第 5 章

CHAPTER 5

FFmpeg 二次开发

IPC 视频监控

视频监控是指使用摄像头对人或物体照相，并将其实时转换为计算机可以处理的电子信号，然后存储在磁盘或与网络相连的信息服务器上，以便能够远程观看，或者利用安全系统中的人工智能进行自动视频分析、报警和报警处理的技术系统。结合 FFmpeg 技术，可以很方便地开发 IPC 视频监控系统。

3min

5.1 视频监控系统简介

视频监控(Cameras and Surveillance)是安全防范系统的重要组成部分，传统的监控系统包括前端摄像机、传输线缆、视频监控平台。摄像机可分为网络数字摄像机和模拟摄像机，可对前端视频图像信号进行采集。视频监控是一种防范能力较强的综合系统，以其直观、准确及时和信息内容丰富而广泛应用于许多场合。近年来，随着计算机、网络及图像处理、传输技术的飞速发展，视频监控技术也有了长足的发展。最新的监控系统可以使用智能手机担当，同时对图像进行自动识别、存储和自动报警。视频数据通过 3G/4G/5G/WiFi 传回控制主机(也可以由智能手机担当)，主机可对图像进行实时观看、录入、回放、调出及存储等操作，从而实现移动互联的视频监控。

完整的视频监控系统由摄像、传输、控制、显示和记录登记 5 大部分组成。摄像机通过网络线缆或同轴视频电缆将视频图像传输到控制主机，控制主机再将视频信号分配到各监视器及录像设备，同时可将需要传输的语音信号同步录入录像机内。通过控制主机，操作人员可发出指令，对云台的上、下、左、右的动作进行控制及对镜头进行调焦变倍的操作，并可通过视频矩阵实现多路摄像机的切换。利用特殊的录像处理模式，可对图像进行录入、回放、调出及存储等操作。视频监控主要包括摄像机、光圈镜头、硬盘录像机、矩阵、控制键盘和监视器等。视频监控的应用越来越广泛，例如在智能家居系统中，视频监控系统属于家庭安防系统的一部分，是一个常见的选配系统，尤其是在别墅应用中。完整的视频监控系统主要包括以下几个组成部分。

(1) 视频采集系统：主要由各观测点的摄像机组成，完成视频图像信号采集。用于采

集被监控点的监控信息，并可以配备报警设备。监控前端可分为普通摄像头和网络摄像头。普通摄像头可以是模拟摄像头，也可以是数字摄像头。原始视频信号传到视频服务器，经视频服务器编码后，以 TCP/IP 通过网络传至其他设备。网络摄像头(IPC)是融摄像、视频编码、Web 服务于一体的高级摄像设备，内嵌了 TCP/IP 协议栈，可以直接连接到网络。

(2) 云台镜头控制系统：主要由云台和控制器组成，用于完成在监控中心遥控摄像机的观测位置的变动和观测点图像的放大、缩小处理。

(3) 信号传输系统：分为有线和无线传输，并且传输方式和传输线材对信号影响较大。

(4) 视频处理系统：主要完成对视频信号的数字化处理、图像信号的显示、图像信号的存储及图像信号的远程传输。

(5) 管理中心：承担所有前端设备的管理、控制、报警处理、录像、录像回放、用户管理等工作，各部分功能分别由专门的服务器各司其职。

(6) 监控中心：用于集中对所辖区域进行监控，包括电视墙、监控客户终端群，系统中可以有一个或多个监控中心。

(7) PC 客户端：在监控中心之外，也可以由 PC 接到网络上进行远程监控。

(8) 无线网桥：用于接入无线数据网络，并访问互联网。通过无线网桥，可以将 IP 网上的监控信息传至无线终端，也可以将无线终端的控制指令传给 IP 网上的视频监控管理系统。

5.1.1 视频监控系统的功能及特点

视频监控系统一般采用高清视频监控技术，实现视频图像信息的高清采集、高清编码、高清传输、高清存储、高清显示；基于 IP 网络传输技术，提供视频质量诊断等智能分析技术，实现全网调度、管理及数字化应用，为用户提供一套“高清化、网络化、数字化”的视频图像监控系统，满足用户在视频图像业务应用中日益迫切的需求。一般而言，需要建成统一的中心管理平台，通过管理平台实现全网统一的视频资源管理，对前端摄像机、编码器、解码器、控制器等设备进行统一管理，实现远程参数配置与远程控制等；通过管理平台实现全网统一的用户和权限管理，满足系统多用户的监控、管理需求，真正做到“坐阵于中心，掌控千里之外”。需要实现系统高清化与网络化，以建设全高清监控系统为目标，为用户提供更清晰的图像和细节，让视频监控变得更有使用价值；同时以建设全 IP 监控系统为目标，让用户可通过网络中的任何一台计算机来观看、录制和管理视频信息，并且系统组网便利，结构简单，新增监控点或客户端都非常方便。视频监控系统需要具备以下几个特征。

(1) 系统具备高可靠性、高开放性的特征：通过采用业内成熟、主流的设备来提高系统可靠性，尤其是录像存储的稳定性。另外，系统可接入其他厂家的摄像机、编码器、控制器等设备，能与其他厂家的平台无缝对接。

(2) 具备高数字化、低码流的特征：运用智能分析、带有智能功能的摄像机等提高系统的数字化水平，同时通过先进的编码技术降低视频码流，降低存储成本和网络成本，减弱对网络的依赖性，提高视频预览的流畅度。

(3) 具备快速部署及时维护的特征：通过采用高集成化、模块化设计的设备提高系统部署效率，减少系统调试周期，系统能及时发现前端监控系统的故障并及时告警，快速响应。

(4) 具备高度整合、充分利旧的特征：新建系统能与原有系统高度整合、无缝对接，能充分利用原有监控资源，避免前期投资的浪费。

(5) 安全性高，使用图像掩码技术，防止非法篡改录像资料，只有授权用户才可以进入系统进行查看，调用视频资料，可以对不同身份的管理人员发放不同权限的管理账号；有效防止恶意破坏；配合强大日志管理功能，保证了专用系统的安全使用。服务器端和客户端之间所传输的数据，全部经过加密。

(6) 服务器平台构架方便，在大楼监控机房(如市公安局、区公安局和各派出所)都可以方便地安装客户端软件，只需分配用户不同权限的登录账号，便可查看前端摄像机监控点的图像资料。

(7) 权限管理为了保证上网人员的隐私和录像资料的安全，具有操作权限管理，系统登录、操作进行严格的权限控制，保证系统的安全性。

(8) 远程视频监控人员可远程任意调取网吧存储的监控图像，并可远程发出控制指令，进行录像资料的智能化检索、回放、调整摄像机镜头焦距、控制云台巡视或局部细节观察。

(9) 可以本地录像，保存一定时间段内的本地视频监控录像资料，并能方便地查询、取证，为事后调查提供依据。

(10) 随时随地的监控录像功能，无论身在何处，任何密码授权的用户通过身边的计算机联网连接到监控网点，可以看到任意监控网点的即时图像并根据需要录像，避免了地理位置间隔原因而造成监督管理的不便。

(11) 系统可扩容性强，若需要添加新的监控网点，则可在服务器端添加相应的子节点和设备信息。

(12) 可以和电子地图相结合，可以通过电子地图更加直观地查看各监控点所分布的地理位置，并且在电子地图上实时显示监控设备的运行状态。

5.1.2　视频监控系统的工作原理及结构

对于视频监控系统，根据系统各部分功能的不同，整个系统可以划分为 7 层，如图 5-1 所示，主要包括表现层、控制层、处理层、传输层、执行层、支撑层和采集层。当然，由于设备集成化越来越高，对部分系统而言，某些设备可能会同时以多个层的身份存在于系统中。

视频监控系统的各层含义如下所示。

(1) 表现层可以被最直观地感受到，它展现了整个视频监控系统的品质，如监控电视墙、监视器、高音报警扬声器、报警自动驳接电话等都属于这一层。

(2) 控制层是整个视频监控系统的核心，它是系统科技水平的最明确体现。常见的控制方式有两种，即模拟控制和数字控制。模拟控制是早期的控制方式，其控制台通常由控制器或者模拟控制矩阵构成，适用于小型局部视频监控系统，这种控制方式成本较低，故障率较小，但对于中大型视频监控系统而言，这种方式就显得操作复杂且无任何价格优势了，这

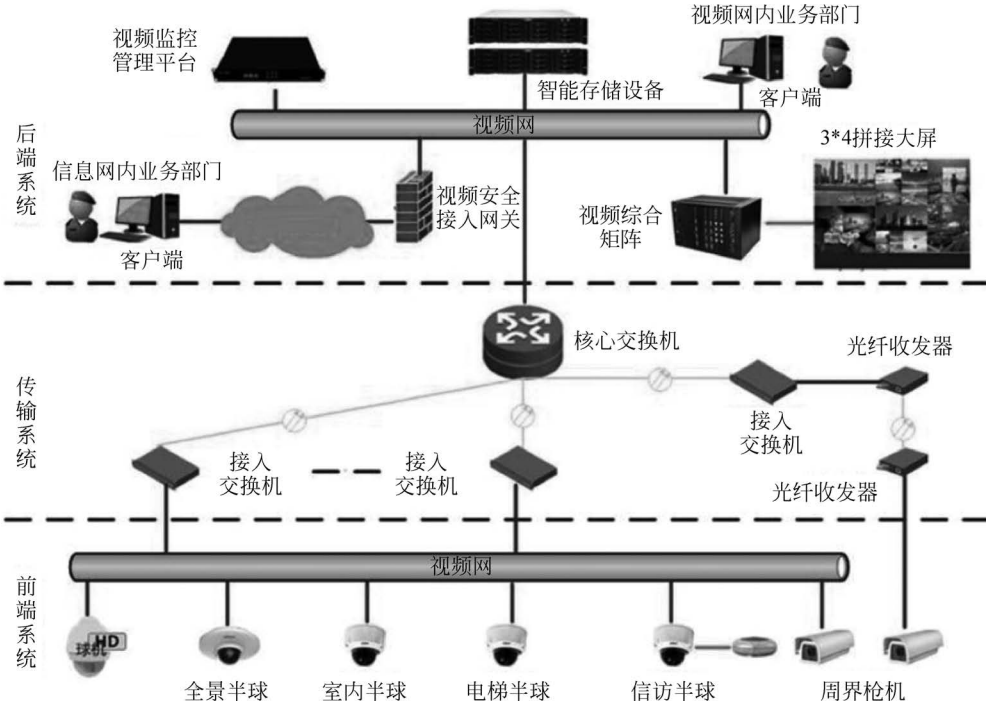

图 5-1　视频监控系统架构

时更为明智的选择应该是数字控制。数字控制是将工控计算机作为监控系统的控制核心，它将复杂的模拟控制操作变为简单的鼠标单击操作，将巨大的模拟控制器堆叠缩小为一个工控计算机，将复杂而数量庞大的控制电缆变为一根串行电话线；将中远程视频监控变为事实、为因特网远程监控提供可能，但数字控制也不是十全十美的，控制主机的价格十分昂贵、模块浪费的情况、系统可能出现全线崩溃的危机、控制较为滞后等问题仍然存在。

（3）处理层（音视频处理层）将传输层送过来的音视频信号加以分配、放大、分割等处理，将表现层与控制层加以连接。音视频分配器、音视频放大器、视频分割器、音视频切换器等设备都属于这一层。

（4）传输层相当于视频监控系统的血脉。在小型视频监控系统中，最常见的传输层设备是视频线、音频线；对于中远程监控系统而言，常使用的是射频线、微波；对于远程监控而言，通常使用因特网这一廉价载体。大多数人在数字安防监控上存在一个误区，认为控制层使用的数字控制的视频监控系统就是数字视频监控系统了，其实不然。纯数字视频监控系统的传输介质一定是网线或光纤。信号从采集层出来时，就已经被调制成数字信号了，数字信号在已趋成熟的网络上传输在理论上是无衰减的，这就保证远程监控图像的无损失显示，这是模拟传输无法比拟的。当然，高性能的回报也需要高成本的投入。

（5）执行层是控制指令的命令对象，在某些时候，它和支撑层、采集层不太容易截然地分开，一般认为受控对象即为执行层设备。例如，云台、镜头、解码器和球机等。

(6) 支撑层用于后端设备的支撑,保护和支撑采集层、执行层设备,包括支架、防护罩等辅助设备。

(7) 采集层是整个视频监控系统品质好坏的关键因素,也是系统成本开销最大的地方。它包括镜头、监控摄像机和报警传感器等。

5.1.3　视频监控系统的总体结构设计

通常情况下,视频监控系统以用户需求为出发点,以用户价值为落脚点,并结合产品亮点进行组合设计。系统具备很多优势,如下所示。

(1) 有利于系统维护:可以采用视频质量诊断技术,自动地对前端监控点的视频图像是否完好、设备是否在线等进行实时、不间断的检测,以及时发现前端系统运行发生的问题并告警通知,有效保障系统高质量运行。

(2) 方便系统部署:实现软件与硬件部署的一体化、视频解码与上墙显示的一体化及网络、模拟、数字视频信号可集中处理的一体化,方便安装调试,减少了部署时间。

(3) 方便系统扩容:采用标准化的设备,可接入第三方平台软件,而且平台开放性高,可兼容其他厂家的摄像机、存储等设备;视频综合平台采用模块化设计,设计时留有一定的冗余,方便系统后期的升级与扩容。

(4) 降低存储和网络传输成本:可以采用码流低的摄像机,最大可减少 3/4 的存储占用空间,降低了存储成本;通过采用低码流的网络高清智能摄像机,同等图像质量下,720P 码率只需 1～2MB,1080P 码率只需 3～4MB,从而降低了网络开销,降低了网络成本。

(5) 降低系统功耗:从前端摄像机到存储都采用新技术可降低功耗,从整体上降低功耗,达到节能减排的效果。

(6) 良好的视觉效果:实现全高清模式,并且可实现对大场景进行高清监控,满足用户对高清监控的需求,提高用户的体验度;通过先进的智能编码技术,有效地降低视频码流,减少视频预览不流畅等现象。

(7) 便捷的管理效果:实现全网络监控,满足用户对数字化组网的要求,方便用户对系统网络化管理,轻松做到足不出户就能掌控全局;采用智能网络摄像机、智能球机和智能分析技术,体现了高度的数字化水平,可让用户体验丰富的智能效果。

网络高清方案从逻辑上可分为视频前端系统、传输网络、视频存储系统、视频解码拼控、大屏显示和视频信息管理应用平台等几部分。

(1) 视频前端系统:前端支持多种类型的摄像机接入,例如配置高清网络枪机、球机等网络设备,按照标准的音视频编码格式及标准的通信协议,可直接接入网络并进行音视频数据的传输。

(2) 传输网络:负责将前端的视频数据传输到后端系统。

(3) 视频存储系统:视频存储系统负责对视频数据进行存储,例如可以配置云存储进行数据存储。

(4) 视频解码拼控:完成视频的解码、拼接、上墙控制,例如可以配置视频综合平台以

实现对前端所有种类视频信号的接入，完成视频信号以多种显示模式进行输出。

（5）大屏显示：接收视频综合平台输出的视频信号，完成视频信号的完美呈现。

（6）视频信息管理应用平台：负责对视频资源、存储资源、用户等进行统一管理和配置，用户可通过应用平台进行视频预览、回放。

网络高清方案的物理拓扑如图 5-2 所示。

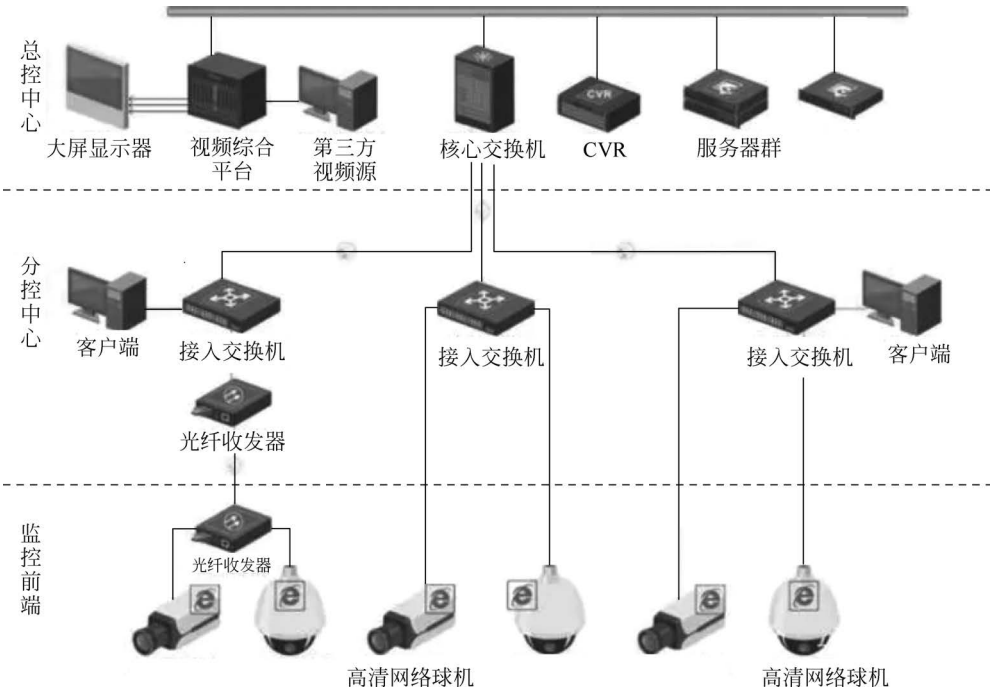

图 5-2　高清视频监控物理拓扑结构

网络高清方案物理拓扑结构的组成部分如下所示。

（1）总控中心：负责对分控中心分散区域高清监控点的接入、显示、存储、设置等，主要部署核心交换机、视频综合平台、大屏、云存储、客户端、平台、视频质量诊断服务器等。

（2）分控中心：负责对前端分散区域高清监控点的接入、存储、浏览、设置等功能，主要部署接入交换机、客户端等。

（3）监控前端：主要负责各种音视频信号的采集，通过部署网络摄像机、球机等设备，将采集到的信息实时传送至各个监控中心。

（4）传输网络：整个传输网络采用接入层、核心层两层传输架构设计。前端网络设备就近连接到接入交换机，接入交换机与核心交换机之间通过光纤连接。部分设备因传输距离问题通过光纤收发器进行信号传输，再汇入接入交换机。

（5）视频存储系统：视频存储系统采用集中存储方式，使用云存储设备，支持流媒体直存，减少了存储服务器和流媒体服务器的数量，确保了系统架构的稳定性。

（6）视频解码拼控：视频综合平台通过网线与核心交换机连接，并通过多链路汇聚的

方式提高网络带宽与系统可靠性。视频综合平台采用电信级 ATCA 架构设计，集视频智能分析、编码、解码、拼控等功能于一体，极大地简化了监控中心的设备部署，更从架构上提升了系统的可靠性与健壮性。

(7) 大屏显示：大屏显示部分采用最新 LCD 窄缝大屏拼接显示。

(8) 视频信息管理应用平台：部署于通用的 x86 服务器上，服务器直接接入核心交换机。

其中，根据不同场景的不同需求，可以灵活地选择合适的前端监控产品，满足室内外各种场景下的监控需求。网络高清摄像机通过其全新的硬件平台和最优的编码算法提供高效的处理能力和丰富的功能应用，旨在向用户提供最优质的图像效果、最丰富的监控价值、最便捷的操作管理和最完善的维护体系。例如，室外可以依据固定枪机与球机搭配使用和交叉互动原则，以保证监控空间内的无盲区、全覆盖，同时根据实际需要配置前端基础配套设备，如防雷器、设备箱及视频传输设备和线缆；室内可以采用红外半球与室内球机搭配使用，确保满足安装的美观与细节的不丢失需求要求。

视频解码拼控系统采用集图像处理、网络功能、日志管理、设备维护于一体的电信级综合处理平台设计，即视频综合平台，满足数字视频切换、视频编解码、视频编码数据网络集中存储、电视墙管理、开窗漫游显示等功能。

大屏显示系统不仅包含用来显示视频图像的大屏显示部分，还包括解码控制等产品。大屏显示系统建设的总体目标是：系统充分考虑到先进性、可靠性、经济性、可扩充性和可维护性等原则，建成一套采用先进成熟的技术、遵循布局设计优良、设备应用合理、界面友好简便、功能有序实用、升级扩展性好的液晶大屏幕拼接系统，以达到满足大屏幕图像和数据显示的需求。整个大屏系统可以分为以下几部分。

(1) 前端信号接入部分：大屏显示系统支持各类型信号的接入，如模拟摄像机、高清数字摄像机、网络摄像机等信号，除能接入远端摄像机之外，还能接入本地的 VGA 信号、DVD 信号及有线电视信号等，满足用户接入所有信号类型的需求。

(2) 解码、控制部分：前端摄像机信号接入视频综合平台之后，可由视频综合平台对各种信号进行解码和控制，输出到大屏显示屏幕上，并可通过在控制主机上安装的拼接控制软件实现对整个大屏显示系统的控制与操作，实现上墙显示信号的选择与控制。

(3) 上墙显示部分：大屏显示系统支持 BNC、VGA、DVI 和 HDMI 等多种信号的接入显示，通过控制软件对已选择需要上墙显示的信号进行显示，通过视频综合平台可实现信号的全屏显示、任意分割、开窗漫游、图像叠加、任意组合显示和图像拉伸缩放等一系列功能。

5.1.4　视频监控系统的存储结构设计

随着视频监控系统的规模越来越大，以及高清视频的大规模应用，视频监控系统中需要存储的数据和应用的复杂程度在不断提高，并且视频数据需要长时间持续地保存到存储系统中，并要求随时可以调用，对存储系统的可靠性和性能等方面都提出了新的要求。在未来的复杂系统中，数据将呈现爆炸性海量增长，提供对海量数据的快速存储及检索技术，显得

尤为重要，存储系统正在成为视频监控技术未来发展的决定性因素。

面对百 PB 级的海量存储需求，传统的 SAN 或 NAS 在容量和性能的扩展上会存在瓶颈，而云存储可以突破这些性能瓶颈，而且可以实现性能与容量的线性扩展，这对于追求高性能、高可用性的企业用户来讲是一个新选择。云存储是在云计算(Cloud Computing)概念上延伸和发展出来的一个新的概念，是指通过集群应用、网格技术或分布式文件系统等功能，应用存储虚拟化技术将网络中大量各种不同类型的存储设备通过应用软件集合起来协同工作，共同对外提供数据存储和业务访问功能的一个系统，所以云存储可以认为是配置了大容量存储设备的一个云计算系统。依据云存储的功能特点，针对大容量视频数据的存储和管理及满足视频监控领域特殊的应用需求，量身设计了一套视频云存储监控系统。

视频云存储监控系统可以同时应用于视频、图片混合存储，承担整个系统内的视频/图片的数据写入/读取工作。云存储监控系统一方面采用了基于云架构的分布式集群设计和虚拟化设计，在系统内部实现了多设备协同工作、性能和资源的虚拟整合，最大限度地利用了硬件资源和存储空间；另一方面，通过对云存储的存储功能、管理功能进行打包，通过开放透明的应用接口和简单易用的管理界面，与上层应用平台整合后，为整个安防监控系统提供了高效、可靠的数据存储服务。

在视频云存储监控系统的设计中，采用的核心技术如下：

(1) 采用存储全域虚拟化技术对具有海量存储需求的用户提供透明存储构架，可持续扩容以避免瓶颈限制，可以更有效地进行资源管理，灵活增减空间，达到最大程度地合理利用空间的效果。

(2) 采用集群技术，解决单/多节点失效问题，并利用负载均衡技术及各存储节点的性能，提升系统的可靠性和安全性。

(3) 采用离散存储技术，在保障用户高效地进行读写的同时保证了业务的持续性。

(4) 采用统一完善的接口，降低对接成本、平台维护成本和用户管理的复杂度。

(5) 采用开放的集成构架，使其可兼容业界各类 iSCSI/FC 存储设备，保护用户现有存储投资资源。

(6) 采用数据备份和容灾技术，保证云存储中的数据不丢失，保证云存储服务的安全稳定。

云存储监控系统采用分层结构设计，整个系统从逻辑上分为 5 层，分别为设备层、存储层、管理层、接口层和应用层，如图 5-3 所示。

(1) 设备层：是云存储最基础、最底层的部分，该层由标准的物理设备组成，支持标准的 IP-SAN、FC-SAN 存储设备。在系统组成中，存储设备可以是 SAN 架构下的 FC 光纤通道存储设备或 iSCSI 协议下的 IP 存储设备。

(2) 存储层：在存储层上部署云存储流数据系统，通过调用云存储流数据系统，实现存储传输协议和标准存储设备之间的逻辑卷或磁盘阵列的映射，实现数据(视频、图片、附属流)和设备层存储设备之间的通信连接，完成数据的高效写入、读取和调用等服务。

(3) 管理层：融合了索引管理、计划管理、调度管理、资源管理、集群管理、设备管理等

图 5-3　云存储逻辑架构图

多种核心的管理功能。该层可以实现存储设备的逻辑虚拟化管理、多链路冗余管理、录像计划的主动下发，以及硬件设备的状态监控和故障维护等；整个存储系统的虚拟化的统一管理；上层服务（视频录像、回放、查询、智能分析数据请求等）的响应。

（4）接口层：应用接口层是云存储最灵活多变的部分，接口层面向用户应用提供完善及统一的访问接口。接口类型可分为 Web Service 接口、API、Mibs 接口，可以根据实际业务类型，开发不同的应用服务接口，提供不同的应用服务。该层可以实现和行业专属平台、运维平台的对接，及与智能分析处理系统之间的对接；视频数据的存储、检索、回放、浏览转发等操作；关键视频数据的远程容灾；设备及服务的监控和运维等。

（5）应用层：从逻辑上划分，除了应用层外，其他 4 层都属于通常云存储的范畴，但是在视频云存储监控系统中，为了与视频监控系统的建设和应用更加紧密地结合，更加符合用户的业务需求，将应用层纳入了整个系统架构中，从根本上提高了视频云存储监控系统的针对性。

可将行业视频监管平台、运维平台、智能分析平台等通过相应的接口与云存储监控系统对接，实现与云存储监控系统之间的数据及信令的交互。行业视频监控平台可与云存储系统进行配置录像计划、配置存储策略、检索视频资源、重要录像的备份存储等指令的交互，辅助流数据、视频数据、图片数据的存取。运维平台采用标准的 SNMP 实现并提供 Mibs 接

口，对云存储系统及服务进行监控管理，以便及时地将产生的告警传递给用户。将智能分析平台可与云存储系统进行对接，实现基础数据的读取，以及对经过存储的二次分析后的片段信息和文本信息进行写入和检索。

云存储监控系统主要由管理节点和存储节点（物理存储设备）两部分组成，如图 5-4 所示。系统内部需要配置的元数据信息由管理服务器统一管理，管理节点还需要负责集群内部的负载均衡、失败替换等管理职能；视频云存储监控系统可以组建海量的存储资源池，容量分配不受物理硬盘数量的限制，并且存储容量可进行线性在线扩容，性能和容量的扩展都可以通过在线扩展完成。

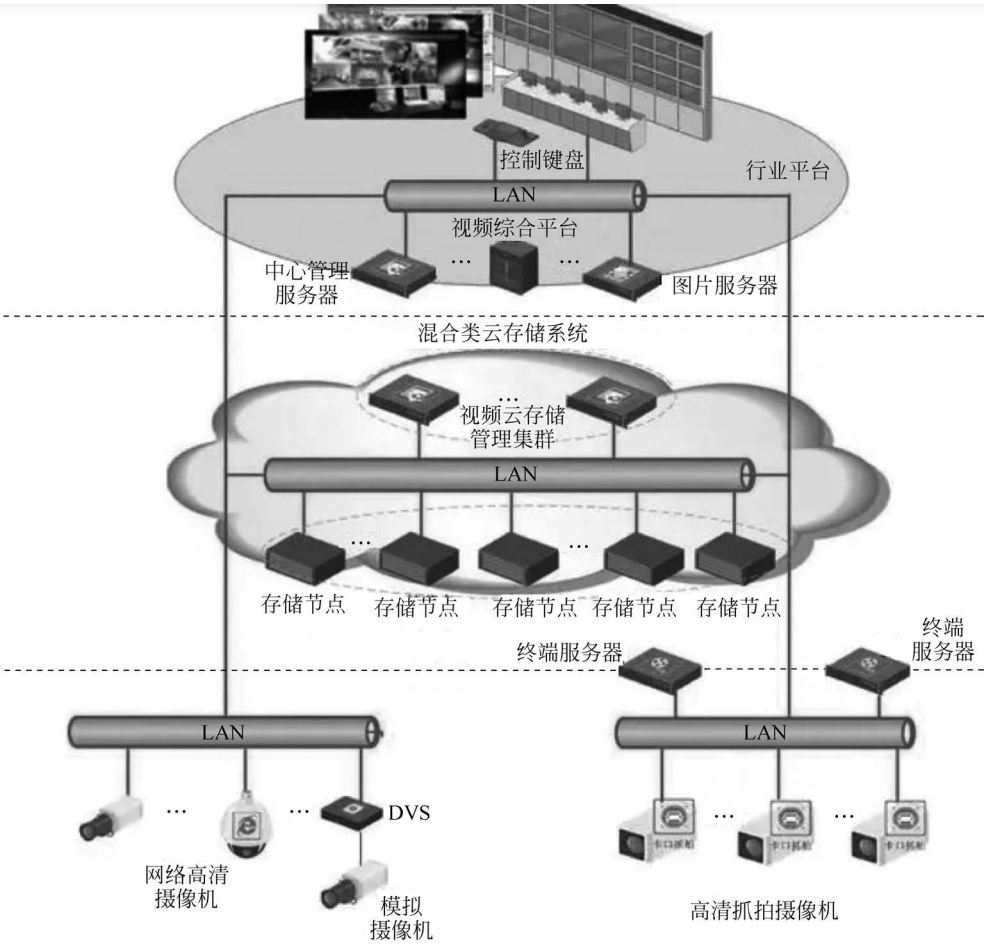

图 5-4　云存储物理架构图

（1）视频云存储管理节点（CVMN）：部署管理服务器是视频云存储监控系统的核心节点，作为云存储监控系统的调度中心，负责云存储监控系统资源管理、索引管理、计划管理、策略调度等。根据项目对存储容量需求、前端支撑数目、性能要求和可靠性要求，存储管理节点可以按照两种方式部署，即双机部署和集群部署。

(2) 视频云存储节点(CVSN)：作为云存储监控系统业务的具体执行者，负责视频数据存储、读取、存储设备管理、存储空间管理等。

5.2　FFmpeg 读取网络摄像头

使用 FFmpeg 可以打开网络摄像头并循环读取视频帧数据，然后可以调用 Qt 对视频帧进行渲染。打开 Qt Creator，创建一个基于 Widget 的 Qt Widgets Application 项目(项目名称为 FFmpegMonitor_Libx264)，在.pro 文件中配置 FFmpeg 的头文件和库文件路径，代码如下：

```
//chapter5/FFmpegMonitor_Libx264/FFmpegMonitor_Libx264.pro
INCLUDEPATH += $$PWD/FFmpeg431dev/include

LIBS += $$PWD/FFmpeg431dev/lib/avcodec.lib \
        $$PWD/FFmpeg431dev/lib/avdevice.lib \
        $$PWD/FFmpeg431dev/lib/avfilter.lib \
        $$PWD/FFmpeg431dev/lib/avformat.lib \
        $$PWD/FFmpeg431dev/lib/avutil.lib \
        $$PWD/FFmpeg431dev/lib/swresample.lib \
        $$PWD/FFmpeg431dev/lib/swscale.lib
```

运行并编译该项目，程序运行效果如图 5-5 所示。笔者这里只演示了一路 RTSP 流，读者可以配置多路网络摄像头进行测试。通常情况下，市面上的 IPC 都支持 RTSP 流，另外也可以通过 VLC 来推送 RTSP 流。

图 5-5　基于 Qt 和 FFmpeg 的视频监控客户端

使用FFmpeg读取网络摄像头和读取本地摄像头的流程和代码几乎一模一样，这也是FFmpeg的优势。笔者封装了一个C++类（AVFFmpegNetCamera），专门用来读取网络摄像头，该类的头文件中的代码如下：

```
//chapter5/FFmpegMonitor_Libx264/avffmpegnetcamera.h
#ifndef AVFFMPEGNETCAMERA_H
#define AVFFMPEGNETCAMERA_H

#include <QObject>
#include <QMutex>
#include <QImage>
#include <QThread>

extern "C"{
    #include <libavcodec/avcodec.h>
    #include <libavformat/avformat.h>
    #include <libavfilter/avfilter.h>
    #include <libswscale/swscale.h>
    #include <libavutil/frame.h>
};

#include "t3ffmpegh2645encoder2.h"

class AVFFmpegNetCamera : public QObject
{
   Q_OBJECT
public:
    explicit AVFFmpegNetCamera(QObject *parent = nullptr);

    bool Init();
    void Play();
    void Deinit();

    void SetUrl(QString url){this->url = url;}
    QString Url()const{return url;}
    int VideoWidth()const{ return videoWidth; }
    int VideoHeight()const{return videoHeight;}
    void setStopped(int st){ this->m_stopped = st; }
    void setChannelIndex(int ci) {this->m_channelIndex = ci;}
    void setIsPlayingbacked(int np){this->m_bIsPlayingbacked = np; }

    //声明成员变量
private:
    QMutex mutex;
    AVFormatContext *pAVFormatContext;
    AVCodecContext *pAVCodecContext;
    AVFrame *pAVFrame;
    SwsContext *pSwsContext;
```

```
    AVPacket pAVPacket;
    AVPicture pAVPicture;

    QString url;
    int videoWidth;
    int videoHeight;
    int videoStreamIndex;
    int m_channelIndex;

    int m_stopped;
    int m_bIsPlayingbacked;

signals:
    void GetImage(const QImage &image);

public slots:

};

#endif //AVFFMPEGNETCAMERA_H
```

该类的构造函数主要用于进行变量的初始化工作，代码如下：

```
//chapter5/FFmpegMonitor_Libx264/avffmpegnetcamera.cpp
AVFFmpegNetCamera::AVFFmpegNetCamera(QObject * parent)
    : QObject(parent)
{
    pAVFormatContext = NULL;
    pAVCodecContext = NULL;
    pAVFrame = NULL;
    pSwsContext = NULL;

    videoWidth = 0;
    videoHeight = 0;
    videoStreamIndex = -1;

    m_stopped = 0;
}
```

该类的 Init()函数主要用来初始化 FFmpeg 的网络库并打开网络流，然后分析流信息，再根据音视频流参数找到对应的解码器，同时准备好 SwsContext 进行颜色空间转换等，主要代码如下：

```
//chapter5/FFmpegMonitor_Libx264/avffmpegnetcamera.cpp
//初始化
bool AVFFmpegNetCamera::Init(){
```

```
//初始化网络流格式,使用 RTSP 网络流时必须先执行
avformat_network_init();

//申请一个 AVFormatContext 结构的内存并进行简单初始化
pAVFormatContext = avformat_alloc_context();
pAVFrame = av_frame_alloc();

//打开视频流
int result = avformat_open_input(
          &pAVFormatContext, url.toStdString().c_str(), NULL, NULL);
if (result < 0){
   qDebug() << "打开视频流失败 ";
   return false;
}

//获取视频流信息
result = avformat_find_stream_info(pAVFormatContext, NULL);
if (result < 0){
   qDebug() << "获取视频流信息失败 ";
   return false;
}

//获取视频流索引
videoStreamIndex = -1;
for (int i = 0; i < pAVFormatContext->nb_streams; i++) {
   if (pAVFormatContext->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO) {
      videoStreamIndex = i;
      break;
   }
}

if (videoStreamIndex == -1){
   qDebug() << "获取视频流索引失败 ";
   return false;
}

//分析编解码器的详细信息
pAVCodecContext = pAVFormatContext->streams[videoStreamIndex]->codec;
videoWidth = pAVCodecContext->width;
videoHeight = pAVCodecContext->height;

//分配 AVPicture,格式为 RGB24
avpicture_alloc(&pAVPicture,AV_PIX_FMT_RGB24,videoWidth,videoHeight);

//解码器指针
AVCodec *pAVCodec = NULL;

//获取视频流解码器
```

```
    pAVCodec = avcodec_find_decoder(pAVCodecContext->codec_id);//eg: h264,h265

    //颜色空间转换,结构体为 SwsContext
    //相关函数包括 sws_getContext、sws_freeContext、sws_scale(...)
    pSwsContext = sws_getContext(
              videoWidth,videoHeight,AV_PIX_FMT_YUV420P,
              videoWidth,videoHeight,AV_PIX_FMT_RGB24,
              SWS_BICUBIC,0,0,0);

    //打开对应解码器
    result = avcodec_open2(pAVCodecContext,pAVCodec,NULL);
    if (result<0){
        qDebug()<<"打开解码器失败 ";
        return false;
    }
    qDebug()<<"初始化视频流成功 ";
    return true;

    return true;
}
```

该类的 Deinit()函数主要用来释放相关的内存,进行反初始化,主要代码如下:

```
//chapter5/FFmpegMonitor_Libx264/avffmpegnetcamera.cpp
void AVFFmpegNetCamera::Deinit(){
    avformat_network_deinit();

    av_frame_free(&pAVFrame);
    avformat_free_context(pAVFormatContext);
    avpicture_free(&pAVPicture);
    sws_freeContext(pSwsContext);

}
```

该类的 Play()函数的主要功能是读取网络摄像头,然后解封装、解码,最后实现视频预览,主要代码如下:

```
//chapter5/FFmpegMonitor_Libx264/avffmpegnetcamera.cpp
//读取网络摄像头,解封装、解码,预览
void AVFFmpegNetCamera::Play(){
    //1. av_read_frame : demuxing
    //2. avcodec_send_packet, avcodec_receive_frame: decoding
    //3. sws_scale: yuv420p -->rgb24,
    //4. QImage, emit signal
    //
    //准备 libx264 和 libx265 编码器
    //判断:如果是 RTSP 直播摄像头,则录制;如果是回放,则不用录制
    T3FFmpegH2645Encoder2 objFmpgH2645Encoder;
```

```
    if( ! this->m_bIsPlayingbacked ){
        //文件名
        QString strFileName = QDateTime::currentDateTime().toString("yyyy-MM-dd-hhmmss")
                + "-channel" + QString::number( m_channelIndex) + ".h264" ;
        objFmpgH2645Encoder.setOutfile( strFileName );
        objFmpgH2645Encoder.setVideoWidth(videoWidth);
        objFmpgH2645Encoder.setVideoHeight(videoHeight);
        objFmpgH2645Encoder.initLibx2645();
    }

    int ret = -1;
    while(1){
        if(m_stopped){
            break;
        }

        if (av_read_frame(pAVFormatContext, &pAVPacket) >= 0){
            if(pAVPacket.stream_index == videoStreamIndex){
                qDebug()<<"开始解码 "<< QDateTime::currentDateTime().toString("yyyy-MM-dd
HH:mm:ss ");
                //avcodec_decode_video2(pAVCodecContext, pAVFrame, &frameFinished, &pAVPacket);


                /* 将音视频压缩包发给解码器 */
                ret = avcodec_send_packet(pAVCodecContext, &pAVPacket);
                if (ret < 0)
                {
                    fprintf(stderr, "Error submitting the packet to the decoder\n");
                    continue;
                }

                ret = avcodec_receive_frame(pAVCodecContext, pAVFrame);
                if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
                    continue;

                //编码视频帧,格式为 FrameYUV420p
                if( ! this->m_bIsPlayingbacked ){
                    objFmpgH2645Encoder.encodeLibx2645OneFrame( pAVFrame );
                }


                if (ret >= 0){//结束
                    mutex.lock();
                    sws_scale(pSwsContext,(const uint8_t* const *)pAVFrame->data,pAVFrame->
linesize,0,videoHeight,pAVPicture.data,pAVPicture.linesize);
                    //发送获取一帧图像信号
                     QImage image(pAVPicture.data[0],videoWidth,videoHeight,QImage::Format_
RGB888);
                    emit GetImage(image);
```

```
                mutex.unlock();
            }
            //-re: 帧率
            QThread::msleep(33);
        }
    }

    //释放资源,否则内存会一直上升
    av_free_packet(&pAVPacket);
}

qDebug() << "OK to exit, bye...\n" ;
if( ! this->m_bIsPlayingbacked ){
    objFmpgH2645Encoder.quitLibx2645();
}
return;
}
```

5.3　FFmpeg 实现 H.264/H.265 编码的 C++类封装

在预览网络视频流的同时,已经实现了视频的编码录制工作,核心代码如下:

```
//chapter5/FFmpegIPCMonitor/t3ffmpegh2645encoder.cpp
    //判断:如果是 RTSP 直播摄像头,则录制;如果是回放,则不用录制
    T3FFmpegH2645Encoder2 objFmpgH2645Encoder;
    if( ! this->m_bIsPlayingbacked ){
        //filename: QDateTime::currentDateTime().toString("yyyy-MM-dd=hhmmss");
         QString strFileName = QDateTime::currentDateTime().toString("yyyy-MM-dd=
hhmmss")
                  + "-channel" + QString::number( m_channelIndex) + ".h264" ;
        objFmpgH2645Encoder.setOutfile( strFileName );
        objFmpgH2645Encoder.setVideoWidth(videoWidth);
        objFmpgH2645Encoder.setVideoHeight(videoHeight);
        objFmpgH2645Encoder.initLibx2645();
    }
```

可以看出,笔者专门封装了一个 C++类(T3FFmpegH2645Encoder2),用来调用 libx264 或 libx265 对视频帧进行编码并存储,该类的头文件如下;

```
//chapter5/FFmpegIPCMonitor/t3ffmpegh2645encoder.h
#ifndef T3FFMPEGH2645ENCODER2_H
#define T3FFMPEGH2645ENCODER2_H

#include <QObject>
extern "C"{
```

```
    #include <libavformat/avformat.h>
    #include <libavcodec/avcodec.h>
    #include <libavutil/imgutils.h>
    #include <libavutil/opt.h>
};
#include <iostream>
using namespace std;

class T3FFmpegH2645Encoder2 : public QObject
{
    Q_OBJECT
public:
    explicit T3FFmpegH2645Encoder2(QObject * parent = nullptr);

    //初始化、编码、退出
    int initLibx2645();
    int quitLibx2645();
    int encodeLibx2645OneFrame(AVFrame * pFrameYUV420p); //输入参数
    void setOutfile(QString strfilename){this->m_outfile = strfilename;}
    void setVideoWidth(int ww){in_w = ww;}
    void setVideoHeight(int hh) {in_h = hh; }

private:
    int _encode(AVCodecContext * avCodecCtx,
                AVPacket * pack,
                AVFrame * frame,
                FILE * fp = NULL);

private:
    AVFormatContext * pFormatCtx;
    AVOutputFormat * pOutputFmt;
    AVStream * pStream;
    AVCodecContext * pCodecCtx;
    AVCodec * pCodec;
    AVPacket * pkt;
    AVFrame * pFrame;

    FILE * out_file;
    QString m_outfile;
    uint8_t * pFrameBuf;
    int m_frameIndex;
    int in_w, in_h;

signals:
public slots:
};

#endif //T3FFMPEGH2645ENCODER2_H
```

该类的构造函数主要用于对变量进行初始化，代码如下：

```
//chapter5/FFmpegIPCMonitor/t3ffmpegh2645encoder.cpp
T3FFmpegH2645Encoder2::T3FFmpegH2645Encoder2(QObject * parent) : QObject(parent)
{
   //init the member variables to zero
   pFormatCtx = NULL;
   pOutputFmt = NULL;
   pStream = NULL;
   pCodecCtx = NULL;
   pCodec = NULL;
   pkt = NULL;
   pFrame = NULL;

   out_file = NULL;
   pFrameBuf = NULL;
   m_frameIndex = 0;

   in_w = 640;
   in_h = 480;
}
```

该类的 initLibx2645()函数主要用于初始化 libx264 和 libx265 编码器，核心代码如下：

```
//chapter5/FFmpegIPCMonitor/t3ffmpegh2645encoder.cpp
int T3FFmpegH2645Encoder2::initLibx2645(){
   //1: 定义变量和结构体
   //2: 打开文件
   //3: 进入 FFmpeg 的流程

   //读取本地文件(二进制):yuvtest1 - 352x288 - yuv420p.yuv
   int nFrameNum = 100;
   char * pstrOutfile = this->m_outfile.toLocal8 位().data();
   out_file = fopen(this->m_outfile.toLocal8 位().data(), "wb");
   if (out_file == NULL) {
      printf("cannot create out file\n");
      return -1;
   }


   //准备编码器
   uint8_t * pFrameBuf = NULL;
   int frame_buf_size = 0;
   int y_size = 0;
   int nEncodedFrameCount = 0;
   pFormatCtx = avformat_alloc_context();
   pOutputFmt = av_guess_format(NULL, this->m_outfile.toLocal8 位().data(), NULL);
   pFormatCtx->oformat = pOutputFmt;
```

```
//除了以下方法,还可以使用 avcodec_find_encoder_by_name()获取 AVCodec
pCodec = avcodec_find_encoder(pOutputFmt->video_codec);
if (!pCodec) {
    //cannot find encoder
    return -1;
}
pCodecCtx = avcodec_alloc_context3(pCodec);
if (!pCodecCtx) {
    //如果失败,则返回-1
    return -1;
}
pkt = av_packet_alloc();
if (!pkt){
    return -1;
}
pCodecCtx->codec_id = pOutputFmt->video_codec;
pCodecCtx->codec_type = AVMEDIA_TYPE_VIDEO;
pCodecCtx->pix_fmt = AV_PIX_FMT_YUV420P;
pCodecCtx->width = in_w;
pCodecCtx->height = in_h;
pCodecCtx->time_base.num = 1;
pCodecCtx->time_base.den = 25;
pCodecCtx->bit_rate = 400000;
pCodecCtx->gop_size = 250;

//H.264 编码参数
//pCodecCtx->me_range = 16;
//pCodecCtx->max_qdiff = 4;
//pCodecCtx->qcompress = 0.6;
pCodecCtx->qmin = 10;
pCodecCtx->qmax = 51;
//可选参数,b 帧数量
pCodecCtx->max_b_frames = 3;

AVDictionary *param = NULL;
//H.264 编码器
if (pCodecCtx->codec_id == AV_CODEC_ID_H264) {
    //av_dict_set(&param, "profile", "main", 0);
    av_dict_set(&param, "preset", "slow", 0);
    av_dict_set(&param, "tune", "zerolatency", 0);
}
//H.265 编码器
if (pCodecCtx->codec_id == AV_CODEC_ID_HEVC) {//AV_CODEC_ID_HEVC,AV_CODEC_ID_H265
    av_dict_set(&param, "profile", "main", 0);
    //av_dict_set(&param, "preset", "ultrafast", 0);
    //note uncompatilable://av_dict_set(&param, "tune", "zero-latency", 0);
}

if (avcodec_open2(pCodecCtx, pCodec, &param) < 0) {
```

```
        //如果失败,则返回 - 1
        return - 1;
    }

    pFrame = av_frame_alloc();
    if (!pFrame) {
        fprintf(stderr, "Could not allocate the video frame data\n");
        return - 1;
    }
    pFrame - > format = pCodecCtx - > pix_fmt;
    pFrame - > width = pCodecCtx - > width;
    pFrame - > height = pCodecCtx - > height;

    int ret = av_frame_get_buffer(pFrame, 32);
    if (ret < 0) {
        fprintf(stderr, "Could not allocate the video frame data\n");
        return - 1;
    }

    frame_buf_size = av_image_get_buffer_size(pCodecCtx - > pix_fmt, pCodecCtx - > width,
pCodecCtx - > height, 1);
    pFrameBuf = (uint8_t * )av_malloc(frame_buf_size);
    av_image_fill_arrays(pFrame - > data, pFrame - > linesize,
        pFrameBuf, pCodecCtx - > pix_fmt, pCodecCtx - > width, pCodecCtx - > height, 1);

    y_size = pCodecCtx - > width * pCodecCtx - > height;

    return 0;
}
```

在读取摄像头时可以获得原始的视频帧,然后调用 encodeLibx2645OneFrame()函数对视频帧进行实时编码,并存储到本地文件中,代码如下:

```
//chapter5/FFmpegIPCMonitor/t3ffmpegh2645encoder.cpp
//摄像头的一帧数据,使用 libx264 或 libx265 进行编码
int T3FFmpegH2645Encoder2::encodeLibx2645OneFrame(AVFrame * pFrameYUV420p){

    //读取摄像头原始数据,将格式转换为 YUV420p
    int ret = av_frame_make_writable(pFrame);
    if (ret < 0) {
        return - 1;
    }

    //本地帧,YUV 分量
    pFrame - > data[0] = pFrameYUV420p - > data[0];//Y
    pFrame - > data[1] = pFrameYUV420p - > data[1];//U
```

```
    pFrame->data[2] = pFrameYUV420p->data[2];//V

    //显示时间戳
    pFrame->pts = m_frameIndex++;

    //编码细节
    _encode(pCodecCtx, pkt, pFrame, out_file);

    return 0;
}
```