句柄与传入的对象时，就能找到 H264 数据源再次查询 H264 帧列队了，这样就保证了即使 H264 队列中暂时无数据也能继续运行了。

5.3.3 AAC音频数据源的设计与实现

H264RealTimeSource的实现对于AAC实时音频数据源AACRealTimeSource的实现有很大的参考意义，它们的处流程类似，不同的是一些具体参数的设置，此外，AACRealTimeSource从AAC队列中取到AAC帧后不必经过其他过虑器，可以直接交给MPEG4GenericRTPSink进行分析、封装、发送。

表5-3 AACRealTimeSource部分主要成员

Table 5-3 The main members of AACRealTimeSource

	成员	描述
变量	fAVPktList	CList<AVPacket*>，保存AAC帧的队列
	fCS	CRITICAL_SECTION*，临界区，用于编码线程与直播线程互斥读写互斥
	fMicroSecsPerFrame	int，一帧的时间，单位毫秒
	frameNum	int，已读取的帧数
	fStartToGetFrame	bool，开始取帧标识
	fSourceStopped	bool，结束数据源
	fSamplingFrequency	int，音频采样率
	fChannels	int，音频声道数
	fConfigStr	char*，AAC帧配置信息
函数	createNew()	AACRealTimeSource*，静态函数，用于创建新对象
	H264RealTimeSource()	构造函数，私有，执行一些初始化工作
	doGetNextFrame()	FramedSource类纯虚函数的实现，调用deliverFrame()以取帧
	deliverFrame0()	静态成员，让当前对象调用deliverFrame()以从队列中取帧，同时也用于任务调度
	deliverFrame()	用于从队列中取帧

构造函数同样为私有，由createNew()新建对象，需要传入运行环境对象、AAC帧队列、AAC规格索引、采样率、声道数及临界区等。从AAC帖队列中取帧的处理也H264RealTimeSource类似，调用deliverFrame()开始取帧，设置取帧标识为true，让AAC编码线程将生成的AAC帧放入AAC帧队列。进入临界区，查询AAC帧队列时，若无数据帧，则将deliverFrame0句柄与当前对象放入延时任务列表，设置延时时间为两音频帧间的时间间隔左右；若有数据，则检查取到的帧数据是否有ADTS头，有就去掉，再将数据拷贝到FramedSource::fTo，设置好FramedSource::fPresentationTime、FramedSource::fFrameSize；之后离开临界区，释放内存，删除AAC帧队列头部元素，进行后续处理即可。

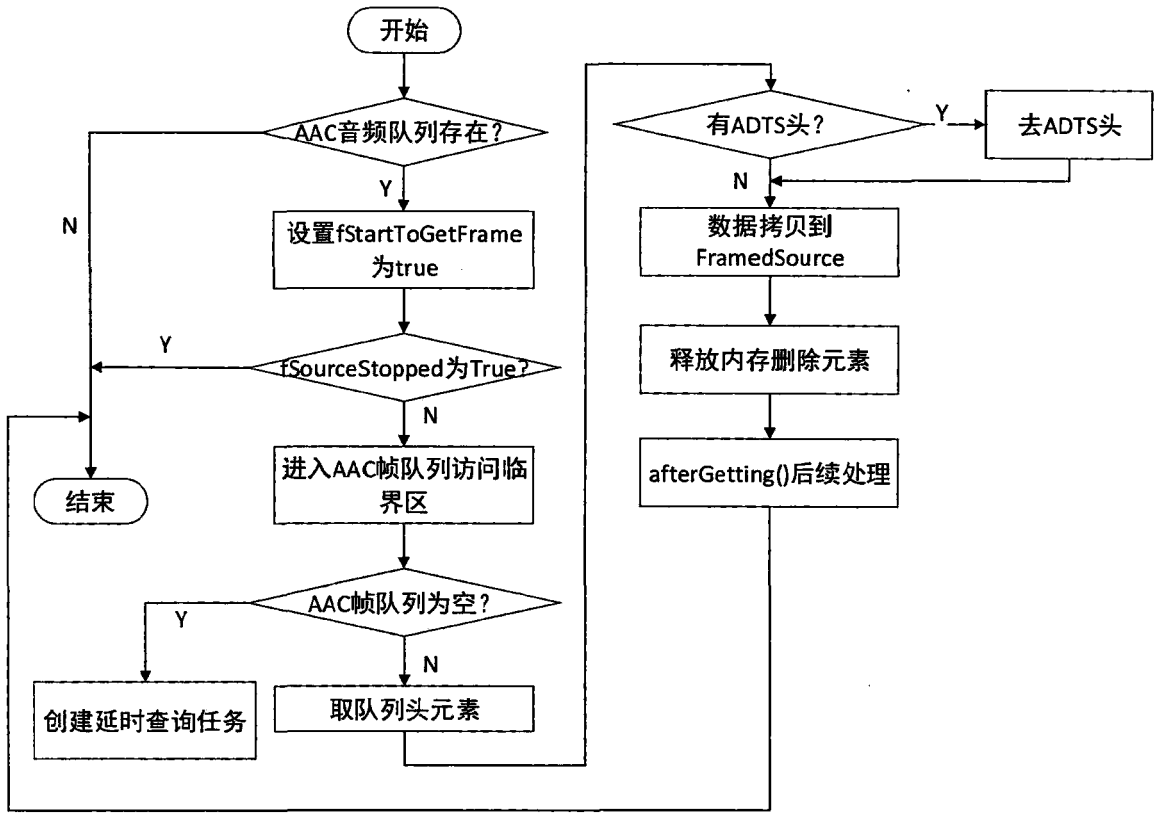


图 5-6　deliverFrame 一次取帧处理流程图

Fig. 5-6 The flow diagram of getting frame with 'deliverFrame' once

5.4 发送端媒体间同步的控制

5.4.1　发送端媒体间同步的必要性

多媒体同步指的是维护一个或多个媒体流时间约束关系的过程[56]，通常可以分为媒体内同步和媒体间同步[57]。媒体内同步指维持一个媒体流内部各单元的时间关系，媒体间同步指维持多个媒体流之间的时间关系[58]。

在网络环境中，多媒体数据从发送端到接收端通常会受延时抖动、发送端与接收端时钟偏差[59]、数据丢失、网络传输条件改变等因素的影响，在接收端必然会导致不同步现象，造成视频和音频间明显的时间差异，严重地影响观看效果。因此，当接收端接收到流媒体数据后，需要利用有关同步技术进行同步处理。

但是，如果多媒体数据在发送以前就已经不同，那么，接收端无论怎么控制，都很难达到较好的同步效果。因而，为了使视音频在播放时能有较好的同步效果，不仅要对接收端加以控制，还需要对发送端加以控制[60]。

5.4.2 媒体间同步控制

本文系统需要对视音频实时直播，这种情况下音频与视频同等重要，视频和音频的采集、编码、发送又在不同的线程，如果任由其自由运行，必然会导致视频和音频处理步调不一致，编码帧堆叠过多，造成没能及时发送出去，进而影响接收端的接收与同步处理。因此，在源端对视频和音频编码帧进行一定程度的同步控制是很有必要的，本文进行的非精确同步控制，主要包括以下两个方面。

(1) 稳定采集速度

视频与音频的采集工作通常分别在不同的线程或进程独立进行，互不干扰，稳定各自的采集速度，让其保持与编码设定的速度（由帧率或采样率决定）持平，一方面可以减少媒体内的抖动，避免画面或声音不流畅，另一方面通过这种控制，使得采集视频帧和音频帧的时候在单位时间内能采集到合适的帧数，不多采不少采，多采或少集在根源上就已经导致视频和音频数据在时间上的不对等，自然会影响到同步工作。稳定采集速度的控制在本文4.2小节中有说明。

(2) 适当提高访问速度

采集到的视频和音频帧，经过有关处理、编码后存入各自的帧队列里，视频和音频数据源分别从对应的帧队列中取帧用于发送，存入与读取会有一个时间差，实验显示这个时间差很小，对同步的影响不大。但读取发送时，是将读取到编码帧的时间用于计算生成RTP时间戳，因此，如果读取发送太慢，造成编码帧队列的堆叠，就会导致最新的帧不能及时打包发送，由于打包时时间戳已经失准，接收端的同步必然会受到不良影响。

因此，要适当提高实时数据源访问编码帧的速度。取到帧后的打包与发送是非常快的，那么，要提高访问速度，延时任务延时时间的设置就比较重要了，它直接影响数据源对帧队列的访问速度，在数据源实现章节也说明了延时时间的设置，不宜过长也不宜过短，最好是一帧的时长（或说是两帧的时间间隔）。

本文的同步控制是非精确的，是以视音频的实时性做为保障，没有依据时间戳进行精细控制，实际同步效果会有所波动，但经过多次实验测试，同步效果良好，可以接受。

5.5 本章小结

本章主要实现了基于 live555 框架的 RTSP 视音频直播服务器，首先具体分析了live555的协议关系、内容结构以及其请求处理与数据处理发送的过程，接下来根据这些分析结果，实现了H264与AAC的实时数据源和RTSP服务器平台，最后说明了视音频媒体间同步控制的必要性和方法。下一章将对系统进行相关测试。

第六章　系统测试

前面两章分别实现了视音频直播系统的两个重要方面：视音频编码数据的前端获取与RTSP直播服务器，接下来的章节将对直播系统进行相关测试，以检查其功能完整性、运行稳定性，检查其是否达到设计目标。

6.1 测试环境

6.1.1　网络环境

系统运行的目标网络是某专用网络，数据通信使用的是专线，测试时暂时没有条件使用该专用网络，因此，暂时使用若干台TP-LINK TL-SF1008型8口百兆交换机交换机搭建一个临时的局域网，模拟专用网络，用于测试。网段为192.168.1.0，子网掩码为255.255.255.0。

6.1.2　设备与配置

系统环境的搭建需要多种设备，主要包括视音频前端采集设备、直播服务器主器、若干客户端，其具体说明如下：

(1) 视音频前端采集设备

视频采集设备：三台海康威视DS-2CD2210D-I5网络摄像机、一台海康威视DS-2CD3210D-I3网络摄像机。使用海康威视在线设备侦测工具SADP(V2.0)设置四台摄像机IP地址分别为192.168.1.64、192.168.1.65、192.168.1.66、192.168.1.67，子网掩码255.255.255.0，端口号8000。

音频采集设备：三台得胜（TAKSTAR）MS-148台式电容麦克风、一台HNM-SM2600混音器。

(2) 直播服务器

一台通用PC机，操作系统为64位Windows7，CPU为Intel i7-3770K/3.5GHz/四核，内存4G，硬盘1T，显卡芯片NVIDIA GeForce GTX 650。安装Microsoft Visual 2010开发平台、Wireshark-1.11.0抓包工具、海康威视在线设备侦测工具SADP(V2.0)。主机IP地址为192.168.1.234，子网掩码255.255.255.0。

(3) 客户端

三台普通PC机，操作系统为Windows7，CPU为Intel i5-3470/3.2GHz/双核，内存4G，硬盘750G，集成显卡。安装Wireshark-1.11.0抓包工具、VCL Media Player2.1.5播放器。三台主机的IP分别为192.168.1.208、192.168.1.209、192.168.1.210，子网掩码为255.255.255.0。

6.2 系统测试

6.2.1 测试方案

对系统进行测试主要是测试系统功能的完整性和运动的稳定性，检查各模块能否正常运行，验证系统能否长时间工作不出现异常、不崩溃。主要测试内容包括以下几个方面：

(1) 系统能通过组播的方式发送由网络摄像机与麦克风采集处理得到的实时视音频数据。

(2) 系统的采集、多合一处理、编码、存储、直播等模块都能正常工作。

(3) 系统允许多台 PC 主机通过 VLC 或 ffplayer 播放器随时接收视音频数据。

(4) 系统能较长时间正常运行。

(5) 视音频同步性测试。

针对上述具体测试内容，提出对应的测试方案，具体如下：

a) 任选一台客户端主机，启动 wireshark 网络抓包工作，抓取网络数据，分析网络数据信息。

b) 启动服务器端，运行一段时间，通过存储的.mp4 文件，验证除直播以外的其他模块。

c) 在三台客户机上运行 vlc 播放器，接收数据，验证除存储以外的其他模块；并多次点击"stop"和"play"，验证服务器端的稳定性。

d) 服务器端连续运行 24 小时以后，验证其稳定性。

e) 在服务器端运行期间，不定期在网络摄像机视野范围内数数并出手指，检查视频和音频是否同步。

6.2.2 测试步骤与结果分析

1. 运行直播服务器，从三台客户机中任选一台，启动预装的 wireshark 抓取网络中的数据，设置过滤条件为"udp"，筛选出源地址为服务器主机 IP（192.168.1.6）的数据包，解析为"RTP"数据，如图 6-1 所示。可以看到，目的地址 192.168.1.234，为组播地址，通过数据包的类型 96、97，其中 96 为 H.264 视频数据包，97 为 AAC 音频数据包。

```
181 30.8807100 192.168.1.234  232.232.205.100  H264    1490 PT=DynamicRTP-Type-96, SSRC=0xD2A2AFCD, Seq=19745, Time=230353686 FU-A
182 30.8944760 192.168.1.234  232.232.205.100  H264    1490 PT=DynamicRTP-Type-96, SSRC=0xD2A2AFCD, Seq=19746, Time=230353686 FU-A
183 30.8946460 192.168.1.234  232.232.205.100  H264    1490 PT=DynamicRTP-Type-96, SSRC=0xD2A2AFCD, Seq=19747, Time=230353686 FU-A
184 30.8947940 192.168.1.234  232.232.205.100  H264    1490 PT=DynamicRTP-Type-96, SSRC=0xD2A2AFCD, Seq=19748, Time=230353686 FU-A
185 30.8949290 192.168.1.234  232.232.205.100  H264     459 PT=DynamicRTP-Type-96, SSRC=0xD2A2AFCD, Seq=19749, Time=230353686 FU-A
186 30.9296950 192.168.1.234  232.232.205.100  RTP      501 PT=DynamicRTP-Type-97, SSRC=0x48515CEE, Seq=19335, Time=334819831, Mark
187 30.9307350 192.168.1.234  232.232.205.100  RTP      503 PT=DynamicRTP-Type-97, SSRC=0x48515CEE, Seq=19336, Time=334819881, Mark
```

图 6-1 RTP 报文

Fig. 6-1 RTP message

2. 运行直播服务器一段时间后，检查在其根目录中检查是否有.mp4 媒体文件，如果有，则使用播放器打开该媒体文件，检测画面组合情况、清晰度、声音清晰度。

3. 在三台客户机中都打开 VLC2.1.5 播放器，点击菜单栏中的“媒体”，选择“打开网络串流”，在地址栏中输入”rtsp://192.168.1.234:8554/live”（live 为服务器端定义的流名称），在“显示更多选项”中设置缓冲时间 400ms，然后确定播放。三个客户机 VLC 画面流畅，设置人员在视野内活动，对比发现延迟为 1s 左右。

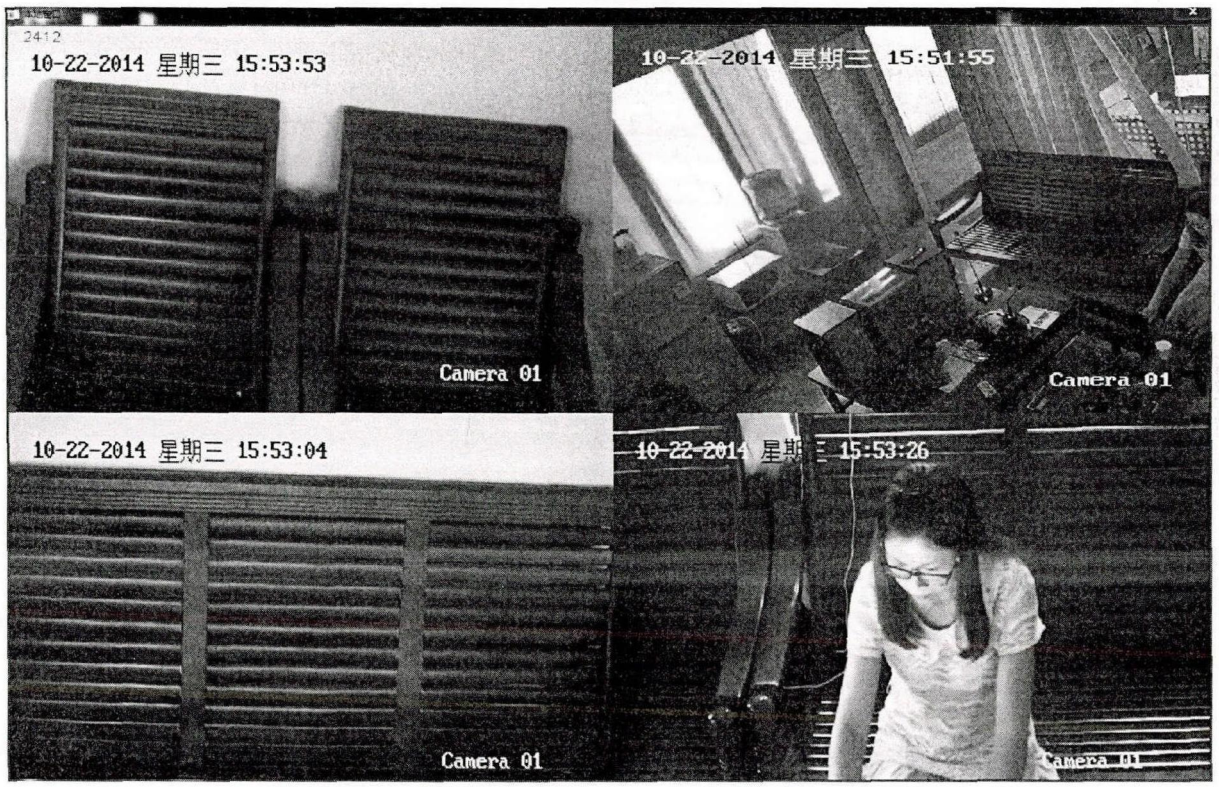


图 6-2 服务器端实时画面

Fig. 6-2 The reall-time image in server

注：四台网络摄像机的时间没有进行同步，因此四个子画面中显示的时间不同。

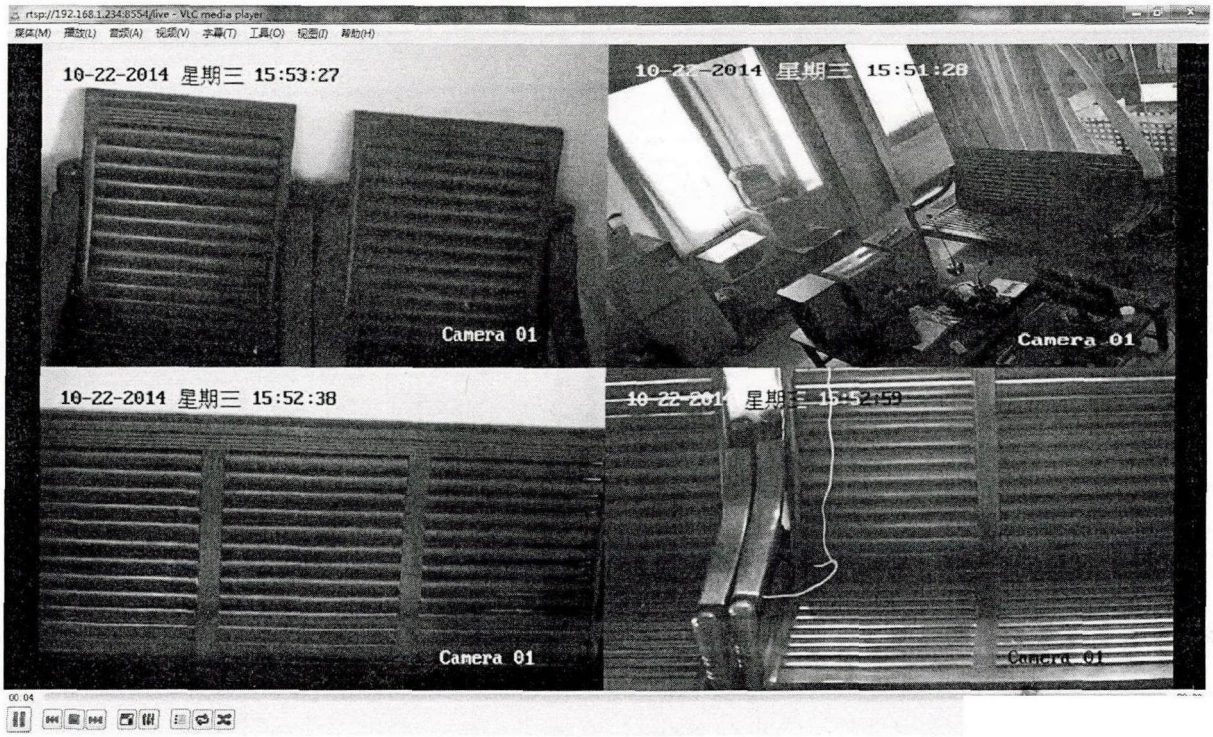


图 6-2 某客户机 VLC 接收播放的实时画面

Fig. 6-2 The real-time image received and presented with VLC in a client

在服务器端运行期间，在客户机的 VLC 播放器上，多次重复“stop”和“play”，每次”stop”再”play”后，播放器经过短暂黑屏后都能继续播放，且服务器端运行正常。

4. 让服务器端与客户机持续运行 24 小时以上，客户机 VLC 播放器都正常播放显示，服务器端无异常，说明服务器端有将好的稳定性。

5. 在服务器端与客户机持续运行 24 小时期间，不定期地摄像机视野范围内按秒读数，同时出对应手指，重复多次，在客户机 VLC 播放器观看显示画面与声音基本同步，有很少的差异。停止服务器后，打开存储的媒体文件，观看也发现多次按秒读数与出手指画面基本同步，无明显差异 。

6.3 本章小结

本章对前面两章实现的视音频直播服务器进行了测试，主要测试了直播服务器的功能完整性、运行的稳定性，以及视音频流的同步情况，本文设计并实现的多源视音频直播服务器能够进行数据组播、性能稳定、功能满足设计要求、视音频数据延迟较小且同步性良好。

第七章　总结与展望

本文内容来源于南宁博海软件科技公司的博海金天秤智能科技法庭系统，是该系统的重要组成部分，目前该系统首期项目已在广西百色市法院安装并验收。论文系统的主要的设计目的是多源合一，实现实时庭审过程的存储与共享。具体来说是为了将一个法庭中的法庭、被告、原告等多个席位场景的图像画面与声音能融合保存到一起，这样就可以很方便实现对庭审现场全局场景的记录，对于留存取证、场外观摩都有重要的意义。本文设计实现主要包括两部分：前端数据处理部分和直播服务器部分。

前端数据处理部分主要负责从多个网络摄像机中获取视频流，利用厂商提供的SDK解码得到视频图像，按照一定的排列方式，将从多个摄像机获取的实时图像经过缩放处理拼合到一张图像上，实现视频的多源合一，音频的处理主要是由多支电容式麦克风采集多路声音信号，经过混音器的混合，实现多路音频的多源合一，再由计算机完成这路混合音频的数字化采集，这样就由这一路视频一路音频完成了对合局场景的呈现；接下来，借助开源项目实现了视频的H264编码与音频的AAC编码，完成了对拼合图像与混合音频的压缩编码处理，为视音频的存储与网络直播奠定了基础；同时，前端也实现视音频的复用，将H264视频与AAC音频存储到一个MP4或MKV媒体文件中。

直播服务器接收视音频编码线程传递过来的编码数据，经过分析、封装后，发送到网络中，以便远程观摩庭审全局场景。本文研究分析了live555项目，对该项目进行了修改、继承和重写，以便其能接收实时编码生成的H264与AAC数据经过RTP封装后组播出去，同时借助该项目实现了RTSP直播服务器，最后对视音频的源端同步控制处理进行了分析与探讨。

由于时间和个人能力问题，只完成了直播服务器的重要部分，虽然经过测试，在功能完整性、运行稳定性、延时与同步性都有较好的效果，但有一些需要进一研究、改善的地方：

1) 增加对“rtp over tcp”的支持，本文采用UDP组播，在网络状况较好的专用网络效果还可以，有必要增加“rtp over tcp”以适应较差的网络。

2) 运行服务器发现CPU占用率较高，对主机配置要求较高，还需要进行检查分析，找到占用CPU资源较高的地方进行优化处理。

3) 完善各模块的控制，增加存储、直播等模块的开关控制，允许用户自由选择是开启或关闭存储、直播等功能，并保证重复操作不异常。

4) 关于图像的多流合一，几个线程通过互斥量互斥地访问画布图像，造成拼合效率不是很高，需要进一步研究更高效的访问机制。

参考文献

[1] 钟玉琢,向哲,沈洪.流媒体和视频服务器[M].北京: 清华大学出版杜, 2003.6
[2] 陈洪彬. 前沿流媒体实用手册[M]. 中国科学技术出版社, 2003
[3] Wu D, Hou YT, Zhu Wetal. Streaming video over the Internet:approaches and directions[J].IEEE Transacitions on Circuits and System for Video Technology.2001.11(3):282-300
[4] 流媒体. http://baike.baidu.com/view/794.htm
[5] 杨波. 流媒体系统的关键技术研究[D]. 北京邮电大学,2006.
[6] 韩东东. 基于MINA框架的RTSP移动流媒体代理服务设计与实现[D].西南交通大学, 2011:2-3
[7] 刘大红. 基于RTSP流媒体服务器的设计与实现[D]. 西安电子科技大学,2013:6-8
[8] Chang SF, Sikora T, Puri A.Overview of the MPEG-7 standard[J]. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY.2001.11(6).688-690.
[9] Burnett I, Van de Walle R, Hill K, et al. MPEG-21: Goals and Achievements[J].IEEE MULTIMEDIA.2003.10(4).60-61.
[10] Stephan Wenger. H.264/AVC Over IP[J]. IEEE Transactions on Circuits and System for Video Technology 7(13):645-65, 2003.
[11] Wiegand T, Sullivan GJ, Bjontegaard G, et al.Overview of the H.264/AVC video coding standard[J].IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY.2003.13(7).560-562.
[12] Ohm Jens-Rainer, Sullivan Gary J, Schwarz Heiko,et al.Comparison of the Coding Efficiency of Video Coding Standards-Including High Efficiency Video Coding (HEVC)[J].IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY.2012.22(12).1669-1672.
[13] AVS.AVS 标准工作简况与进展[EB/OL]. http://www.avs.org.cn. 2006.3.
[14] 曾金.嵌入式流媒体服务器的设计和实现[D].南京邮电大学.2011.03
[15] QuickTime Developers.QuickTime Streaming Server. http://www.apple.com/quicktime/streamingserver
[16] Windows MediaDevelopers.Windows Media. http://www.microsoft.com/windows/windowsmedia/default.msp.
[17] Helix Server Developers.Helix Server. http://www.realnetworks.com/products/media_delivery.html.
[18] 罗明宇,卢锡城,韩亚欣.Internet 多媒体实时传输技术[J].计算机工程与应用.2000.36(9).119
[19] 李燕灵,马瑞芳,左力.基于 RTP/RTCP 的实时视频数据传输模型及实现[J].微电子学与计算机.2005.22(8).138-140
[20] Wikipedia. http://en.wikipedia.org/wiki/Real_Data_Transport
[21] 姜浩然,徐林.基于 RTMP 的流媒体服务器的研究[J].计算机与数字工程.2011.39(10).104
[22] 周怡兵. 流媒体服务器软件的设计与实现[D]. 华中科技大学, 2011
[23] 方群,王敏,吉逸.基于 RTSP/RTP 的媒体点播服务器的设计与实现[J].计算机工程与设

计.2006.27(1).4-5
[24]I.Elsen, F.Hartung, U.Horn, et al. Streaming Technology in 3G Mobile Communication Systems[J].IEEE computer.2001.34(9):46-52.
[25]P.Frojdh, U.Horn, M.Kampmann,et al. Adaptive Streaming within the 3 GPP Packet-Switched Streaming Service[J].IEEE Network.2006.20(2):34-40.
[26]H.Schulzrinne, A.Rao, R.Lanphier. Real Time Streaming Protocol[S]. IETF RFC2326.1999
[27]张丽.流媒体技术大全[M].中国青年出版社.2001.11
[28]H.Schulzrinne, S.Casner, R.Frederick, et al.RTP:A Transport Protocol for Real-Time Applications.IETE RFC3550.2003.
[29]Thomas Wiegand,Gary J Sullivan,Gisle B Jontegaard.Overview of the H.264/AVC Video Coding Standard[J].IEEE Transactions on Circuits and Systems for Video Technology,2003(7):560-576.
[30]毕厚杰. 新一代视频压缩标准 H.264/AVC[M]. 北京：人民邮电出版社.2005
[31]S.Wenger, M.M.Hannuksela, T.Stockhammer, et al. RTP Payload Format for H.264 Video[S]. IETE RFC3984.2005
[32]ISO/IEC 13818-7:2005.33-34
[33]http://wiki.multimedia.cx/index.php?title=MPEG-4_Audio
[34] M.Handley, V.Jacobson. SDP: Session Description Protocol[S]. IETE RFC2327.1998.
[35]J.van der Meer, D.Mackie, V.Swaminathan, et al. RTP Payload Format for Transport of MPEG-4 Elementary Streams[S]. IETE RFC3640.2003
[36]郭晓强，付光涛，李小雨.AAC 音频压缩编码标准的 ADTS 与 LATM 格式分析[J]. 现代电视技术, 2008, 1: 140-142
[37]吴功宜. 计算机网络[M].北京：清华大学出版社, 2011: 83-85
[38]Gary Bradski, Adrian Kaehler. Leaning OpenCV[M]. O'Reilly, 2009: 159: 149-150
[39]周静. 基于视觉注意机制的图像检索方法研究[D].华南理工大学，2012
[40]林润杰. 一个基于 RTSP 的移动视频监控系统[D]. 华南理工大学.2012
[41]孙泉. 支持 H264 的实时流媒体服务器的设计与实现[D].北京邮电大学,2010:11-13
[42]望重. 嵌入式网络视频监控系统的研究与实现[D]. 江苏大学, 2010: 33-35
[43]王瑞.基于 H.264 嵌入式视频服务器的研究[D]. 中国地质大学，2013
[44]郝瑞林.H.264 视频编码器的研究与分析[J]. 北京:北京邮电大学网络技术研究院.
[45]路锦正.MPEG-4/H.264 视频编解码工程实践[M].北京:电子工业出版社,2011。
[46]FFmpeg project homepage[EB/OL].http://ffmpeg.org/about.html,2009-05-07/2011-11-16.
[47]邱振光，朱秀昌. MP4 复用器的设计与实现[J]. 电视技术，2011,35(9):34-39
[48]张勇波,宋晓丽.FLV 文件解析及其在网站中的应用[J].计算机与现代化,2011,8:124-125.
[49]田润澜,肖卫华,王健平.Mastroska 在机载视频记录系统中的应用[J].长春工业大学学报,2010,31(2):117-118
[50]刘畅棂，包杰，王宁国. 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(12):38-40
[51]冯进伟. H.264/SVC 流媒体系统优化与实现[D]. 北京邮电大学，2012:40-44
[52]王宝珂. 基于 ARM11 的嵌入式视频监控系统终端的设计[D]. 南京理工大学，2012:56-58

[53]陈锋锋. 基于RTSP的流媒体传输系统的应用开发[D]. 南京邮电大学，2013:17-20
[54]live555.http://www.live555.com/liveMedia/doxygen/html/classBasicUsageEnvironment.html
[55]张雪,董永强.支持IPv4/IPv6的RTSP流媒体应用代理的设计与实现[J].计算机科学,2006,33(3): 140-4.
[56]崔莉, 王敏, 吉逸. 流媒体同步机制的研究[M]. 计算机应用研究, 2005, 22(1): 73~75
[57]葛双全, 李芬. 实时多媒体流同步机制的研究. 电脑与信息技术, 2006, 14(4): 5~8
[58]杨蓓.流媒体系统中音视频同步机制的设计与实现[D].华中科技大学
[59]吴炜，常义林，罗忠。一种新的媒体同步反馈控制算法[J].西安电子科技大学学报，2006,33(3)，359-360
[60]齐成明.音视频同步问题的研究与实现[D].哈尔滨工业大学，2009.

致谢

时光荏苒，须臾之间，二年半的研究生生活即将谢幕，至此，我也即将告别校园，走向社会。校园生活总是让人难以忘怀，这最后二年多的校园生活也将是我最美好的回忆，她丰富了我的学识、磨砺了我的品性，使我能更加自信地面对将来的生活。值此搁笔之际，我想特别感谢一下这几年帮助、陪伴我的人。

首先，由衷地感谢我的导师莫林教授。莫老师和蔼可亲，平易近人，有热情有激情，能充分地调动学生工作与学习的积极性；同时，老师知识渊博、治学严谨，为我树立了很好的学习榜样。无论是在生活上，还是在工作和学习上，莫老师给我的帮助和指导都让我受益良多，在此谨向莫老师表示最诚挚的敬意和感谢。

另外，我想感谢我的诸位师兄曾繁璞、孙学斌、林文彪等，感谢你们在生活与学习的帮助与指导，感谢各位师兄不厌其烦地为我解疑答惑。同时，也感谢我的同门褚立超、师弟罗挺。感谢实习期间，公司同事李红英、梁清霞、覃方等的帮助。

特别要感谢我的家人，你们一如继往的包容、无微不至的关怀和照顾，你们支持与鼓励，是我不断前进的动力；感谢我的女友李超颖，感谢你的陪伴。

最后，由衷地感谢在百忙之中为我评审论文和参与答辩的各位专家、教授！

攻读学位期间发表论文情况

无