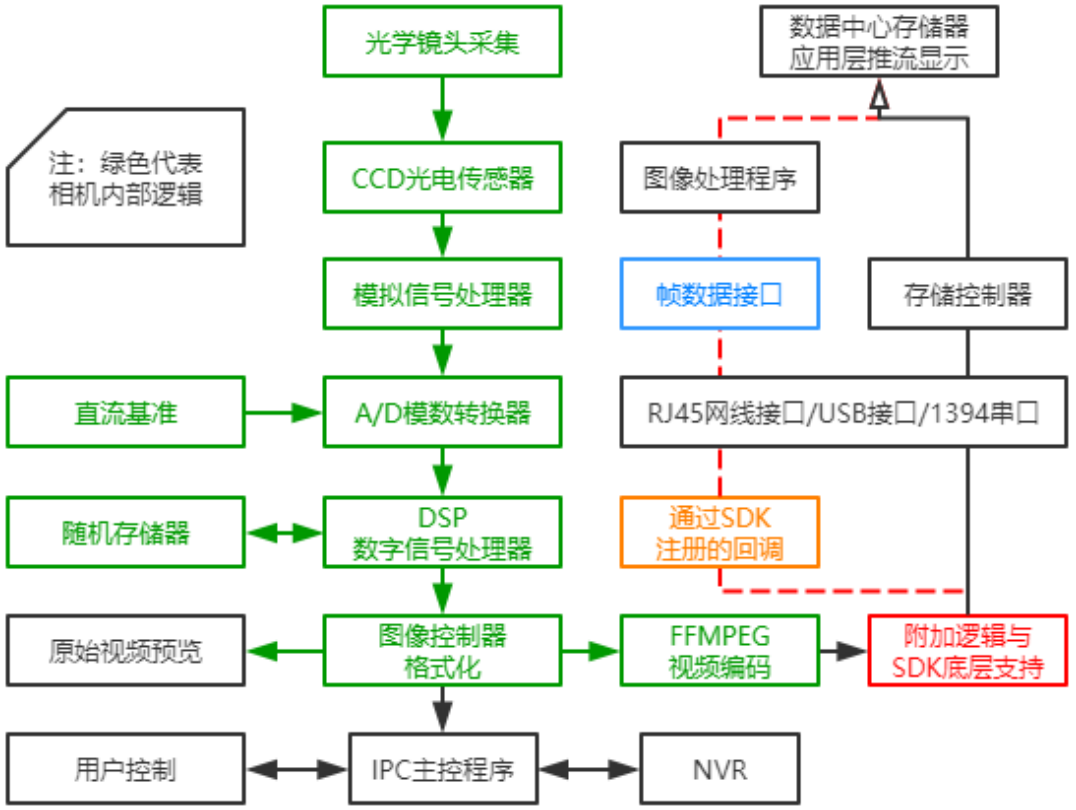
数据驱动层IPC取帧接口

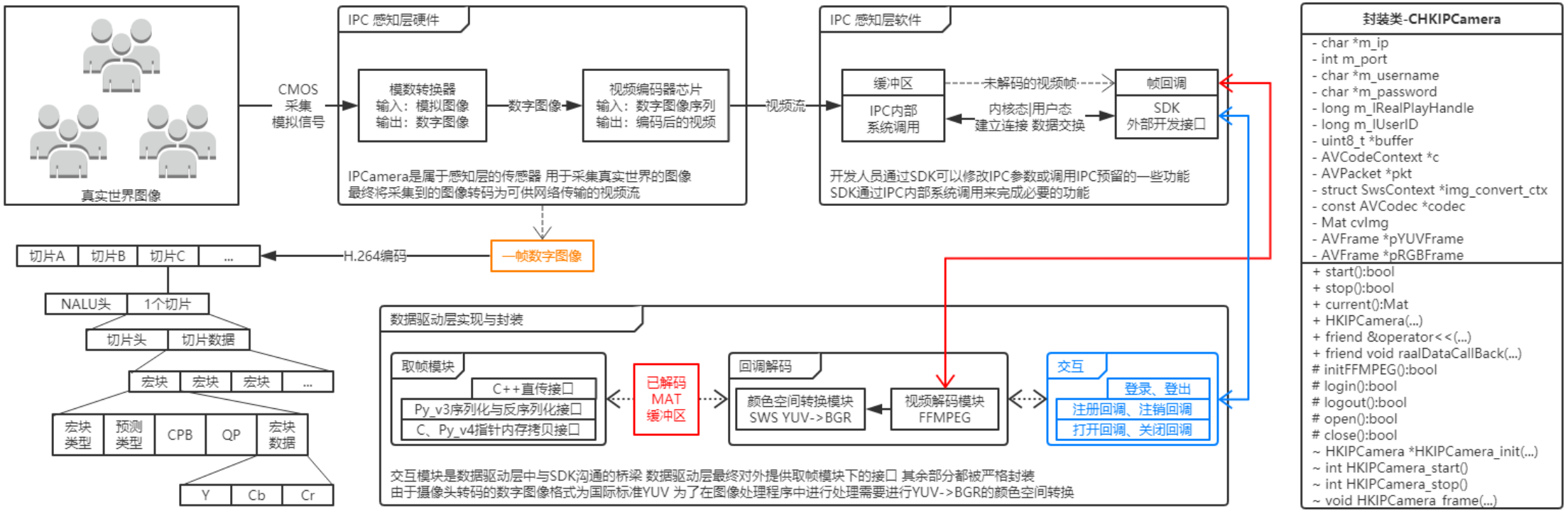
2019-02-06

功能介绍

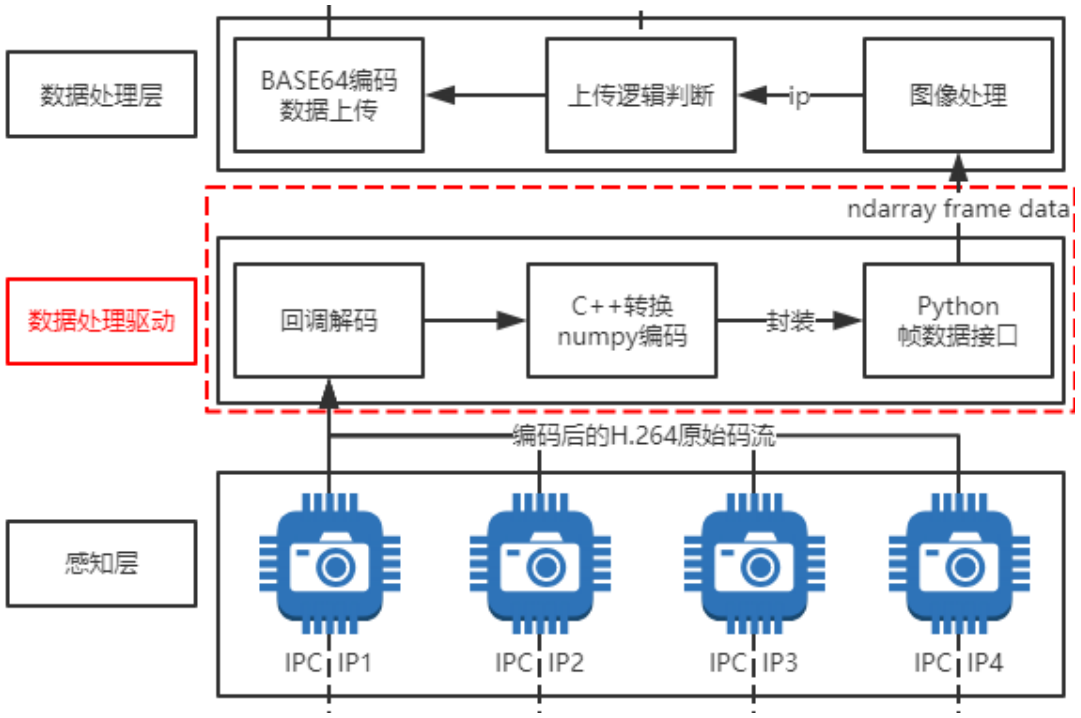
- 输入： H.264视频流
- 输出： 一帧Mat图像
- 功能： 通过抽取相似代码并进行封装实现取帧功能模块化
- 意义： 提高系统功能重用性 降低维护成本



处理流程与封装类定义

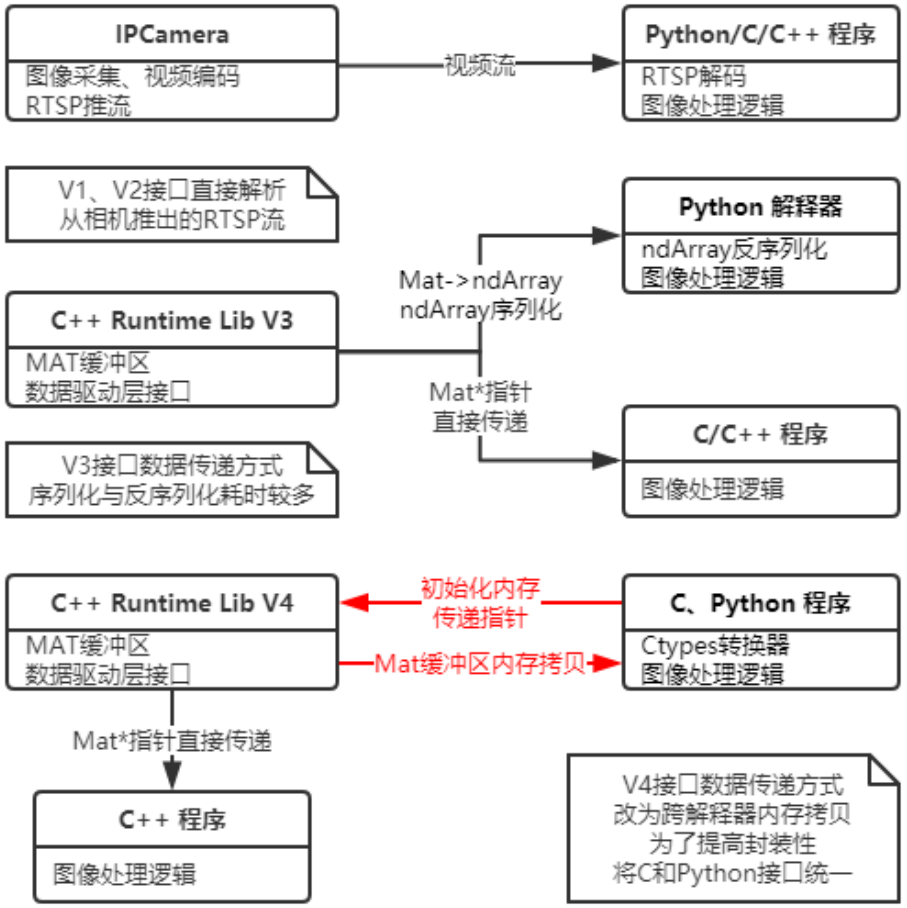

真实世界图像
CMOS
采集
模拟信号
IPC 感知层硬件
模数转换器
输入：模拟图像
输出：数字图像
数字图像
视频编码器芯片
输入：数字图像序列
输出：编码后的视频
视频流
IPCamera是属于感知层的传感器 用于采集真实世界的图像
最终将采集到的图像转码为可供网络传输的视频流
IPC 感知层软件
缓冲区
未解码的视频帧
帧回调
IPC内部
系统调用
内核态|用户态
建立连接 数据交换
SDK
外部开发接口
开发人员通过SDK可以修改IPC参数或调用IPC预留的一些功能
SDK通过IPC内部系统调用来完成必要的功能
封装类-CHKIPCamera
- char *m_ip
- int m_port
- char *m_username
- char *m_password
- long m_lRealPlayHandle
- long m_lUserID
- uint8_t *buffer
- AVCodeContext *c
- AVPacket *pkt
- struct SwsContext *img_convert_ctx
- const AVCodec *codec
- Mat cvImg
- AVFrame *pYUVFrame
- AVFrame *pRGBFrame
+ start():bool
+ stop():bool
+ current():Mat
+ HKIPCamera(...)
+ friend &operator<<(...)
+ friend void raalDataCallBack(...)
# initFFMPEG():bool
# login():bool
# logout():bool
# open():bool
# close():bool
~ HKIPCamera *HKIPCamera_init(...)
~ int HKIPCamera_start()
~ int HKIPCamera_stop()
~ void HKIPCamera_frame(...)
一帧数字图像
H.264编码
切片A
切片B
切片C
...
NALU头
1个切片
切片头
切片数据
宏块
宏块
宏块
...
宏块类型
预测类型
CPB
QP
宏块数据
Y
Cb
Cr
数据驱动层实现与封装
取帧模块
C++直传接口
Py_v3序列化与反序列化接口
C、Py_v4指针内存拷贝接口
已解码
MAT
缓冲区
回调解码
颜色空间转换模块
SWS YUV->BGR
视频解码模块
FFMPEG
交互
登录、登出
注册回调、注销回调
打开回调、关闭回调
交互模块是数据驱动层中与SDK沟通的桥梁 数据驱动层最终对外提供取帧模块下的接口 其余部分都被严格封装
由于摄像头转码的数字图像格式为国际标准YUV 为了在图像处理程序中进行处理需要进行YUV->BGR的颜色空间转换

接口在整个系统中的位置

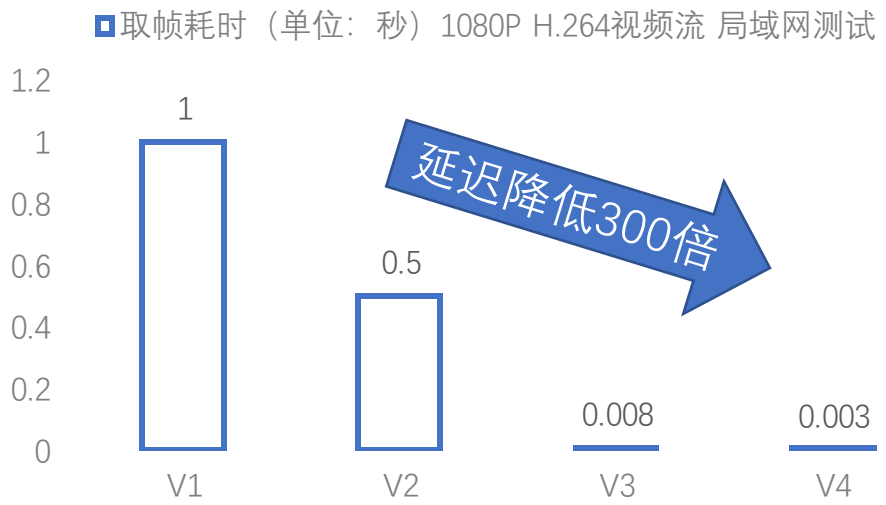


- 数据驱动层前是感知层，感知层的传感器负责采集数据
- 数据驱动层后是数据处理层，包含系统的核心业务逻辑
- 例如：图像处理逻辑
- 数据驱动层从感知层获取数据并进行格式化，为数据处理层提供数据支持
- 本接口就是数据驱动层的核心

迭代历史与版本差异

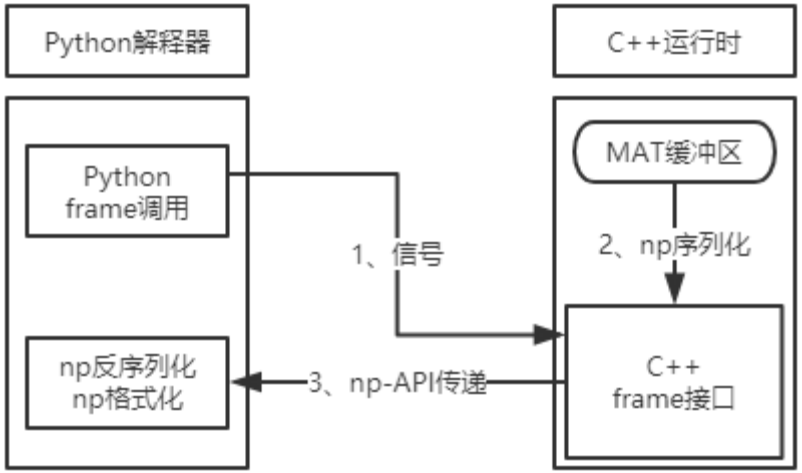


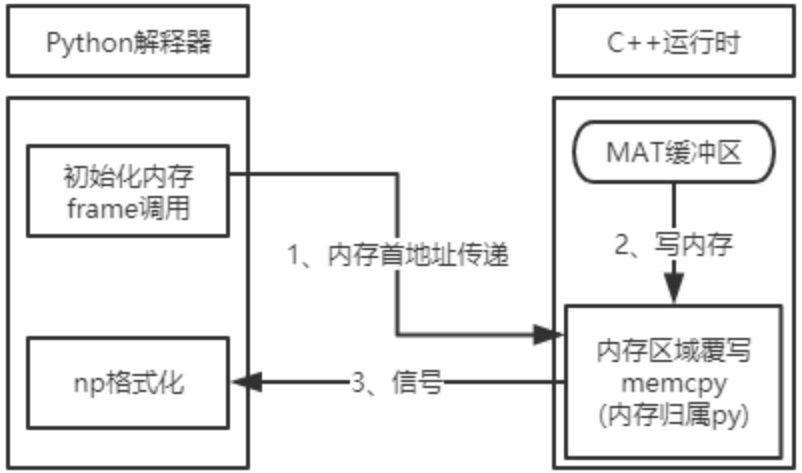
- V1、V2版本的接口依赖于IPC内部的RTSP编码和软件层的解码 实现简单但延迟高达0.5~1s
- V3版本通过直接解码未推送的视频流显著降低取帧延迟到0.008s左右（序列化与反序列化耗时严重）
- V4版本更新了解码API实现异步解码 而且针对Python接口的序列化性能问题进行了改进 直接传递指针进行内存拷贝来降低通信成本 最终将取帧延迟降低到0.003s左右



接口V4的主要修改——取帧速度提升

V3接口由于采用了Python-Numpy-API进行数据传输，需要对C++的Mat格式进行序列化，然后在python中进行反序列化来获得Mat数据。但由于1080P图像较大，导致序列化和反序列化耗时较多。V4版本修改了数据传递逻辑，直接传递内存首地址，在C++中进行内存覆写，然后在python中调用，不进行序列化和反序列化操作，从而提升了取帧速度。





接口V4的主要修改——封装性提升

- C++的函数编译后会生成函数名和形参表作为函数特征码
- 函数特征码是编译过程中对函数的唯一标识 用于实现重载、成员函数等C语言不具备的功能
- 对与class中的成员函数 形参表中会额外增加this指针用于调用类内属性和方法 （注：仅返回值类型不同不能构成重载）
  - 普通函数 void fun (int a) => 函数名 void fun 形参表 int a
  - 成员函数 Cxx::void fun (int a) => 函数名 void fun 形参表 Cxx *this int a
- 这种命名冲突导致在不修改回调函数定义的情况下**无法**将其作为成员函数封装 也**无法**调用类内部属性和方法

接口V4的主要修改——封装性提升

- Q:那么如何封装回调函数?
- A:我们的做法是使用友元函数配合额外指针参数传递对象。
- Cxx.h 函数定义:
  - public:
    - friend bool fun (int a, void* p);
    - 封装性提升体现在将回调函数作为成员函数可以将以下全局变量作为类内私有属性封装
- Cxx.cpp 函数实现:
  - bool fun (int a, void* p) {…}
- 调用示例:
  - 设置: setCallBack (fun,100,this);
  - 取用: Cxx cp = (Cxx*) p;

```
uint8_t *buffer; //for sws fill picture
AVCodecContext *c;
AVPacket *pkt;
struct SwsContext *img_convert_ctx;
const AVCodec *codec;
cv::Mat cvImg;
AVFrame *pYUVFrame;
AVFrame *pRGBFrame;
```