[一、视频传输方案整体架构 2](#_Toc52120756)

[1.需求背景 2](#_Toc52120757)

[2.整体架构 2](#_Toc52120758)

[二、技术调研 2](#_Toc52120759)

[1.视频传输应用及现状 2](#_Toc52120760)

[2.视频编码 2](#_Toc52120761)

[2.1编码现状 2](#_Toc52120762)

[2.2编码算法 2](#_Toc52120763)

[2.2.1编码算法 3](#_Toc52120764)

[2.2.2对比 3](#_Toc52120765)

[2.3软硬编码 3](#_Toc52120766)

[2.3.1Mediacodec 3](#_Toc52120767)

[2.3.2ffmpeg 3](#_Toc52120768)

[2.3.2.1x264 3](#_Toc52120769)

[2.3.2.2h264 3](#_Toc52120770)

[3视频解码 3](#_Toc52120771)

[3.1编码现状 3](#_Toc52120772)

[3.2软硬解码 3](#_Toc52120773)

[3.2.1软解 3](#_Toc52120774)

[3.2.2硬解 3](#_Toc52120775)

[三、服务端实现方案 3](#_Toc52120776)

[1.服务端整体方案（架构） 3](#_Toc52120777)

[2.服务端—Android侧 3](#_Toc52120778)

[2.1实现概述 3](#_Toc52120779)

[2.2 渲染 3](#_Toc52120780)

[2.2.1 skia 4](#_Toc52120781)

[2.2.2 OpenGL 4](#_Toc52120782)

[3.服务端—协议侧 4](#_Toc52120783)

[3.1 采集 4](#_Toc52120784)

[3.1.1 read fb0 4](#_Toc52120785)

[3.1.2 VirtualDisplay 4](#_Toc52120786)

[3.1.3 优劣、效率 4](#_Toc52120787)

[3.2编码 4](#_Toc52120788)

[3.2.1ffmpeg+x264实现 4](#_Toc52120789)

[3.2.2编码关键参数设置 4](#_Toc52120790)

[四、客户端实现方案 4](#_Toc52120791)

[1.客户端整体方案（架构） 5](#_Toc52120792)

[2.解码 5](#_Toc52120793)

[2.1 解码实现 5](#_Toc52120794)

[2.2解码关键参数（若有） 5](#_Toc52120795)

[3. yuv转换 5](#_Toc52120796)

[4 客户端-显示 5](#_Toc52120797)

[4.1 16ms G-sync 5](#_Toc52120798)

[5 同步 5](#_Toc52120799)

[五、自适应策略 5](#_Toc52120800)

[1. 网络自适应 5](#_Toc52120801)

[2. 场景自适应 5](#_Toc52120802)

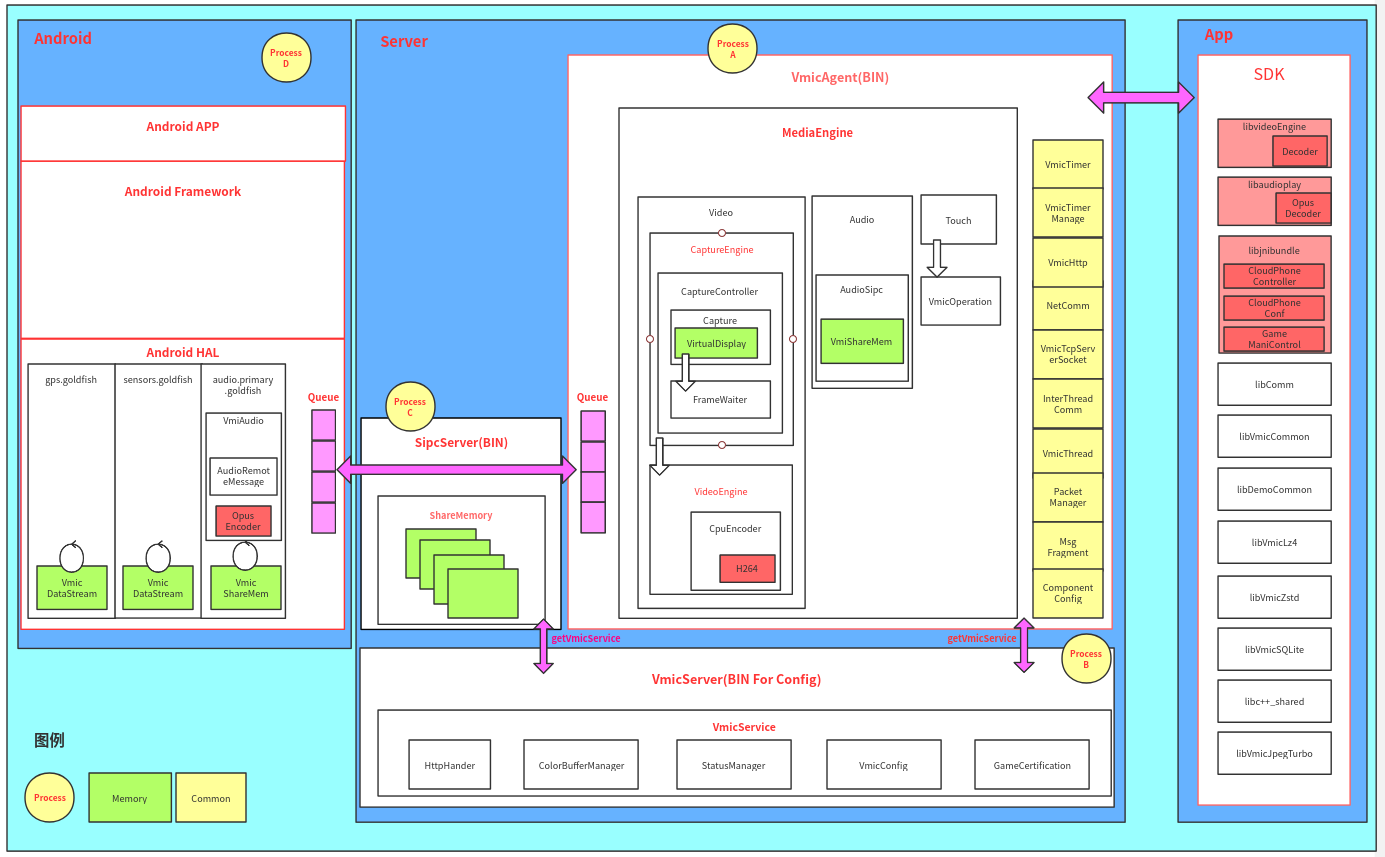
# 一、视频传输方案整体架构

## 1.需求背景

借助5G高带宽、低时延的特性，结合云端大容量、高性能的运算及存储资源，终端的运算和存储能力逐渐迁移到云端，本地和云端只需进行指令、视频流的交互，便可实现远程操控云端系统，畅想云端无限存储、计算资源的效果。云-端视频传输是云侧和端侧协同的关键技术，同时可为云手机、云游戏、云办公等场景赋能，提供更流程、更安全、体验更佳的5G云服务。

## 2.整体架构

服务端（安卓侧+协议侧）+客户端整体架构图，概述方案架构



如上图所示，本方案由服务端和客户端两个整体部分，而这两个部分由分别由自己的两个小模块组成，分别为:服务端Android侧、服务端协议侧、客户端协议侧、客户端Android侧。

服务端Android侧：用来模拟、采集或者注入相应的硬件数据，如扬声器数据的采集，各种sensor的模拟或者麦克风的注入，借由这样的模式来使得整个云手机的组件或者功能的完整。

服务端协议侧：这个部分整个方案的核心之一，它是集数据封装，传送，解析，编码，解码等功能以达到Android数据到客户端的完美迁移。

客户端协议侧：这个部分是另一个核心。与服务端协议侧一样，具有相同类似的功能。它的作用是把来自服务端协议侧的数据按照Android的方式更好更优的为客户端的展现提供数据的预处理。

客户端Android侧：此部分与用户交互最为紧密，它的目的是把客户端协议侧的数据翻译出来与用户交互相应。

# 二、技术调研

## 1.视频传输应用及现状

视频编码指通过压缩技术，将原始视频格式的文件转换成另一种视频格式文件的方式，以去除空间、时间维度的冗余，便于储存传输。随着计算机、电子技术的迅猛发展以及人们对高分辨率、高质量图像的迫切需求，1080P、2K甚至4K视频日渐成为主流趋势，但未经处理的原始图像数据量是巨大的，以分辨率为1080P，帧率为30fps，8位色的视频画面为例，播放1小时消耗的数据量为：1920\*1080\*3\*8\*30\*60\*60=626GB。如此庞大的数据量对存储、传输、运算都是极大的负担，同时也是制约多媒体技术发展的瓶颈。为此在不影响图像客观信息表达的前提下，需对原始视频做编码压缩处理降低冗余信息。

图像的各像素之间存在着很强的相关性，利用这些相关性，一部分像素的数据可以由另一部分像素的数据推导出来，据此视频中存在大量数据冗余，如空间冗余、时间冗余、结构冗余、信息熵冗余等。除了数据冗余外，还存在视觉冗余。人眼的一些特性比如亮度辨别阈值，视觉阈值，对亮度和色度的敏感度不同，使得在编码的时候去除阈值之外的数据，也不会被察觉出来。可以利用人眼的视觉特性，以一定的客观失真换取数据压缩。视频编码极大降低了上述两种冗余，使得视频数据量得以极大的压缩，有利于传输和存储。

## 2.视频编码

### 2.1编码现状

### 2.2编码算法

#### 2.2.1编码算法

广泛应用的视频编码算法包括：MPEG-4、H.264、H.265等。

1)MPEG-4

MPEG-4在MPEG-1、MPEG-2基础上发展而来，2000年成为国际标准，适用于低传输速率应用的方案。MPEG-4标准是基于对象和内容的编码方式，与传统的图像帧编码不同，只对相邻帧间的差异进行编码传输，从而提高了压缩比，使其在较低的码率下获得较好的效果。

2)H.264

H.264是ITU-T国际电联与ISO国际标准化组织联合制定的视频编解码技术标准。H.264的优势是具有很高的数据压缩比率，在同等图像质量下，H264的数据压缩率比MPEG-4高1.5-2倍。H.264获得优越性能的代价是计算复杂度增加，对硬件也有一定要求。随着电子器件技术的飞速发展，高性能的DSP芯片被广泛应用在视频编解码中，为复杂的编码算法提供有力的硬件支持。目前，H.264在视频监控、直播、视频点播等场景下具有广泛应用。

3)H.265

H.265是继H.264之后所制定的高压缩率的视频压缩格式，与H.264有着相类似的算法架构，对一些相关技术加以改进而大幅提高视频质量。旨在有限带宽下传输更高质量的网络视频，仅需原先的一半带宽即可播放相同质量的视频，但计算复杂度提升到原来的3倍，编码对硬件资源的消耗以及编码速度相应增大。随着硬件设备更好、带宽更高，用户开始对视频分辨率的要求更高，视频编码对压缩率的需求也越来越强烈，H.265标准同时也支持4K和8K超高清视频，在高性能硬件的支持下H.265将会成为未来的主流视频编解码标准。

#### 2.2.2对比

|  |  |  |  |
| --- | --- | --- | --- |
| **编码算法** | **编码效率** | **编码时间** | **硬件支持** |
| **H.264** | 较低 | 较快 | 广泛支持 |
| **H.265** | 较高 | 较慢 | 部分支持 |

同等编码质量下,H.265比H.264可节省30%左右的码率，但计算复杂度是H.264的3倍之多，编解码时间也相应更耗时，且目前硬件对H.265的解码支持还不够全面。视频传输方案的应用场景对视频画质和传输时延都有较高要求，综合考虑编码效率、时间效率、资源消耗、硬件支持等方面的因素，H.264是目前最优的选择

### 2.3软硬编码

视频编码的实现分为软编码和硬编码两种。软编码使用CPU进行编码，实现直接、简单，参数调整方便，升级易，但CPU负载重，性能一般较硬编码低。硬编码借助GPU、专用的DSP、FPGA、ASIC芯片等，支持的数据宽度更大，运算更复杂，效率更高。但硬编码对硬件有较强的依赖，通用性较差。

#### 2.3.1Mediacodec

MediaCodec类是Android提供的用于访问低层多媒体编/解码器接口，它是Android低层多媒体架构的一部分，可实现H.264等视频编解码。MediaCodec支持硬件编解码以及软件编解码,硬件编解码直接调用手机平台硬件的编解码能力，效率很高，且CPU占用大幅降低。但一定程度上依赖于设备，由于MediaCodec的硬编解码由厂商所提供，同时安卓设备的硬件相互之间差异很大，所以在硬解码实现上会有所差别，可能导致不同设备的兼容问题。

#### 2.3.2ffmpeg

通常采用FFmpeg +x264，FFmpeg 是媒体处理工具的集合，包含了很多的媒体文件处理工具，例如媒体文件格式解析工具、编解码器等，同时FFmpeg 支持一些外部的库，例如x264等。X264是一个开源的H.264/MPEG-4 AVC视频编码函数库，基本支持h264的所有特性，通过合理配置各种参数能够得到较好的压缩率和编码速度。

Mediacodec的硬编解码优势在于速度较快，无需引入外部库便可实现编解码。但特性支持有限， Mediacodec仅支持码率控制、码率、帧率、I帧间隔等10余个指标的调整，且由于底层编码能力的限制调整的参数也不一定都会生效。量化、运动预测、帧内分片等相关参数无法调试，可配置参数远远少于软编码，压缩率一般偏低，且服务端底层的硬件支持与手机设备差异较大，存在一定不可控性。FFMPEG +x264软编码压缩率比较高，支持H264特性较多，包括码率控制、帧预测、运动估计、颜色控制等100+个相关参数的自由选择，灵活可控。为达到清晰、流畅的用户视觉体验，在视频传输方案的调优目标为画质优先，保证画质前提下视频传输的时延越低越优。综上，为达到更好的编码效果，服务端选择FFMPEG+x264对采集到的视频画面进行软编码，同时为了优化ffmpeg编码效率引入NEON。NEON通过向量化进行速度优化，可使复杂视频编解码器的效率提升60-150%，CPU占有率降低2%-7%，弥补了软编码效率低的缺陷。

##### 2.3.2.1 x264编码器

为免费开放原始码的 H.264 / MPEG-4 AVC 编码器，是最好的有损视频编码器之一。凭借开源、免费的优势，目前 x264 已经成为各个压片组的唯一 h.264 编码器，大家在网络上能下载的各类 h.264 视频绝大部分都是由 x264 压制的。

由于 x264 是开源软件，因此在这几年里一直都有不同的编译版本和修改版或者说分支，其中之一就是利用 OpenCL 这个 API 尝试实现在 GPU 这类处理器上实现海量线程并行加速。对 x264 使用 OpenCL 进行硬件加速的尝试有几个项目，而在“官方”这边，目前主要是针对 x264 中的 lookahead 操作模块做硬件加速。

Lookahead 操作属于 x264 中的一个复杂模块，作用是对主编码器模块尚未分析的帧进行编码成本估算。例如自适应 B 帧定齤位、显式权重预测、受限缓存码率控制（buffer-constrained ratecontrol）的位元分配等等，都会用到 lookahead 操作的结果。基于速度考量，x264 的 lookahead 是在一半分辨率上操作的，采用的是简化版运动向量预测和帧间分析。

按照 Jason Garrett-Glaser（x264 首席开发员）和 Steve Borho（Mulricore Ware 方案架构师）在 AMD Fusion Developer Summit 2012 上的介绍，当时的 x264 OpenCL 在 Tahita（RADEON HD 7970）上达到两倍性能，而画面品质和 C 语言版 x264 相当。

虽然 OpenCL x264 有一定的性能提升，但是 x264“官方”认为这个东西目前还有很多地方未完善。即使是同一个 GPU 厂商提供的 OpenCL 驱动，也会出现不同版本有差异性，从而导致 OpenCL 程序出现无法执行的问题，因此并未将这个“部件”加入到正式版中。

## 3视频解码

客户端依照H.264规范进行解码，将H.264流转化为YUV图像序列。客户端的视频解码有软解码和硬解码两种实现。

### 3.1编码现状

### 3.2软硬解码

#### 3.2.1软解

软解码由CPU执行，对视频流格式及算法优化具有普适性，但CPU占有率高且效率相对较低。FFmpeg可实现视频的软解码，FFmpeg 提供了很多编解码器，且它的媒体操作也很丰富，可以支持非常多的媒体类型，同时很多的处理功能也由FFmpeg 提供。

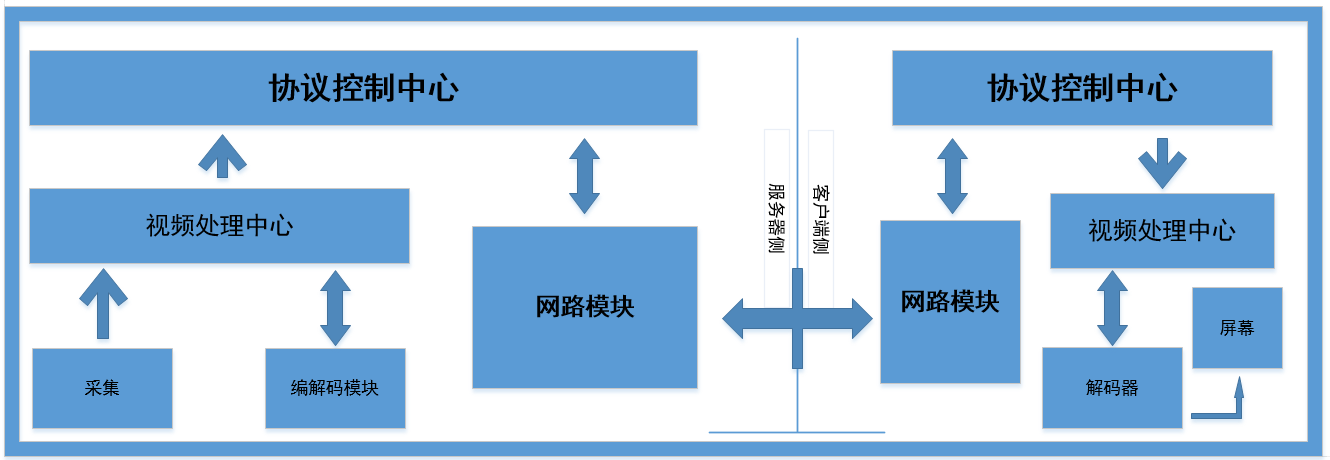
#### 3.2.2硬解

硬解码是通过GPU等具备图形处理功能的芯片完成解码过程，GPU 内部固定了一些特定的算法可以并行处理大量图形图像数据，速度快于软解码。同时硬解码释放了CPU的运算压力，功耗也较小。对统一视频流进行解码，硬解码的CPU占有率约为软解码的1/10，耗电量约为软解码的1/2。Mediacodec可实现硬件加速的视频解码，但灵活性较差，硬件支持的算法固定，固件升级才可能带来算法升级，Mediacodec解码架构的一些特性限制了视频解码的效率和质量，如考虑到解码B帧是需先解码前后帧，为避免播放被短暂阻塞实现架构上为main或high profile提供了7个缓冲区，如果帧速率为30 FPS，则对应210ms的额外延迟，极大影响了解码时间效率。

软硬解码有各自的优势，也各有缺陷。目前FFmpeg已实现对MediaCodec的支持，因此融合FFmpeg和Mediacodec的优势，采用FFmpeg +Mediacodec结合的方式，一方面采用FFmpeg完备的解码架构，另一方面利用Mediacodec的编码器直接调用底层硬件的编解码能力，保证解码质量和解码效率的同时，最大限度的降低CPU占有率、功耗等资源消耗。

# 三、服务端实现方案

## 1.服务端整体方案（架构）

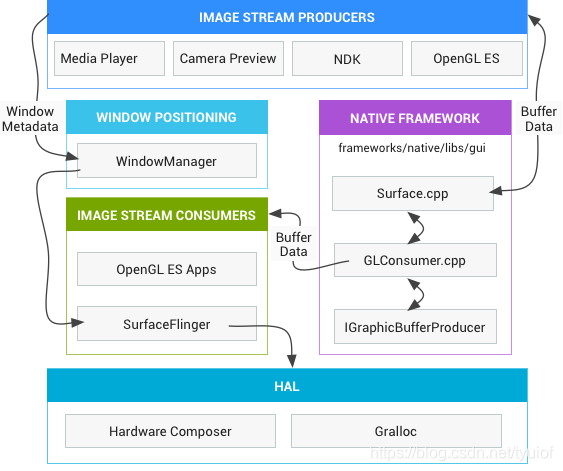


## 2.服务端—Android侧

### 2.1实现概述

如前面图所示，整个视频产生的起点位于服务器采集的这一个部分。因为服务端这边是不涉及物理屏幕的，所以这边所做的事情就是把Android底层渲染好的数据采集到，交给视频处理中心做编码，经过编码后做数据打包，发送到客户端。这是整个服务端的一个大致流程，现在先把服务端分解开来可分为：图像数据产生（Android侧），图像数据采集（协议侧）。

图像数据产生（Android侧）如下图：



图像的生产者主要有MediaPlayer，CameraPrevier，NDK，OpenGl ES。MediaPlayer和Camera Previer是通过直接读取图像源来生成图像数据，NDK（Skia），OpenGL ES是通过自身的绘制能力生产的图像数据。那么在他们其中Skia和OpenGL是Android中最重要的图像生产者，也就是Android的笔杆子。

### 2.2 图像生产者

#### 2.2.1 skia

Skia是谷歌开源的一款跨平台的2D图形引擎，目前谷歌的Chrome浏览器、Android、Flutter、以及火狐浏览器、火狐操作系统和其它许多产品都使用它作为图形引擎，它作为Android系统第三方软件，放在external/skia/ 目录下。虽然Android从4.0开始默认开启了硬件加速，但不代表Skia的作用就不大了，其实Skia在Android中的地位是越来越重要了，从Android 8开始，可以选择使用Skia进行硬件加速，Android 9开始就默认使用Skia来进行硬件加速。Skia的硬件加速主要是通过 copybit 模块调用OpenGL或者SKia来实现分。

#### 2.2.2 OpenGL

OpenGL（英语：Open Graphics Library，译名：开放图形库或者“开放式图形库”）另一个Android的笔杆子是用于渲染2D、3D矢量图形的跨语言、跨平台的应用程序编程接口（API）。这个接口由近350个不同的函数调用组成，用来绘制从简单的图形比特到复杂的三维景象。OpenGL常用于CAD、虚拟现实、科学可视化程序和电子游戏开发。

OpenGL的高效实现（利用了图形加速硬件）存在于Windows， UNIX平台和Mac OS。这些实现一般由显示设备厂商提供，而且非常依赖于该厂商提供的硬件。开放源代码库Mesa是一个纯基于软件的图形API，它的代码兼容于OpenGL。但是，由于许可证的原因，它只声称是一个“非常相似”的API。

## 3.服务端—协议侧

### 3.1 采集

#### 3.1.1 read fb0

FrameBuffer 文件是 Linux (Android是基于Linux的) 对显示设备的一种抽象设备，相当于显存。Android 的 SurfaceFlinger 想更新屏幕的时候，就会把相应的改变写入到FrameBuffer里。

Android平台上，FrameBuffer 文件的绝对路径一般是： /dev/graphics/fb0 。所以，如果想截图，其中一种方法就是把FrameBuffer里头的图像数据取出来，转换成bitmap，然后存储起来或者给ImageView来显示出来。

#### 3.1.2 VirtualDisplay

在目前的 Android 中，支持多种屏幕（Display，后文提到的 Display 都是指如下的各种屏幕）类型：内置的主屏幕、通过 HDMI 连接的外接屏幕、虚拟屏幕（Virtual Display）

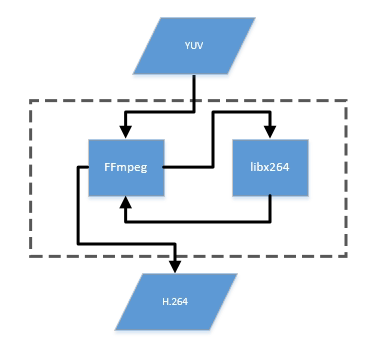
前两种都是有具体的物理屏幕设备的，而与之相反的 Virtual Display 则没有，是由 SurfaceFlinger 模拟出来的。

在创建虚拟屏幕VirtualDisplay前，需要先建立与Surface Flinger连接。然后通过绑定与BufferQueue关联的生产者和消费者,BufferQueue的生产者来自Surface(通过SurfaceFlinger管理)，

消费者为刚创建的虚拟屏幕Virtual Display，最后通过Virtual Display对终端屏幕进行实时屏幕截图。

### 3.2编码

#### 3.2.1ffmpeg+x264实现



编码流程如图所示，把采集好经过YUV转换后的数据注入到FFmpeg里面，然后在FFmpeg的回调中的到编码好的H.264数据流。X264扮演了一个具体施工的角色，负责编码的具体流程，而FFmpeg则为一个宏观的调用者，它首先配置好x264的编码器，等待数据到来，一旦数据产生则直接传给x264 ，接着x264按照先前的参数设置编码好视频数据，向FFmpeg吐出H.264的数据流。

#### 3.2.2编码关键参数设置

##### tun

针对特定类型的影片微调参数设定值，以获得更好的品质或压缩率。

Film (电影): 胶片电影。

Animation (动画): 例如卡通/日本动画。

Grain (胶片颗粒): 颗粒感很重的影片。

Still Image (静止影像): 例如幻灯片效果的影片。

PSNR: 优化 PSNR 值。

SSIM: 优化 SSIM 值。

Fast Decode (快速解码): 用于低性能播放设备。 (※ 压缩率极低)

Zero Latency (零延迟): 处里时间低延迟。主要用于直播等。 (※ 压缩率极低)

##### Preset

Preset 是一个选项集合，这设定一编码速度来决定压缩比。速度越慢则会得到更好的压缩编码效率 (画质-位元率比 或 画质-档案大小比)。也就是说，若你设定一个目标位元率或是档案大小，则越慢的 Preset 将会得到更好的输出品质。而对于设定一个恒定品质 (CRF) 或是恒定量化值 (QP)，你可以透过选择更慢的 Preset 来简单的节省位元率 (也就是得到更小的档案大小)。

一般而言是使用你所能忍受的最慢 Preset。目前 Presets 依速度递减排序是: ultrafast, superfast, veryfast, faster, fast, medium, slow, slower, veryslow, placebo。该默认 Preset 是 medium。忽略 placebo 因为它会比 veryslow 浪费更多时间而且效果差异太小。

而越慢的 Preset 其 Reference frame (ref) 值也越高，例如 veryslow preset 的 ref 为 16。由于通常不需这么高的 ref，你可以透过指定一个 Level 来限制 ref，或是手动指定一个合理的 ref 值 (1 ~ 6)。

preset -> tune -> 使用者设定选项 -> fast first pass -> profile -> level

当参数冲突时，后者会取代前者或是影响前者

preset slower 的参数为:

--b-adapt 2 --direct auto --me umh

--partitions all --rc-lookahead 60

--ref 8 --subme 9 --trellis 2

当设定:

‧ x264 Preset: Slower

‧ Reference frame (ref): 4

等同于:

--b-adapt 2 --direct auto --me umh

--partitions all --rc-lookahead 60

--ref 4 --subme 9 --trellis 2

输出 H.264 的 Reference frame (ref) 为 4

当设定

‧x264 Preset: Slower

‧Level: 3.2

输出 H.264 的 Reference frame (ref) 为 5

##### Profile

Profile 越高 压缩率 也越高，但编码复杂度也越高，对播放硬件性能要求较高

x264 支援输出的 H.264 Profile:

baseline, main, high, high10, high422, high444

没有特殊需求使用 High Profile 即可。

##### GOP

一个 keyframe (IDR frame) 到下一个 keyframe 的范围。

所以将 GOP 设为无限大虽然可以大幅增加压缩率，但这样就不能随意跳转到其他时间点播放/解码。

##### QP (VBR) 、 CRF (VBR)

指定输出品质

##### Single pass (ABR) 、 2-pass (VBR)

指定输出 Bitrate / 档案大小

通常只会使用两种速率控制: Constant Rate Factor (CRF) 或 2-Pass Bitrate。速率控制是指决定多少位元将被用于每个划格的方法。这将决定档案大小且品质如何分布。而 CRF 的位元率分配效果是最佳的，若无控制输出大小的需求则使用 CRF 即可。

##### Constant Quantizer (QP)

设定x264以恒定量化值（Constant Quantizer）模式来编码视讯。此方法可以让每一个划格获得相同品质 (设定值)。

设定值范围为 0–69，0 为最无损，默认值为 23。比较低的数值会得到比较高的品质，合理的范围 21 - 28。

##### Constant Ratefactor (CRF)

此方法允许编码器自动分配位元速率来试着达到一定输出品质。让每个划格得到它需要的位元数来保持所需的品质等级。CRF 会得到最佳的位元速率分配结果，缺点是你不能直接指定一个目标位元率或是档案大小。

设定值范围为 0–51.0，0 为最高品质，默认值为 23。比较低的数值会得到比较高的品质，合理的范围 18 - 28。考虑 18 为视觉无损 (或接近)，所以它看起来应该与输入相同 (或接近) 但它技术上不是无损。

##### Bitrate

自订 Bitrate 控制输出 Size，直接编码输出视频。由于没有先扫描过一次，x264 无法得知影片各时间点的复杂程度。所以 x264 只能在编码中不断猜后之后的复杂程度，这样将不能精准依各时间点的复杂程度等比例分配位元率，因为如果之前分配了过多流量，那么之后就必下修流量来使输出位元率平均值等于使用者设定值。

##### Bitrate, 2-Pass

此方法令编码器 1st-Pass 将会得知影片各个划格的复杂程度，2nd-Pass 时依照输入视讯各个划格的复杂程度等比例分配位元数输出，并且最终平均位元速率会等于设定值。2-Pass 位元速率分配结果会接近 CRF，缺点是耗费大量时间。

##### 1st-Pass CRF + 2-Pass Bitrate

此方法令编码器在 1st-Pass 时将会得知影片各个划格所需要的位元数，2nd-Pass 时依照各个划格所需要的位元数等比例分配使最终位元速率总平均值会等于设定值。2-Pass 位元速率分配结果会接近 CRF，缺点是耗费大量时间。

##### Maximun B-frame

当设定 B-frame 时，重复部分比较多/变化较少的 Frames 会被编码为 B-frame

此值限制 B-frame 的最大连续数量。

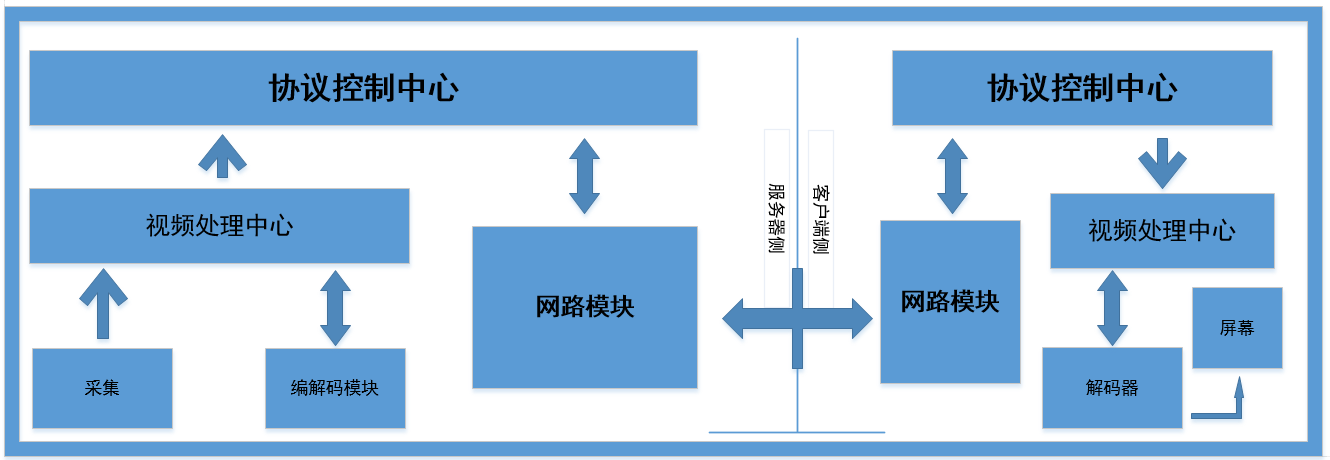
##### Reference frame (ref)

设定一个 P-frame 所能参考的 Frames 数量。

ref 会影响播放相容性

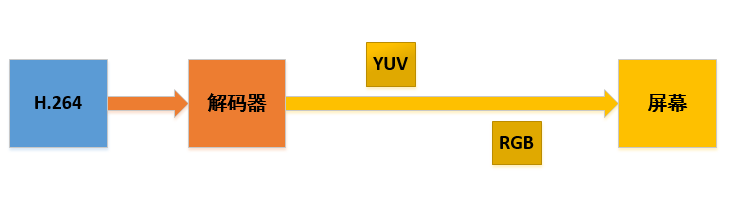
# 四、客户端实现方案

## 1.客户端整体方案（架构）



上图是客户端服务器的视频的整体架构图，那么客户端负责的就是右边的那一块，把H264的流最终显示在屏幕上。

具体说来如下：



是将来自服务器编码好的H264流通过解码器，解成YUV，在这一步还不能放在surface即Android的显示控件上显示，需要再继续转化成RGB才可以被显示控件接受，最后在手机屏幕上显示出来。

## 2.解码

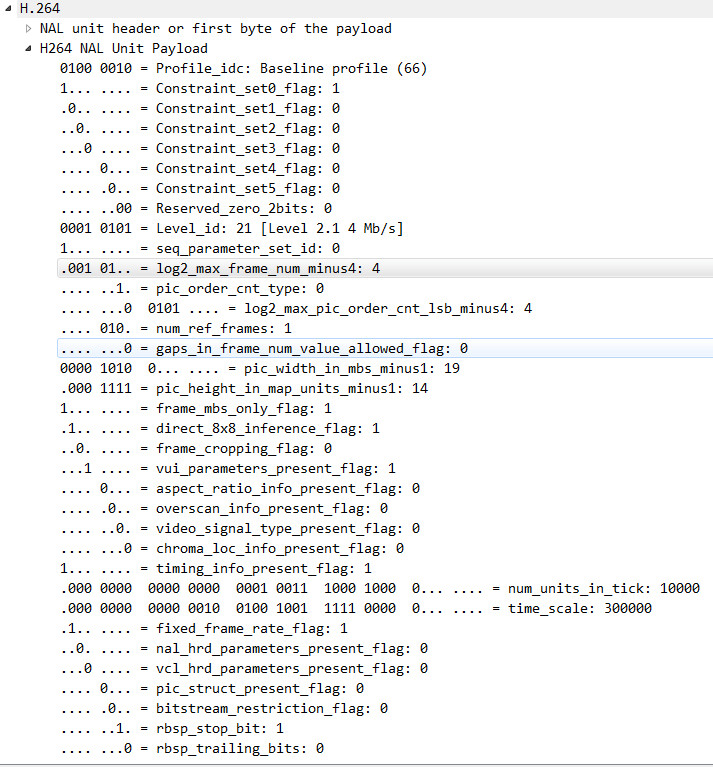
### 2.1 解码实现

解码的部分是H264的流送入到解码器，解码器返回解码好的YUV数据。在这个过程中，解码器并不会像编码器一般需要大量针对视屏的参数设置，因为有关视频信息的参数已经在H264流里面包含。

一般来说H264开始的时候会有SPS和PPS来表征当前H264所包含的视频通用信息，也就是想要把视频内容还原回来的附加参数。

#### SPS

一般的，SPS的数据格式类似下面的图（数据并不是一样的，格式大体一样）：



在H.264标准协议中规定了多种不同的NAL

Unit类型，其中类型7表示该NAL Unit内保存的数据为Sequence Paramater

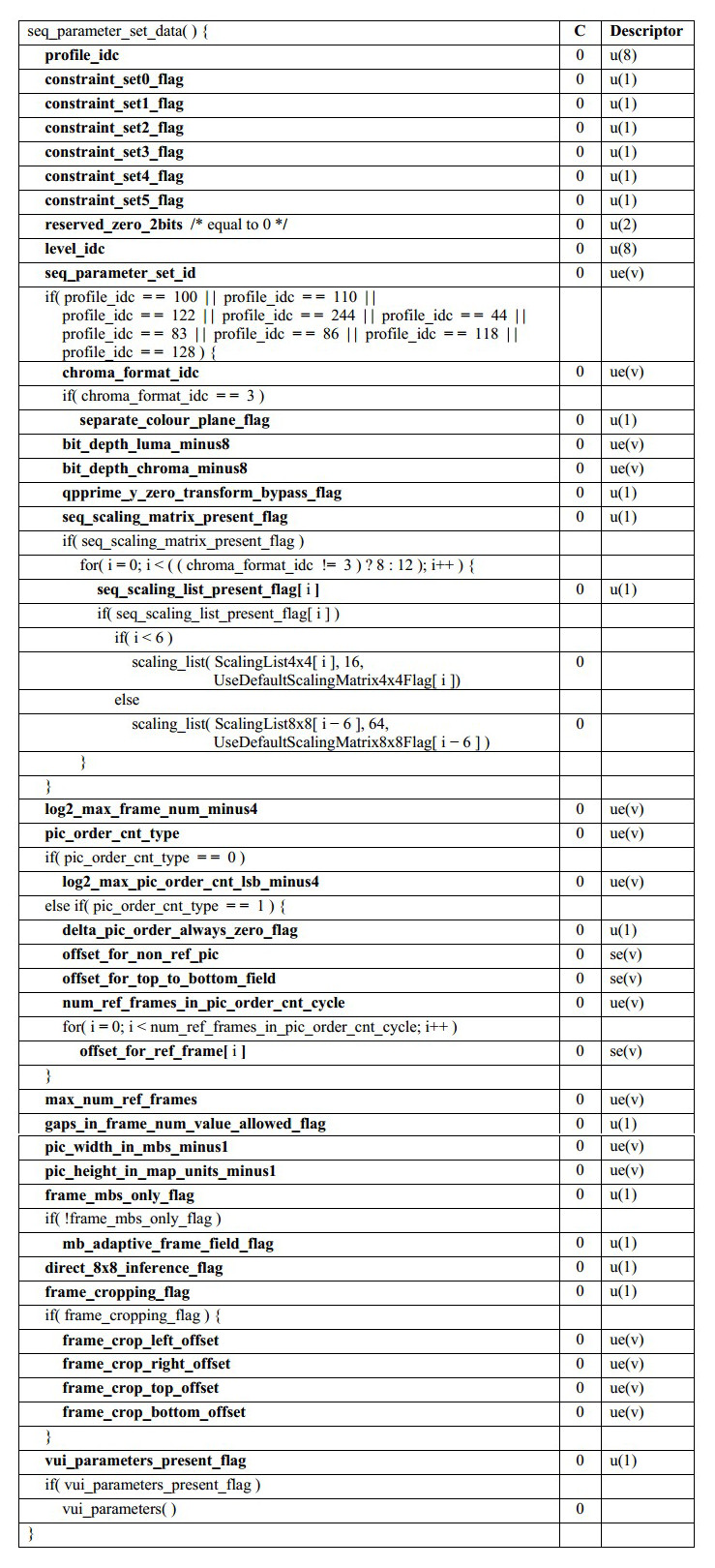
Set。在H.264的各种语法元素中，SPS中的信息至关重要。如果其中的数据丢失或出现错误，那么解码过程很可能会失败。SPS及后续将要讲述的图像参数集PPS在某些平台的视频处理框架（比如iOS的VideoToolBox等）还通常作为解码器实例的初始化信息使用。

SPS即Sequence Paramater Set，又称作序列参数集。SPS中保存了一组编码视频序列(Coded video sequence)的全局参数。所谓的编码视频序列即原始视频的一帧一帧的像素数据经过编码之后的结构组成的序列。而每一帧的编码后数据所依赖的参数保存于图像参数集中。一般情况SPS和PPS的NAL

Unit通常位于整个码流的起始位置。但在某些特殊情况下，在码流中间也可能出现这两种结构，主要原因可能为：

解码器需要在码流中间开始解码；编码器在编码的过程中改变了码流的参数（如图像分辨率等）；

在做视频播放器时，为了让后续的解码过程可以使用SPS中包含的参数，必须对其中的数据进行解析。其中H.264标准协议中规定的SPS格式位于文档的7.3.2.1.1部分，如下图所示：



其中的每一个语法元素及其含义如下：

(1) profile\_idc：

标识当前H.264码流的profile。H.264中定义了三种常用的档次profile：

基准档次：baseline profile;

主要档次：main profile;

扩展档次：extended profile;

在H.264的SPS中，第一个字节表示profile\_idc，根据profile\_idc的值可以确定码流符合哪一种档次。判断规律为：

profile\_idc = 66 → baseline profile;

profile\_idc = 77 → main profile;

profile\_idc = 88 → extended profile;

在新版的标准中，还包括了High、High 10、High 4:2:2、High 4:4:4、High 10 Intra、High

4:2:2 Intra、High 4:4:4 Intra、CAVLC 4:4:4 Intra等，每一种都由不同的profile\_idc表示。

另外，constraint\_set0\_flag ~ constraint\_set5\_flag是在编码的档次方面对码流增加的其他一些额外限制性条件。

在上图码流中，profile\_idc = 0x42 = 66，因此码流的档次为baseline profile。

(2) level\_idc

标识当前码流的Level。编码的Level定义了某种条件下的最大视频分辨率、最大视频帧率等参数，码流所遵从的level由level\_idc指定。

当前码流中，level\_idc = 0x1e = 30，因此码流的级别为3。

(3) seq\_parameter\_set\_id

表示当前的序列参数集的id。通过该id值，图像参数集pps可以引用其代表的sps中的参数。

(4) log2\_max\_frame\_num\_minus4

用于计算MaxFrameNum的值。计算公式为MaxFrameNum = 2^(log2\_max\_frame\_num\_minus4 +

4)。MaxFrameNum是frame\_num的上限值，frame\_num是图像序号的一种表示方法，在帧间编码中常用作一种参考帧标记的手段。

(5) pic\_order\_cnt\_type

表示解码picture order count(POC)的方法。POC是另一种计量图像序号的方式，与frame\_num有着不同的计算方法。该语法元素的取值为0、1或2。

(6) log2\_max\_pic\_order\_cnt\_lsb\_minus4

用于计算MaxPicOrderCntLsb的值，该值表示POC的上限。计算方法为MaxPicOrderCntLsb = 2^(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4)。

(7) max\_num\_ref\_frames

用于表示参考帧的最大数目。

(8) gaps\_in\_frame\_num\_value\_allowed\_flag

标识位，说明frame\_num中是否允许不连续的值。

(9) pic\_width\_in\_mbs\_minus1

用于计算图像的宽度。单位为宏块个数，因此图像的实际宽度为:

frame\_width = 16 × (pic\\_width\\_in\\_mbs\_minus1 + 1);

(10) pic\_height\_in\_map\_units\_minus1

使用PicHeightInMapUnits来度量视频中一帧图像的高度。PicHeightInMapUnits并非图像明确的以像素或宏块为单位的高度，而需要考虑该宏块是帧编码或场编码。PicHeightInMapUnits的计算方式为：

PicHeightInMapUnits = pic\\_height\\_in\\_map\\_units\\_minus1 + 1;

(11) frame\_mbs\_only\_flag

标识位，说明宏块的编码方式。当该标识位为0时，宏块可能为帧编码或场编码；该标识位为1时，所有宏块都采用帧编码。根据该标识位取值不同，PicHeightInMapUnits的含义也不同，为0时表示一场数据按宏块计算的高度，为1时表示一帧数据按宏块计算的高度。

按照宏块计算的图像实际高度FrameHeightInMbs的计算方法为：

FrameHeightInMbs = ( 2 − frame\_mbs\_only\_flag ) \* PicHeightInMapUnits

(12) mb\_adaptive\_frame\_field\_flag

标识位，说明是否采用了宏块级的帧场自适应编码。当该标识位为0时，不存在帧编码和场编码之间的切换；当标识位为1时，宏块可能在帧编码和场编码模式之间进行选择。

(13) direct\_8x8\_inference\_flag

标识位，用于B\_Skip、B\_Direct模式运动矢量的推导计算。

(14) frame\_cropping\_flag

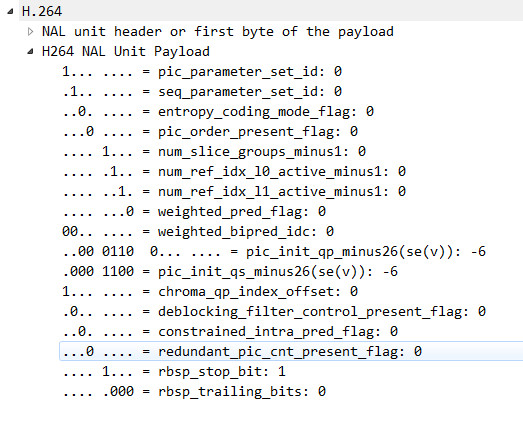
标识位，说明是否需要对输出的图像帧进行裁剪。

(15) vui\_parameters\_present\_flag

标识位，说明SPS中是否存在VUI信息。

### PPS

和SPS一样，PPS的数据格式类似下面的图（数据并不是一样的，格式大体一样）：

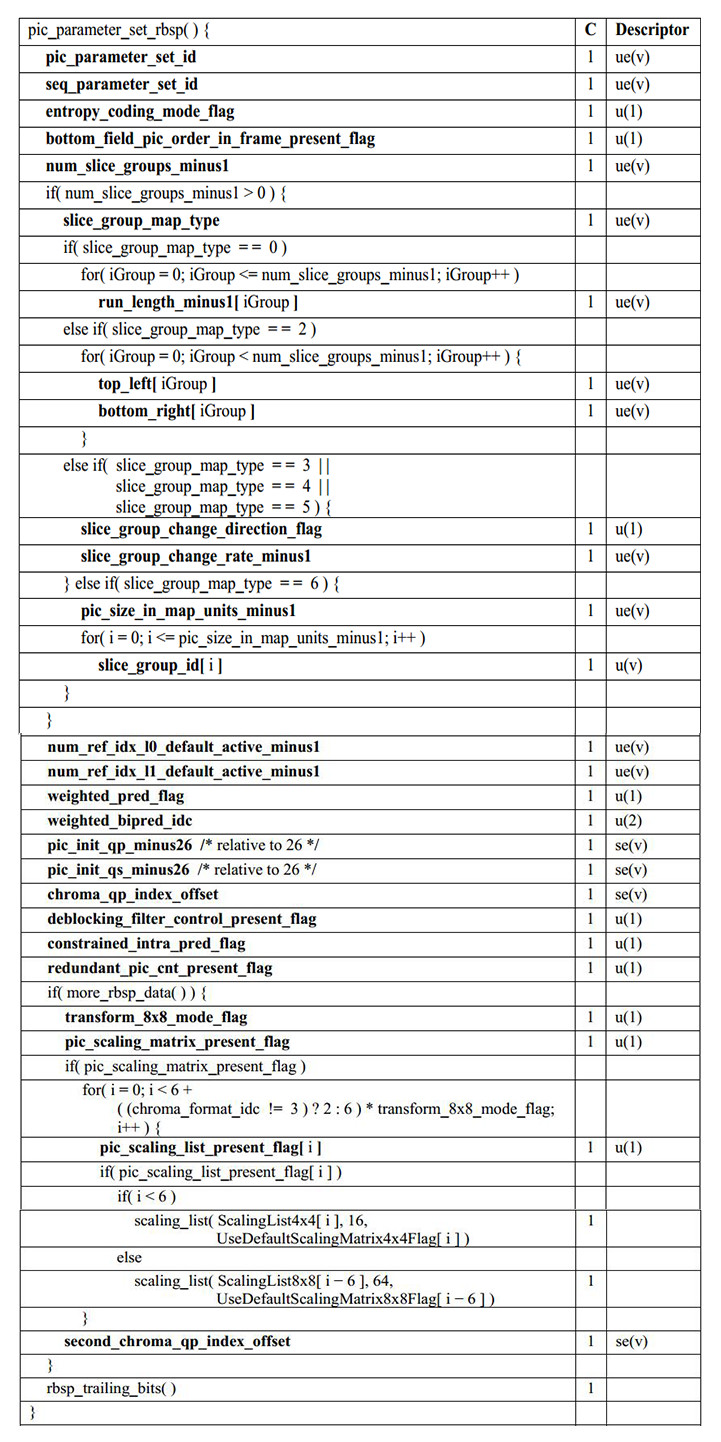


除了序列参数集SPS之外，H.264中另一重要的参数集合为图像参数集Picture Paramater

Set(PPS)。通常情况下，PPS类似于SPS，在H.264的裸码流中单独保存在一个NAL Unit中，只是PPS NAL

Unit的nal\_unit\_type值为8；而在封装格式中，PPS通常与SPS一起，保存在视频文件的文件头中。

在H.264的协议文档中，PPS的结构定义在7.3.2.2节中，具体的结构如下表所示：



其中的每一个语法元素及其含义如下：

(1) pic\_parameter\_set\_id

表示当前PPS的id。某个PPS在码流中会被相应的slice引用，slice引用PPS的方式就是在Slice header中保存PPS的id值。该值的取值范围为[0,255]。

(2) seq\_parameter\_set\_id

表示当前PPS所引用的激活的SPS的id。通过这种方式，PPS中也可以取到对应SPS中的参数。该值的取值范围为[0,31]。

(3) entropy\_coding\_mode\_flag

熵编码模式标识，该标识位表示码流中熵编码/解码选择的算法。对于部分语法元素，在不同的编码配置下，选择的熵编码方式不同。例如在一个宏块语法元素中，宏块类型mb\_type的语法元素描述符为“ue(v)| ae(v)”，在baseline profile等设置下采用指数哥伦布编码，在main profile等设置下采用CABAC编码。

标识位entropy\_coding\_mode\_flag的作用就是控制这种算法选择。当该值为0时，选择左边的算法，通常为指数哥伦布编码或者CAVLC；当该值为1时，选择右边的算法，通常为CABAC。

(4) bottom\_field\_pic\_order\_in\_frame\_present\_flag

标识位，用于表示另外条带头中的两个语法元素delta\_pic\_order\_cnt\_bottom和delta\_pic\_order\_cn是否存在的标识。这两个语法元素表示了某一帧的底场的POC的计算方法。

(5) num\_slice\_groups\_minus1

表示某一帧中slice group的个数。当该值为0时，一帧中所有的slice都属于一个slice group。slice group是一帧中宏块的组合方式。

(6)num\_ref\_idx\_l0\_default\_active\_minus1

表示当Slice Header中的num\_ref\_idx\_active\_override\_flag标识位为0时，P/SP/B

slice的语法元素num\_ref\_idx\_l0\_active\_minus1和num\_ref\_idx\_l1\_active\_minus1的默认值。

(7) weighted\_pred\_flag

标识位，表示在P/SP slice中是否开启加权预测。

(8) weighted\_bipred\_idc

表示在B Slice中加权预测的方法，取值范围为[0,2]。0表示默认加权预测，1表示显式加权预测，2表示隐式加权预测。

(9) pic\_init\_qp\_minus26和pic\_init\_qs\_minus26

表示初始的量化参数。实际的量化参数由该参数、slice header中的slice\_qp\_delta/slice\_qs\_delta计算得到。

(10) chroma\_qp\_index\_offset

用于计算色度分量的量化参数，取值范围为[-12,12]。

(11) deblocking\_filter\_control\_present\_flag

标识位，用于表示Slice header中是否存在用于去块滤波器控制的信息。当该标志位为1时，slice header中包含去块滤波相应的信息；当该标识位为0时，slice header中没有相应的信息。

(12) constrained\_intra\_pred\_flag

若该标识为1，表示I宏块在进行帧内预测时只能使用来自I和SI类型宏块的信息；若该标识位0，表示I宏块可以使用来自Inter类型宏块的信息。

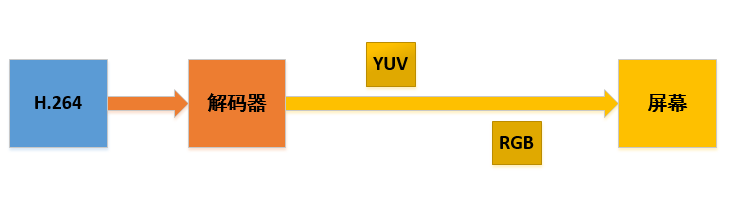
(13) redundant\_pic\_cnt\_present\_flag

标识位，用于表示Slice header中是否存在redundant\_pic\_cnt语法元素。当该标志位为1时，slice header中包含redundant\_pic\_cnt；当该标识位为0时，slice header中没有相应的信息。

## 3. yuv

YUV，是一种颜色编码方法。常使用在各个视频处理组件中。 YUV在对照片或视频编码时，考虑到人类的感知能力，允许降低色度的带宽。

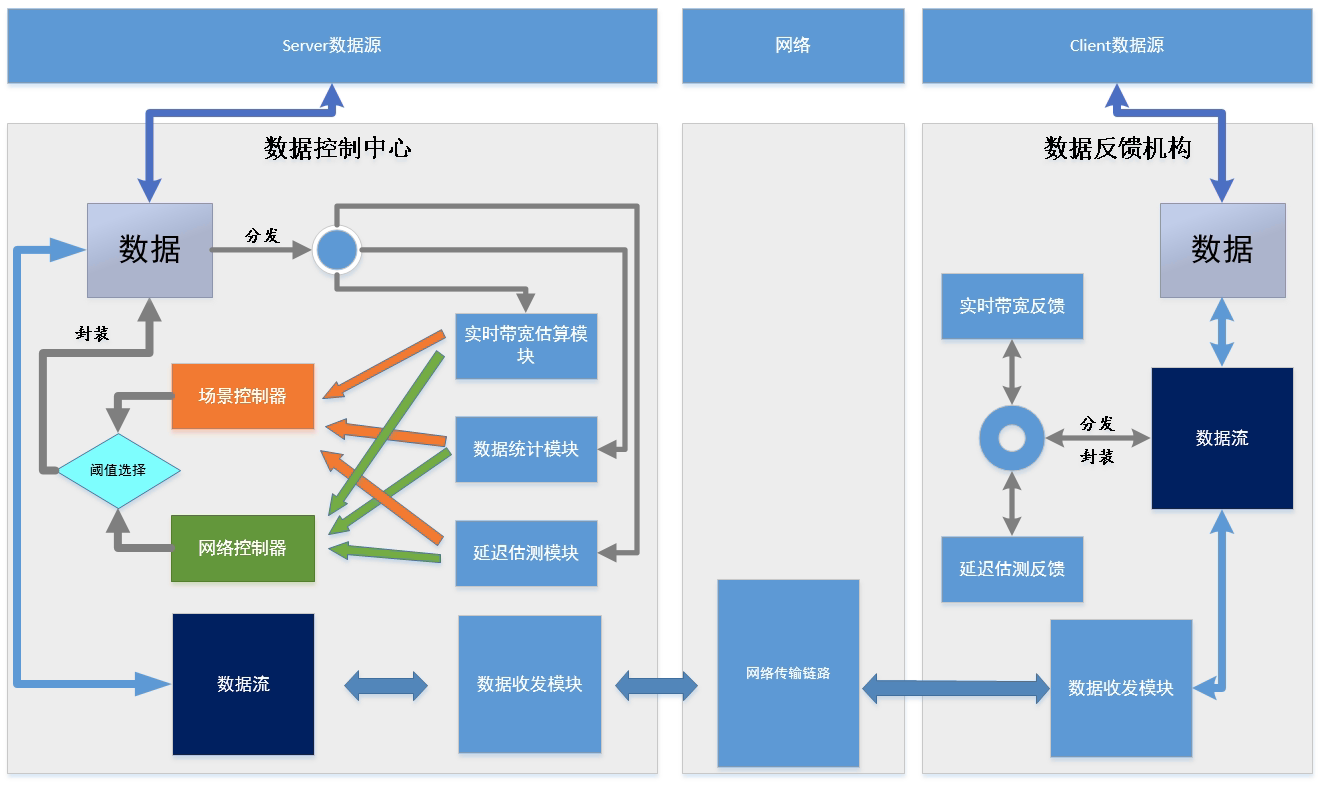
YUV是编译true-color颜色空间（color space）的种类，Y'UV, YUV, YCbCr，YPbPr等专有名词都可以称为YUV，彼此有重叠。“Y”表示明亮度（Luminance或Luma），也就是灰阶值，“U”和“V”表示的则是色度（Chrominance或Chroma），作用是描述影像色彩及饱和度，用于指定像素的颜色。



但是从上流程图中可知，手机屏幕是需要RGB的，所以在编解码过程中用YUV以加快速度优化编解码，而在显示过程中，就需要转化成RGB来显示。

# 五、自适应策略

整个云手机在运行过程中很多方面都会或深或浅的影响它的整体显示效果或者端到端的显示时延，下面针对不同的情况自适应的调整一些显示策略，在不降低或者牺牲少部分的显示效果来满足整体实时的交互性。由此指定了下面两个策略，大体都是通过减少带宽的方式来获取整体的流畅度，或者增大带宽的利用率。整体架构如下：



其中：实时带宽估计模块和实时带宽反馈互为系统，延迟估计模块和延迟估计反馈一起。数据统计为独立的模块。

实时带宽估计：

此模块作用是提供当前IP链路的实时传输带宽预测值，每秒更新一次。所谓实时传输带宽就是当前链路单位时间内的带宽总消耗。假如说当前在图中的数据区单位时间内产生了1M字节的数据，只要这些数据在下一个单位时间里面全部传送完毕，那么就说当前链路的实时传输带宽为1M字节，但是如果只传送了800K字节的数据，那么就只有800K字节。

延迟估测：

系统会数据流中的每一个数据包上做上标记，当延迟反馈端收到这个标记的时候则会就近把反馈标记加到即将要发送到服务器的数据中，以此来立即响应服务器的延迟估算标记。最后服务器收到反馈就会通过时差来最终确定传输延迟。

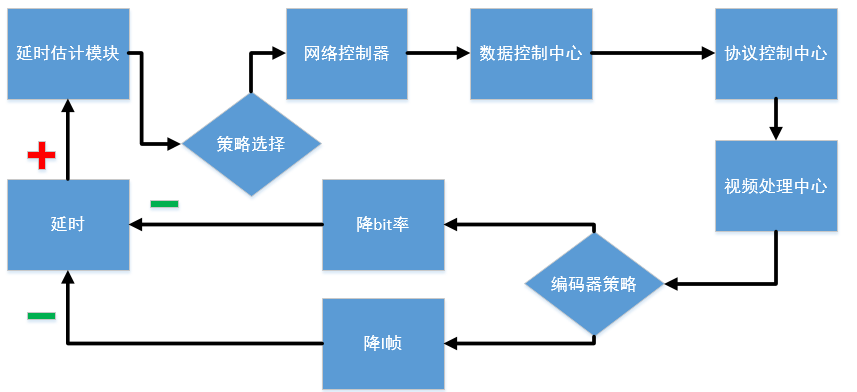
数据统计：

此模块作用为统计所有占用网络资源的数据。

## 网络自适应

因网络的特殊性，传输链路上状态是实时变化的，因为链路资源的公共性，云手机传输的链路时而变大时而变得很小，要想达到流畅度和体验感的平衡，就需要具备适应带宽变化的能力。

网络自适应对应的是网络控制器，通过延迟模块，可以得到当前实时的传输延时，当延迟大于某个阈值的时候，那么就说明当前链路中某地方发生了拥堵，那么此时就需要告诉数据控制中心需要降低本链路的带宽，数据中心会把拥塞状态发给协议控制中心，协议控制中心要求视频处理中心降低视频输出带宽，视频处理中心通过动态调整编码参数降低码率或减少I帧输出。整个控制流程构成了一个闭环的负反馈系统如下图所示：



## 2. 场景自适应

在生活中有很多的场景，比如开会，上学，工作等等，对应到手机中自然也有一系列的场景，如游戏场景，购物场景，看小说场景，视频场景等。这些场景对于编码起来说是非常具有操作价值，如视频场景，我们知道电影的帧率大概在24帧并且相连的两帧有极强的关联，一般来说关联性越强，视频的冗余信息就越多，可编码压缩的空间也就越大，相应的I帧的间隔也就可以很大，最后输出的带宽也就明显变小。