分类号：TP39

10710-2014124083

长安大学

硕 士 学 位 论 文

基于 FFmpeg 的高清实时直播系统设计与实现

席文强

导师姓名职称	张卫钢 教授		
申请学位级别	工学硕士	学科专业名称	智能交通与信息系统工程
论文提交日期	2017 年 5 月 5 日	论文答辩日期	2017 年 5 月 24 日
学位授予单位	长安大学		

Design and Implementation of High Definition Real-time Live System based on FFmpeg

A Thesis Submitted for the Degree of Master

Candidate: Xi wenqiang

Supervisor:Prof.Zhang weigang

Chang'an University, Xi'an, China

论文独创性声明

本人声明：本人所呈交的学位论文是在导师的指导下，独立进行研究工作所取得的成果。除论文中已经注明引用的内容外，对论文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本论文中不包含任何未加明确注明的其他个人或集体已经公开发表的成果。

本声明的法律责任由本人承担。

论文作者签名：[signature] 2017年 6月 3 日

论文知识产权权属声明

本人在导师指导下所完成的论文及相关的职务作品，知识产权归属学校。学校享有以任何方式发表、复制、公开阅览、借阅以及申请专利等权利。本人离校后发表或使用学位论文或与该论文直接相关的学术论文或成果时，署名单位仍然为长安大学。

（保密的论文在解密后应遵守此规定）

论文作者签名：[signature] 2017 年 6 月 3 日

导 师 签 名：[signature] 2017 年 6月3 日

摘　要

目前，我们生活中常见的直播系统大多采用流媒体数据逐块推流的设计，并且在服务器端对推流数据进行二次加工处理，此类直播系统的直播延时往往比较高，同时，由于采用了图像质量较低的压缩编码算法，因此其直播效果饱受诟病。为了克服此类直播系统出现的问题，本文设计实现了一套高清低延时的直播系统，在面对例如抢险指挥、实况直播、实时会议等对清晰度与实时性要求较高的应用场合有着重要作用。

论文首先对高清实时直播系统所采用的音视频处理技术以及流媒体传输协议做了简单分析与介绍，接着针对现有直播系统中存在的问题与不足展开研究，通过对 FFmpeg 音视频处理流程进行优化改造，设计并实现了高清实时直播数据采集、视频颜色空间转换、视频解码显示、音频数据重采样、多线程音视频高清编码、流媒体数据网络传输以及直播数据服务器转发等功能。同时在实现过程中对于现有直播系统所存在音视频质量不高，网络延时较大等问题做了以下改进工作。

1. 在设备信号采集过程中，通过对采集速率的控制使音视频观感与系统处理效率达到最优平衡。并且根据人眼对亮度信息较敏感的特性，对视频数据做了 YUV 颜色空间转换。

2. 设计实现了音视频解码数据缓冲区，通过临界区资源与读取缓冲区完成了多线程间同步以及视频数据的 SDL 显示，提高了音视频编解码处理过程的流畅度与稳定性。

3. 采用高清 H.264 视频编码与 AAC 高级音频编码技术，提高了直播音视频质量并减少了网络带宽压力，同时对音频的杂音问题做了重采样处理，消除了解码与采集过程中加入的杂音噪声。

4. 采用时间戳同步技术对音视频数据进行同步，并且基于 RTMP 流媒体传输协议，通过文件流写入的方式对直播数据进行逐帧网络传输，极大地降低了数据发送延时。

5. 基于 Nginx 搭建了 RTMP 流媒体服务器，将收到的直播数据立即转发到客户端，不仅提高了直播服务器面对高并发请求的处理能力，还降低了系统的网络延时。并且使用户在 Flash 网页便可以收看到直播画面，提高了客户端部署的便捷性。

最后，在对直播系统的各项功能测试与输出音视频质量评测之后表明，本文所设计实现的高清实时直播系统功能正确，运行流畅，直播画面清晰，且保持较低的直播延时。

关键词：高清实时直播，音视频处理，FFmpeg，H.264，AAC，Nginx，RTMP

Abstract

At present, the common live broadcast system in our life is mostly used streaming media data block-by-block design, and do the secondary processing for the flow data in the server side, the live delay of such live broadcast system is often relatively high, at the same time, its live effect has been criticized because of using a low quality image compression coding algorithm. In order to overcome the problems of such live system, this paper designed and realized a set of high-definition low-latency live broadcast system, it is a important role in high requirement applications for clarity and real-time such as emergency command, live broadcast, real-time meetings.

The paper analyzed and introduced the audio and video processing technology and the streaming media transmission protocol used in HD real-time live broadcast system at first, and then, studied existing problems and shortcomings in live broadcast system, by optimizing and reforming FFmpeg audio and video processing process, designed and realized real-time live data acquisition, video color space conversion, video decoding display, audio data re-sampling, multithreading audio and video HD encoding, streaming media data transmission and live data server forwarding and other functions. The following improvements proceeded at the same time in the realization process for the problems that the low quality audio and video, larger network latency and other issues in existing live broadcast system.

1. In the process of equipment signal acquisition, to achieve the best balance of audio and video perception and system processing efficiency by controlling of thdacquisition rate. And the YUV color space conversion did for video data according to more sensitive characteristics of human eye on the brightness of the information.

2. Design and realized audio and video decoding data buffers, the multithread synchronization and SDL display of video data are completed through the critical area resource and the read buffer, which improved fluency and stability of audio and video codec processing.

3. By using high-definition H.264 video encoding and AAC advanced audio coding technology, improved the quality of live audio and video and reduce the network bandwidth pressure, done a re-sampling process for audio noise problems at the same time, eliminated

the noise in the decoding and acquisition process.

4. Timestamp synchronization technology is used to synchronize audio and video data, by writing the file stream to transmit live data frame by frame network based on RTMP streaming media transmission protocol, it greatly reduced delay of the data transmission.

5. Build RTMP streaming media server Based on Nginx, that will forward the received live data immediately to the client, it not only improves the processing capacity of live server in face of high concurrent requests, but also reduces the system's network latency. The users will also be able to watch the live screen on the Flash page, improved the convenience of client deployment.

At last, after the functional test and output audio and video quality evaluation of the live system, it is shown that designed and realized HD real-time live system in this paper is functioning correctly, running smoothly, the live picture is clear and the broadcast delay is kept low.

Key words: HD real-time live, Audio and video processing, FFmpeg, H.264, AAC, Nginx, RTMP

目　录

第一章 绪论

1.1 课题研究背景

认识世界，是改造世界的基础，而在改造世界过程中，知识经验的积累与信息资源的共享是促进人类社会稳步向前发展的主要途径。在早期的人类社会，知识的传递以语言、文字、绘画、手势等简单形式作为主要载体，其传播范围较窄、内容单一且及时性低下等因素制约社会发展。之后进入现代社会，知识积累的载体与传播平台随着科技的发展而愈发变得丰富。在现代社会中，互联网作为高效、即时、较广传播范围的交互平台，始于 1969 年美国的阿帕网，经过四十多年的高速发展，逐步成为推进经济发展与社会进步的重要力量。而多媒体作为一种丰富的信息表现载体，随着网络技术发展而产生多种变形，流媒体就是其在网络环境下的一种特殊衍生形式，因为拥有优良的网络适应性而越来越受到人们的关注，并在网络应用与社会生活中扮演着举足轻重的角色。

如今，网络直播已经发展成为流媒体技术一大重要应用领域，因具有内容丰富、信息获取便捷等诸多优点而成为生活娱乐与工作中不可缺少的一部分。根据外文文献记载，最早的网络直播出现在 1995 年的 7 月 17 日，当时来自苹果计算机公司的 David B.Pakeman 与企业家 Michael Dorf 在 7 月 17 日至 7 月 22 日推出了麦金托什纽约音乐节，此活动引来纽约市超过 15 家俱乐部的音频网络直播。1996 年 10 月 31 日，英国摇滚乐队 Caduseus 在英国威尔士，从晚上 11 点到 12 点通过网络播放了他们一小时的音乐会，这是第一个实时流媒体和同步直播视频，全球二十多个国家可以在线观看。

国内最早成气候的直播平台是 YY 直播，可以说是国内网络视频直播行业的奠基者，早在 2005 年，原来专注于即时网络语音通信的 YY 语音和专注于陌生人视频社交的 9158 开始发展 PC 端的直播聊天室，同时，六间房由视频网站转为 PC 端的秀场直播，成为早期直播平台的起步标志。2010 年，YY 直播经过商业转型发展成为虎牙直播，与此同时，受美国 Twitch TV 与 Justin.TV 分离而独立成立直播平台的影响，在随后几年中，国内各种投资纷纷涌入到直播这个行业内，从 2015 年第二季度开始，国内的直播行业驶入了发展的快车道，从 YY 直播、斗鱼直播，发展到映客直播，越来越多的商业巨头涌入到网络直播行业中。截至到 2016 年 5 月，国内直播上线的平台已经超过 130 多家，行业潜在经济规模超过了 520 亿人民币。并且，观看网络直播的国内用户达到了 3.14 亿，占全国网民总数的 51.7%，并且这个规模还在持续上涨中。

1.2 直播技术发展现状

尽管国内直播行业起步较晚，但是由于网络信息交互的及时性，以及国内巨额的行业投资加持，国内直播系统中的技术发展与国外接近一致。目前较为成熟的直播系统实现通常采用客户端加服务器模式，主要实现流程是由直播软件捕获并生成音视频数据，在对数据进行编码处理后传输到服务器，最后经过服务器推送到用户窗口进行播放。尽管技术逻辑都类似，但不同直播系统采用的音视频编码与流媒体服务器都不尽相同，因此产生较大的实现效果差异。

通常，画面质量是最直接的感官体验。直播软件所采集的原始音视频数据的观看质量最好，但是由于原始信号的信息量巨大，对网络传输带宽造成了极大的浪费，因此在对音视频传输前，往往要经过编码压缩处理。在音视频处理技术中，编码技术占据了重要的地位，在相同码率帧率情况下，采用优秀的编码器将会获得更高的视频质量。为了使视频编码统一规范，从 1988 年开始，国际电讯联盟（International Telecommunication Union）的视频编码专家组（VCEG）与国际标准化组织（International Standardization Organization）的运动图像专家组（MPEG）完成了 H.26x 和 MPEG-x 系列的视频编码标准，用于满足各种不同的应用场景需求[1]。其中 H.26x 标准主要用于实时网络视频通信，侧重于网络传输编码。而 MPEG 标准则主要用于视频的存储、广播、流媒体应用等，扩展性、灵活性较好。我国的音视频编解码技术标准工作于 2002 年 6 月开始，并于 2005 年完成了第一个自主知识产权的视频编码校准 AVS（Advanced Vusial System），其压缩效率比 MPEG-2 增加了一倍以上，其编码性能已经基本接近于 H.264[2]。

此外，合适的网络传输协议选择也是直播系统非常重要的一部分。通常情况下，由于 TCP 协议的数据包确认与超时重发等机制影响，会造成较为严重的网络延时，因此不适合作为实时流媒体传输的网络协议。目前较为主流的流媒体传输协议包括 RTMP 实时消息传输协议、RTSP 实时流传输协议、RTP 实时传输协议与 RTCP 实时控制协议等[3]。由于 RTP 协议自身的局限性，通常只作为一个基本协议框架，配合 RTCP 协议进行使用。RTSP 协议因其具有良好的网络实时传输特性而被广泛使用，但 RTSP 通常最少需要两个通道进行命令和数据传输，相比之下，RTMP 协议一般在 TCP 一个通道上传输命令和数据，以嵌入超文本传输协议的方式穿透防火墙而不被拦截，同时增加了 SSL 等安全功能，使流媒体数据传输的安全性进一步提高。在传输效率上，RTMP 使用可变大小的消息包头，在传输大块数据包时甚至取消消息头字段，因此传输效率非常高。

1.3 课题研究意义

尽管不同的直播系统大多采用类似的音视频编码标准与流媒体传输协议，可是由于技术差别造成了不同的实现效果，因此，目前大多数直播系统存在视频画质不清晰、直播实时性差、对高清流媒体非良好支持等诸多问题。此外，大多数直播平台因为对并发问题的规避与系统稳定性的偏重，采用了编码数据按块传输，此举大大提高了直播系统的延时性，造成直播画面与实际时间相差甚远的情况。与此同时，以 1080P、2K、4K 为主流的更高清晰度，分辨率的流媒体对直播系统编解码效率与网络传输实时性提出了更高的要求和挑战，主要体现在以下几个方面。

（1）系统传输的低延时可控性。在直播视频流过程中，若假设采集到的第一帧视频时间为 t，而用户收到并播放的第一帧视频时间为 d，则传输延时为 $\Delta t = d - t$。由于在一些特殊应用场合，例如抢险指挥、电视实况转播、医疗救助、视频会议等，需要严格把控直播系统的延时，此时就对直播系统的实时性与传输质量提出了较为严格的要求。

（2）音视频编码质量和效率。尽管计算机硬件以及移动终端的性能在逐年提升，但稳定的直播平台应具有更好的兼容适配性，新的画面质量对老旧设备提出了严峻的考验，此时应尽量降低系统的硬件门槛，优化画面采集效率和编解码算法流程，保证有更宽的适用性。

（3）高清、甚至超清的流媒体数据增加了直播带宽的要求，在网络条件非理想状况下，容易产生数据丢失和拥塞，并对流媒体服务器产生高负荷压力，因此传输前，在不影响观感质量的前提下，应保持稳定较高的压缩编码帧率。

（4）视频画面质量是直播系统最直观的评价，通常使用峰值信噪比 PSNR（Peak Signal Noise Ratio）等参数对其进行评定[4]。为了获得较高的图像质量，除了使用更高清的编解码标准外，还需要更大的带宽支持，另外，在高并发的网络使用环境下，多条流媒体竞争带宽资源时，直播系统的稳定性就会面临严重的挑战，客户端可能会出现视频画面卡顿，甚至丢失信号。

显然，以上几点都是关系到直播系统稳定性与质量的重要因素，也是未来直播技术发展的趋势所在。本文以解决现有直播系统实时性低、编解码效率较低和对高清音视频的非良好支持问题出发，以当前主流的 FFmpeg 音视频处理框架为依托，对音视频采集、解码、编码流程进行了优化工作，该框架具有移植性高、实现灵活与数据处理高效等特点，为直播系统的音视频处理提供了强大可靠的平台。设计实现了解码缓冲区技术，充

分挖掘平台的处理潜力，使直播系统的编解码流程更稳定高效。同时，利用多线程编程技术将音视频的采集、解码与编码分离，使直播系统可以进行高帧率和高分辨率的编解码工作。最后，在网络传输技术上，采用基于 Nginx 的高并发 RTMP 流媒体服务器，以逐帧传输的方式充分保障了直播系统的实时性与带宽利用率，为解决现有直播系统问题提供了有效的方法与途径，具有较高的经济价值和社会意义。

1.4 论文主要工作内容

本文针对现有直播系统所存在的问题并结合优秀的编解码标准与网络传输协议，设计实现了一种基于 RTMP 协议的实时高清直播系统，通过 FFmpeg 音视频处理技术将个人计算机的桌面视频信号与麦克风以及扬声器的音频信号进行采集，利用 H.264 与 AAC 编码技术将音视频数据压缩编码成网络传输特性较好的数据格式，最后经过 FLV 文件封装规范，将数据传输到流媒体服务器上，通过服务器转播功能，用户将在网页上浏览到直播内容。在直播系统设计实现后，对系统的各项功能进行测试，并给出相关结论与测试结果。

本文完成的主要工作如下：

1. 分析并研究直播系统的各项关键技术，结合现阶段直播系统实现中所存在的一些问题，设计出直播系统的主体框架与技术实现的逻辑思路，并对相关技术进行深层次的研究学习。

2. 完成了在 Windows 下对 FFmpeg 的编译与移植，根据系统需求对相关功能进行了裁剪，并对 VS2010 下的音视频开发环境进行了快速配置。

3. 经过对 FFmpeg 框架进行研究学习，掌握了音视频数据相关处理流程，并针对直播系统在编解码层次上做了诸多优化实现。

4. 对音视频采集速率进行了研究分析，并根据系统平衡性选择最优的采集速率，实现了在数据源头对流媒体质量与码率进行控制。

5. 在现有直播系统解码技术基础上，通过音视频解码缓冲区的设计，提高了音视频编解码的流畅度与稳定性。并且对音频的杂音问题进行了简单的重采样处理，消除了解码与采集过程中加入的杂音噪声。

6. 采用独立线程的设计思想，将音视频采集、解码、编码分为多线程同步运行，在线程间采用临界区资源进行线程同步，保证了数据的完整性。

7. 采用时间戳同步技术对音视频数据进行同步，并且基于 RTMP 流媒体传输协议，

通过文件流写入的方式对直播数据进行逐帧网络传输，极大地降低了数据发送延时。

8. 搭建基于 Nginx 的 RTMP 流媒体服务器，通过直播数据转发技术的实现，让用户在网页上就可以收看到直播画面，降低了使用门槛并提高了便捷性。

9. 在直播系统整体设计实现完成后，对各项功能进行了测试，并对编码出的视频质量进行了相关评价工作。

1.5 论文的组织结构

论文的组织结构如下：

第一章：绪论。首先对课题的研究背景进行了介绍，通过对直播技术发展现状的分析，简单阐述了直播系统实现中的相关技术。在课题研究意义中，对现有直播系统中的若干问题进行分析与说明，并提出更优的设计与实现方法。

第二章：音视频编码技术与 RTMP 传输协议简介。重点介绍了直播系统设计实现过程中所涉及到的视频颜色空间、H.264 视频编码标准、AAC 音频编码原理、FFmpeg 音视频处理技术以及 RTMP 流媒体传输协议。

第三章：基于 FFmpeg 高清实时直播系统总体设计。首先针对现有直播系统中存在的一些问题提出了几点功能性需求，同时对直播系统的非功能性需求做了分析，最后对系统的总体框架进行了设计，包括对直播软件的功能设计与直播服务器端的功能设计。

第四章：高清实时直播软件功能实现。首先对直播系统开发环境进行搭建，然后对直播软件中音视频数据采集、解码、编码等关键技术进行实现，并且在实现过程中对设计采用的新技术进行详细阐述，包括音视频数据采集速率控制、音视频解码缓冲区设计与实现、音频重采样实现、单独线程的编解码实现等。

第五章：高清实时直播服务器搭建与数据传输实现。首先对 Nginx 服务器框架进行研究介绍，根据研究结果对 RTMP 流媒体服务器进行搭建实现，最后利用时间戳技术对编码后的音视频数据进行了同步工作，并根据 FLV 封装格式与 RTMP 传输协议进行了逐帧音视频数据网络传输的实现，提高了直播系统实时性。此外，在传输过程中，利用对传输速率的控制减轻了流媒体服务器的负担，进一步提高了直播系统的稳定性。

第六章：高清实时直播系统测试。首先完成了对直播软件界面的设计与实现，通过对直播软件安装过程、用户登录与服务器联合测试等单元测试，并在服务器检测端查看转发情况，完成系统的总体测试。最后，对采集编码后的视频序列进行 MediaInfo 树状图信息验证、峰值信噪比 PSNR、时间与空间复杂度计算评价，并给出了评价结果。

第二章 音视频编码技术与 RTMP 传输协议简介

2.1 视频颜色空间介绍

根据颜色的定义得知，其本质是通过人类大脑、眼睛与生活经验对光所产生的一种视觉反应，通俗解释就是，颜色是人通过大脑对光的一种主观感觉。而颜色空间，是基于坐标系统和子空间上对彩色的一种说明表述。因此可以通过三种单色光的混合，来模拟出人眼所能感知到的几乎任何的颜色，这就是三基色原理，而所谓的三原色也便是由人类生理因素造成的。

颜色空间有很多种，常见的有 RGB、CMY、YUV 等。其中 CMY 是工业印刷所采用的颜色空间，它与 RGB 相对应，不同的是，RGB 是源于发光源混合相加，而 CMY 是依据无源物体对光的吸收与反射进行相减混合。此外，CMY 源自油墨染料的三基色：青（Cyan）、品红（Magenta）和黄（Yellow）的首字母缩写。

2.1.1 RGB 颜色空间

RGB 模型基础首次确定于 1931 年的英国剑桥市会议上，通过 CIE 国际照明委员会召开的此次会议，使用数学方法对真实基色进行推导演算，从而规定了新的颜色系统。新的颜色系统选择了以汞弧光谱滤波产生的色度稳定三原色：700nm（Red）、546.1nm（Green）、435.8nm（Blue）三种波长的单色光。RGB 颜色空间就是指采用这三种单色光为三基色而相混加的三维彩色空间。在 RGB 三维模式下，任意色光都可以通过不同分量色进行混合而成，关系如公式 2.1 所示：

$$F = r[R] + g[G] + b[B] \tag{2.1}$$

其中 F 为所要表示的色光，而 r、g、b 分别为三基色的百分比系数，有 $r+g+b=1$ 的关系。各分量的比例系数如公式 2.2 所示。

$$r = \frac{R}{R+G+B} \quad g = \frac{G}{R+G+B} \quad b = \frac{B}{R+G+B} \tag{2.2}$$

RGB 颜色空间作为常见的颜色空间，经常被应用到视频处理、设备画面显示等相关领域[5]。然而随着人们对色彩显示技术研究的逐渐深入与追求的提升，RGB 颜色空间的弊端也逐渐凸显出来。在自然环境下，人类的视觉系统往往对亮度的变化比较敏感，对色彩变化反而较为迟钝；同时 RGB 色彩分量在显示过程中容易受到亮度的影响，当

亮度降低时，三原色的分量在此情况下的显示效果也会降低[6]。另一方面，RGB 变换是非线性变换，尽管各个颜色分量之间的相关性由于非线性变换的原因而变的非常小，但是由于奇异点与计算量等各方面因素限制，其显示效果在实际使用中差强人意。

2.1.2 YUV 颜色空间

现如今的 PAL 彩色电视标准中所采用的便是 YUV 颜色空间。其显示原理基于对人眼视觉成像分辨敏感度的研究模型，对图像的色彩差别与亮度差别进行对比分析，结果显示人眼对亮度的敏感度较高，对色彩的敏感度相比而言较低，因此使用 Y 来表示视频图像的亮度信息，通过 U 和 V 表示视频图像的色差信息[7]。对人眼敏感度较高的亮度信息 Y 进行图像细节表示，对敏感度相对较低的色度 U 和 V 进行着色渲染。通常较为常见 YUV 颜色空间格式有 Y:U:V=4:4:4、4:2:2、4:2:0等不同比例。如图 2.1 所示：

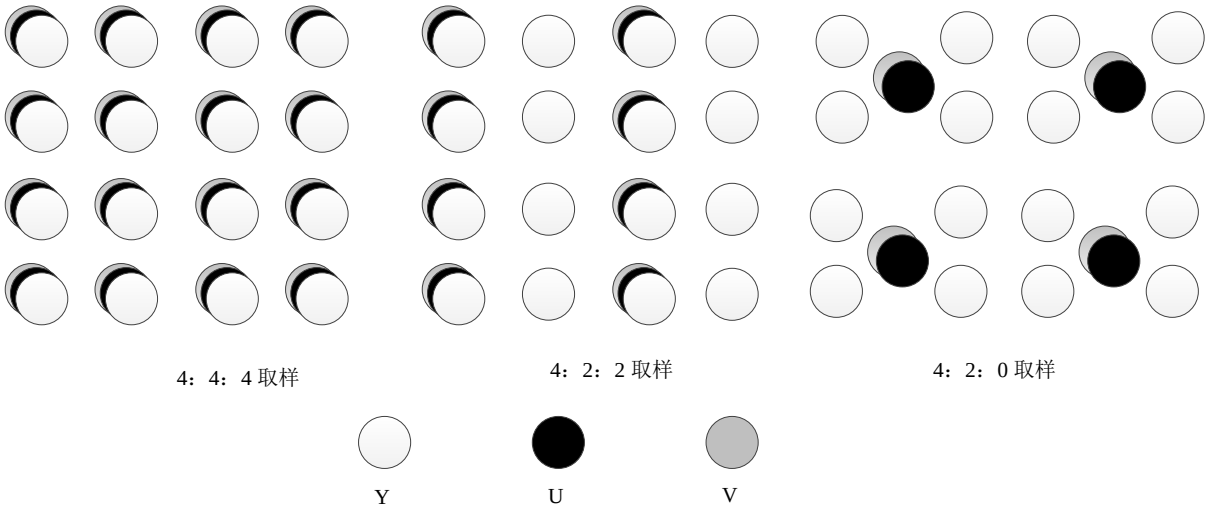


图 2.1　YUV 颜色空间

在 YUV 444 格式中，各信道拥有同样的水平和垂直采样率，保证了音视频数据分量信号的完整性，因此其数据量是三种格式中最大的。在 YUV 422 格式中，UV 色差信道的采样率与亮度信道 Y 采样率之比为1:2。而 YUV 420 格式并不意味着只有 Y 和 U 分量，其主要表达在同一行中只对一种色度分量进行存储，例如，当第一行的 YUV 信道采样率之比为4:2:0的话，则第二行的采样率之比就是4:0:2。因此，在三种不同采用格式中，YUV 420 的数据量最小。本课题所采用的颜色空间就是 YUV 420 。

2.2 高清 H.264 视频编码研究

视频编码，是将如 RGB 或 YUV 原始视频像素数据进行压缩工作，在经过编码压缩

后视频序列整体的数据量将会被大幅降低，通常情况下，如果一个视频序列在不经过编码压缩而进行存储传输，其占有的数据空间与网络带宽将会非常的大。因此，视频编码标准的制定具有重大的意义。其次，各种视频编码标准的应用面与侧重点各不相同，下表显示目前较为主流的视频编码标准。

表 2-1　主要视频编码标准一览表

名称	推出机构	推出时间	目前使用领域
HEVC(H.265)	MPEG/ITU-T	2013	未普及
H.264	MPEG/ITU-T	2003	各个领域
MPEG-4	MPEG	2001	不温不火
MPEG-2	MPEG	1994	数字电视
VP8	Google	2008	未普及
VP9	Google	2013	研发中

由表可见，有两种新推出的视频编码方案：VP9 与 HEVC，但是目前两种编码标准由于研发阶段较长，因此还没有达到一定的实用性与普及程度。而且， MPEG 系列视频标准对本地文件存储功能有着良好的支持特性，而 H.26x 系列视频标准则体现出对于网络传输特性的侧重。因此，高清实时网络流媒体直播系统通常选用较为常见的 H.264 视频编码标准。

2.2.1 H.264 标准基本框架

2001 年 ITU-T 与 IOS/IEC 两大视频标准制订组织合作成立视频联合小组 JVT，共同开发制定新一代视频编码标准[8]。因此，H.264 作为合作研究的成果孕育而生，继承了前代标准中的高效编码算法，并作为 MPEG-4 的第 10 部分发布，因此又被称为 MPEG-4 AVC。从技术角度分析，H.264 标准着重在压缩编码效率和网络传输的可靠性能进行多方位提升，拥有比较先进的标准设计思想和理念。因此，H.264 编码标准适应于多种应用环境与领域的需求。并且，H.264 标准中没有对编解码结构如何实现做细致硬性规定，只要满足标准中编码器输出码流语法和比特流解码格式的条件，就可以使不同编解码器模型下的 H.264 数据进行交互沟通，因此使 H.264 具有更高的环境适应性。

根据前代的视频标准，H.264 标准提供了不同的应用场景下的不同层级和档次。包括 Baseline Profile 基本档次、Main Profile 主要档次和 Extended Profile 扩展档次，2005 年增加了高档次 FRExt，主要用于高精度要求等领域[9]。不同档次对编码器的要求也各不相同，分为不同的 Levels 级，每一级支持不同的视频编码分辨率与码率。各档次如表 2-2 所示。

表 2-2 H.264 档次表

档次	功能模块	主要应用
Baseline	帧内预测、帧间预测、适应性变长编码 CAVLC 等	视频会议、可视电话等低延时领域。
Main	隔行视频、B_SLICE 帧间预测、加权预测、自适应算术编码等	视频存储、数字视频广播等领域。
Extended	SP 与 SI 片码流切换、误码性能改进、数据分割	流媒体应用领域。
FRExt	增加了保真度范围、可变量化矩阵、自适应尺寸变换	数字电影等无损领域。

此外，H.264 编码采用了分层的设计思想，将整个编码框架分为两层：视频编码层 VCL（Video Coding Layer）和网络抽象层 NAL（Network Abstraction Layer）。其中，VCL 层主要负责视频编解码部分，包括对视频序列的帧内帧间预测、DCT 变换编码、熵编码等模块。而 NAL 层主要负责下层网络传输部分，包括数据封装打包、组帧、逻辑信道的信令等功能，分层结构如图 2.2 所示[10]。

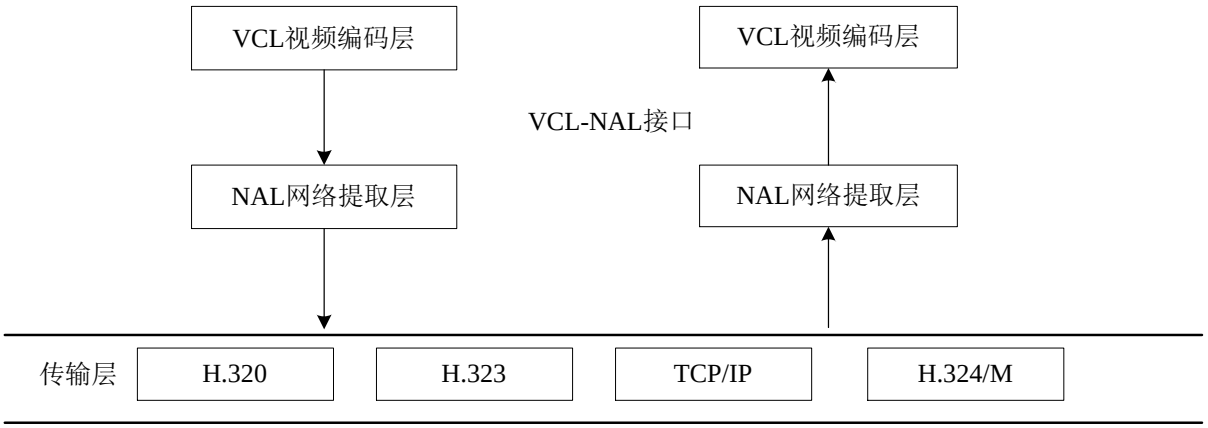


图 2.2 H.264/AVC 分层结构

前代编码标准都以高压缩比高效率为主要目的，而随着网络技术的发展渗透，在对编码效率的提升要求外，对视频质量与网络适应性要求也在提高，此前的编码标准模式面对复杂网络环境的亲和力不足等弱点逐渐显露出来，因此，H.264 就是对编解码性能与不同网络环境扩展性能支持的增强，通过分层结构的技术模式，VCL 与 NAL 接口使得这两层在不需要对码流重规划的情况下就可以互通，并且提高了可移植性，在不同的系统里可以保证编码设计与网络接口设计不冲突，同时达到最佳的编码与网络传输性能。

2.2.2 H.264 视频编码原理

视频数据的编码压缩是基于两个前提条件，数据冗余和视觉冗余。这两种非必要数

据大量占用有效信息的存储比率。其中，数据冗余主要指视频图像中相同信息的重复存储，消除这些重复信息后并不会导致信息损失，因此这种压缩编码称之为无损压缩。而视觉冗余则指的是在人眼的视觉感知系统中，对色彩亮度等信号强弱变化的分辨能力不同，在一定范围内对不同信号的敏感度不同，在对视觉冗余信息进行消除后，对图像的感知变化不大。因此，可以利用此项特性对视频图像进行一定的失真编码。这种编码压缩方式属于有损压缩。

H.264 视频编码标准正是基于上述两种条件对数据进行高压缩比处理，在图像信息损失不大的情况下，大幅提高了视频数据的传输与存储效率。一般情况下，视频压缩编码采用多种编码方式混合的形式，在压缩编码的基础上，对输出数据进一步压缩处理，极大地提高了数据的压缩效率。常用的编码方式如下。

一. 变换编码

变换编码主要原理是通过不同域之间的相互转换而产生相关性较小的系数，通过对转换系数进行编码来去除视频图像之间的相关性。一般是通过空间域转换到频率域，再对转换系数进行编码处理，这样就可以实现去除相关性，且使图像能量更加集中。常见的正交变换有 DFT 变换，DCT 变换等。

视频信号经过离散余弦变换后还需要对其进行数字量化处理，在量化过程中，可以对图像的高频细节信息不做或者做少量传递，因此在量化过程中，进一步降低了数据大小。此外，数字量化过程对图像信息的丢失情况比较严重，因此是图像信息损伤的主要原因之一。量化过程可以用以下公式 2.3 表示：

$$F_Q(u,v) = round\frac{F(u,v)}{Q(u,v)\times q} \tag{2.3}$$

其中 $F_Q(u,v)$ 表示对信号量化后的 DCT 系数，$F(u,v)$ 表示对信号量化前的 DCT 系数，$Q(u,v)$ 表示量化加权矩阵，q 表示量化步长，*round* 表示归整，即对得到的值进行整数取值，取最接近的整数值[11]。

二. 熵编码

熵编码的基本原理是通过对信源中出现次数较多或者概率较大的符号进行短编码，对于出现次数较少或者概率较小的符号进行长编码，进而使其可以在统计意义上获得较短的平均码长[12]。VLC 编码常用的有 *Huffman* 编码、算术编码、RLC 编码等。

一种较为高效的实现方式是当量化器输出直流系数后，对输出的交流系数进行Z型扫描，在二维量化系数转换为一维序列后，再进行 RLC 游程编码，最后对编码后的数据做另外一种变长编码。游程编码这一简单高速的压缩方法在变换编码之后使用，具有出众的实现效果并得到广泛的应用。

三．运动估计与运动补偿

通常视频中的图像序列在时间间隔较小的情况下不会产生非常大的变化，因此拥有较高的相关性，可以使用运动估计（motion estimation）和运动补偿（motion compensation）对其进行消除。时间相关性体现在序列中各图像帧之间的相似程度，在画面主体小幅变化的视频序列中，画面之间就体现出极大的时间相关性。为了提高编码效率与压缩比，没有必要对每一帧单独进行编码压缩，对此情况，运动估计和运动补偿的工作便是完成对相邻图像帧中变化较大的部分进行编码，从而进一步减少了数据量。

运动估计的实现逻辑是将当前的视频图像分成若干个图像小块，通过对前一帧或者后一帧图像进行相似度对比，搜寻两块相似度最高的图像块，这个搜索过程便称之为运动估计。

通过计算两个图像块之间的位置信息而得到一个运动矢量，然后根据运动矢量信息对当前图像块与参考图像块相减，便会得到一个残差图像块，对残差图像块信息进行编码存储就可以获得较高的压缩比，这个相减的过程便称之为运动补偿[13]。由于编码过程中的参考图像选择非常重要，所以一般情况下编码器将视频序列帧分为三种不同的类型帧：关键帧 I(Intra)、双向预测帧 B(Bidirection prediction)和预测帧 P(Prediction)。

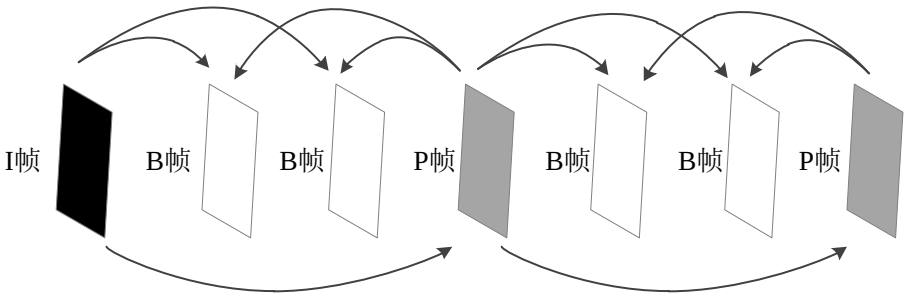


图 2.3　I、B、P 帧结构顺序

图中显示的视频序列中，关键帧 I 不参考其他图像帧而进行独立编码，主要进行帧内的压缩算法编码，因此它是不需要进行运动估计和运动补偿，所以在一般编码数据中，I 帧的数据量往往比其他帧类型大。预测帧 P 通过对前一个关键帧 I 或者预测帧 P 进行运动补偿，然后对残差值进行编码；而双向预测帧 B 在帧间预测中通过对前后两个方向

的参考帧进行双向预测，因此，与 P 帧相比，B 帧拥有更高的压缩比[14]。

四．混合编码

如图 2.4 所示，混合编码通过对视频图像序列分别进行 DCT 变换编码、熵编码、运动估计与运动补偿，在配合使用下达到较高的编码压缩效率。

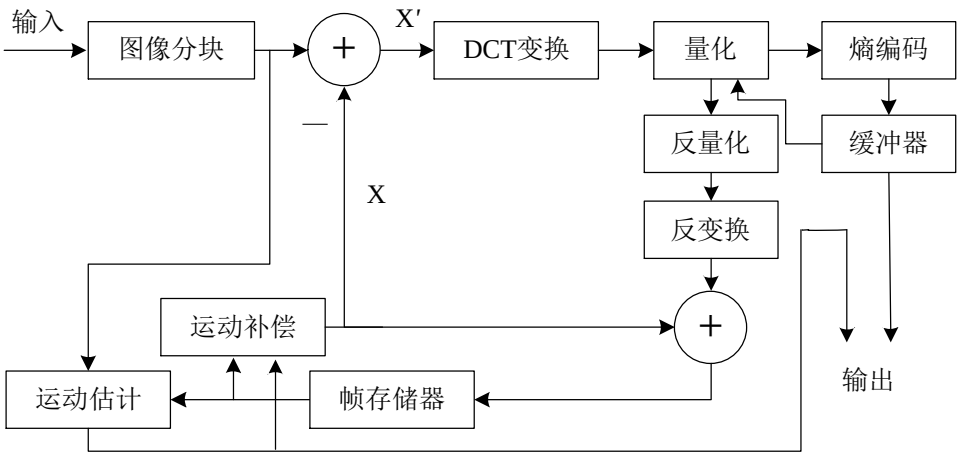


图 2.4　混合编码模型

首先，图像分块后分为两路，下路分块通过运动估计与运动补偿得到分块 X，上路分块通过与 X 分块相减得到残差图像块 X′。在经过 DCT 变换与量化，一部分经过熵编码输出到缓冲器中输出，另一部分通过对量化与 DCT 变换的逆过程得到 X′，将 X′ 与 X 分块进行相加并输出到帧存储器中。

2.2.3 H.264 预测技术研究

压缩编码的实现通常是在以高复杂度为代价的前提下来换取压缩性能的提升，H.264 作为一种高效的视频编解码技术标准继承了前代的基于块匹配的混合编码模式外，同时还引入了多种先进编码技术，例如不同模式下的高精度运动补偿、多帧预测、基于 4×4 像素块的 DCT 变换、自适应除块滤波等技术[15]。其中，帧内与帧间预测技术是大幅提高视频序列编码效率的两大重要技术。

1. 帧内预测

此前的编码标准都采用帧间预测的编码方式，忽略了各个图像宏块之间的相关性，因而编码后的 I 帧数据比较大。对此，H.264 引入了帧内预测算法来缩减图像帧的空间冗余。通过对宏块预测值与实际值相减带来的差值进行编码，从而大大降低码率。对于亮度与色度分开的 H.264 编码标准提供 4×4 与 16×16 两种块大小的亮度宏块帧内预测方

式，分别应对于图像亮度细节度不同的情况。当细节较多的时，编码器采用4×4像素的帧内预测，对于细节较少的画面则采用16×16方式预测。而对色块的预测，H.264 采用了8×8的色度块预测方式[16]。

（1）4×4亮度预测

4×4亮度预测模式共有 9 种，其主要原理是参考当前像素块上方的 9 个像素和左边的四个像素块，以自上而下的方式对当前块进行预测[17]。具体预测模式如图 2.5 所示，图中箭头方向代表了每种预测模式的方向，将要被预测的像素块为 a~p 块，而 A~M 块则是相邻块中已经重建的参考像素块。

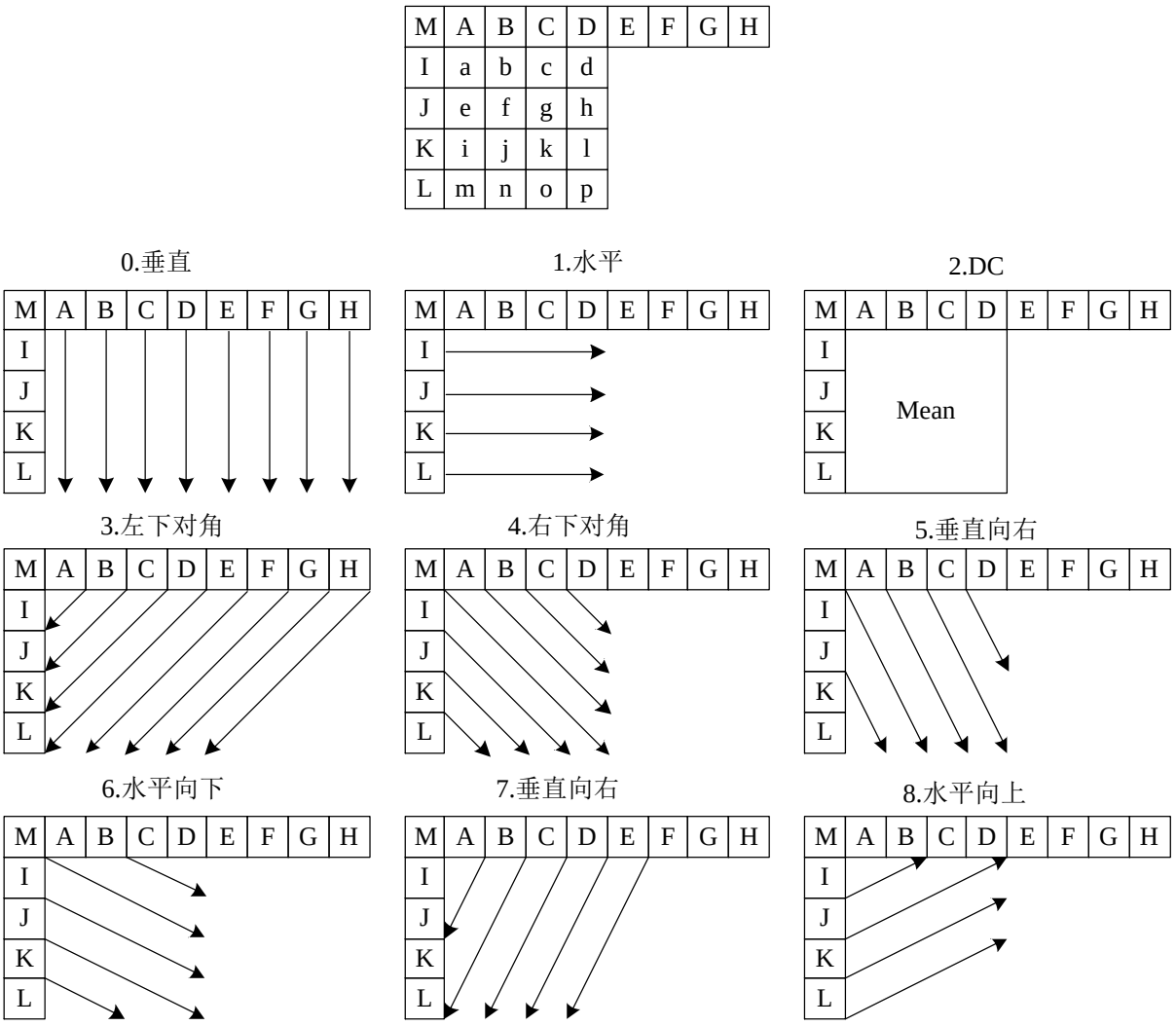


图 2.5　4×4 亮度块预测模式

（2）16×16亮度预测模式

16×16亮度预测模式共有四种不同的预测模式， Mode 0 垂直模式、 Mode 1 水平模

式、 Mode 2 DC 模式和 Mode 3 平面模式[18]。

垂直模式下，宏块上方的已经编码的像素 H 作为当前宏块按列参考块进行预测；水平模式下，宏块左边的已经编码的像素 H 作为当前宏块按行参考块进行预测； DC 模式下，宏块上方与左方的各像素 H 和 V 的平均值作为当前宏块的预测值[19]；平面模式下，宏块左方与上方的像素值通过内插得到预测值，如图 2.6 所示。

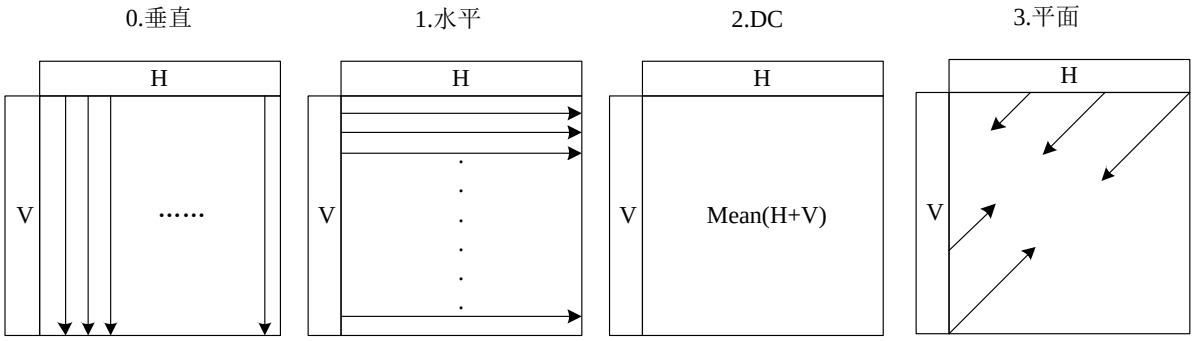


图 2.6　16×16 亮度块的 4 种预测模式

（3）8×8 色度块预测模式

对于色度信息只有亮度信息一半的 4:2:0 采用视频序列，8×8 色度块预测模式中的宏块通过对左面与上面的色度块进行预测而得到。而 16×16 亮度预测中的其他四种预测方法与 8×8 色度块预测类似。

2. 帧间预测

帧间预测的基本思想就是通过参考关键帧或预测帧对当前帧进行差值预测编码，从而达到对时间上的相关性与冗余信息进行消除的目的。H.264 标准在之前的编码框架基础之上，采用了更高精度的 1/4 像素运动搜索、可变大小的参考帧预测、多参考帧预测等技术，进一步提高了编码性能。

（1）预测块可变大小

预测块可变大小指的是将每个宏块分割成不同尺寸的子块，分割格式尺寸分别有 16×16、16×8、8×16、8×8、8×4、4×8、4×4。在进行运动估计时，通过运动搜索算法比较每种分块模式下的代价而选择最佳分块方式。这种宏块分割方法提高了各宏块之间的相互作用，同时又通过适当的划分，提高了运动估计的精确度[20]。缺点是会影响编码算法的压缩性能。分块如图 2.7 所示。

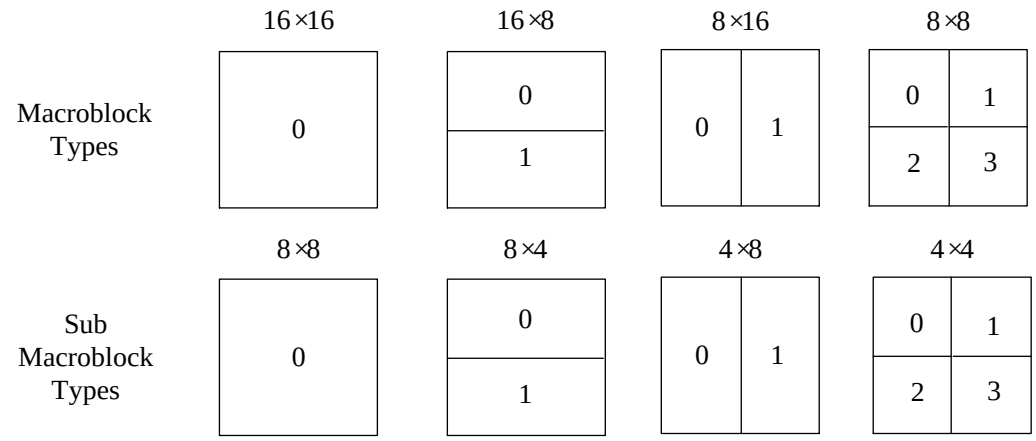


图 2.7 H.264 预测块可变大小模式

（2）1/4 亚像素运动估计

实验证明，1/4 亚像素精度已经基本上达到了运动估计性能提升的极限，更高的精度不仅对编码算法的提升不明显，而且还会导致较高的计算复杂度，因此，H.264 编码对亮度分量采用了 1/4 与 1/2 两种大小的运动估计，其中 1/4 亚像素点由 1/2 像素通过内插的方式得到[21]。1/2 像素点 Q 插值如式 2.4 所示。

$$Q = round((A - 5 \times B + 20 \times C - 5 \times E + F) \div 32) \tag{2.4}$$

其中，A、B、C、D、E、F 分别代表相邻 6 个垂直整像素。

1/4 像素点 P 插值如式 2.5 所示。

$$P = round((C + Q) \div 2) \tag{2.5}$$

其中 C 为半像素点中的整像素，Q 为水平半像素点。

2.3 音频 AAC 编码原理研究

AAC 编码属于感知编码，利用人耳对音频信号振幅、频率的有限分辨能力等特性，在编码过程中某些感知不到的信号不编码，在量化时候允许较大范围的失真；对感知明显的成分进行细致量化编码，尽可能保存音频细节，因此极大的减少了压缩数据量，而且保持较高音质。具体框架如图 2.8 所示。

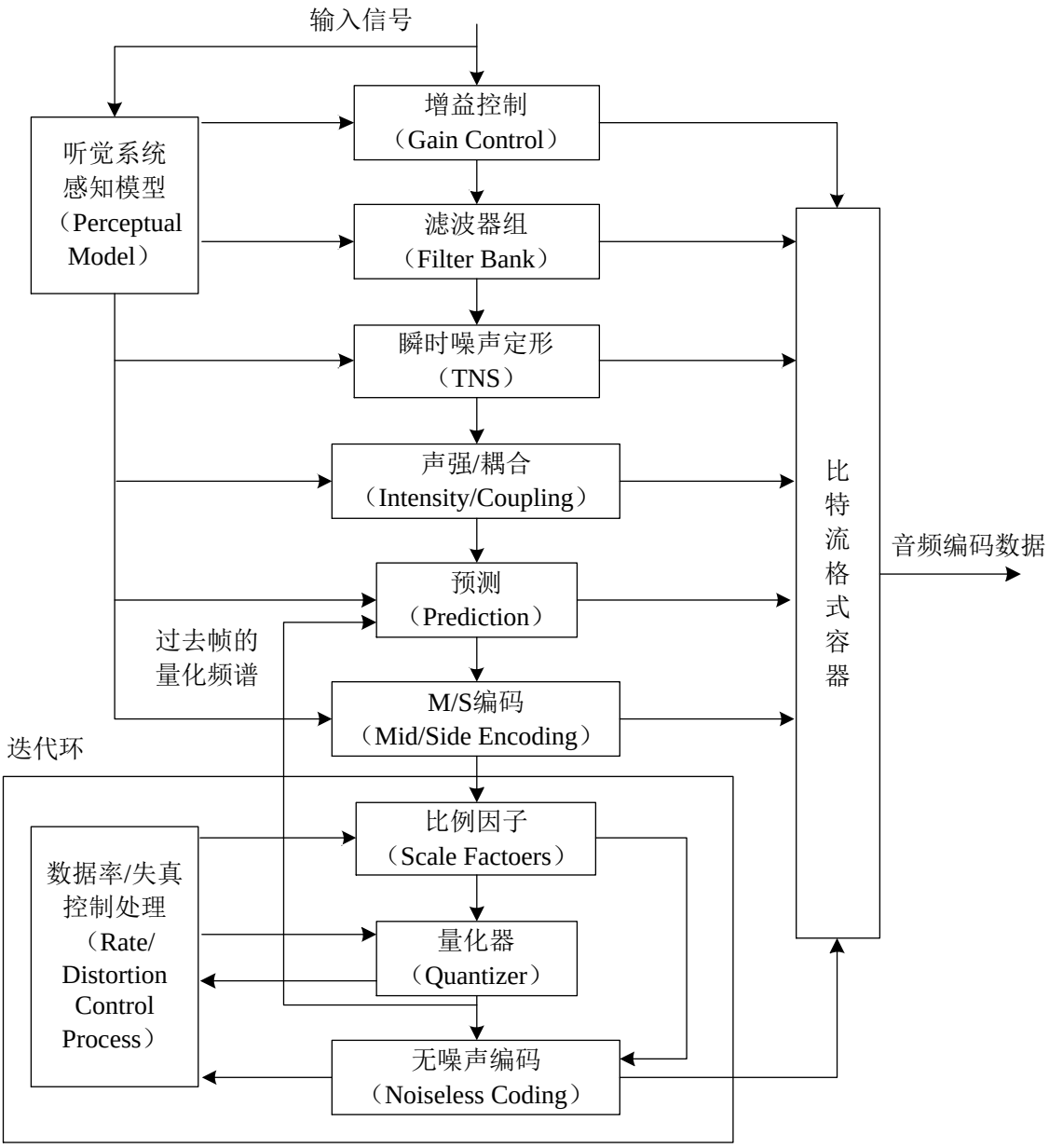


图 2.8　AAC 编码原理框图

首先，输入的音频信号经过听觉感知模型而计算出获得所需要的窗类型，再经过加强处理的增益控制模块进入滤波器组，TNS 瞬时噪声定形参数编码所需要的感知熵和 M/S 强度立体声所需要的信息由感知模块提供，并带有当前音频信号帧的掩蔽阈值。在所得阈值的基础上，可进一步计算出信号的信掩比，即 SMR。在最后的量化模块中可以利用此值来降低低频域信号在量化计算中产生的量化噪音。

在 TNS 模块对频谱系数滤波处理后，声强/耦合模块、预测模块以及 M/S 模块都会对音频信号做降噪处理，减少编码的冗余信息达到压缩码流的目的[22]。其中预测编码模块利用前两帧的频谱预测当前的系数而去除了帧间的频谱系数冗余，对帧间的预测残差值进行编码。在量化阶段需要比例因子信息对人耳感知特性信息进行压缩编码，根据心

理声学得到的 SMR 和掩蔽阈值，通过双层迭代循环进行量化处理。最后编码后的频谱数据与边信息完成组帧操作，形成编码后的音频 AAC 比特流数据输出。

2.4 FFmpeg 音视频处理技术介绍

FFmpeg 是一个包含多种音视频处理方法的 Linux 下开源项目，拥有目前音视频处理领域较为领先的编解码库 libavcodec，同时也是被各种项目应用所广泛采用的一个编解码解决方案。如使用 FFmpeg 作为内核播放器的 Mplayer、暴风影音、QQ 影音、KMPlayer 等。事实上，FFmpeg 的音视频处理能力确实很强大，几乎囊括了现存所有的音视频编码标准。因此，只要是与音视频处理技术开发相关，几乎都离不开它。此外，FFmpeg 在不仅在音视频处理方面具有较高的效率，其还具有非常好的扩展性，可以在多个平台下进行编译移植。对其包含的几个主要模块做以下简单介绍：

libavcodec：包含 avcodec 音视频编解码器、DXVA2（DirectX Video Acceleration）视频硬件加速规范、VDA 硬件加速、xvmc 等功能模块，是 FFmpeg 类库的核心，实现了大部分编解码器的功能，并被广泛使用。

libavformat：提供对音视频复用封装、字幕流、音视频分离等处理技术的通用框架，包括获取编解码所需要用到的上下文结构体信息、流媒体信息、复用 Muxer 与解复用 Demuxer 等，同时支持多个媒体资源的输入输出访问方式。

libswscale：Swscale 类库用于对音视频进行像素数据的格式转换、色彩映射转换、图像大小的拉伸、缩放显示等。

libavdevice：AVDevice 可以读取多媒体设备数据，进行硬件采集、加速，或者输出数据到指定的多媒体设备、内存以及本地文件中。

libavutil：包括一些内存的操作、CRC（Cyclic Redundancy Check）循环冗余校验、内存分配管理、大小端序列转换等功能。

libavfilter：AVFilter 可以给音视频添加各种滤镜效果，例如给视频添加水印。

2.5 RTMP 的流媒体传输协议研究

RTMP 是实时消息传输协议（Real Time Messaging Protocol）的简称，由 Adobe 公司专为 Flash 平台与流媒体系统之间的音视频数据传输而开发的私有协议。在 Flash Player 已经安装在全世界将近 99%的计算机背景下，使用 RTMP 技术的流媒体系统将会体现出一个非常明显的优势，就是使用 Flash Player 作为观看直播的用户端将不需要安

装任何插件，只需要打开对应的直播网页就可以收看实时直播，因此，其便利性被众多直播系统所采用。

尽管私有协议，但根据 Adobe 曾经发布过的一份《RTMP Specification》，便可以对 RTMP 的实现细节进行推敲研究，具有很高的参考意义。

2.5.1 RTMP 协议研究概述

RTMP 协议是应用层协议，因此 RTMP 块流不提供优先级别或类似可靠性控制来保障信息传输，因此通常配合 TCP 这样的可靠传输层协议来建立连接，RTMP 块流保证跨流的所有消息可以按照时间戳序列传输。在传输层连接建立完成后，RTMP 客户端与服务器之间需要再通过“握手”来建立基于传输层之上的网络连接 NetConnection，主要用于传输例如 SetChunkSize、SetAckWindowSize 等控制信息。随后通过 CreateStream 命令会创建一个网络流 NetStream 连接，用于传输具体的音视频数据和控制这些信息传输的命令信息。连接示意如图 2.9 所示。

图 2.9　RTMP 流连接图

图中 NetConnection 表示服务器程序与客户端应用程序之间的网络连接通道，而 NetStream 则代表了对音视频数据网络传输的消息通道，尽管在客户端与服务器之间只存在一个网络连接通道，然而根据此连接通道可以建立多个流媒体数据网络传输流通道。

2.5.2 RTMP 的三次握手

RTMP 连接的建立都是以握手为开始，双方分别发送大小固定的三个数据块，当连接成功后才可以有效的进行数据信息传输[23]。尽管 RTMP 协议本身没有硬性地规定这些消息 chunk 的具体发送顺序，但实现过程中仍需要注意：

（1）握手开始于客户端发送 C0、C1 块。

（2）服务器收到 C0 或 C1 后发送 S0 和 S1。

（3）当客户端收齐 S0 和 S1 后，开始发送 C2。

（4）当服务器收齐 C0 和 C1 后，开始发送 S2。

（5）当客户端和服务器分别收到 S2 和 C2 后，握手完成。

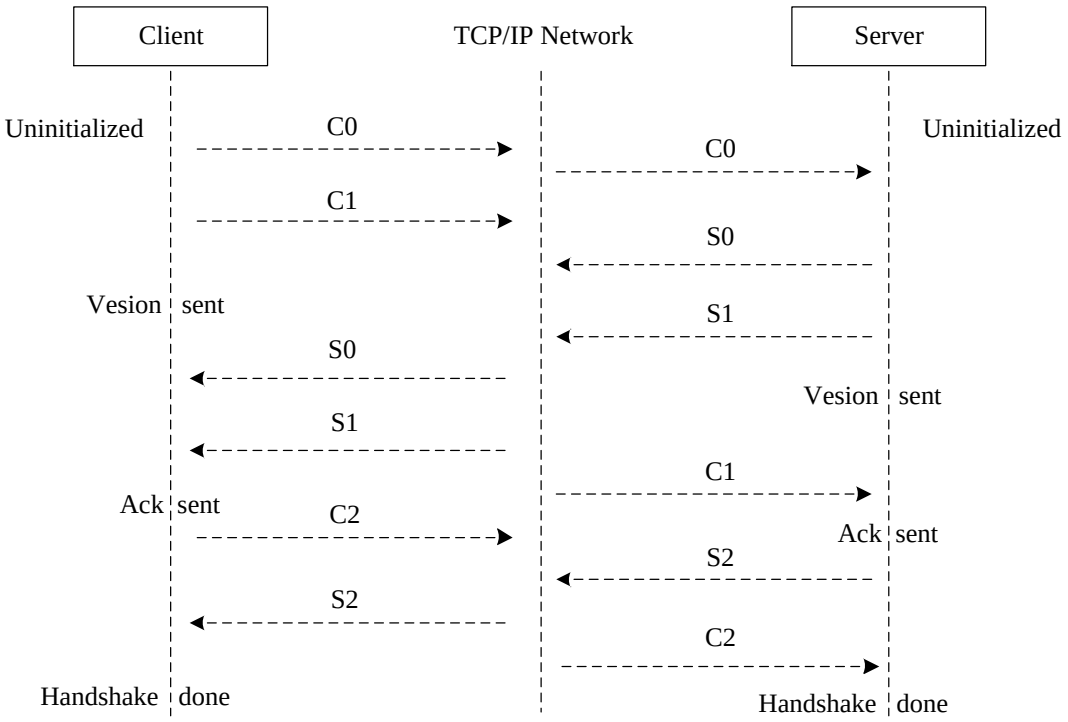


图 2.10　三次握手示意图

以上的发送顺序主要目的是在保证“握手”身份验证的基础功能上尽量减少通信的次数，这一点可以通过 wireshark 抓 FFmpeg 推流包进行验证。

2.5.3 RTMP Message 消息

消息是 RTMP 协议中基本的数据单元，服务器与客户端进行通信时，消息可能包含音频、视频、数据或者其他信息。因此，不同的消息种类有着不同的 Message type id，代表着不同的功能。

Message Type ID 1~7：此类消息用于协议控制，一般由 RTMP 协议的本身管理过程中需要使用到的信息，用户无需操作。

Message Type ID 8~9：此类消息用于音频和视频的传输。

Message 消息的报文结构如图 2.11 所示：

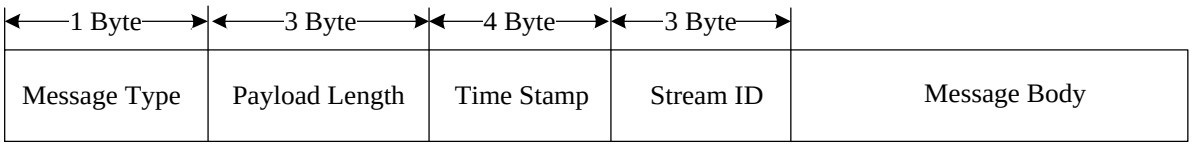


图 2.11　消息报文结构图

其中，消息头包括以下内容：

Message Type：消息类型，用于表示消息类型。

Payload Length：负载长度，表示负载信息的字节数，为大端模式。

Time Stamp：消息的时间戳信息，以 4 个字节大端方式打包。

Stream ID：每个消息的唯一标识，分割 Chunk 和收集 Chunk 合成 Message 的时候都需要此 ID 信息来分辨是否为同一消息或消息流。以 4 字节小端存储。

Message Body：消息的负载数据。

2.5.4 RTMP Chunk 分块

Chunk 块流是 RTMP 协议数据传输的基本单位。其分块的主要意义就是使高层协议的大消息分割成小的消息，保证消息不被阻塞。同时对于数据量比较小的消息，则可以通过 Chunk Msg Header 字段来压缩消息。

在消息被分割成多个 Chunk 块的过程中，消息负载部分 Message Body 被分割成多个最大容量为 128 Byte 的数据块，并在块的头部添加相应的 Header 消息头，组成消息块。消息分块过程如图 2.12 所示。

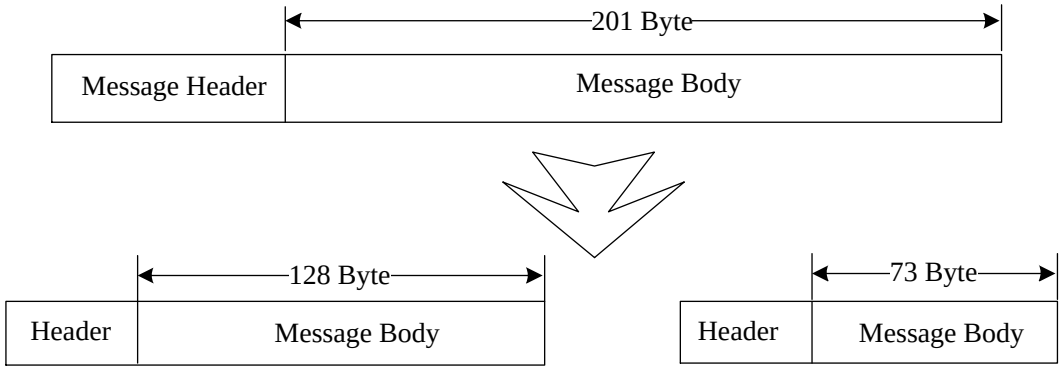


图 2.12　消息分块图示

在 RTMP 握手连接建立后，消息（Message）被分割为以 Chunk 为单位的数据单元，被创建的 Chunk 都关联到唯一的块流 ID，多个 Chunk 组成一个 Chunk 流，因此 Chunk 流可以看作是数据的传输通道，承载了来自一个消息流的一类消息。在传输过程中，必须在 Chunk 发送完之后再发送下一个 Chunk。在接收端，每个 Chunk 都根据块 ID 被收集成消息。Chunk 由以下四部分组成：

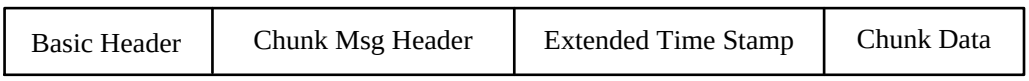


图 2.13　RTMP chunk 基本构成

（1）Basic Header：基本的头信息，用于定义 chunk 类型和 ID，1~3 字节。

（2）Chunk Msg Header：块消息头编码要发送的消息信息，有 0、3、7、11 字节四

种长度，长度和格式取决于 Basic Header 字段中 chunk type 和 fmt。共有四种不同的格式：timestamp 时间戳、message length 消息数据长度、message type id 消息类型 id 和 msg stream id 消息的流 id。

（3）Extended Time Stamp：扩展时间戳字段是在发送普通时间戳为 0xffffff 时发送，当普通时间戳的值小于 0xffffff 时，此字段不需要出现。需要注意的是，扩展时间戳存储的是完整的值，而不是差值。

（4）Chunk Data：块数据字段，发送与协议无关的数据。

2.6 本章小结

本章主要重点介绍了直播系统设计实现过程中所涉及到的视频颜色空间、H.264 视频编码标准、AAC 音频编码原理、FFmpeg 音视频处理技术以及 RTMP 流媒体传输协议，为后面章节的展开与系统实现铺平了理论道路。

第三章 基于 FFmpeg 高清实时直播系统总体设计

3.1 基于 FFmpeg 高清实时直播系统需求分析

基于 FFmpeg 的高清实时直播系统最终目标是实现多媒体数据的高清实时共享，因此直播系统应具有对桌面视频信号与麦克风设备信号采集功能，利用软件技术与音视频编解码技术对采集到的原始音视频数据进行解码、编码、音视频同步、复用封装。最后，通过组建流媒体服务器将压缩编码后的流媒体数据进行网络转发，完成直播系统实现。

此外，针对现有直播系统中所存在的不足，应设计出解决的思路与方法。例如对高清、超清或者更高分辨率的流媒体的支持；在对音视频处理过程中，在编解码流程上对系统的处理效率与稳定性进行提高；在网络延时较高的问题上，采用逐帧发送与逐帧转发等方式提高直播系统的实时性等等。

3.1.1 系统功能性需求

本文所设计实现的高清实时直播系统针对现有直播系统中存在的一些问题，对系统提出以下几点功能性需求。

1. 设备数据采集速率控制

在 Windows 平台下对设备的数据采集主要包括计算机桌面画面数据、麦克风设备音频数据和系统内置扬声器音频数据。在数据采集过程中，需要对采集的速率进行控制，速率过高或者过低都将会影响整个系统的效率和音视频质量。

2. 视频颜色空间转换

目前主流的显示颜色空间分为两种：RGB 颜色空间与 YUV 颜色空间。RGB 颜色空间是图像或者视频处理中最常见的颜色空间，但由于人眼感官对亮度较为敏感，其三原色分量会因亮度不同而产生误差，因此显示效果不够优秀。基于此，YUV 颜色空间以 Y 表示亮度信号，U、V 表示色差信号改善了显示问题，因此被彩色电视标准所使用。直播系统在对采集到的 RGB 显示画面数据进行 YUV 颜色转换后，再进行高清编码压缩。

3. 解码数据缓冲策略

解码数据缓冲区的设计实现，提高了系统软件在采集数据处理过程中的流畅度与稳定性，消除了多线程同步、音视频解码与编码效率不同步等问题的影响。对于经过颜色空间转换后的 YUV 视频数据，写入到视频缓冲区，在视频编码线程中通过读取访问获取到数据后进行编码压缩。而 PCM 音频数据同样经过处理后写入到音频缓冲区，在音

频编码线程中读取、编码。线程之间通过临界区资源进行线程同步。

4. 高清视频编码

原始的视频数据往往是非常巨大的，而编码压缩技术的出现解决的这一问题。目前视频编码标准分为两类，一种是 H.26x 系列的高清视频编码标准，一种是 MPEG-x 系列的视频编码标准[24]。对比之下，H.26x 系列具有更好的网络传输特性，且压缩效率更高，故应采用 H.264 作为高清视频编码标准。

5. 高级音频编码

在音频编码技术中，MP3 音频编码标准与 AAC 音频编码标准都属于有损压缩，但是由于种种因素，目前较为常见的是 MP3 编码。不可否认的是，无论编码效率、高频段编码还是听觉感知质量，AAC 编码始终优于 MP3，按照 MPEG 规范上的说法，AAC 生来就为取代 MP3，故采用 AAC 作为音频编码标准。

6. 时间戳同步技术

因为采集到的视频与音频数据在时间和空间上都是相互独立的，因此在编码完成后，需要对音频与视频数据进行音视频同步。音视频同步过程就是对音视频帧序列进行时间排序，原始 H.264 数据往往没有时间戳信息，因此在同步阶段给音视频数据打上时间戳，通过时间戳比较，将 H.264 与 AAC 数据依次同步。

7. 逐帧推流

目前大多数直播系统所采用的是流媒体数据按块传输的设计，因此极大地增加了直播延时。对此，本文采用逐帧传输的设计，在音视频数据同步完成后，通过传输速率控制算法，将数据帧依次传输到服务器端，保证了系统时延可控。

8. 高并发直播服务器

系统在直播过程中对于高并发请求的处理需要高效的直播服务器支持，同时在网络延时问题的处理上，应选择实时性较好的流媒体传输协议。因此系统采用基于 Nginx 的 RTMP 流媒体直播服务器，不仅提高了直播时的最大请求连接数量，同时也更进一步地提高了系统的实时性。此外，对于直播用户端的部署设计，通过 RTMP 协议可以搭配 Flash 播放器，提高了用户端的易用性。

3.1.2 系统非功能性需求

直播系统的非功能性需求主要体现除功能性需求以外的其他特性，包括系统的性能、易用性、稳定性与界面美观等，具体说明如下：

1. 安全性

安全性体现在软件设计的多个方面。例如对登录用户身份的识别验证设计，保证直播软件在未授权的情况下不启动功能界面；软件即将开始运行前，当检测到当前应用环境或场合不适合直播系统展开工作时，对用户的操作进行提示；在软件运行过程中，遇到意外情况可以及时安全地停止，保护设备不受损坏；

2. 易用性

直播系统的易用性表现在软件安装布置简单，容易上手。在不具备音视频专业基础知识的用户在使用时，提供默认的参数设置，同时添加对各项音视频参数设置的说明。同时在客户端设计中，采用通用的 FLASH 播放器作为接收端，具有很高的便捷性，不必再重新安装客户端插件等。

3. 性能

如表 3-1 所示，直播软件的性能需求主要体现在如下几个方面。

表 3-1　直播软件性能需求表

项目	需求
视频分辨率	支持各范围视频分辨率，如超高分辨率 2K、4K 等
视频帧率	采集编解码帧率大于 24FPS
音视频比特率	视频 56Kbps~4000Kbps、音频 16Kbps~512Kbps。
硬件占用率	CPU 占用小于 20%、内存占用小于 130M
网络带宽	网络带宽占用小于 2Mbps
硬件需求低	单核 CPU，256M 内存、普通网卡下可以正常运行

4. 编解码稳定性

系统在数据采集、编解码过程中对硬件不可避免的造成高负荷，为提高采集帧率与编解码效率，应利用多线程编程技术。在对数据的处理流程中，使用多线程可以提高工作效率，同时，线程间同步技术让各线程可以稳定安全的完成任务。

5. 可维护性

在软件实现过程中，使用较为通用的匈牙利命名法，根据函数、变量、常量等相应的用途添加合适的前缀名或后缀名。这样可以方便管理使用人员对系统进行日常维护，同时对将来的扩展开发提供了便利。

6. 界面美观

直播软件界面应具有简洁、美观、大方等风格，同时添加相应的软件控制功能，如开启直播、结束直播与直播参数设置等。方便用户使用。

3.2 系统总体框架设计

本文所设计的直播系统组成大致可分为两部分，直播软件部分与流媒体服务器。直播系统软件设计采用 Windows 平台下的 MFC 应用程序框架，完成对音视频数据的采集与编解码处理。直播服务器采用基于 Nginx 的 RTMP 流媒体服务器，完成对数据的接收转发功能。此外，客户端采用 Adobe 公司的 Flash Player 完成对发布数据的接收与播放功能，系统整体拓扑结构如图 3.1 所示。

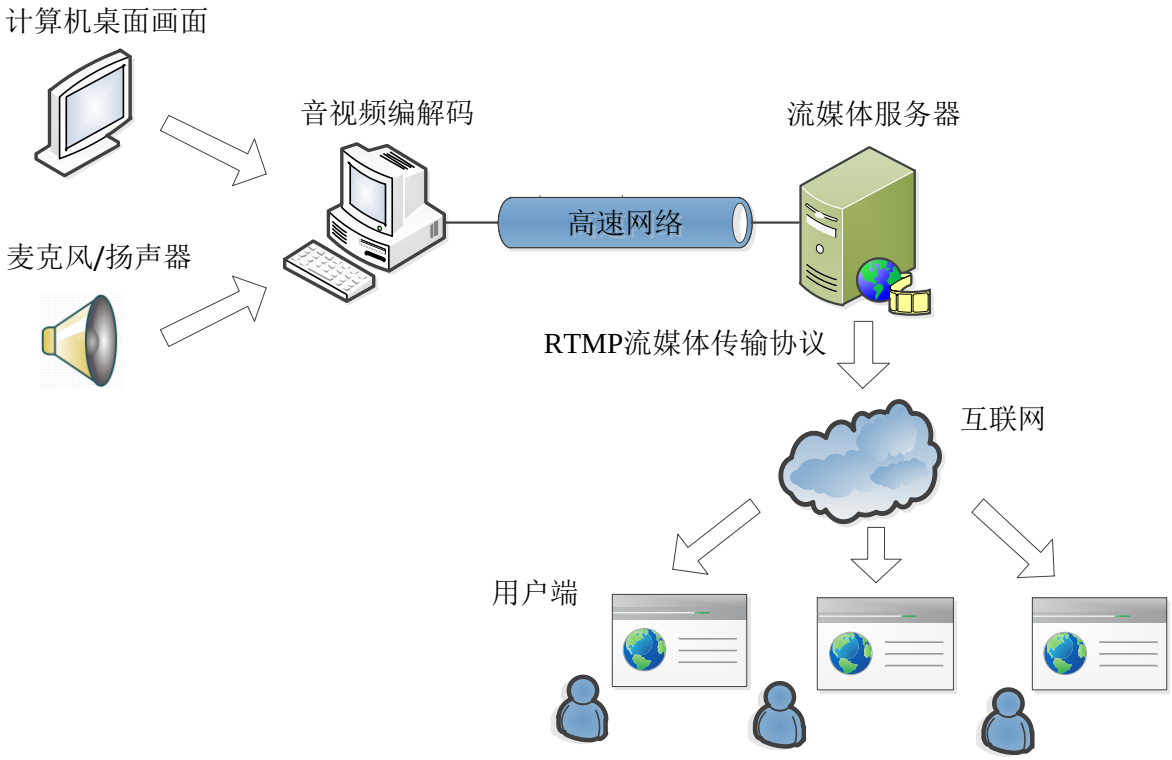


图 3.1 直播系统拓扑图

直播开始后，直播软件将采集到的音视频数据进行编解码处理，通过流媒体逐帧推送到服务器端，同时根据需求进行直播数据本地存储。服务器将收到的流媒体数据片发布到相应的直播地址，完成直播工作。直播系统架构图如 3.2 所示。

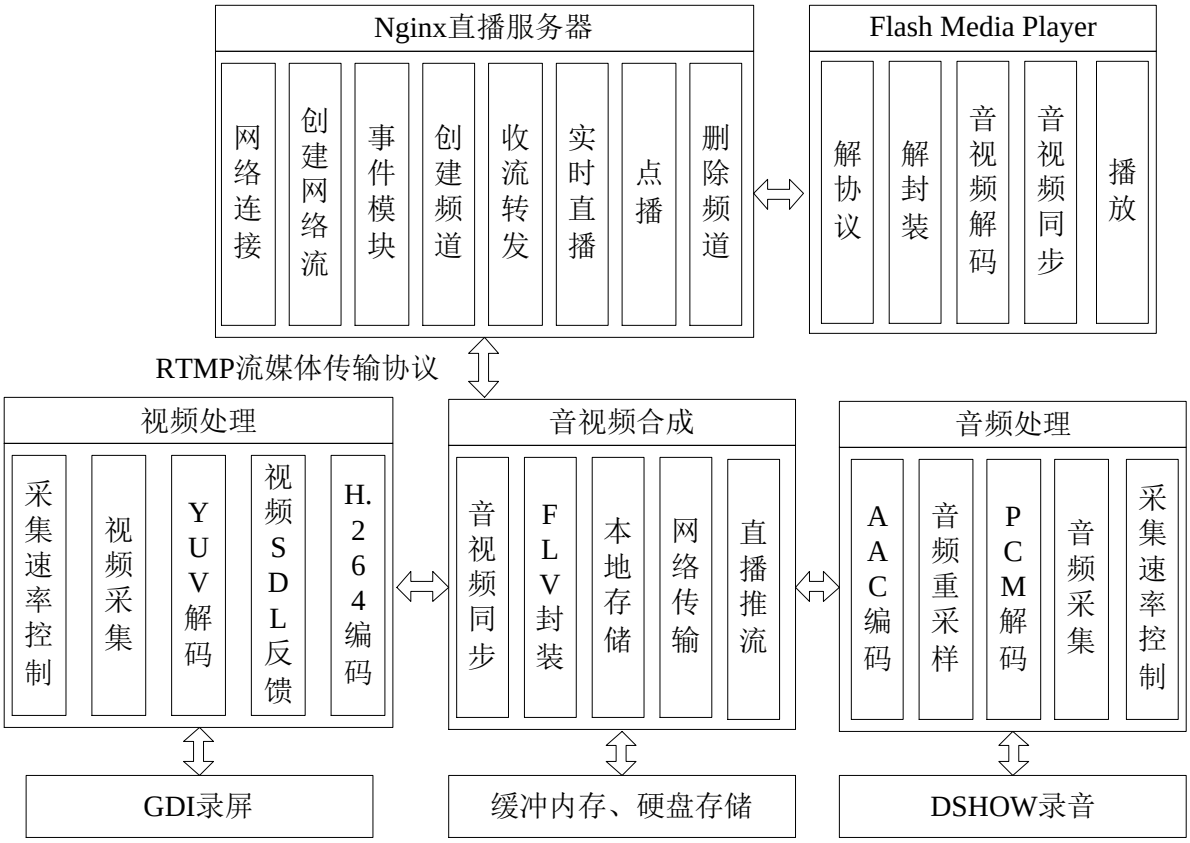


图 3.2　直播系统架构图

从系统架构图中可以看出，直播系统相关的音视频处理工作在应用软件端中实现，通过多线程编程技术，使软硬件资源得到充分利用。同时，在直播服务器端，搭建了基于 Nginx 的流媒体直播服务器，使用 RTMP 流媒体传输协议，将处理封装后的音视频直播数据进行了网络转发。

3.2.1 直播软件功能设计

直播软件在对音视频数据处理中采用多线程编程，将运算量较大的音视频采集、音视频解码与音视频编码封装等功能分为多个线程，线程间使用临界区资源进行线程同步。在采集功能中，通过对采集速率进行控制而达到对硬件资源与音视频质量的平衡，在解码功能中，通过设计解码缓冲区提高编解码的效率与稳定性。最后，通过时间戳同步技术与 FLV 封装规范，对编码同步后的音视频数据进行 RTMP 逐帧推流，在与服务器建立流媒体连接通道后通过传输速率控制完成传输工作。

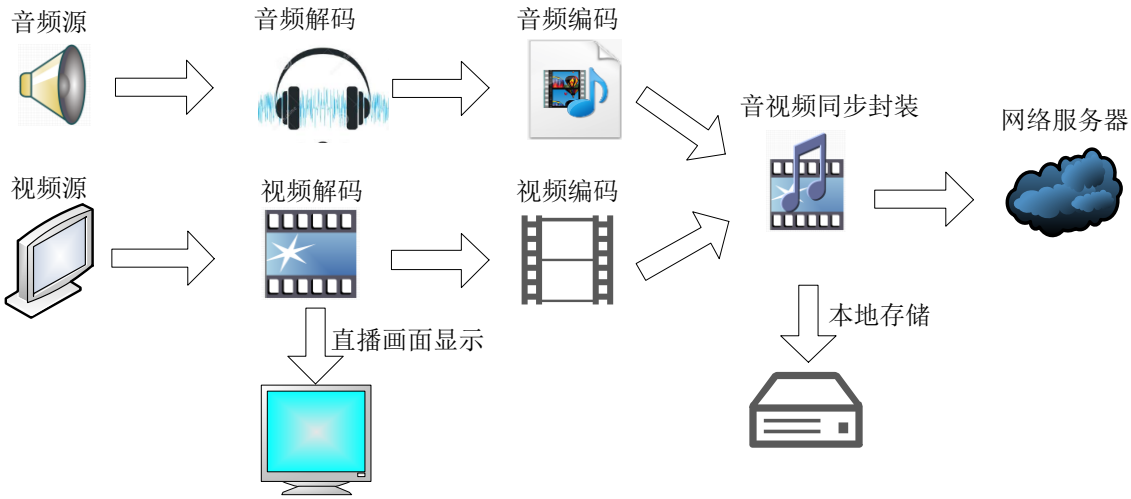


图 3.3　直播软件功能

根据直播软件各相关功能之间的联系，将软件设计分为如图 3.4 所示的 5 个模块。主要包括文件系统模块、音视频模块、图形模块、直播参数配置模块以及直播控制模块。

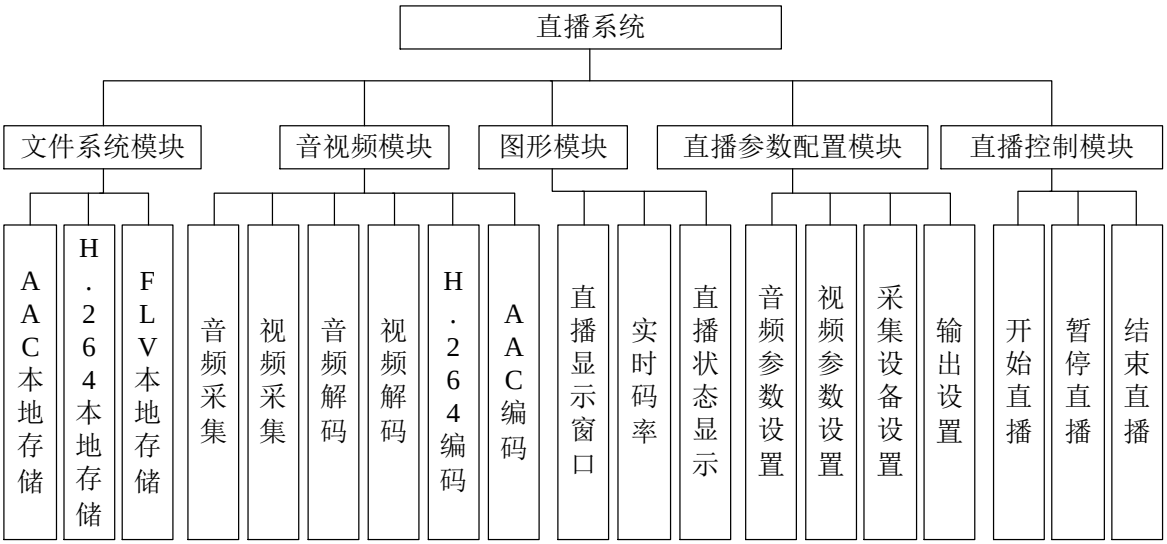


图 3.4　直播软件功能框图

1. 文件系统模块

文件系统模块主要功能是将采集编码后的压缩音视频数据本地存储。对音频数据进行 AAC 压缩编码存储，对视频数据进行 H.264 压缩编码存储，提供特殊查询情况下的分离多媒体数据，在复用封装成 FLV 文件后，同样对封装后的文件进行本地存储，以提供对日后点播、录像查看功能。

2. 音视频模块

音视频模块是直播软件系统的核心模块，完成对音视频数据采集、解码、编码等一系列操作，因此功能繁多，若在单一线程内对音视频数据进行操作，将达不到高清低延

时标准。为改善音视频质量，将音视频模块功能分解为多线程操作。

线程一：视频数据采集。通过设备流将采集到的视频数据解码，并将解码完成后的数据写入到视频数据缓存区，通过临界区同步多个线程的资源访问。

线程二：音频数据采集。音频采样，即将连续的模拟信号转化为离散的量化信号，根据奈奎斯特（Nyquist）采样定理，采样频率 *Fs* 大于信号频率的两倍时，可以完全重构原信号，所以为保证声音不失真，直播器音频采样频率采用 44.1KHz，即每秒 44100 个样本数据，在一帧音频 1024 个采样数据下，一帧的解码时间必须控制在 23.22ms 内[25]。因此带来大量的采样数据，造成较高的硬件负荷。对此，音频采集同样分为一个单独的采集线程，通过音频数据缓冲区与临界区资源完成不同线程间的数据访问与线程同步。

线程三：音视频编码同步。音视频编码模块放在同一进程内，可以方便对音视频时间戳的修改设置，并减少内存资源的消耗。在完成对音频，视频数据解码后，通过查询判断音视频缓冲区数据大小，对读取到的视频帧进行 H.264 编码，对音频帧进行 AAC 编码。编码完成后，再选择合适的时间基准（TimeBase），使用循环判断对音视频数据添加合适的显示时间戳（PTS）、解码时间戳（DTS），并逐帧封装发送到直播服务器。

3. 图形模块

图形模块包括在直播软件主界面对视频画面的显示、实时码率图的显示、各项音视频参数与音视频设备源的设置与显示功能。系统直播过程中对用户的反馈功能，是用户做出合理操作的重要指标，在直播画面出现异常或质量下降时，可以方便及时查找并解决问题。在画面显示模块中，将对视频数据进行获取、渲染画面、显示内容、更新纹理层等操作。同时，音视频数据的各项参数状态、编码效率参数与直播器的效率状态显示数据等，也是用户对直播工作状态判断的一个参考。对此，系统将采用操作系统消息反馈机制，实时更新各项参数数据，并展示在监视面板上，方便用户查看。

4. 直播参数配置模块

直播参数配置模块主要包括：音频参数设置、视频参数设置、采集设备源以及输出文件设置。参数设置模块设置好的参数将会被用于音视频模块、图形模块等其他软件模块，是用户自定义直播内容、质量、功能的入口。

5. 直播控制模块

直播控制模块主要包括直播器的开始功能、暂停以及结束直播功能。在异常发生时，用户可以更方便的立刻结束并关闭直播器。

3.2.2 直播软件流程设计

直播软件功能程序在启动前，会有一个登录的窗口生成，以便初步验证使用者的账号信息。在验证通过之后，进入直播功能主程序。软件设计流程图如下所示。

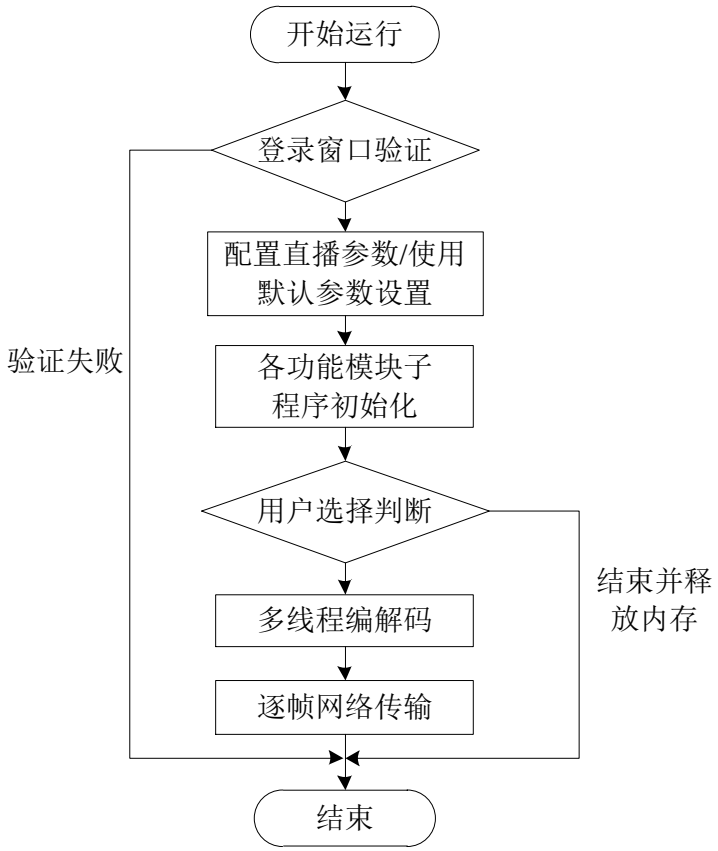


图 3.5 软件功能流程图

进入直播功能主程序后，用户需要对直播音视频相关参数进行专业的自定义设置，同时，面相那些非专业用户，提供了默认的配置参数。在直播开启后，软件首先会对各功能子程序相关函数与变量进行初始化，包括采集模块、音频解码器、音频编码器、视频解码器、视频编码器、SDL 显示模块等。

初始化完毕后，通过对用户选择的直播任务，判断对应的逻辑标志位字符，以获取用户的需求信息：如直播的开启、音视频数据的本地存储、暂停直播与结束直播等。在多线程编码功能模块中，包括对直播数据的采集、速率控制、解码显示、编码压缩等功能子模块。在对直播音视频进行编解码处理后，通过音视频同步与 FLV 格式封装，最终完成直播音视频数据逐帧网络传输功能。

3.2.3 服务器功能设计

直播服务器的主要功能是在软件端将直播数据推流过来后，将直播画面与音频数据

发布到客户接收端。当直播软件推流功能与服务器建立流媒体传输链接时，服务器便创建监听并等待流媒体数据的接收。收到一帧流媒体数据后，立即发布到对应的客户端地址中，保障了直播数据的及时性。同时，客户端采用普及率非常高的 Flash Player，无需安装另外的播放插件，具有更高的便捷性。

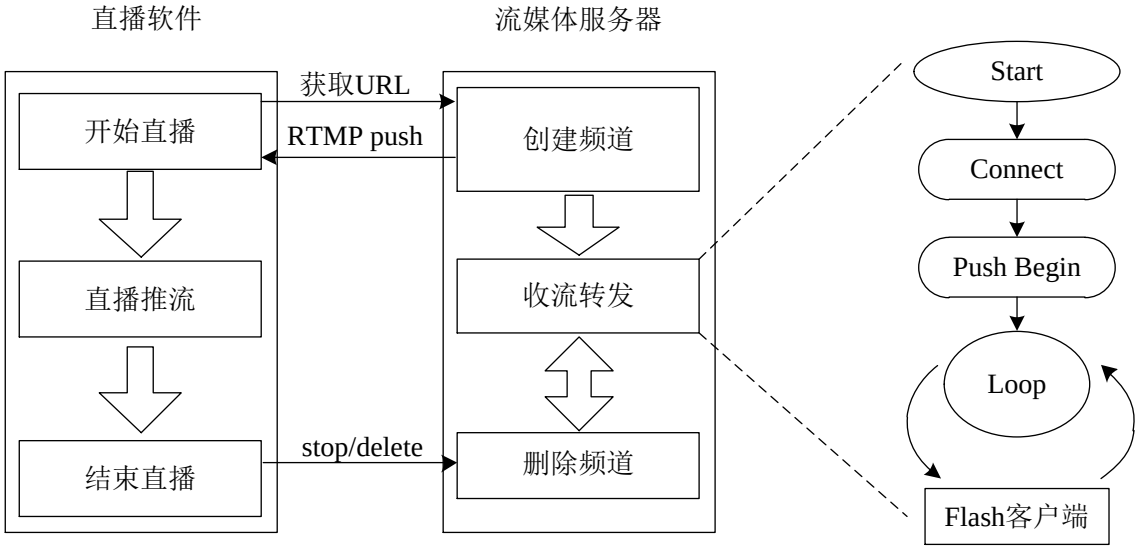


图 3.6　服务器功能框图

当直播软件发起直播连接请求时，通过直播 URL 与流媒体服务器建立连接，连接建立后，服务器在创建直播频道后向直播软件推送命令消息完成应答。直播过程中，服务器将收到的流媒体数据帧经过循环转发到 Flash 客户端，在直播结束后，服务器收到结束消息后清空数据通道并删除直播频道。

3.3 本章小结

本章首先针对现有直播系统中存在的一些问题提出了几点功能性需求，同时对直播系统的非功能性需求做了分析，接着对系统的总体框架进行了设计，包括对直播软件的功能设计与直播服务器端的功能设计。

第四章 高清实时直播软件功能实现

4.1 直播软件开发环境搭建

4.1.1 FFmpeg 编译移植

由于 FFmpeg 是在 Linux 下开发的开源音视频处理项目，因此在 Windows 环境下的对 FFmpeg 进行编译工作是比较困难的，不仅因为 FFmpeg 使用了 AT&T 与 C99 两种语法，而且 Windows 下的 VS 开发工具对 C 语言的支持度比 GCC 要差很多，不能直接使用 MSVC 进行编译，需要借助工具在 Windows 下配置一下类似与 Linux 的编译环境，以达到编译 FFmpeg 的目的，具体过程如下。

1. 下载安装 MinGW+Msys，在点击 continue 进行结束安装后，进入到下载环节。选择基础安装，全选必要的下载项目，点击应用配置并开始下载。

2. 配置 MinGW：为方便 VS2010 调用 FFmpeg 的动态库，可以通过 call “E:\Vs2010\VC\bin\vcvars32.bat”让 FFmpeg 编译时产生 Windows 下调用 DLL 对应的 lib。其中 call 对应的目录为 VC 安装程序的目录。

3. 下载配置 yasm。

4. 编译 x264：将下载好的 last_x264_tar_bz2 解压到 D:\ffmpeg\目录下，启动 MinGW32，切换到 x264 目录下后，再执行：

```
$./configure –enable-shared –disable-asm
$make
$make install
```

5. 编译 SDL：与 x264 类似，在 SDL 目录下进行编译工作。使用文本编辑工具打开 sdl-config 文件，并且把 prefix=/usr 改为 mingw 的安装路径。

6. 编译 FFmpeg：切换到 FFmpeg 解压目录下，执行编译配置命令。

在编译 FFmpeg 时，可以根据功能需求不同，对 FFmpeg 源码编码做出相应的裁剪，具体实现在于对参数选项的优化。其中，参数分为基本选项、高级选项以及特殊的优化参数。基本选项主要负责配置输出目录等基本信息，高级选项参数负责编译器的类别与网络功能等信息，优化参数主要对编译时不需要的功能模块以及文件适当裁剪和修改。部分优化参数如表 4-1 所示：

表 4-1 FFmpeg 编译优化参数说明

配置参数	参数功能
--disable-mpegaudio-hp	启动更快的 MPEG 音频解码功能（默认禁用）
--disable-protocols	禁用 I/O 协议支持（默认禁用）
--disable-ffserver	禁止生成 ffserver
--disable-ffplay	禁止生成 ffplay
--enable-small	启用优化文件尺寸大小
--enable-memalign-hack	启用模拟内存排列，由内存调试器干涉
--disable-strip	禁用剥离可执行程序和共享库
--disable-encoder=NAME	禁用 NAME 编码器
--enable-encoder=NAME	启用 NAME 编码器
--disable-decoder=NAME	禁用 NAME 解码器
--ensable-decoder=NAME	启用 NAME 解码器
--disable-encoders	禁用所有编码器
--disable-decoders	禁用所有解码器
--disable-muxers	禁用所有复用器
--disable-demuxers	禁用所有解复用器

在对各个优化参数的分析与功能研究后，使用下面的配置参数，此外，增加了对最新 H.265 编码标准的编译。

```
./configure --disable-static --enable-shared --enable-gpl --enable-version3 --disable-w32threads
--enable-avisynth --enable-bzlib --enable-fontconfig --enable-frei0r --enable-gnutls
--enable-iconv --enable-libass --enable-libbluray --enable-libcaca --enable-libfreetype
--enable-libgsm --enable-libilbc --enable-libmodplug --enable-libmp3lame
--enable-libopencore-amrnb --enable-libopencore-amrwb --enable-libopenjpeg --enable-libopus
--enable-librtmp --enable-libschroedinger --enable-libsoxr --enable-libspeex --enable-libtheora
--enable-libtwolame --enable-libvidstab --enable-libvo-aacenc --enable-libvo-amrwbenc
--enable-libvorbis --enable-libvpx --enable-libwavpack --enable-libx264 --enable-libx265
--enable-libxavs --enable-libxvid --enable-decklink --enable-zlib
```

以上参数也将会在 VS2010 环境配置测试中以对话框的形式反馈。

4.1.2 VS2010 下的快速配置

在 Visual Studio 开发环境下对第三方类库的使用通常需要加入头文件、静态库或动态链接库。课题采用 VS2010 作为集成开发环境，因此，在对 FFmpeg 编译完成后，需要将其加入链接到项目中。具体步骤如下

1. 拷贝编译好的 FFmpeg 开发文件，将头文件（*.h）拷贝至直播软件项目文件夹下的 include 子文件夹下，而静态库文件（*.lib）到子文件夹 lib 下，动态库文件（*.dll）拷贝至项目子文件夹根目录下。

2. 配置开发项目属性。在项目属性附加包含目录中添加 include 目录，在导入库配置中，添加库目录 lib，并在链接器附加依赖项中输入库文件名：avcodec.lib、avdevice.lib、avfilter.lib、avformat.lib、avutil.lib、postproc.lib、swresample.lib、swscale.lib、SDL2.lib。动态库不需要配置。

3. 配置环境测试。如果使用 C 语言，则直接包含目录即可。若是在 C++中使用 FFmpeg，则需要添加：

```
#define _STDC_CONSTANT_MACROS
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libavdevice/avdevice.h"
#include "SDL2/SDL.h"
#include "SDL2/SDL_thread.h"
#include "libavutil/opt.h"
#include "libavutil/imgutils.h"
#include "libavutil/pixfmt.h"
#include "libavutil/mathematics.h"
#include "libavutil/time.h"
#include "libavutil/audio_fifo.h"
#include "libswresample/swresample.h"
};
```

在完成后，使用 avcodec_configuration()函数打印出 FFmpeg 配置信息，用 AfxMessageBox()输出显示。如图 4.1 所示。

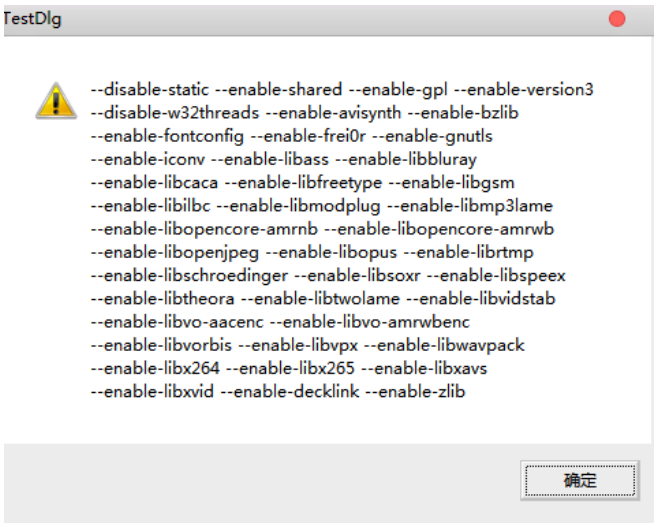


图 4.1 FFmpeg 配置信息显示图

4.2 基于 FFmpeg 的音视频数据采集实现

音视频数据采集功能实现的基本思想是利用 FFmpeg 中多媒体设备交互类库 libavdevice，读取计算机的多媒体信号并将数据输出到指定的多媒体设备上。在此模块中，输入设备为计算机显示器和音频设备，输出设备为解码模块入口，通过循环读取输入设备流数据帧，将读取到的音视频数据交给解码模块，完成音视频采集功能。图 4.2 展示了音视频处理中各个阶段之间的逻辑关联。

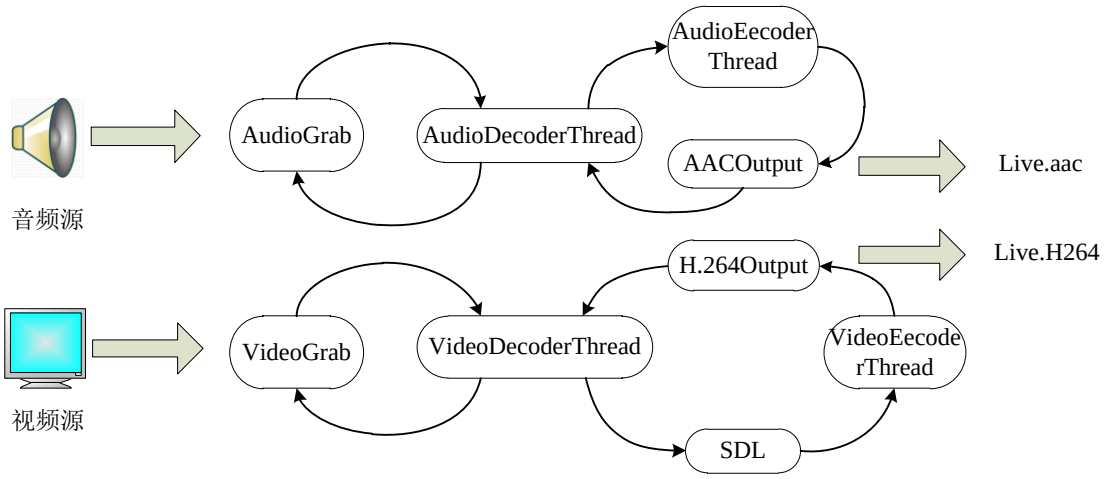


图 4.2　音视频采集逻辑图

4.2.1 音视频采集设备名获取

在采集音视频数据前，需要列出采集设备名称。在 Windows 平台下利用 FFmpeg.exe 在 cmd 下读取 DirectShow 设备数据。

```
ffmpeg -list_devices true -f dshow -i dummy
```

命令执行后的输出结果如图 4.3 所示。

```
[dshow @ 00000000004f2020] DirectShow video devices (some may be both video and audio devices)
[dshow @ 00000000004f2020]  "USB2.0 UVC VGA WebCam"
[dshow @ 00000000004f2020]     Alternative name "@device_pnp_\\?\usb#vid_13d3&pid_5710&mi_00#7&d3172
57&0&0000#{65e8773d-8f56-11d0-a3b9-00a0c9223196}\global"
[dshow @ 00000000004f2020]  "screen-capture-recorder"
[dshow @ 00000000004f2020]     Alternative name "@device_sw_{860BB310-5D01-11D0-BD3B-00A0C911CE86}\{
4EA69364-2C8A-4AE6-A561-56E4B5044439}"
[dshow @ 00000000004f2020] DirectShow audio devices
[dshow @ 00000000004f2020]  "榵~厠椋?(High Definition Audio 璁惧  )"
[dshow @ 00000000004f2020]     Alternative name "@device_cm_{33D9A762-90C8-11D0-BD43-00A0C911CE86}\
榵~厠椋?(High Definition Audio 璁惧  )"
[dshow @ 00000000004f2020]  "virtual-audio-capturer"
[dshow @ 00000000004f2020]     Alternative name "@device_sw_{33D9A762-90C8-11D0-BD43-00A0C911CE86}\{
8E146464-DB61-4309-AFA1-3578E927E935}"
```

图 4.3　采集设备名

中文设备名会出现乱码情况，尽管此时的 cmd 当前代码页数为 936（简体中文），

但与 codepage 中的代码页不一致，所以乱码情况依旧出现。

此时使用 chcp 65001 指令，设置为 UTF-8 代码页，便可以解决中文设备名乱码问题，如图 4.4 所示。

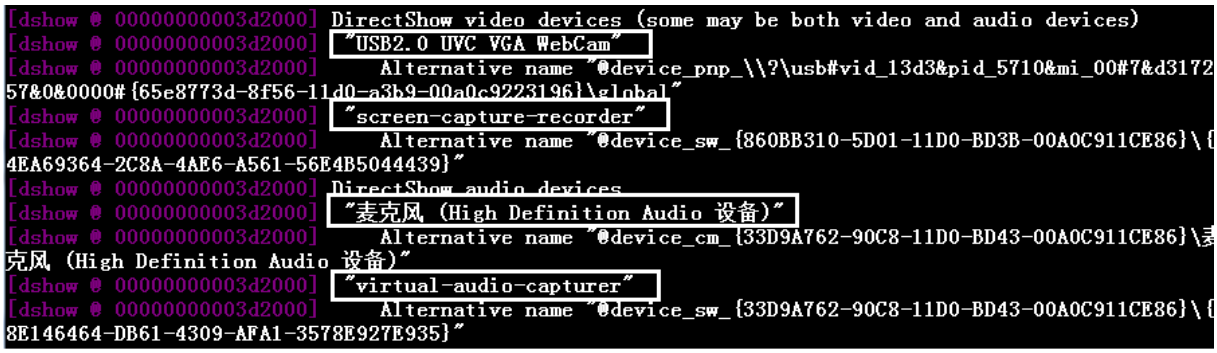


图 4.4 正常显示中文设备名图

若不熟悉 ANSI 转码 UTF-8 还可以使用 DirectShow SDK 自带的 GraphEdit 工具，通过插入过滤器 Audio Capture Sources 和 Video Capture Sources 来查看音视频输入设备的简体中文名称。

4.2.2 音视频采集速率控制

采样速率与帧率是音视频属性的一项重要参数，是播放控制、音视频编码、网络传输等功能实现的基础参数依据。一般情况下，较高的采集速率与帧率是音视频视听观感的质量直接体现，与此同时，在追求较高的刷新速率与采集速率的同时，对硬件的处理能力、编码器性能、软件结构等众多条件提出了更高的要求。以笔记本桌面画面视频序列为例，在每秒 50 帧速率下，画面分辨率为 1366×768 每秒所产生的数据为：6556800Byte，即每秒 6.5MB 的运算量对 CPU 与硬盘、内存等硬件设备产生了极高的负载。

因此，在音视频应用的各个领域中，根据不同的应用环境，对视频帧率与音频采集率做了明确规定与限制。例如电影的帧率控制在 23.976FPS，PAL 制电视领域采用 25FPS 的帧率，而 NTSC 制电视采用 29.97FPS 的帧率。在编码中的帧率往往采用分数的形式表示，因此就有了规定中帧率的小数存在。在音频方面，音频采样率的提高对声音的还原过程具有很大的提升。一般来说，音频采样频率划分为 22.05KHz 广播音质、44.1KHz 音频 CD 音质和 48KHz 音频 DVD 三个等级。

在本课题中，将采集模块在独立的线程中实现，根据直播系统需求与音视频相关基础知识，视频定为 24FPS，音频采样频率使用 44.1KHz。

视频帧率控制的实现是通过在获取桌面视频流前，通过 av_dict_set()函数将

framerate 字段设置为 24 帧。其函数声明如下：

int av_dict_set(AVDictionary **pm, const char *key, const char *value, int flags)

函数第一个参数是指向一个 AVDictionary 类型的选项指针，此指针保存了各种设置参数。例如帧率 framerate、分辨率 video_size、像素格式 pixel_format 等。第二个参数表示需要设置的参数名称，例如 framerate、video_size 等，第三个参数表示相应参数的设置数值。最后一个参数为标志位，一般设置为 NULL。

此外，在对于桌面视频流分辨率获取的实现上，可以不做设置，这样可以对不同分辨率下的桌面进行动态调整，通过分析视频流信息可以得到分辨率参数。音频的采样速率可以在系统音频设备属性中设置，如图 4.5 所示。

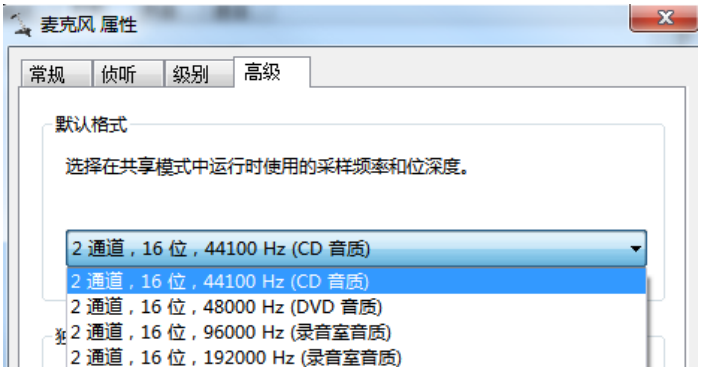


图 4.5　音频采集速率设置

4.2.3 音频数据采集

音频采集选择通过 dshow 输入设备格式进行捕获。这里需要将 wchar 类型的字符转换为 UTF-8 类型，转换函数调用 Win32 SDK 下的 WideCharToMultiByte()函数，核心代码如下：

```
char* dup_wchar_to_utf8(wchar_t *wbuffer)
{
    char *str = NULL;
    int maxlen = WideCharToMultiByte(CP_UTF8, 0, wbuffer, -1, 0, 0, 0, 0);
    str= (char *) av_malloc(maxlen);
    if (str)
        WideCharToMultiByte(CP_UTF8, 0, wbuffer, -1, str, maxlen, 0, 0);
    return str;
};
```

将转换后的输入设备名保存在自定义设备指针里：

1. 捕获麦克风设备输出：

```
char * psDevName = dup_wchar_to_utf8(L"audio=麦克风  (High Definition Audio  设备)");
```

2. 捕获计算机扬声器设备输出：

```
char * psDevName = dup_wchar_to_utf8(L"audio=virtual-audio-capturer");
```

音频采集流程与视频采集流程类似，将捕获到的音频数据读取到 AVPacket 结构体内，若读取失败则重新读取，若成功，则交由解码器。

4.2.4 视频数据采集

视频采集功能的实现，是将 gdigrab 作为一种输入格式指针，通过查找输入流信息来获取抓屏数据地址，判断是否为视频流数据。若捕获到视频流，则判断视频帧的位置，并根据捕获到的视频帧查找相应的解码器并打开，最后对捕获到的视频流进行循环读取数据，将读取到的视频帧保存在 AVPacket 结构体内，若视频帧读取成功，则交由解码器解码，失败则返回读取循环。如图 4.6 所示。

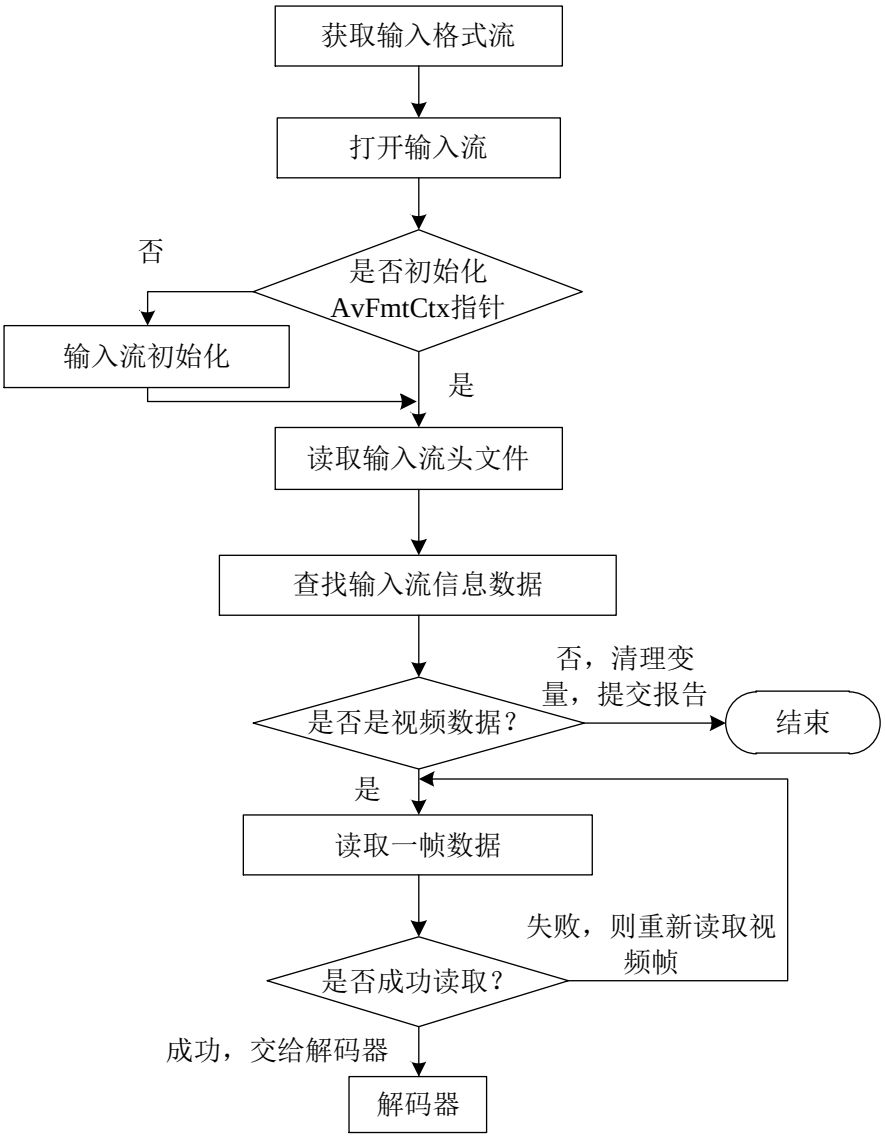


图 4.6　视频画面采集流程

其中，gdigrab 获取桌面视频流的原理是通过对计算机桌面画面的绘制，保存为 16 位 BMP 格式的图片，同时加上捕获中的相关信息，以每秒 24 张的图片流通过解码器处理，最后转换为 24 帧每秒的 YUV 视频流。

4.3 基于缓冲策略的音视频解码

音视频解码与编码分为两个独立线程同时工作，这样可以使帧率达到高清标准，同时增强了编解码流程的稳定性。解码后的数据通过 AVFifoBuffer 结构体缓存指针，写入到音视频缓存块中。在编码线程中，通过循环查询缓存块大小，读取音视频数据帧，进行编码工作。

4.3.1 客户端解码缓冲区设计

缓冲技术是目前音视频直播系统中被广泛采用的一种应用级音视频质量控制技术，通过缓冲区的设置实现，可以减少音视频数据的丢失，平滑编解码过程中的抖动，提高了直播系统的编解码性能与稳定性[26]。然而缓冲技术的采用同时对直播系统的实时性加入了少量的延时，因此对缓冲所带来的延时问题与系统的整体性能平衡是缓冲技术所面临的一个核心问题。

缓冲技术通常采用两种设计与实现方式，第一种是服务器缓冲，主要解决网络条件极端不理想情况下的数据丢包问题。第二种是客户端缓冲，通过内存划分的方式将数据写入到缓存队列中，在编解码模块中对队列数据量进行合理控制，进而对直播的视频流进行平滑处理，保证乱序的数据包恢复正常的编码与传输顺序。在设计与实践过程中，本课题将服务器缓冲技术的设计思想应用到客户端选择发送功能模块中去，再配合以客户端缓冲技术，达到了良好的实验效果。

视频缓冲区与音频缓冲区设计原理类似，这里以音频缓冲区为例作以解释说明。本课题解码后的 PCM 音频数据格式为 44.1KHz 的双声道 16 位立体声，即一秒的音频数据内含有 44100 个采样数据。通常，一帧 PCM 的音频数据内含有 2048 个采样数据（双声道），而一帧 AAC 音频数据内包含 1024 个采样数据。双声道音频文件采样数据通常按照时间的先后顺序交叉存入到内存块中。因此，为了配合音频的 AAC 编码与低延时特性，音频缓冲区的容量设计大小为 22050 个采样数据，其中包含一字节的音频信息头，理论上最差延时为 500ms，而系统实际运行中，通过对缓冲区的容量控制与音视频编码循环，在缓冲区未满时就通过读取数据来降低延时，读取最小缓冲区数据大小控制在一

帧 AAC 采样数据的大小之上，即通过 if(av_audio_fifo_size(fifo_audio) >= 1024)语句判断音频缓冲区中数据存储情况，再进行音视频编码或者等待缓存数据写入。缓冲区具体结构如图 4.7 所示。

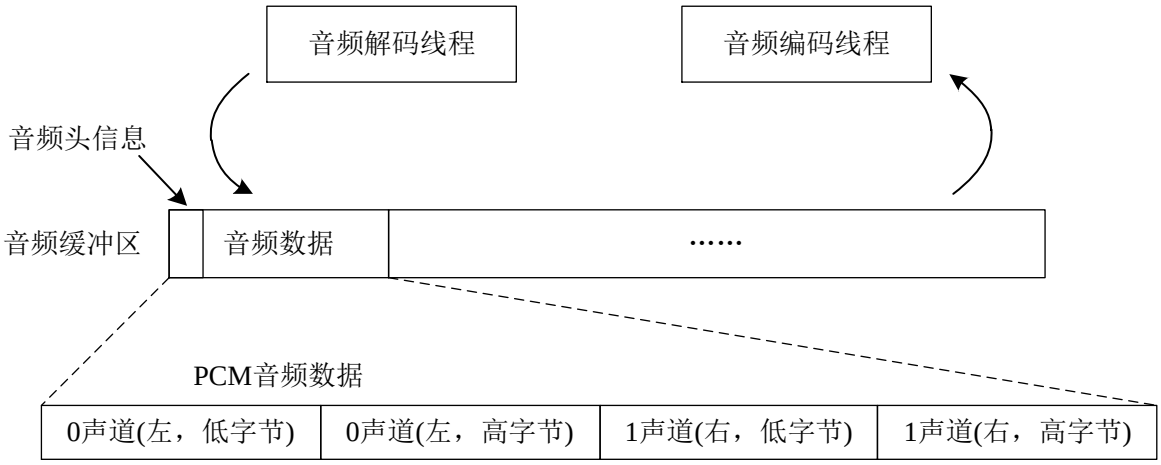


图 4.7 音频缓冲区

视频缓冲区设计与音频缓冲区设计类似，这里不再赘述。服务器端的缓冲技术通过网络传输缓冲等待实现，具体实现细节在网络传输实现小节进行详细阐述。

4.3.2 视频解码器框架

解码器的流程框架与编码器形成互补结构，当编码码流进入解码器时，解码器通过对接收到的压缩数据进行熵解码重排序，得到宏块的残差系数以及量化系数等相关信息 X，对其进行反量化、反变换操作得到残差宏块 Dn' ，然后根据码流中的头信息进行预测块 P 重建，当预测块 P 与残差宏块 Dn' 相加后得到 uF_u' ，经过去块效应滤波便得到最后解码输出的图像 Fn' [27]。如图 4.8 所示。

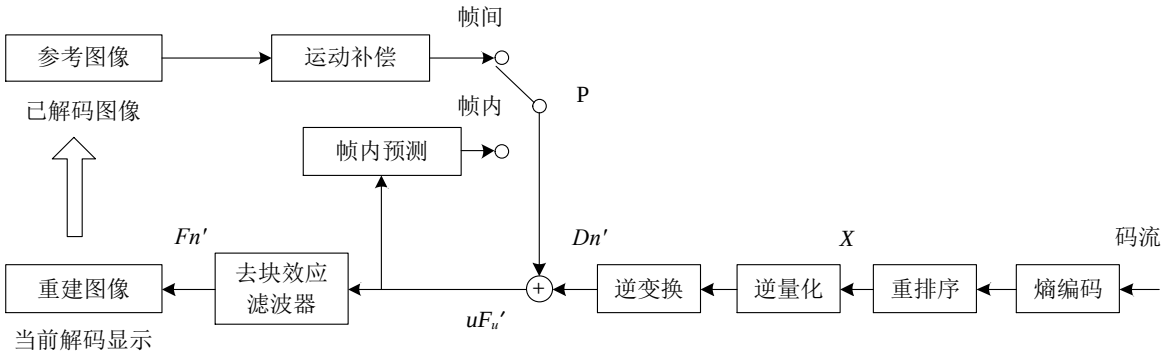


图 4.8 解码原理框架

4.3.3 视频解码

视频解码功能的实现首先通过avformat_find_stream_info()函数获取一部分视频流数据，从中捕获一些相关的音视频信息。通过给每个媒体流的 AVStream 结构体赋值，实现了解码器的初步查找、打开、音视频帧的读取等功能。

在获取视频对应的帧信息后，向系统申请解码帧缓存，这里设置为 24 帧解码视频帧大小的缓存，并对缓存块的像素空间格式，帧尺寸，分辨率等参数进行设置。最后，利用记录到的帧位置来确定相应的解码器 ID，并进入到初始化解码器流程，对解码结构体缓存进行赋值初始化、分配内存空间，进而打开解码器，读取视频流，开始循环解码过程。

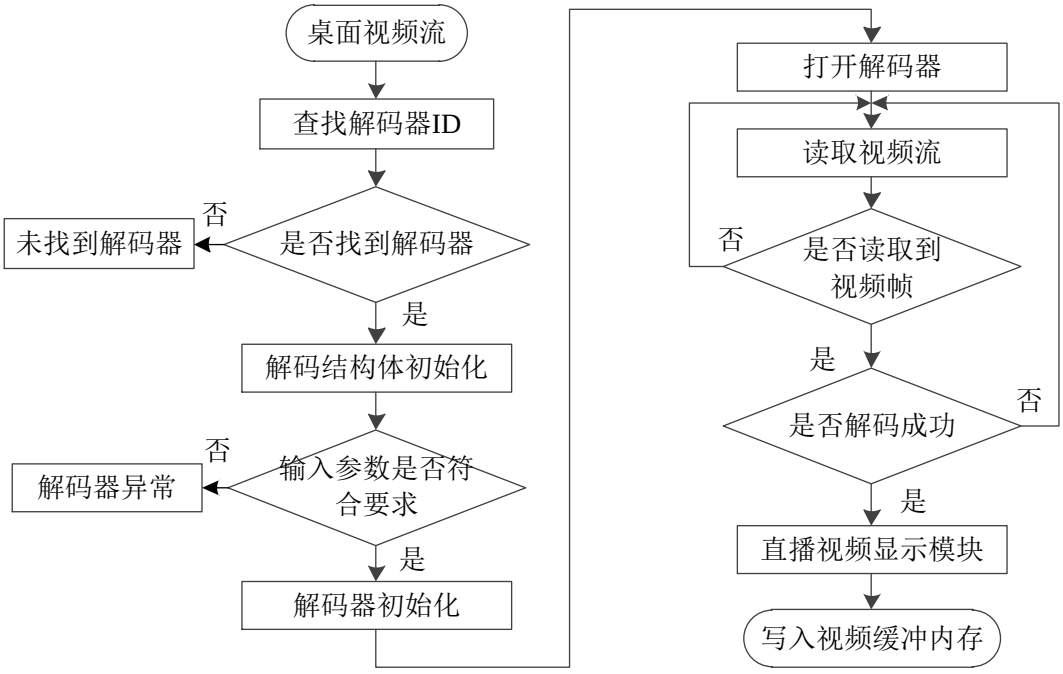


图 4.9　视频解码流程图

视频解码前，读取到的视频流数据保存在 AVPacket 结构体内，结构体数据成员包含显示时间戳、解码时间戳、音视频标识等信息。具体成员和解释如下：

```
typedef struct AVPacket {
    int64_t pts;                    //显示时间戳
    int64_t dts;                    //解码时间戳
    uint8_t *data;                  //压缩编码数据
    int     stream_index;           //标识该 AVPacket 所属的视频/音频流
    struct {
        uint8_t *data;
        int          size;
```

```
        enum AVPacketSideDataType type;
    } *side_data;
    int     duration;                   //时长信息
} AVPacket;
```

解码后，将解码出的 YUV 数据存储在 AVFrame 结构体内。AVFrame 包含了较多的解码信息与结构成员变量，如 data[]中存储打包的数据或平面信息，可以直接用 fwrite 函数将 data 中的 YUV 数据写到本地文件内。而 pict_type 标识图像的帧类型，如关键帧 I、预测帧 P 等等。sample_aspect_ratio 表示视频图像的宽高比，以分数形式呈现。

AVFrame 中的 QP 表存储了内存中视频图像宏块的 QP 值，其标号按行由左向右开始。其中 qscale_table[1]的 1 代表第一行中第一列图像宏块的 QP 值，而 qscale_table[3]则代表第一行中第三列图像宏块的 QP 值，因为宏块的大小通常是16×16的，因此对于每行宏块数的计算公式为[28]：

$$m_{stride} = width \div 16 + 1 \tag{4.1}$$

而宏块的总数为：

$$m_{sum} = ((height + 15 >> 4) \times width \div 16 + 1 \tag{4.2}$$

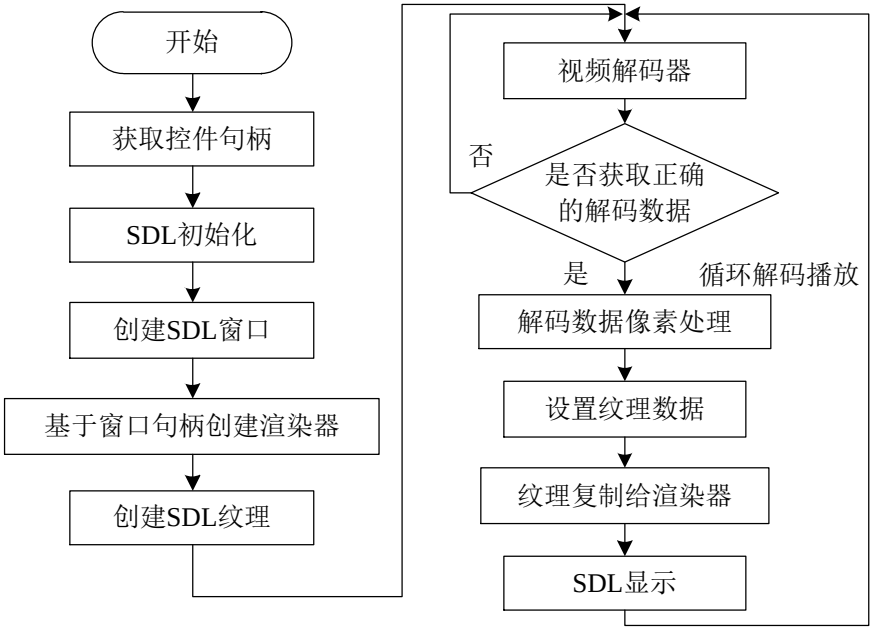
4.3.4 视频反馈窗口

视频反馈窗口，是嵌入软件主界面上的播放窗口，通过 Picture 控件，改变控件框架为图片，在直播器未工作时，显示一张自定义图片，在工作时，将控件句柄通过线程函数的 lpParam 传递到解码显示模块，通过 CreateWindowFrom()函数在控件上创建播放显示窗口，最后，通过 SDL 创建纹理，进行渲染更新操作，在界面上呈现实时直播画面。显示模块部分核心代码如下：

```
if(SDL_Init(SDL_INIT_VIDEO))
{
    AfxMessageBox( "SDL 初始化失败",MB_OK,0);
    return -1;
}
HWND ScreenHwnd=((CPARAM*)lpParam)->ScreenHwnd; //Screen hwnd;
SDL_Window *screen;
screen=SDL_CreateWindowFrom(ScreenHwnd);
if(!screen)
{
    AfxMessageBox("SDL: 创建播放窗口失败",MB_OK,0);
    return -1;
}
```

```
sdlRenderer = SDL_CreateRenderer(screen, -1, 0);
sdlTexture = SDL_CreateTexture(sdlRenderer, SDL_PIXELFORMAT_IYUV,
SDL_TEXTUREACCESS_STREAMING,pCodeCtx_Decode->width,pCodeCtx_Decode->height);
……
SDL_UpdateTexture( sdlTexture, NULL, pFrameSws->data[0], pFrameSws->linesize[0] );
SDL_RenderClear( sdlRenderer );
SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, NULL);
SDL_RenderPresent( sdlRenderer );
```

客户端视频反馈具体流程如图 4.10 所示



4.10 SDL 显示流程图

在直播视频显示流程中，首先通过线程函数的 lpParam 获取到传递进来的主界面播放器窗口的句柄，对 SDL 模块进行初始化后，使用 SDL 类库中的创建窗口函数 CreateWindowFrom()将播放模块与句柄进行绑定。在播放循环前对 SDL 相关结构体变量进行初始化，当解码器成功输出一帧视频图像时，通过对 SDL 纹理与渲染器进行更新，最后对视频数据完成播放。

4.3.5 音频解码

音频解码单元通过音频流信息而查找音频解码器，在成功找到解码器 ID 后初始化音频解码结构体单元，用来存放解码后的音频数据。解码前的准备工作完成后，进行解码器的初始化，打开解码器进行解码循环[29]。最后，解码成功后的数据存入到解码结构体单元内，再由音频解码缓冲区读入到内存块中，通过线程间数据同步完成解码单元与

编码单元的数据通信。解码流程如图 4.11 所示。

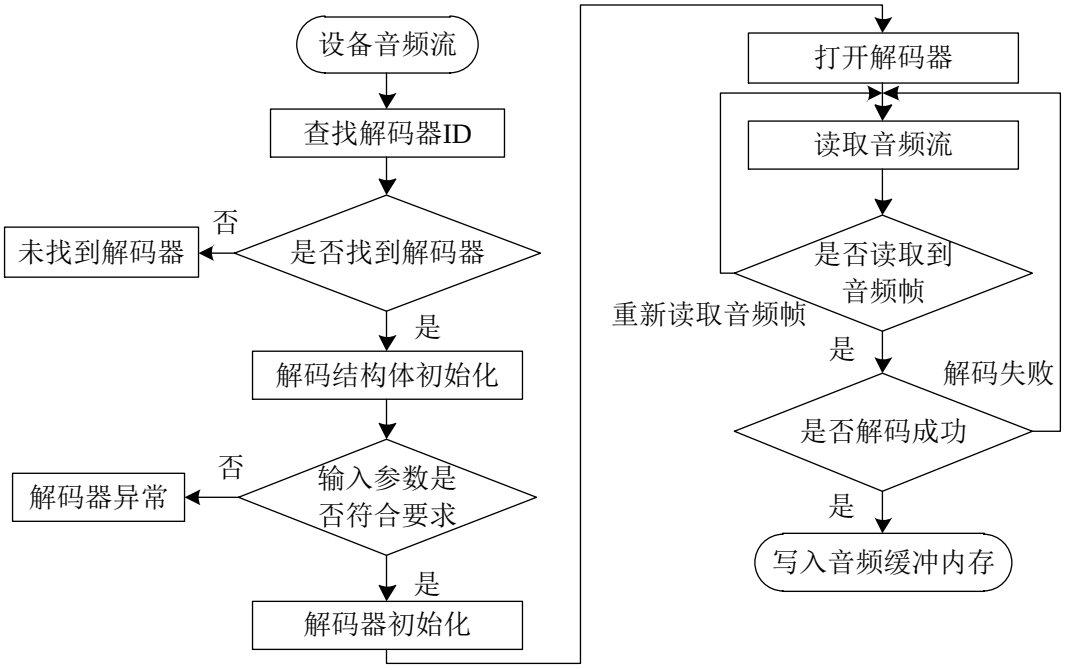


图 4.11 音频解码流程

由于采用了最新的 FFmpeg 类库，新版本中的音频解码器方面发生了较大的变化，新版中 avcodec_decode_audio4() 函数在解码后输出的音频采样数据格式为 AV_SAMPLE_FMT_FLTP(float,planar)，而不再是 AV_SAMPLE_FMT_S16(signed 16 bits)，因此会有杂音现象发生。最后采用的解决方法是使用 SwrContext 对音频采样数据进行重采样，再写入到 AVAudioFifo 缓存结构体中。

音频 Resample 重采样部分核心代码如下：

1. 配置音频重采样各项参数。

```
uint64_t    out_channel_layout=AV_CH_LAYOUT_STEREO;
int         out_nb_samples=pCodeCtx_aac->frame_size;
int         out_sample_rate=44100;
int         out_channels=av_get_channel_layout_nb_channels(out_channel_layout);
int         out_buffer_size=av_samples_get_buffer_size(NULL,out_channels,
out_nb_samples,out_sample_fmt, 1);
AVSampleFormat      out_sample_fmt=AV_SAMPLE_FMT_S16;
uint8_t       *out_buffer=(uint8_t *)av_malloc(MAX_AUDIO_FRAME_SIZE*2);
int64_t         in_channel_layout=
av_get_default_channel_layout(pCodeCtx_aac->channels);
```

2. SwrContext 初始化。

```
struct SwrContext    *au_convert_ctx = swr_alloc();
```

3. 以 AVoption 方式设置重采样上下文

```
//设置输入源的通道布局
av_opt_set_int(au_convert_ctx, "in_channel_layout",       src_ch_layout, 0);
//设置输入输出采样率
av_opt_set_int(au_convert_ctx, "in_sample_rate",           src_rate, 0);
av_opt_set_int(au_convert_ctx, "out_sample_rate",          dst_rate, 0);
//设置输出源的采样格式
av_opt_set_sample_fmt(au_convert_ctx, "out_sample_fmt", dst_sample_fmt, 0);
```

4. 参数设置

```
au_convert_ctx=swr_alloc_set_opts(au_convert_ctx,out_channel_layout,
out_sample_fmt,out_sample_rate,in_channel_layout,pCodeCtx_aac->sample_fmt,
pCodeCtx_aac->sample_rate,0, NULL);
swr_init(au_convert_ctx);
```

5. 对 AVFrame 进行重采样。

```
swr_convert(au_convert_ctx,&out_buffer, MAX_AUDIO_FRAME_SIZE,
                        (const uint8_t **)pFrame->data , pFrame->nb_samples);
```

4.4 音视频编码实现

一般情况下，音视频的编解码任务对硬件设备造成了较高负担，从而导致音视频采集数据与编解码数据帧率不匹配、序列帧混乱、时间信息错误等诸多问题，因此，课题采用音视频解码与编码多线程方式规避此问题。编码模块流程如图 4.12 所示。

首先选择相应的编码器并设置相关视频编码参数与音频编码参数，将设置好的信息通过 avcodec_find_encoder()函数查找到相对应的音视频编码器，在打开编码器后填充音视频数据区域格式，在设置好相对应的时间基准后，比较音视频时间信息得到此刻应进行视频编码还是音频编码，从而进入到对应的编码流程。

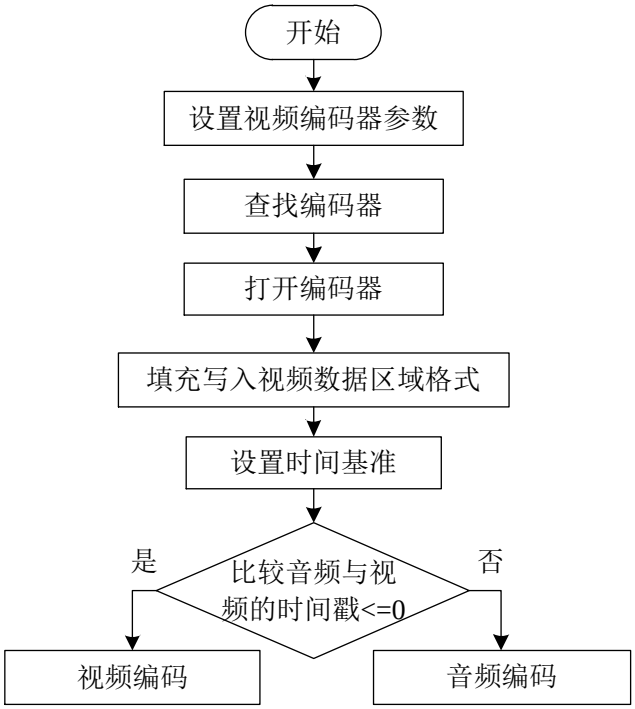


图 4.12　音视频编码模块流程

4.4.1 视频 H.264 编码框架

H.264 编码器采用前向编码与后向重建的方式，通过运动预测与整数变换的混合编码对输入的视频序列帧进行编码。

在前向编码分支中，当前帧 *Fn* 以宏块为单位进行帧内或帧间预测编码。在帧内编码模式下，重建块 *Fn′* 通过去块效应滤波后的 $\mu F_u'$ 得到，预测宏块 P 通过当前已编码块与重建块预测得到。在帧间模式下，首先通过运动估计与运动补偿得到宏块 P，经过与当前帧 *Fn* 的宏块相减得到残差 *Dn*，接着对残差块进行整数变换、量化、重排序以及熵编码完成对宏块的编码并通过 NAL 层存储或传输[30]。

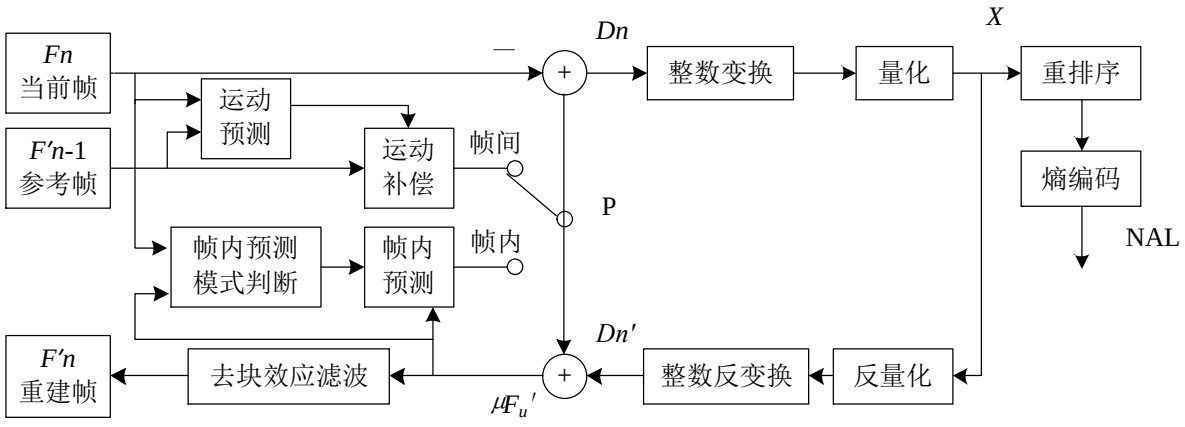


图 4.13　H.264 编码框架图

后向重建的主要意义在于提供进后续宏块预测编码的参考帧 *Fn-1′* 宏块。首先，由于有损量化过程，宏块系数 *X* 经过反量化与整数反变换后得到残差宏块 *Dn* 的近似值 *Dn′*[31]。预测块 P 与 *Dn′* 相加得到未滤波重构宏块 *μF_u′* ，通过减少块效应而最终得到重构宏块 *Fn′* ，对图像宏块进行重建后可得到重建图像[32]。

4.4.2 视频 H.264 编码实现

NAL 网络提取层通过以 NALU 为单元来支持 H.264 编码数据在网络中传输与存储，而 NALU 是由编码后的 H.264 视频序列得到。一个 NALU 包含 NAL 头信息与来自 VLC 的 RBSP（Raw Byte Sequence Payloads）原始字节序列载荷两部分。其中，NAL 头信息为一字节，包括三个固定字段，用于说明 NALU 的优先级参数与类型。RBSP 是有关编码图像的重要信息，其可以是压缩数据，也可以是参数集等相关附加信息。具体结构如图 4.14 所示。

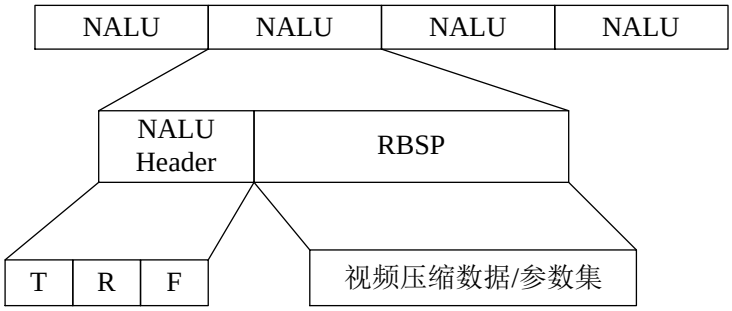


图 4.14　NALU 结构图

图中 T 代表 NALU 类型位，共有 32 种类型，其中 1~12 通常为 H.264 使用。R 代表重要性指示位，标识 NALU 的重要性，其中 00 表示该 NALU 的不重要信息，11 则表示该 NALU 信息很重要。F 代表禁止位，在 H.264 编码中该值通常被置为 0。参数集分为两种：序列参数集 SPS 和图像参数集 PPS，分别代表包含一个图像序列的所有信息与包含一个图像所有片的信息。在 FFmpeg 中的 H.264 视频编码实现具体流程如图 4.15 所示。

为了使音视频的时间信息与帧序列不发生混乱，因此，将视频编码与音频编码合并为同一线程中操作。在音视频编码线程中，首先根据 H.264 与 AAC 的时间基准和当前时间戳信息判断进入视频编码循环或者音频编码循环。其中 H.264 的时间基准设置为 AVRational video_timebase={ 1, 90000}，为分数形式。

经过 av_compare_ts()函数判断到此时应该进入到 H.264 视频编码循环后，首先通过 if(av_fifo_size(fifo_video)>=size_video)对视频解码缓冲区进行查询判断，当缓冲区中的

解码数据不足以编码出一帧视频时，则跳出视频编码循环，进行视频或者音频编码选择。当缓冲区内存有足够的视频解码数据，则进入 H.264 视频编码流程。首先使用 av_fifo_generic_read()函数读取缓存区的解码数据，保存在 picture_buf 中，再将申请的编码内存块参数设置成相应的编码格式，进行编码操作。在编码完成后，判断视频帧是否包含 DTS 或者 PTS 等相关信息，通常情况下 H.264 裸流视频序列是没有相关信息的。因此，在此处应根据时间基准换算算法写入对应的时间信息与时间间隔信息，完成后写入到输出设备上下文中。

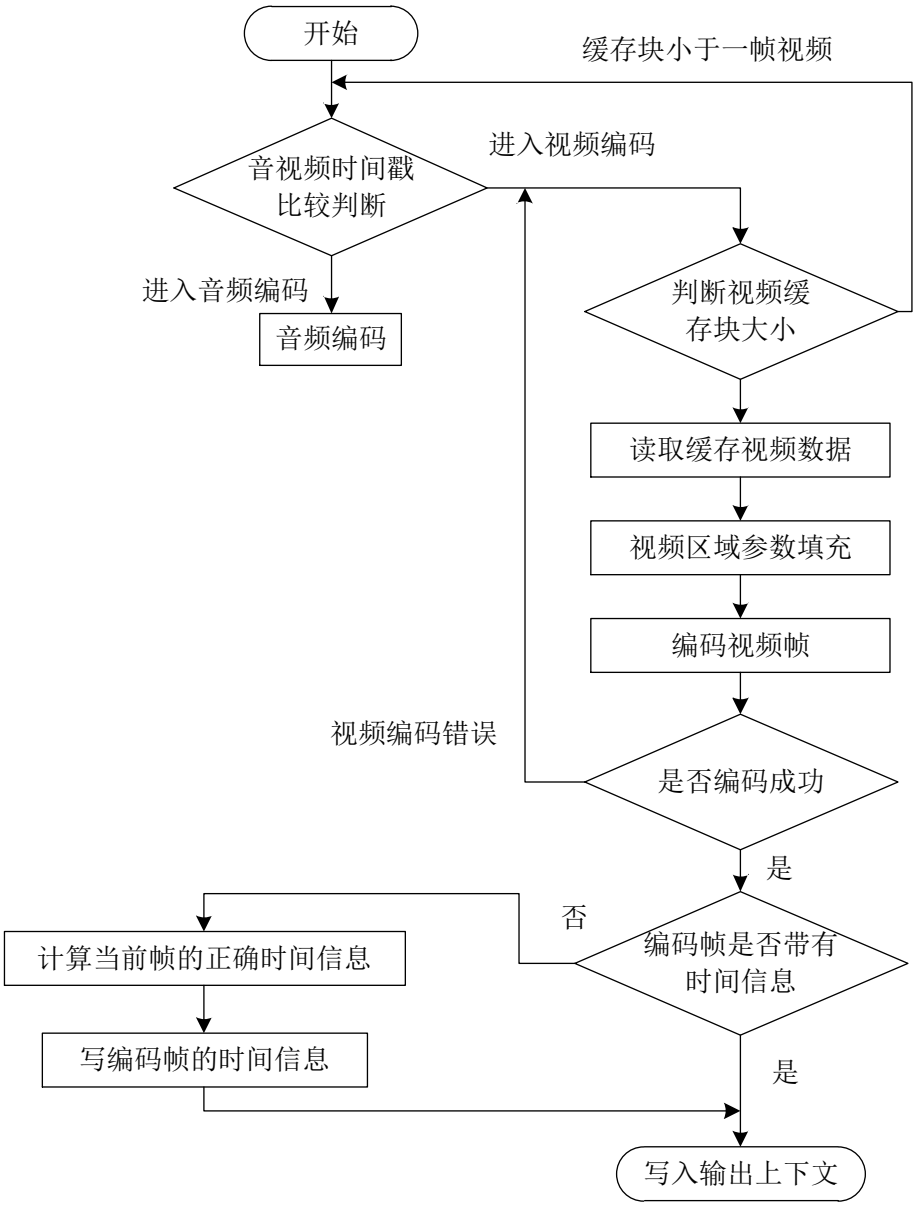


图 4.15　H.264 视频编码实现流程图

4.4.3 音频 AAC 编码实现

AAC 高级音频编码格式分为两种，一种为 ADIF 音频数据交换格式（Audio Data Interchange Format），另一种为 ADTS 音频数据传输流（Audio Data Transport Stream），其中 ADTS 更具备音频流的特征，其支持一个有同步字的比特流，因而在解码过程中可以在流的任意位置开始，在直播系统中的传输与处理方面的支持性更高[33]。其具体的帧结构如图 4.16 所示。

……	syncword	header()	error_check()	raw_data_block()	……

图 4.16　AAC 帧结构图

ADTS 的信息头部大小是可变的，其固定头信息为 7 字节。其中 syncword 是长度为 12bit，所有位为 1 的同步字段，用来标识视频序列比特流中的帧头位置信息与结束位置信息。头信息分为两部分，前部分为数据信息固定头信息，另一部分为帧间不固定的可变头信息。音频 AAC 编码流程如图 4.17 所示。

在音频编码流程中，首先通过 AVRationl 结构体设置 AAC 的时间基准与帧率。AVRationl audio_timebase={ 1, 44100}，时间基准为分数形式。音频帧率设置为 AVRationl audio_r_frame_rate={ 1024, 1}，每帧为 1024 个采样数据。

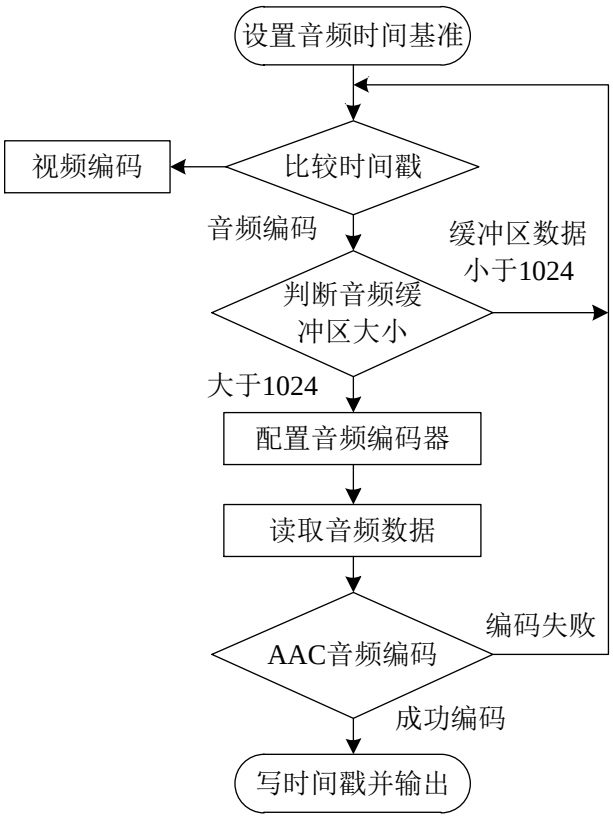


图 4.17　AAC 音频编码流程

进入编码循环后，通过比较时间戳信息 av_compare_ts()函数后进入音频编码流程，查询音频缓冲区大小是否足以编码一帧音频，通过 av_audio_fifo_size()函数对 1024 进行返回值比较。若大于等于 1024 个采样数据，则对音频编码结构体进行初始化，在对音频编码器完成配置后进行 AAC 编码。

4.5 本章小结

本章主要完成了直播软件音视频处理技术的实现。首先对 FFmpeg 编译移植和软件开发环境的配置进行阐述说明，接着对直播软件中音视频数据采集、解码、编码等关键技术进行实现，并且在对实现过程中的新技术进行详细阐述，包括对音视频数据采集速率控制、音视频解码缓冲区设计与实现、音频重采样实现、单独线程的编解码实现等。

第五章 高清实时直播服务器搭建与数据传输实现

5.1 基于 Nginx 流媒体服务器搭建

Nginx 是由俄罗斯软件工程师 Igor Sysoev 使用 C 语言从头开发的免费开源 web 服务器软件。于 2004 年发布，聚焦于多种 web 服务器功能特性：负载均衡、缓存、访问控制、宽带控制。并且其拥有的较高稳定性、支持高并发连接、内存消耗少和功能丰富等众多优点而被全世界越来越多的人关注。

5.1.1 高并发的重要性

由于 web 服务的兴起，并发问题逐渐成为网站与服务器架构的最大挑战。在早期，并发问题产生的主要原因在于客户端的接入速度比较慢，而如今，大量的多媒体数据或图片数据消耗着服务器资源，通常大型网站采用分离图片或多媒体服务器的方式来规避这样的风险。然而，来自移动端和新型的应用架构，通常以持久连接更新等服务对网站服务器架构提出了更高的要求。

以目前互联网网站占有率第一的 Apache（阿帕奇）架构为例，一个网站产生出小于 100KB 的响应，而对于生成页面的生成可能耗费 1 秒钟，最后将整个页面送往客户端后则需要 10 秒后才可以将连接关闭。因此，随着持久连接的大量应用产生的并发处理问题变得日益严重，为了处理持续增长的用户负载与更高数量级的并发任务，除了提升更高效的服务器组件、更高的网络带宽等基础设备，web 服务器性能也应该能够支持跳跃性的扩展升级。

在应对并发问题上，Nginx 没有对服务请求建立新进程，而是基于事件模型。传统基于进程或者线程的模型，会因为并发连接的处理而阻塞网络或者 I/O 操作，不仅如此，包括内存分配与初始化，需要消耗额外的 CPU 时间。因此，Nginx 大量的使用多路复用与事件通知，并且按类型给不同的进程分配不同的任务，使用高效的单线程循环处理连接，使其在普通硬件上就可以处理较高的并发连接。

5.1.2 Nginx 架构分析

Nginx 模块化的架构允许开发者在不修改核心的前提下自由扩展 web 服务器的功能。其主要可分为核心模块(core)、事件模块(event)、协议模块(HTTP)、负载均衡器等，各个模块代码与 Nginx 的核心代码一起编码。此外，核心模块同时封装了对内存池的实现、

网络连接处理、自旋锁等操作，对应的源码实现分别在 ngx_spinlock.c、 ngx_palloc.c、ngx_connection.c 下。图 5.1 展示了 Nginx 架构设计。

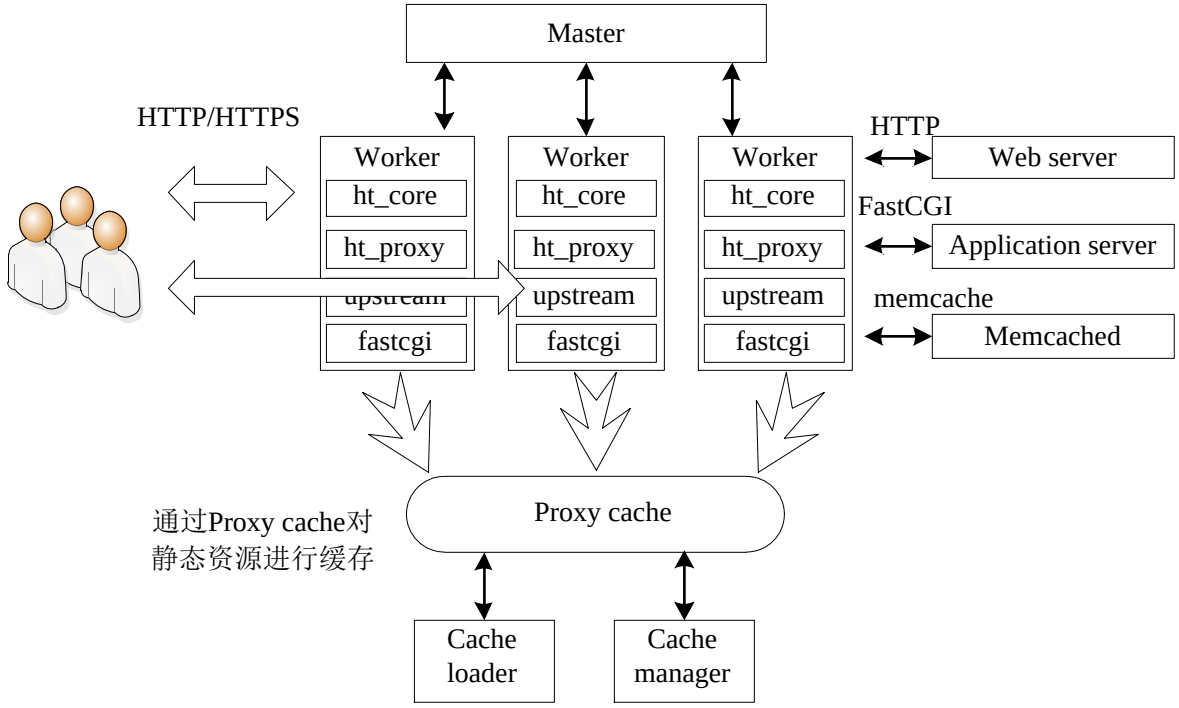


图 5.1 Nginx 高层架构设计

通过对 Nginx 架构分析了解到，其监听套接字在 Nginx 启动阶段完成初始化，启动后的程序会包括一个 Master 进程和多个 Worker 进程，Master 负责对外界信号或者请求的接收，通过 Worker 进程使用这些套接字来接受和读取请求，并做出输出响应，通过高效的事件处理循环来完成对大量连接的处理。标准的 HTTP 或 HTTPS 模块作为 Nginx 的常用模块来实现对静态文件、反向代理、远程 FastCGI 服务的实现[34]。同时支持 HTTP 下的 SSL 安全协议。

5.1.3 Nginx 配置文件

配置文件是 web 服务器可扩展性的重要组成，日常的维护通常耗费大量的人力物力，因此，Nginx 以简化维护为目的而重新设计了 C 风格的配置文件。

Nginx 的配置文件名通常为 nginx.conf 的文本文件，支持正则表达式，并且按区域划分，分别为全局区、事件区、HTTP 区、server 区以及 location 区等。各配置区的说明如图 5.2 所示。

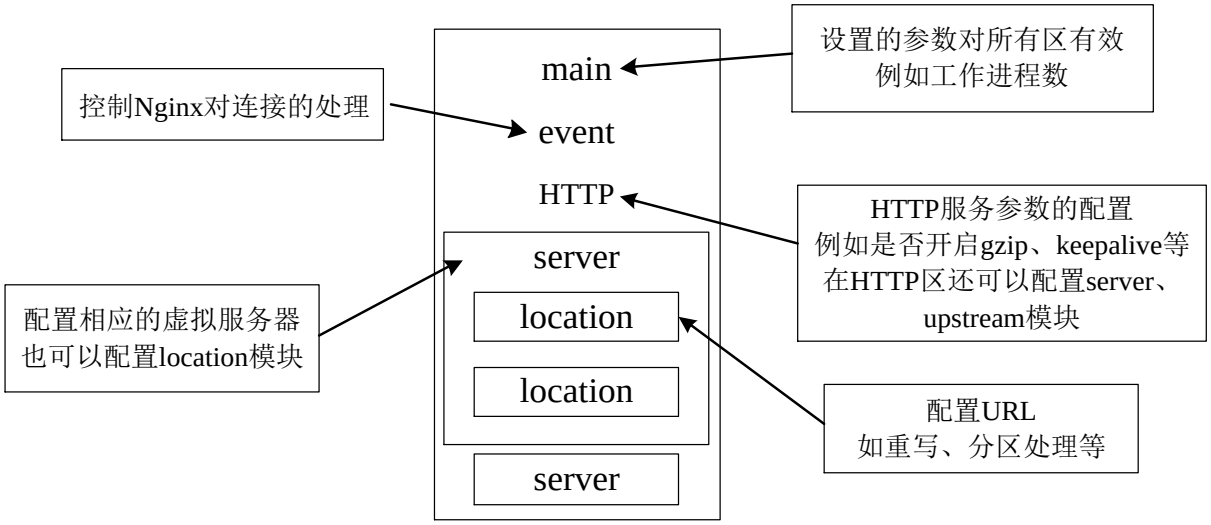


图 5.2　Nginx 配置文件分区说明图

5.1.4 基于 Nginx 的 RTMP 流媒体服务器搭建

Nginx 本身就是一个拥有较高稳定性、支持高并发连接、内存消耗少且功能丰富的 HTTP 服务器，其较多优点被全世界越来越多的人关注。而 FFmpeg 又是非常优秀的开源音视频技术解决方案，因此，本课题结合 FFmpeg，采用基于 RTMP 的 Nginx 服务器作为实时高清直播系统的流媒体服务器。

（1）下载 Nginx 所需要的模块安装包

1.gzip 模块需要 zlib 库

2.rewrite 模块需要 pcre 库

3.ssl 功能需要 openssl 库

下载下来的文件如下图所示：

```
[root@RedHat6 video]# ls -al
总用量 5448
drwxrwxrwx. 2 root  root     4096 3月   10 14:53 .
drwxr-xr-x. 6 root  root     4096 3月   10 14:53 ..
-rwxr--r--. 1 samba samba  910812 12月  31 10:39 nginx-1.10.2.tar.gz
-rwxr--r--. 1 samba samba  545844 12月  31 10:29 nginx-rtmp-module-master.zip
-rwxr--r--. 1 samba samba 1474795 12月  31 10:58 openssl-fips-2.0.13.tar.gz
-rwxr--r--. 1 samba samba 2053336 12月  31 11:02 pcre-8.38.tar.gz
-rwxr--r--. 1 samba samba  571091 12月  31 11:15 zlib-1.2.8.tar.gz
[root@RedHat6 video]#
```

图 5.3　模块安装包

（2）安装上面三个依赖包

依赖包安装顺序依次为：openssl、zlib、pcre，然后安装 Nginx 包

```
openssl：
[root@RedHat6 openssl-fips-2.0.13]# tar -xvzf openssl-fips-2.0.13.tar.gz
[root@RedHat6 openssl-fips-2.0.13]# cd openssl-fips-2.0.13
[root@RedHat6 openssl-fips-2.0.13]# ./config && make && make install
pcre:
[root@RedHat6 pcre-8.38]# tar -xvzf pcre-8.38.tar.gz
[root@RedHat6 pcre-8.38]# cd pcre-8.38
[root@RedHat6 pcre-8.38]# ./configure && make && make install
zlib:
[root@RedHat6 zlib-1.2.8]# tar -xvzf zlib-1.2.8.tar.gz
[root@RedHat6 zlib-1.2.8]# cd zlib-1.2.8
[root@RedHat6 zlib-1.2.8]# ./configure && make && make install
```

（3）配置并编译 Nginx

使用 Nginx 所默认提供的服务器配置并添加 rtmp 模块。

```
[root@RedHat6 nginx-1.10.2]# ./configure --add-module=../nginx-rtmp-module-
master
[root@RedHat6 nginx-1.10.2]# make
[root@RedHat6 nginx-1.10.2]# make install
```

(4)运行测试 Nginx

进入安装目录/usr/local/nginx/sbin，运行命令./nginx

```
[root@RedHat6 sbin]# ./nginx
[root@RedHat6 sbin]#
```

图 5.4　运行命令

打开浏览器在地址栏输入服务器的 IP 地址，如图 5.5 所示成功地搭建了 Nginx 服务器搭。

图 5.5　服务器搭建成功

5.1.5 流媒体服务器点播配置

在搭建好的服务器上对配置文件 nginx.conf 进行视频点播服务配置。首先对配置文件进行端口号、数据块大小进行设置。在 application vod 字段中添加视频存放位置，并在 application vod_http 添加视频点播源地址。点播服务配置成功后，在视频存放位置目录里放置了一个 ido.mp4 的播放文件。如图 5.6 所示

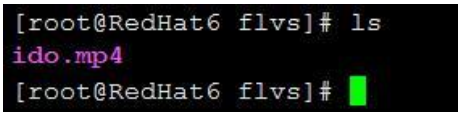


图 5.6　点播存放的视频文件

文件放好之后，重新启动 nginx。

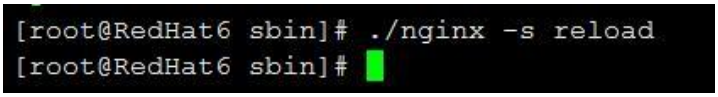


图 5.7　重启 nginx

打开测试所选用的视频播放软件 VLC media-> open network stream。填写需要点播的节目地址 rtmp://localhost/vod/ido.mp4，如图 5.8 所示。

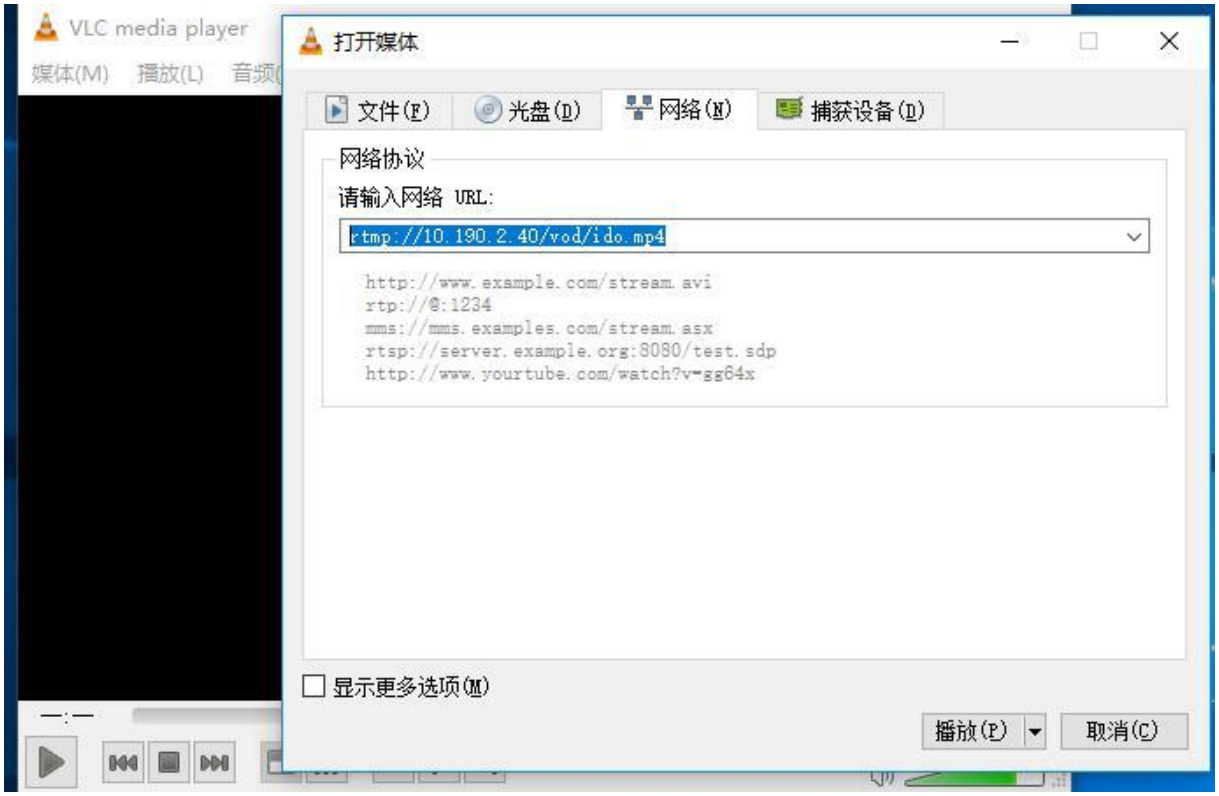


图 5.8　填写点播测试地址

点击 Play 就可以完成播放，点播功能测试成功如图 5.9 所示。

图 5.9 视频点播测试

5.1.6 流媒体服务器直播配置

在点播服务配置完成后，开始对直播服务的配置。

首先对并发数的设置。在安装好 Nginx 后，默认的最大并发数为 1024，如果想提升并发数，可以通过修改 *worker_connections* 来扩大，数值越大并发数也就越大。当然需要按照实际的需求情况进行配置，当数值设置过高时会加大对 cpu 的负载，因此按照反向代理模式下最大连接数的理论计算公式：

$$\text{最大连接数}=worker\_connections \times worker\_processes \div 4 \tag{5.1}$$

接着对 location 字段进行配置，部分代码如下：

```
location /stat {
                rtmp_stat all;
                rtmp_stat_stylesheet stat.xsl;
               }
 location /stat.xsl {
                root /usr/local/nginx/nginx-rtmp-module/;
               }
```

最后，对于直播服务，需要在 server 字段添加应用名，这里设置为 live，提供多个频道直播的开启。开启 live、live2 两个频道之后，重新启动 nginx 并打开浏览器，就会看到 live 与 live2 字段，证明配置已经生效，如图 5.10 所示。

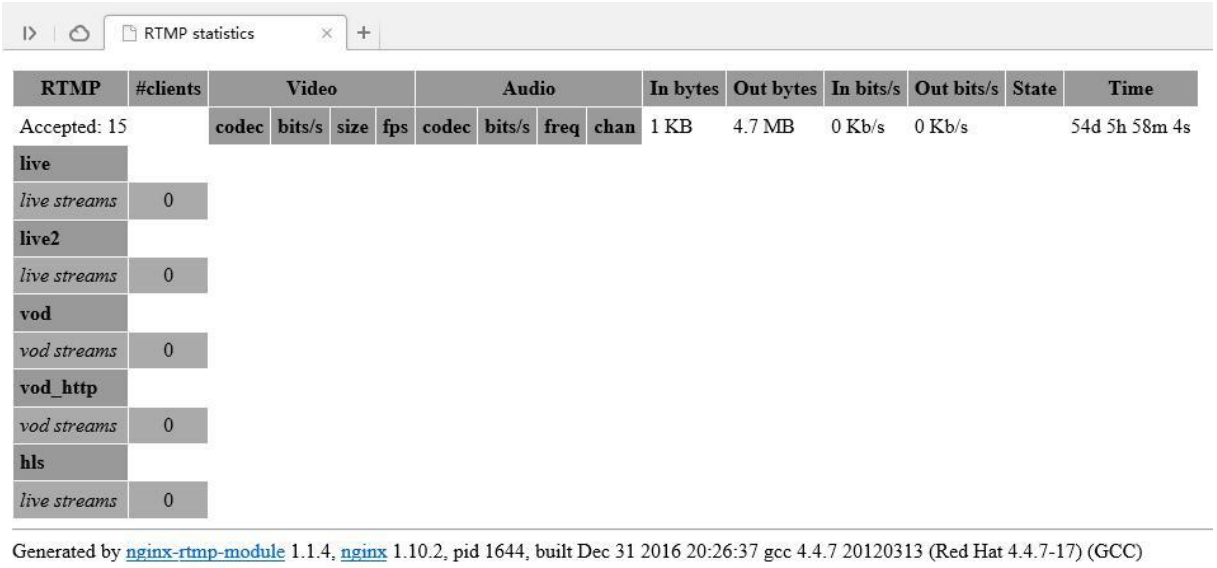


图 5.10　直播服务配置成功

5.2 音视频同步与网络传输实现

在网络传输功能实现上，通过向 RTMP 服务器传输音视频数据必须按照 FLV 的封装格式进行封包，因此，在完成 H.264 编码后的视频数据与 AAC 音频数据需要添加 FLV 头文件并将音频同步与视频同步包推送到服务器端。

5.2.1 FLV 封装格式研究

FLV 是由 Adobe 公司设计开发的一种流媒体复用封装格式，通常后缀名为.flv，其文件格式包括头文件和文件包两部分，结构如图 5.11 所示。

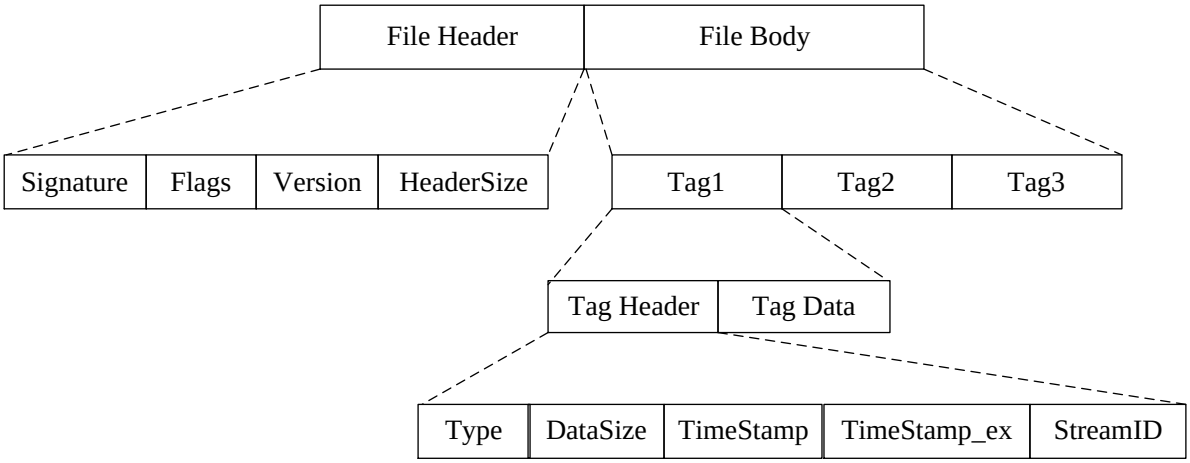


图 5.11　FLV 文件格式图

FLV 头文信息端通常由文件标识 Signature、标志位 Flags、版本 Version 与信息头大小 HeaderSize 组成。其中文件标识为“FLV”三字节，由 0x46，0x4c，0x66 表示。版本字

段为一字节表示 0x01，标志位 Flags 的前 5 位保留，必须为 0，第 6 位表示是否存在音频 Tag，第 7 位是保留位，第 8 位表示是否存在视频 Tag。信息头大小 HeaderSize 为 4 字节，表示从头文件到数据包的字节数。

File Body 由多个文件标签 Tag 组成，每个 Tag 同样包含头信息与数据信息两部分。其中 Tag 头包括 Type 字段，表示文件标签的类型，音频用 0x08 表示，视频用 0x09 表示；DataSize 字段表示该文件标签的数据部分的大小；TimeStamp 表示该 Tag 的时间戳；TimeStamp_ex 为时间戳的扩展字节，当 24 位数值不够用时，该字节最高位将时间戳扩展为 32 位数值，StreamID 总为 0[35]。

5.2.2 音视频同步实现

音视频同步即指对相互独立存在的音频和视频数据在时间顺序上进行正确的排序。一般来说，目前比较成熟并且较为常用的同步技术就是时间戳同步技术[36]。其根据不同的音频与视频编码打上显示时间戳或者解码时间戳。而时间戳本身也就指明了音频或者视频数据在何时应该播放或者解码出来，对音视频的同步效果比较好。

在 FFmpeg 中存在两种时间戳：DTS 与 PTS。DTS 即 Decoding Time Stamp 解码时间戳，表示音视频解码时的时间[37]。PTS 即 Presentation Time Stamp 显示时间戳，表示音视频显示时的时间。此外，在音视频同步时需要流长度信息，即音视频的 duration，表示音视频帧之间的间隔长度。此外，在流媒体同步技术中，存在媒体内同步与媒体间同步两种形式。媒体内同步描述同一媒体内部各帧之间的时间关系，媒体外间同步描述主要消除各媒体间的时间偏移[38]。媒体内同步如图 5.12 所示。

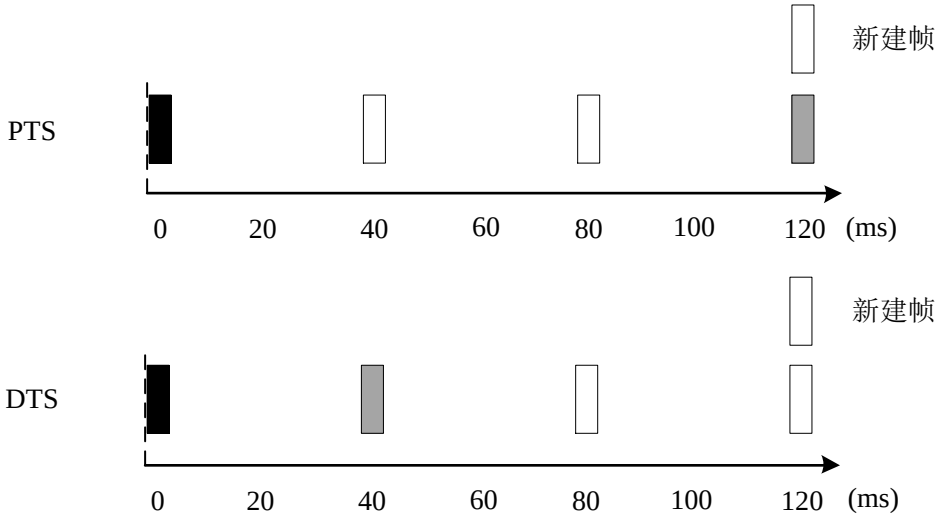


图 5.12 媒体内同步

对一个 24 帧每秒的 H.264 视频序列，其帧内间隔为 40ms，当视频帧的显示顺序为 I、B、B、P 时，由于解码 B 帧时需要 I 和 P 帧的信息，因此在存储序列时需要以 I、P、B、B 的顺序，当新加入一帧视频数据时，可以通过判断新建帧的帧类型，再进行准确时间戳同步。

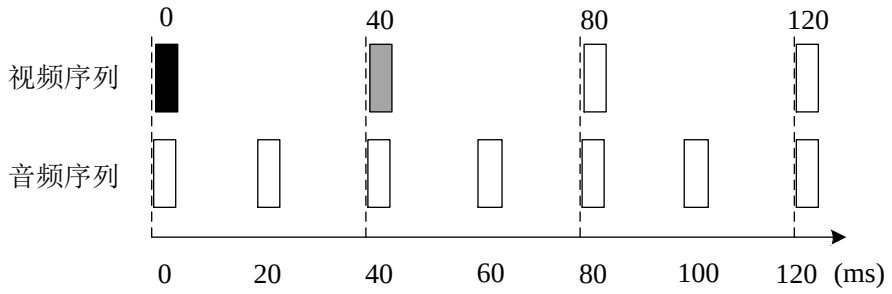


图 5.13　媒体间同步

媒体间同步通过对音频当前时间戳与时间基准的比较，选择合适的音频或视频帧进行编码同步工作。由于音频的流长度较小，因此在进行音视频同步时，可将第一帧视频 I 帧作为第 0 帧的时间基准。如图 5.13 所示。

当通过 av_read_frame()函数从设备流中读取到音视频数据时，会获得存在数据包中的 PTS 与 DTS，但此时间戳并不是解码帧准确的时间戳，无法进行音视频数据同步。因此，在音视频编码流程中，首先通过 av_compare_ts()函数比较时间戳，决定进行视频编码或者音频编码，在通过编码器对音视频数据编码后，判断编码帧是否存在时间戳，并且根据帧类型进行时间戳写入。部分核心代码如下：

```
if(pkt.pts==AV_NOPTS_VALUE)
{
    pkt.stream_index=0;
    AVRational time_base=pFormatCtx_out->streams[0]->time_base;
    AVRational r_framerate1 = pFormatCtx_Video->streams[VideoRet]->
r_frame_rate;
    AVRational time_base_q = { 1, AV_TIME_BASE };
    int64_t calc_duration = (double)(AV_TIME_BASE)/av_q2d(r_framerate1);
    pkt.pts=av_rescale_q(frame_index*calc_duration, time_base_q, time_base);
    pkt.dts=pkt.pts;
    pkt.duration=av_rescale_q(calc_duration, time_base_q, time_base)*2;
    pkt.pos=-1;
    cur_pts_v=pkt.pts;
    frame_index++;
}
```

5.2.3 RTMP 文件流写入与传输速率控制

为了提高直播系统的实时性，课题采用按帧发送的形式，因此，在与 RTMP 服务器建立连接的时候，通过文件流写入的方式进行通信连接。

```
const char *outFileUrl="rtmp://123.57.222.98:1935/RTMP";
avformat_alloc_output_context2(&pFormatCtx_out, NULL, "flv", outFileUrl);
if (avio_open(&pFormatCtx_out->pb, outFileUrl, AVIO_FLAG_WRITE) < 0)
{
AfxMessageBox("输出流建立失败");
    return -1;
}
```

在建立连接后，新建一个媒体流，通过头信息写入完成对音视频信息头的传输。

```
pVideoStream=av_new_stream(pFormatCtx_out,0);
if(avformat_write_header(pFormatCtx_out, NULL) < 0)
{
    AfxMessageBox("头信息写入失败");
    return -1;
}
```

最后，在编码完成后，按帧写入到文件流，并实时传输到服务器端。

```
int re=av_interleaved_write_frame(pFormatCtx_out, &pkt);
if (re<0)
{
    AfxMessageBox("视频写入错误");
}
```

在发送流媒体数据时，应注意传输速率的控制，因为 FFmpeg 处理数据的速度是很快的，瞬间将所有编码数据发送出去会造成流媒体服务器的高负载与网络拥塞，因此需要按照视频的实际帧率与时间戳信息进行发送。部分代码如下：

```
int64_t start_time=av_gettime();
if(pkt.stream_index==videoindex)
{
    AVRational time_base=ifmt_ctx->streams[videoindex]->time_base;
    AVRational time_base_q={1,AV_TIME_BASE};
    int64_t pts_time = av_rescale_q (pkt.dts, time_base, time_base_q);
    int64_t now_time = av_gettime() - start_time;
    if (pts_time > now_time)
        av_delay(pts_time - now_time);
}
```

5.3 本章小结

本章主要完成了直播服务器的搭建与数据网络传输实现。首先对 Nginx 服务器框架进行研究介绍，根据研究结果对 RTMP 流媒体服务器进行搭建实现，最后利用时间戳技术对编码后的音视频数据进行了同步工作，并根据 FLV 封装格式与 RTMP 传输协议进行了逐帧音视频数据网络传输的实现，提高了直播系统实时性。此外，在传输过程中，利用对传输速率的控制减轻了流媒体服务器的负担，进一步提高了系统的稳定性。

第六章 高清实时直播系统测试

高清实时直播系统在实现完成之后，需要对各项功能进行测试与评价。并且通过不断的测试，才能更新并且纠正系统在实现上的错误与不完善的地方，系统内部的某些参数和变量也是在不断修改中达到实际效果最佳的数值。因此，本章将会对直播系统的系统功能、网络延时、音视频传输质量等进行系统性的测试。

6.1 界面设计与实现

在直播软件主界面中，主要包括直播软件菜单栏、直播画面视频反馈窗口区、输入输出与编码设置窗口区、实时编码速率曲线图展示窗口区与直播音视频属性查看窗口区等。直播软件主界面如图 6.1 所示。

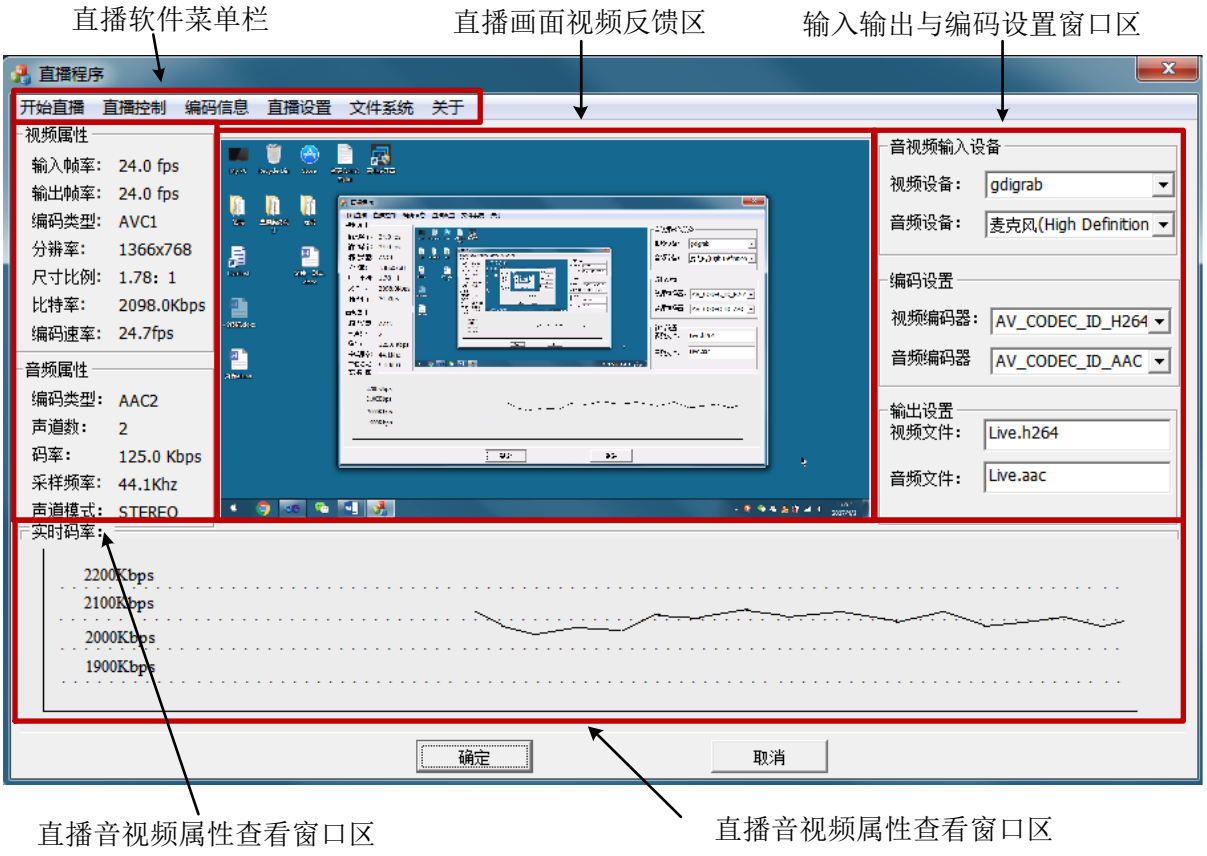


图 6.1 直播软件主界面

6.1.1 系统菜单栏

系统菜单栏功能，是直播器各项功能的集合。

1. 开始直播菜单。是不可弹出菜单，其事件响应函数的功能是根据直播设置菜单内

的参数，或者提供默认参数开启直播进程。包括视频采集进程、音频采集进程、音视频编码传输进程、本地文件存储、界面窗口显示等功能。

2. 直播控制菜单。弹出后提供两个子菜单，包括暂停直播与结束直播。

3. 编码信息菜单。提供各项音视频特征参数，包括分辨率、像素格式、比特率、帧率等关键信息窗口化整合展示功能。

4. 直播设置菜单。提供直播进程开始前各项参数的自定义设置功能，通过参数配置，实现对不同计算机软硬件环境以及网络情况的优化调整。

5. 文件系统菜单。提供直播媒体数据本地储存功能，包括音频、视频、封装音视频文件，方便用户日后查阅回看。

6. 关于菜单。直播软件的版本号显示与软件的说明。

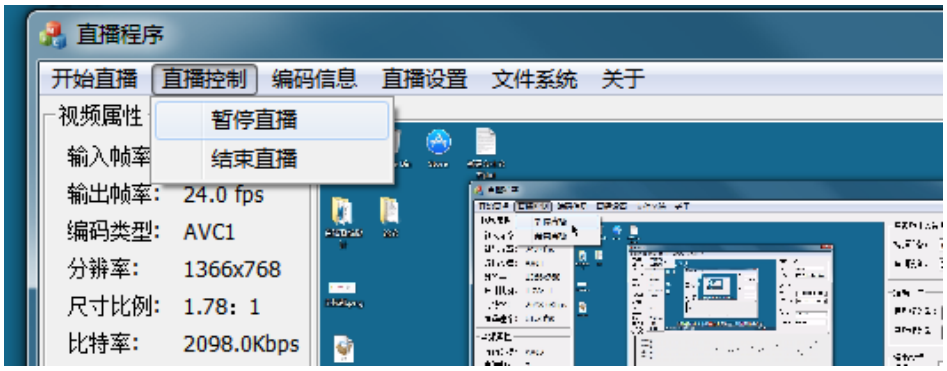


图 6.2　系统菜单

6.1.2 系统登录界面

在主界面窗口类的 OnInitDialog 方法中，生成登录窗口类，于直播器窗口程序初始化前运行，以判断是否是正确的用户或管理员运行程序。登录界面包括用户名与密码输入，登录账户判断功能。若用户或管理员输入错误的帐号密码，直播程序结束运行，直播窗口也不会生成。

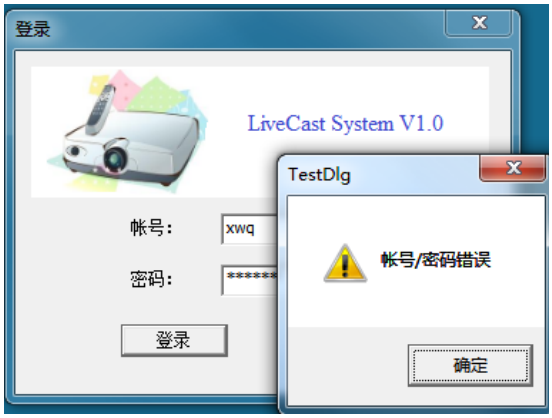


图 6.3　直播软件登录界面

6.1.3 音视频属性展示

音视频属性展示窗口区，是将音视频各项参数显示在直播器的界面上，方便用户查看。包括音频编码信息、编码速率、声道模式等，视频属性信息展示与视频相关的属性，如视频输入输出帧率、分辨率、编码速率等。

在音视频输入输出与编码设置窗口区中，通过对选项的设置而修改直播软件的编码方式、直播设备采集源、输出路径等。

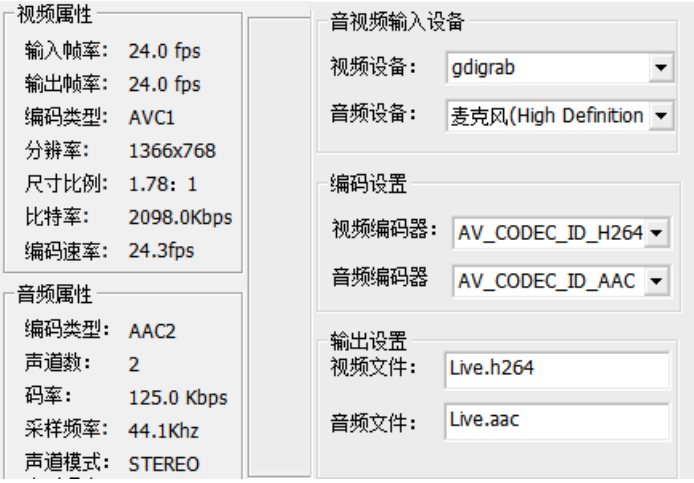


图 6.4 音视频属性区

6.1.4 实时编码速率曲线图

通过实时编码速率曲线图，用户可以从另一角度更清晰的了解当前直播器的编码效率与实时工作状态。

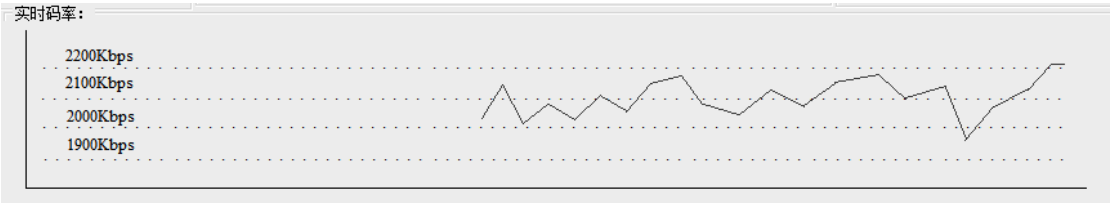


图 6.5 实时编码速率曲线图

6.2 高清实时直播系统总体测试

6.2.1 系统测试环境

首先对整个直播系统进行总体性功能测试，目的是检测直播平台对基本功能的实现效果。本次测试的系统硬件环境如表 6-1 所示：

表 6-1　直播系统测试环境表

项目	配置
操作系统	Microsoft Windows7
处理器	Intel i5-2410m，2.3GHz，双核四线程
内存	6G
网卡	常规 Realtek 笔记本电脑以太网卡
硬盘	西数 5400RPM，500G 硬盘

从表中可以看出本次测试的硬件环境条件一般，相比于目前大部分计算机，测试机器的各项性能都较低，因此测试结果具有很高的适用性。

6.2.2 安装流程测试

直播系统通过 VS2010 的 Installer 工程对直播软件进行封装发布，包含了软件需要使用的动态链接库以及可执行文件。在打包完成后，对安装项目进行调试运行，生成安装包。和普通软件安装过程类似，双击运行便进入了安装过程。

图 6.6　安装过程图

在安装过程中会提示用户对安装目录进行选择，最后在指定的目录下，安装程序会将封装文件解压出来，并在桌面生成快捷方式。

6.2.3 登录与运行测试

在完成软件安装后，点击快捷方式启动直播软件。在启动后，会首先生成一个登录窗口，在登录窗口输入正确的用户名与密码才可以进行登录操作，如果用户名与密码不匹配，则拒绝登录，以保护直播系统的数据安全。

图 6.7 直播软件登录窗口

在输入了正确的用户名与密码后，进入直播界面。

图 6.8 进入直播界面图

在直播软件未开始直播时，主界面中心位置是视频反馈窗口，直播运行时作为视频播放器窗口。左边是音视频属性区，负责对直播过程中的参数呈现。右边是采集设备与输出功能显示区，显示了采集源设备名、编码格式与输出文件。下方是实时码率图，在直播运行时输出实时的码率曲线。

6.2.4 服务器联合测试

在打开 RTMP 流媒体服务器后，运行直播系统，对直播软件的网络传输功能进行测试，结果如下：

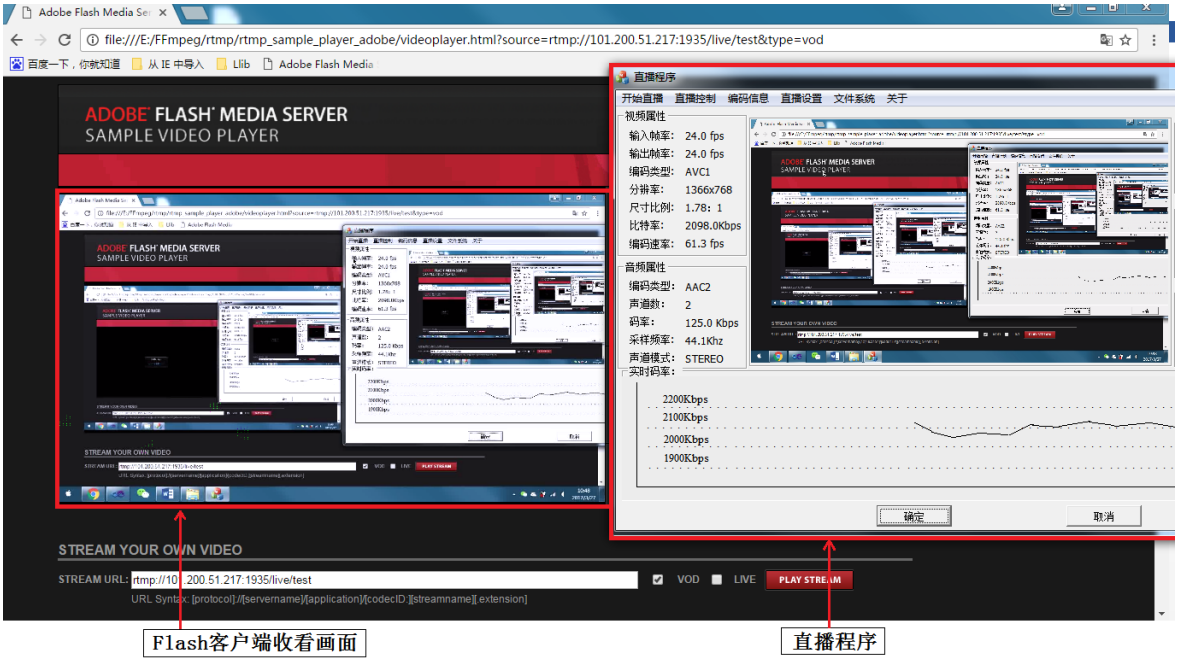


图 6.9　集成测试效果图

在与 RTMP 服务器进行连接后，直播软件将桌面画面传送给 RTMP 流媒体服务器，用户通过 FLASH 播放器可以完成对直播画面的接收与观看。在网络测试过程中，直播系统工作流畅，编解码帧率稳定，在 H.264 视频编码优秀的压缩性能下，网络带宽占用较小，客户端接收显示更新速度较快。

RTMP statistics

RTMP	#clients	Video				Audio				In bytes	Out bytes	In bits/s	Out bits/s	State	Time
Accepted: 11		codec	bits/s	size	fps	codec	bits/s	freq	chan	12.68 MB	9.76 MB	168 Kb/s	0 Kb/s		21m 35s
live															
live streams	1														
test	1	H264 High 3.2	165 Kb/s	1366x768	24	AAC LC	36Kb/s	44100	2	1.01 MB	0 KB	165 Kb/s	0 Kb/s	active	1m 15s
live2															
live streams	0														
vod															
vod streams	0														

图 6.10　RTMP 服务器监测图

通过 RTMP 服务器端检测数据可以验证直播软件传输的视频编码格式为 H.264，音频编码格式为 AAC。同时对网络带宽的占用率比较低，得益于高效率的压缩编码算法的实现。

6.2.5 服务器并发测试

最后对直播服务器的并发性能进行测试。通过使用 WebBench 软件模拟出多个客户端同时请求，对服务器进行并发性能压力测试，测试分别模拟出 10、512、1204、5120 个客户端请求对直播服务器进行连接，测试结果图如下所示。

```
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]# webbench -c 10 http://101.200.51.217/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://101.200.51.217/
10 clients, running 30 sec.

Speed=5386 pages/min, 75860 bytes/sec.
Requests: 2693 susceed, 0 failed.
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]#
```

图 6.11　10 个客户端的并发测试

```
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]# webbench -c 512 http://101.200.51.217/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://101.200.51.217/
512 clients, running 30 sec.

Speed=4834 pages/min, 68202 bytes/sec.
Requests: 2402 susceed, 15 failed.
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]#
```

图 6.12　512 个客户端的并发测试

```
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]# webbench -c 1024 http://101.200.51.217/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://101.200.51.217/
1024 clients, running 30 sec.

Speed=4574 pages/min, 64532 bytes/sec.
Requests: 2251 susceed, 36 failed.
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]#
```

图 6.13　1024 个客户端的并发测试

```
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]# webbench -c 5120 http://101.200.51.217/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://101.200.51.217/
5120 clients, running 30 sec.

Speed=4002 pages/min, 52292 bytes/sec.
Requests: 1948 susceed, 53 failed.
[root@iZ2ze4tmy66i2m53qv4i5oZ webbench-1.5]#
```

图 6.14　5120 个客户端的并发测试

由并发性能测试结果显示，直播服务器在同时应对多个客户端请求时，随着并发数的增高而服务性能变差，但由于在初步直播设计时，未加入大规模的服务器集群，因此，在此条件下的测试结果也是较为满意的。

同时，在高并发下的直播系统延时测试结果如图 6.15 所示。

图 6.15　直播系统延时测试结果图

由测试结果图可以看出，直播系统的延时为 0 秒，但延时在实际测试过程中其实是有的，在 200~500ms 左右，相比于现有直播系统中动辄 10-30 秒的延时有了极大的提高。如此低的延时在抢险指挥、医疗救助、实时转播等特殊场合具有很高的应用价值。

6.3 视频质量评价

在直播系统功能性测试完成后，再对直播输出的音视频数据进行音视频质量评价。评价通过对输出的音视频文件进行空间复杂度与空间复杂度计算，并且通过 PSNR 峰值信噪比对视频图像的质量进行计算评价。

首先使用直播软件的本地存储功能获取一小段直播音视频数据，通过 MediaInfo 工具查看到音视频数据的相关信息，如分辨率、比特率、颜色空间、比特率等，图 6.16 所示为通过 MediaInfo 显示的直播音视频数据信息的树状图。

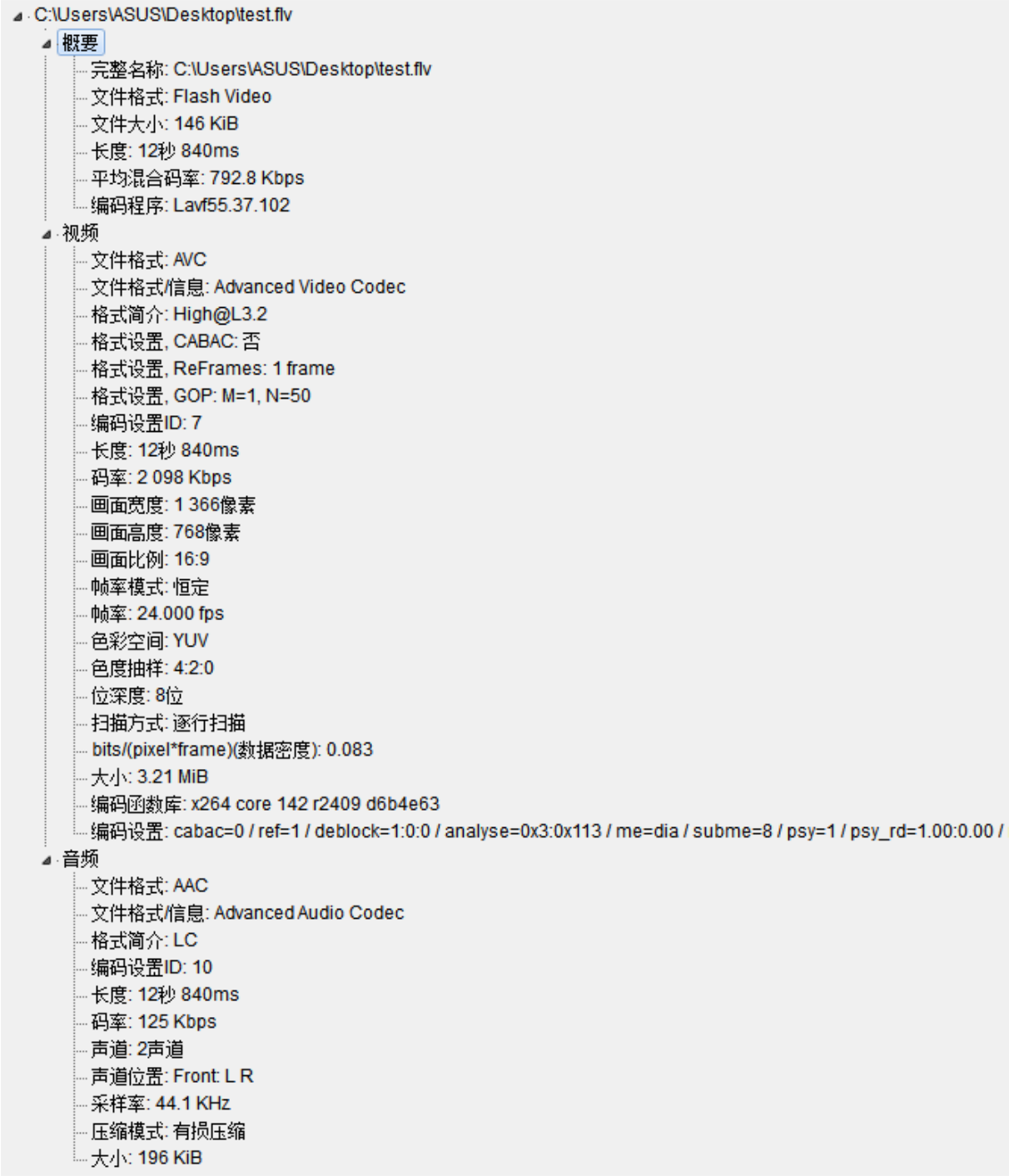


图 6.16　直播音视频数据信息树状图

由树状图中的信息可以清楚的了解到直播系统所输出的音视频相关信息。输出的封装格式为 FLV，包含视频与音频两种媒体数据。

视频采用 H.264/AVC 高清编码，画面分辨率为1366×768，是 14 吋测试笔记本屏幕最高分辨率；码率为高清 2098Kbps；画面比例为 16:9；24 帧每秒；YUV420 颜色空间。

音频采用 AAC 高级音频编码，双声道立体声；码率为 125Kbps；采样频率为 44.1KHz。

首先对一段输出视频序列复杂度特性进行计算。图 6.17 所示为原视频序列其中的某一帧。

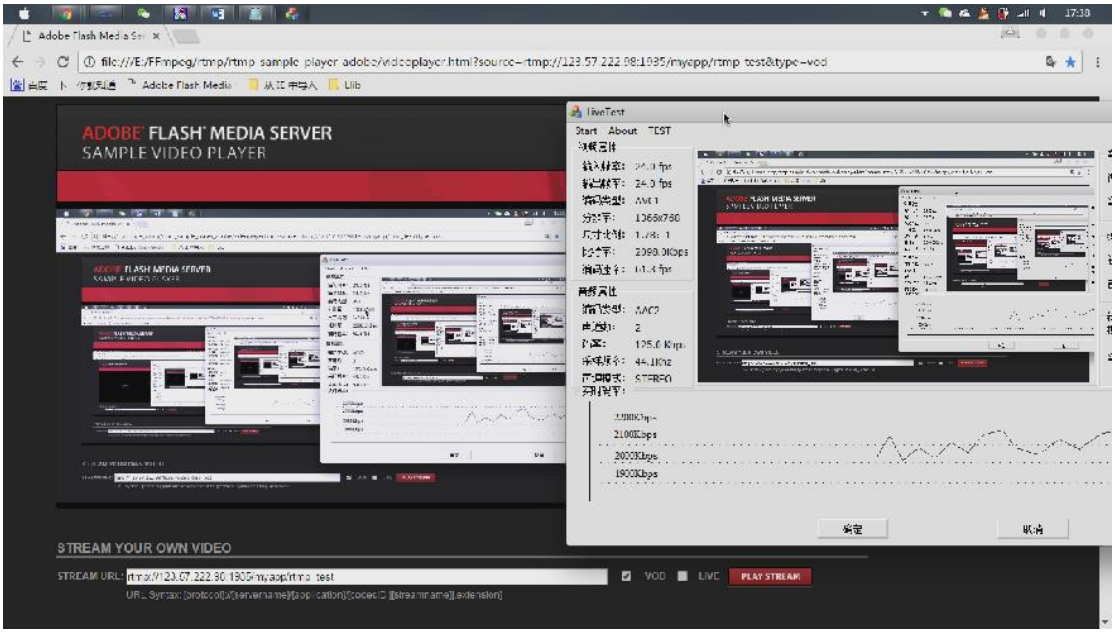


图 6.17　原视频序列帧

6.3.1 空间复杂度

图像的空间信息复杂度通常采用 SI（Spatial perceptual Information）表示，当视频序列在空间上拥有更复杂的场景细节出现时，其 SI 值就会越高。通常，SI 数据通过 Sobel 滤波得到。图 6.18 展示经过 Sobel 滤波后的画面信息。

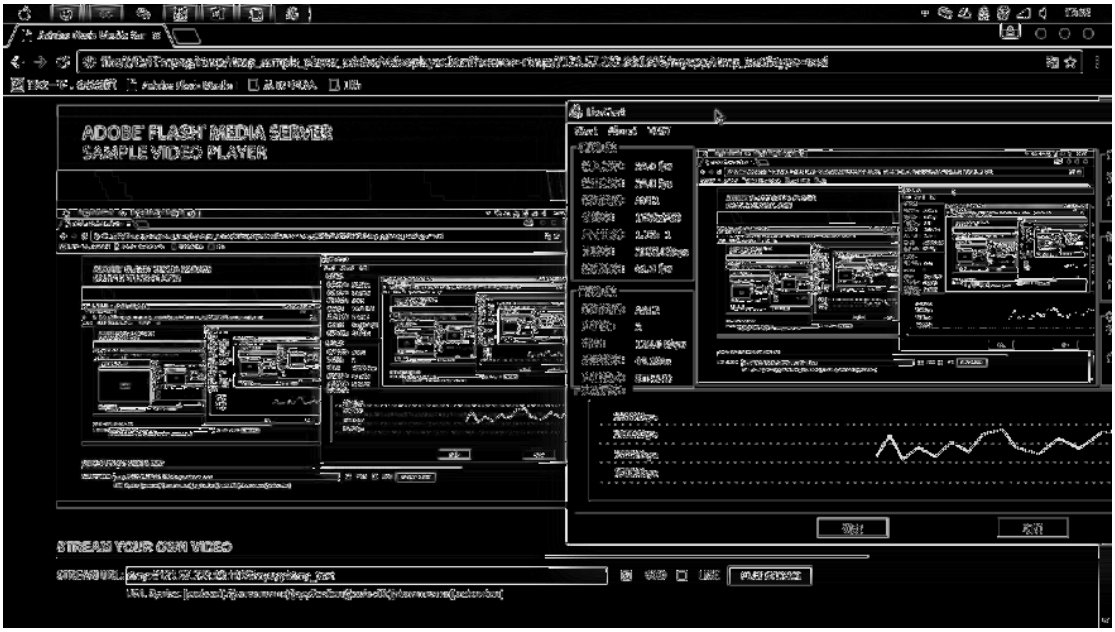


图 6.18　Sobel 滤波后图像信息

6.3.2 时间复杂度

图像的时间信息复杂度通常用 TI（Temporal perceptual Information）来表示，当视频序列在时间上拥有较为剧烈的运动变化时，其 TI 值就会变高。在对视频序列相邻帧之间的帧差做标准差之后，就得到视频序列的 TI 值。

图 6.19 视频序列帧差后图像信息

在对生成的 TI 与 SI 数据进行 Excel 数据制图，便可以呈现出复杂度折线图。如图 6.20 所示。

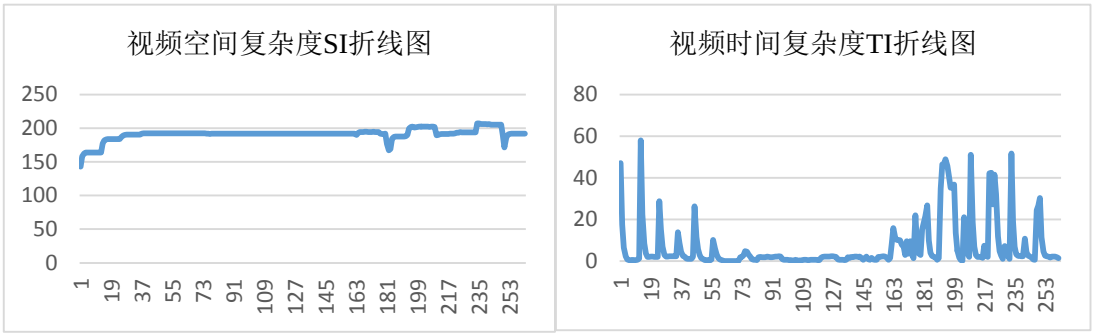


图 6.20 SI 与 TI 折线图

从 SI 折线图中可以直观的看出，该视频序列画面的空间细节比较平稳，且细节较多，因此呈现了较高的空间复杂度。由 TI 折线图可以看出在视频序列的前段，由于视频帧的内容运动比较剧烈，因此造成了较多的波峰出现，中间部分图像内容运动平滑，因此折线稳定且数值较低，在视频尾部由于关闭直播软件引起视频画面帧内容较大变化，因此也造成较多的波峰。

6.3.3 峰值信噪比

PSNR（Peak Signal to Noise Ratio）峰值信噪比是一种对视频序列中图像的客观评价，通过原始视频序列的参考帧与失真视频序列进行逐帧对比，在失真视频的相似程度上对视频序列做以判断[39]。通常 PSNR 的取值范围为 20-40，其数值越大，表示视频序列的质量越好[40]。

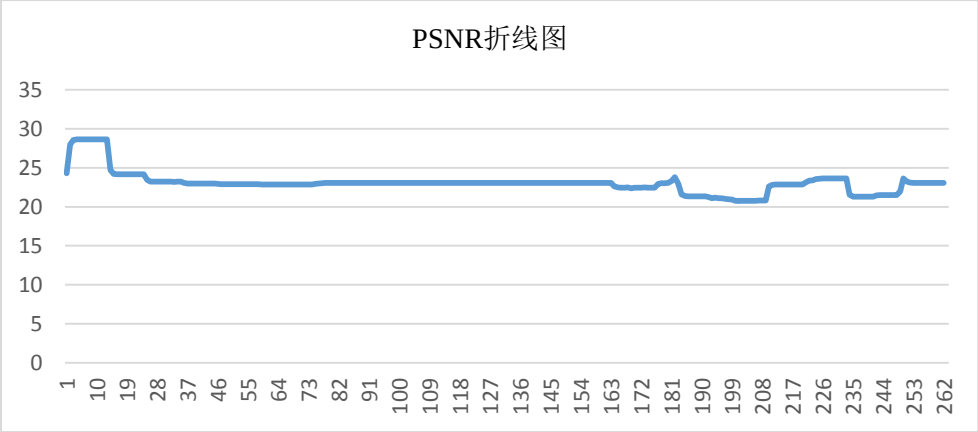


图 6.21　PSNR 折线图

由折线图可以看出直播系统输出的视频序列失真较小，输出画面稳定。

6.4 本章小结

本章首先完成了对直播软件界面的设计与实现，通过对直播软件安装过程、用户登录与服务器联合测试等单元测试，并在服务器检测端查看转发情况，完成系统的总体测试。最后，对采集编码后的视频序列进行峰值信噪比 PSNR、时间与空间复杂度评价，给出质量评价结果。

结论与展望

网络直播技术的发展不仅丰富了人们的文化生活，而且在一些特殊场合有着重要的作用。然而随着直播系统大规模应用，其在实现中的一些不足逐渐显露出来，因此，为了克服现有直播系统中存在的若干问题，本文设计实现了一套高清实时直播系统，在对系统进行测试后表明，在直播画质与音效、直播实时性与客户端的便捷性等方面均有不同程度的提高。论文在此基础上主要进行了以下几方面的研究和工作：

1. 论文首先通过实际使用各种直播系统，并翻阅大量资料，深入了解目前直播系统中存在的一些弊端。再结合这些问题的特点与存在的原因，明确了对现有直播系统提升与改进的方向。

2. 接着对高清实时直播系统实现过程中所采用的关键技术做了研究概述，包括视频颜色空间、高清 H.264 视频编码标准、AAC 高级音频编码原理、FFmpeg 音视频处理技术与 RTMP 流媒体传输协议，之后便提出了高清实时直播系统的总体设计方案。

3. 完成对 FFmpeg 源码的移植编译、裁剪与 VS2010 开发环境下的快速配置工作，同时详细研究了 FFmpeg 对音视频的处理逻辑流程，并根据高清实时直播系统的需求对 FFmpeg 处理流程进行优化改造。

4. 对于采集速率过高而引起计算机高负载与缓冲数据溢出等问题，通过加入采集速率控制功能，使音视频观感与系统处理效率达到最优平衡。并且根据人眼对亮度信息较敏感的特性，对视频数据做了 YUV 颜色空间转换。

5. 针对音视频编解码过程中出现的抖动与数据丢失问题，设计实现了音视频解码数据缓冲区。并且利用临界区资源完成了对多线程间的同步工作，同时使用 SDL 技术对视频解码数据进行了反馈显示。

6. 使用 H.264 高清视频编码与 AAC 高级音频编码标准对直播数据进行高质量压缩编码，在提高直播音视频质量的同时减小了网络带宽的压力。此外，还对音频采集与解码过程中出现的杂音进行了音频重采样处理。

7. 使用时间戳同步技术完成了对 H.264 数据与 AAC 数据的音视频同步工作，并且根据 FLV 复用封装格式规范，使用 RTMP 流媒体传输协议对直播音视频编码数据进行了逐帧网络传输，极大地降低了数据传输延时。

8. 在应对直播服务器的高并发与转发延时问题上，基于 Nginx 搭建了 RTMP 流媒体直播服务器，通过直播数据立即转发的实现，更进一步地减小了网络延时，并且对直

播服务器在应对高并发请求方面的处理能力做了一定的提升。此外，通过 Flash 播放器可以轻松地使用户收到到高清实时直播画面，提高了客户端部署的便捷性。

本文的工作虽然在网络延时、音视频处理效率、直播画面质量等方面取得了一些成果，但是由于时间关系与直播系统本身的复杂性等因素，以下工作还有待于今后完善：

1. 本文只完成了 Windows 平台下的高清实时直播系统，然而面对移动直播领域还需要对直播系统进行移植与改进工作，例如在 3G、4G 网络环境下的动态降低实时流媒体帧率与图像质量。

2. 对于直播服务器的功能方面还有待丰富，如直播音视频数据的最小分片功能、对直播服务器的分布式部署与集群间的负载均衡。

3. 在网络与信息安全日益严峻的今天，需要对直播数据进行加密处理，同时应对系统进行多项安全防护与漏洞排查工作。

参考文献

[1] 周骑, 诸强. 无线视频监控系统研究及应用[J] . 无线互联科技, 2015, (01): 78-80+160

[2] 周枫, 薛荧荧, 李千目. 视频监控与编码技术的研究[J] . 软件, 2015, (04): 84-92

[3] 韩德, 王云珊. 基于WMS的流媒体系统分析与应用[J] . 西昌学院学报(自然科学版), 2007, (04): 71-74

[4] 孙元波. 网络视频直播质量控制技术研究[D] . 国防科学技术大学, 2004

[5] 张学习, 杨宜民. 彩色图像工程中常用颜色空间及其转换[J] . 计算机工程与设计, 2008, (05): 1210-1212

[6] Tkalcic M, Tasic J F. Colour spaces: perceptual, historical and applicational background[C] // Eurocon 2003. Computer As A Tool. the IEEE Region. IEEE, 2003: 304-308 vol.1

[7] 丁绪星. 基于 HVS 的静止彩色图像小波压缩编码[J] . 仪器仪表学报, 2006, (08): 977-980

[8] 孙华. H.264 视频编码标准的分层设计与功能[J] . 广播与电视技术, 2004, (04): 31-33

[9] 刘斌. 基于 FPGA 的 H. 264 视频编码器[D] . 杭州电子科技大学, 2012

[10] Marpe D, Schwarz H, Wiegand T. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard[J] . IEEE Transactions on Circuits & Systems for Video Technology, 2003, 13(7): 620-636

[11] 肖鹏. H.264 压缩视频的超分辨率重建技术研究[D] . 南京邮电大学, 2011

[12] 兰洋, 郑高群, 李尚柏. 基于概率排序的静态奇偶编码压缩算法[J] . 四川大学学报(自然科学版), 2003, (02): 244-250

[13] 刘丹, 王相海. 视频帧差图像编码研究[J] . 计算机工程与应用, 2007, (07): 45-48

[14] 杜小钰, 马力妮, 潘峰. H.264 多参考帧技术的探索与应用[J] . 计算机技术与发展, 2008, (07): 200-202

[15] 薛莹莹. 基于 H.264 的错误隐藏技术研究[J] . 河南科技, 2015, (06): 30-32

[16] 李敬磊. H.264 解码器中帧内预测的硬件实现[A] . 北京图像图形学学会. 图像图形技术研究与应用 2009——第四届图像图形技术与应用学术会议论文集[C] . 北京图像图形学学会: , 2009:5

[17] Po L M, Ma W C. A novel four-step search algorithm for fast block motion estimation[J] . IEEE Transactions on Circuits & Systems for Video Technology, 1996, 6(3): 313-317

[18] 田川, 王永生. 关于 H.264 帧内预测模式选择快速算法的研究[J] . 计算机工程与应用, 2006, (11): 13-15

[19] 郑航. 基于 FPGA 的 H.264 基本档次编码器关键算法研究与设计[D] . 电子科技大学, 2009

[20] Malvar H S, Hallapuro A, Karczewicz M, et al. Low-complexity transform and quantization in H.264/AVC[J] . IEEE Transactions on Circuits & Systems for Video Technology, 2003, 13(7): 598-603

[21] 熊承义, 董朝南. 基于中心点预测的分数像素运动估计改进算法[J] . 中南民族大学学报(自然科学版), 2010, (01): 68-72

[22] 朱剑英. 基于 DCT 变换的图像编码方法研究[D] . 南京理工大学, 2004

[23] Langsim. RTMP 流媒体播放过程[EB/OL] , http://blog.csdn.net/langsim/article/details/46500469, 2015-06-15

[24] 杨克伟. 视频压缩算法研究[D] . 厦门大学, 2014

[25] 石利琴. 论数字音频信号的采样[J] . 出版与印刷, 2003, (S1): 61-63

[26] 杜少炜, 师卫. 实时视频流客户端自适应缓冲管理策略研究[J] . 科技情报开发与经济, 2007, (31): 200-201

[27] 边朝政. 用于光通信的视频编码器的研究[J] . 科技资讯, 2013, (33): 12-13

[28] Li H, Li Z G, Wen C. Fast Mode Decision Algorithm for Inter-Frame Coding in Fully Scalable Video Coding[J] . IEEE Transactions on Circuits & Systems for Video Technology, 2006, 16(7): 889-895

[29] Aucouturier J J, Canonne C. Musical friends and foes: The social cognition of affiliation and control in improvised interactions[J] . Cognition, 2017, 161: 94-108

[30] 闫晶. 视频会议图像获取和处理技术研究[D] . 中北大学, 2008

[31] 边朝政. 用于光通信的视频编码器的研究[J] . 科技资讯, 2013, (33): 12-13

[32] 谢静苑. 基于特征图像分类以及稀疏表示的超分辨率重建[D] . 南京邮电大学, 2013

[33] Schuijers E, Breebaart J. Low complexity parametric stereo coding[J] . Audio Engineering Society Convention, 2004

[34] 王利萍. 基于 Nginx 服务器集群负载均衡技术的研究与改进[D] . 山东大学,

2015

[35] 林志伟. 面向移动终端的流媒体实时通信服务器关键技术研究[D] . 厦门大学, 2012

[36] 刘丽霞, 边金松, 穆森. 基于 FFMPEG 解码的音视频同步实现[J] . 计算机工程与设计, 2013, (06): 2087-2092

[37] 蔡岳峰. 基于流媒体的多声道音频播放器设计与实现[D] . 华中科技大学, 2013

[38] 吴炜, 常义林. 在接收端实现的媒体同步控制算法[J] . 系统工程与电子技术, 2006, (10): 1587-1591

[39] 郭振. 基于灰度投影的电子稳像系统及评价方法[D] . 天津大学, 2009

[40] 张兆林, 史浩山, 万帅. 基于线性回归分析的视频质量评估方法[J] . 西北工业大学学报, 2012, (03): 451-456

攻读硕士学位期间取得的研究成果

发表论文：

[1] PID 控制器矫正及参数整定的研究

实用新型专利：

[2] 一种用于信号灯黄闪的控制电路

致谢

本课题是在张卫钢教授的悉心指导下完成的，导师在学习与生活中的严格要求，让我不仅在专业知识方面有了很大提升，在生活与为人处世方面的关心与指导，更是让我受益颇深。尤其是导师渊博的知识，严谨的治学态度以及崇高的敬业责任精神令我钦佩，值得我终生学习。在此向我的导师，张卫钢教授表达最衷心的感谢和诚挚的心意。

同时，感谢网络工程系的任卫军老师，在攻克技术难关的时候给予了莫大的关心和无私的帮助。

还要感谢我的同门，实验室里的师弟师妹，不仅在课题的完成上提供很多帮助，在生活中的陪伴和鼓励，给了我巨大的勇气与动力来面对艰难。谢谢你们。

最后，感谢长安大学给了我这样一个学术氛围浓厚，科研精神卓越的学习生活环境。祝愿我的母校光辉历程更辉煌，人才辈出代代强，桃李满天扬四海，硕果累累振中华。