Rockchip Linux Rockit开发指南

文件标识：RK-KF-YF-532

发布版本：V0.8.0

日期：2020-10-28

文件密级：□绝密 □秘密 □内部资料 ■公开

免责声明

本文档按"现状"提供，瑞芯微电子股份有限公司（"本公司"，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

"Rockchip"、"瑞芯微"、"瑞芯"均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

前言

Rockit定位于通用媒体pipeline, 将常用媒体组件插件化，以积木化的方式构建灵活的应用pipeline。开发者借助Rockit可以开发丰富的媒体应用。

概述

本文主要描述了Rockit媒体开发参考。主要介绍Rockit的应用开发接口；媒体pipeline的拓扑关系；媒体插件的类型和参数；自定义插件的开发等。

产品版本

芯片名称	内核版本
RV1126/RV1109	Linux 4.19

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本号	作者	修改日期	修改说明
V0.6.1	程明传	2020-09-11	rockit特性和接口
V0.6.2	许丽明	2020-09-11	如何开发rockit插件和应用
V0.7.0	程明传	2020-09-15	完善rockit应用开发接口
V0.8.0	方兴文	2020-10-28	UVC应用定制与部分接口完善

目录

---

rockit 的特性

rockit框架的具有以下特性：

- 稳定的操作接口抽象。
- 稳定的媒体接口抽象。将平台媒体接口转换为通用媒体接口。
- 稳定的插件抽象。
- 支持通用插件管理(TaskGraph)。插件组装，数据传递和控制等。
- 支持多种媒体应用开发。

rockit框架当前支持的媒体插件：

当前稳定的插件见下表，**插件名的宏定义**见RTNodeCommon.h中的定义。

特别说明: 插件的参数配置实例非常多，请参考SDK中的配置

文件名称：${rv1126_sdk}/external/rockit/sdk/conf/aicamera_rockx.json

文件名称：${rv1126_sdk}/external/rockit/sdk/conf/aicamera_stasteria.json

类型	插件名	插件作用	输入类型	输出类型
Device	rkisp	RK-ISP	N/A	multi(NV12 etc)
Device	alsa_capture	RK-alsa	N/A	PCM
Device	alsa_playback	RK-alsa	PCM	N/A
Codec	rkmpp_dec	RK-MPP编码器	NV12	H264/H265/MJPEG
Codec	rkmpp_enc	RK-MPP解码器	H264/H265/MJPEG	NV12
Filter	alg_3a / alg_anr	RK-Audio3A-Old	PCM	PCM
Filter	skv_aec / skv_agc / skv_bf	RK-Audio3A-Skv	PCM	PCM
Filter	resample	Audio-Resample	PCM	PCM
Filter	rkrga	RK-RGA 2D加速	multi	multi
Filter	rockx	RK-NPU AI加速	multi(优选NV12)	RTResult
Filter	st_asteria	ST Asteria AI加速	multi(优选NV12)	STResult
Filter	rkeptz	EPTZ复合插件	multi(优选NV12)	RTResult

rockit应用的开发接口

插件(RTTaskNode)的接口

- 适用范围：软件工程师开发插件
- 最佳实践：框架不支持的插件才需要开发，尽量不要自行开发插件。
- 相关文件：RTTaskNode.h

插件的基类是 RTTaskNode。将功能模块(如ISP/RGA/MPP/Rockx)封装到插件(RTTaskNode)之后，通用的TaskGraph能够灵活的调用这些插件，按照配置文件，构建多种应用场景。插件开发中, 我们需要了解以下接口。

```
// 输入参数：context: 节点上下文，用于存放节点初始化所需的各种参数。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。失败后将不能成功创建节点。
// 函数功能：完成节点的初始化工作。
virtual RT_RET RTTaskNode::open(RTTaskNodeContext *context);

// 输入参数：context: 节点上下文，用于存放节点处理所需的各种参数，输入/输出数据。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：完成节点数据处理工作。RT_RET: RT_OK为执行成功，其他为失败。
virtual RT_RET RTTaskNode::process(RTTaskNodeContext *context);

// 输入参数：context: 节点上下文，用于存放节点反初始化所需的各种参数
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：完成节点反初始化工作
virtual RT_RET RTTaskNode::close(RTTaskNodeContext *context);

// 输入参数：meta: 用户的配置参数
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：用户配置节点参数
virtual RT_RET RTTaskNode::invoke(RtMetaData *meta);
```

和RTTaskNode相关的类包括：InputStreamManager, OutputStreamManager, InputStreamHandler, OutputStreamHandler和RTTaskNodeContext。

任务图(TaskGraph)接口

- 适用范围：软件工程师开发应用
- 最佳实践：调用任务图(TaskGraph)接口开发应用，避免利用插件层接口自行管理插件和数据流。
- 相关文件：RTTaskGraph.h

任务图的基类是RTTaskGraph。任务图(RTTaskGraph)的主要作用如下：

- 管理插件系统的配置文件和初始化。
- 管理插件系统的数据流，包括：数据输入、数据输出和数据传递。
- 管理插件系统的控制流，包括：init/prepare/start/stop等。
- 管理插件系统的任务执行。

插件的运行机制的原理如下：Scheduler从TaskNode提取原子任务，并将任务提交SchedulerQueue，执行器(Executor)从SchedulerQueue去原子任务并调度到具体的执行器(如: ThreadPool)。插件任务的执行，可以被看做是一个黑箱；不用关心运行过程，关注RTTaskGraph的输入和输出即可。

任务图(RTTaskGraph)是面向应用开发的类。应用开发中, 我们需要了解以下接口。。

```
// 输入参数：configFile: json配置文件
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：根据用户配置生成TaskGraph
RT_RET   RTTaskGraph::autoBuild(const char* configFile);

// 输入参数：覆盖配置文件中的参数配置
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
```

```
// 函数功能：准备内部运行资源
RT_RET   RTTaskGraph::prepare(RtMetaData *params = RT_NULL);

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：启动任务图运行
RT_RET   RTTaskGraph::start();

// 输入参数：默认为阻塞，将会等待停止运行。非阻塞通常用于异步操作，配合waitUntilDone
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：停止任务图运行
RT_RET   RTTaskGraph::stop(RT_BOOL block = RT_TRUE);

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：恢复任务图运行
RT_RET   RTTaskGraph::resume();

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：暂停任务图运行
RT_RET   RTTaskGraph::pause();

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：清除任务图内部数据
RT_RET   RTTaskGraph::flush();

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：退出时释放任务图内部资源
RT_RET   RTTaskGraph::release();

// 输入参数：cmd: 见枚举定义；params：cmd的额外参数。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：自定义扩展API，根据用户配置节点参数
RT_RET   RTTaskGraph::invoke(INT32 cmd, RtMetaData *params);

// 输入参数：streamName，流的名字；streamId，流的ID号；streamCallback，流的回调。
// 输出参数：NONE
// 函数功能：观察指定的输出流
RT_RET   RTTaskGraph::observeOutputStream(const std::string& streamName,
              INT32 streamId,
              std::function<RT_RET(RTMediaBuffer *)> streamCallback);

// 输入参数：streamId，流的ID号。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：取消观察指定的输出流
RT_RET   RTTaskGraph::cancelObserveOutputStream(INT32 streamId);

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：等待观察者的输出。若要做成同步输出时，才需调用此接口
RT_RET   RTTaskGraph::waitForObservedOutput();

// 输入参数：NONE
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：等待运行结束，主要用于MAIN函数阻塞不退出
```

```
RT_RET   RTTaskGraph::waitUntilDone();
```

和RTTaskGraph相关的类包括：TaskNode、 Scheduler、SchedulerQueue、Executor、ThreadPool。上述这些类是内部类，对于开发rockit应用来说不用关心。

RTTaskGraph的控制操作

RTTaskGraph读取插件配置文件之后，会自动构建Pipeline, 然后自动工作。通过少数几个接口，即可完成复杂数据流的协同Pipeline处理。应用开发者仅需要关注插件配置和RTTaskGraph即可开发复杂应用。

```
RTTaskGraph *graph = new RTTaskGraph();
graph->autoBuild("your_graph.json");
// 准备pipeline, 包括：插件准备、链接输入和输出等
graph->prepare();
// 启动pipeline，打通整个数据流
graph->start();
// 停止pipeline
graph->stop();

// 设置pipleline的参数
// 配置框架的参数: GRAPH_CMD_PRIVATE_CMD
// 配置节点的参数: GRAPH_CMD_TASK_NODE_PRIVATE_CMD,)
RtMetaData params;
params.setInt32(kKeyTaskNodeId, nodeId);
params.setCString(kKeyPipeInvokeCmd, "qp-control");
params.setInt32("qp_init",  24);
params.setInt32("qp_step",  8);
params.setInt32("qp_min",   4);
params.setInt32("qp_max",   24);
params.setInt32("min_i_qp", 4);
params.setInt32("max_i_qp", 24);
graph->invoke(GRAPH_CMD_TASK_NODE_PRIVATE_CMD, &params);
```

监听TaskGraph数据输出的接口

应用开发者需要关注RTTaskGraph的输入和输出。RTTaskGraph的输入/数据源插件(NodeIsp/NodeDemuxer)是自驱动的输入的(下级消耗数据)，关注这类插件的配置即可。RTTaskGraph的Sink插件是自驱动输出(本级消耗数据)，关注这类插件的配置即可。RTTaskGraph的Filter插件(AIVisionFilter)一般做中间处理，这类插件作为输出，需要应用开发者将插件的数据(AIVisionData)取走做额外的后处理。

```
// 自定义的OBSERVER后处理函数，一般用于应用程序和ROCKIT的输出数据对接。
RT_RET YOUR_OBSERVER_FUNC(RTMediaBuffer *buffer) {
    // 以下示例：获取AI检测结果
    RTAIDetectResults *aiResult  = RT_NULL;
    buffer->getMetaData()->findPointer(OPT_AI_DETECT_RESULT,
                          reinterpret_cast<RT_PTR *>(&aiResult));
    if (!strcmp(aiResult->vendor, "rockx")) {
        RTRknnAnalysisResults *result = (RTRknnAnalysisResults *)aiResult-
>privData;
    } else {
        OtherAnalysisResults *result = (OtherAnalysisResults *)aiResult-
>privData;
    }
```

```
    buffer->release();
    return RT_OK;
}

RTTaskGraph *graph = new RTTaskGraph();
graph->autoBuild("your_ai_vision.json");
// 准备pipeline, 包括: 插件准备、链接输入和输出等
graph->preare();
// 观察pipeline的输出
graph->observeOutputStream("ai_rockx", ${stream_id} << 16, YOUR_OBSERVER_FUNC);
// 启动pipeline，打通整个数据流
graph->start();
// 其他操作....
```

关键数据流发生变化时的处理流程

分辨率变化

@TODO

数据格式变化

@TODO

核心插件参数说明

特别说明: 插件的参数配置实例非常多，请参考SDK中的配置

文件名称：${rv1126_sdk}/external/rockit/sdk/conf/aicamera_rockx.json

文件名称：${rv1126_sdk}/external/rockit/sdk/conf/aicamera_stasteria.json

插件RKISP

参考上述SDK中的配置文件。

插件RKRGA

参考上述SDK中的配置文件。

插件RKROCKX

参考上述SDK中的配置文件。

插件RKMPP

参考上述SDK中的配置文件。

插件UAC相关

参见文档：linux的SDK中的《Rockchip Linux UAC App开发指南》。

如何开发rockit插件

下列使用一个demo来介绍图配置的结构

```
// 创建外部节点， 外部节点需要继承RTTaskNode
// 基础接口需要完成open/process/close
class RTRockitDemoNode : public RTTaskNode {
```

```
 public:
    RTRockitDemoNode() {}
    virtual ~RTRockitDemoNode() {}

    virtual RT_RET open(RTTaskNodeContext *context) { return RT_OK; }
    virtual RT_RET process(RTTaskNodeContext *context);
    virtual RT_RET close(RTTaskNodeContext *context) { return RT_OK; }
};

// 用于节点创建， 该函数指针将存于RTNodeStub.mCreateObj中
static RTTaskNode* createRockitDemoNode() {
    return new RTRockitDemoNode();
}

// 节点处理函数，将输入的数据处理， 然后输出到下级节点
RT_RET RTRockitDemoNode::process(RTTaskNodeContext *context) {
    RTMediaBuffer *inputBuffer  = RT_NULL;
    RTMediaBuffer *outputBuffer = RT_NULL;

    // 判断输入是否为空
    if (context->inputIsEmpty()) {
        return RT_ERR_BAD;
    }

    // 取出输入的buffer
    inputBuffer = context->dequeInputBuffer();
    // 取出一块未被使用的输出Buffer
    outputBuffer = context->dequeOutpuBuffer(RT_TRUE, inputBuffer->getLength());
    if (RT_NULL == outputBuffer) {
        inputBuffer->release();
        return RT_ERR_BAD;
    }
    // 将输入的数据拷贝至输出（demo为最简单的拷贝处理）
    rt_memcpy(outputBuffer->getData(), inputBuffer->getData(), inputBuffer-
>getLength());
    // 设置输出buffer的范围
    outputBuffer->setRange(0, inputBuffer->getLength());
    // 标记EOS
    if (inputBuffer->isEOS()) {
        outputBuffer->getMetaData()->setInt32(kKeyFrameEOS, 1);
    }
    // 输入Buffer使用完成，调用释放
    inputBuffer->release();
    // 将输出buffer带出，完成处理流程
    context->queueOutputBuffer(outputBuffer);

    return RT_OK;
}

//节点信息存根， 用于完成节点注册
RTNodeStub node_stub_rockit_demo {
    // 节点uid， 节点的唯一标识符 (0~1000)
    .mUid           = kStubRockitDemo,
    // 节点名， 主要用于节点查找、创建
    // corp_role_name，命名保证唯一
    .mName          = "rockit_demo",
    // 版本号
    .mVersion       = "v1.0",
```

```
    // 节点创建方法; 改成宏定义。
    .mCreateObj     = createRockitDemoNode,
    .mCapsSrc       = { "video/x-raw", RT_PAD_SRC,  {RT_NULL, RT_NULL} },
    .mCapsSink      = { "video/x-raw", RT_PAD_SINK, {RT_NULL, RT_NULL} },
};

// 检测uuid重复，并报错
RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo);
```

节点注册

节点注册需要存根，存根基础信息如下：

```
//节点信息存根， 用于完成节点注册
RTNodeStub node_stub_rockit_demo {
    // 节点uid， 节点的唯一标识符
    .mUid           = kStubRockitDemo,
    // 节点名， 主要用于节点查找、创建
    .mName          = "rockit_demo",
    // 版本号
    .mVersion       = "v1.0",
    // 节点创建方法
    .mCreateObj     = createRockitDemoNode,
    .mCapsSrc       = { "video/x-raw", RT_PAD_SRC,  {RT_NULL, RT_NULL} },
    .mCapsSink      = { "video/x-raw", RT_PAD_SINK, {RT_NULL, RT_NULL} },
};
```

在存根定义的地方全局调用下列函数完成注册

```
RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo);
```

节点的关键函数

```
// 输入参数：context: 节点上下文，用于存放节点初始化所需的各种参数。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。失败后将不能成功创建节点。
// 函数功能：完成节点的初始化工作。
RT_RET open(RTTaskNodeContext *context);

// 输入参数：context: 节点上下文，用于存放节点处理所需的各种参数，输入/输出数据。
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：完成节点数据处理工作。RT_RET: RT_OK为执行成功，其他为失败。
RT_RET process(RTTaskNodeContext *context);

// 输入参数：context: 节点上下文，用于存放节点反初始化所需的各种参数
// 输出参数：RT_RET: RT_OK为执行成功，其他为失败。
// 函数功能：完成节点反初始化工作
RT_RET close(RTTaskNodeContext *context);
```

上下文功能描述

RTTaskNodeContext，存放节点初始化、处理、反初始化所需要的信息。

```
// 存放节点参数。例如编码节点，则有可能存放width,height, bitrate等信息
```

```
RtMetaData* options();

// 输入参数：streamType: 输入包类型。默认为none，在多输入情况下决定了从哪个输入包队列取出数据
// 输出参数：RTMediaBuffer: 输入包RTMediaBuffer。
// 函数功能：获取一块输入包
RTMediaBuffer *dequeInputBuffer(std::string streamType = "none");

// 输入参数:
//      block: 是否阻塞。 默认为阻塞，阻塞时等待有空闲的packet，有则立即返回空闲输出包。
//      size: 需要请求的packet大小，将会返回一块比size大的buffer。
//      streamType: 输出包类型。默认为none，在多输出情况下决定了从哪个输出池中取出空闲
buffer
// 输出参数: RTMediaBuffer, 空闲输出包RTMediaBuffer。
// 函数功能：获取一块空闲的输出包
RTMediaBuffer *dequeOutputBuffer(
                        RT_BOOL block = RT_TRUE,
                        UINT32 size = 0,
                        std::string streamType = "none");

// 输入参数:
//      packet: process产生的输出数据。
//      streamType: 输出包类型。默认为none，在多输出情况下决定了流向哪个下级节点的输入。
// 输出参数: RT_RET: RT_OK为成功，其他为失败。
// 函数功能：存放一块输出包，将会将这块输出包流向下级节点
RT_RET queueOutputBuffer(RTMediaBuffer *packet, std::string streamType =
"none");

// 输入参数: streamType 输入类型。默认为none，在多输入情况下决定判断哪个输入队列为空
// 输出参数: RT_BOOL, RT_TRUE为空，RT_FALSE为非空。
// 函数功能: 判断输入是否为空
RT_BOOL inputIsEmpty(std::string streamType = "none");

// 输入参数: streamType: 输出类型。默认为none，在多输出情况下决定判断哪个输出队列为空
// 输出参数: RT_BOOL: RT_TRUE为空，RT_FALSE为非空。
// 函数功能: 判断输出是否为空
RT_BOOL outputIsEmpty(std::string streamType = "none");
```

如何开发rockit应用

目前我们支撑自动构建和手动构建基于ROCKIT框架的应用程序。自动构建rockit应用是指使用JSON配置文件，自动构建插件并自动构建PIPELINE。手动构建rockit应用是指开发者手动构建插件并手动构建PIPELINE。自动构建rockit应用只需要了解配置项和少数接口即可开发应用程序，推荐使用自动构建rockit应用的方式开发应用程序。开发基于ROCKIT框架的应用程序，主要步骤包括：

- 评估ROCKIT已有插件是否满足应用程序需求，是否需要开发或扩展新的插件。
- 确认插件参数、连接关系和控制关系，然后按照应用程序需求配置TaskGraph的插件配置。
- 应用程序加载JSON配置文件，完成应用程序和ROCKIT的数据流/控制流对接。
- 应用程序的业务逻辑和应用稳定性的测试验证。

应用demo见${SDK_ROOT}/external/rockit

自动构建rockit应用(推荐)

目前我们支撑自动构建和手动构建基于ROCKIT框架的应用程序。自动构建rockit应用是指使用JSON配置文件，自动构建插件并自动构建PIPELINE。自动构建rockit应用接口比较友好，推荐应用程序工程师这种方式开发基于ROCKIT框架的应用程序。

图配置文件实例

```
{
    // 配置一级目录，pipe_0即为一个图，目前不允许在一个配置里配置多张图，所以这里通常为"pipe_0"
    "pipe_0": {
        // 配置二级目录，为节点配置信息
        "node_0": {
            // 配置节点通用信息，目前只有node name
            "node_opts": {
                "node_name"       : "rkisp"
            },
            // 配置节点context信息，例如数据源、输出buffer的类型、个数、大小等
            "node_opts_extra": {
                "node_source_uri" : "/dev/media1",
                "node_buff_type"  : 0,
                "node_buff_count" : 4,
                "node_buff_size"  : 460800
            },
            // 配置数据流通用信息，例如类型、名字等
            "stream_opts": {
                "stream_output"   : "image:nv12_0",
                "stream_fmt_out"  : "image:nv12"
            },
            // 配置节点的个性化信息，如此处将会配置到rkisp节点内
            "stream_opts_extra": {
                "opt_entity_name" : "rkispp_scale1",
                "opt_width"       : 640,
                "opt_height"      : 480,
                "opt_vir_width"   : 640,
                "opt_vir_height"  : 480,
                "opt_buf_type"    : 1,
                "opt_mem_type"    : 4,
                "opt_use_libv4l2" : 1,
                "opt_colorspace"  : 0
            }
        },
        "node_1": {
            "node_opts": {
                "node_name"       : "rkmpp_enc"
            },
            "node_opts_extra": {
                "node_buff_type"  : 0,
                "node_buff_count" : 4,
                "node_buff_size"  : 460800
            },
            "stream_opts": {
                "stream_input"    : "image:nv12_0",
                "stream_output"   : "image:h264_0",
                "stream_fmt_in"   : "image:nv12",
                "stream_fmt_out"  : "image:h264"
            },
            "stream_opts_extra": {
                "opt_width"      : 640,
                "opt_height"     : 480,
                "opt_vir_width"  : 680,
                "opt_vir_height" : 480,
```

```
                "opt_bitrate"    : 1000000,
                "opt_codec_type" : 6,
                "opt_frame_rate" : 30,
                "opt_profile"    : 100,
                "opt_level"      : 52,
                "opt_gop"        : 30,
                "opt_qp_init"    : 24,
                "opt_qp_step"    : 4,
                "opt_qp_min"     : 12,
                "opt_qp_max"     : 48
            }
        },
        "node_2": {
            "node_opts": {
                "node_name"       : "fwrite"
            },
            "node_opts_extra": {
                "node_source_uri" : "/data/test.yuv",
                "node_buff_type"  : 1,
                "node_buff_count" : 0
            },
            "stream_opts": {
                "stream_input"    : "image:h264_0",
                "stream_fmt_in"   : "image:h264"
            }
        }
}
```

图配置参数列表

仅列举通用信息， 非图、节点通用信息请自行在节点代码内部查询。
仅表示当前版本的配置参数信息，但未来版本不限于此。
宏定义名为RTNodeCommon.h中的宏定义

参数名	宏定义名	功能	是否为必要信息	备注
node_name	OPT_NODE_NAME	节点名	是	系统将会通过节点名找到对应节点，完成构建。
node_source_uri	OPT_NODE_SOURCE_URI	数据源	否	节点内部将会取这个值打开文件、设备等读写信息。
node_buff_type	OPT_NODE_BUFFER_TYPE	输出buffer类型	否	为0时表示buffer由节点内部分配；为1时表示buffer由节点外部分配，节点仅提供RTMediaBuffer结构。
node_buff_count	OPT_NODE_BUFFER_COUNT	输出buffer个数	是	为0表示不分配
node_buff_size	OPT_NODE_BUFFER_SIZE	输出buffer大小	否	
stream_input	OPT_STREAM_INPUT_NAME	数据流输入名	否	用于link节点数据流，需要对应上级节点的stream_output。如果在节点内部存在多输入情况，请使用stream_input_ + index方式区分，如stream_input_0。
stream_output	OPT_STREAM_OUTPUT_NAME	数据流输出名	否	用于link节点数据流，需要对应下级节点的stream_input。如果在节点内部存在多输出情况，请使用stream_output_ + index方式区分，如stream_output_0。
stream_fmt_in	OPT_STREAM_FMT_IN	数据流输入类型	否	定义数据输入类型，该类型在节点内部用于判断数据流的类型。 多输入情况需要与stream_input对应，如stream_fmt_in_0。
stream_fmt_out	OPT_STREAM_FMT_OUT	数据流输出类型	否	定义数据输出类型，该类型在节点内部用于判断数据流的类型。 多输入情况需要与stream_output对应，如stream_fmt_out_0。

自动构建应用示例

下面我们描述如何利用上面的配置文件完成图的自动构建

```
#define RT_GRAPH_TRANSCODING_FILE
"/data/file_h264_rkmpp_dec_rkmpp_h265_enc_write.json"

RT_RET unit_test_graph_transcoding(INT32 index, INT32 total) {
    (void)index;
    (void)total;

    // 创建一个空白图
    RTTaskGraph *transcodingGraph = new RTTaskGraph("UVCGraphTranscodingTest");
    // 通过配置文件完成自动化构建
    transcodingGraph->autoBuild(RT_GRAPH_TRANSCODING_FILE);
    // 完成图的准备工作
    transcodingGraph->perpare();
    // 启动图
    transcodingGraph->start();
    // ....
    // 停止图
    transcodingGraph->stop();
    transcodingGraph->release();
    // 销毁图
    rt_safe_delete(transcodingGraph);

    return RT_OK;
}
```

手动构建rockit应用(不推荐)

目前我们支撑自动构建和手动构建基于ROCKIT框架的应用程序。手动构建rockit应用是指开发者手动构建插件并手动构建PIPELINE。自动构建rockit应用只需要了解配置项和少数接口即可开发应用程序，推荐使用自动构建rockit应用的方式开发应用程序。

手动构建rockit示例

```
// 创建空白图
RTTaskGraph *demoGraph = new RTTaskGraph("rockit_demo");
// 通过配置信息完成节点创建
RT_NODE_CONFIG_STRING_APPEND(nodeConfig, XXX, XXX);
RT_NODE_CONFIG_STRING_APPEND(streamConfig, XXX, XXX);
RTTaskNode *freadNode = demoGraph->createNode(nodeConfig, streamConfig);
XXX
......
// 链接节点数据流
demoGraph->linkNode(freadNode, demoNode, fwriteNode);
// 图资源准备
demoGraph->prepare();
// 图开始工作
demoGraph->start();
```

下面介绍详细的步骤

手动创建插件示例

下列使用一个例子介绍如何手动完成节点创建，节点参数的使用方法请参考《4.2 自动构建rockit应用》。

```
// 定义节点配置和数据流配置
```

```
std::string nodeConfig;
std::string streamConfig;
// RT_NODE_CONFIG_STRING_APPEND 加入配置信息
RT_NODE_CONFIG_STRING_APPEND     (nodeConfig, OPT_NODE_NAME,
 NODE_NAME_FREAD);
RT_NODE_CONFIG_STRING_APPEND     (nodeConfig, OPT_NODE_SOURCE_URI,
 "/data/test.h264");
RT_NODE_CONFIG_NUMBER_APPEND     (nodeConfig, OPT_NODE_BUFFER_TYPE,     0);
RT_NODE_CONFIG_NUMBER_APPEND     (nodeConfig, OPT_NODE_BUFFER_COUNT,    4);
RT_NODE_CONFIG_NUMBER_APPEND     (nodeConfig, OPT_NODE_BUFFER_SIZE,     1024 *
1024);
// RT_NODE_CONFIG_NUMBER_LAST_APPEND 加入最后一项配置信息
RT_NODE_CONFIG_NUMBER_LAST_APPEND(nodeConfig, OPT_FILE_READ_SIZE,       1024 *
1024);
RT_NODE_CONFIG_STRING_APPEND     (streamConfig, OPT_STREAM_OUTPUT_NAME,
"fread_out");
RT_NODE_CONFIG_STRING_LAST_APPEND(streamConfig, OPT_STREAM_FMT_OUT,
"image:h264");
// 创建节点
RTTaskNode *freadNode = demoGraph->createNode(nodeConfig, streamConfig);
```

上述创建方法中，OPT_NODE_NAME为节点名， 用于从node factory中查找到对应节点，务必要正确才能完成创建。

手动链接插件示例

链接节点:

```
// 输入参数:
//     srcNode: 链接的上游节点，提供链接中的输出。
//     dstNode: 链接的下游节点，提供链接中的输入。
//         ...: 可支持传入多个节点
// 输出参数: RT_RET, 链接成功返回RT_OK，失败返回具体原因的返回值。
// 函数功能: 完成节点的链接，完成链接后，上游节点process完成的输出数据将会流向下游节点的输入队
列。
//          需要注意的是，节点输入输出类型等信息需要匹配才能完成链接。
RT_RET linkNode(RTTaskNode *srcNode, RTTaskNode *dstNode, ...);
```

取消链接节点:

```
// 输入参数:
//     srcNode: 链接的上游节点，提供链接中的输出。
//     dstNode: 链接的下游节点，提供链接中的输入。
// 输出参数: RT_RET, 链接成功返回RT_OK，失败返回具体原因的返回值。
// 函数功能：取消插件之间的链接关系。
RT_RET   unlinkNode(RTTaskNode *srcNode, RTTaskNode *dstNode);
```

如何扩展现有应用

目前内置支持UVC+NN、UAC应用，UVC与NN应用对应接口RTUVCGraph、RTUACGraph，详见external\rockit\sdk\headers下对应头文件的接口。下面将介绍如何在现有应用上扩展以及重写部分插件。

UVC应用定制

UVC功能裁剪

RTUVCGraph应用支持ZOOM、Pan&Tile、EPTZ、NN可根据项目需求进行功能裁剪，减少内存等资源占用。external\rockit\sdk\conf下有提供以下几种裁剪，可参考进行个性化需求定制。

配置文件名称	功能说明
aicamera_uvc.json	仅包含UVC预览功能
aicamera_rockx_only.json	仅包含Rockx功能
aicamera_uvc_zoom.json	支持UVC、ZOOM、Pan&Tile
aicamera_uvc_zoom_eptz.json	支持UVC、ZOOM、Pan&Tile、EPTZ
aicamera_uvc_zoom_rockx.json	支持UVC、ZOOM、Pan&Tile、Rockx
aicamera_uvc_zoom_eptz_rockx.json	支持UVC、ZOOM、Pan&Tile、EPTZ、Rockx

将上述配置文件重命名为aicamera_rockx.json，即可裁剪覆盖原来支持的功能。

UVC插件重写

当内置的插件不能满足项目需求时，可根据需要自定义插件，替换UVC pipeline中的插件，具体步骤如下：

- 编写自定义插件

  在app\aiserver\src\vendor工程目录下创建自定义插件。该目录下有eptz、rockx等插件示例，可参考实现。新增节点实现可参见**<如何开发rockit插件>**。

- 修改配置文件中插件名

  修改external\rockit\sdk\conf\aicamera_rockx.json，将要修改的插件名称替换成自定义插件名称。

通过以上两个步骤后，重写编译固件即可完成UVC插件自定义。当节点间链接关系需要发生变化时，可参考如下**<配置节点链接>**说明

配置节点链接

配置文件中的节点配置可参见**<图配置文件实例>**，节点之间的链接关系既支持配置文件自动链接(通过上级的节点的stream_output与下级节点的stream_input匹配链接)，也支持通过NodeID直接指定上下级链接方式(可用于复杂的场景节点链接)。通过以下示例来说明指定式链接方式。

示例一：单链接

```
"link_0": {
    "link_name"          : "uvc",
    "link_ship"          : "0,7"
},
```

link_name: 该场景功能是UVC，代码中选择开启UVC来根据此名字查找对应链接关系，所以名字不可变。

link_ship: "0,7" 表示节点0与节点7建立链接，即节点0的数据流向节点7。

示例二：多链接

```
"link_1": {
    "link_name"          : "nn_isp",
    "link_ship"          : "2,8-8,11,1000-8,12,1000-8,13,1000"
},
"link_2": {
    "link_name"          : "nn_linkout",
    "link_ship"          : "10,8-8,11,1000-8,12,1000-8,13,1000"
}
```

link_name: 该场景功能是NN算法，"nn_isp"表示NN算法数据源来自isp(节点2是ispp scale1)，同样名字不可变；"nn_linkout"也是NN算法功能，与"nn_isp"的区别在于数据源是来自节点10(EPTZ裁剪、缩放等处理后的图像数据)。

link_ship: "2,8-8,11,1000-8,12,1000-8,13,1000"，链接是以”-“分割表示链接分组，”,“分割表示一组链接。即”2,8“表示节点2链接节点8；”8,11,1000“表示节点8链接节点11，节点11链接节点1000；”8,12,1000“表示节点8链接节点12，节点12链接节点1000；节点11、12、13分别表示人脸、人脸特征点、骨骼算法，也可以用单个节点实现所有NN算法时，此时可以合并NN节点，如按以下配置可修改为stasteria算法，对应节点11也需配置为stasteria节点。

```
"link_1": {
    "link_name"          : "nn_isp",
    "link_ship"          : "2,8,11,1000"
},
"link_2": {
    "link_name"          : "nn_linkout",
    "link_ship"          : "10,8,11,1000"
}
```

UAC应用定制(新增Audio 3A插件)

- 明确UAC功能需求
- 评估哪些媒体插件可以复用，哪些媒体插件需要开发。
- UAC的音频3A插件，参考Rockchip Linux UAC相关文档。
- 明确UAC的pipeline的插件的拓扑结构，并定义配置文件。
- 新增Audio 3A插件，按照自定义媒体插件的方式实现。