

rkmedia代码解析

一、目录架构

一级目录

.git	2022/7/11 星期一 17:...	文件夹	
cmake	2022/3/10 星期四 14:...	文件夹	
examples	2022/3/10 星期四 14:...	文件夹	demo示例
include	2022/3/10 星期四 14:...	文件夹	头文件
src	2022/6/20 星期一 17:...	文件夹	源码
.clang-format	2022/3/10 星期四 14:...	CLANG-FORMAT...	1 KB
.editorconfig	2022/3/10 星期四 14:...	EDITORCONFIG ...	1 KB
.gitignore	2022/3/10 星期四 14:...	GITIGNORE 文件	1 KB
CMakeLists.txt	2022/6/20 星期一 17:...	文本文档	6 KB
Config.in	2022/3/10 星期四 14:...	IN 文件	2 KB
easymedia_test.sh	2022/3/10 星期四 14:...	Notepad++ Doc...	2 KB
format.sh	2022/3/10 星期四 14:...	Notepad++ Doc...	1 KB
LICENSE	2022/3/10 星期四 14:...	文件	2 KB
rkmedia.rvmk	2022/3/10 星期四 14:...	RVMK 文件	4 KB

二级目录(src)

c_api	2022/6/20 星期一 17:...	文件夹	c库接口 RK MPI
client	2022/3/10 星期四 14:...	文件夹	socket接口 客户端
filter	2022/3/10 星期四 14:...	文件夹	音频主动降噪、回声消除
flow	2022/6/20 星期一 17:...	文件夹	基础Flow: 音视频编解码、文件读写、移动侦测、遮挡检测、源、输出
guard	2022/3/10 星期四 14:...	文件夹	帧通过控制Flow: 常搭配抓拍, 在需要时允许通过一帧, 编解码成jpeg并保持
live555	2022/3/10 星期四 14:...	文件夹	rtsp推流服务: 集成gb28181推流
ogg	2022/3/10 星期四 14:...	文件夹	ogg音频编解码
rkaudio	2022/3/10 星期四 14:...	文件夹	rk音频编解码库
rkmpv	2022/6/20 星期一 17:...	文件夹	rkmpv视频编解码库
rknn	2022/7/5 星期一 11:01	文件夹	图像分析处理: AI算法、抓拍
rkrqa	2022/7/5 星期一 17:24	文件夹	rkrqa图像处理
server	2022/3/10 星期四 14:...	文件夹	socket接口服务端
socket	2022/3/10 星期四 14:...	文件夹	socket接口库
stream	2022/6/20 星期一 17:...	文件夹	视频流源、音频流源Flow
thirdparty	2022/6/20 星期一 17:...	文件夹	第三方算法库: 旷视、小视、云天等
uv	2022/3/10 星期四 14:...	文件夹	
am_capture_stream.cc	2022/6/20 星期一 17:...	Notepad++ Doc...	2 KB
buffer.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	19 KB
CMakeLists.txt	2022/6/20 星期一 17:...	文本文档	6 KB
codec.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	9 KB
decoder.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	1 KB
demuxer.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	1 KB
encoder.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	3 KB
filter.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	1 KB
flow.cc	2022/6/20 星期一 17:...	Notepad++ Doc...	33 KB
image.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	7 KB
libeasymedia.pc.cmake	2022/3/10 星期四 14:...	CMAKE 文件	1 KB
libminimedia.pc.cmake	2022/3/10 星期四 14:...	CMAKE 文件	1 KB
lock.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	2 KB
media_config.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	28 KB
media_type.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	2 KB
message.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	3 KB
muxer.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	1 KB
sound.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	3 KB
stream.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	4 KB
utils.cc	2022/3/10 星期四 14:...	Notepad++ Doc...	10 KB

二、Flow

2.1 Flow基类

Flow是一个处理单元。可以配置成同步、异步模式、缓存溢出处理方式(阻塞或丢弃)。几乎所有的rkmedia类都继承于Flow。这些子类都可以指定同步或异步运行模式、缓存溢出处理方式等。

//继承Flow的类。这个Flow或Flow的子类都可以指定"input_model"、"thread_model"

音频: audio_dec、audio_enc

视频: video_dec、video_enc

文件: file_read_flow、file_write_flow

算法处理: filter、继承filter的src/rknn目录下的类和src/thirdparty目录下的第三方算法、移动侦测、遮挡检测

图像处理: rkrga

音频处理: AEC、ANR

rtsp推流: live555

连接物联网平台: link_flow

音、视频源: source_stream

视频保存: output_stream

下面是一个的file_write_flow的json配置: 保存单帧jpeg图片

```
"Flow_10": {
    "flow_index": {
        "flow_index_name": "sink_0",           //自定义的flow名称
        "flow_type": "sink",                  //flow类型
        "in_slot_index_of_down": "0",
        "out_slot_index": "0",
        "stream_type": "file",                 //流类型
        "upflow_index_name": "video_enc_1"     //上一级
    },
    "flow_name": "file_write_flow",           //Flow的名字，对应
    "flow_param": {                           //不同的Flow参数不一样，具体可以看该
        "pipe_index": "0",                    //管道id
        "file_prefix": "main",                 //文件名前缀
        "file_suffix": ".jpeg",               //文件名后缀
        "input_model": "dropfront",            //帧缓存溢出的处理方式，丢弃
        "mode": "w+",                           //fopen的mode参数，若存在，
        "path": "/userdata/media/photo0",      //文件保存路径
        "save_mode": "single_frame",           //保存模式: 单帧
        "thread_model": "asynccommon"          //线程模式: 异步(线程运行)
    },
    "stream_param": {}
}
```

是"video_enc_1"jpeg编码

src/flow/file_flow.cc, FilewriteFlow::GetFlowName()的返回值

Flow的构造函数

最前一帧

则文件长度清零再写入

```
/*
 * @brief file_write_flow的构造函数。json配置文件中的一些参数被传到这里
 * @param param: 对应json配置文件中的"flow_param"
 */
FilewriteFlow::FilewriteFlow(const char *param) : file_index(0) {
    std::map<std::string, std::string> params;
    if (!parse_media_param_map(param, params)) {
        SetError(-EINVAL);
        return;
    }
    std::string s;
```

```

std::string value;
// "file_prefix": "main"
file_prefix = params[KEY_FILE_PREFIX];
if (file_prefix.empty()) {
    RKMEDIA_LOGI("FilewriteFlow will use default path\n");
    CHECK_EMPTY_SETERRNO(value, params, KEY_PATH, EINVAL)
    path = value;
} else {
    // "path": "/userdata/media/photo0"
    file_path = params[KEY_PATH];
    // "file_suffix": ".jpeg"
    file_suffix = params[KEY_FILE_SUFFIX];
    path = GenFilePath();
}
// "save_mode": "single_frame"
save_mode = params[KEY_SAVE_MODE];
if (save_mode.empty())
    save_mode = KEY_SAVE_MODE_CONTIN;

CHECK_EMPTY_SETERRNO(value, params, KEY_OPEN_MODE, EINVAL)
// "path": "/userdata/media/photo0"
PARAM_STRING_APPEND(s, KEY_PATH, path);
// "mode": "w+"
PARAM_STRING_APPEND(s, KEY_OPEN_MODE, value);
// "save_mode": "single_frame"
PARAM_STRING_APPEND(s, KEY_SAVE_MODE, save_mode);
fstream = REFLECTOR(Stream)::Create<Stream>("file_write_stream", s.c_str());
if (!fstream) {
    fprintf(stderr, "Create stream file_write_stream failed\n");
    SetError(-EINVAL);
    return;
}

SlotMap sm;
sm.input_slots.push_back(0);
sm.thread_model = Model::ASYNCCOMMON;           // 默认异步方式
sm.mode_when_full = InputMode::DROPPRONT;       // 缓存溢出处理方式
sm.input_maxcachenum.push_back(0);              // 最大缓存帧数
sm.process = save_buffer;                       // 帧处理函数

// 配置Flow参数
if (!InstallSlotMap(sm, "FilewriteFlow", 0)) {
    RKMEDIA_LOGI("Fail to InstallSlotMap for FilewriteFlow\n");
    return;
}
// 设置Flow标签, RKMEDIA_LOG_xxx()打印时会打印这个标签
SetFlowTag("FilewriteFlow");
}

```

每个Flow在创建时，都要调用Flow::InstallSlotMap()来指定这个Flow的同步模式、缓存溢出处理方式和帧处理函数，数据帧在传递时会调用这个Flow的帧处理函数，完成这个Flow对一帧数据的处理。

filter_flow.cc

```

FilterFlow::FilterFlow(const char *param)
: support_async(true), thread_model(Model::NONE),
  input_pix_fmt(PIX_FMT_NONE) {

    .....

    SlotMap sm;
    int input_maxcachenum = 2;
    ParseParamToSlotMap(params, sm, input_maxcachenum);
    //同步模式
    if (sm.thread_model == Model::NONE)
        sm.thread_model =
            !params[KEY_FPS].empty() ? Model::ASYNCATOMIC : Model::SYNC;
    thread_model = sm.thread_model;
    //缓存溢出处理方式
    if (sm.mode_when_full == InputMode::NONE)
        sm.mode_when_full = InputMode::DROPCURRENT;
    //创建filter_name对应的Filter对象, filter_name就是json配置文件中"flow_param"中的
    "name", 如"face_capture"
    int input_idx = 0;
    for (auto &param_str : separate_list) {
        auto filter =
            REFLECTOR(Filter)::Create<Filter>(filter_name, param_str.c_str());
        if (!filter) {
            RKMEDIA_LOGI("Fail to create filter %s<%s>\n", filter_name,
                param_str.c_str());
            SetError(-EINVAL);
            return;
        }
        filters.push_back(filter);
        sm.input_slots.push_back(input_idx++);
        sm.input_maxcachenum.push_back(input_maxcachenum);
    }
    sm.output_slots.push_back(0);
    auto &hold = params[KEY_OUTPUT_HOLD_INPUT];
    if (!hold.empty())
        sm.hold_input.push_back((HoldInputMode)std::stoi(hold));
    //帧处理函数
    sm.process = do_filters;
    //log标签
    std::string tag = "FilterFlow: ";
    tag.append(filter_name);
    //配置Flow参数
    if (!InstallSlotMap(sm, tag, -1)) {
        RKMEDIA_LOGI("Fail to InstallSlotMap for %s\n", tag.c_str());
        SetError(-EINVAL);
        return;
    }
    .....
}

```

Flow在创建之后，还要调用Flow::AddDownFlow()来添加下级Flow，数据帧在本级Flow处理完后就会传递到下级Flow（优先传递给同步模式运行的Flow）。

比如mediaserver中

flow_pipe.cpp

```
//初始化一个Flow
int FlowPipe::InitFlow(int flow_index) {
    .....
    for (auto name : v) {
        int upflow_index = GetFlowIndex(name);
        auto &upflow = flow_units_[upflow_index]->GetFlow();
        //添加下级Flow
        upflow->AddDownFlow(flow, in_slot_index_of_down, out_slot_index);
        LOG_DEBUG("flow_name %s upflow_name %s in_slot %d out_slot %d\n",
                  flow_index_name.c_str(), name.c_str(), in_slot_index_of_down,
                  out_slot_index);
        out_slot_index++;
    }
    .....
}
```

完成上述配置后，只要源Flow有数据帧，这些数据帧就会自动按照事先配置好的管道顺序逐级传递下去。

2.2 源Flow

源Flow(source_stream)是一条管道的开始，它从摄像头、麦克风或RTSP拉流获取视频流、音频流，逐帧传递给下级Flow。

src/flow/source_stream_flow.cc

```
/**
 * @brief 源flow的构造函数
 * @param param: json配置文件中的"flow_param"
 */
SourceStreamFlow::SourceStreamFlow(const char *param)
    : loop(false), read_thread(nullptr) {
    std::list<std::string> separate_list;
    std::map<std::string, std::string> params;
    if (!ParseWrapFlowParams(param, params, separate_list)) {
        SetError(-EINVAL);
        return;
    }
    // 需要创建的source_stream的名称，如"name": "v4l2_capture_stream"，对应
    src/stream/camera/v4l2_capture_stream.cc
    // V4L2CaptureStream::GetStreamName()的返回值
    std::string &name = params[KEY_NAME];
    const char *stream_name = name.c_str();
    const std::string &stream_param = separate_list.back();
    //创建一个Stream类的源stream，见src/stream目录，有不同类型的源stream，他们可以从不同的来
    源获取源数据，如v4l2、文件、rtsp等
    stream = REFLECTOR(Stream)::Create<Stream>(stream_name, stream_param.c_str());
    if (!stream) {
        RKMEDIA_LOGI("Create stream %s failed\n", stream_name);
        SetError(-EINVAL);
        return;
    }
}
```

```

}
//log标签
tag = "SourceFlow:";
tag.append(name);
// SetAsSource()是Flow基类的方法，里面调用了InstallSlotMap()，设置了源Flow的默认配置
if (!SetAsSource(std::vector<int>({0}), void_transaction00, tag)) {
    SetError(-EINVAL);
    return;
}
//创建源Flow运行线程
loop = true;
read_thread = new std::thread(&SourceStreamFlow::ReadThreadRun, this);
if (!read_thread) {
    loop = false;
    SetError(-EINVAL);
    return;
}
//log标签
SetFlowTag(tag);
}

```

源Flow有独立的一条线程，不断获取数据帧。

```

void SourceStreamFlow::ReadThreadRun() {
    //设置线程名
    prctl(PR_SET_NAME, this->tag.c_str());
    source_start_cond_mtx->lock();
    if (waite_down_flow) {
        if (down_flow_num == 0 && IsEnable()) {
            source_start_cond_mtx->wait();
        }
    }
    source_start_cond_mtx->unlock();
    //开始循环
    while (loop) {
        if (stream->Eof()) {
            // TODO: tell that I reach eof
            SetDisable();
            break;
        }
        //从stream中读取一帧
        auto buffer = stream->Read();
#ifdef RKMEDIA_TIMESTAMP_DEBUG
        if (buffer)
            buffer->TimeStampReset();
#endif // RKMEDIA_TIMESTAMP_DEBUG
        //把数据帧送到下一级Flow
        SendInput(buffer, 0);
    }
}

```

2.3 其他的Flow

filter_flow、encoder_flow、decoder_flow等等，和源Flow类似，也是继承于Flow。注意区分Flow和由Flow创建的Stream、Decoder、Encoder等：

比如2.1 源Flow的构造函数中，"v4l2_capture_stream"是由source_flow创建的Stream，它不是一个Flow。"v4l2_capture_stream"才是从v4l2获取视频流的类

```
//创建一个Stream类的源stream，见src/stream目录，有不同类型的源stream，他们可以从不同的来源
获取源数据，如v4l2、文件、rtsp等
stream = REFLECTOR(Stream)::Create<Stream>(stream_name, stream_param.c_str());
if (!stream) {
    RKMEDIA_LOGI("Create stream %s failed\n", stream_name);
    SetError(-EINVAL);
    return;
}
```

三、自定义插件

第三方算法、rtsp拉流，或者需要用其他的rtsp推流方式，都需要自定义一个rkmedia插件（flow、stream等）。

3.1 自定义源Stream

典型的例子就是V12分支中，src/stream/rtsp目录下，自定义了两个rtsp拉流的stream。

(1) 继承Stream类

(2) 自定义Stream名字。提供一个公共方法 static const char *GetStreamName(); 返回这个stream的名字。创建stream的时候会调用这个方法，名字匹配才会创建

```
static const char *GetStreamName() { return "rtsp_source_stream"; }
```

(3) 重写Open()、Close()、std::shared_ptr Read()方法，其他的虚函数按需求可以不重写。

(4) 注册自定义插件

```
//注册到Stream类型的factories中
DEFINE_STREAM_FACTORY(RTSPSourceStream, Stream)

const char *FACTORY(RTSPSourceStream)::ExpectedInputDataType() {
    return nullptr;
}

const char *FACTORY(RTSPSourceStream)::OutPutDataType() { return TYPE_ANYTHING;
}
```

3.2 自定义Filter

以接入第三方算法为例

(1) 继承Filter类

(2) 自定义Stream名字。提供一个公共方法 `static const char *GetStreamName();` 返回这个stream的名字。创建Filter的时候会调用这个方法，名字匹配才会创建

(3) 重写`IoCtrl()`方法，需要回调函数的话可以声明一个私有成员`RknnCallBack callback_`，通过调用`IoCtrl()`设置回调函数。

同步运行方式：重写`Process()`方法

异步运行方式：重写`SendInput()`和`FetchOutput()`方法。`SendInput()`输入未处理的帧，`FetchOutput()`输出已处理的帧

```
// sync call, input and output must be valid
virtual int Process(std::shared_ptr<MediaBuffer> input,
                   std::shared_ptr<MediaBuffer> &output);

// some filter may output many buffers with one input.
// sync or async safe call, depends on specific filter.
virtual int SendInput(std::shared_ptr<MediaBuffer> input);
virtual std::shared_ptr<MediaBuffer> FetchOutput();
```

(4) 注册自定义插件

```
//注册到Filter类型的factories中
DEFINE_COMMON_FILTER_FACTORY(MEGFaceDetect)

const char *FACTORY(MEGFaceDetect)::ExpectedInputDataType() {
    return TYPE_ANYTHING;
}

const char *FACTORY(MEGFaceDetect)::OutPutDataType() { return TYPE_ANYTHING; }
```