基于类MJPEG协议的流媒体服务框架说明

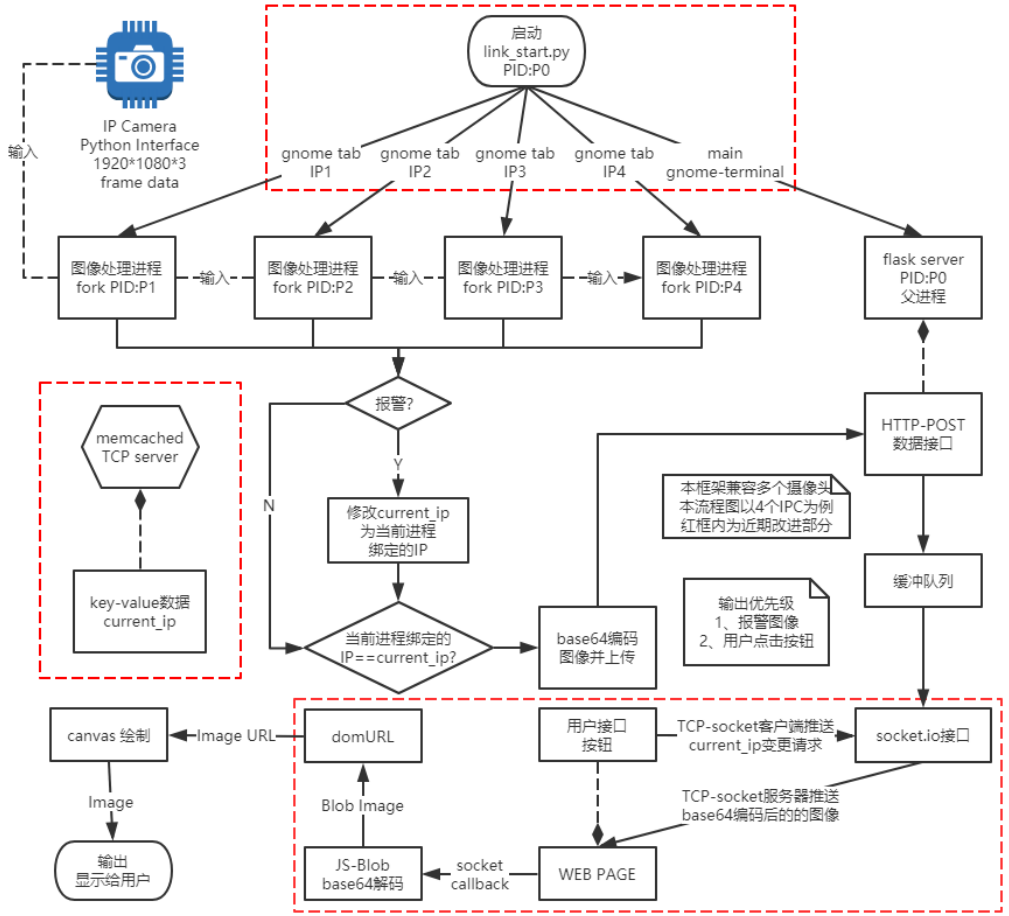
The Description of FLASK-MJPEG-MODULE-V2.0

2019-01-10

问题描述

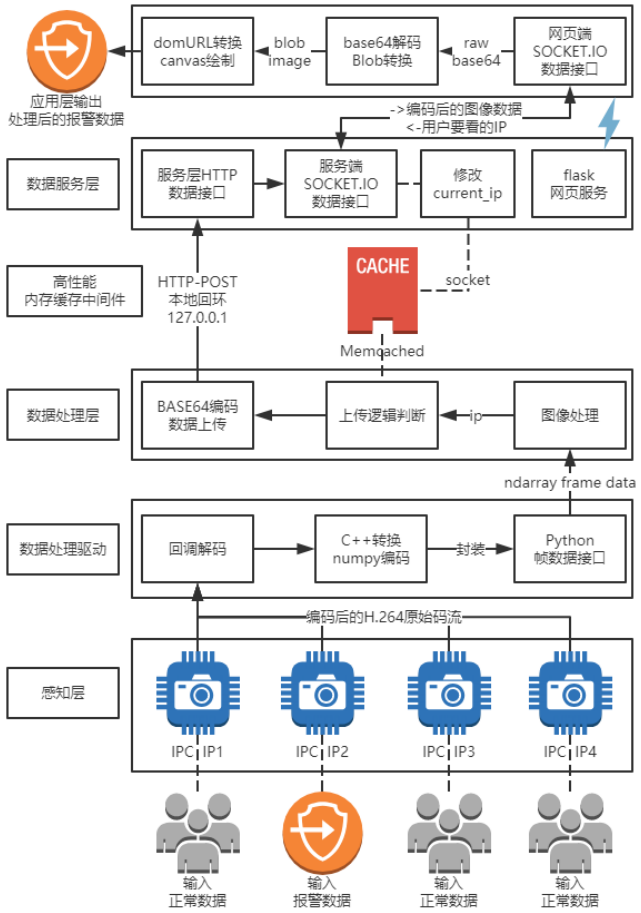
- 目前使用的H.264编码+nginx_rtmp_module分发+flash播放的方式
- 在CPU负载较高时存在以下问题：
  - 首帧等待长
  - 画面延迟高
  - 切换延迟高
- 此外，还存在以下不太严重的问题：
  - 前端兼容性差（需要安装并启用flash）
  - 网络波动断流
- 因此需要设计一套新的流媒体服务解决方案

处理流程——Pipeline of module v2



- 定义：
  - N为IPC的数量
  - I_x为IP为IP_x的IPC当前帧图像(通过接口得到ndarray)
- 输入：集合$\{I_{cam}|cam \in IP_1..IP_N\}$
- 输出：$I_{current\_ip}$
- 详细流程：
- 启动主进程会fork出N个子进程分别对应N个摄像头的图像处理
- 主进程执行服务端程序 memcache中存储的current_ip可以根据用户点击网页按钮改变 也会通过子进程的报警逻辑改变 报警优先度大于用户
- 每帧图像处理完会用socket取memcache判断是否应该上传到server
- 判断通过则对图像进行base64编码并上传
- Server通过web-socket与网页端建立TCP长连接 实时推送图像数据
- 网页端通过socket.io设置回调 监听、获得数据
- 图像数据先转换为blob 再转为domUrl 最后通过canvas绘制 完成流程

系统架构



- 层次说明：
  - 感知层：物联网设备采集外界信息 获取原始数据
  - 数据驱动层：将传感器采集的数据转化为可处理的格式化数据
  - 数据处理层：运用各种具体的方法进行数据挖掘
  - 中间件：提供高性能数据访问接口 本例中提供低延迟全局变量访问
  - 数据服务层：存储转发数据处理层提供的处理后的数据
  - 数据应用层：具体应用 客户端、网页、app
- 层间交互：
  - 层间数据交互全在内存中完成 除浏览器CSS、JS缓存外无硬盘读写
  - 图像处理进程数据上传使用本地回环 只走网卡驱动 不占用网卡带宽
- 关键技术：
  - python跨解释器通信
  - 低延迟进程通信Memcached
  - 类MJPEG协议
  - HTTP1.1长连接web-socket
- 架构重用性：
  - 各模块间无耦合 维护方便
  - 每层可单独提取并替换 重用性强
  - 如：CDS替换为安全工具/替换为可疑人员/替换为火焰烟雾检测

系统技术特点与适用场景

- 技术特点：
  - 首帧显示等待时间小于0.2s
  - 线路切换延迟小于0.2s
  - 图像延迟小于0.5s
  - 画质清晰
  - CPU占用率低
  - 前端兼容性强
  - 扩展性强
  - 部署简单
  - 崩溃自恢复
- 适用场景：
  - 用户要求图像延迟低
  - 用户要求切换延迟低
  - 用户要求首帧等待时间短
  - 用户要求画质清晰
  - 用户要求网页展示
  - 节约CPU运算资源
  - 网络带宽大于4.5Mbps
    - (注：v2版本已显著改善多路处理情况下的带宽占用)

架构优势——对比H.264+nginx_rtmp+flash

	本架构v1	本架构v2	对照组
首帧等待	0.2s~0.5s	小于0.2s	0.5s~1.0s
低负载图像延迟	0.5s~1.0s	小于0.5s	1.0s取决于HLS分片
高负载图像延迟（4路）	1.0s~2.0s	小于0.5s	2.0s~断开
低负载切换延迟	小于0.1s	小于0.2s	1~2个GOP 约为2s
高负载切换延迟（4路）	小于0.1s	0.2s~0.5s	5~20个GOP 约为14s
带宽占用(1080P)	4.0Mbps * 相机数量	4.5Mbps	2.0Mbps~4.0Mbps
前端兼容性	HTML5	HTML5	需要flash 兼容性差
画质对比	清晰 不流畅	清晰 流畅	较清晰 流畅
中断续流	强	强	小中断强 大中断弱
网络抖动影响（稳定性）	无影响	无影响	一旦GOP延迟就断开

测试环境 Ubuntu 18.04-LTS 7700k 16g ddr4 2600MHz 无显卡 Chrome 70+

(理论上使用更高频率的内存可以让v2架构的延迟更低)

结果展示

- 高负载首帧等待、图像延迟、切换延迟展示（视频）
- 网络波动与中断续流展示（视频）
- 稳定性展示（视频）

v1架构与v2架构的对比

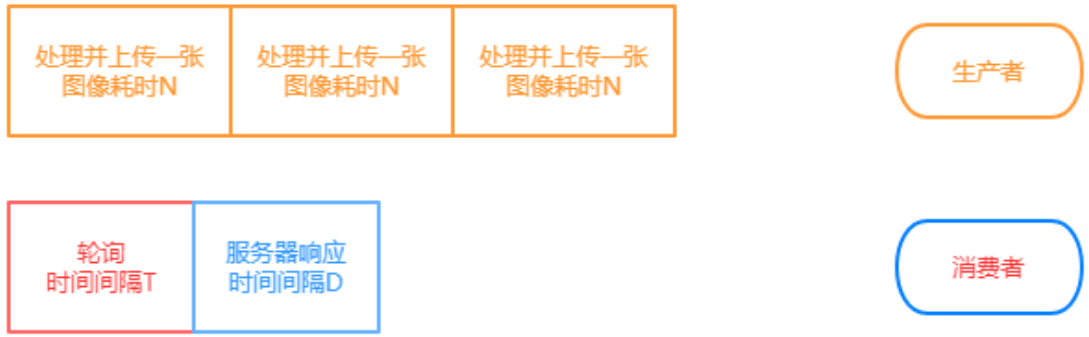
Optional Reading Section

Optional：v1架构与v2架构的对比

- V2架构的主要修改：
  - 1、服务器进程通信方式修改为本机socket（快速 通用）
  - 2、前后端交互方式由HTTP轮询修改为web-socket（低延迟 节约资源）
  - 3、添加memcached用于全局缓存current_ip
    - 实现自动切换报警画面
    - 实现上传前逻辑判断
    - 显著节省带宽
  - 4、修改前端绘制流程
    - 本地缓冲解码更稳定
    - 减少请求服务器次数

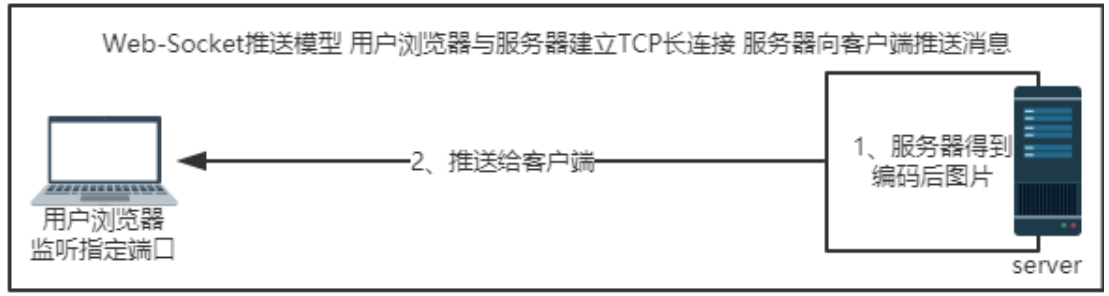
Optional：延迟降低与CPU占用降低原理

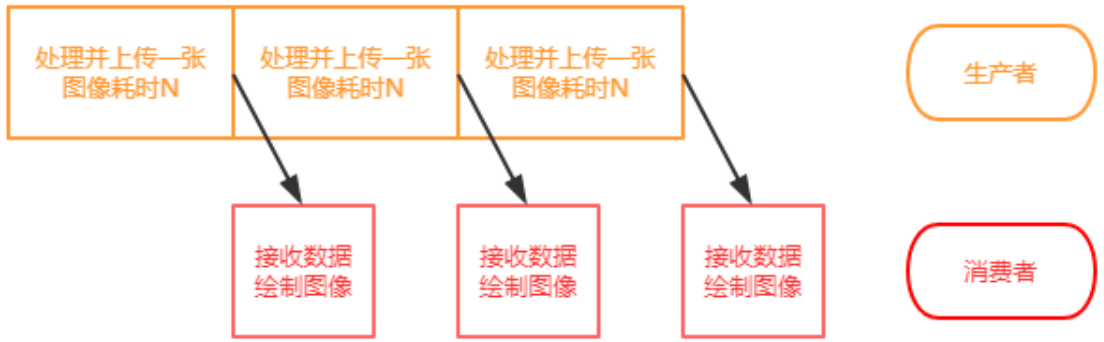
- HTTP轮询模型



由上述定义不难发现 HTTP轮询模型是一个消息不对等的生产者消费者模型
在实际应用中 为了保证生产者的所有数据都被接收 需要保证T+D<=N/2
显然这造成了大量的CPU运算资源和总线I/O资源的浪费

- Web-Socket推送

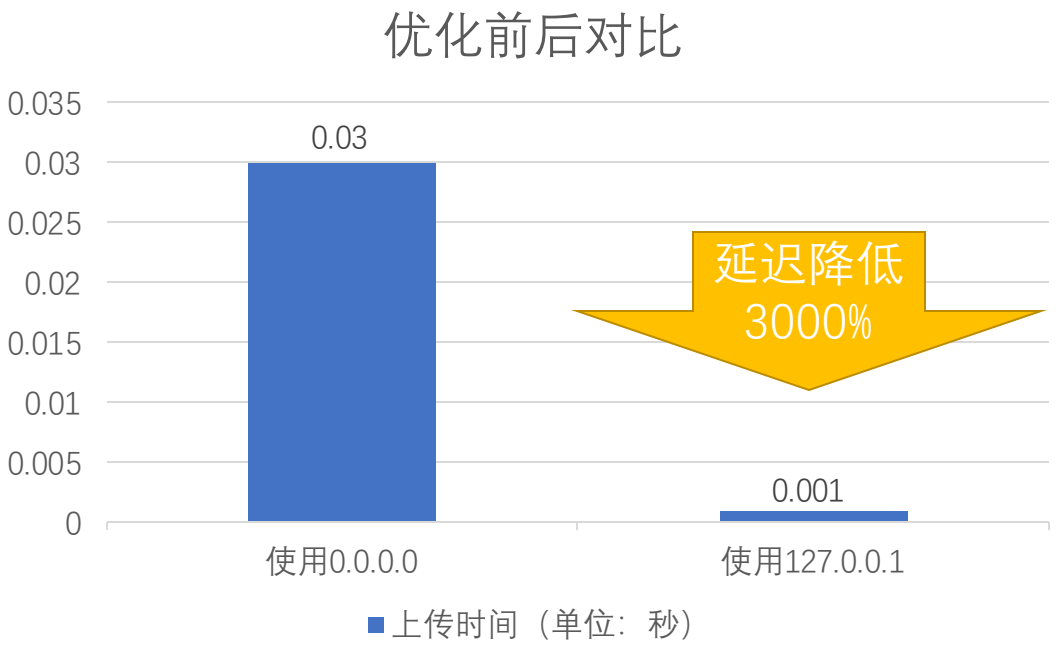


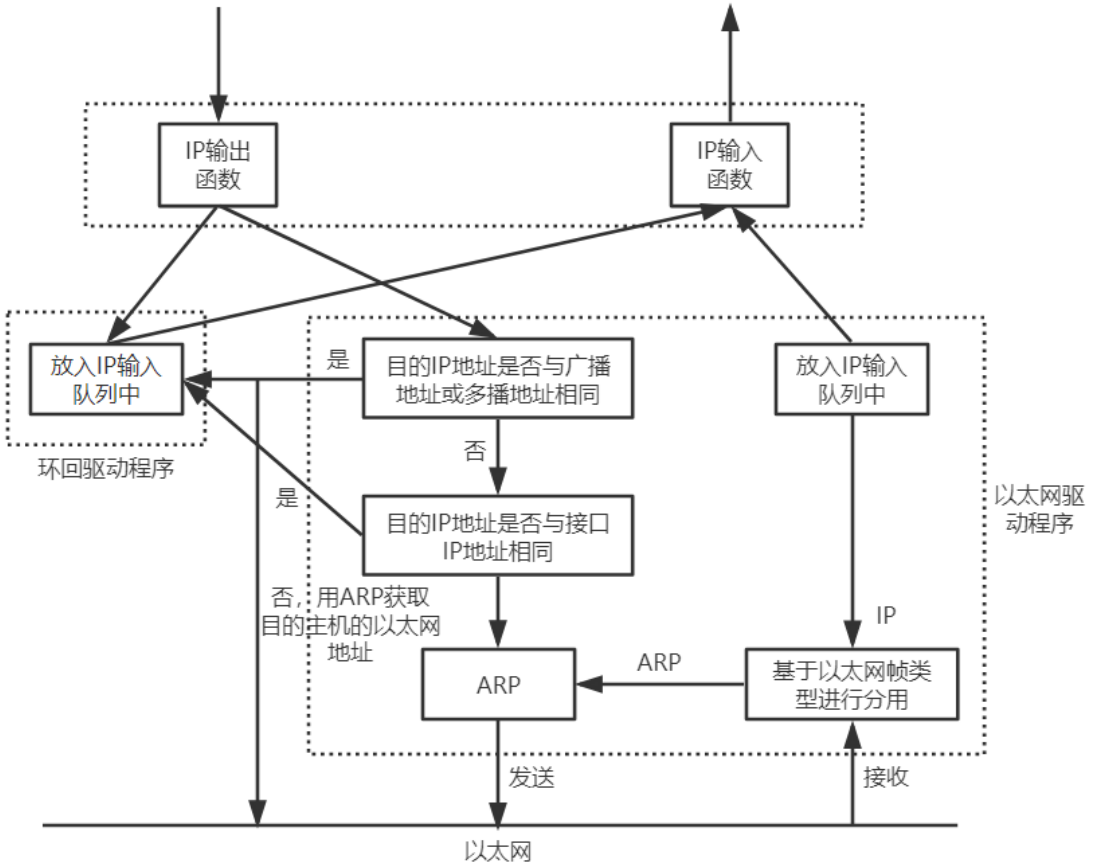


通过使用Web-socket建立前后端TCP长连接（需要支持HTTP1.1协议）
有效解决了生产者和消费者的信息不对称问题
显著节约CPU运算资源和总线I/O资源

Optional：运行效率优化原理

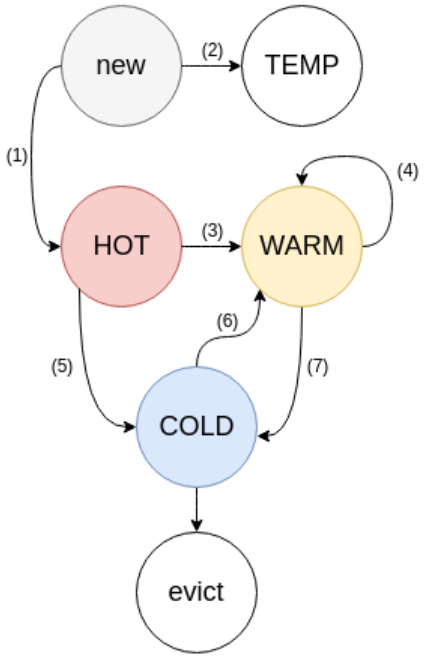
- 网卡驱动本地环回检测器
- 只走虚拟设备 不走物理设备





Optional：加入高速内存缓存

- Memcached（基于Unix domain socket的IPC解决方案）
- 本质是分布式缓存系统（key-value型内存数据库）
- 在本机利用Unix domain socket进行数据传输
- 对以太网使用普通socket进行数据传输
- 缓存使用Segmented LRU



- Unix domain socket 又叫 IPC(inter-process communication 进程间通信) socket，用于实现同一主机上的进程间通信。
- socket 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX domain socket。
- 虽然网络 socket 也可用于同一台主机的进程间通讯(通过 loopback 地址 127.0.0.1)，但是 UNIX domain socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC 机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。
- UNIX domain socket 是全双工的，API 接口语义丰富，相比其它 IPC 机制有明显的优越性，目前已成为使用最广泛的 IPC 机制，比如 X Window 服务器和 GUI 程序之间就是通过 UNIX domain socket 通讯的。
- Unix domain socket 是 POSIX 标准中的一个组件，linux 系统也是支持它的。

Special Thanks

- Stack Overflow
- Memcached
- Python-Memcached
- Socket.io
- Flask-Socket.io