001-硬编码-视频H264编码

> 作者: CC老师
> 仅提供学习使用,转载传播需注明出处

一.学习背景

5G网络作为第5代的移动通信网络,它的网络峰值传播速度可1以达到`10Gbps/s`.这比4G的的传输速度快数百倍.举个例子,整部超高画质电影下载可在1秒钟之内下载完成.

当然,随着5G技术的诞生,用在智能终端分享3D电影,游戏或者超高画质节目的时代已经毫无悬念的向我们走来.

想必大家也逐步了解,国内外的互联网公司也已经布局音视频,3D技术方面的开发者招聘和相关产品研发.目前落地推广最普遍的就是直播类项目和小视频类的项目.当然未来的方向肯定不止如此.

> **那么我们现在带着问题来学习?**
>
> - 为何编码?
> - 何为编码?

1.1 为何编码?

从存储角度和网络传输以及通用性 3个角度,压缩已经成了不可或缺的动作.压缩编码最基本的指标,就是压缩比. 压缩比通常都是小于1(如果等于或者大于1,是不是就失去了编码的意义了.编码的目的就是为了压缩数据体量).

1.2 何为编码?

编码就是按照一定的格式记录采样和量化后的数据.

1.2.1编码中软编码和硬编码的区别?

- 硬编码: 使用非CPU进行编码,例如使用GPU芯片处理
- 软编码: 使用CPU来进行编码计算.

1.2.2 软编码与硬编码的区分?

- 软编码: 实现直接、简单，参数调整方便，升级易，但CPU负载重，性能较硬编码低，低码率下质量通常比硬编码要好一点。
- 硬编码：性能高，低码率下通常质量低于硬编码器，但部分产品在GPU硬件平台移植了优秀的软编码算法（如X264）的，质量基本等同于软编码。

> **读者理解:**
> 硬编码,就是使用GPU计算,获取数据结果,优点速度快,效率高.
> 软编码,就是通过CPU来计算,获取数据结果.

1.2.3 压缩算法

压缩算法分为2种,有损压缩与无损压缩.

- 无损压缩:解压后的数据可以完全复原,在常用的压缩格式中,无损压缩使用频次较低
- 有损压缩:解压后数据不能完全复原,会丢失一部分信息.压缩比越小,丢失的信息就会越多.信号还原的失真就会越大.

需要根据不同的场景(考虑因素包括存储设备,传输网络环境,播放设备等)选用不同的压缩编码算法.

二. 直播APP需求剖析

2.1 直播项目流程

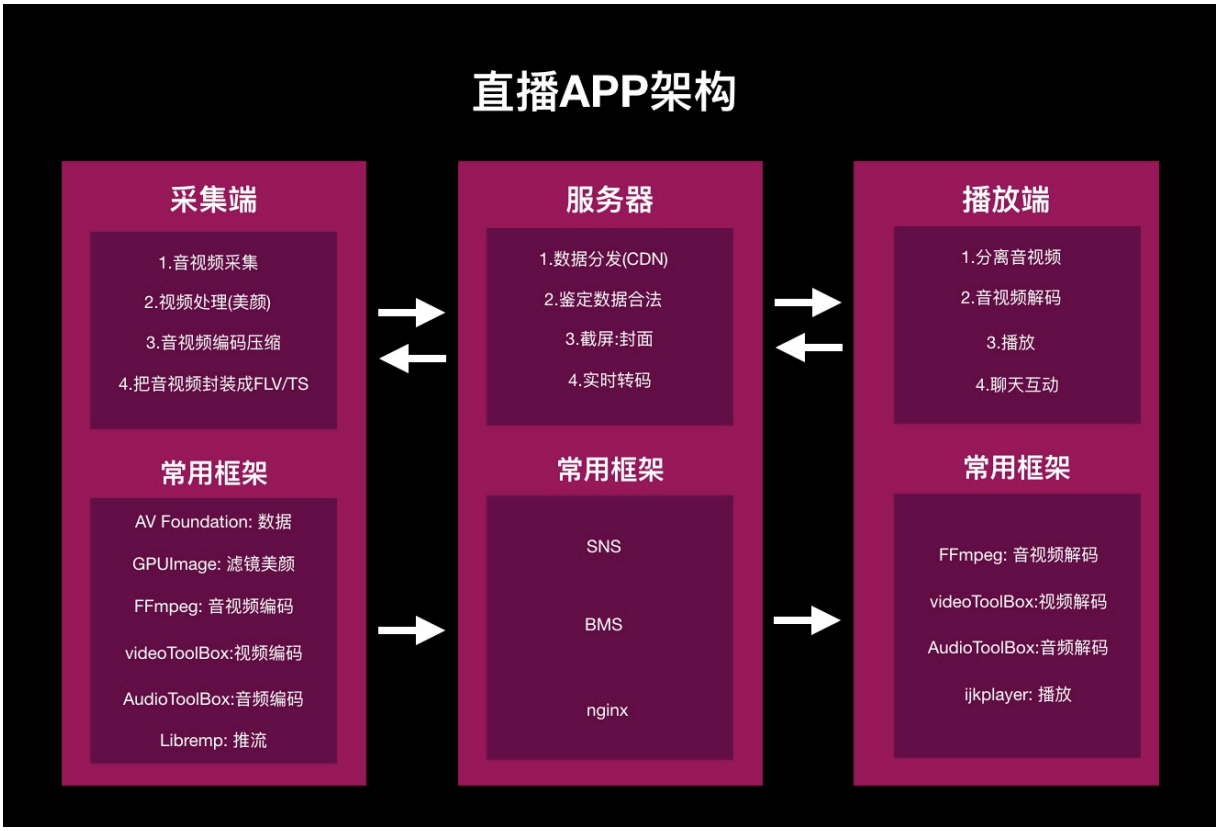
在直播项目中,一般常见有8个步骤.

- 音视频采集
- 视频滤镜
- 音视频编码
- 推流
- 流媒体服务器处理
- 拉流
- 音视频解码
- 音视频播放

这个在开发者面试一些有意向或者目前业务中包含直播需求的公司,最为常见的面

试题.不管在我们过往的工作经验是否有直播或音视频相关经验.这个一块都是你必须能了解.希望大家可以简单的了解.

2.2 相关框架的学习与使用场景



- **采集视频,音频**
  - 使用iOS原生框架 `AVFoundation.framework`
- **视频滤镜处理**
  - 使用iOS原生框架 `CoreImage.framework`
  - 使用第三方框架 `GPUImage.framework`

> `CoreImage` 与 `GPUImage` **框架比较:**
> 在实际项目开发中,开发者更加倾向使用于 `GPUImage` 框架.
> 首先它在使用性能上与iOS提供的原生框架,并没有差别;其次它的使用便利性高于iOS原生框架,最后也是最重要的 `GPUImage` 框架是开源的.而大家如果想要学习 `GPUImage` 框架,建议学习 `OpenGL ES` ,其实 `GPUImage` 的封装和思维都是基于 `OpenGL ES` .
> GPUImage OC版本下载地址
> GPUImage Swift版本下载地址

- **视频\音频编码压缩**

    - 硬编码
        - 视频: `VideoToolBox` 框架
        - 音频: `AudioToolBox` 框架
    - 软编码
        - 视频: 使用 `FFmpeg` , `X264` 算法把视频原数据YUV/RGB编码成H264
        - 音频: 使用 `fdk_aac` 将音频数据PCM转换成AAC

- 推流
    - 推流: 将采集的音频.视频数据通过流媒体协议发送到流媒体服务器
    - 推流技术
        - 流媒体协议: RTMP\RTSP\HLS\FLV
        - 视频封装格式: TS\FLV
        - 音频封装格式: Mp3\AAC

- 流媒体服务器
    - 数据分发
    - 截屏
    - 实时转码
    - 内容检测

- 拉流
    - 拉流: 从流媒体服务器中获取音频\视频数据
    - 流媒体协议: RTMP\RTSP\HLS\FLV

- 音视频解码
    - 硬解码
        - 视频: `VideoToolBox` 框架
        - 音频: `AudioToolBox` 框架
    - 软解码
        - 视频: 使用 `FFmpeg` , `X264` 算法解码
        - 音频: 使用 `fdk_aac` 解码

- 播放
    - `ijkplayer` 播放框架
    - `kxmovie` 播放框架

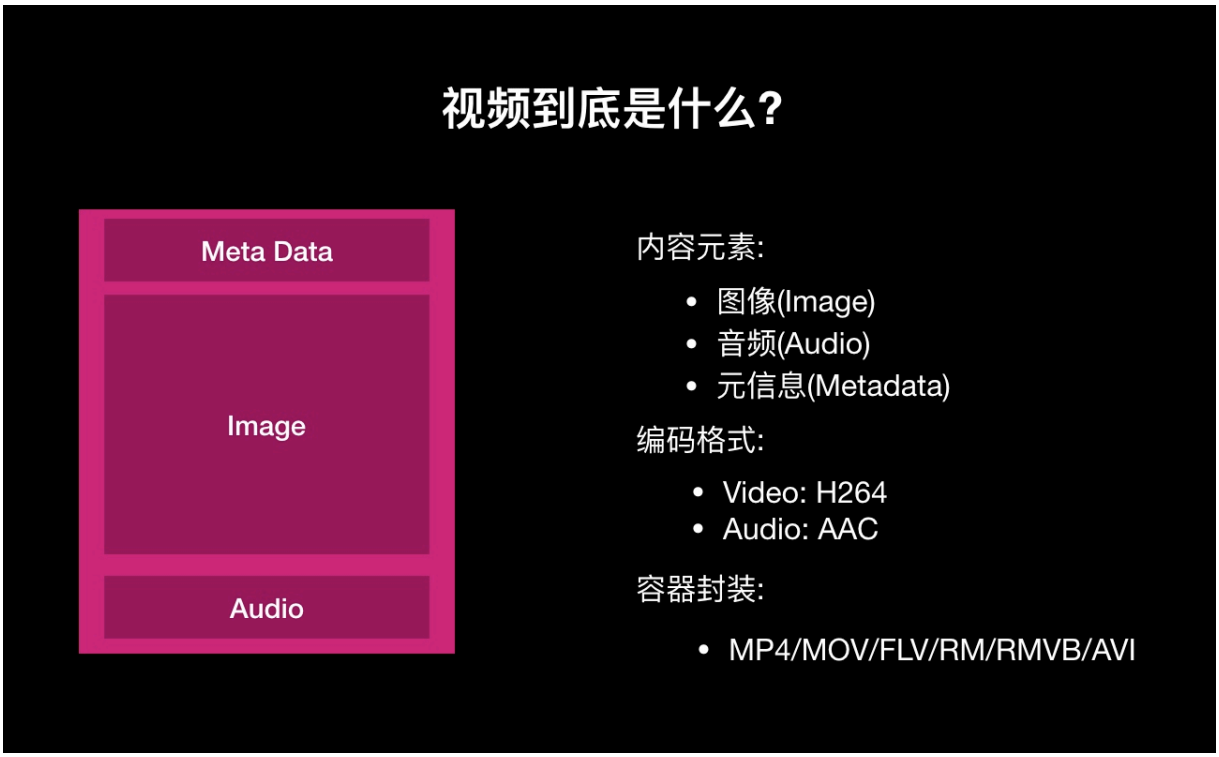
> `ijkplayer` , `kxmovie` 都是基于FFmpeg框架封装的

三.了解视频

作为开发者和产品测试之间的斡旋.是不可避免的.这是咱们在做开发过程中,但还是 要去思考.到底是需求不合理还是我们对这个需求的没有把握.

在前面我说过,5G时代的到来,势必会对移动互联网的冲击将会是最刺激的.为何?一旦网络速度+设备的运行速度不受限制时,设想移动设备的暂时不可替代性是不是又更加的明显了.至少在目前为止,没有一款硬件可以替代手机设备在用户的地位.

那我们来了解一下视频的常规知识.只有在了解视频的本质,你才有可能面对需求应对自如.



3.1 视频的构成:

- 图像
- 音频
- 元信息

图像: 视频内容本身就是一帧一帧的图片构成.人眼只要1秒钟连续播放16张以上的图片,就会认为这是一段连贯的视频.这种物理现象叫视觉暂留.

音频: 视频一定是由音频+图像内容构成的.所以音频在视频中是单独的一个部分.针对这一块我们需要单独编码的.

元信息:元信息其实就是描述信息的信息.用于描述信息的结构\语义\用途\用法等.比如视频元信息就包含了视频的具体信息,比如编码格式,分辨率等等.

3.2 视频中的编码格式

- **视频编码格式**
    - H264编码的优势:
        - 低码率
        - 高质量的图像
        - 容错能力强
        - 网络适应性强
    - **总结:** H264最大的优势,具有很高的数据压缩比率,在同等图像质量下,H264的压缩比是MPEG-2的2倍以上,MPEG-4的1.5~2倍.
    - **举例:** 原始文件的大小如果为88GB，采用MPEG-2压缩标准压缩后变成3.5GB，压缩比为25:1，而采用H.264压缩标准压缩后变为879MB，从88GB到879MB，H.264的压缩比达到惊人的102:1
- **音频编码格式:**
    - AAC是目前比较热门的有损压缩编码技术,并且衍生了LC-AAC,HE-AAC,HE-AAC v2 三种主要编码格式.
        - LC-AAC 是比较传统的AAC,主要应用于中高码率的场景编码(>= 80Kbit/s)
        - HE-AAC 主要应用于低码率场景的编码(<= 48Kbit/s)
    - **优势**:在小于128Kbit/s的码率下表现优异,并且多用于视频中的音频编码
    - **适合场景**:于128Kbit/s以下的音频编码,多用于视频中的音频轨的编码

3.3 容器(视频封装格式)

封装格式:就是将已经编码压缩好的视频数据 和音频数据按照一定的格式放到一个文件中.这个文件可以称为容器. 当然可以理解为这只是一个外壳.

通常我们不仅仅只存放音频数据和视频数据,还会存放 一下视频同步的元数据.例如字幕.这多种数据会不同的程序来处理,但是它们在传输和存储的时候,这多种数据都是被绑定在一起的.

- **常见的视频容器格式**:
  - `AVI`：是当时为对抗quicktime格式（mov）而推出的，只能支持固定CBR恒定定比特率编码的声音文件
  - `MOV`:是Quicktime封装
  - `WMV`:微软推出的，作为市场竞争
  - `mkv`： 万能封装器，有良好的兼容和跨平台性、纠错性，可带外挂字幕
  - `flv`：这种封装方式可以很好的保护原始地址，不容易被下载到，目前一些视频分享网站都采用这种封装方式
  - `MP4`： 主要应用于mpeg4的封装，主要在手机上使用。

四.视频压缩的可能性

视频压缩,该从那几个方向去进行数据的压缩了? 实际上压缩的本质都是从冗余信息开始出发压缩的. 而视频数据之间是有极强的相关性.也就是这样会产生大量的冗余信息.这样的冗余包括空间上的冗余信息和时间上的冗余信息.

- **使用帧间编码技术可以去除时间上的冗余信息,具体包括如下**
  - **运动补偿**: 运动补偿是通过先前的局部图形来预测,补偿当前的局部图像.它是减少帧序列冗余信息很有效的方法.
  - **运动表示**: 不同区域的图像需要使用不同的运动矢量来描述运动信息
  - **运动估计**: 运动估计就是从视频序列中抽取运动信息的一整套技术.

4.1 编码概念

IPB帧
视频压缩中,每帧代表着一副静止的图像.而进行实际压缩时,会采用各种算法以减少数据的容量.其实IPB帧是最常用的一种方式:

- **I帧**: 帧内编码帧(intra picture),I帧通常是每个GOP(MPEG所使用的一种视频压缩技术)的第一帧.经过适度的压缩.作为随机访问的参考点,可以当做静态图像.I帧可以看做一个图像经过压缩后的产物.I帧压缩可以得到6:1的压缩比而不会产生任何可察觉的模糊现象.I帧压缩去除了视频空间的冗余信息.
- **P帧**: 前后预测编码帧(predictive-frame),通过将图像序列中前面已编码帧的时间冗余信息充分去除来压缩传输数据量的编码图像.
- **B帧**: 双向预测编码帧(bi-directional interpolated prediction frame),既要考虑源图像序列前面已编码帧,又要顾及源图像序列后面的已编码帧之间的时间冗余信息,来压缩传输数据量的编码图像.

> 读者角度解读

如果从编码的角度,获取我们顺序思考会存在难度.但如果我们从解码的角度来思考就显得不是那么不可理解了.

- **I帧**,自身可以通过视频解码算法解压成一张单独的完整的视频画面.所以I帧去掉的是视频帧在空间维度上的冗余信息.
- **P帧**,需要参考前面的一个I帧或P帧解码成一个完整的视频画面
- **B帧**,需要参考前面的一个I帧或者P帧以及后面的一个P帧来生成一个完整的视频画面.
- **所以,P和B帧去掉的视频帧在时间上维度上的冗余信息**.

4.2 解码中PTS 与 DTS

`DTS(Decoding Time Stamp)`,主要用于视频的解码;
`PTS(Presentation Time Stamp)`,主要用于解码节点进行视频的同步和输出.

在没有B帧的情况下,DTS和PST的输出顺序是一样的.因为B帧会打乱了解码和显示顺序.所以一旦存在B帧,PTS和DTS势必会不同.实际上在大多数编解码标准中,编码顺序和输入顺序并不一致.于是需要PTS和DST这2种不同的时间戳.

4.3 GOP概念

两个I帧之间形成的一组图片,就是GOP(Group of Picture).
通常在编码器设置参数时,必须会设置`gop_size`的值.其实就是代表2个I帧之间的帧数目. 在一个GOP组中容量最大的就是I帧.所以相对而言,`gop_size`设置的越大,整个视频画面质量就会越好.但是解码端必须从接收的第一个I帧开始才可以正确解码出原始图像.否则无法正确解码.

五.了解VideoToolBox 硬编码

VideoToolBox 官方文档

在iOS4.0,苹果就已经支持硬编解码.但是硬编解码在当时属于私有API. 不提供给开发者使用
在2014年的WWDC大会上,iOS 8.0 之后,苹果开放了硬编解码的API。就是`VideoToolbox.framework`的API。`VideoToolbox`是一套纯C语言API。其中包含了很多C语言函数.`VideoToolbox.framework`是基于`Core Foundation`库函数,

基于C语言

`VideoToolBox`实际上属于低级框架,它是可以直接访问硬件编码器和解码器.它存在于视频压缩和解压缩以及存储在像素缓存区中的数据转换提供服务.

硬编码的优点

- 提高编码性能(使用CPU的使用率大大降低,倾向使用GPU)
- 增加编码效率(将编码一帧的时间缩短)
- 延长电量使用(耗电量大大降低)

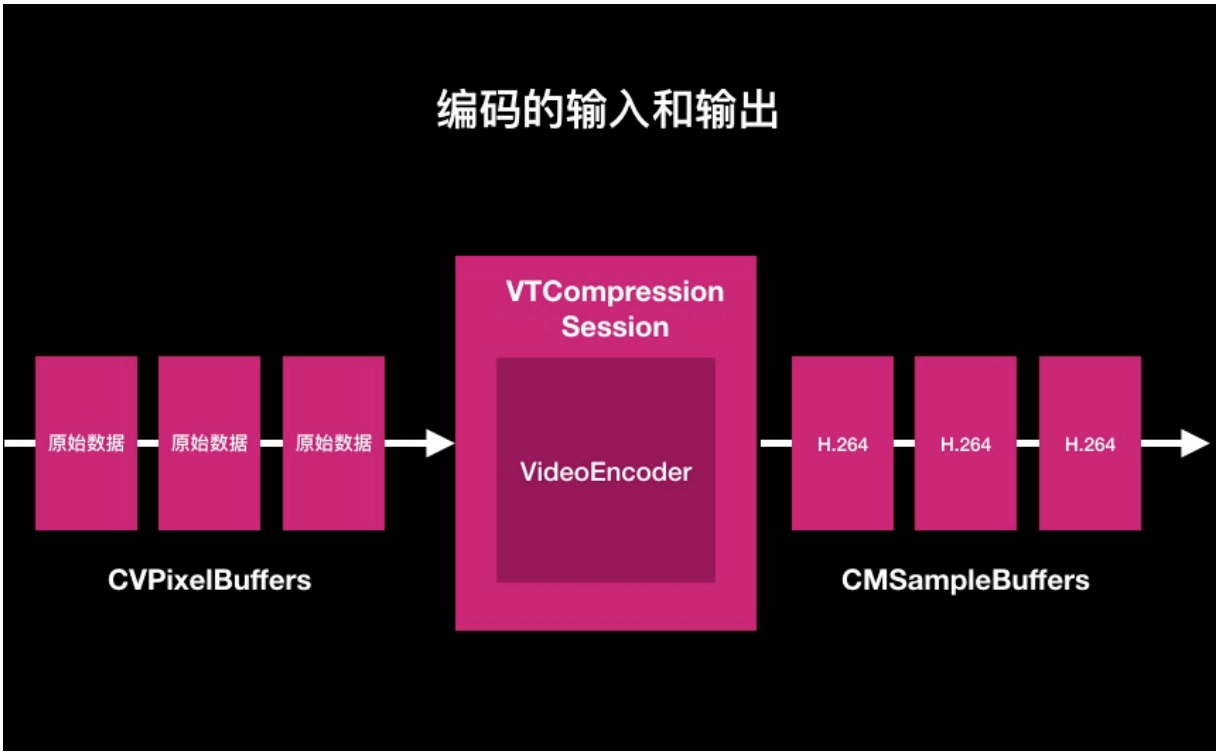
这个框架在音视频项目开发中,也是会要频繁使用的.如果大家有想法去从事音视频的开发.那么这个框架将会是你学习的一个重点.

VideoToolBox框架的流程

- 创建session
- 设置编码相关参数
- 开始编码
- 循环获取采集数据
- 获取编码后数据
- 将数据写入H264文件

5.1 编码的输入和输出

在我们开始编码工作之前,需要了解`VideoToolBox`进行编解码的输入输出分别是什么? 只有了解了这个,我们才能清楚知道如何去向`VideoToolBox`添加数据,并且如何获取数据.



如图所示,左边的三帧视频帧是发送給编码器之前的数据,开发者必须将原始图像数据封装为CVPixelBuufer的数据结构.该数据结构是使用VideoToolBox的核心.

Apple Developer CVPixelBuffer 官方文档介绍

5.2 CVPixelBuffer 解析

在这个官方文档的介绍中,CVPixelBuffer 给的官方解释,是其主内存存储所有像素点数据的一个对象.那么什么是主内存了?
其实它并不是我们平常所操作的内存,它指的是存储区域存在于缓存之中. 我们在访问这个块内存区域,需要先锁定这块内存区域.

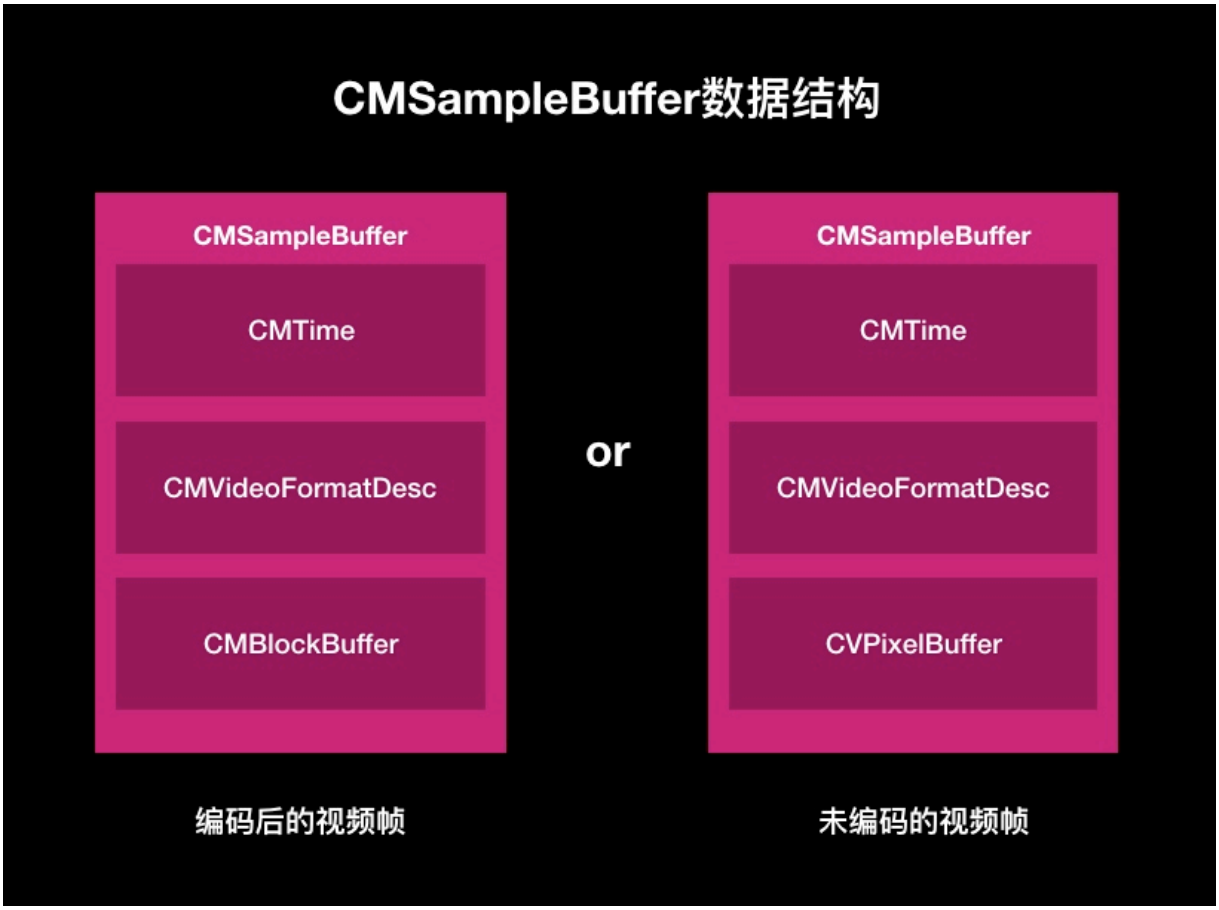
```
//1.锁定内存区域:
   CVPixelBufferLockBaseAddress(pixel_buffer,0);
//2.读取该内存区域数据到NSData对象中
   Void *data = CVPixelBufferGetBaseAddress(pixel_buffer);
//3.数据读取完毕后,需要释放锁定区域
   CVPixelBufferRelease(pixel_buffer);
```

单纯从它的使用方式,我们就可以知道这一块内存区域不是普通内存区域.它需要加锁,解锁等一系列操作.
作为视频开发,尽量减少进行显存和内存的交换.所以在iOS开发过程中也要尽量减

少对它的内存区域访问.建议使用iOS平台提供的对应的API来完成相应的一系列操作.

在`AVFoundation`回调方法中,它有提供我们的数据其实就是`CVPixelBuffer`.只不过当时使用的是引用类型`CVImageBufferRef`,其实就是`CVPixelBuffer`的另外一个定义.

`Camera`返回的`CVImageBuffer`中存储的数据是一个`CVPixelBuffer`,而经过`VideoToolBox`编码输出的`CMSampleBuffer`中存储的数据是一个`CMBlockBuffer`的引用.



在**iOS**中,会经常使用到`session`的方式.比如我们使用任何硬件设备都要使用对应的`session`,麦克风就要使用`AudioSession`,使用`Camera`就要使用`AVCaptureSession`,使用编码则需要使用`VTCompressionSession`.解码时,要使用`VTDecompressionSessionRef`.

5.3 视频编码步骤分解

第一步: 使用VTCompressionSessionCreate方法,创建编码会话;

```
    //1.调用VTCompressionSessionCreate创建编码session
        //参数1: NULL 分配器,设置NULL为默认分配
        //参数2: width
        //参数3: height
        //参数4: 编码类型,如kCMVideoCodecType_H264
        //参数5: NULL encoderSpecification: 编码规范。设置NULL由videoToolbox自己选择
        //参数6: NULL sourceImageBufferAttributes: 源像素缓冲区属性.设置NULL不让videToolbox创建,而自己创建
        //参数7: NULL compressedDataAllocator: 压缩数据分配器.设置NULL,默认的分配
        //参数8: 回调  当VTCompressionSessionEncodeFrame被调用压缩一次后会被异步调用.注:当你设置NULL的时候,你需要调用VTCompressionSessionEncodeFrameWithOutputHandler方法进行压缩帧处理,支持iOS9.0以上
        //参数9: outputCallbackRefCon: 回调客户定义的参考值
        //参数10: compressionSessionOut: 编码会话变量
        OSStatus status = VTCompressionSessionCreate(NULL, width, height, kCMVideoCodecType_H264, NULL, NULL, NULL, didCompressH264, (__bridge void *)(self), &cEncodeingSession);
```

第二步:设置相关的参数

```
/*
    session: 会话
    propertyKey: 属性名称
    propertyValue: 属性值
*/
VT_EXPORT OSStatus
VTSessionSetProperty(
  CM_NONNULL VTSessionRef       session,
  CM_NONNULL CFStringRef        propertyKey,
  CM_NULLABLE CFTypeRef         propertyValue ) API_AVAILABLE(macosx(10.8), ios(8.0), tvos(10.2));
```

- `kVTCompressionPropertyKey_RealTime`:设置是否实时编码
- `kVTProfileLevel_H264_Baseline_AutoLevel`:表示使用`H264`的`Profile`规格,可以设置`Hight`的`AutoLevel`规格.
- `kVTCompressionPropertyKey_AllowFrameReordering`:表示是否使用产生B帧数据(因为B帧在解码是非必要数据,所以开发者可以抛弃B帧数据)

- `kVTCompressionPropertyKey_MaxKeyFrameInterval`：表示关键帧的间隔,也就是我们常说的gop size.
- `kVTCompressionPropertyKey_ExpectedFrameRate`：表示设置帧率
- `kVTCompressionPropertyKey_AverageBitRate` / `kVTCompressionPropertyKey_DataRateLimits` 设置编码输出的码率.

第三步: 准备编码

```
//开始编码
VTCompressionSessionPrepareToEncodeFrames(cEncodeingSession);
```

第四步: 捕获编码数据

- 通过AVFoundation 捕获的视频,这个时候我们会走到AVFoundation捕获结果代理方法:

```
#pragma mark - AVCaptureVideoDataOutputSampleBufferDelegate
//AV Foundation 获取到视频流
-(void)captureOutput:(AVCaptureOutput *)captureOutput didOutputSample
Buffer:(CMSampleBufferRef)sampleBuffer fromConnection:(AVCaptureConne
ction *)connection
{
    //开始视频录制，获取到摄像头的视频帧，传入encode 方法中
    dispatch_sync(cEncodeQueue, ^{
        [self encode:sampleBuffer];
    });

}
```

第五步:数据编码

- 将获取的视频数据编码

```
- (void) encode:(CMSampleBufferRef )sampleBuffer
{
    //拿到每一帧未编码数据
```

```
    CVImageBufferRef imageBuffer = (CVImageBufferRef)CMSampleBufferGe
tImageBuffer(sampleBuffer);

    //设置帧时间，如果不设置会导致时间轴过长。时间戳以ms为单位
    CMTime presentationTimeStamp = CMTimeMake(frameID++, 1000);

    VTEncodeInfoFlags flags;

    //参数1：编码会话变量
    //参数2：未编码数据
    //参数3：获取到的这个sample buffer数据的展示时间戳。每一个传给这个sess
ion的时间戳都要大于前一个展示时间戳.
    //参数4：对于获取到sample buffer数据,这个帧的展示时间.如果没有时间信息
,可设置kCMTimeInvalid.
    //参数5：frameProperties：包含这个帧的属性.帧的改变会影响后边的编码帧
.
    //参数6：ourceFrameRefCon：回调函数会引用你设置的这个帧的参考值.
    //参数7：infoFlagsOut：指向一个VTEncodeInfoFlags来接受一个编码操作.
如果使用异步运行,kVTEncodeInfo_Asynchronous被设置；同步运行,kVTEncodeInf
o_FrameDropped被设置；设置NULL为不想接受这个信息.

    OSStatus statusCode = VTCompressionSessionEncodeFrame(cEncodeingS
ession, imageBuffer, presentationTimeStamp, kCMTimeInvalid, NULL, NUL
L, &flags);

    if (statusCode != noErr) {

        NSLog(@"H.264:VTCompressionSessionEncodeFrame faild with %d",
(int)statusCode);

        VTCompressionSessionInvalidate(cEncodeingSession);
        CFRelease(cEncodeingSession);
        cEncodeingSession = NULL;
        return;
    }

    NSLog(@"H264:VTCompressionSessionEncodeFrame Success");

}
```

第六步: 编码数据处理-获取SPS/PPS

当编码成功后,就会回调到最开始初始化编码器会话时传入的回调函数,回调函数的原型如下:

`void didCompressH264(void *outputCallbackRefCon, void *sourceFrameRefCon, OSStatus status, VTEncodeInfoFlags infoFlags, CMSampleBufferRef sampleBuffer)`

- **判断status,如果成功则返回0(noErr);成功则继续处理,不成功则不处理.**
- **判断是否关键帧**
  - **为什么要判断关键帧呢?**
  - 因为VideoToolBox编码器在每一个关键帧前面都会输出SPS/PPS信息.所以如果本帧未关键帧,则可以取出对应的SPS/PPS信息.

```
 //判断当前帧是否为关键帧
CFArrayRef array = CMSampleBufferGetSampleAttachmentsArray(sampleBuffer, true);
CFDictionaryRef dic = CFArrayGetValueAtIndex(array, 0);
bool isKeyFrame = !CFDictionaryContainsKey(dic, kCMSampleAttachmentKey_NotSync);

bool keyFrame = !CFDictionaryContainsKey((CFArrayGetValueAtIndex(CMSampleBufferGetSampleAttachmentsArray(sampleBuffer, true), 0)), kCMSampleAttachmentKey_NotSync);
```

- **那么如何获取SPS/PPS信息了?**

```
 //判断当前帧是否为关键帧
    //获取sps & pps 数据 只获取1次，保存在h264文件开头的第一帧中
    //sps(sample per second 采样次数/s),是衡量模数转换（ADC）时采样速率的单位
    //pps()
    if (keyFrame) {

        //图像存储方式，编码器等格式描述
        CMFormatDescriptionRef format = CMSampleBufferGetFormatDescription(sampleBuffer);

        //sps
        size_t sparameterSetSize,sparameterSetCount;
        const uint8_t *sparameterSet;
        OSStatus statusCode = CMVideoFormatDescriptionGetH264ParameterSetAtIndex(format, 0, &sparameterSet, &sparameterSetSize, &sparameterSetCount, 0);

        if (statusCode == noErr) {
```

```
            //获取pps
            size_t pparameterSetSize,pparameterSetCount;
            const uint8_t *pparameterSet;

            //从第一个关键帧获取sps & pps
            OSStatus statusCode = CMVideoFormatDescriptionGetH264Para
meterSetAtIndex(format, 1, &pparameterSet, &pparameterSetSize, &ppara
meterSetCount, 0);

            //获取H264参数集合中的SPS和PPS
            if (statusCode == noErr)
            {
                //Found pps & sps
                NSData *sps = [NSData dataWithBytes:sparameterSet len
gth:sparameterSetSize];
                NSData *pps = [NSData dataWithBytes:pparameterSet len
gth:pparameterSetSize];

                if(encoder)
                {
                    [encoder gotSpsPps:sps pps:pps];
                }
            }
        }

    }
```

第七步 编码压缩数据并写入H264文件

当我们获取了SPS/PPS信息之后,我们就获取实际的内容来进行处理了

```
 CMBlockBufferRef dataBuffer = CMSampleBufferGetDataBuffer(sampleBuff
er);
    size_t length,totalLength;
    char *dataPointer;
    OSStatus statusCodeRet = CMBlockBufferGetDataPointer(dataBuffer,
0, &length, &totalLength, &dataPointer);
    if (statusCodeRet == noErr) {
        size_t bufferOffset = 0;
        static const int AVCCHeaderLength = 4;//返回的nalu数据前4个字
节不是001的startcode,而是大端模式的帧长度length

        //循环获取nalu数据
```

```
        while (bufferOffset < totalLength - AVCCHeaderLength) {

            uint32_t NALUnitLength = 0;

            //读取 一单元长度的 nalu
            memcpy(&NALUnitLength, dataPointer + bufferOffset, AVCCHe
aderLength);

            //从大端模式转换为系统端模式
            NALUnitLength = CFSwapInt32BigToHost(NALUnitLength);

            //获取nalu数据
            NSData *data = [[NSData alloc]initWithBytes:(dataPointer
+ bufferOffset + AVCCHeaderLength) length:NALUnitLength];

            //将nalu数据写入到文件
            [encoder gotEncodedData:data isKeyFrame:keyFrame];

            //move to the next NAL unit in the block buffer
            //读取下一个nalu 一次回调可能包含多个nalu数据
            bufferOffset += AVCCHeaderLength + NALUnitLength;


        }

    }


}

//第一帧写入 sps & pps
- (void)gotSpsPps:(NSData*)sps pps:(NSData*)pps
{
    NSLog(@"gotSpsPp %d %d",(int)[sps length],(int)[pps length]);

    const char bytes[] = "\x00\x00\x00\x01";

    size_t length = (sizeof bytes) - 1;

    NSData *ByteHeader = [NSData dataWithBytes:bytes length:length];

    [fileHandele writeData:ByteHeader];
    [fileHandele writeData:sps];
    [fileHandele writeData:ByteHeader];
    [fileHandele writeData:pps];
```

```
}


- (void)gotEncodedData:(NSData*)data isKeyFrame:(BOOL)isKeyFrame
{
    NSLog(@"gotEncodeData %d",(int)[data length]);

    if (fileHandele != NULL) {

        //添加4个字节的H264 协议 start code 分割符
        //一般来说编码器编出的首帧数据为PPS & SPS
        //H264编码时，在每个NAL前添加起始码 0x000001,解码器在码流中检测起始码，当前NAL结束。
        /*
         为了防止NAL内部出现0x000001的数据，h.264又提出'防止竞争 emulation prevention"机制，在编码完一个NAL时，如果检测出有连续两个0x00字节，就在后面插入一个0x03。当解码器在NAL内部检测到0x000003的数据，就把0x03抛弃，恢复原始数据。

         总的来说H264的码流的打包方式有两种,一种为annex-b byte stream format 的格式，这个是绝大部分编码器的默认输出格式，就是每个帧的开头的3~4个字节是H264的start_code,0x00000001或者0x000001。
         另一种是原始的NAL打包格式，就是开始的若干字节（1，2，4字节）是NAL的长度，而不是start_code,此时必须借助某个全局的数据来获得编 码器的profile,level,PPS,SPS等信息才可以解码。

         */
        const char bytes[] ="\x00\x00\x00\x01";

        //长度
        size_t length = (sizeof bytes) - 1;

        //头字节
        NSData *ByteHeader = [NSData dataWithBytes:bytes length:length];

        //写入头字节
        [fileHandele writeData:ByteHeader];

        //写入H264数据
        [fileHandele writeData:data];

    }

}
```