文件编号：FO-20240607-X-0

SOPHON-STREAM 插件开发规范

修订记录

版本号	修订日期	作者	修订内容
1.0	2024.06.07	徐逸舟	初始制作。

目录

第一章 总则

第一条 目的

本指南旨在讲解 sophon-stream 中 element 的工作原理及开发测试流程，帮助用户快速开发支持自定义功能的插件。

第二条 适用范围

本文档适用于如下情况：

1) 算能 PCIe 或 SoC 模式的设备，处理器型号包括 BM1684、BM1684X、BM1688、CV186AH
2) sophon-stream 对设备的 SDK 版本有要求。对于 BM1684 和 BM1684X 设备，SDK 版本需要大于等于 V23.03.01；对于 BM1688 和 CV186AH 设备，SDK 版本需要大于等于 V1.5

第三条 引用文件

表 一-1：引用文件列表

序号	引用文件名称	文件编号
1.	HowToMake.md	SDK/sophon-stream/docs
2.	Sophon_Stream_User_Guide.md	SDK/sophon-stream/docs

第四条 术语与定义

表 一-2：术语和定义列表

序号	术语	定义说明
1.	sophon-stream	面向算丰开发平台的数据流处理工具。
2.	element	sophon-stream 框架内的功能单元

第二章 流程介绍

本文用于指导使用者在 sophon-stream 框架中添加具有自定义功能的 element，从而可以快速搭建起满足需求的业务逻辑。

本文档有执行步骤 checklist，可以在配置过程中参考 checklist 进行逐步排查。

第三章 Checklist

Cat.	Check 项目	Check 方法	参考结果	Check 结果	RP &时间
插有算能加速卡的 x86 设备，或算能边缘算力盒子	设备是否存在	检查 PCIE 接口是否已插入算能 PCIE 设备，或边缘设备是否完整无缺损	有		
	设备是否上电	检查算能设备是否灯亮或者风扇是否运作	灯亮或者风扇正常工作		
	驱动是否正常	命令 bm-smi 可以运行，并且设备数量、状态均正常	能		
sophon-stream 开发与编译环境	是否已拉取完整的 sophon-stream 仓库	sophon-stream 目录是否存在，git status 提示无异常	目录存在且状态正常		
	交叉编译环境是否正常	按照/docs/HowToMake.md 文档搭建环境，并尝试编译，观察有无报错	编译无报错		

第四章 添加自定义插件

第五条 插件工作原理介绍

sophon-stream 框架中，element 是实际工作的最小数据结构。所有 element 都是继承自同一个抽象基类 sophon_stream::framework::Element。该基类提供了通用的初始化接口 init()和多线程启停与工作逻辑，也提供了主要的纯虚函数 initInternal()和 doWork()，分别用于自定义初始化和自定义线程函数。

因此，添加自定义的 element，实际上就是添加一个新的派生类。它继承自 sophon_stream::framework::Element 基类，需要实现独有的初始化函数 initInternal()及工作函数 doWork()，以及一些可能的其它自定义功能。

第六条 添加深度学习算法插件

sophon-stream 仓库中提供了快速添加深度学习算法插件的方法。仅需要按照如下命令操作，即可在 sophon-stream/element/algorithm 目录下生成一个由用户自定义算法命名的子目录。

```
cd sophon-stream/element/algorithm/
./algorithm_maker.sh <your_alg_name>
```

其中，<your_alg_name>是用户自定义的插件名。

下文将以 yolov3 算法为例进行说明。

运行./algorithm_make.sh yolov3 命令，将在当前目录下生成 yolov3 子目录，其目录结构为：

```
yolov3
├── CMakeLists.txt
├── include
│   ├── yolov3_context.h # 声明 context 结构，存放算法的只读参数。
│   ├── yolov3.h # Element 的派生类
│   ├── yolov3_inference.h # 声明推理类
│   ├── yolov3_post_process.h # 声明后处理类
│   └── yolov3_pre_process.h # 声明预处理类
├── README.md
└── src
    ├── yolov3.cc # 实现算法 Element 的初始化和工作逻辑
    ├── yolov3_inference.cc # 实现推理功能，一般不需要额外修改
    ├── yolov3_post_process.cc # 实现后处理功能
    └── yolov3_pre_process.cc # 实现预处理功能
```

接下来，需要实现算法的初始化功能、数据处理逻辑以及具体的预处理和后

处理算法。

其中，初始化功能即 yolov3.cc 文件内的 initInternal()函数，该函数以一个 json 字符串为输入，由用户自定义的规则进行解析。一般此处会解析如模型路径、预处理的均值、方差、后处理的阈值等参数。

数据处理逻辑即 yolov3.cc 文件内的 doWork()函数。该函数主要实现了从输入队列弹出数据、凑 batch、处理、push 进输出队列的过程。对于常规的 CV 模型来说，模板代码中的该部分已经可满足需求，一般不需要额外修改。

预处理算法和后处理算法分别在 yolov3_pre_process.cc 和 yolov3_post_process.cc 中实现，该部分实现细节可以参考算法源码和 BMCV 用户手册等文档。

需要注意，由于预处理和后处理一般都是只针对一张或一组图片进行操作，因此原则上不应存在线程冲突，也就不需要使用互斥量进行保护。

1) yolov3_context.h

本节说明 yolov3_context.h 文件中涉及的内容。

yolov3_context.h 文件中应包含一个继承了 sophon_stream::framework::Context 的类，用于定义模型的预处理、推理、后处理相关参数。

使用 algorithm_maker.sh 生成的模板应如下所示：

```
class Yolov3Context : public ::sophon_stream::framework::Context
{
  public: int deviceId; // 设备 ID
  std::shared_ptr<BMNNContext> bmContext;
  std::shared_ptr<BMNNNetwork> bmNetwork;
  bm_handle_t handle; std::vector<float> mean; // 前处理均值， 长度为 3，顺序为 rgb
  std::vector<float> stdd; // 前处理方差，长度为 3，顺序为 rgb
  bool bgr2rgb; // 是否将 bgr 图像转成 rgb 推理
  /** * @brief 最小的置信度阈值。详细说明请参考 README */
  float thresh_conf_min = -1;
  /** * @brief 置信度阈值，key：类名，value：阈值
  * 该参数支持对不同的类别设置不同的阈值 */
  std::unordered_map<std::string, float> thresh_conf;
  /** * @brief NMS IOU 阈值 */
  float thresh_nms; std::vector<std::string> class_names;
  /** * @brief 决定是否启用类别阈值 */
   bool class_thresh_valid = false;
  /** * @brief * 类别数量，从 model 中读取。
  需要和 thresh_conf、class_names 的长度做校验 */
  int class_num = 80;
  int m_frame_h, m_frame_w; int net_h, net_w, m_net_channel;
  int max_batch;
  int input_num;
  int output_num;
  int min_dim;
  bmcv_convert_to_attr converto_attr;
  /** * @brief json 文件中定义的 ROI，若此项生效，则只对 ROI 划定的区域做算法 */
  bmcv_rect_t roi;
  bool roi_predefined = false;
  };
```

上面包含了深度学习算法中大部分常见的参数。由于涵盖范围较广，很多参数并不一定是每个算法都必须的。例如，对于目标检测算法，往往会需要后处理的类别置信度阈值和 NMS 的 IOU 阈值，但对于分类算法（如 Resnet），则不需要这两个参数，可以将其从成员中删去。

2) yolov3. h

本节说明 yolov3.h 文件中应包含的内容。

yolov3.h 文件声明了继承自 sophon_stream::framework::Element 的派生类。其形式如：

```
class Yolov3 : public ::sophon_stream::framework::Element
{
public:
Yolov3();
~Yolov3() override;
const static std::string elementName;
/**
* @brief * 解析 configure，初始化派生 element 的特有属性；调用 initContext 初始化算法相关参数
* @param json json 格式的配置文件
* @return common::ErrorCode
* 成功返回 common::ErrorCode::SUCCESS，失败返回 common::ErrorCode::PARSE_CONFIGURE_FAIL
*/
common::ErrorCode initInternal(const std::string& json) override;
/**
* @brief * element 的功能在这里实现。例如，算法模块需要实现组 batch、调用算法、发送数据等功能
* @param dataPipeId pop 数据时对应的 dataPipeId
* @return common::ErrorCode 成功返回 common::ErrorCode::SUCCESS
*/
common::ErrorCode doWork(int dataPipeId) override;
void initProfiler(std::string name, int interval);
```

```
    std::shared_ptr<::sophon_stream::framework::Context>      getContext()
{ return mContext; }
    std::shared_ptr<::sophon_stream::framework::PreProcess>
getPreProcess() { return mPreProcess; }
    std::shared_ptr<::sophon_stream::framework::Inference>  getInference()
{ return mInference; }
    std::shared_ptr<::sophon_stream::framework::PostProcess>
getPostProcess() { return mPostProcess; }
    /** * @brief 从 json 文件读取的配置项，应按需增删 */
    static                constexpr                const                char*
CONFIG_INTERNAL_MODEL_PATH_FIELD = "model_path";
    private:
    std::shared_ptr<Yolov3Context> mContext; // context 对象
    std::shared_ptr<Yolov3PreProcess> mPreProcess; // 预处理对象
    std::shared_ptr<Yolov3Inference> mInference; // 推理对象
    std::shared_ptr<Yolov3PostProcess> mPostProcess; // 后处理对象
    bool use_pre = false; bool use_infer = false; bool use_post = false;
    std::string mFpsProfilerName;
    ::sophon_stream::common::FpsProfiler mFpsProfiler;
    common::ErrorCode initContext(const std::string& json);
    void process(common::ObjectMetadatas& objectMetadatas);
    };
```

3) yolov3.cc

本节说明 yolov3.cc 文件中应包含的内容。

首先，yolov3.cc 文件中必须实现 initInternal()和 doWork()两个函数。

模板自动生成的内容如下所示：

```
common::ErrorCode Yolov3::initInternal(const std::string& json) {
   common::ErrorCode errorCode = common::ErrorCode::SUCCESS;
    do {
        // json 是否正确
        auto configure = nlohmann::json::parse(json, nullptr, false);
        if (!configure.is_object()) {
        errorCode = common::ErrorCode::PARSE_CONFIGURE_FAIL;
        break;}
        // 判断当前 element 运行算法的预处理/推理/后处理阶段
        auto stageNameIt =
configure.find(CONFIG_INTERNAL_STAGE_NAME_FIELD);
        if (configure.end() != stageNameIt && stageNameIt->is_array()) {
          std::vector<std::string> stages =
                    stageNameIt->get<std::vector<std::string>>();
          if (std::find(stages.begin(), stages.end(), "pre") != stages.end()) {
            use_pre = true;
            mFpsProfilerName = "fps_yolov3_pre";}
          …
          mFpsProfiler.config(mFpsProfilerName, 100); }
         // 新建 context,预处理,推理和后处理对象
         mContext = std::make_shared<Yolov3Context>();
         mPreProcess = std::make_shared<Yolov3PreProcess>();
         mInference = std::make_shared<Yolov3Inference>();
         mPostProcess = std::make_shared<Yolov3PostProcess>();
         if (!mPreProcess || !mInference || !mPostProcess || !mContext) {
           break; }
         mContext->deviceId = getDeviceId();
         // 初始化上一节中定义的 context
         initContext(configure.dump());
         // 前处理初始化
         mPreProcess->init(mContext);
         // 推理初始化
         mInference->init(mContext);
         // 后处理初始化
         mPostProcess->init(mContext); } while (false);
          return errorCode;
        }
```

在这里，算法插件的自定义初始化放在了 initContext()函数中。这样设置与直接在 `initInternal()` 中初始化没有本质上的区别。

initContext()函数形如：

```
common::ErrorCode Yolov3::initContext(const std::string& json)
{
    common::ErrorCode errorCode = common::ErrorCode::SUCCESS;
    do {
    auto configure = nlohmann::json::parse(json, nullptr, false);
    if (!configure.is_object()) {
    errorCode = common::ErrorCode::PARSE_CONFIGURE_FAIL; break; }
    // 从 json 中解析模型路径。解析其它参数可以参考此行代码编写。
    auto modelPathIt
        = configure.find(CONFIG_INTERNAL_MODEL_PATH_FIELD); }
   while (false);
   return common::ErrorCode::SUCCESS;
}
```

这里只演示了从 json 中解析模型路径的部分。实际开发时，应该还需要包含读入模型、解析模型输入输出形状参数等步骤。

doWork()函数的形式也基本固定。代码模板自动生成的 doWork()函数已经包含了一个线程函数应该有的功能：从前一个队列中 pop 数据、处理数据、再向后面的队列 push 数据。因此，这段代码一般不需要用户修改。doWork()函数中提供了 process()函数用于调用实际的预处理/推理/后处理过程，用户可以只专注于该部分的开发。

4) yolov3_pre_process.h 和 yolov3_pre_process.cc

yolov3_pre_process.h 文件包含对预处理类的声明。

```
class Yolov3PreProcess : public ::sophon_stream::framework::PreProcess
{
public:
/** * @brief 对一个 batch 的数据做预处理*/
common::ErrorCode preProcess(std::shared_ptr<Yolov3Context> context,
     common::ObjectMetadatas& objectMetadatas);
void init(std::shared_ptr<Yolov3Context> context);
private:
/** * @brief 为一个 batch 的数据初始化设备内存 */
void initTensors(std::shared_ptr<Yolov3Context> context,
       common::ObjectMetadatas& objectMetadatas);
```

这里只涉及两个函数。preProcess()是算法预处理阶段实际调用的函数，它从

输入的 objectMetadatas 中获取一个 batch 上的图像信息，然后经过处理将其转化为 tensor 送给 NPU 推理。initTensors()函数是对每个 batch 的图像数据的初始化操作。它按照当前 element 初始化阶段获取到的模型信息，为每帧图像申请推理所需的设备内存，并使用智能指针管理其生命周期和析构函数。 目前，initTensors()函数的具体时间已经比较通用，基本不需要特异性的更改。用户只需要在 preProcess()函数中填入自己的预处理逻辑即可完成该模块的开发。

5) yolov3_inference.h 和 yolov3_inference.cc

yolov3_inference.h 文件包含对推理类的声明。

```
class Yolov3Inference : public ::sophon_stream::framework::Inference
{
public:
~Yolov3Inference() override;
/** * @brief init device and engine */
void init(std::shared_ptr<Yolov3Context> context);
/** * @brief network predict output */
common::ErrorCode predict(std::shared_ptr<Yolov3Context> context,
common::ObjectMetadatas& objectMetadatas);
private:
/** * @brief 组合 inputTensor，batchsize==1 时不调用 */
std::shared_ptr<sophon_stream::common::bmTensors>
   mergeInputDeviceMem( std::shared_ptr<Yolov3Context> context,
   common::ObjectMetadatas& objectMetadatas);
/** * @brief 申请 outputTensors * /
std::shared_ptr<sophon_stream::common::bmTensors>
getOutputDeviceMem( std::shared_ptr<Yolov3Context> context);
/** * @brief * 将更新的 outputTensors 分配到每一个 ObjectMetadata 上，
batchsize==1 时不调用 */
void splitOutputMemIntoObjectMetadatas(
std::shared_ptr<Yolov3Context> context,
common::ObjectMetadatas& objectMetadatas,
std::shared_ptr<sophon_stream::common::bmTensors> outputTensors);
};
```

推理模块相关的函数如上所示。其中，外层 Inference()函数直接调用的是 predict()函数。该函数以一个 batch 的数据为输入，先将输入内存转化为连续并为其申请输出内存，然后进行推理，再把推理得到的连续内存分块到 batch 内的各帧图像上。特别地，当 batch_size 为 1 时，将直接申请输出内存并进行推理，省略了转化连续内存和将连续内存分块的操作。

一般来说，推理模块的代码已经具有一定的通用性，用户不需要修改。

6) yolov3_post_process.h 和 yolov3_post_process.cc

yolov3_post_process.h 文件包含对后处理类的声明。

```
class Yolov3PostProcess : public ::sophon_stream::framework::PostProcess
{
public:
void init(std::shared_ptr<Yolov3Context> context);
/**
* @brief 对一个 batch 的数据做后处理
* @param context context 指针
* @param objectMetadatas 一个 batch 的数据
*/
void postProcess(std::shared_ptr<Yolov3Context> context,
   common::ObjectMetadatas& objectMetadatas);
```

如上所示，yolov3_post_process.h 中需要用户自己实现的函数只有 postProcess()一个。该函数以一个 batch 的图片为输入，经过自定义的后处理，获得对应的输出结果，例如检测框的位置、置信度、类别等。

第七条 添加非深度学习的插件

非深度学习的插件和深度学习算法插件本质上是一致的，都符合第五条的描述。

但一般来说，基于深度学习的 CV 算法可以概括性地区分为预处理、推理和后处理三个阶段，因此可以使用一个统一的形式来组织其目录结构。而非深度学习的插件则不一定具有如上所述的阶段。依据插件的具体功能，如跟踪、绘图、编解码、向量召回、发送 http 请求等，其代码组织形式可以相对灵活，因此本仓库没有为非深度学习的插件提供统一的模板。

不过，添加非深度学习的插件时，也可以参考其它插件的基本框架，然后填入自己需要的功能。例如，可以参考仓库中位于/element/tools/blank 目录下的空白插件。这是一个实验性的插件，没有实现具体功能，因此代码行数最少，只 override 了基类的纯虚函数来保证编译可以通过。

用户可以按照该插件的形式，实现自己的初始化及工作逻辑。 具体地，可以参考上文。

第五章 编译及测试

第八条 编译

首先，在 sophon-stream 工程目录下的 CMakeLists.txt 文件中增加 checkAndAddElement(element/algorithm/yolov3)，该命令效果是将 yolov3 插件加入 `sophon-stream` 的编译流程。

然后参考工程目录下的/docs/HowToMake.md 编译即可。 编译完成后，在/build/lib 目录下可以看到 libyolov3.so 文件。

第九条 测试

最后，需要将新添加的插件与其它插件连接起来测试功能。

这里以 yolov3 插件为例。这类深度学习算法插件测试功能的方式一般是相似的，分为以下几步。

7) 编写配置文件

首先应参考 yolov5 例程，新建一个 samples/yolov3 例程的测试目录。然后需要正确编写 yolov3_demo.json，engine.json，decode.json，yolov3.json 四个配置文件，分别对应输入数据、graph 定义、解码插件、yolov3 插件四个部分。

8) 编写绘图函数

绘图函数主要作用是可视化地验证结果正确性。绘图函数位于/samples/include/draw_funcs.h。对于检测任务来说，draw_funcs.h 文件中已经包含了比较通用的绘图函数，可以直接调用，无需重复开发。

如果 draw_funcs.h 中未提供需要的绘图函数，用户自己实现后还应该在/samples/src/main.cc 中指明配置文件中 draw_func_name 字段和绘图函数的关系，以便通过 json 文件调用到预期的绘图函数。

9) 运行，观察结果

正确编写配置文件和绘图函数后，应实际运行来验证结果是否正确。运行方式可以参考/samples 目录下的各个例程，即统一以 main 二进制文件作为入口，由传入的 json 文件决定 graph 如何搭建。例程运行过程中的日志一般包括数据传递、各个插件的 fps 等。

运行结束后，可以在 results 目录下看到保存的图片。

10) 关注插件的安全性

插件的安全性指是否存在内存泄漏等。具体地，可以在程序运行时分别通过 top 命令和 bm-smi 命令观察系统内存和设备内存是否会持续上涨。若有上涨，则说明插件中某处缺少内存释放逻辑，需要排查解决。若非深度学习算法插件，其测试方法与上文基本相同，但配置文件中可能涉及的 element 及其连接规则需要结合插件功能来具体设置。运行结束后，应将结果与预期结果对比，判断功能是否正常。