mediaserver架构

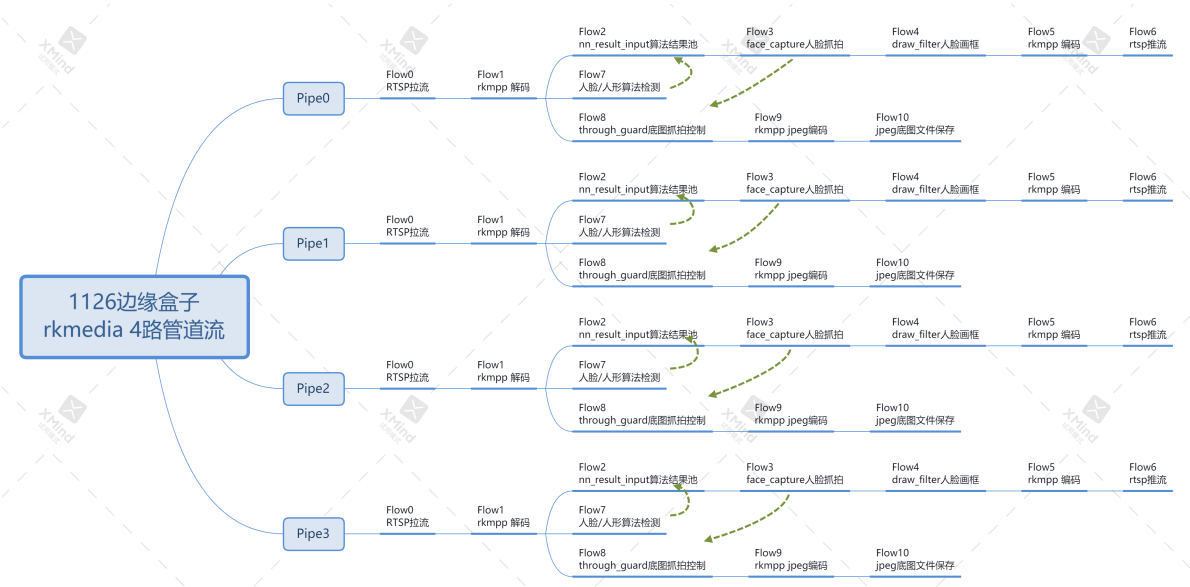
一、管道框架和概述

mediaserver概述

根据应用需求创建若干个rkmedia中的功能组件，按照指定的顺序使视频流从源(摄像头或者rtsp拉流)流向目的地(文件、rtsp推流等)，中途可能经过若干个其他的组件，比如AI算法、编码、解码、录像、抓拍等。同时接收视频流处理事件(如算法结果)或者其他程序的事件(如启/停算法)，提供dbus远程调用接口供其他进程进行参数的设置或者处理结果的保存等。

管道架构

以V12边缘计算盒子为例



配置文件概述

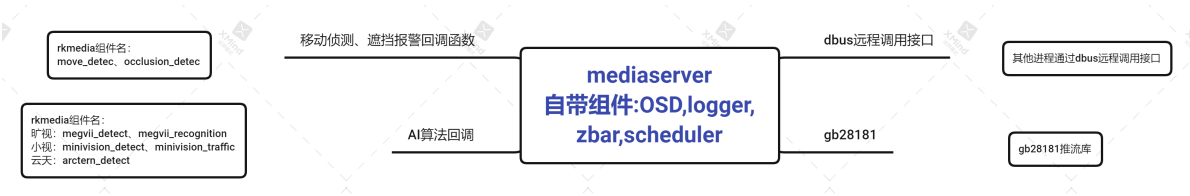
配置文件参考设备上/oem/usr/share/mediaserver/rv1109/rtsp_vdec_nn_rtsp.conf或者固件SDK中app/mediaserver/src/conf/rv1109/rtsp_vdec_nn_rtsp.conf

该配置文件是一个json文件。

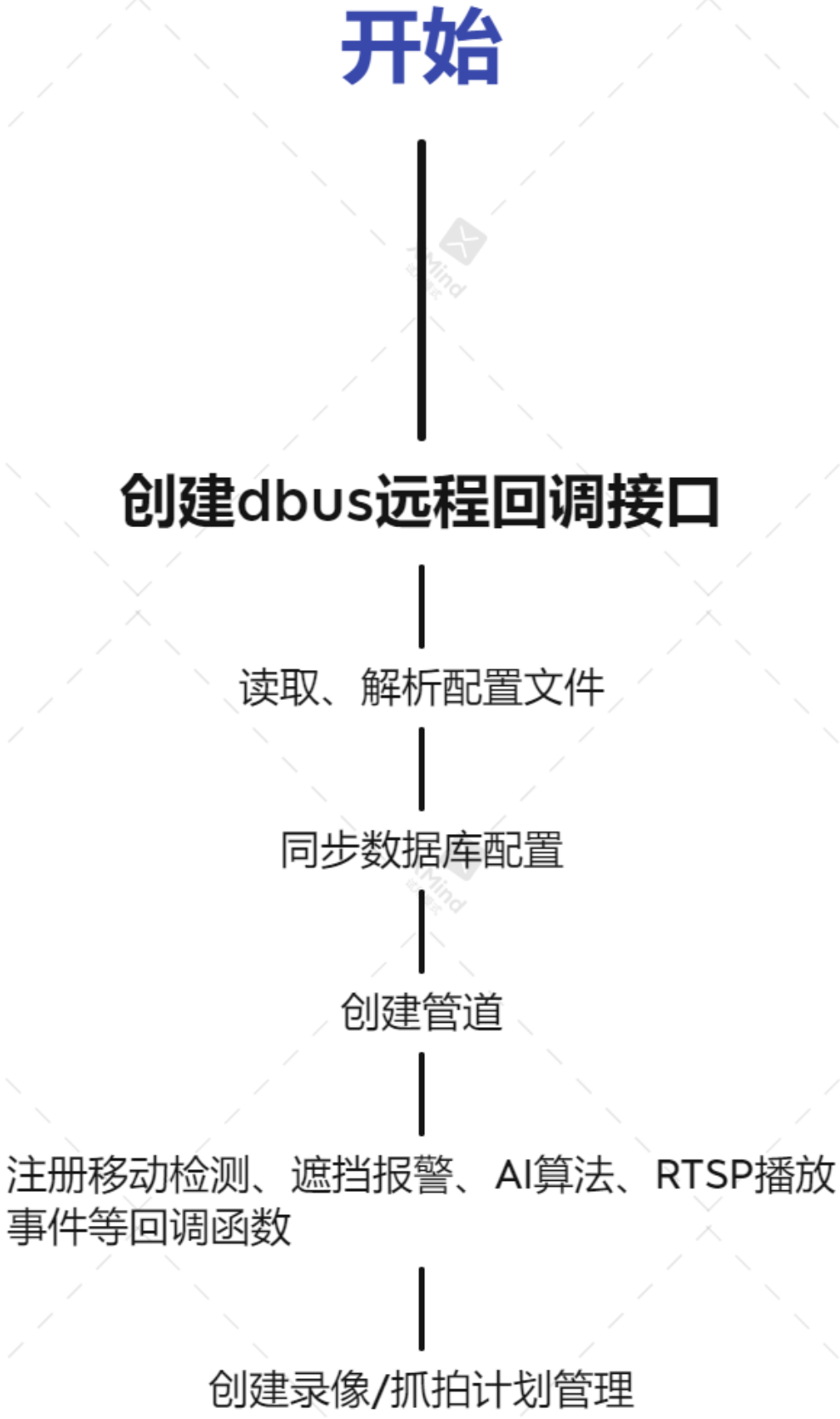
管道：Pipe_n表示管道n。每一条管道至少包含一条线程(获取源视频流)。

Flow：每条管道有若干个Flow，每一个Flow都对应一个rkmedia组件。比如rtsp拉流的Flow名字是rtsp_source_stream，是rkmedia里面一个同名的组件（详细描述见rkmedia架构文档）。

应用架构


rkmedia组件名：
move_detec、occlusion_detec
rkmedia组件名：
旷视：megvii_detect、megvii_recognition
小视：minivision_detect、minivision_traffic
云天：arctern_detect
移动侦测、遮挡报警回调函数
AI算法回调
mediaserver
自带组件:OSD,logger,
zbar,scheduler
dbus远程调用接口
gb28181
其他进程通过dbus远程调用接口
gb28181推流库

运行流程


开始
创建dbus远程回调接口
读取、解析配置文件
同步数据库配置
创建管道
注册移动检测、遮挡报警、AI算法、RTSP播放事件等回调函数
创建录像/抓拍计划管理

二、架构解析

2.1 创建dbus远程调用接口

在src/dbus目录下，rockchip.mediaserver.control.conf文件在打包固件时将被推到/oem/usr/share/dbus-1/system.d目录，dbus在系统启动时会检索这个目录，将里面的*.conf文件解析，注册相关dbus接口。系统的其他应用的dbus配置文件在/etc/dbus-1/system.d。dbus远程接口声明可通过*.xml文件进行声明，参考src/dbus/control/dbus_media_control.xml

```
int MediaServer::RegisterDbusProxy(FlowManagerPtr &flow_manager) {
  if (need_dbus) {
    //创建数据库远程调用接口
    flow_manager->RegisterDBserverProxy(dbus_server_->GetDBserverProxy());
    //创建数据库事件远程调用接口(可参考dbserver dbus接口的定义)
    flow_manager->RegisterDBEventProxy(dbus_server_->GetDBEventProxy());
    //创建存储管理dbus远程调用接口
    flow_manager->RegisterStorageManagerProxy(dbus_server_->GetStorageProxy());
    //创建Ispserver远程调用接口
    flow_manager->RegisterIspserverProxy(dbus_server_->GetIspserverProxy());
  }
  return 0;
}
```

2.2 解析配置文件

mediaserver.cpp: MediaServer::MediaServer() 中

```
flow_manager->ConfigParse(media_config);
```

FlowParser是一个管理json配置文件的类。这个类可以解析、更新、保存json配置文件。

Flow json配置分析：

```
"Flow_4": {
            "flow_index": {
                "flow_index_name": "megvii_detect_0",
                "flow_type": "io",
                "in_slot_index_of_down": "0",
                "out_slot_index": "0",
                "stream_type": "filter",
                "upflow_index_name": "video_dec_0"
            },
            "flow_name": "filter",
            "flow_param": {
                "input_model": "dropfront",
                "name": "megvii_detect",
                "thread_model": "asynccommon"
            },
            "stream_param": {
                "blur_threshod": "0.1",
                "db_path": "/userdata/face.db",
                "detect_fake": "1",
                "detect_quality": "0",
                "enable": "1",
```

```
                "enable_face_detect": "0",
                "input_data_type": "image:nv12",
                "pipe_index": "0",
                "rect_h": "20",
                "rect_w": "20",
                "score_threshod": "0.7",
                "timeout": "-1"
            }
}
```

”flow_index“中的参数指名了当前Flow的名字(可随意命名)、类型、流类型(stream_type)、上级Flow的名字。其中，上级Flow的名字("upflow_index_name")指定了这个Flow的视频帧的来源，即上级Flow处理完一帧数据后，会将这一帧数据传递给这个Flow。一个Flow可以是多个Flow的上级Flow。

“flow_name”指定了这个Flow的名字，不可随意命名。规则如下：

```
源：source_flow
解码：video_dec
编码：video_enc
rtsp推流：live555_rtsp_server
AI算法、抓拍、画框等：filter
保存抓拍图、录像文件：file_write_flow
读取文件：file_read_flow
```

"flow_param"指明了这个Flow的帧输入模式、运行方式（同步、线程异步）、子类名称(实际创建的帧处理组件，有些Flow可能没有)。

```
"input_model":帧输入方式。
    dropfront:如果队列满了，丢弃最早一帧
    dropcurrent:如果队列满了，丢弃当前帧
    blocking:如果队列满了，阻塞
```

"stream_param"里面指定了这个rkmedia组件初始化时的参数，根据不同组件，参数会不一样，实际根据rkmedia对应组件的代码实现配置。

2.3 同步数据库

上述的配置文件只是期望mediaserver启动时进行的参数配置，但实际上程序运行一段时间后，使用者可能会更改其中的一些参数，而这个更改都会保存到数据库中，下一次启动时，数据库中的配置将会覆盖json文件的配置。

flow_manager.cpp

```
int FlowManager::ConfigParse(std::string conf) {
  LOG_INFO("flow manager parse config\n");
  flow_parser_.reset(new FlowParser(conf.c_str()));     //解析json文件
  SyncConfig();     //同步数据库、存储管理，将覆盖从json文件解析出来的配置
  return 0;
}
```

```
int FlowManager::SyncConfig() {
#ifdef ENABLE_DBUS
  SyncDBConfig();   //同步数据库
  SyncSMConfig();   //远程调用存储管理storager_manager接口，同步抓拍、录像等路径
#endif
  SyncFlowParser(); //根据数据库配置，调整其他数据库未记录的Flow参数的值
  return 0;
}
```

同步数据库动作通过dbus远程调用了dbserver的接口，以下代码段列出了一个调用关系

flow_manager.cpp

```
int FlowManager::SyncDBConfig() {
    std::map<std::string, std::string> config_map;
    std::unique_ptr<FlowDbProtocol> db_protocol;
    db_protocol.reset(new FlowDbProtocol);
    for (int id = 0; id < MAX_CAM_NUM; id++) {
        std::string db_config = SelectVideoDb(id);      //dbus远程调用dbserver接口
获取Video配置
        if (db_config.empty()) {
            LOG_INFO("select video database empty\n");
            break;
        }
        int pipe_index = GetPipeIndexById(id, StreamType::CAMERA);  //获取管道
index
        if (pipe_index >= 0) {
            int link_index = flow_parser_->GetFlowIndex(pipe_index,
StreamType::LINK);
            if (link_index >= 0) {
                LOG_INFO("link flow pipe only use video config by conf file\n");
                break;
            }
            db_protocol->DbDataToMap(db_config, config_map);    //std::string数据
转换成std::map<>类型，只是便于解析
            for (auto it : config_map) {
                flow_parser_->SyncVideoDBData(id, it.first.c_str(),
it.second.c_str());     //更改json配置
            }
            config_map.clear();
        }
    }

    ......
}
```

flow_manager.h

```
class FlowManager {
    ......

    std::string SelectVideoDb(int id) {
        std::string empty;
        if (db_server_)
            return db_server_->SelectVideoDb(id);
        return empty;
    }

    .......
}
```

dbus_dbserver.cpp

```
std::string DBusDbServer::SelectVideoDb(int id) {
  char tmp[128];
  sprintf(tmp, DB_SELECT_VIDEO_CMD, id);
  return ProctectCmd(tmp, -1);
}

std::string DBusDbServer::ProctectCmd(char *tmp_cmd, int try_times) {
  std::string rst;
  while (try_times) {
    try {
      rst = Cmd(tmp_cmd);       //Cmd()是dbserver注册到dbus上的一个远程调用接口,在
src/dbus/database/dbus_dbserver.xml中声明
      break;
    } catch (DBus::Error err) {
      LOG_ERROR("%s:try cmd: %s, time:%d, fail\n", __FUNCTION__, tmp_cmd,
try_times);
    }
    try_times--;
  }
  return rst;
}
```

2.4 创建管道

每条管道、每个Flow的参数都准备好后，就可以创建管道了。每条管道中的Flow都是rkmedia中的一个组件(c++类)

```
int FlowManager::CreatePipes() {
  LOG_INFO("flow manager create flow pipe\n");
  /* 逐条管道创建、初始化 */
  for (int index = 0; index < flow_parser_->GetPipeNum(); index++) {
    //新建一条管道
    auto flow_pipe = std::make_shared<FlowPipe>();
    //获取该管道的FlowUnit,FlowUnit保存着json配置
    auto &flow_units = flow_parser_->GetFlowUnits(index);
    //创建所有这条管道的Flow
    flow_pipe->CreateFlows(flow_units);
    //保存管道，运行时配置参数需要用到
    flow_pipes_.emplace_back(flow_pipe);
```

```
  }
  //绑定重复使用的Flow，比如麦克风录音的Flow
  BindReuseFlow();
  //初始化管道，绑定上级flow、注册OSD服务、注册回调函数等
  for (int index = 0; index < flow_pipes_.size(); index++) {
    auto &flow_pipe = flow_pipes_[index];
    //初始化管道，绑定上级flow
    flow_pipe->InitFlows();
// flow_pipe->InitMultiSlice();
#ifdef ENABLE_OSD_SERVER
    //注册OSD服务
    flow_pipe->RegisterOsdServer();
#endif
    flow_pipe->BindControler();
    //注册算法回调函数
    flow_pipe->RegisterCallBack();
  }
  //注册RTSP推流播放事件回调，这里是强制插入一个IDR帧
  RegisterRtspHandler();

  //注册gb28181回调
  #if 1  // jian-bing.guo@telpo.2021-10-26
  RegisterGetVideoStreamCB();
  RegisterGetAudioStreamCB();
  #endif

  //创建算法结果处理，人脸画框处理单元。但这只是中转站，实际抓拍、画框还是会把结果传到rkmedia其他组件完成
#ifdef USE_ROCKFACE
  CreateNNHandler();
#endif
  return 0;
}
```

2.5 创建抓拍/计划录像管理

```
void FlowManager::CreateSchedules() {
  LOG_INFO("Create Schedules Manager\n");
  pthread_mutex_lock(&schedule_mutex);
  if (!schedules_manager_) {
    //新建一个计划管理SchedulesManager
    schedules_manager_.reset(new SchedulesManager());
    if (schedules_manager_) {
        //启动抓拍/录像计划管理线程
      schedules_manager_->start();
    }
  }
  pthread_mutex_unlock(&schedule_mutex);
}
```

三、经常修改的部分

3.1 算法

3.1.1 注册回调函数

算法回调函数的注册，src/flows/flow_pipe.cpp

```
int FlowPipe::RegisterCallBack() {
  auto encoder_flow =
      GetFlow(StreamType::VIDEO_ENCODER, RKMEDIA_STREAM_NAME_RKMPP);
  auto draw_flow = GetFlow(StreamType::FILTER, RKMEDIA_FILTER_DRAW_FILTER);
  if (encoder_flow && draw_flow)
    draw_flow->Control(easymedia::S_NN_DRAW_HANDLER, encoder_flow.get());

  //注册RK人脸检测算法回调
  auto detect_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKFACE_DETECT);
  if (detect_flow)
    detect_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册RK人体检测算法回调
  auto body_detect_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKFACE_BODYDETECT);
  if (body_detect_flow)
    body_detect_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册RK人脸抓拍回调
  auto face_capture_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_FACE_CAPTURE);
  if (face_capture_flow)
    face_capture_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册RK人脸比对算法回调
  auto face_recognize_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKFACE_RECOGNIZE);
  if (face_recognize_flow)
    face_recognize_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册RK Rockx算法回调(未使用Rockx,可忽略)
  auto rockx_flow = GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKX_FILTER);
  if (rockx_flow)
    rockx_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册旷视算法回调
  auto megvii_detect_flow = GetFlow(StreamType::FILTER,
RKMEDIA_FILTER_MEGVII_DETECT);
  if (megvii_detect_flow)
    megvii_detect_flow->Control(easymedia::S_NN_CALLBACK, FlowCallBack);
    //注册郭工移动监测事件、入侵事件、遮挡报警事件回调
  TelpoRegisterMDCallback();
  TelpoRegisterInvadeDetectCallback();
  TelpoRegisterOcclusionDetectCallback();
  return 0;
}
```

```
/**
 * @brief 取消各回调函数的注册
 */
int FlowPipe::UnRegisterCallBack() {
  TelpoUnRegisterMDCallback();
```

```
  TelpoUnRegisterInvadeDetectCallback();
  TelpoUnRegisterOcclusionDetectCallback();
  auto face_capture_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_FACE_CAPTURE);
  if (face_capture_flow)
    face_capture_flow->Control(easymedia::S_NN_CALLBACK, nullptr);
  auto body_detect_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKFACE_BODYDETECT);
  if (body_detect_flow)
    body_detect_flow->Control(easymedia::S_NN_CALLBACK, nullptr);
  auto detect_flow =
      GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKFACE_DETECT);
  if (detect_flow)
    detect_flow->Control(easymedia::S_NN_CALLBACK, nullptr);
  auto encoder_flow =
      GetFlow(StreamType::VIDEO_ENCODER, RKMEDIA_STREAM_NAME_RKMPP);
  auto draw_flow = GetFlow(StreamType::FILTER, RKMEDIA_FILTER_DRAW_FILTER);
  if (encoder_flow && draw_flow)
    draw_flow->Control(easymedia::S_NN_DRAW_HANDLER, nullptr);
  auto rockx_flow = GetFlow(StreamType::FILTER, RKMEDIA_FILTER_ROCKX_FILTER);
  if (rockx_flow)
    rockx_flow->Control(easymedia::S_NN_CALLBACK, nullptr);
  auto megvii_detect_flow = GetFlow(StreamType::FILTER,
RKMEDIA_FILTER_MEGVII_DETECT);
  if (megvii_detect_flow)
    megvii_detect_flow->Control(easymedia::S_NN_CALLBACK, nullptr);
  return 0;
}
```

从上面的注册函数知道，算法结果的回调函数都是FlowCallBack()：

```
/**
 * @brief 算法结果回调函数
 * @param handler: 算法处理对象指针，应根据type来判读是哪一个rkmedia类
 * @param *ptr: 通常是(RknnResult*)，可能是数组。也可能是其他类型的结构体，应根据type的值
来确定
 * @param size: *ptr的元素个数，即*ptr的数组大小。
                实际可能用于其他用途：例如type==NNRESULT_TYPE_FACE_PICTURE_UPLOAD时，
size可能是-1
 */
void FlowCallBack(void *handler, int type, void *ptr, int size) {
  switch (type) {
  //未使用Rockx，可忽略
#ifdef USE_ROCKX
  case NNRESULT_TYPE_OBJECT_DETECT: {
    SetRockXNNResultInput((RknnResult *)ptr, size);
  } break;
#endif

#ifdef USE_ROCKFACE
  //人脸检测结果
  case NNRESULT_TYPE_FACE: {
    //结果传递到nn_result_input（算法结果的缓存，实现算法的异步处理，下一步将结果传递到抓拍、
画框等）
    SetNNResultInput((RknnResult *)ptr, size);
```

```
  } break;
  //人脸识别结果
  case NNRESULT_TYPE_FACE_REG: {
    for (int i = 0; i < size; i++) {
      RknnResult *nn_array = (RknnResult *)ptr;
      RknnResult *nn = &nn_array[i];

      FaceReg *face_reg = &nn->face_info.face_reg;
      //人脸识别
      if (face_reg->type == FACE_REG_RECOGNIZE) {
         //face_reg->user_id>=0表示识别到一个已经录入数据库的人脸，user_id是人脸库的id，
可查阅/userdata/face.db
        if (face_reg->user_id >= 0) {
#ifdef ENABLE_DBUS
          char *similarity_buf = new char[8];
          snprintf(similarity_buf, sizeof(similarity_buf), "%f ",
                   face_reg->similarity);
          //记录人脸识别结果，这里会同步到web的 人脸识别管理->控制记录
          dbserver_control_record_set(face_reg->user_id, (char *)"", (char
*)"Processed",
                                      similarity_buf);
          delete[] similarity_buf;
#endif
          LOG_INFO("recognize user_%d,similarity = %f, pic_path = %s\n",
              face_reg->user_id, face_reg->similarity, face_reg->pic_path);
        } else {
          //有人脸经过，但不是已经录入人脸库的人脸
          LOG_DEBUG("recognize Unknow, similarity = %f, pic_path = %s\n",
              face_reg->similarity, face_reg->pic_path);
        }
      }
      //人脸注册
      else if (face_reg->type == FACE_REG_REGISTER) {
        //人脸注册成功，face_reg->user_id即人脸库入库id，查阅/userdata/face.db
        if (face_reg->user_id >= 0) {
#ifdef ENABLE_DBUS
          //结果同步到系统数据库/userdata/sysconfig.db
          dbserver_face_load_complete_by_path(face_reg->pic_path, 1,
                                              face_reg->user_id);
#endif
          LOG_INFO("register user_%d succeful, pic_path = %s\n", face_reg-
>user_id,
              face_reg->pic_path);
        }
        //重复注册
        else if (face_reg->user_id == -1) {
#ifdef ENABLE_DBUS
          //记录注册失败结果
          dbserver_face_load_complete_by_path(face_reg->pic_path, 2, -1);
#endif
          LOG_INFO("face register repeated\n");
        }
        //注册失败，由于其他原因注册失败
        else if (face_reg->user_id == -99) {
#ifdef ENABLE_DBUS
```

```
          dbserver_face_load_complete_by_path(face_reg->pic_path, -1, -1);
#endif
          LOG_INFO("register user_%d fail, pic_path = %s\n", face_reg->user_id,
              face_reg->pic_path);
        }
      }
      //异常
      else {
        assert(0);
      }
    }
  } break;
#endif
#if (defined(USE_ROCKFACE) && defined(ENABLE_DBUS))
  //人脸抓拍结果
  case NNRESULT_TYPE_FACE_PICTURE_UPLOAD: {
    if (size) {
      //记录抓拍路径到数据库
      SetFacePicUpload((char *)ptr, size);
    }

    //抓拍底图
    //获取管道0。在边缘计算盒子，size参数是管道id，即此处为：
    //      std::shared_ptr<FlowPipe> pipe_n = GetFlowPipe(size,
StreamType::FILTER);
    std::shared_ptr<FlowPipe> pipe_0 = GetFlowPipe(0, StreamType::FILTER);
    if(pipe_0) {
        //获取该管道下的through_guard。through_guard是抓拍控制Flow，允许通过的帧都将压缩
成jpeg并保存起来
        auto through_guard = pipe_0->GetFlow(StreamType::JPEG_THROUGH_GUARD,
RKMEDIA_FILTER_NAME_THROUGH);
        if (through_guard) {
            //允许通过接下来的一帧，即抓拍一张图片
            int count = 1;
            through_guard->Control(easymedia::S_ALLOW_THROUGH_COUNT, &count);
        }
    }
  } break;
#endif
#if (defined(USE_ROCKFACE) && defined(ENABLE_OSD_SERVER))
  //算法授权状态。若未授权，将在视频流中显示"算法未授权水印"；已授权则会去掉这个水印
  case NNRESULT_TYPE_AUTHORIZED_STATUS: {
    SetOsdNNResult((RknnResult *)ptr, type, size);
  } break;
  //RK人体识别结果/算法区域入侵结果，这里表示已经触发区域入侵事件
  case NNRESULT_TYPE_BODY: {
    //显示区域入侵报警水印
    SetOsdNNResult((RknnResult *)ptr, type, size);
    //上报区域入侵事件
    region_invade_event_t invade_event;
    memset(&invade_event,0,sizeof(region_invade_event_t));
    invade_event.info_cnt = size;
    for (int i = 0; i < size; i++) {
      RknnResult *nn_array = (RknnResult *)ptr;
      RknnResult *nn = &nn_array[i];
```

```
      rockface_det_t *body_rect = &nn->body_info.base;
    if (i < MAX_ROI_INVADE) {
        invade_event.data[i].region_id = body_rect->id;
        invade_event.data[i].x = body_rect->box.left;
        invade_event.data[i].y = body_rect->box.top;
        invade_event.data[i].w = body_rect->box.right - body_rect->box.left;
        invade_event.data[i].h = body_rect->box.bottom - body_rect->box.top;
    }

    }
    //将入侵事件传递给gb28181等
    if (invade_event.info_cnt) {
        FlowManagerPtr &flow_manager = FlowManager::GetInstance();
        if (flow_manager) {
            flow_manager->InvadeDetectCallback(&invade_event);
        }
    }
  } break;
#endif
  //人流量统计结果
  case NNRESULT_TYPE_TRAFFIC: {
    // char display_text[128]={0};

    // std::map<std::string, std::string> map;

    RknnResult *nn_array = (RknnResult *)ptr;
    if(nn_array[0].traffic_info.traffic_in > 0 ||
nn_array[0].traffic_info.traffic_out > 0) {
      LOG_DEBUG("new traffic in %d, new traffic out %d\n",
              nn_array[0].traffic_info.traffic_in,
              nn_array[0].traffic_info.traffic_out);
      traffic_in += nn_array[0].traffic_info.traffic_in;
      traffic_out += nn_array[0].traffic_info.traffic_out;
      //人流量结果记录进数据库
      dbserver_traffic_statistic_set(traffic_in, traffic_out);
    }

    //显示人流量水印，这里占用了channel水印通道，使用时应该把channel水印关闭
    // sprintf(display_text, "流入 %d 人, 流出 %d 人", traffic_in, traffic_out);
    // map.emplace(DB_OSD_ENABLED, "1");
    // map.emplace(DB_OSD_POSITION_X, "10");
    // map.emplace(DB_OSD_POSITION_Y, "10");
    // map.emplace(DB_OSD_DISPLAY_TEXT, display_text);
    // map.emplace(DB_OSD_FRONT_COLOR, "0xff8f1f");
    // map.emplace(DB_OSD_FONT_SIZE, "32");
    // SetOsdRegion(OSD_DB_REGION_ID_CHANNEL, map);
  } break;
  default:
    break;
  }
}
```

3.1.2 关键部分

```
case NNRESULT_TYPE_FACE: {
    SetNNResultInput((RknnResult *)ptr, size);
  } break;
```

SetNNResultInput()会把算法结果送到nn_result_input进行算法结果缓存。

注意在flow_nn_handler.cpp，

void NNHandler::NNResultInput(RknnResult *result, int size);

会进行人脸大小的过滤，即配置文件"draw_filter" Flow中"stream_param"的"min_rect": "0"，"min_rect"表示人脸面积大小(人脸框长*宽)，参数是"0"表示不进行人脸大小过滤。

```
void NNHandler::NNResultInput(RknnResult *result, int size) {
    ......
    /**
     * draw_handlers_中保存了nn_result_input对象的指针，在
FlowManager::CreateNNHandler()中可以找到，
        DrawHandler()在构造时传进来的参数flow就是nn_result_input对象指针
     */
    for (auto &iter : draw_handlers_) {
        ...
        //把过滤后的算法结果传递到nn_result_input
        iter->flow_->SubControl(easymedia::S_NN_INFO, fix_result, size);
        ...
    }

    ......
}


int FlowManager::CreateNNHandler() {
  nn_handler_ = std::make_shared<NNHandler>();
  for (int i = 0; i < flow_pipes_.size(); i++) {
    auto flow_unit = flow_pipes_[i]->GetFlowunit(StreamType::FILTER,
                                                 RKMEDIA_FILTER_NN_INPUT);
    if (!flow_unit)
      continue;
    auto &flow = flow_unit->GetFlow();
    if (!flow)
      continue;
    auto reference_flow_unit =
        flow_pipes_[i]->GetFlowunit(StreamType::VIDEO_ENCODER);
    if (!reference_flow_unit)
      reference_flow_unit = flow_pipes_[i]->GetFlowunit(StreamType::CAMERA);
    if (!reference_flow_unit)
      continue;
    std::string width, height;
    reference_flow_unit->GetResolution(width, height);
    int w = atoi(width.c_str());
    int h = atoi(height.c_str());
    auto &draw_handlers = nn_handler_->GetDrawHandler();
    draw_handlers.emplace_back(new DrawHandler(flow, w, h));
```

```
  }
  return 0;
}
```

在配置文件中可以看到，nn_result_input的下一级Flow就是face_capture，然后是draw_filter。所以nn_result_input就是为人脸抓拍、人脸画框提供人脸、人形、物等算法识别结果的。

3.1.3 算法的开/关

（1）启动时同步数据库。mediaserver在初始化时会同步一次数据库，/userdata/sysconfig.db中表SmartCover，iFaceEnabled和iFaceRecognitionEnabled控制了人脸检测和人脸识别开关。

flow_manager.cpp

```
int FlowManager::SyncDBConfig() {
    for (int id = 0; id < MAX_CAM_NUM; id++) {
        //获取数据库SmartCover表格信息
        std::string smart_cover_db = SelectSmartCoverDb(id);
        LOG_ERROR("select smart_cover_db %s\n", smart_cover_db.c_str());
        if (smart_cover_db.empty() ||
            smart_cover_db.find(DB_MEDIA_TABLE_ID) == std::string::npos) {
          LOG_DEBUG("select smart_cover_db id %d empty\n", id);
        } else {
          //转换成std::map，方便处理
          db_protocol->DbDataToMap(smart_cover_db, config_map);
          for (auto it : config_map) {
            //修改flow_parser中对应的json配置
            flow_parser_->SyncSmartCoverDBData(id, it.first.c_str(),
                                               it.second.c_str());
          }
          config_map.clear();
        }
    }
}
```

（2）运行时，其他进程调用dbus远程接口。

src/dbus/control/dbus_feature_control.cpp

int32_t SetFaceDetectEn() 的dbus接口声明在src/dbus/control/dbus_media_control.xml

其他进程调用SetFaceDetectEn()远程接口时

```
int32_t DBusFeatureControl::SetFaceDetectEn(const int32_t &id,
                                            const int32_t &enable) {
  LOG_INFO("DBusFeatureControl::SetFaceDetectEn id %d enable %d\n", id, enable);
#if (defined(USE_ROCKFACE))
  int ret = 0;
  easymedia::FaceDetectArg fda;
  ret = GetFaceDetectArg(id, fda);
  if (ret < 0)
    return ret;
  fda.enable = enable;
  SetFaceDetectArg(id, fda);

  easymedia::NNinputArg nia;
```

```
  ret = GetNNInputArg(id, nia);
  if (ret < 0)
    return ret;
  nia.enable = enable;
  SetNNInputArg(id, nia);
#endif
  return 0;
}
```

3.1.4 适配新算法

(1) 注册回调函数。见"3.1.1 注册回调函数"

(2) 同步数据库。见"2.3 同步数据库"。在同步SmartCover时同步算法的开关

(3) 同步dbus接口。以下接口按需求适配

flow_export.cpp

```
//获取人脸检测开关
int GetFaceDetectArg(int id, easymedia::FaceDetectArg &fda);
//设置人脸检测开关，网页的人脸检测开关可触发
int SetFaceDetectArg(int id, easymedia::FaceDetectArg fda);

//获取人脸识别开关
int GetFaceRegArg(int id, easymedia::FaceRegArg &cfa);
//设置人脸识别开关，网页的人脸识别开关可触发
int SetFaceRegArg(int id, easymedia::FaceRegArg fda);

#define RKMEDIA_FILTER_ROCKFACE_RECOGNIZE "rockface_recognize"
//注册人脸照片，网页注册人员可触发
void SetImageToRecognize(const int32_t &id, const std::string &path) {
  std::string stream_name = RKMEDIA_FILTER_ROCKFACE_RECOGNIZE;  //rk自带的人脸识别
Flow
  //历遍所有管道
  for (int i = 0; i < GetFlowPipeNum(); i++) {
    //获取人脸识别类的对象
    auto flow_pipe = GetFlowPipe(i, StreamType::FILTER, stream_name);
    if (flow_pipe) {
      auto flow_unit = flow_pipe->GetFlowunit(StreamType::FILTER, stream_name);
      auto flow = flow_pipe->GetFlow(StreamType::FILTER, stream_name);
      if (flow && flow_unit) {
        easymedia::FaceRegArg arg;
        arg.type = easymedia::USER_ADD_PIC;
        snprintf(arg.pic_path, sizeof(arg.pic_path), "%s", path.c_str());
        //注册人脸图片
        flow->Control(easymedia::S_NN_INFO, &arg);
      }
      break;
    }
  }
}

//删除一个人脸数据库
void DeleteFaceInfoInDB(const int32_t &id, const int32_t &faceId);
//清空人脸库
```

```
void ClearFaceDBInfo();
```

(4) 适配回调函数

flow_pipe.cpp

```
//见3.1.1 注册回调函数
void FlowCallBack(void *handler, int type, void *ptr, int size);
```