基于FFmpeg的网络视频监控系统的设计与实现

重庆大学硕士学位论文

（学术学位）

学生姓名：薄建彬

指导教师：刘京诚　教 授

专　　业：仪器科学与技术

学科门类：工　学

重庆大学光电工程学院

二O一四年四月

Design and Implementation of Video Surveillance System Based on FFmpeg

A Thesis Submitted to Chongqing University

in Partial Fulfillment of the Requirement for the

Master's Degree of Engineering

By

BoJianbin

Supervised by Prof. LiuJingcheng

Specialty: Instrument Science and Technology

College of Opto-electronic Engineering of Chongqing University ,

Chongqing, China

April 2014

摘　　要

社会经济的不断发展影响着每个人生活的方方面面，人们也逐渐关注起了公共安全问题，相关安防产品的影子正大量出现在我们周围。计算机与网络技术的不断完善使得视频监控具有了数字化、网络化的特点。现在各地市的大学都普遍存在着学生众多，校园周边的环境复杂等特点，尤其是高校的实验室以其重要性、敏感性的特点，需要更高的安全防护措施。然而，RTP/RTCP 协议在有些情况下并不合适，针对特定问题需要对协议进行不同程度的二次开发；另一方面，在移动互联网快速发展的今天，系统终端变得多样化，让人们对监控方式和监控手段有了多样性、灵活性的新需求。本文针对这两方面的问题，结合最新的科技动态，设计并实现了一套完整的解决方案。

本文使用了 FFmpeg 这一跨平台的多媒体解决方案来负责混流、解码、编码的工作，采用了最新的 H.264 的编码标准来提高压缩率。制定了应用层协议以保证高效简洁的通信，加强了信息的交互，提高了客户端在媒体控制中的作用。服务器使用 POSIX 线程标准，提供了 TCP 和 UDP 两种协议的支持，根据嵌入式平台的特点，在自适应码率、多客户连接等方面进行了细致的设计。客户端拥有良好的人机界面，采用 Qt 框架来实现多平台的支持，不仅支持桌面环境，还支持嵌入式环境和众多的智能移动终端。论文主要的工作如下：

① 研究了 V4L2 接口和 FFmpeg 多媒体解决方案，详细介绍了用 V4L2 接口来采集数据的一般步骤和 FFmpeg 编解码的一般流程。

② 完成了嵌入式平台环境的搭建，内容涉及交叉环境的建立、Bootloader 的移植、x264 与 FFmpeg 的安装和移植等。

③ 研究了 Linux 网络模型和 POSIX 线程，开发出了高可靠性、支持多连接且自适应码率的实时视频服务器。根据实时性的要求制定了应用层协议，增强了客户与服务器间的交互，并将视频质量配置的决定权移至客户端，建立了客户端导向性的处理模式。

④ 研究了 Qt 框架的特点并几个主要的模块，然后从类间的联系和模块划分的角度对客户端软件的设计进行了详细的阐述。

关键词：视频监控，嵌入式系统、FFmpeg，Qt

ABSTRACT

Development of social economy affects every aspect of everyone's life; there is a serious concern about public safety matter. Now, we can easily find many security products used in our surrounding. The continuous improvement of computing and network technology makes video surveillance with digital and network characteristics. However RTP and RTCP protocols don't function properly in some cases, they seem too complicated and inflexible in some simple occasions. Sometimes we need direct exchange. On the other hand, the system terminals are diversified because of the rapid development of mobile Internet, which makes people have new demands for diversity and flexibility in ways of video surveillance and video control. In order to solve the two issues, this paper introduces a complete solution combined with the latest science and technology.

This system uses the latest H.264 video compression standard to get higher ratio, and uses FFmpeg for muxing, encoding and decoding, which is a complete, cross-platform solution. This paper developed an application layer protocol to ensure simple and efficient communication, information interactions between clients and servers, and client's leadership in the media controlling system. Servers use POSIX threads standard, supporting both TCP and UDP protocols and it is optimized for many aspects, such as adaptive bit rate, multiple customer connections and real-time performance. Client software based on Qt framework have elegant man-machine interface, it supports desktop environment, embedded environment and intelligent mobile terminals. The main work is as follows:

① V4L2 interface and FFmepg multimedia solutions are introduced in detail, detailing general steps for using V4L2 interface to collect media data and processes for encoding and decoding with FFmpeg.

② Embedded development environment is built,covering the establishment of cross-compilation environment, migration of Bootloader, installation and migration of x264 and FFmpeg.

③ Different patterns of server model are introduced in this chapter, and this chapter also gives a lot of description about POSIX threads, which is available on many Unix-like operating systems. After that, this chapter gives description about the development of a real-time video server, which shows features of high reliability,

support for multiple connections and adaptive bit rate. Application layer protocols are developed under the needs of real-time interaction and embedded system; it enhances interactions between client and server, and gives the right of video quality evaluation to clients.

④ Characteristics of Qt framework and several major modules are introduced. In the end, this chapter shows main structure of client software from angle of classe links and module functions.

Keywords: Video surveillance, Embedded system, FFmpeg, Qt

目　录

1 绪 论

1.1 视频监控的背景及意义

随着技术的不断发展，特别是计算机科技、网络技术、视频处理技术的日益完善，网络视频监控形成了一个高速发展模式。尤其在高速数字化的过程中，监控系统被广泛的应用于一些安全性要求高的地方。

2005年7月7日伦敦的公共交通系统发生了连环爆炸，两周后，数次恐怖袭击事件再次降临于伦敦，造成了大量的伤亡和短时交通网络的崩溃。袭击事件不管从任意性还是从破坏性的角度来说都让每一个人都惊恐不安。伦敦爆炸发生后，伦敦警方迅速作出回应，在不到两周的时间里爆炸案的嫌疑人也已经全部落网。 侦破过程中，伦敦人民共同参与、协同合作无疑加速了嫌疑人快速落网的时间，不可忽略的是，整座城市的视频系统在其中发挥了重要作用 [1]。

我国安防监控行业的发展起步较晚，虽然目前得到了快速的发展，但整个行业的水平对比国外来看仍有相当的差距，上世纪90年代以后，我国的城市视频监控才形成可观的水平，主要应用于交通管理和一些公共设施上，如火车站、铁路轨道、高速公路、商场及机场的安全。经济和社会的进一步发展使得全国各地的社会环境显得异常的复杂，在一些经济发展较快的城市中，人口的流动性往往很大，由此产生了比较大型的，涉及安全、信息、教育的公共区域项目。近年来，城市的总体监控系统正在迅速发展，影响到人们生活的方方面面，比如城市管理、教育系统、天气、公民安全和交通疏导等方面。从2006到2010年间，安防行业得到了比较快速的发展且形成了门类齐全的体系，涌现并壮大了一批行业领先的公司和产品。在2010年，整个行业产值突破了两千多亿元，平均每年达到了超过二十个百分点的强劲增长，在这里面，安防系列产品占到了约为一千亿元的份额，特别是，相关电子产品的增长速度惊人。视频监控领域在整个安防生态链中有举足轻重的作用，不仅体现在产品的相关性上，更是体现在所占有的份额上，根据目前的安防行业的产值和增长速度来看，在2015年时，视频监控产值将贡献安防电子产品份额的一半，达到千亿元的水平，成为增长最快、规模最大的领域。

自从1990年以后，计算机网络开始了迅速的发展，其中以因特网最具代表性，给人们的学习和工作带来了从没有过的便捷，网络普及的速度达到了从没有过的高度，迅速的在全球范围内展开、普及，引起了一场全球性的信息革命[2]。视频就在这种背景下有了网络化、数字化的特点。目前，在信息网络中使用的非常普遍媒体交互平台就是基于嵌入式的多媒体技术和客户端/服务器结构下的视频传输系统[3] 。通过Internet或以太网我们可以很方便的实现资源共享，为远程网络视频监

控打下了坚实的基础。因此拥有很好的发展前景。

1.2 视频监控的发展历程

视频监控[4]的发展有三个比较明显的过程：全模拟的视频监控系统、半数字化视频监控系统、网络数字视频监控系统[5]。

① 全模拟的监控系统主要由采集模块、传输模块和存储模块来组成。这三个模块的处理方式均采用模拟信号的形式，并且用同轴电缆来负责媒体信号的传输，这是该阶段监控系统的主要特点。主要的设备为前段摄像机与音视频切换矩阵主机。缺点有传输距离有限、模拟信号受干扰程度大、施工复杂等方面[6]。

② 半数字化视频监控系统与全模拟监控系统的不同点是：半数字化系统的存储模块和采集模块均为数字形式。该形态下多媒体监控系统的主要表现形式为硬盘录像机，作为半数字化系统中可分为PC与嵌入式DVR两种的核心产品。特征是以本地存储为主要功能，并使用模拟的形式来进行传输。不足的地方有监控范围有限；远程显示、管理交流的功能有限。

③ 网络数字视频监控系统是第三代的监控技术，机器计算能力的提升和高效的多媒体压缩算法的出现使得全数字化成为可能。主要的特征为利用快速的计算能力让媒体的压缩、传输和存储全数字化，并与网络相结合，让媒体信息间的交互更加的自由。这类监控系统在多样化传输方式、编码技术、平台的管理方式等方面有了很大的进步。

在科技日益进步、日益发展的今天，在3G移动互联网时代，尤其是在以Android和iOS设备的兴起后，移动智能终端得到了迅速的普及，以多种多样的形式出现在人们的日常生活和工作中。移动智能终端的普及速度之快令人咋舌，超过了历史上任何一种消费类技术，包括PC技术和网络技术的革新。在此，智能终端将“移动”和“智能”这两个词带入了人们的生活和学习中，深刻的影响和改变着人们日常的行为生活方式。根据一份有关智能设备的统计资料显示，2013年的时候，国内智能终端的保有量达到了七亿台，与2012年底相比，在不到一年的时间里保有量增加了将近三亿台，显然这样的增长是非常迅速的。我们必须重视这样的发展趋势，让我们的生活更加的便捷、更加的舒适。因此视频监控的方式就有了更加广阔和灵活的平台。因此，将移动互联网与视频监控结合在一起是网络监控时代的另一个令人兴奋的挑战。

1.3 视频编码简介

视频的编码格式就是指特定的视频压缩技术。视频的帧与帧的内容之间有很大的联系，这就表明在帧与帧之间有大量的冗余数据。通过对其进行压缩来去除

多余的或不重要的信息，既节约了空间又提高了通信质量。用于网络传输的视频编码技术主要有：

① MJPEG。MJPEG是一种帧内的压缩格式，分别对每一帧的图像数据进行独立的编码，并不研究帧间数据的冗余信息。这样的处理方式可以获得很清晰的图像，因为帧的数据是独立的，并不依赖前面或后面的帧，所以可以自由、灵活的编辑和处理视频，可以提高相关视频应用的准确性和实时性。但比起其优点缺点是非常的明显，主要表现在编码的效率上，由于是帧内压缩，这需要大量的存储空间和网络带宽[7]。

② MPEG。MPEG是动态图像专家组的简称，一个专为音频和视频制定标准的组织。其制定的标准在全球范围内有广泛的应用，主要涉及MPEG-1、MPEG-2、MPEG-4等多个音视频解决方案。

MPEG-4是一个涉及音频和视频数据的压缩方法，最初的版本在1998年获得通过[8]。MPEG-4吸收了之前MPEG-1/2的许多优点，并且也从其他格式中借鉴了许多可取的部分。它在比特率处在较低的水平下依旧能带来清晰度很高的图像[9]。在对之前标准进行优化的同时，MPEG-4也加入了一些新的东西，比如针对3D渲染的模型语言支持等。在帧间的内容变化剧烈时，MPEG-4的处理方法本身确保不会显示方块的现象。这种拥有高压缩率特点的算法在网络上传输多媒体信号时有良好的表现，[10]。

③ H.264是MPEG-4的第十部分，初始的版本公布于2003年[11]。与MPEG-2、H.263等旧标准相比其优点主要有以下几个方面：一是它提供高质量视频的同时所需的带宽更低；二是没有增加很多复杂的设计，这样保证了可行性与低成本；三是在各种的应用、网络中提供了很大的灵活性。在技术的实现层次上采用了多参考帧的补偿，使用最大16*16最小4*4的块来进行运动的补偿及估计，并采用增加权重的方法来进行补偿。这些特点明显的提高了性能，也是该算法与以前编码算法所不同的地方。同样的质量下，H.264所需要的码率仅仅是MPEG-2所需的一半或更少。

1.4 论文的主要工作和结构安排

嵌入式产品有很大的优点，体现在高可靠性与专用性、高灵活性、受限的体积、低成本等方面，多种多样的涉及娱乐、教学、生产方面的嵌入式产品正越来越多的出现在生活周围。得益于近年来的信息革命，嵌入式市场获得了再一次的动力与又一轮的增长[12]。同时，在移动互联网时代的背景下，造成了平台多样化、系统多样化的问题[13]。

主要的研究工作如下：

① 编写高实时性，高可靠性、多连接的流媒体视频服务器，并同时支持TCP与UDP两种运输层协议，为客户端的不同性能要求提供保障。并根据网络的实时情况来自动调节码率，以保证视频的流畅。

② 针对平台及终端多样化、系统多样化的特点来开发跨平台、多连接的流媒体视频客户端，以适应当今的移动智能终端的发展。

③ 完成应用层协议的制定，以精简有效的方式为可靠的数据交互提供保障，并将H.264这个领先的编码标准与FFmpeg多媒体框架应用于流媒体的开发中，以此来提供高质量的视频。

论文的具体安排如下：

第一章：主要阐述了视频监控的背景、意义，说明了在当今社会视频监控的重要性和必要性。并简要描述了其发展历程。而后，介绍了一些视频编码标准和它们的特点。

第二章：给出了系统的总体框架，包括设计目标和系统的功能模型。介绍了在Linux中的视频采集接口。给出了FFmpeg这个领先的开源多媒体解决方案的简述。最后，描述了如何来搭建开发环境，具体有PC端开发环境的搭建，嵌入式平台中环境的搭建

第三章：介绍了RTP协议与RTCP协议。并根据监控系统的需求简化了客户与服务器间的通信规范，给出了重新制定的协议格式与适用范围。

第四章：涉及到流媒体服务器的方方面面。首先介绍了服务器的多种设计范式。最后，给出了服务器的详细的实现细节。

第五章：是有关客户端的实现章节，首先是总体的介绍了Qt这个跨平台的GUI开发框架，其中包含了Qt的历史特点。其次是客户端中类的构造层次及功能模型。最后分别对TCP和UDP介绍了其解码模块和显示模块的实现细节。

第六章：是总结与展望章节，对论文进行了总结，提出了不足和下一步的改进工作。

2 系统的总体设计

2.1 系统设计目标

整个的系统是由视频采集、编码端和显示端构成。前者是运行着linux系统的嵌入式设备，具体负责视频的采集、分拆、封装的工作，作为提供服务的一端，为向其提出申请的客户端提供实时视频数据；后者作为接受服务的一方，来向用户提供视频信号的显示、控制功能。

系统的设计目标为：形成一套完整的实时视频监控方案，涉及系统的移植、视频的采集、视频的编码、视频的传输和客户端软件的提供。视频的提供方不仅要完成视频的采集工作，还要根据不同的需求来制定不同的传输机制，以建立实时性优先和准确性优先的两种策略。客户端能够同时连接多个视频服务器，在提供录像保存、录像检索、截图等基本功能的同时还要以界面友好的方式呈现出来。客户端具有跨平台的特点，既能在PC平台上部署，也能在目前日益流行的智能设备终端快速部署，保证监控手段和监控方式的多样性。将网络环境对视频数据的影响降到比较低的水平，在网络环境变化的情况下，通信的双方不仅能够感知还能够做出应对策略来保证视频信息的完整性与流畅性。

2.2 系统的设计原则

① 可靠性。只有可靠性高的系统才能满足当今生产和生活的需要。可靠性就是整个系统不被外界环境影响的能力。目前硬件规模、软件规模都是越来越大，哪怕是单一零件或软件功能的丧失都会导致整个系统的崩溃，进而会导致系统功能性的丧失。因此可靠性是在设计、实施和部署过程中最为总要的因素。

② 健壮性。健壮性反映了硬件和软件系统应对异常输入时的处理能力。一个好的设计不仅能够对合理、正确的输入做出系统制定的处理，还要对状况之外的输入有合理的处理方式。外界环境的复杂难免有异常的信号或干扰，对其不恰当的处理方式会带来错误的输出甚至功能性的缺失。

③ 可修改性。技术在不断的进步，需处理的问题也在不断改变，在之后的时间里可能会对整个系统或是某个环节有更高的要求，我们需要在最小代价下来扩展、改善系统的处理能力。因此在设计、实施时要留有足够的空间来承载未来可能发生的变化，比如，在软件的编写过程中需要有良好的规范、统一的接口和详细的说明文档。

④ 出色的人机接口。软件不能仅完成被赋予的功能，还要在人机接口的部分为用户提供最大的便利。人机交互界面要让系统提供的功能直观化、形象化，在

简化执行的步骤的同时更要让执行的步骤具有可预测性。出色的人机接口对于系统功能的顺利执行的速度和准确度都有重要的指导意义。

2.3 视频监控系统的总体框架

系统的总体组成如图2.1：

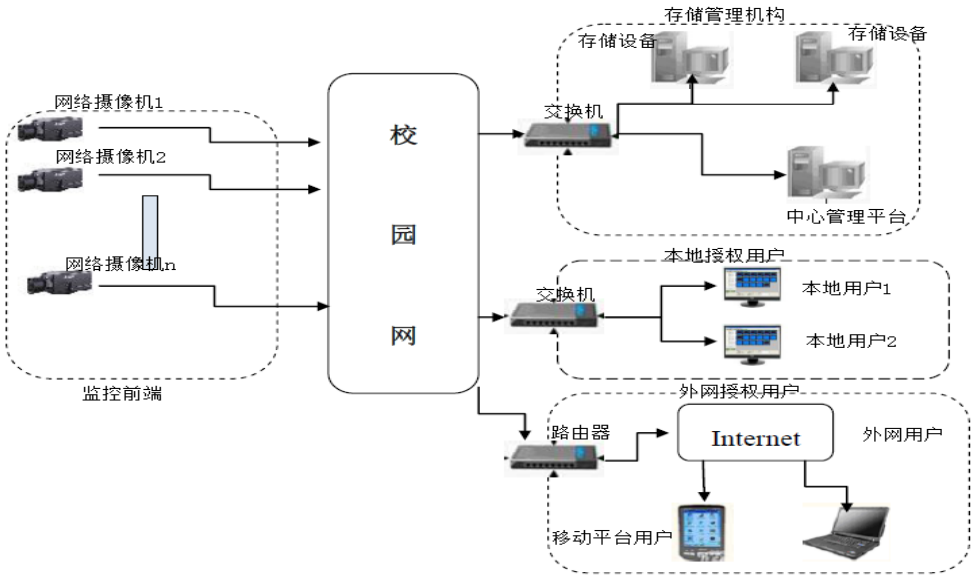

图 2.1 总体组成

Fig 2.1 Overall composition

系统总体可分为客户端和服务器两个方面，两者之间通过网络相连。服务器是嵌入式设备，在本设计中是基于S3C2440的TQ2440开发板，开发板与摄像头连接，组成可采集视频数据的嵌入式设备。服务器在工作时一直处于监听网络连接的状态，这由单独的监听线程予以保证。至少有一个的连接被接受时才进行视频数据的获取，否则处于阻塞中。单个服务器可接受多个连接，可以服务多个客户，但是能有效服务的客户数取决于设备的硬件水平。客户端软件用Qt来编写，单一客户端可连接多个服务器。Qt目前能支持的平台众多，因此客户端不仅能快速部署到Windows、Linux等PC平台，还可以部署到众多的IOS、Android设备上。

2.4 系统的功能模型

系统的功能模型如图2.2：

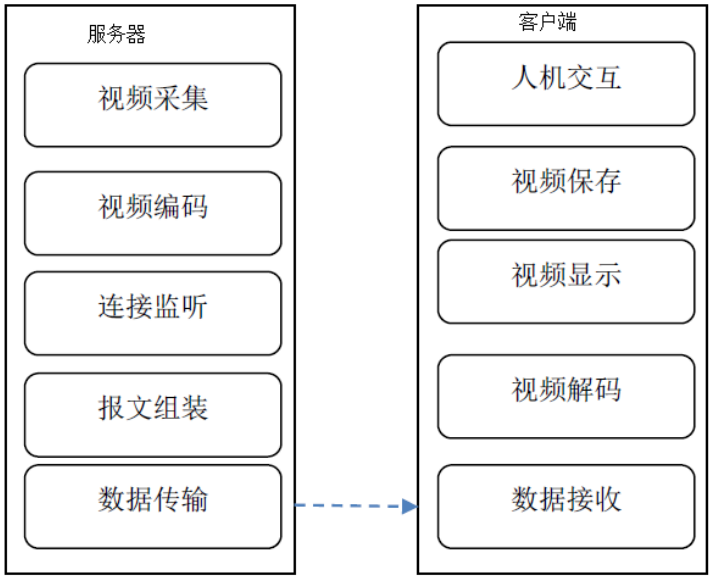

图 2.2 功能模型

Fig 2.2 Function model

服务器是嵌入式设备端，具有的有视频采集、视频编码、连接监听、报文组装和数据传输功能。视频采集模块负责基本图像的获取；视频编码模块及时的将采集来的数据进行压缩编码；连接监听模块用来接受客户端的连接；报文组装模块会将帧信息组装成符合通信协议的报文；数据传输模块负责最底层的数据发送工作。

客户端作为数据的接收方，具有的功能模块有数据接收、视频解码、视频显示、视频保存、和人机交互。数据接收模块负责最底层数据的接收，并对帧接收的完整性和及时性负责；视频解码模块将数据接收模块传入的帧数据进行解码，并送交显示模块；显示模块负责图像的显示；视频保存模块通过响应人员的指令来进行视频数据的保存工作；人机交互模块负责处理人机交互功能的处理，不仅负责响应用户的启动、停止、录像等功能的完成，还要负责对是否存在延迟做出判断。

2.5 视频的获取与处理

2.5.1 V4L2 视频采集接口

V4L2是一个linux设备中涉及视频获取和输出的API框架并与Linux内核联系密切，能够支持许多的USB摄像头和其他设备[14]。V4L2是Video For Linux Api 的第二版，第一版在Linux Kernel2.6.38中予以删除[15]，V4L2修补了许多设计缺陷并最早在2.5.x的内核中予以出现，在2.6.27的内核中予以最终的完善[16]，Linux中相关

的头文件的位置为/usr/include/linux/videodev2.h。

V4L2具有简易的操作流程，一般流程如下：

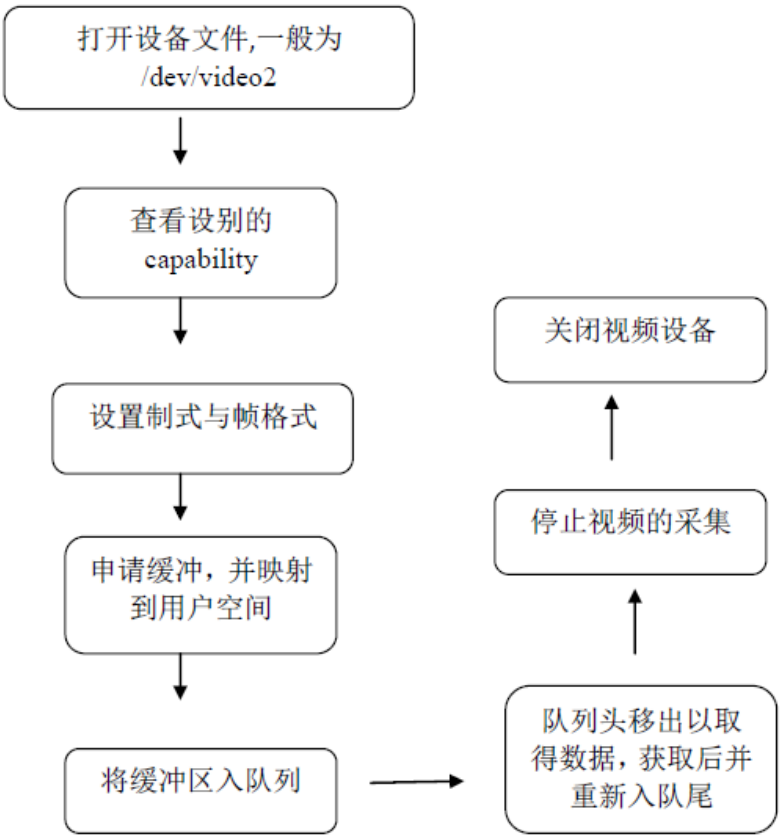

图 2.3 视频采集的基本流程

Fig.2.3 Basic process of picture capturing

① 打开视频设备

在Linux中，设备通常都被当做是文件，使用系统调用open()来打开。open()函数的原型为 int open(const char *pathname, int oflag, …/*mode_t mode*/);

因此在这里变为：

int fd;

fd = open(“/dev/video0”,O_RDWR,0);

fd是返回的最小的未用的文件描述符。

② 设置属性及方式

打开了设备以后要对其进行属性的设置，用ioctl()函数进行更改。ioctl()是类UNIX系统中I/O操作的杂物箱。许多I/O操作都可以用ioctl()来完成。函数原型为：

#include<unistd.h> /*System V*/

#include<sys/ioctl.h> /*BSD and Linux*/

#include<stropts.h> /*XSI STREAMS*/

int ioctl(int filedes,int request,…)[17];

在V4L2的开放中常用的request值为：

1. VIDIOC_REQBUFS：申请内存[18]
2. VIDIOC_QUERYBUF：查询驱动申请的内存区信息[19]
3. VIDIOC_S_FMT：设置当前驱动的捕获格式
4. VIDIOC_QBUF：缓存加入空闲队列
5. VIDIOC_DQBUF：获取已经放有视频数据的空间

涉及到与设置格式有关的结构体是v4l2_format和v4l2_pix_format，该结构用于描述驱动程序与应用程序间的数据格式，如用于描述图像宽度和高度的width和height域；用于描述像素格式的pixelformat域等。

具体设置时需调用ioctl()函数：

```
struct v4l2_format    fmt;
/*把fmt赋予具体值*/
ioctl(fd, VIDIOC_S_FMT,&fmt);
```

③ 申请缓冲并映射

```
struct v4l2_requestbuffers{
_32            count;
enum v4l2_buf_type       type;
enum v4l2_memory         memory;
_u32           reserved[2];
};
struct v4l2_requestbuffers req;
ioctl(fd,VIDIOC_REQBUFS,&req);
```

其中count值就是缓存的数量，即保存着多少帧的视频的帧数。该结构中的memory域记录着采集的方法，分为read()来直接读取和采用mmap()内存映射的方式。其中mmap()的方式是把设备文件映射到内存中，避免了read()方法在用户与内核空间之间不停的复制数据。mmap()的函数原型为：

```
#include<sys/mman.h>
void * mmap( void *addr,size_t len, int prot , int flag , int fd, off_t off );
```

接下来的工作就是将设备文件映射到用户空间中去，涉及到的主要结构体为struct v4l2_buffer，并以VIDIOC_QUERYBUF作为参数调用ioctl()函数。

对队列中视频的每一帧都以VIDIOC_QUERYBUF作为参数调用ioctl()函数。然后映射到用户空间：

mmap(NULL,buf.length,PROT_READ | PROT_WRITE,MAP_SHARED,fd,buf.m.offset);

其中，buf为v4l2_buffer结构体数据，PROT_READ | PROT_WRITE表示映射区可读也可写，fd是该设备文件的文件描述符，MAP_SHARED表示进程对映射区的存储操作将会导致修改映射文件本身。

④ 获取视频数据时需用到以VIDIOC_DQBUF为参数的ioctl()函数，ioctl()函数返回一个v4l2_buffer结构，该结构中的index域记录了返回队列的序号。

⑤ 重新入队列需用到以VIDIOC_QBUF为参数的ioctl()函数。应用程序需将欲置入空闲队列的帧序号通过参数传入ioctl()函数。

⑥ 关闭视频设备

首先使用munmap()函数进行解除映射，关闭文件描述符。munmap()函数的原型如下：

int munmap(void * ad , size_t size);

然后调用函数ioctl的释放函数：

ioctl(fd, VIDIOC_STREAMOFF, &type)

其中type为enum v4l2_buf_type型变量。

2.5.2 FFmpeg 多媒体解决方案

FFmpeg是一个领先的开源多媒体解决方案，全部由C语言编写完成[20]。基本上能够做目前对多媒体所能做到的一切，如解码、编码、转码、复用，解复用、滤波和播放等。而且，它所能支持的领域是很广泛的，不管是一些旧的、古老的格式还是目前前沿的格式都能得到很好的支持。它所包含的工具[21]如表2.1：

表 2.1 FFmpeg 里的工具

Table 2.1 Tools in FFmpeg

工具	功能
ffserver	一个用于实时直播的多媒体流的服务器。
ffplay	使用 FFmpeg 和 SDL 函数库开发出的多媒体播放器。
ffmpeg	一个基于命令行的工具，提供了在不同的多媒体格式间转换的功能。
ffprobe	一个简单的多媒体流的分析器。

FFmpeg所包含的开发库[22]的结构如表2.2：

表 2.2 FFmpeg 中的函数库

Table 2.2 FFmpeg developers libraries

组成部分	功能
libavutil	包含了一系列用于简化编程的函数，如随机数生成器、一些易用的数据结构体、核心的多媒体实用工具。
libavcodec	一个包含了音视频的编码器和解码器的库
libavformat	一个提供用于多媒体容器格式复用器和解复用器的库
libavdevice	一个包含用于抓取的输出、输入设备并渲染成通用多媒体软件功能的函数库
libavfilter	一个提供媒体过滤器功能的函数库
libswscale	一个提供高质量图片缩放与色彩空间、像素格式转换功能的函数库
libswresample	一个高度优化过的函数库，支持数据的重采样功能，并支持格式间的转换

① FFmpeg的解码

FFmpeg的解码流程通常可以用图2.4来表示：

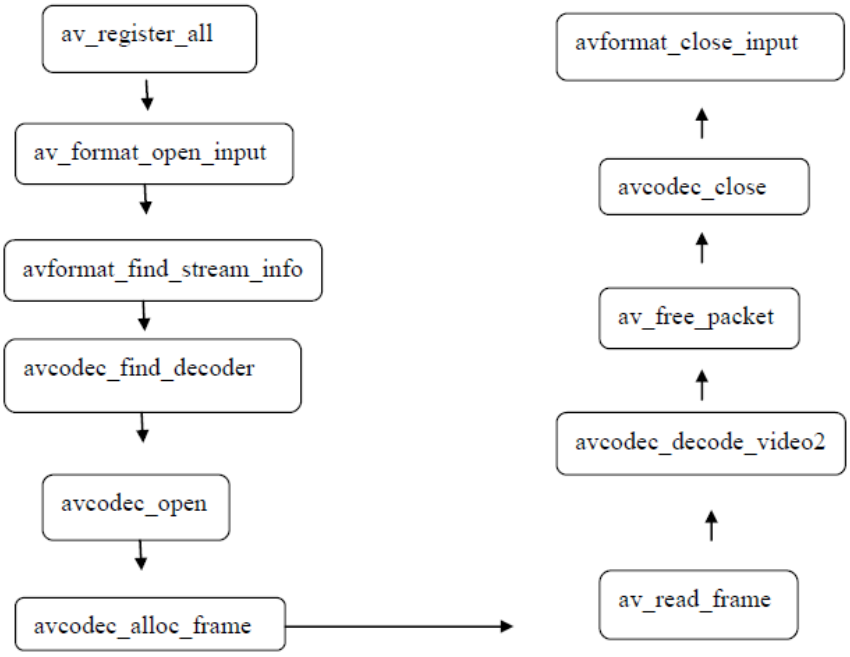

图 2.4 解码流程

Fig 2.4 Process of decoding

av_register_all：用于在程序文件中不仅注册编码器和解码器的库，还包含了所有的文件格式。

avformat_open_input：解复用器读取媒体文件并将其分割成功能上独立的数据块。

avformat_find_stream_info：读取文件并得到文件的流信息，然后为pFormatCtx->streams来填上适当的信息。

avcodec_find_decoder：根据传入的编码ID来找到解码器。

avcodec_open：用传入的AVCodec结构来初始化AVCodecContext结构。

avcodec_alloc_frame：用来申请一个AVFrame结构体并将该结构体的值都赋予默认值，值得注意的是这个函数只是申请AVFrame结构体本身而不为该结构体中的域申请内存。释放该结构用的函数是av_frame_free。

av_read_frame：这个函数返回存储在文件中的信息，分割成以帧为单位的数据，并每次调用都返回一帧。它并不会忽略帧间无效的信息，以次来尽可能的给解码器最大的信息量。

avcodec_decode_video2：这是视频信息的解码函数，将传入的AVPacket结构中的数据解码。

av_free_packet：释放AVPacket中的数据。

avcodec_close：关闭给予的AVCodecContext结构并且释放与其相关的所有数

据。

avformat_close_input：关闭AVFormatContext。

② FFmpeg的编码

编码的简要流程如图2.5：

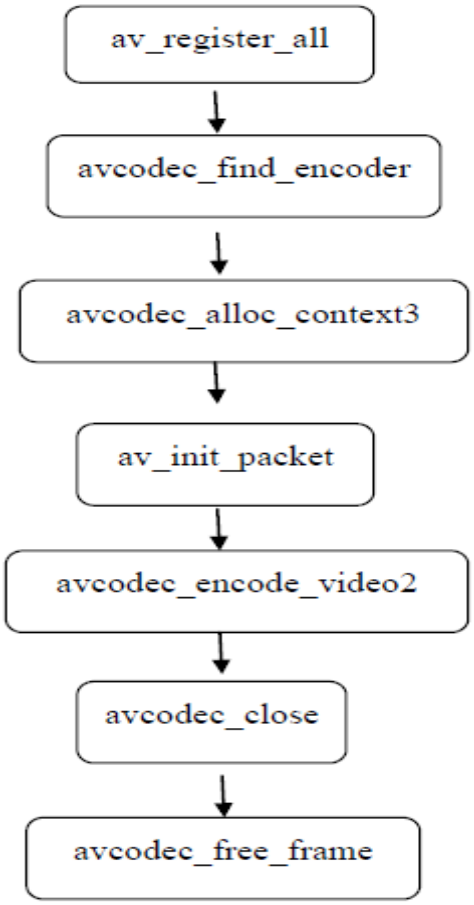

图 2.5 编码流程

Fig 2.5 Process of encoding

avcodec_find_encoder：根据编码ID来寻找注册了的编码器。

avcodec_alloc_context3：分配一个AVCodecContext结构并返回该结构的指针，将其结构成员设为默认值。分配的AVCodecContext需用avcodec_close予以关闭随后调用av_free来释放空间。

av_init_packet：初始化除data和size成员外的AVPacket结构体成员。Data与size必须单独初始化。

avcodec_encode_video2：将视频中一帧的信息进行编码，函数的原型为：

```
int avcodec_encode_video2(AVCodecContext * avctx , AVPacket * avpkt ,
                          const AVFrame * frame , int * got_packet );
```

从frame中得到原始的视频数据并将其写到接下来的packet中，当有数据需输出时会写到avpkt中，该函数执行完毕后avpkt中不总是有数据的，编码器根据需要会重新排序输入的帧信息。

avcodec_free_frame：释放AVFrame结构并且结构体里面动态分配的数据也将随之释放。

③ FFmpeg中重要的结构如表2.3：

表 2.3 FFmpeg 中重要的结构体

Table 2.3 Important structures in FFmpeg

结构	功能
AVFormatContext	该解决方案中最基础的的一个结构，是其他结构和函数操作的基础。描述了媒体文件的构成。是最根本的抽象。
AVStream	是对一个媒体流的描述，如视频流、音频流。
AVCodecContext	提供了很多的编码器必要的媒体信息,用以提供编码、解码器的上下文。
AVCodec	用来存储编解码信息的结构体
AVFrame	存放原始帧数据的结构
AVPacket	FFmpeg 使用此结构来存放编码以后、解码之前的流媒体数据，还有时间戳、时长等附加信息。
SwsContext	存储着图像转换的上下文信息

2.6 开发平台的建立

2.6.1 PC 端开发环境的搭建

PC端主要涉及的是客户端开发的搭建工作，所用的GUI开发框架是Qt4.8（目前最新的版本是Qt5.2.1），这里主要介绍Qt4.8版本的获取、安装及搭建，其他版本或论文中未提及的平台的搭建工作可通过查阅Qt的官方网站（www.qt-project.org）获得。关于Qt更详细的介绍将在第四章中介绍，下面将介绍Qt的获取、安装、配置工作。

① Qt/X11的安装

在Digia的Qt项目中下载Qt4.8的文件 qt-everywhere-opensource-src-4.8.5.tar（获取你所需的版本）。

将当前路径切换到存放文件的目录处，例如”/tmp”文件夹。则：

进入相应目录：

cd /tmp

解压缩该压缩文件：

tar xvf qt-everywhere-opensource-src-4.8.5.tar

执行configure工具：

cd qt-everywhere-opensource-src-4.8.5

./configure

要编译Qt，需输入命令：

make

下面就是Qt的安装命令：

make install

② Qt/Windows,Qt/Mac的安装

这两者的安装与在Linux下的安装类似，这里不再细述，其中在Windows下需要MinGW的支持，它是GCC编译器和GNU工具集移植到Windows平台下的产物，包括许多相关的函数库和可执行程序。

Qt Creator集成了一系列的开发工具，一个可视的GUI编辑器Qt Designer(图2.6),一个API的帮助手册Qt Assistant（图2.7）,一个为翻译人员提供友好翻译界面的Qt Linguist。图2.8给出了集成开发工具Qt Creator的主界面。

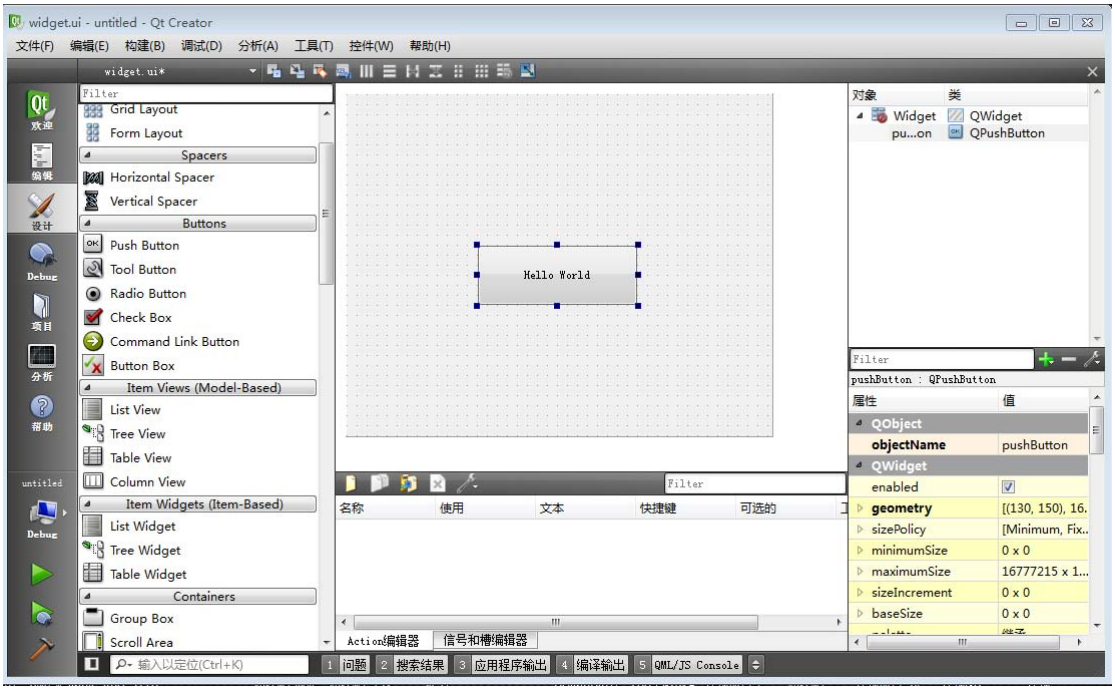

图 2.6 Qt Designer 主界面

Fig 2.6 Main interface of Qt Designer

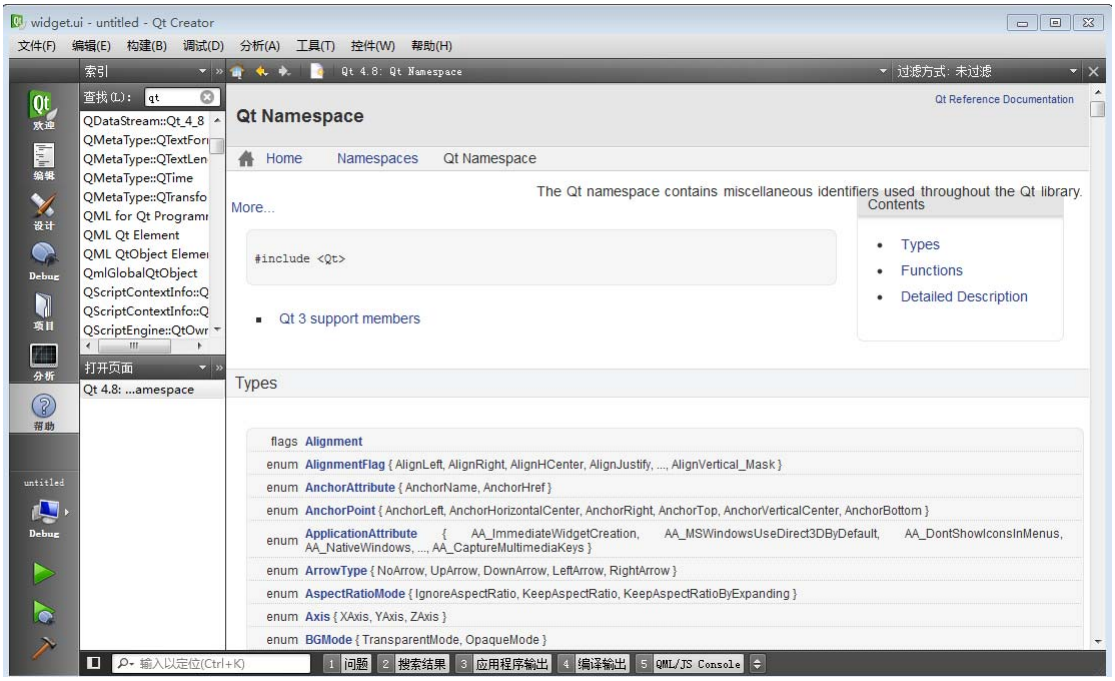

图 2.7 Qt Assistant 主界面

Fig 2.7 Main interface of Qt Assistant

图 2.8 Qt Creator 的主体界面

Fig 2.8 Main interface of Qt Creator

编译工具可以使应用程序的生成变得简单，针对Qt应用程序有许多编译方法，

比如CMake，这是一个第三方的编译工具，可以生成具体平台的makefile文件。开发者一般比较喜欢一些集成开发环境来开发应用软件，目前Visual Studio和QtCreator均对Qt有比较好的支持。此外，还有Qt本身提供的qmake工具。

在安装Qt的时候会将qmake工具一起安装，qmake是一个生成指定平台的makefile文件。它具体使用与平台无关的.pro文件来生成，可以依照生成的这个makefile文件来直接编译成与特定平台相关的可执行文件。qmake工具包含了Qt元对象系统的逻辑规则。比如要生成工程文件client.pro的makefile，则可以输入：

qmake client.pro

然后通过make命令来编译成可执行程序。

2.6.2 嵌入式平台的搭建

① TQ2440开发平台的硬件资源

TQ2440开发平台是根据三星公司S3C2440平台扩展而来，该类平台具有高性能、低功耗的特点。硬件资源如表2.4：

表 2.4 TQ2440 硬件特性

Table 2.4 Hardware features of TQ2440

硬件列表	说明
CPU	Samung S3C2440AL，主频 400MHZ
内存	64MB SDRAM
Nand Flash	256MB Nand Flash
Nor Flash	2MB Nor Flash
串口	一个五线串口
网口	100M DM9000 网卡
USB 接口	一个 HOST，一个 Device
摄像头	130w 像素摄像头，V4L2 驱动
电源	5V 电源供电
音频接口	立体声音输出，可录音

TQ2440如图2.9：

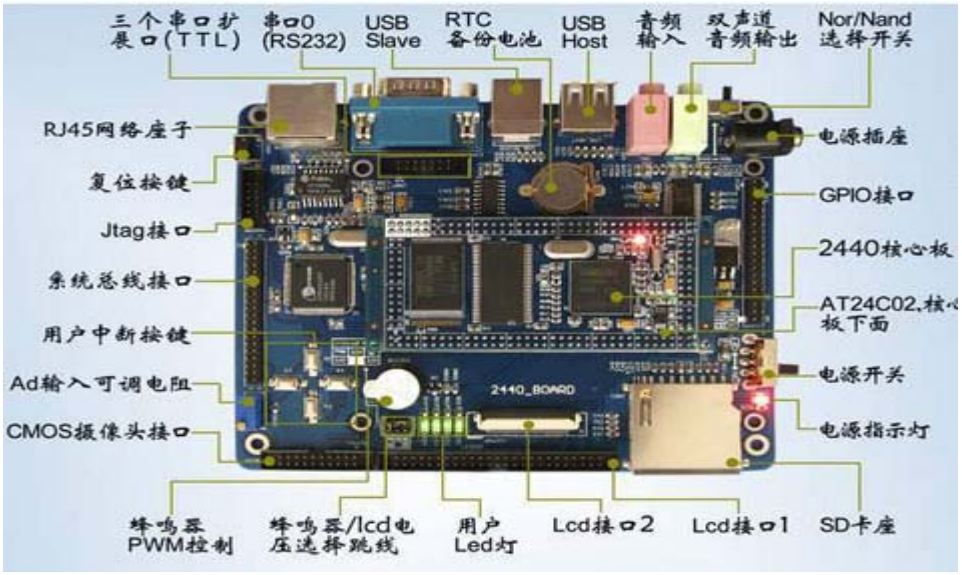

图 2.9 TQ2440 的硬件

Fig 2.9 Hardware of TQ2440

② 建立交叉编译环境

在开发PC机的软件时，可以直接在PC机上来完成整个的编译过程并发布最终的软件。嵌入式的硬件有极高的特殊性，其CPU资源和存储空间都很有限，并不能够安装基于Linux的发行版系统。这就要求开发人员专门为具体的目标硬件来定制操作系统。通过PC机来搭建软件系统，需要在PC机上完成整个软件的编辑和编译，并烧写到设备中，最后在目标板中验证代码。

交叉编译环境的搭建是嵌入式开发的第一个步骤，同时也是非常重要的一个步骤。只有建立起了交叉编译环境，才能在PC机上编译出运行于其他平台的软件。交叉编译器具有多样性的特点，主要取决于体系结构、内核版本的多样性。

GCC是一个由GNU项目组开发的编译器，所能支持的语言在不断的增多，同时也是工具链中的一个核心组成部分，在开源界拥有很高的地位。1.0的版本发布于1987年，之后的不断发展使它支持了多种编程语言，如C++、Objective-C、Fortran、Java等。它所能支持的处理器众多，比如ARM、PowerPC、x86、x86-64、Alpha、Atmel AVR、MorphoSys等。不仅如此，一些比较冷门、不知名的处理器架构在官方的发布版本中也得到了足够的支持[23]。

安装arm-linux-gcc的一般步骤如下：

首先下载arm-linux-gcc-4.4.3.tar.gz，".tar.gz"的命名方式表示此文件不仅是经过tar程序的打包还经过gzip进行压缩[24]。

解压文件：

tar –zxvf arm-linux-gcc-4.4.3.tar.gz;“-z”参数在tar命令中表示通过gzip命令来进行解压缩，”-x”参数在tar命令中表示解包的意思，”-v”参数在tar命令中的意义是在解压的过程中将正在进行处理文件的名字显示出来。”-f”后面跟着的是要处理的文件的名字。需要处理的文件信息很多，解压的过程需要不短的时间，完成后会在本目录下形成名为usr文件夹，将里面的arm文件夹拷贝到系统的软件资源放置区/usr/local中。所用的命令为：

cd usr/local

cp –rv arm /usr/local

最后是修改环境变量，环境变量能够帮我们在Linux系统上做很多事情。修改它可以通过修改特定的文件，其中之一就是修改/etc/profile文件。打开/etc/profile后添加的内容是：

PATH=$PATH:/usr/local/opt/toolschain/4.4.3/bin

export PATH

③ Bootloader移植

Bootloader是一段程序，用于完成系统的初始化[25]。由于嵌入式系统的硬件（CPU、外设等）多种多样，因此需要不同的BootLoader来完成初始化。目前没有一个万能的解决方案，即使是用的比较广泛的工具也要根据硬件环境做相应的修改。目前用的广泛的Bootloader有Blob、U-Boot、vivi等。开发中需要把Bootloader放置于CPU的起始运行地址，这样的话就能保证上电后就开始运行，在ARM中，系统上电的地址通常是0x00000000。

Bootloader的启动过程大多是需要两个阶段的，当然，单阶段和多阶段的启动过程也有不少。在两阶段的启动中，第一阶段主要来完成一些依赖于具体硬件体系的初始化，然后给出下一阶段的入口地址；第二阶段则负责复杂功能的处理，就不可能由汇编语言来实现，通常情况下编程语言是C。

在众多的Bootloader软件中，vivi是其中的一个佼佼者。它是由韩国的一家名为Mizi公司开发的[26]，该家公司专注于为设备提供嵌入式系统的解决方案，尤其注重linux系统下的开发。Vivi支持ARM9处理器。vivi的移植过程中通常的步骤如下：

下载tar.gz源文件。使用tar –zxvf命令进行解压。解压缩后获得的是该工具的源代码。

在Makefile中修改三个宏定义：

ARM_GCC_LIBS宏定义了lib库文件的路径。

CROSS_COMPILE宏定义了可执行文件的具体路径。

LINUX_INCLUDE_DIR宏描述了头文件的文件夹名。

1) 修改nand flash分区信息。通过分区可以重新规划各个模块的存储空间大小。

2) 执行make memuconfig来进行配置。

3) 最后执行make命令来完成编译。

④ 编译内核

内核的移植需要到www.kernel.org上下载内核源代码，目前最新的版本是3.14。内核源码的目录下会有一个隐藏的文件:.config，该文件用于记录用户已选择的内核功能，通过命令”ls -a”可以看到。进行内核功能配置时最通常也是最方便的方式是运行指令make menuconfig，系统会用文字来模拟图形界面的效果，让内核功能的选择变得直观，而且并不需要X Window的支持。挑选完内核功能后依次执行make clean、make dep和make zImage，功能依次是清理中间文件、编译有依赖的文件和最终编译成映像文件。编译完成后，./arch/arm/boot里面就是最终形成的内核文件[27]。

⑤ x264的移植

x264是一个可以将视频流转换成H.264格式的函数库，在GPL协议下发布。x264不仅提供了命令行模式，还提供了格式转化的API，FFmpeg的某些有关于H.264的编解码函数中就用了x264提供的API,所以在用FFmpeg进行有关H.264编解码的时候需要有x264的支持。

交叉编译x264需要之前建立好的arm-linux-gcc工具。在获得了x264的源代码后需要修改配置选项[28]，具体的说就是修改config.mak文件：

```
prefix = /home/bo/x264
exec_prefix = ${ prefix }
bindir = ${ exec_prefix }/bin
libdir = ${ exec_prefix }/lib
includedir = ${ prefix }/include
ARCH = ARM
SYS = LINUX
CC = arm-linux-gcc
CFLAGS = -WALL –I. -O4 –ffast-math –D_X264_ -DHAVE_MALLOC_H –DHAVE_PTHREAD –s –formit-frame-pointer
LDFLAGS = -lm –lpthread –s
AS = nasm
ASFLAGS = -O2 –F elf
VFW = no
GTK = no
```

EXE =

VIS = no

HAVE_GETOPET_LONG = 1

DEVNULL = /dev/null

CONFIGURE_ARGS = ‘—enable-shared’ ‘—prefix = /home/bo/x264’

SONAME = libx264.so.56

default ： ${SONAME}

修改完成后进行保存，接着使用编译命令make来进行编译，并使用make install来进行安装。

⑥ FFmpeg的安装与移植

FFmpeg的安装流程不是很麻烦，这里介绍在Ubuntu、Debian系列Linux发行版中的安装[29]：

1) 安装Yasm：

```
cd ~/FFmpeg
wget http://www.tortall.net/projects/yasm/releases/yasm-1.2.0.tar.gz
tar xzvf yasm-1.2.0.tar.gz
cd yasm-1.2.0
./configure --prefix="$HOME/ffmpeg_build" --bindir="$HOME/bin"
make
make install
make distclean
export "PATH=$PATH:$HOME/bin"
```

图 2.10 安装 Yasm

Fig 2.10 Install Yasm

2) 安装libx264：

```
cd ~/FFmpeg
wget http://download.videolan.org/pub/x264/snapshots/last_x264.tar.bz2
tar xjvf last_x264.tar.bz2
cd x264-snapshot*
./configure --prefix="$HOME/ffmpeg_build" --bindir="$HOME/bin" --enable-static
make
make install
make distclean
```

图 2.11 安装 libx264

Fig 2.12 Install libx264

3) 安装FFmpeg

```
wget http://ffmpeg.org/releases/ffmpeg-snapshot.tar.bz2
tar xjvf ffmpeg-snapshot.tar.bz2
cd ffmpeg
PKG_CONFIG_PATH="$HOME/ffmpeg_build/lib/pkgconfig"
export PKG_CONFIG_PATH
./configure --prefix="$HOME/ffmpeg_build" --extra-cflags="-I$HOME/ffmpeg_build/include" \
  --extra-ldflags="-L$HOME/ffmpeg_build/lib" --bindir="$HOME/bin" --extra-libs="-ldl" --enable-gpl \
  --enable-libass --enable-libfdk-aac --enable-libmp3lame --enable-libopus --enable-libtheora \
  --enable-libvorbis --enable-libvpx --enable-libx264 --enable-nonfree --enable-x11grab
make
make install
make distclean
hash -r
```

图 2.13 安装 FFmpeg

Fig 2.13 Install FFmpeg

在下载好了x264的源码后修改config.mak文件，找到CC和AS项目，并将其修改为”CC=arm-linux-gcc,AS=arm-linux-as”。然后执行make命令[30]。

在安装FFmpeg中按如下方式配置：

./configure --arch=arm --target-os=linux --cc=arm-linux-gcc --enable-cross-compile --enable-shared --disable-network --disable-armv6 --disable-armv6t2 --disable-ffmpeg --disable-ffplay --disable-ffserver --enable-avfilter --enable-gpl --enable-swscale --enable-postproc --enable-gpl --enable-pthreads

make

make install

然后将编译生成的库拷贝到arm平台上。

2.7 本章小结

本章给出了系统的总体设计方案。首先给出了系统的设计目标，随后给出了系统的总体组成和功能模型。从总体上对本系统做出了具体的描述。给出了在嵌入式平台中进行视频和编码的手段：V4L2接口和FFmpeg解决方案。最后描述了建立开发平台的步骤。

3 应用层协议的设计

3.1 TCP/IP 协议与流媒体

目前存在着各种不同的硬件设备上跑着各样的操作系统，但是TCP/IP协议却能够让彼此异构的个体间通信，这点足以体现出网络协议的魅力，尤其是在上世纪90年代开始得到了高速的发展并越来越多、越来越明显改变着我们的日常生活。

TCP/IP协议是一个分层的的结构（图3.1），是多个协议的组合。其中的每一层分别负责不同的功能。在相同的对等层次上，都有对应的协议进行彼此间的通信。

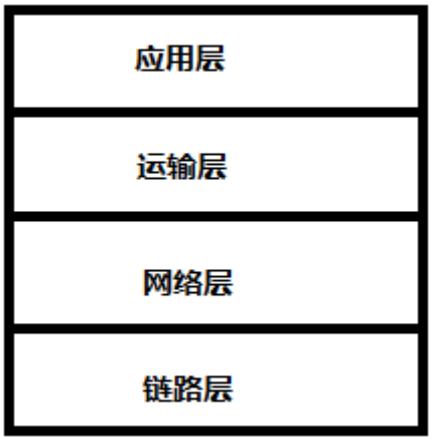

图 3.1TCP/IP 协议

Fig 3.1 TCP/IP protocol

① 链路层描述了不经过路由器连接的主机间的通信方法，描述了传送网络层数据报到相邻主机间所需的协议。并由其来处理物理细节，比如定义了介质间的电子、机械功能和规程特性[31]。

② 网络层主要来处理诸如分组选路之类的数据在网络中的活动。其中最重要的协议有IP、ICMP以及IGMP协议。

③ 运输层有两个重要的协议TCP和UDP协议。它们来具体负责网络上两台主机中应用程序的通信。TCP用来为两台主机提供可靠的通信保证。UDP是一种简单的服务，并不保证数据的可靠传输。

④ 应用层为具体进程负责，定义应用程序间的数据格式。

TCP/IP的层次划分更详细的描述如图3.2：

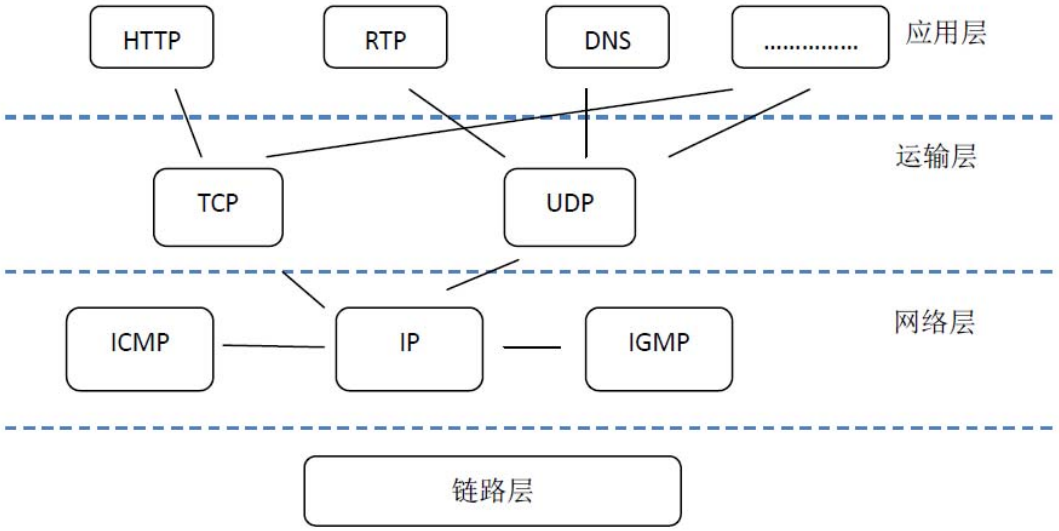

图 3.2 TCP/IP 协议中不同层次的协议

Fig 3.2 Various protocols at the different layers in the TCP/IP protocol suite

技术的不断发展让我们开始有了在公有的互联网平台上传输音视频的需求。但是这个需求有着很大的障碍需要克服，因为多媒体信息与非多媒体信息有着很大的不同。通常来讲多媒体要求的信息量很大。比如一幅分辨率为640*480的照片（低质量），若用不经压缩的RGB格式来传输，即每个像素用24位来描述，则单是这一张照片就需要900KB的存储要求。因此当传输视频数据时则需要更大的数据量。其次是传输多媒体数据时对时间、同步的要求比较高。主要体现在两个方面：第一，网络环境的复杂化使得多媒体分组到达接收端时的时间没有保证，通常是以非恒定的速度到达。如果边接受边还原的话肯定会产生很大的失真，这就要求在接受端需要有一定量的缓存来暂时存储到达的数据，第二，多媒体信息通常包括视频和音频信息，在源端且不能完全保证同步的情况下再经过复杂互联网环境的传输，就更增加了到达接受端时间的不确定性，因此，在接收端需要细致的对音频和视频信息进行同步。

实时流媒体信息在传输过程中会出现数据的重复、丢失、差错等情况，对于这些异常情况的处理策略将很大程度上影响着流媒体传输的质量。在TCP/IP协议中，运输层具体来负责网络上主机应用进程间的通信，它直接对应用层负责，是应用程序在TCP/IP协议族中所能接触到的最底层，运输层收到了数据并进行格式的封装后直接交由网络层进行下一步的处理。运输层主要有两种不同的协议（TCP协议和UDP协议）来满足不同程序中的不同需求，分别代表着两种不同的策略（表3.1）。

表 3.1 TCP 和 UDP 的不同

Table 3.1 The differences of TCP and UDP

运输层协议	特征
TCP	无连接，开销小，不可靠的交付，面向报文，无流量控制，首部开销小。
UDP	有连接，开销大，可靠交付，面向字节流，有流量控制、拥塞控制，点对点通信，首部开销大。

当传输的数据出现丢失的情况下，TCP协议会对出错的数据进行重传（图3.3），这就增加了多媒体数据的延时。在实时流媒体中对时间的要求通常是苛刻的，因此一般的策略是宁愿丢失少量的分组也不要太晚到达的分组。在连续的多媒体流中少量的分组丢失对多媒体的质量影响不大，因而这种处理策略是可以的。

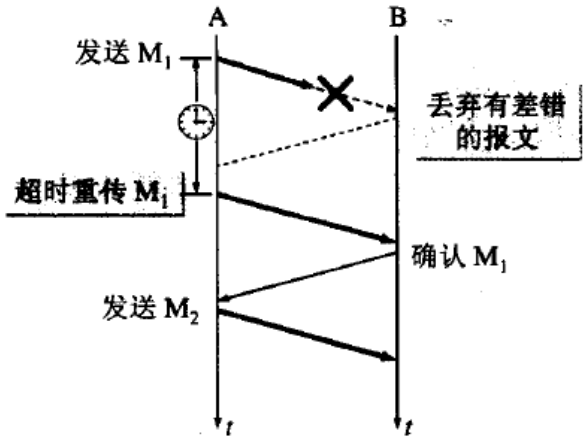

图 3.3 TCP 协议的重传

Fig 3.3 Retransmission of TCP protocol

另一个方面来讲，人们对于多媒体数据的应用也是多方面的，既有实时性优先的需求也有质量优先的需求。因此用TCP协议来传输也是应用的一个方面，这个时候便是宁愿降低实时性也要保证流媒体的质量，这种情况通常会对视频信息在客户端有一个保存的需求。

3.2 RTP 与 RTCP 协议

RTP是一个网络传输协议，用于实时数据的传输，最初在1996年予以公布。RTP协议的一个主要的功能是提供数据报的时间信息，另一个主要的功能是实现不同流的同步。RTP本身并不提供可靠的数据传输机制，这包括两个方面：第一，并

不提供数据的准确接收；第二，并不提供流量控制。目前来讲，大部分的RTP应用都是在UDP协议之上的，但RTP协议本身并没有对运输层做具体的协议限制。流量控制功能是由RTCP协议来提供的，规则要求，参与者周期性的传送RTCP报文，用于在进程之间交换媒体的信息。服务器根据RTCP包中的统计信息来决定是否改变传输策略。RTP数据报格式如图3.4：

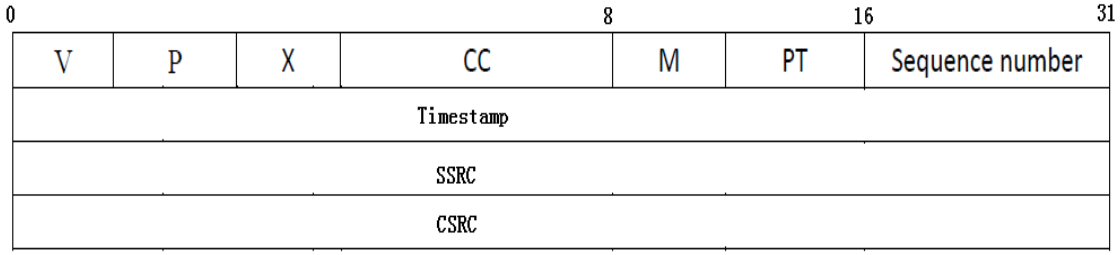

图 3.4 RTP 协议

Fig 3.4 RTP protocol

在RTP报文中，V域指的是版本号，当前的值为2。P域的设置表示末端会有附加的数据填充。X域决定后面是否有扩展的头部。CC域指定了后面CSRC的数目。M域由应用程序决定。PT域定义了载荷的类型。Sequence number是RTP数据报的计数字段，一个数据报的发送对应着字域加一，该域的作用是让接收端重排乱序的包序列。Timestamp域记录着抽样时间，随着包的发送而增长，该域的作用是保证不同流间的精确同步和防止包的抖动。SSRC域标识了一个确切的同步源，接收端以此来识别不同的RTP会话。CSRC域记录了所有有贡献的源，该域的作用是用于混流后的记录。

RTCP有五种数据包的格式，分别是发送端报告、接收端报告、源描述、结束传输和特定应用。发送端周期性的向接收端发送发送端报告，内容包括绝对时钟时间、 SSRC、该流中的包含分组的数目和字节数目、RTP时间标志等，其中的NTP绝对时钟用于同步不同的流。接收端周期性的向所有点发送接收端报告，接收端报告主要有两个作用：第一，发送端可以了解当前网络环境的拥塞状态，并据此来调整数据的发送速率；第二，可以让所有的参与者通过调整RTCP的发送频率来更改RTCP报文在分组中的百分比。源描述报文承载着有关会话成员的信息。结束报文用来通知离开。特定应用报文功能的定义由应用程序自己定义。

典型的工作机制是应用程序将媒体信息打包成RTP数据报进行发送，通常来讲，RTP使用的端口号是偶数的。在数据的发送方和接收方都会周期性发送RTCP数据报，发送频率会根据参与方的数量做调整。RTCP提供数据的统计信息，发送端可

以得到丢包率和RTCP本身的往返时间（RTT），这进而影响发送端的质量调整策略。RTCP使用的是跟在RTP端口后的奇数号端口。

3.3 协议概述

视频监控系统和通常意义上的流媒体有一些不同，在音视频会议、混频器、音视频直播等应用中需要不同媒体流之间的精确同步，比如，在音视频会议中，音频和视频中RTP和RTCP的传输可以通过两个UDP端口对，也可以使用多播地址，各自有独立的SSRC域，使用了相同的规范名，才将两个会话进行耦合，在RTP报文传输期间需要不停的向接收端或所有参与者发送RTCP发送端报告或RTCP接收端报告，发送端需要根据丢包率或RTCP报文的RTT，并依据某些策略来进行质量、速度的控制，不仅如此，所有参与者也要依据收到的RTCP报文来调整RTCP的发送频率，RTP与RTCP两者之间的协调保证了所有参与者间的流量平衡。

视频监控系统是一个典型的C/S模式，服务器负责采集和传输，客户端负责视频信息的控制，这两个端点并不是一个对等的关系。此时，监控系统体现的更多的是“控制”而不是“最大能力的传输”。单纯的根据丢包率来降低发送速度在实时监控中是毫无意义的，这只会加剧时间的延迟。服务器对于丢包率或时延临界点的选取可能并不符合客户端的要求，再者，降低码率带来流畅性改进的同时会使视频的质量出现下降。服务器对于清晰度和流畅性之间平衡的掌握可能与客户端用户的要求并不一致。丢失率上升后，最好的做法是由客户端来决定是选取降低码率或帧率还是重传丢失的报文。

对视频质量的评价应由客户端来做出，采用的具体处理策略也应由客户端来做出，客户端根据丢包率或用户的体验来做决定，这给整个监控系统增添了灵活性。这就要求服务器提供一个灵活的接口用来服务。因此针对视频监控这种需求，扩展RTP/RTCP协议族或者重新制定一个简洁、高效的通信协议就显得很有必要。扩展RTP/RTCP协议主要通过定义RTCP协议中的APP分组来实现，以期实现更灵活的功能。

3.4 协议的格式与规则

必须对整个通信过程的方方面面作出约定才能实现应用进程间的相互通信。比如数据信息和控制信息的功能划分及格式约定、双方通信交流的规程、使用的同步方式等。这些都是由协议的制定来实现。

3.4.1 TCP 部分

在服务器的设计中采用了单连接的方式，而在传输帧数据的同时又要传输控制信息，因此协议的制定、通信的流程必须要做到足够的简洁才能避免控制信息

和数据信息间的干扰，才能有效的保证服务器和客户端间高效而无差错的通信。其中TCP的控制协议如图3.5：

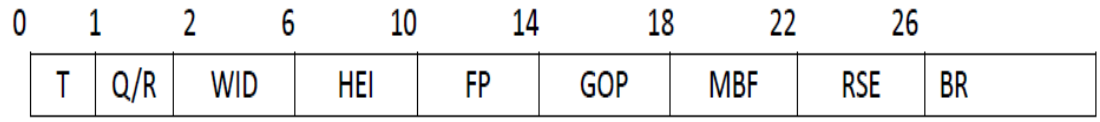

图 3.5 控制协议

Fig 3.5 Control protocol

① T：此字段是“协议类型”字段，用以区分控制字段和数据字段。

② Q/R：此字段是“要求/应答”字段，用以区分的报文类型是“请求”、“应答”和“请求延时”：

1) “请求”报文在刚建立连接后必须由客户端向服务器发出，此时之后的 WID、HEI、FP、GOP、MBF 和 BR 字段有效，以这些参数来作为需求向服务器请求多媒体数据。客户端发送后，服务器必须以“应答”报文的方式来向客户端传输服务器方面可接受（也就是欲传输的）多媒体参数类型。之后的任意时间，客户端都可以向服务器发送“请求”报文来要求更改媒体信息，同样，服务器必须以“应答”报文来响应。

2) “应答”报文是由服务器向客户端发出的，来作为客户端“请求”报文的应答。在此报文中，之后的 WID、HEI、FP、GOP、MBF 和 BR 字段有效，以此来描述此次连接的多媒体参数。

3) “延时”报文是由客户端根据延时评价策略做出需要延时决定后而向服务器发送的报文。“延时”报文发送后服务器必须向客户端发送“应答报文”来回应，“应答报文”里面记录着服务器调整后媒体的参数描述信息。

③ WID和HEI：用以描述服务器端发出视频的原始宽度和高度。

④ FP：FP中记录的是视频的帧率，即每秒显示的帧数。

⑤ GOP：GOP中记录的是图像组的长度。MPEG或H.264编码将帧分为I，P，B三种帧类型。I帧没有任何依赖，是单独图像压缩后的数据，P帧是前向预测帧，对其进行解码需要依赖前面的I或P帧，B帧是双向预测帧，对其进行解码需要依赖于前面的I或P帧和后面的P帧。GOP就是描述的这样的一个分组的帧数。

⑥ MBF：此字段描述的是AVCodecContext结构中的max_b_frames成员。表示的意义是在两个非B帧间所允许插入B帧的最大数目。就提高压缩率的角度来讲该值应该设的大一些，从提高实时性的角度来讲该值应该小一些。

⑦ RSE：作为保留字段。

⑧ BR：记录的是码率。即每秒钟所传输的比特率。

当T表示“数据字段”时，报文所遵循就应该是数据字段的报文格式，如图3.6：

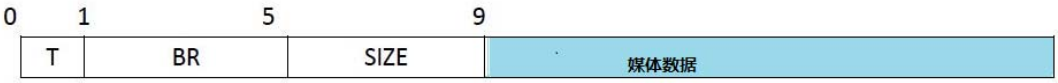

图 3.6 数据协议

Fig 3.6 Data protocol

① T：该字段是“协议类型”字段，用以区分是数据字段还是协议字段。

② BR：BR记录的是码率。客户端用此字段与目前记录的码率进行比较，如果不相等则丢弃该媒体数据。

③ SIZE：SIZE字段描述了之后媒体数据的字节大小。客户端根据SIZE来接受多媒体数据。

④ 媒体数据紧跟在SIZE后，用以承载帧信息。

因此作为TCP连接部分的数据交互流程如图3.7：

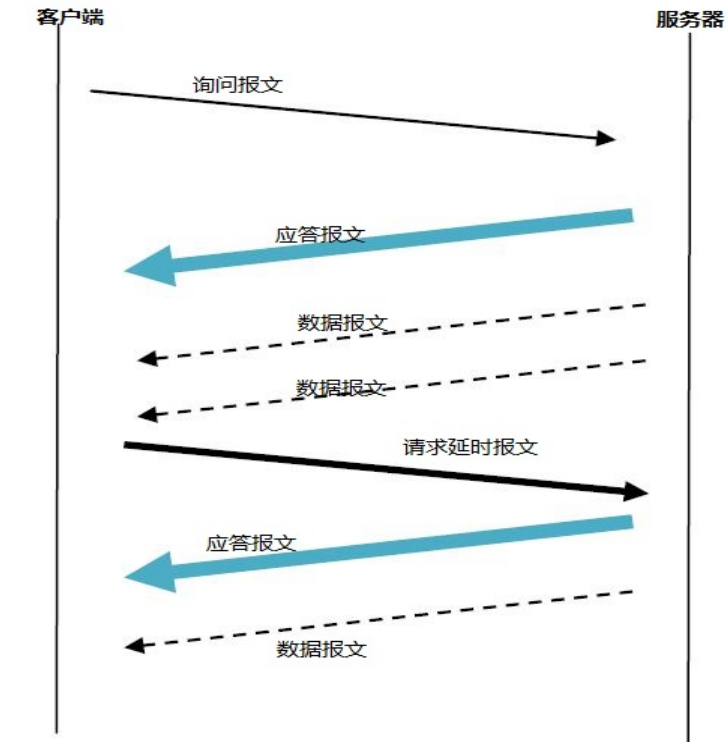

图 3.7 数据交互

Fig 3.7 Data exchange

3.4.2 UDP 部分

在用UDP协议进行数据通信的过程中原则上遵守“宁愿丢失少量的报文也不进行重传”的约定，但是为了让服务器有更好的扩展能力，服务器制定了重发的策略，但是最终是否重发由客户端来决定，并不强迫重发。在协议的设计的部分与用TCP来传输的协议有相似的部分也有不同的部分，这主要是由于UDP协议本身的特点所造成的：一是UDP是无连接的、二是数据包到达时会出现乱序的情况。格式如图3.8：

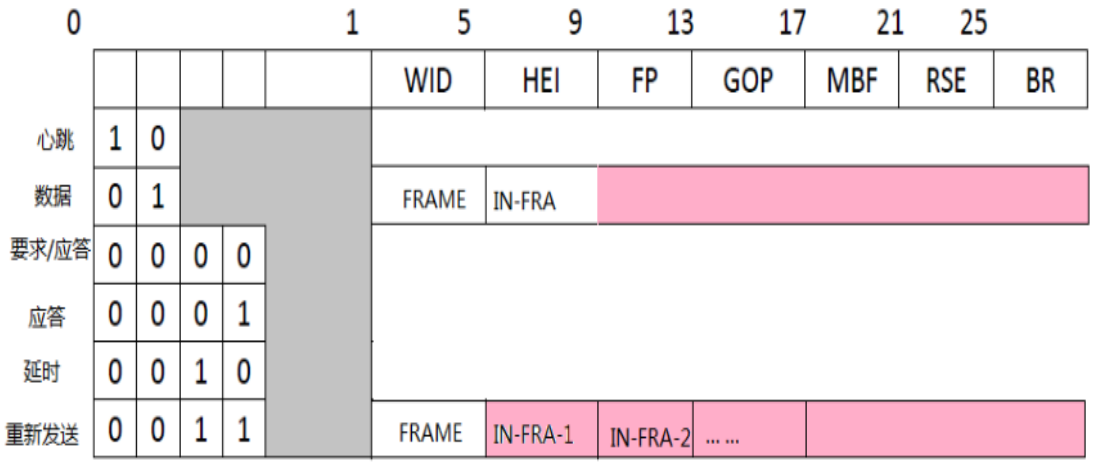

图 3.8 协议格式

Fig 3.8 Protocol format

① 心跳：“心跳”数据包是由客户端发送到服务器的报文，用以表述客户端的在线。

② 数据：“数据”报文是由服务器向客户端来发送，里面承载着多媒体数据。FRAME字段描述的是帧序号，IN-FRA描述的是帧内编号。IPv4和IPv6都有一个必须支持的最小的数据报大小，叫做最小重组缓冲区大小。其值分别为576字节和1500字节。因此在应用层而言我们不能确定对方能否支持接收大于此值的数据报[32]。如果UDP一次性发送了高字节的数据报就会极有可能导致分片，在面向字节流的TCP协议中操作系统会根据策略将数据报进行切割，但是UDP协议却没有此种处理。每一帧的字节多则几万字节，这样大的数据量必须经过切割后再通过UDP协议发送，导致的结果是一个帧的由多个UDP数据报进行传输，这就是帧内序号（IN-FRA）的作用。

③ 要求/应答：“要求/应答”报文是由客户端向服务器来发送，与TCP中的“请求”报文类似，功能是在连接建立时或连接中进行媒体信息交互。在该报文中，

WID、HEI、FP、GOP、MBF、RSE和BR字段有效。服务器收到后将发送“应答”报文进行回应。

④ 应答：“应答”报文是由服务器向客户端发送的，用来通知媒体的信息。可作为客户端“请求/应答”的回应也可作为客户端“延时”报文的回应。该报文中WID、HEI、FP、GOP、MBF、RSE和BR字段有效。

⑤ 延时：“延时”报文是由客户端发出的，用来表示客户端在最近的时间内没有接受到足够量的帧，需要服务器来降低码率。服务器必须向客户端发送“应答报文”来回应，“应答报文”里面记录着服务器调整后媒体的参数描述信息。

⑥ 重新发送：该种报文是由客户端向服务器发送，用来记录服务器需要重发的帧号和帧内序号，由FRAME和IN-FRA-n(n表示数字)来标定需重发的帧数据。在不需要重传的逻辑中此种报文并不使用。

3.5 协议的特点

新制定的协议具有以下特点：

① 可以将此协议作为完整的通信协议在客户端和服务器间使用，也可以将该协议中的非数据部分作为RTCP中的APP分组，数据部分仍可以采用RTP协议发送。前者可以获得针对监控的最简洁的接口，后者能够在保留RTP/RTCP传输方式的同时带来新的扩展功能。

② 将媒体信息质量的评价权移至客户端。这样就使得视频的控制具有极大的灵活性，客户端可以通过丢包率或用户的具体体验来做出是否改变媒体信息的决定。

③ 客户端依赖于网络地址来分流。RTP/RTCP的处理中因为有SSRC域的存在使其可以不依赖于网络地址来分流，但是，监控系统中客户端与服务器不是一个对等的关系，添加SSRC域只会增加处理的复杂度。

④ 将时间戳与帧序号合并。取消了RTP报文中的timestamp域，将时间信息交给新协议中的帧序号域来标识，并以帧率为单位。在不要求精确时间的情况下可以减少处理的难度。

⑤ 简洁、灵活的处理方式。没有RTCP中统计信息的交流，降低了服务器和客户端两方面的处理难度，在认为需要更改媒体信息时，客户端直接向服务器发送指令。

3.6 本章小结

本章首先在叙述TCP/IP协议时给出了流媒体的概念，并介绍了目前应用广泛的RTP实时传输协议和RTCP传输控制协议，给出了两者的作用、特点。之后制定

了针对视频监控系统设计的简单、有效的协议，并给出了该协议的特点。

4 嵌入式平台视频服务器的设计与实现

4.1 服务器端设计概述

4.1.1 Linux I/O 模型

类Unix操作系统中主要有四种常用的I/O模型，分别是阻塞I/O、非阻塞I/O、I/O复用和异步I/O。

① 阻塞I/O模型是非常常见的一种模型，在这样一种方式中，某个系统调用可能会导致阻塞，直到出现了某种异常或是完成了完整或部分数据的传输。这是应用程序中非常有效的一种策略，避免了大量的上下文切换[33]，处理方式如图4.1。

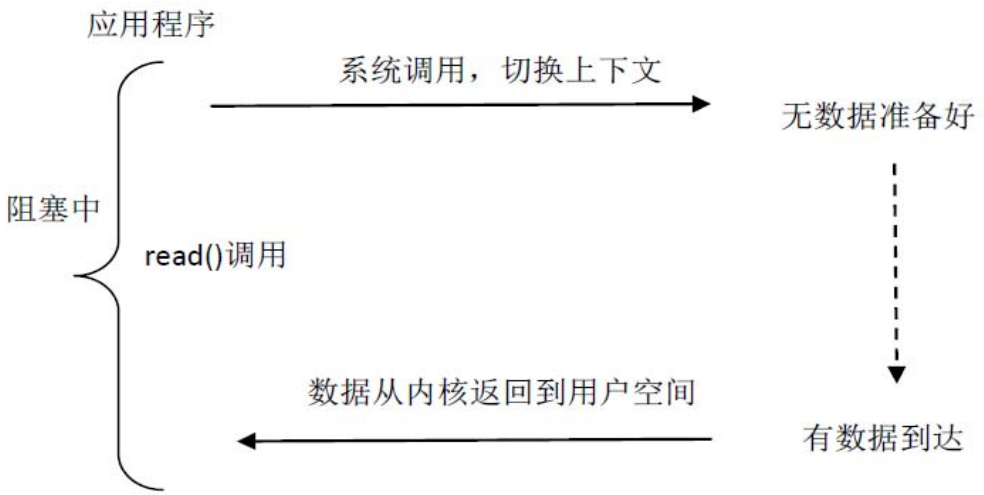

图 4.1 阻塞 I/O

Fig 4.1 Blocking I/O

② 非阻塞I/O（图4.2）与阻塞I/O的一个根本区别是在阻塞I/O中会引起阻塞的情况下并不会在非阻塞I/O中引起阻塞，此时会返回一个错误。因此在一次调用满足不了需求的情况下，应用程序需要多次调用。表面上效率不高，但是当服务器需要处理大量连接的情况下，非阻塞I/O对于某连接的状态是反应最快的，避免了阻塞在一个套接口上导致的处理器时间的浪费和其他接口的处理延时。因此有时不得不使用非阻塞I/O，虽然多次的调用会导致频繁的上下文切换。

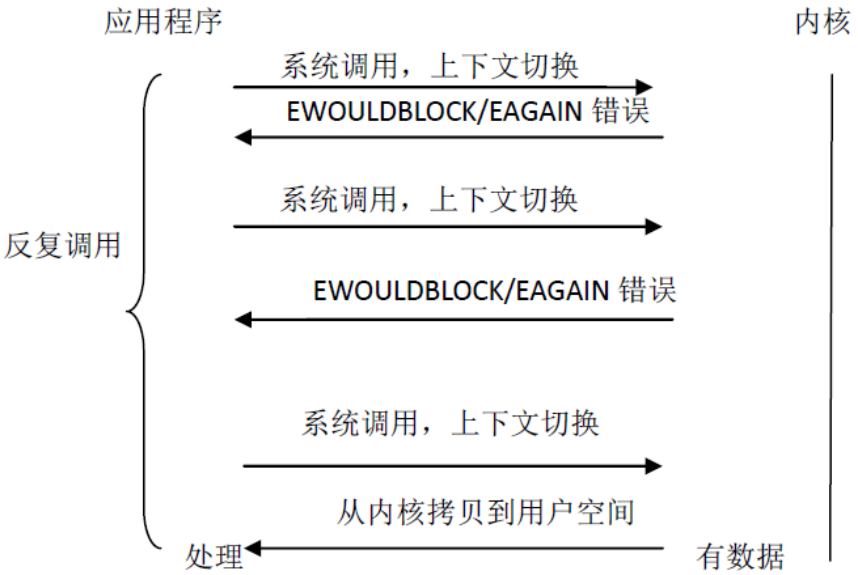

图 4.2 非阻塞 I/O

Fig 4.2 Nonblocking I/O

③ I/O复用模型中使用了套接字集，并通过select()和poll()这两个函数来完成整个套接字集的状态查询，此时，应用进程会阻塞在套接字集上而不是单个的套接字上，这种策略使得应用程序对套接字集状态的反应就会比较快。带来便利的同时也相应导致了I/O复用的弱点：完成一次套接字的处理需要两个而不是单个的系统调用[34]。

④ 异步I/O模型（图4.3）并不会立即得到函数调用后的结果，比如在执行read()系统调用的时候，调用立即返回，返回的情况下就表明请求已成功发起。当内核有数据到达并后台完成读操作时应用程序可以处理其他的事物。响应时，会产生一个信号或回调函数来完成I/O操作。

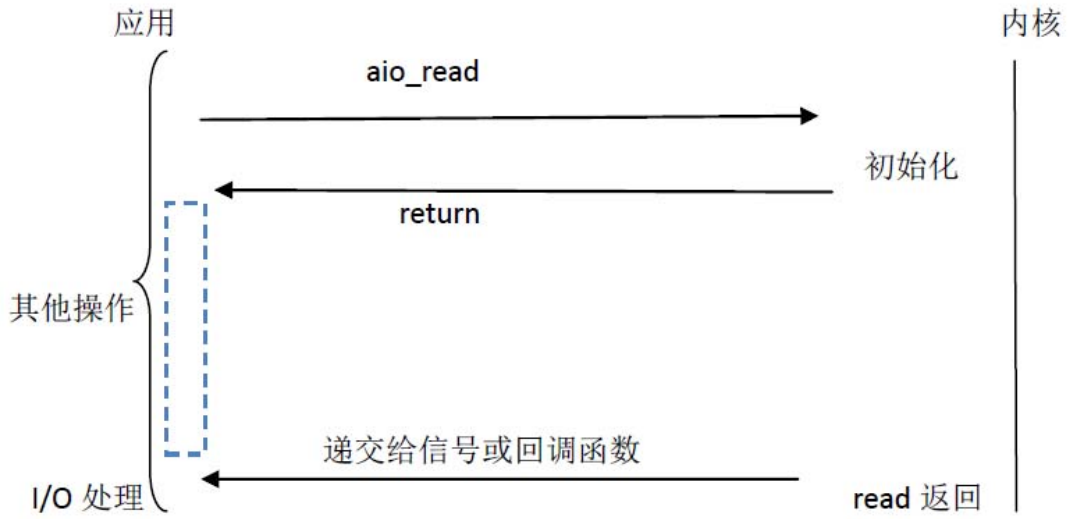

图 4.3 异步 I/O

Fig 4.3 Asynchronous I/O

4.1.2 POSIX 线程

线程是应用程序中的一个基本逻辑，也是能被操作系统调度的最基本单位。POSIX线程有时也被叫做Pthread，在1995年得到了标准化。在许多类遵循POSIX规范的Unix平台中都被广泛的支持，如FreeBSD、Mac OS X、Solaris、OpenBSD和GNU/Linux。甚至在微软的Windows系统中能看到它的踪影。POSIX线程标准定义了一系列的数据结构和函数，使用时需要POSIX线程运行库和相关的头文件pthread.h。

在POSIX规范中，线程的创建函数为pthread_create()：

#include<pthread.h>

int pthread_create(pthread_t * id ,const pthread_attr_t * attr , void *(*fun)(void *), void *ar);

每个线程都有一个类型为pthread_t的线程ID来标识，一些平台中与unsigned int类型相同，但是不保证所有的平台都有此种约定。第二个参数描述的是线程的类型，如分离状态、线程栈的大小等，不过通常情况下将其置为NULL，表示采用默认值。fun给出了线程函数的初始地址，ar作为参数传入线程函数中。

进程内的线程共享该进程的整个内存空间，这使得线程要比进程更加容易的相互通信，当然这要在一定的同步机制下。

同一进程的线程间共享的有：

① 全局变量

② 文件描述符

③ 进程指令

④ 当前工作目录

⑤ 信号处理函数

⑥ 用户ID和组ID

线程间有相同也有存在差异的地方，每个线程都有各自的：

①主要用于存放返回地址和局部变量的栈

②线程ID

③寄存器组合

④信号掩码

⑤优先级

⑥错误值errno

多线程的使用能够最大限度的改善异步事件的处理，应用程序通过将任务分解从而提高程序的吞吐量，因为即使有些线程在运行中不得不阻塞，其他的线程仍可以运行。尤其对于交互式程序而言尤为重要，多线程通过将响应逻辑和处理逻辑的分开来改善响应时间。

线程的退出函数为：

#inlude<pthread.h>

void pthread_exit(void * ptr);

参数ptr是一个无类型指针，其他线程用函数pthread_join()能获得这个线程返回值。

int pthread_join(pthread_t th,void ** ptr);

默认的情况下，线程的状态将会被内核保留，直到另外的线程对其调用pthread_join()。通过执行pthread_detach()或创建线程时修改线程属性可以改变这一默认行为：

#include<pthread.h>

int pthread_detach(pthread_t id);

若成功则返回0，不成功时返回错误编号。执行后tid对应的线程处于一个独立的状态，它在终止时，它的存储资源将被立即释放。

初期的UNIX系统中就已有信号的概念，但是初期UNIX的信号机制主要面临了两方面的问题：第一是版本分裂，各个UNIX版本之间的信号模型并不兼容，这阻碍了平台之间的可移植性；第二是信号机制并不可靠，会出现丢失和其他异常的情况。POSIX.1的标准对信号机制做了统一，为UNIX及类UNIX系统间的移植打下了基础。

应用程序一般都需要处理信号。信号模型使应用程序具有了处理异步事件的能力，但是哪怕是在单线程的设计中，处理这类情况也是非常困难的。在多线程应用程序与信号的交互中需要格外的注意。

线程与信号的模型中有两个基本点：一是线程中信号的处理方式是共用的，这意味着当任何一个线程更改了某信号的处理方式或处理函数后，所有处于同一进程的多个线程都将接受这种行为方式的改变；二是每个线程都可以自主选择阻止一些信号。当信号被接收时，任何线程都有可能接收到，所以应用程序不应对信号的接收线程做出假定。

线程模型中用于屏蔽信号的函数为pthread_sigmask()，该函数实现的功能与进程中的sigprocmask()类似，只不过前者用于多线程的环境中。函数的原型为：

int pthread_sigmask(int how, const sigset_t *set , sigset_t * old);

how值描述了修改的方式，比如阻塞、解阻塞等。set是传入的等待处理的信号集，old返回函数执行前的信号屏蔽字。

4.2 服务器逻辑

4.2.1 TCP 服务器设计与实现

并发服务器结构如图4.4：

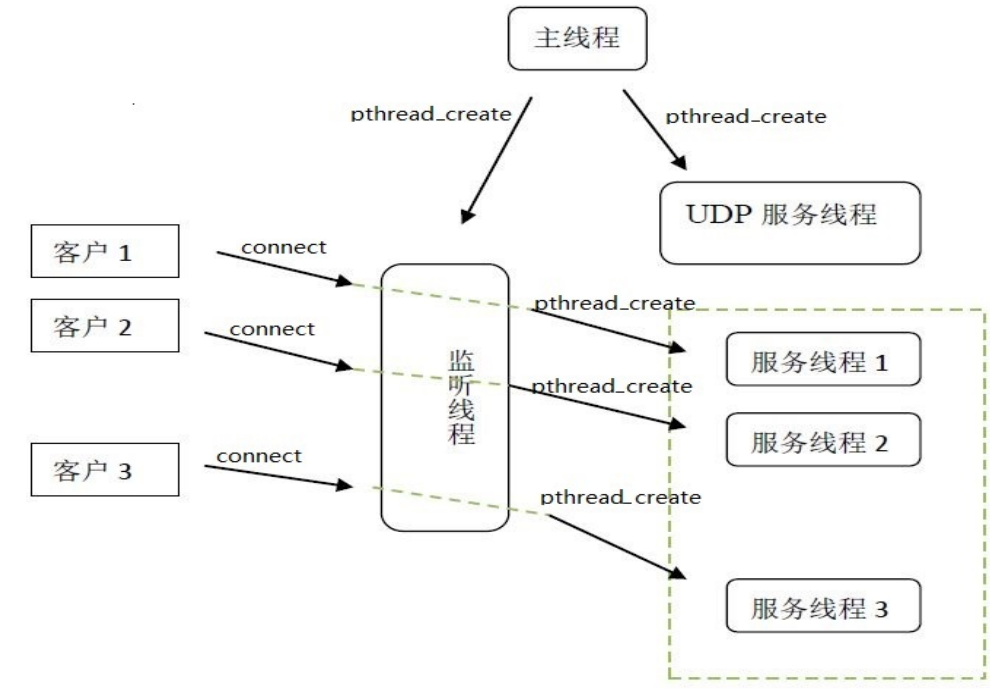

图 4.4 服务器结构

Fig 4.4 Server architecture

① 并发TCP服务器的可行性分析

主线程创建出一个单独的线程来接受连接，成为监听线程，使用的是阻塞接口。之后又创建出另一个单独的线程用于UDP连接。监听线程接受客户的连接并对于每个连接创建出一个单独的服务线程予以处理。

多线程一般来讲比多进程拥有一些优点，比如fork()是昂贵的，即使当今使用写时复制的技术来避免在子进程中复制所有的数据空间，fork()来创建进程仍然比pthread_create()创建线程显得昂贵。再者，多进程间的通信同步机制虽然有很多（unix域套接字、FIFO、管道、信号量、消息队列等）。但是进行进程间的同步仍然要比线程间同步复杂不少。同一进程里的所有线程共享进程的所有信息，如全局内存、堆、栈、文件描述符和程序文本。当然，在提供同步简易性的同时也带来了同步问题。

目前比较流行的服务器模型有Event Loop +Thread Pool的多线程模型（图4.5）或是Event Loop + Non-blocking I/O的单线程模型。

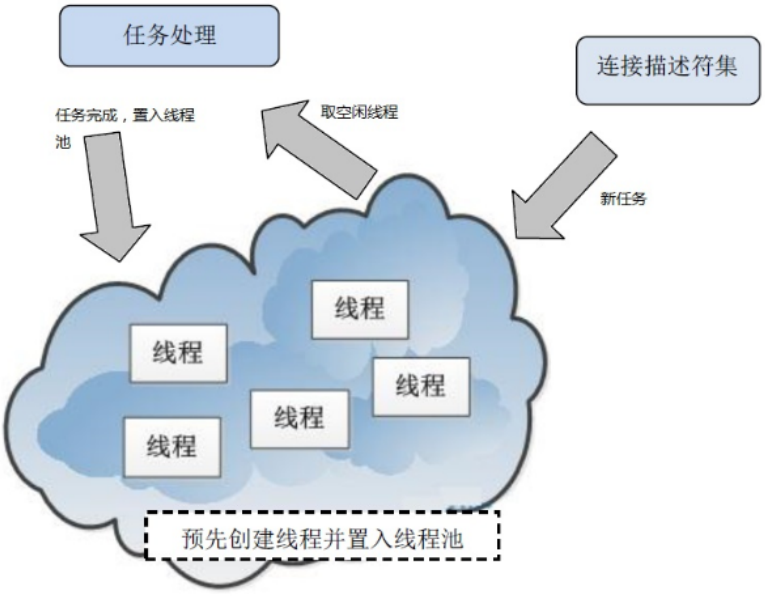

图 4.5 任务轮询和线程池

Fig 4.5 Event loop and Thread poll

每个任务对应的是一个线程的一段时间，任务完成后并不销毁，而是等待新任务的到来。该种模型下需要应用程序在初始时就创建大量的任务线程，并置入空闲队列。这种处理方法避免了大量线程的创建和销毁，节约了cpu时间。

Event Loop +Non-blocking I/O模型是单线程模型，全部的逻辑处理都在单个线

程中完成，非阻塞接口的使用让服务器在“处理任务”和“接受新任务”两者间取得平衡。该种模式目前用的很广泛，能够避免多线程模式中线程切换所耗费的cpu时间。

以一个连接对应一个线程的方式在连接数较少的情况下非常有效，但是当连接数很大的时候就显现出了很大的弊端。当并发数很多的情况尤其是长连接比较多的情况下存在着很严重的资源浪费，连接的存在并不表明任务处理需求的存在，在空闲的时间仍然占用了一个完整的线程资源无疑加重了服务器的负担。

任何的设计都是建立在一定的条件下，任何的优化也都是相对于某一具体背景而言。嵌入式系统的硬件环境一般要比PC端弱，更不比功能型的专用服务器，就应用来讲也很少应用于高并发的环境。Event Loop +Non-blocking I/O的单进程模型在实际的编程中有很大的难度，很多业务并不适合业务逻辑的分片。Per-connection-Per-thread（图4.6）方案由于自身简洁的特点在该种情况下拥有很大的应用空间。该方案能够简化服务器的处理，在连接数较少的情况下具有很高的效率。

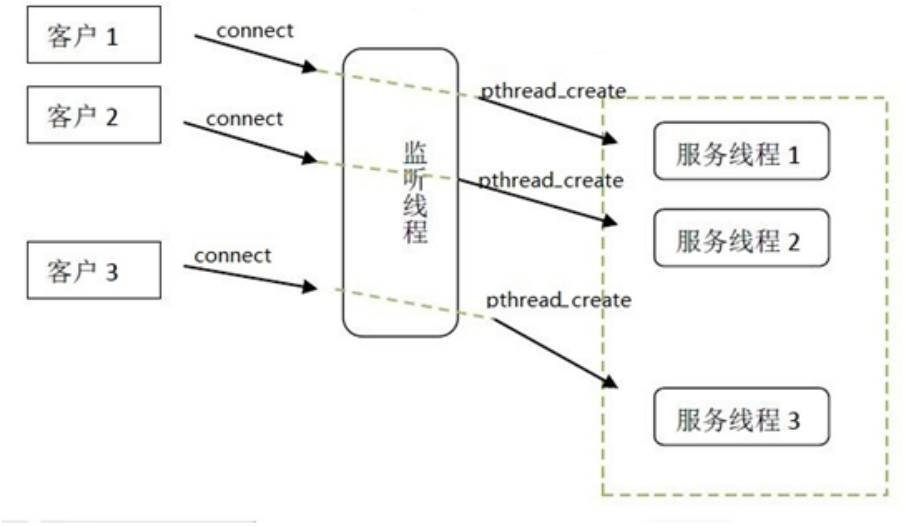

图 4.6 每个线程一个连接

Fig 4.6 Per-Ponnection-Per-Thread

② 并发TCP服务器的线程功能

1) 主线程，主线程主要的作用是完成一些初始化并周期性采集图像，当获取图像的时候将其压缩为 H.264 格式，随后放入全局数据队列：

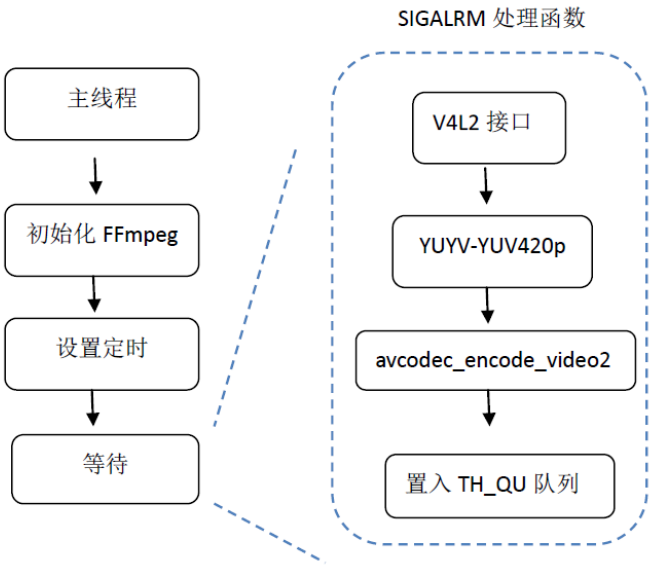

图 4.7 主线程

Fig 4.7 Main thread

用函数setitimer()来执行定时功能，它可以实现Linux下很精确的定时控制。函数原型[35]如下：

int setitimer(int type, const struct itimerval *value , struct itimerval *ovalue);

涉及到的结构如下：

struct itimerval{

struct timeval it_interval;

struct timeval it_value;

} ;

系统在it_value后产生SIGALRM信号，之后的每隔it_interval的时间都将产生SIGALRM信号。timeval是个粒度为微妙的结构，因此理论上该定时函数的粒度是微妙（更主要的还要看系统的时间粒度）。

在用setitimer()注册后，每隔it_interval的时间就会产生SIGALRM，该信号在闹钟超时后就会发出。alarm()函数也会产生此种信号，但是alarm()的实现定时的粒度较大，它的参数是个无符号整形，单位是秒，因此只能实现以秒为级别的定时。SIGALRM信号出现时能选择的处理方式有三种：忽略、捕捉和默认。如果选择的处理方式是捕捉，也就意味着接收到该信号时将执行一个用户函数。这里将该信号连接到一个处理函数sig_alrm()函数。即通过语句：

signal(SIGALRM,sig_alrm);

在sig_alrm()函数中完成了从图像的获取、格式的转换到编码最后入全局队列

的任务。YUV是一种像素的存储格式，在这种格式里色度和亮度是分开的，能够实现数据压缩的基础是色度的部分缺失并不会对图像有太大影响。

TH_QU是一个很重要的全局队列，定义如下：

```
struct thread_queue{
struct queue Q;
int32_t speed;
};
struct thread_queue * TH_QU[20];
```

每个TCP连接（此时TH_QU[i]不为空）对应着TH_QU中的一个条目，里面是一个thread_queue结构，Q项目描述的是一个欲发送的帧队列，speed描述了该连接目前的码率，服务器根据speed域来放压缩后的包。

多线程就会涉及到同步的问题，需要确保每个线程看到相同的数据视图，这是多线程编程的基本规范。在本主线程中，将压缩数据放入thread_queue结构中的Q、查看或修改TH_QU数组成员等都要涉及到多线程的同步问题。问题的出现是在多个线程用于对某个数据修改权限的时候。在POSIX规范中，可以使用pthread互斥量来保护全局数据，确保访问数据的排他性。当一个线程获取了互斥量也就意味着该线程此时拥有了该数据的访问、修改权限。除此之外的其他线程在该线程未释放锁的时候申请访问该数据将导致阻塞。在拥有锁的线程释放锁以后，阻塞中的线程才有可能对共享数据区的访问。这样种处理方式的顺利进行需要一个必需的前提条件：线程必须遵守相同的访问规范。如果应用程序中的一个或多个线程并不在访问数据前进行对互斥量的申请，也就导致了数据视图不同步的问题[36]。

在POSIX线程中，用于多线程间同步数据的数据类型为pthread_mutex_t。

queue的结构为：

```
struct queue{
AVPacketList *first_pack,*last_pack;
int size;
pthread_mutex_t lock;
pthread_cond_t cond;
int flush;}
```

AVPacketList是一个FFmpeg提供的一个构成帧队列的结构体，成员只有两个：AVPacket和AVPacketlList *next。主线程负责实时将抓取的图像压缩并放入queue里面，而服务线程负责从queue里面取得线程。所以queue这个结构就涉及到多线程并发的问题。lock成员是一个pthread_mutex_t类型的成员，任何访问first_pack和

last_pack的成员都会预先对lock加锁。然而，当服务线程获得锁的时候队列为空（即first_pack和last_pack都为NULL）时，如果不采取其他方法来控制，程序将会陷入死锁。这正是条件变量pthread_cond_t的作用。

条件变量提供了另一种线程间的同步方式，条件变量在共享数据变化时为线程提供了汇合点。初始化函数为pthread_cond_init()。

下面是两个条件变量很重要的函数：

pthread_cond_wait()和pthread_cond_signal()。

两者都是执行失败时返回错误编号，成功时返回0。pthread_cond_wait()函数的一个参数是已经被锁住的互斥量，该函数把线程放入一个等待的列表中，之后对传入的互斥量进行解锁，最重要的是，这两个操作是原子操作。这样就将受保护数据的任何变化都会及时的得到传递。pthread_cond_signal()函数会向传入的条件变量发送信号，据此唤醒等待该条件的一个线程。

2) 监听线程

监听线程的功能主要是来接受TCP连接并完成线程数据及全局数据的初始化。

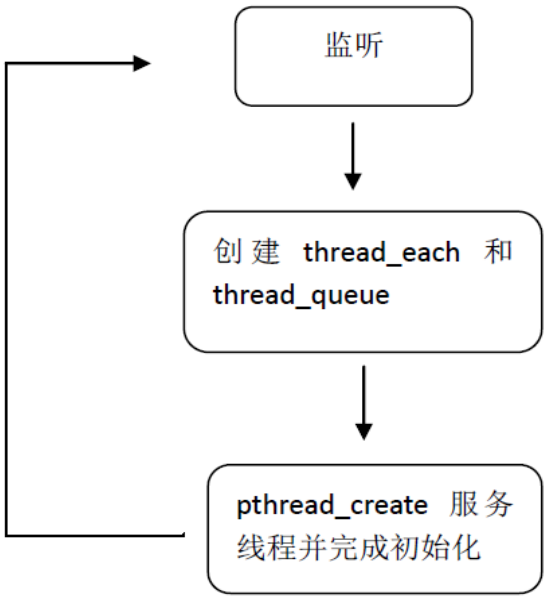

图 4.8 监听线程

Fig 4.8 Listener thread

监听时用的是阻塞套接字。处在TCP连接中的listen状态，当有连接到达时监听线程将完成对TH_QU数组的同步并且为即将创建的线程准备thread_each结构。thread_each结构定义如下：

struct thread_each{

```
int fd;
int i_;
int speed_;
uint_32_t thread_count;
};
```

每个线程都有一个thread_each结构，该结构给出了服务线程完成功能所必须的一些关键量。fd域代表了已连接套接字，服务线程在完成媒体控制信息、帧数据的发送时都是向该套接字来传递。i_记录了专属于一个连接的TH_QU数组的序号，服务线程由该标号来取得压缩后的数据包。speed_给出了一个连接目前的码率等级，初始值为负值，表示当前没有帧数据的传输，服务线程在与客户端进行控制信息的交互后才会完善此字节。thread_count字段记录了客户线程序号。

申请的thread_each结构的地址将作为线程参数传入新创建的线程：

```
each_thread=(struct thread_each *)malloc(sizeof(struct thread_each));
each_thread->fd_ = cli;
each_thread->i_ = I;
each_thread->speed_ = -1;
each_thread->thread_count = ++thread_count;
pthread_create(&tid,NULL,sock_send,(void *)each_thread);
```

其中sock_send是服务线程的地址。

3) 服务线程

服务线程涉及到一个连接的方方面面，如帧信息的发送、双方控制信息的交流等。当客户端终端连接后服务线程也将终止。为了提高服务器对连接状态的反应速度，在设置为非阻塞I/O的同时用select()函数来完成I/O的复用。

套接口在默认情况下是阻塞的，在全双工通信的方式下，用阻塞套接字会延迟服务端对连接状况的得知。非阻塞接口不会出现这种情况，对于输入操作来讲，如果条件不被满足，相应的调用将会返回一个错误，errno值为EAGAIN或EWOULDBLOCK。不返回错误的条件是至少有一个数据字节可读（TCP协议）或至少有一个数据报可读（UDP协议）。对于输出操作来讲，如果发送缓冲区没有空间，输出函数调用将立即返回错误，errno值为EAGAIN或EWOULDBLOCK。发送缓冲区中有一部分空间的情况下，输出操作返回的值将是能够复制进缓冲区的字节数，同时操作也将立即返回。讲一个套接口置为非阻塞方式的方法为：

```
int val;
val = fcntl(fd,F_GETFL,0);
fcntl(fd,F_SETFL,val | O_NONBLOCK);
```

其中fd是待处理的套接口描述字。

在用非阻塞接口的同时，使用了select()函数来完成I/O的复用。select()函数根据传入的描述符集和时间参数来决定是否阻塞和阻塞的时间。等待的事件发生或时间到期后函数才返回。select()能够处理任何的描述字。函数的原型如下：

#include <sys/select.h>

#include<sys/time.h>

int select (int max, fd_set *r_set, fd_set *w_set,fd_set *e_set,const struct timeval *timeout);

timeval是一个精确到微妙的结构，timeout参数来告知内核等待的时间，并且最终会出现三种可能：

第一：空指针。timeout是空指针的情况下,该函数将永远等待下去。除非有至少一个的描述字返回。

第二：赋予非零值：select()根据传入的timeval结构将最多等待一个固定的时间。除非有至少一个的描述字返回。

第三：赋予零值：select()函数将检查描述字后立即返回。

r_set、w_set和e_set分别代表着读套接字集、写套接字集和异常套接字集。一个套接口准备好读的条件是：

a. 接受缓冲区中的字节数目大于或等于该套接口的接收低潮标记值。该值可以使用SO_RCVLOWAT来修改该值。

b. 接受了FIN的TCP连接。

c. 是一个有完成三路握手连接的监听套接口。

d. 有一个错误待处理。

下列四个条件满足时，将准备好写：

a. 发送缓冲区空闲的字节数大于或等该套接口发送低潮标记值。

b. 该套接口的写这一半被关闭。

c. 已经建立起非阻塞connect()申请的连接。

d. 有一个错误待处理。

max参数描述了select()管理的描述字个数。

图4.9是服务线程的流程：

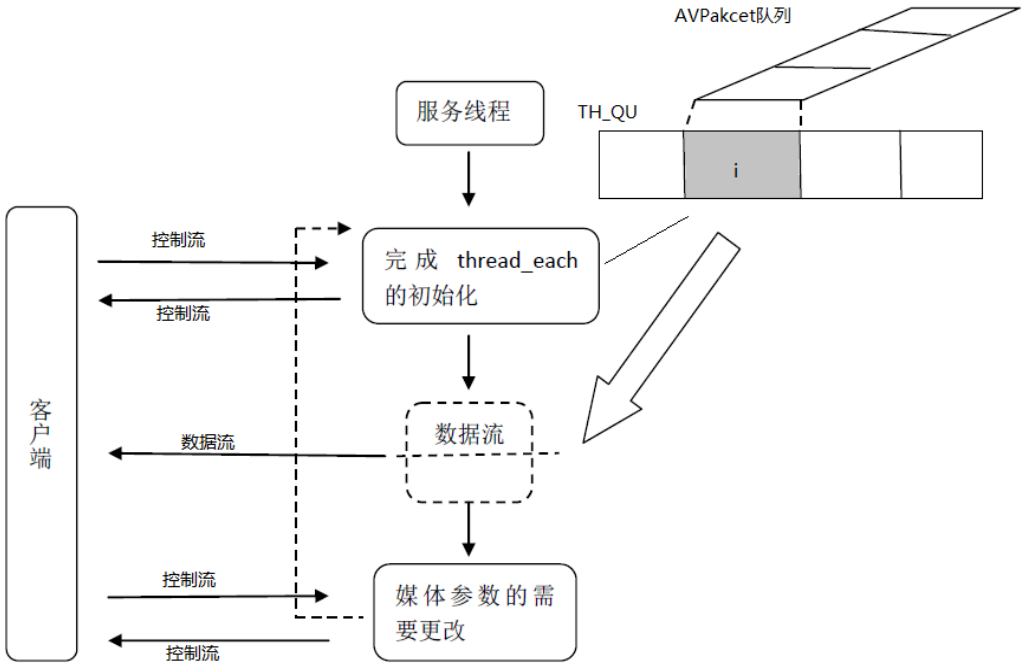

图 4.9 服务线程

Fig 4.9 Service thread

在创建服务线程的初始，thread_each结构并不包含有效的数据，只是指定了监听线程所分配的TH_QU数组中的序号，以此来寻址AVPacket队列。客户端在连接后会与服务器交换一些用以描述流媒体的信息，然后完成最终的初始化。初始化后主线程不断的向这个AVPacket队列放压缩包，服务线程不断的从队列中取出并发往客户端。从而完成帧信息的发送。期间客户端会检测丢包率的情况，当丢包率很大的情况下向服务器发送延时报文后，服务器将重新根据该连接的当前媒体信息来调整码率，完成又一次的初始化。主线程会在TH_QU中看到这样的一种变化并改变发往AVPacket队列中压缩包的压缩参数。

4.2.2 UDP 服务器设计与实现

UDP服务器不同于TCP服务器的多线程操作。UDP的所用操作均在单线程中进行。基于以下的几个原因：

① 服务器的逻辑总体来说是简单的，各个连接的在状态在服务器看来并无多大差异。

② UDP与TCP连接不同，UDP不用考虑连接的传输状况，只是到了发送数据的时刻给予发送，并没有对方确认的过程。所以各个连接间的速度差异并不会对UDP服务器产生影响。这进一步简化了逻辑。为单线程的处理提供了可能。

UDP服务器的逻辑如图4.10：

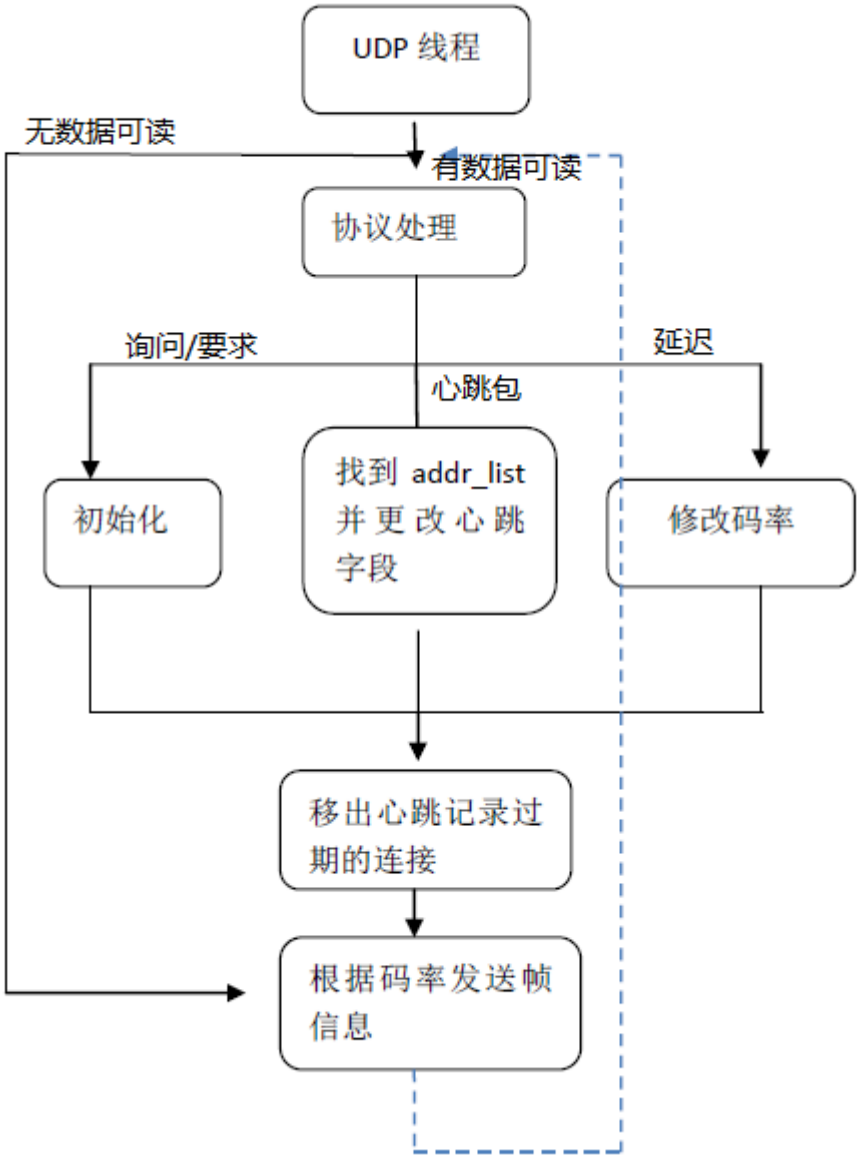

图 4.10 UDP 线程

Fig 4.10 UDP thread

当有数据可读的时候需要去分析报文的类别再做相应的处理，如进行连接的初始化，修改心跳包信息和修改对应连接的码率。然后去移除超时的连接。最后的一个过程是帧数据的发送。

在UDP服务器中有一些重要的结构来支持着线程的功能：

struct addr_list *ADDRLIST[6];

int32_t SPEED_TO_THQU[6];

其中addr_list结构定义如下：

```
struct addr_list{
struct sockaddr_in addr;
time_t    heart_beat;
socklen_t len;
struct addr_list *next;
uint32_t time；
};
```

这个结构用来描述客户端的地址和所处的状态。addr字段是套接口的地址（目前仅支持IPv4），len是套接口地址的字节大小。heart_beat记录的是该连接接收上一个心跳包的时间，每接受到一个心跳包就更新此字段。同时服务器也是根据此字段来判断客户端是否仍在线上。Next是下一个地址，用于组成队列。time字段记录的是已发送的帧序号。其中sockaddr_in的结构如下[37]：

```
struct sockaddr{
uint8_t sin_len;
sa_family_t sin_family;
in_port_t sin_port;
struct in_addr_ sin_addr;
};
```

len成员是代表着这个结构的长度。sin_family在POSIX规范里是一个uint8_t类型的成员，值有AF_INET、AF_INET6、AF_LOCAL、AF_ROUTE和AF_KEY。sin_port代表的是端口，至少是一个无符号的16位数。sin_addr记录的是32位的IPv4地址。

指针数组ADDRLIST的数组成员是一个addr_list的指针，共有6个，代表着6级码率，每个指针成员都形成了一个地址队列（图4.11）。这样的以码率来组织的方式降低了服务器的查找时间。

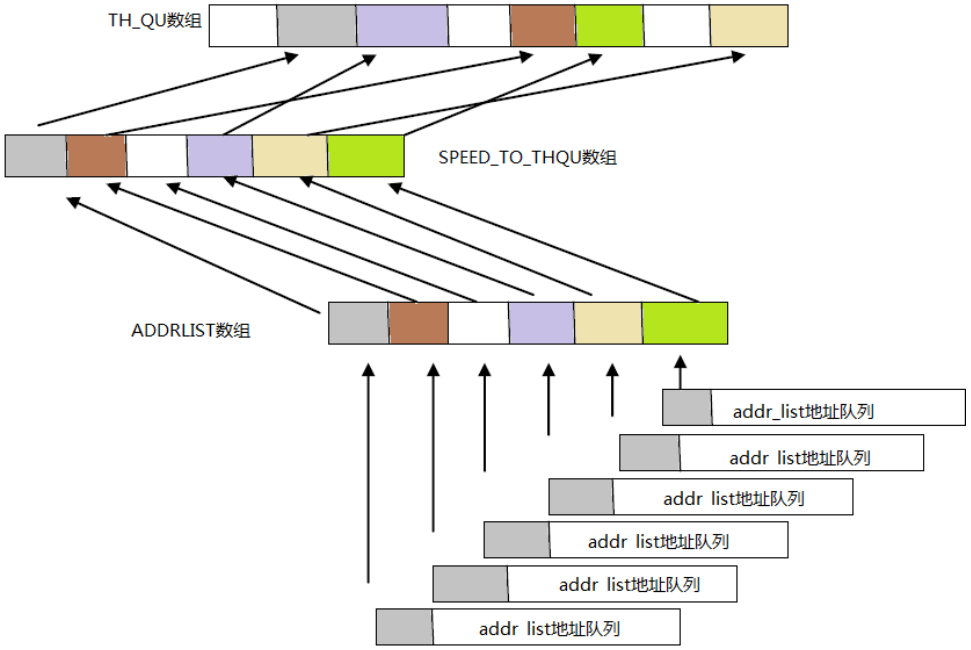

图 4.11 TH_QU 和 ADDRLIST 的关系

Fig 4.11 Relations between TH_QU and ADDRLIST

图4.11显示了TH_QU、SPEED_TO_THQU和ADDRLIST三者之间的映射关系，SPEED_TO_THQU在TH_QU和ADDRLIST之间起到了一个连接的作用，方便的根据ADDRLIST的位置来找到存放压缩包的全局数据TH_QU结构。每个ADDRLIST成员都是代表着一个客户地址队列，按照客户的实时码率来进行组织。不可否认的是，在查询方便的同时带来了同步的问题，在信息有变化的时候需要更加小心的进行线程间的同步。

心跳包超时服务器会将代表这个连接的addr_list从地址队列中去除；连接中的码率在改变时，服务器也会将这个连接的addr_list结构从现在的队列中删除，并重新放置到新码率的队列中去；新的连接加入时，服务器会创建新的addr_list结构并放置到合适的码率代表的队列中。

在有新连接到达时，UDP服务器会根据发来的“要求/应答”报文确定该客户应有的码率，在将该客户地址放置在addr_list队列之前需要做如下的同步工作：

① UDP服务器检查该码率下是否存在客户。如果没有存在客户则执行步骤2；如果存在客户执行步骤4。

② 锁住保护全局结构TH_QU的互斥量，寻找TH_QU中的空闲值。找到后，申请一个thread_queue结构，并将地址写到该空闲处。

③ 初始化该thread_queue结构中的帧队列，即Q域。然后将在该码率下的连接数执行加1操作，记录码率下连接数结构有两个：SPEED和SPEED2，前者是服务

器中总的连接数，后者仅指UDP连接中的记录。这里更新的是前者。

④ 将UDP服务器中该码率下的连接数增1。

⑤ 申请addr_list结构，用以描述这个连接。根据发来的“要求/应答”报文更新地址、时间等字域。最后根据码率来置入addr_list队列。

放置到addr_list队列后就完成了新客户的初始化。多处都进行了同步，这里稍显麻烦的操作是为了在置入帧数据和发送帧数据时的便利。

从服务器收到“延迟”报文到服务器将发送“延迟”报文的地址置于新队列中，这中间也有一系列的检测与同步的过程。

① 首先是码率的检测，如果该连接目前正处于服务器所支持的六级码率中最低的级别里，该“延迟”报文就不被处理，因为这已经是服务器能降低的最大限度。到这里，“延迟”报文的处理已然结束。

② 如果该连接的码率仍有下降的空间就将SPEED2中对应的连接数减1，减1后连接数如果为0，表示在UDP服务器中没有该码率下的客户，需要同步全局结构SPEED、SPEED_TO_THQU和TH_QU。部分代码如下：

```
pthread_mutex_lock( &lock_num );
struct thread_queue *pt;
SPEED[i]--;
if( SPEED[ i ] == 0 )
    flush_avcodec( i );
j = SPEED_TO_THQU[ i ];
queue_destroy( & ( TH_QU[ j ]->Q ) );
pt = TH_QU[ j ];
free(pt);
TH_QU[j] = NULL;
pthread_mutex_unlock( &lock_num );
```

③ 将更改后的码率对应连接数加1，即修改SPEED2结构。如果这是该码率下唯一的连接，就依照处理添加新连接的方式来进行；不是唯一连接的情况下，就直接将该地址添加进addr_list队列。

服务器是在一个循环中执行全部的逻辑操作，用的是UDP协议来进行数据的发送，即使是在帧数据的发送过程中也不会有阻塞的情况。这就为这种轮询的服务器设计增加了反应速度。每进行这样的一次循环都会遍历整个地址结构，查看是否该地址的心跳包是否过期，只要过期便将其移除。移除的操作基本与接受新连接的处理方式相反。

最后一个过程是根据码率将获取的压缩包进行分拆，发送给同一码率队列的

客户地址。

4.3 本章小结

本章完成了嵌入式服务器的编写。简要描述了Linux I/O模型和POSIX线程，并探讨了Per-connection-Per-thread模式在嵌入式领域的可行性。根据所采用运输层协议的不同而采用不同的设计策略，保证了客户连接在服务器层面上的相互独立。

5 客户端 GUI 软件的设计与实现

5.1 Qt 介绍

5.1.1 Qt 简述

Qt是一个跨平台的应用程序和用户界面开发框架，编程语言是QML或C++。不仅能用于GUI软件的开发，在服务器和控制台工具的开发中也很常见到其踪影。使用Qt库，我们可以一次性编写应用程序并在多平台中快速部署，需要做的唯一工作就是依照平台再编译一次，由于所支持平台的众多，Qt将“Code Less,Create More,Deploy Anywhere”的理念发挥到极致。

Qt库的开发最早可以追溯到1991年，并在三年后成立了Trolltech公司。2008年6月Trolltech公司被NOKIA公司收购，Qt项目也就划归到了NOKIA的管理之下，用以增强NOKIA的跨平台能力，也就是在这期间，NOKIA宣布Qt框架的源代码面向公众开放，非NOKIA里的开发人员也可以为该项目贡献代码、翻译等内容，让公众共同的来引导Qt未来的发展。2012年8月，Digia公司完成了对NOKIA的Qt业务的全面收购。

Qt所能支持的平台是众多的，桌面环境支持如下平台：

① Windows：用于Microsoft Windows XP、Vista、7、8。可以用MinGW编译也可以用Visual Studio集成。

② Linux/X11：支持X Window System。也就意味着支持Solaris、Linux、BSD、HP-UX等平台。

③ Mac：用于Apple Mac OS X。基于Cocoa框架。

嵌入式领域：

① 嵌入式Linux。

② 嵌入式Windows：Windows CE 6 和 Windows Embedded Compact 7。

③ 实时操作系统：QNX、VxWorks。

移动领域：

① Android

② IOS

③ BlackBerry 10

④ Sailfish OS

Qt是一个开放源代码的框架。目前有三种授权方式。授权方式的区别并不体现在功能的方面而是在于使用方式的不同。

① Qt商业版：这种授权方式比较适合于商业软件的开发。在能获得技术支持

服务的同时，还可以修改Qt本身的源代码，并不要求使用者共享应用程序的代码和修改过的库代码。商业版需要一定的授权费。

② GNU LGPL：目前的版本支持GNU LGPL v2.1授权方式。这种授权方式下，Qt库可以被专属软件作为类库引用，也可以进行销售，但是不能修改Qt本身的源代码。

③ GNU GPL：目前在GNU GPL v.3.0的授权方式下，使用Qt库的应用软件必须公开源代码。

5.1.2 Qt 的信号和槽

Qt不但是一个完善的C++图形库，也可以用在服务器、控制台工具上。真正意义上形成了一个跨平台软件开发的框架（表5.1）。集成的模块有数据库、XML、Webkit、OpenGL、网络库等。

表 5.1 Qt 的主要模块

Table 5.1 Main modules of Qt

模块	功能
QtCore	是 Qt 中唯一一个必须包含的模块，提供一些最基础的非图形类库。
QtGui	GUI 的核心模块，提供了有关图形应用的类库（在 Qt5 与 Qt4 之间有较大差异）。
QtNetwork	提供了开发网络应用的能力。全面支持 TCP、UDP、HTTP 和 SSL。
QtOpenGL	在 Qt4 上提供了 OpenGL 的支持，在 Qt5 里划到了 QtGui 模块。
QtWebkit	集成 WebKit，提供了 HTML 浏览器引擎。
QtSQL	为 SQL 提供了处理的类库。
QtMultimedia	与音频、视频有关的类库。

QtCore的信号和槽机制提供了对象间的通信功能[38]。信号和槽机制是Qt编程的基础。它可以让互不了解的对象进行通信，类可以各自实现自己的槽也可以各自发射信号。“发射信号”这一行为表示一个用户动作已经发生了或者是表明类的某个状态的改变，通过将信号与槽连接，在动作发生或状态改变的同时槽可以得到自动的执行。槽和普通的C++的成员函数大致相同，可以是私有的、受保护的和公有的。connect()函数负责建立信号与槽间的对应关系[39]，connect函数是在QObject类中定义的，函数的原型为：

bool QObject::connect (const QObject *sender , const char * signal,const QObject *receiver , const char * method , Qt::Connection Type = Qt::AutoConnection)[static]

sender和receiver都是QObject对象的指针，signal和method是描述信号和槽函数的字符串，但在使用时必须将SIGNAL()和SLOT()宏将两者包裹起来。

5.1.3 Qt 的多线程支持

早期的Qt应用程序都是单线程的，这意味着响应用户的动作和该动作的实际执行都是在一个执行线程中执行。但是当涉及到处理一个比较耗时的工作时，这种方式就显得很不方便，因为在执行耗时的工作的时候用户便不再响应。

多线程在响应和处理之间有着很好的平衡。这时候可以将用户的图形界面运行于一个线程中，而将另外的事件处理过程放在另外的一个线程。所以，即使在响应一个比较耗时的工作的时候，GUI程序也会对此时用户的动作进行比较及时的响应，从而消除了假死的可能。

多线程是一个很大的课题，其中的一个最基本的问题就使得用户不得不面对：多线程的同步问题。Qt提供了一系列的类来简化多线程间的同步问题（QMutex、QWaitCondition、QReadWriteLock等）。

Qt中实现多线程的方式是子类化QThread并重新实现它的run()函数即可[40]。默认情况下run()函数通过执行exec()函数的功能来开启这个线程的事件循环。

QThread类通过发送信号started()、finished()和terminated()来通知本线程的状态，或者是也可以执行查询函数isFinished()和isRunning()来询问线程目前的状态。通过调用函数exit()、quit()来终止线程，在极端的情况下，执行terminate()函数可以强制终止一个线程的运行，然而，这是危险的，因为没有留给线程自身清空内存的时间和机会，极易造成内存泄露。应用程序可以调用wait()函数来阻塞执行线程的运行，直到另外线程的终止。

QMutex类提供了线程间的同步方法。它的功能在于确保同一时间只能有一个线程来访问一个一段代码或一些变量。当调用成员函数lock()时，其他线程想再次lock()会陷入阻塞中，直到调用线程执行unlock()。一个非阻塞的锁操作是调用trylock()。同时，Qt提供了一个QMutex的优化类QMutexLocker。QMutexLocker类的析构函数将负责本互斥量的解锁。

QMutex带来了一些性能瓶颈问题，因为每次只能有一个线程进行访问，“读”和“写”是平等对待的。而QReadWriteLock允许多个读线程来同时访问临界数据，但只能允许一个写线程来修改数据。这样的一个粒度有助于提高整体的性能。

QWaitCondition是一种条件变量，类似于POSIX下的pthread_cond_signal()和pthread_cond_wait()两个函数。能够允许线程间的信息通知，提供比QMutex更为精确、更为有效的控制。一个或多个线程一直阻塞直到其他的线程用QWaitCondition调用wakeone()或wakeall()。wakeone()和wakeall()的不同在唤醒线程的数量上，wakeone()随机唤醒一个而wakeall()则唤醒全部。Qt与POSIX线程控制有很多相似的地方（表5.2）。

表 5.2 Qt 线程类型和 POSIX 线程类型

Table 5.2 Qt thread module and POSIX thread module

Qt Thread	POSIX Thread
QMutex	pthread_mutex_t
QReadWriteLock	pthread_rwlock_t
QWaitCondition	pthread_cond_t

5.2 客户端中类的构建

5.2.1 类的构建层次

客户端用Qt来编写，以增加其跨平台的功能。目标是“形成一个多连接、高可靠、界面友好的跨平台GUI软件”。初始界面及几个显示类如图5.1：

图 5.1 初始界面

Fig 5.1 Initial interface

客户端中主要类的构造层次如图5.2：

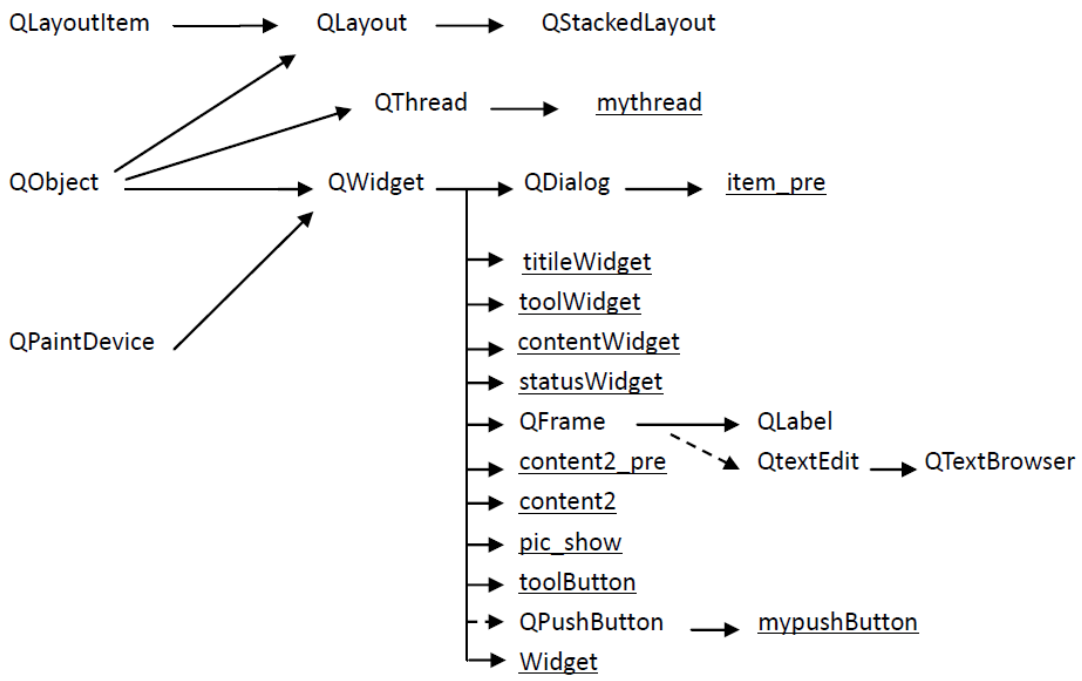

图 5.2 主要类的构造层次

Fig 5.2 Structture of main classes hierarchy

其中虚线箭头表示两类之间并不是直接基类与直接子类的关系，下划线的类表示是在本客户端中实现的类，非下划线类表示该类是出现在Qt Framework中的类。

QWidget在客户端的类树中扮演着极其重要的角色。其实，不仅仅是本客户端，在整个Qt Framework中都是一个很重要的类。QWidget是所有用户界面对象的基础类，是用户界面的基本组成部分。它来获取键盘、鼠标等事件，并在屏幕上绘制自己的图像。每个QWidget类都是矩形的，在Z轴的层次上予以显示。一个QWidget类的显示效果是由父类和排在Z轴高层次上的类来决定的。

每个QWidget类的构造函数接受一或两个标准的参数：

① QWidget *parent ：它是新QWidget类的父类指针。如果parent为0的情况下，这个新QWidget将是一个窗口（没有父类）。如果parent != 0,那么新的QWidget类是parent的组成部分。

② Qt::WindowFlags：该参数描述的是窗口标志。通常情况下是默认值，因为能满足基本的需求。

QObject类是Qt框架中的基础类，同时也是Qt框架中非常重要的一个类。Qt框架中用于对象间通信的信号和槽功能就是实现于此。有了QObject我们才可以用connet()来连接信号和槽；才可以用disconnect()来移除信号和槽的连接。QPaintDevice是可绘对象的基础类，一个绘制设备是二维空间的一种抽象，可以使用QPainter来对这个二维空间进行绘制。坐标的默认原点在左上角，X轴向右方向

增长，Y轴向左方向增长，单位是像素。QWidget图形类正是继承于该类。

5.2.2 类间的控制流

类间的通信是依托Qt Framework间的信号-槽机制来实现的，信号-槽机制有效的确保了控制流的传递。

① 启动/关闭控制流（图5.3）

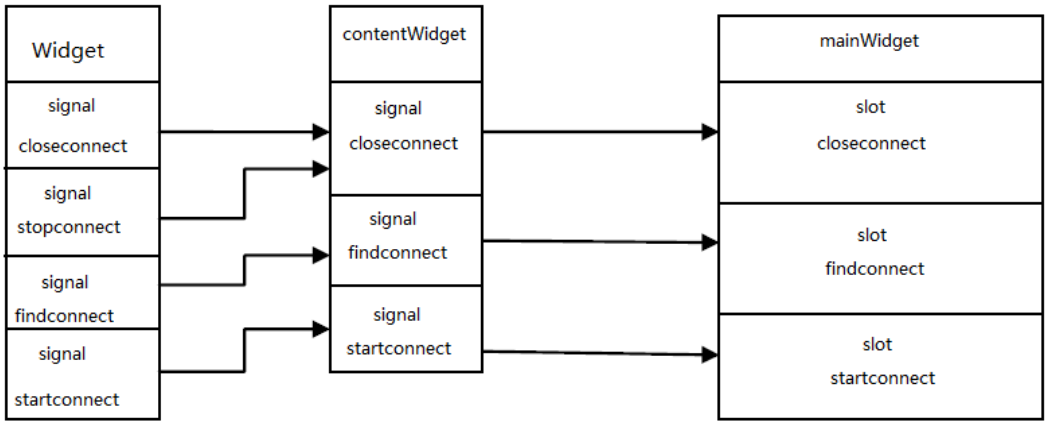

图 5.3 启动/关闭控制流

Fig 5.3 ON/OFF control flow

Widget是一个设备类，每个设备都对应一个Widget类，上面有进行控制的启动、停止、查找和删除功能，分别对应着startconnect()、stopconnect()、findconnect()和closeconnect()信号。因此这是一个直接与用户交互的界面类。信号经过contentWidget的传输到达mainWidget类进行处理。mainWidget有三个专属槽进行处理：

startconnect slot:在目前已有的连接查看是否能找到当前正在申请的连接（根据IP地址和端口号），若找到则执行findconnect槽。若找到，则创建一个conntent2类，再执行findconnect槽。

stopconnect slot：查看该连接的当前连接数（即同一个套接字地址Widget类的个数），若个数大于1,则将个数减1。若个数为1，则删除content2类。并更新QStackedLayout。

findconnect slot：该槽的作用是完成从设备界面到监控主界面的转换。将Widget对应的content2类直接显示出，需要的操作是更新QStackedLayout类。

② 控制流与QStackedLayout类

客户端的主显示类是QStackedLayout类，相应toolWidget中的点击和Widget中的点击。每个点击操作之后都会有QStackedLayout类的更新，因此该类显得很重要。

QStackedLayout类提供了一个类对象的队列，同一时间只能有一个类是可见的。图5.4显示了在本例中的应用：

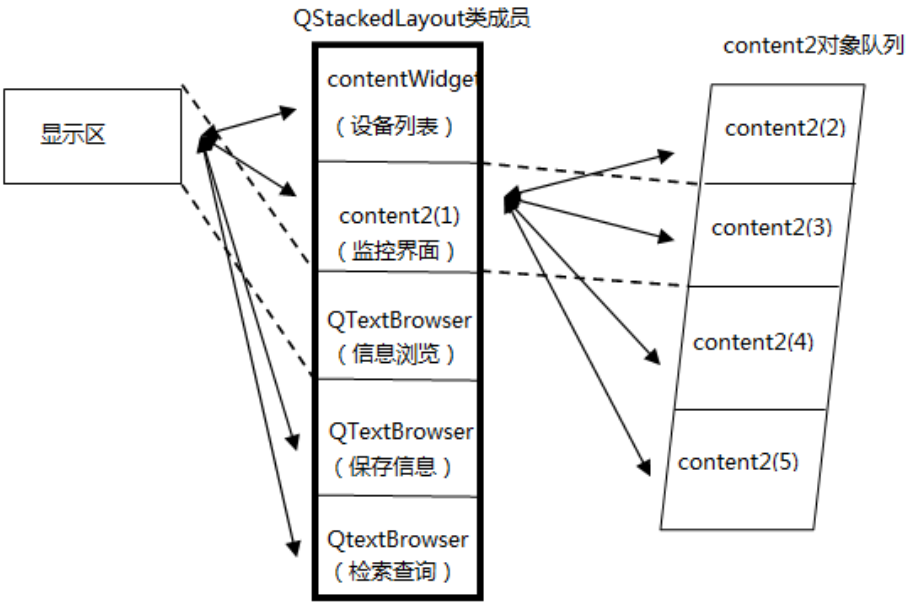

图 5.4 QStackedLayout 的功能

Fig 5.4 Function of QStackedLayout

显示区的决定权在toolWidget类中，“监控界面”的决定权在Widget类中的findconnect信号中。

③ 截图/录像控制流

截图/录像控制流如图5.5：

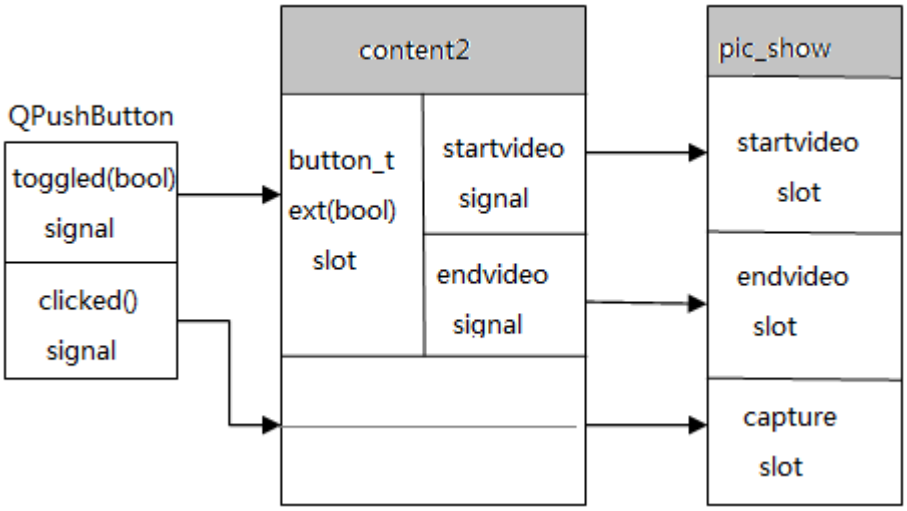

图 5.5 截图/录像控制流

Fig 5.5 Screenshot/Recording control flow

两个按钮（QPushButton），一个用于决定开始和停止录像，一个决定是否截图。通过content2的处理与传输最终连接到pic_show类中。

pic_show类是一个重要的显示类，是content2类的重要组成部分。多线程的控制、录像、截图等功能的处理都是由pic_show类来实现。它与线程类mythread共同组成了整个客户端图像显示的基础。

用FFmpeg工具输出媒体文件的一般步骤为：

1) 用 av_register_all()来注册所有的编解码的库和文件格式。

2) 申请 AVFormatContext 结构，用以描述媒体文件上下文。用 avformat_alloc_context()或是 avformat_alloc_output_context2()，其中后者的原型为：

int avformat_alloc_output_context2(AVFourmatContext **ctx,AVOutFormat * oformat,const char * format_name,const char * filename);

若成功则返回大于 0 的数。

3) 申请视频流结构 AVStream 并初始化解码上下文 AVCodecContext。用到的函数主要有：

根据编码 ID 来找编码器：

AVCodec * avcodec_find_encoder(enum AVCodecID id);

根据媒体文件上下文和编码器来申请视频流结构：

AVStream *avformat_new stream(AVFormatContext *s,const AVCodec * c);

初始化 AVSstream::codec:

需要初始化的有 codec_id、bit_rate、width、height、time_base、gop_size、pix_fmt 和 max_b_frames。

4) 打开所需的编码器：

int avcodec_open2(AVCodecContext *,AVCodec *);

5) 将视频流的头信息写入输出的多媒体文件：

int avformat_write_header(AVFormatContext *s,AVDictionary ** options);

6) 调整时间基并写入媒体文件：

int64_t av_rescale_q(int64_t a , AVRational b,AVRational c);

av_rescale_q()用来将时间戳从一个时基调整到另外一个时基。它的基本动作是执行 a*b/c，但是这个函数是必需的，直接计算会导致溢出的情况发生。

写入媒体文件用到的函数是：

int av_interleaved_write_frame(AVFormatContext *, AVPacket);

av_interleaved_write_frame()函数将负责一个压缩包写到一个媒体文件。在一些情况下会暂存帧数据以便于按照解码时间戳来重新排序。第一个参数是媒体文件的格式上下文，用以对媒体文件做最基本的抽象。第二个传入的参数是要写入的压缩后的帧信息。

7) 最后一步是写入文件尾：

int av_write_trailer(AVFormatContext *);

写媒体流尾到一个输出文件，并释放媒体文件的私有数据。只能在一次成功的调用 avformat_write_header()函数后才能使用。

5.2.3 类间的辅助信息流

辅助信息主要用于向用户提供过往的操作信息，形成日志文件，以便于用户查询、排错。同时也让用户及时的了解目前客户端的所处于的状态。

在辅助信息的处理中主要涉及到一个类的两个对象，一个用于工具选项中的“信息浏览”项目、另一个用于“录制信息”。用到了QTextBrowser类，QTextBrowser类继承自QTextEdit类，是一个用于显示文本的视图类。这里只是简单的使用append成员函数来将文本附加到最后。并在初始化的时候作如下处理：

```
QTextBrowser * browser    = new QTextBrowser;
browser->setStyleSheet(“QTextBrowser{background-color:black}”);
browser->setTextColor(Qt::white);
```

这里使用了Qt的Qt的样式表将背景设为黑色，并通过QWidget的成员函数setTextColor将文字设为白色，这样的处理让“辅助信息”的这一功能更加直观。

辅助信息流结构如图5.6所示：

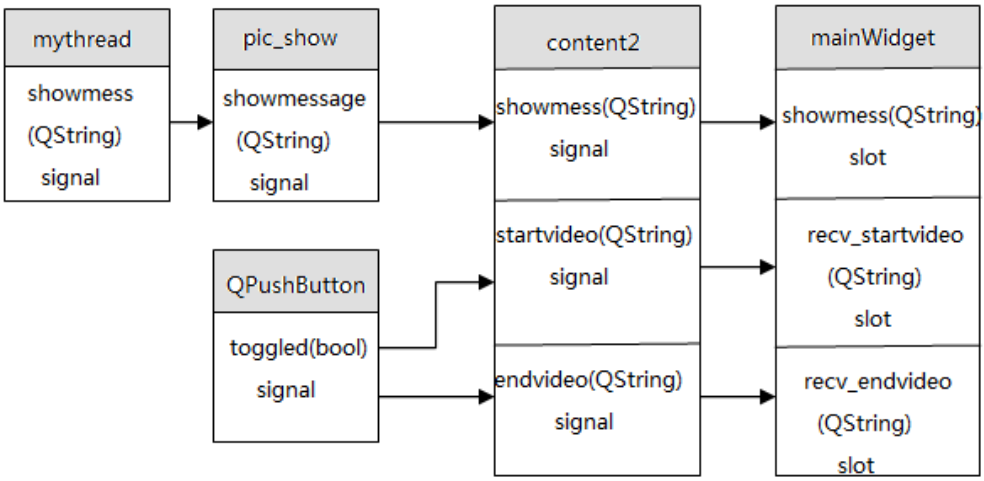

图 5.6 辅助信息流

Fig 5.6 Auxiliary information flow

mainWidget中的showmess()槽是“信息浏览”功能实现函数，showmess()槽直接调用QTextBrowser类来实现相应消息显示，在QTextBrowser中直接调用append()成员函数具体来实现。信号的发出主要由mythread线程类和pic_show图像处理类发出，前者的消息主要来自心跳信息、服务器返回的媒体描述信息以及许多异常信息，后者的消息主要来自视频录制、截图等方面的控制和异常信息。

mianWidget中有recv_startvideo()和recv_endvideo()两个槽来负责处理录制信息的显示，并在每次处理的时候都要统计正在录制中的设备数。消息的起点是content2显示类中的连个按钮，初始时分别标志为“开始录制”和“屏幕截图”。

5.3 模块与模块的功能

解码和显示模块主要涉及如何从网络中获取数据、如何拼凑成帧以及如何显示。解码模块涉及的是客户端获取完整帧信息后的解码，码率控制模块负责在网络出现拥塞后如何控制数据的传输。传输层TCP和UDP协议的不同导致了应用层协议的同，进而导致了数据接收的处理方式不同。通过不同的策略获得了完整的帧信息后用FFmpeg库进行解码并予以显示。

5.3.1 接收模块

客户端分为两个线程，一个用于从网络中获取数据，另一个是客户端的GUI框架。前者的实现是靠子类化QThread而形成的mythread类，他作为pic_show类的成员而出现，mythread与pic_show一起完成了整个的接收、解码、显示的工作。

UDP是无连接的协议，不能保证数据的可靠传输，而且是面向报文的。这些都影响着UDP数据的处理。图5.7显示的是UDP连接下的数据报文格式：

图 5.7 数据报文格式

Fig 5.7 Format of data protocol

FRAME域给出的是帧的序号，服务器每发送一帧便将此字节加1。由于一个帧的数据在一个UDP数据报内放不下，这时就需要IN-FRA域，该字段表示的是帧内序号，用以表示该数据报的信息是处于某帧的第几个片段。服务器在发送的时候帧内序号是按照递减的规则来发送，比如，某帧需要5个UDP数据报才能承载，则服务器在发送的时候将第一个报文的IN-FRA字段置为5，第二个报文的IN-FRA置为4，相应的剩下的为3，2和1。这样处理的好处是能让客户端知道在当前正在接受的帧序号下还有几个报文的数据才能完整的拼出一个独立的帧，如果第一个也就是最高IN-FRA报文丢失的话会影响此种判断，但是，在UDP协议下，此种情况不予处理。

UDP报文在传输的过程中会发送失序和丢失，在尽量保证实时的情况下如何处理这方面的问题也就变得尤为重要。在客户端，接收缓冲区实际上是一个以关键字进行排序的队列，具体的说，是以关键字递减的方式来组织数据，如图5.8：

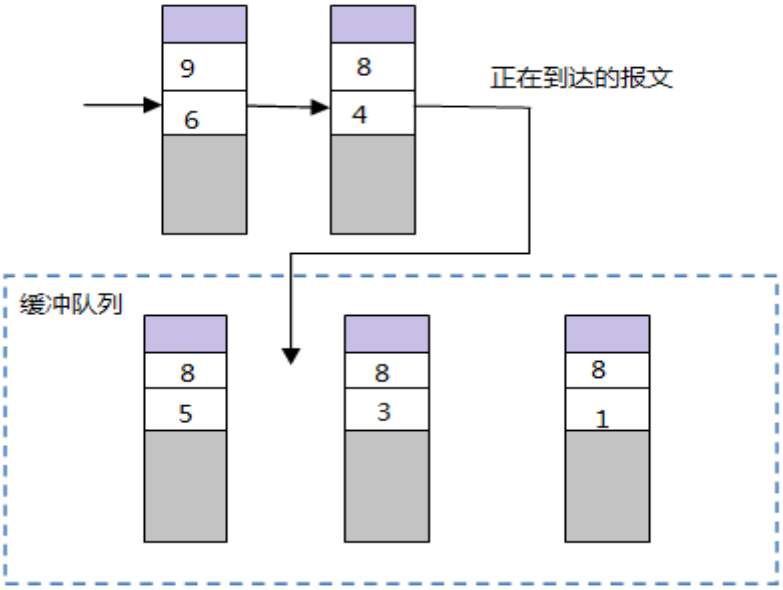

图 5.8 缓冲队列

Fig 5.8 Buffer queue

相同帧序号的报文到达时将按照IN-FRA字域组织成一个队列，以此来消除UDP报文在网络中的失序情况，如图5.8，当帧序号为8帧内序号为4的报文到达时将插入IN-FRA为5和IN-FRA为3之间。并在满足下列两个条件时会把缓冲队列中的数据组装成一个完整的帧数据：第一是缓冲队列中IN-FRA域是完整的，不存在丢失的情况；第二个是新的帧序号报文的到达。IN-FRA域完整与否的检查是不完善的，因为在高IN-FRA报文丢失的情况下客户端同样会做出完整报文的决定，这里的策略是对于此种报文的丢失将不予处理。在第二种情况中，即新帧序号报文到达时接收缓冲区还有上个序号的报文，这便表明有报文的丢失或延迟，为保证实时性，客户端会将不完整的队列打包。

mythread线程在接受数据时用的是非阻塞接口，函数为recvfrom()：

#include<sys/socket.h>

ssize_t recvform(int fd,void *buf,size_t n,int flag,struct sockaddr *from socklen_t addrlen);

当函数执行成功时返回接收到的字节数。fd是描述符，buf和n分别为接收缓冲区和缓冲区的字节数大小。from发送数据报的地址，告诉我们是谁发送的数据报，addrlen是地址的大小。flag参数将会对接收的策略产生影响，比较重要的几个值是MSG_PEEK、MSG_WAITALL和MSG_DONTWAIT。MSG_PEEK允许应用程序查看可读取的数据但是并不丢弃被接收走的数据。MSG_WAITALL告诉内核必须接收到recvfrom()函数所要求的全部数据字节数后才能返回，这无疑给用户增加了极大的便利性。MSG_DONTWAIT会让此次的接收操作立即返回，实现的手段是在执行I/O操作前将描述符指定为非阻塞，在接受操作完成后关闭非阻塞的标志。

mythread线程在发送数据的时候使用了sendto函数：

ssize_t sendto(int fd,const void * buf,size_t n,int fag,const struct sockaddr * to,socklen_t len)[41];

to参数指向的是一个数据报接收者的地址，结构大小由len参数来指定。在UDP模式下，客户与服务器的交互模型如图5.9：

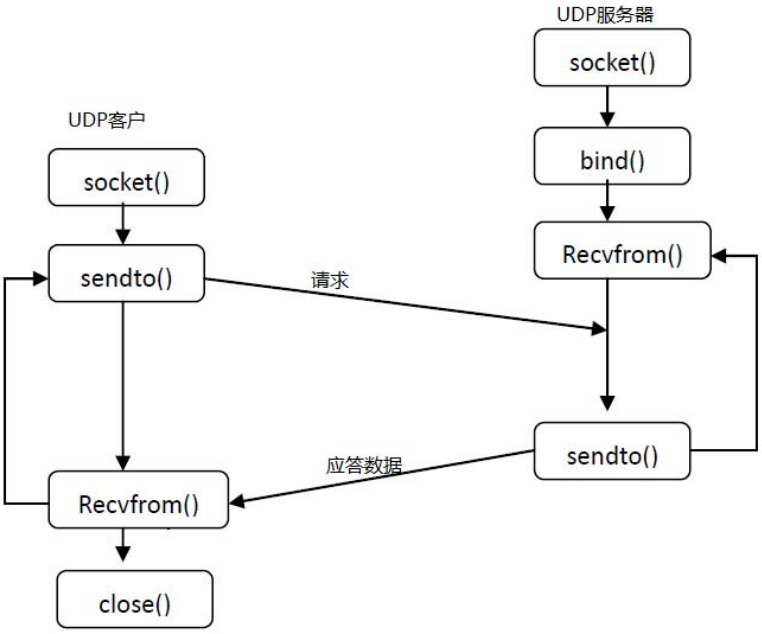

图 5.9 UDP 客户与服务器的交互

Fig 5.9 Socket functions for UDP server/client

客户端在TCP协议下和UDP协议下的接收方式和处理策略有很大的不同。在TCP协议下服务器不用将数据进行分割，相应的，客户端便没有将数据进行拼接的任务。TCP提供了可靠传输的实现，在图4.8中可以看出在制定TCP传输下的数据帧协议时并没有“帧序号”字段和“帧内序号”字段。TCP服务器在获得一帧的压缩包后直接将数据写在TCP数据字段中，即在T、BR和SIZE字段之后，SIZE字段给出的就是该完整帧的字节大小。

TCP协议下的客户端使用的是阻塞的套接口，因为客户端连接线程的逻辑相对简单，非阻塞套接口反倒是增加了逻辑和代码的复杂性。对于数据字段，客户端的读函数直到读够SIZE个字节才返回；而控制字段有固定的字节数，客户端仅需要读够默认的字节数便可。由此，TCP连接下客户端的接收线程就是一个不断的读取数据、分析类型和处理数据的过程。

5.3.2 解码与码率控制模块

mythread线程在根据策略得出缓存中已有完整帧的数据时，会将数据拼接（TCP协议无此过程），然后对其解码，解码完成后会将“完成”的这一消息告知pic_show类，如图5.10：

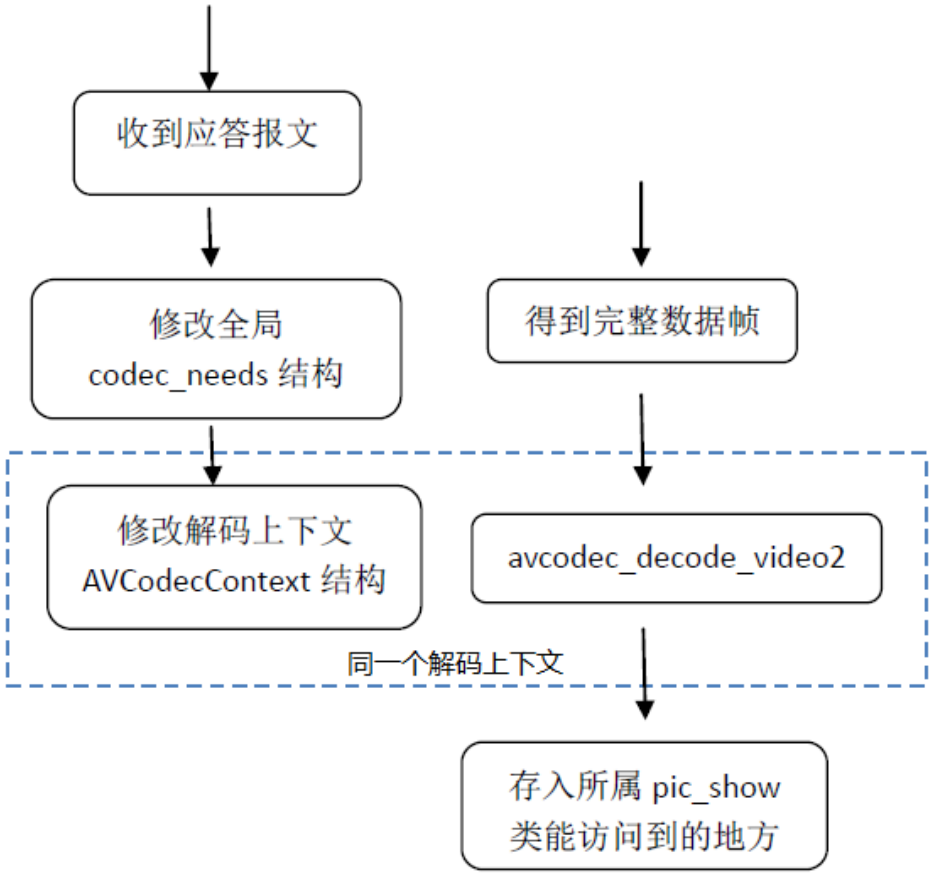

图 5.10 解码流程

Fig 5.10 Decoding process

解码需要在解码之前初始化解码上下文。为了快速的对帧数据进行解码，对上下文的初始化并不等到解码的时候才进行，而是在客户端与服务器间控制协议交流后完成。在客户端有个全局的结构用于记录目前连接的媒体信息codec_need:

```
struct codec_need{
int width;
int height;
int fps;
int gopsize;
int max_b_frames;
uint32_t bit_rate;
};
```

width字段是记录的是视频的宽度，height记录的是视频的高度，fps记录的是视频的帧率，gopsize记录了视频的图像组长度，max_b_frames记录了视频中两个非B帧之间允许插入的B帧的个数,bit_rate代表了当前的码率。在服务器的每次应答

后都更新该结构。AVCodecContext结构的成员与此类似，除了这些成员的初始化外还要初始化的成员是time_base和pix_fmt，pix_fmt描述的是像素格式，类型是enum AVPixelFormat。time_base描述的是时间的刻度，它是AVRational结构：

typedef struct AVRational{

int num;

int den;

};

AVRational结构描述的是一个分数，num域为分子；den域为分母。在AVCodecContext中和AVStream中都有time_base成员，但两者的含义是不同的，初始化时AVCodecContext中time_base成员根据帧率来设定，在这里，num为1，den为codec_needs结构中的fps成员。AVStream中time_base成员一般依据采样率来初始化，典型值如num为1，den为90000。因此在存入容器或转换格式的时候都要进行时间刻度的转换。

解码所用的主函数为avcodec_decode_video2():

int avcodec_decode_video2(AVCodecContext *ctx,AVFrame *frame,int *got,const AVPacket *pkt);

ctx描述的是解码上下文，frame是用来存放解码后帧数据的容器，如果没有完整的帧数据被解压缩出来则将got所指向的整形数据置为零，反之，置为非零值。pkt用来存放原始的未被解码的帧数据。并不是每次执行函数都会输出一个完整的帧数据，也不是每次输出的帧数据就是当时传入的pkt进行解码后产生的。由于B帧的存在，解码时间戳与显示时间戳是不同的，比如当得到一个视频文件，显示时的顺序为IBBP，B帧是一个双向预测帧，因此在显示B帧的时候要知道P帧的内容，也就意味着在文件中存储的顺序为IPBB。FFmpeg会暂时存储帧数据以保证被avcodec_decode_video()输出的帧的PTS与刚传入的帧的DTS相同，也就是播放的次序。

在UDP中少量的真丢失是不影响视频质量的，只有大量的丢失帧数据才影响视频的质量。大量丢失帧数据的情况有时候是难以避免的，尤其在网络环境恶劣的情况下，这就需要一种策略来尽量避免这种丢失情况。在相同的帧数量下，通过适当降低数据量的发送可以抵消部分网络延迟，有助于提高视频的流畅度。一般由客户端用户来负责决定是否需要降低数据量，用户拥有对视频质量评价的最终决定权。客户端根据策略得出需要降低数据量时，就向服务器发送“延迟”报文，以此来让服务器降低数据量的发送。服务器会降低视频的码率并向客户端发送“应答”报文来反应媒体数据的变化，然后服务器重置编码上下文，重新压缩帧数据。

客户端在对视频质量做出评价时可以采用两种方法：一是由用户来评价，用户拥有评价视频质量的最终决定权；二是由丢包率来决定，这也是目前大多数流媒体客户端采用的策略。本客户端采用第二种方法，通过丢包率这一硬指标决定是否发送“延迟”报文。记录当前视频评价信息的结构为check_time:

```
struct check_time{
time_t old_time;
int beats;
int bit_rates;
};
```

old_time成员记录当前检测周期的起始时间，与当前的时间差距最大为10秒。beats成员记录了当前检测周期内共收到的帧数，bit_rate是这个周期内的码率值。

在检测的时候，同一个测试周期内只能存在同一个码率的数据，因此先比较当前的码率与check_time结构中的码率是否相等，如果不等则将check_time结构中的码率改为当前客户端连接中的码率，将beats置为1，将old_time置为系统当前的时间。如果码率相同且系统当前时间与old_time的间隔小于10，只需要将beats字段加1，否则评价丢包率，在可接受的范围内，需将old_time置为系统目前的时间，将beats置1；超出了可接受的范围就发送“延迟”报文。

5.3.3 显示模块

与显示有关的三个类是mythread、content2和pic_show类，如图5.11：

图 5.11 pic_show 和 content2 类

Fig 5.11 pic_show class and content2 class

pic_show类和两个QPushButton对象都是content2的成员。content2类对象也是QStackedLayout结构的组成部分，直接响应用户对于视频显示区的请求，直接对用户的要求负责。

mythread线程完成解码后将帧数据存入公有成员中，这部分数据有线程互斥量的保护，然后通知pic_show类，通知的方式是通过置位共享数据，pic_show类周期性检查这一状态变量，通过QTimer来实现定时：

```
QTimer * time2    = new Qtimer(this);
connect(timer2,SIGNAL(timeout()),this,SLOT(check_run()));
timer2->start(30);
```

QTimer类是Qt框架中一个非常重要的定时器类，为用户的定时行为提供了高层次且非常易用的编程接口。只要将它的的timeout()信号绑定到合适的槽中，随后执行start()成员函数，随后的时间里，QTimer将根据用户所设的时间周期性的发射timeout()信号。这里将timeout()信号与pic_show类的check_run()函数连接，周期性查看是否有新的可被读取的帧数据。

在pic_show类确定有可读的帧信息后会将帧数据拷贝到自身类中，保存到QImage对象里。这里有两个很重要的类：QImage和QPixmap。QImage类是专门为输入、输出而优化过的图像类，提供了直接的像素级别的访问和操纵。QPixmap类是专门为了将图像输出到屏幕上而设计和优化的。pic_show类在得到QImage后会触发一个“重新绘制”事件，并将该QImage对象转换为QPixmap对象存储，接着完成最后的显示工作。

5.4 客户端测试

客户端程序用了跨平台的开发框架Qt来编写，测试用的系统是ubuntu 12.04 LTS，内核版本为Linux 3.2.0-59-generic-pae。运行效果如图5.12、5.13和5.14，当打开软件时默认是只有“设备列表”功能项而没有设备列表条目的，通过选择“添加设备”、并输入IP地址和端口号才能形成设备条目，如图5.12，图5.13添加了三个设备。

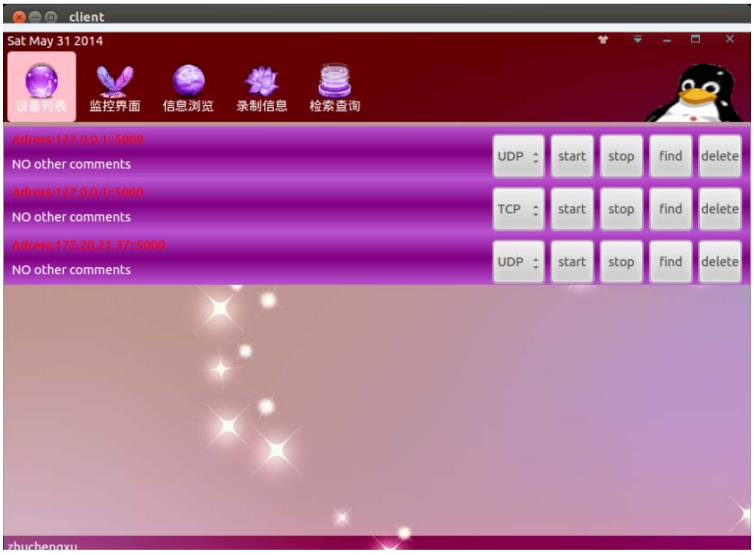

图 5.12 客户端的“设备列表”界面

Fig 5.12 Equipment lists interface

在设备蓝条的最右侧是针对该设备的功能按钮区，第一个用于选择协议方式，选项有“TCP”和“UDP”；下一个是“start”按钮，“start”的功能是开始连接服务器；“stop”按钮的功能是停止连接。“delete”按钮是删除该设备条目；“find”按钮的功能室找到视频的显示区，点击后会出现图5.13的界面。

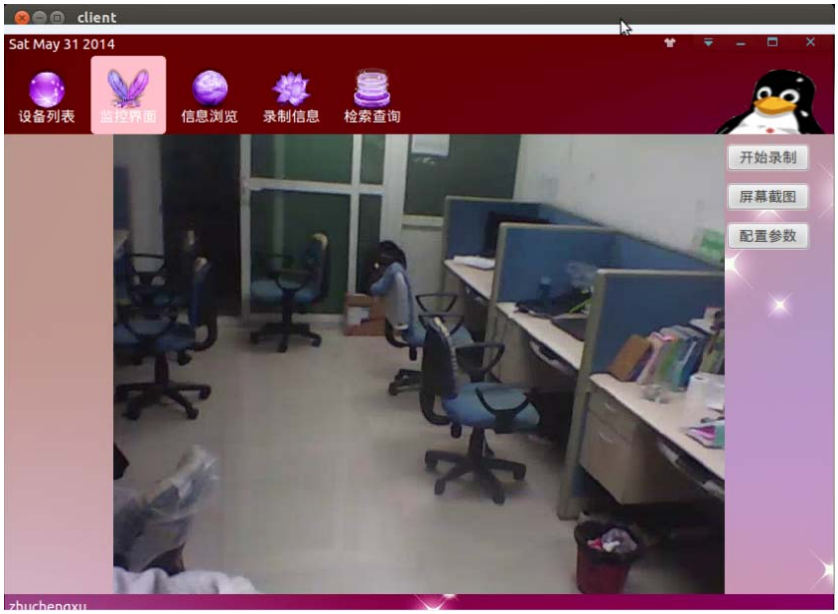

图 5.13 监控主界面

Fig 5.13 Main interface for monitoring

图5.13是视频的主显示区。该显示区所对应的服务器或是被点击了“find”按钮的设备，或是上一个被点击了“find”按钮的设备（用户通过点击“设备列表”功能选项进入显示界面区）。

点击了“信息浏览”功能选项后，会出现图5.14的界面：

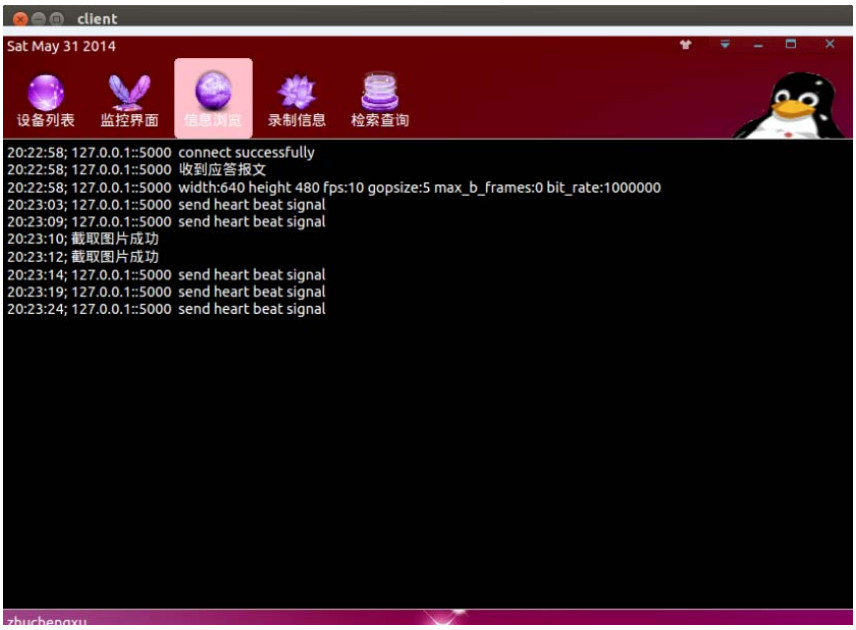

图 5.14 信息浏览界面

Fig 5.14 Information interface

5.5 本章小结

本章完成了客户端GUI的设计与实现。开始部分给出了Qt框架最吸引人的一个编程机制：信号和槽机制。为了描述客户端复杂的设计，随后又给出了GUI中出现的大部分类的继承关系，包括自己实现的类和Qt框架中的类。然后从类间的控制流和GUI中的功能模块这两个方面描述了客户端的设计方法。本章的最后是客户端的测试结果，给出了“设备列表”、“监控界面”和“信息浏览”这三个功能选项的主界面。

6 总结与展望

计算机技术与网络技术让视频的数字化、网络化成为可能，并让视频监控的数字化逐步走向成熟。本文主要对视频数据的采集、编解码和相关平台工具进行了研究，并将FFmpeg这一优秀的开源多媒体框架纳入到视频监控领域。主要做的工作如下：

① 研究了Linux平台的通用视频采集接口，实现了获取视频数据的采集算法。

② 介绍了FFmpeg多媒体解决方案。给出了编解码、混流的一般步骤。并将H.264的编码标准用于视频监控领域。

③ 用运输层的两种协议分别编写了视频监控服务端，满足了不同用户的需求。对不同的传输协议采取了不同的设计模式，在稳定性和并发性上找到了平衡点。采用了变化码率且独立配置的策略。

④ 制定了应用层的传输协议，改变了服务器主导的行为方式，降低了客户端与服务器交互的复杂度，为可靠的通信提供了保障。

⑤ 介绍了Qt开发框架，编写了多连接、跨平台的监控客户端。为多平台的快速部署提供了可能。

该系统基本达到了预期目标，但是也有很多不足的地方，主要体现在：

① 在客户端没有一个多显示窗口的解决方案，因此当用户想要同时查看多个连接的图像时，只能在“设备列表”功能项和“监控界面”功能项之间来回切换，这并不方便。

② 制定的应用层协议并不紧凑，有字节浪费的情况出现。这点在以后的优化中需要修改。

③ 客户端只是负责图像的接收、显示和混流，并没有提供一个进行针对图像内容的后期智能化处理方法，这在当今的实时监控中正显得尤为重要，智能化的监控是未来要努力的一个方向。

致　谢

感谢我的导师刘京诚教授，刘老师有着严谨的治学态度，广博的专业知识与高度的责任感。三年里，无论是生活还是学习中，我都得到了最无私的帮助与最悉心的指导。如果没有刘老师的关怀，我想我坚持不下来。

感谢实验室的兄弟姐妹们，在三年的时间里，我得到了你们许许多多的帮助，我想说，遇见你们是我最开心的事，是我三年来最温馨的记忆。如果没有你们的陪伴，我不知我究竟会成什么样子。

感谢我的朋友刘志萍，没有你，我不知我现在会停在哪里。

感谢我的父母，你们健康、快乐是我最大的心愿。

这是最好的时光。

薄建彬

二０一四年四月　于重庆

参考文献

[1] 宋磊,黄祥林, 沈兰荪. 视频监控系统概述[J]. 测控技术, 2003, 22(5):1-3.

[2] 谢希仁. 计算机网络[M]. 5. 北京:电子工业出版社, 2008:1-3.

[3] 郑磊. 基于嵌入式 Linux 的网络视频监控系统研究[D]. 武汉:武汉理工大学, 2009.

[4] 张文涯. 基于嵌入式 Linux 的网络视频监控系统设计与实现[D]. 成都:西南交通大学, 2009.

[5] 程少炼. 基于 H.264 的嵌入式视频监控系统的研究与实现[D]. 武汉:武汉科技大学, 2011.

[6] 梁志聪. 嵌入式网络视频监控服务器的设计与实现[D]. 武汉:华中科技大学, 2007.

[7] Motion JPEG http://zh.wikipedia.org/wiki/Motion_JPEG

[8] MPEG-4 http://zh.wikipedia.org/wiki/MPEG-4

[9] MPEG-4 框架 http://baike.baidu.com/

[10] 谈新权, 邓大平. 视频技术基础[M]. 武汉:华中科技大学出版社, 2004.

[11] 何华丽. 基于 H264 的多平台视频监控系统的研究与实现[D]. 北京:北京邮电大学, 2009.

[12] 左璐. 嵌入式系统现状与发展前景研究[J]. 现代商贸工业, 2010(15).

[13] 石沙. 基于 Qt 的跨平台视频监控客户端的设计与实现[D]. 西安:西安电子科技大学, 2013.

[14] Blanchette J, Summerfield M. C++ GUI Programming with Qt4[M]. Prentice Hall, 2008:Appendix B.

[15] Video4Linux Version2 http://en.wikipedia.org/wiki/Video4Linux

[16] Gspca development history http://linuxtv.org/wiki/index.php/Gspca

[17] Stevens W, ARago S. Advanced Programming in the UNIX Environment[M]. 2. Addison-Wesley Educational Publishers Inc, 2005:83.

[18] 张辉. 基于 V4L2 的嵌入式视频驱动程序开发与实现[D]. 合肥:安徽大学, 2010.

[19] 王燕乐. 网箱远程监控终端的设计与实现[D]. 舟山:浙江海洋学院, 2012.

[20] About FFmpeg http://ffmpeg.org/about.html

[21] 黄诗文. 基于 ffmpeg 的高性能高清流媒体播放器软件设计[D]. 杭州:浙江大学, 2012.

[22] 刘洁彬. 面向实时监控的流媒体播放器的设计与实现[D]. 北京:北京邮电大学, 2010.

[23] 郑谦益. GNU/Linux 编程[M]. 北京:人民邮电出版社, 2012.

[24] 韦东山. 嵌入式 Linux 应用开发完全手册[M]. 北京:人民邮电出版社, 2008:29-40.

[25] 刘春海. 基于网络的视频监控平台设计与实现[D]. 哈尔滨:哈尔滨工业大学, 2008.

[26] 张晓林, 崔迎炜. 嵌入式系统设计与实现[M]. 北京:北京航空航天大学出版社, 2006:42-58.

[27] 潘超. 基于 Linux 的人脸识别系统的设计与实现[D]. 西安:西安电子科技大学, 2012.

[28] 何勋. 基于 S3C2440 的 H.264 软编解码器移植及优化[D]. 成都:电子科技大学, 2010.

[29] FFmpeg Compilation Guide http://trac.ffmpeg.org/wiki/CompilationGuide

[30] 廖志川. 基于 ARM 的便携式视频监控终端设计与实现[D]. 南昌:南昌航空大学, 2012.

[31] W.Richard Stevens 著, 范建华, 胥光辉等译. TCP/IP 详解卷 1[M]. 北京:机械工业出版社, 2000:1-10.

[32] W.Richard Stevens 著, 杨继张译. UNIX 网络编程[M]. 北京:清华大学出版社, 2006:49.

[33] Linux 下常用的 I/O 模型 http://www.ccvita.com/503.html

[34] 宋敬彬. Linux 网络编程[M]. 2. 北京:清华大学出版社, 2014.

[35] Linux/unix Command:setitimer http://linux.about.com

[36] 杨铸. Linux 下 C 语言应用编程[M]. 北京:北京航空航天大学出版社, 2012.

[37] Neil Matthew.Richard Stones 著, 陈健, 宋健健译. Linux 程序设计[M]. 4. 北京:人民邮电出版社, 2010:513-520.

[38] 柳亚东. 基于 S3C2440 的嵌入式视频网络监控系统[D]. 上海:上海交通大学, 2009.

[39] 张长春. 基于 ARM9 S3C2440 视频监控系统的研究与实现[D]. 广州:广东工业大学, 2010.

[40] Mark Summerfield 著, 白建平等译. Qt 高级编程[M]. 北京:电子工业出版社, 2011:190-195.

[41] 甘刚. Linux/UNIX 网络编程[M]. 北京:水利水电出版社, 2008.