



## 寒武纪 CNStream 用户手册

版本 7.0.0

2025 年 03 月 20 日

目录	i
插图目录	1
表格目录	2
1 版权声明	3
2 快速入门	5
2.1 安装和配置环境依赖和依赖库	5
2.1.1 环境依赖	5
2.1.2 寒武纪安装包	5
2.1.2.1 Cloud	5
2.1.2.2 Edge	6
2.1.3 Ubuntu 环境下安装和配置	7
2.1.4 Docker 环境下安装和配置	7
2.2 编译 CNStream 指令	8
2.3 CNStream 开发样例	10
2.3.1 样例介绍	10
2.3.1.1 图像分类样例	10
2.3.1.2 目标检测样例	11
2.3.1.3 物体追踪样例	11
2.3.1.4 车辆结构化样例	11
2.3.1.5 姿态检测样例	12
3 概述	13
3.1 简介	13
3.2 CNStream 优势	13
3.3 CNStream 软件栈	14
3.4 CNStream 数据流动	14
4 框架介绍	16
4.1 核心框架	16
4.2 cnstream::Pipeline 类	17

4.3	cnstream::Module 类	18
4.4	cnstream::CNFrameInfo 类	19
4.5	cnstream::EventBus 类	20
4.6	cnstream::Event 类	21
4.7	cnstream::Collection 类	21
<b>5</b>	<b>内置模块</b>	<b>23</b>
5.1	简介	23
5.2	数据源模块	23
5.2.1	使用说明及参数详解	25
5.2.2	SourceHandler 使用说明	25
5.2.2.1	FileHandler	25
5.2.2.2	RtspHandler	26
5.2.2.3	ESMemHandler	27
5.2.2.4	ESJpegMemHandler	28
5.2.2.5	ImageFrameHandler	28
5.2.2.6	CameraHandler	29
5.3	推理模块	30
5.3.1	前处理	31
5.3.1.1	自定义前处理方法	31
5.3.2	推理	32
5.3.3	后处理	32
5.3.3.1	自定义后处理方法	33
5.3.4	batch 策略	35
5.3.5	推理引擎	36
5.3.6	使用说明及参数详解	36
5.3.7	开发样例	37
5.3.7.1	自定义前处理开发样例	37
5.3.7.2	自定义后处理开发样例	38
5.4	追踪模块	38
5.4.1	特征值匹配方式	38
5.4.1.1	FeatureMatch	38
5.4.2	使用说明及参数详解	39
5.5	OSD 模块	40
5.5.1	使用说明及参数详解	41
5.6	VEncode 模块	42
5.6.1	使用说明及参数详解	42
5.7	Vout 模块	43
5.7.1	使用说明及参数详解	43

5.8	Kafka 模块	44
5.8.1	使用说明及参数详解	44
5.8.2	自定义 Handler	45
<b>6</b>	<b>自定义模块</b>	<b>47</b>
6.1	概述	47
6.2	自定义普通模块	47
6.3	自定义数据源模块	48
6.4	自定义扩展模块	48
<b>7</b>	<b>创建应用程序</b>	<b>50</b>
7.1	概述	50
7.2	应用程序的创建	50
7.2.1	配置文件方式	50
7.2.1.1	JSON 配置文件的编写	50
7.2.1.2	Pipeline 基本骨架的构建	53
7.3	用户侧 MessageHandle	54
7.3.1	1. EOS_MSG	54
7.3.2	2. FRAME_ERR_MSG	54
7.3.3	3. STREAM_ERR_MSG	54
7.3.4	4. ERROR_MSG	54
<b>8</b>	<b>工具</b>	<b>55</b>
8.1	Inspect 工具	55
8.1.1	工具命令的使用	55
8.1.1.1	打印工具帮助信息	55
8.1.1.2	查看框架支持的所有模块	56
8.1.1.3	查看某个模块的参数	56
8.1.1.4	打印 CNStream 的版本信息	56
8.1.2	配置 Inspect 工具	57
8.2	性能统计	57
8.2.1	机制原理	58
8.2.2	开启性能统计功能	58
8.2.3	内置处理过程	59
8.2.3.1	获取模块输入队列性能数据	59
8.2.3.2	获取模块 Process 函数性能数据	59
8.2.4	统计模块的性能	60
8.2.5	Pipeline 端到端的性能统计	61
8.2.6	获取性能统计结果	61
8.2.6.1	获取 Pipeline 整体性能数据	61

8.2.6.2	获取 pipeline 端到端的性能数据	63
8.2.6.3	获取指定模块的性能数据	63
8.2.6.4	获取指定处理过程的性能数据	63
8.2.6.5	获取每一路数据流的性能数据	64
8.2.7	性能统计数据说明	64
8.2.8	示例代码	65
8.2.8.1	完整性能数据示例	66
8.2.8.2	最近两秒的性能数据打印示例	66
8.3	数据流的追踪	67
8.3.1	开启数据流追踪功能	67
8.3.2	事件记录方式	68
8.3.3	追踪数据占用的内存空间	69
8.3.4	获取追踪数据	70
8.3.4.1	process_traces	70
8.3.4.2	module_traces	70
8.3.5	追踪数据的处理	70
8.3.6	追踪数据可视化	71
<b>9</b>	<b>FAQ</b>	<b>72</b>
9.1	file_list 文件是做什么用的?	72
9.2	怎么输入任意命名的图片?	73
9.3	有没有交叉编译 CNStream 的指导?	73
9.4	parallelism 参数该怎么配置?	73
9.5	使用 module_contrib 目录下的模块出现类未定义错误该怎么解决?	73
9.6	怎么调整 Log 打印等级?	74
<b>10</b>	<b>Release Notes</b>	<b>75</b>
10.1	CNStream 2022-10-17 (Version 7.0.0)	75
10.1.1	新增功能及功能变更	75
10.1.2	废用功能	75
10.1.3	版本兼容	76
10.1.4	版本限制	76
10.2	CNStream 2022-8-26 (Version 6.3.0)	76
10.2.1	新增功能及功能变更	76
10.2.2	版本兼容	76
10.2.3	版本限制	76
10.3	CNStream 2022-1-6 (Version 6.2.0)	76
10.3.1	新增功能及功能变更	76
10.3.2	废用功能	77
10.3.3	版本兼容	77

10.3.4	版本限制	77
10.4	CNStream 2021-10-11 (Version 6.1.0)	77
10.4.1	新增功能及功能变更	77
10.4.2	废用功能	77
10.4.3	版本兼容	77
10.4.4	版本限制	78
10.5	CNStream 2021-8-10 (Version 6.0.0)	78
10.5.1	新增功能及功能变更	78
10.5.2	废用功能	78
10.5.3	版本兼容	78
10.5.4	版本限制	78
10.6	CNStream 2021-1-28 (Version 5.3.0)	79
10.6.1	新增功能及功能变更	79
10.6.2	废用功能	79
10.6.3	版本兼容	79
10.6.4	版本限制	79
10.7	CNStream 2020-9-18 (Version 5.2.0)	79
10.7.1	新增功能及功能变更	79
10.7.2	废用功能	80
10.7.3	版本兼容	80
10.7.4	版本限制	80
10.8	CNStream 2020-07-10 (Version 5.0.0)	80
10.8.1	新增功能及功能变更	80
10.8.2	版本兼容	80
10.8.3	版本限制	80
10.9	CNStream 2020-05-28 (Version 4.5.0)	81
10.9.1	新增功能及功能变更	81
10.9.2	版本兼容	81
10.9.3	版本限制	81
10.10	CNStream 2020-04-16 (Version 4.4.0)	81
10.10.1	新增功能及功能变更	81
10.10.2	废用功能	81
10.10.3	版本兼容	82
10.10.4	版本限制	82
10.11	CNStream 2019-02-20	82
10.11.1	新增功能及功能变更	82
10.11.2	版本限制	82
10.12	CNStream 2019-12-31	82
10.12.1	新增功能及功能变更	82



## 插图目录

8.1 追踪数据可视化 .....	71
-------------------	----



## 表格目录

2.1 CNStream 编译选项 . . . . .	8
8.1 性能统计字段说明 . . . . .	64
8.2 数据流追踪字段说明 . . . . .	68





# 1 版权声明

## 免责声明

中科寒武纪科技股份有限公司（下称“寒武纪”）不代表、担保（明示、暗示或法定的）或保证本文件所含信息，并明示放弃对可销售性、所有权、不侵犯知识产权或特定目的适用性做出任何和所有暗示担保，且寒武纪不承担因应用或使用任何产品或服务而产生的任何责任。寒武纪不应因下列原因产生的任何违约、损害赔偿、成本或问题承担任何责任：（1）使用寒武纪产品的任何方式违背本指南；或（2）客户产品设计。

## 责任限制

在任何情况下，寒武纪都不对因使用或无法使用本指南而导致的任何损害（包括但不限于利润损失、业务中断和信息损失等损害）承担责任，即便寒武纪已被告知可能遭受该等损害。尽管客户可能因任何理由遭受任何损害，根据寒武纪的产品销售条款与条件，寒武纪为本指南所述产品对客户承担的总共和累计责任应受到限制。

## 信息准确性

本文件提供的信息属于寒武纪所有，且寒武纪保留不经通知随时对本文件信息或对任何产品和服务做出任何更改的权利。本指南所含信息和本指南所引用寒武纪文档的所有其他信息均“按原样”提供。寒武纪不担保信息、文本、图案、链接或本指南内所含其他项目的准确性或完整性。寒武纪可不经通知随时对本指南或本指南所述产品做出更改，但不承诺更新本指南。

本指南列出的性能测试和等级要使用特定芯片或计算机系统或组件来测量。经该等测试，本指南所示结果反映了寒武纪产品的大概性能。系统硬件或软件设计或配置的任何不同会影响实际性能。如上所述，寒武纪不代表、担保或保证本指南所述产品将适用于任何特定用途。寒武纪不代表或担保测试每种产品的所有参数。客户全权承担确保产品适合并适用于客户计划的应用以及对应用程序进行必要测试的责任，以避免应用程序或产品的默认情况。

客户产品设计的脆弱性会影响寒武纪产品的质量和可靠性并导致超出本指南范围的额外或不同的情况和/或要求。

## 知识产权通知

寒武纪和寒武纪的标志是中科寒武纪科技股份有限公司在中国和其他国家的商标和/或注册商标。其他公司 and 产品名称应为与其关联的各自公司的商标。

本指南为版权所有并受全世界版权法律和条约条款的保护。未经寒武纪的事先书面许可，不可以任何方

式复制、重制、修改、出版、上传、发布、传输或分发本指南。除了客户使用本指南信息和产品的权利，根据本指南，寒武纪不授予其他任何明示或暗示的权利或许可。未免疑义，寒武纪不根据任何专利、版权、商标、商业秘密或任何其他寒武纪的知识产权或所有权对客户授予任何（明示或暗示的）权利或许可。

- 版权声明
- © 2022 中科寒武纪科技股份有限公司保留一切权利。



## 2 快速入门

本章重点介绍了如何配置和编译 CNStream，以及如何运行寒武纪提供的 CNStream 示例。

更多 CNStream 详细介绍：

- CNStream 详细概念和功能介绍，参考[概述](#)。
- 创建应用的操作指南，参考[创建应用程序](#)。

### 2.1 安装和配置环境依赖和依赖库

用户需要安装依赖包后使用 CNStream。本节描述了如何在 Ubuntu 以及 Docker 环境下配置 CNStream。

#### 2.1.1 环境依赖

CNStream 有以下环境依赖。

- OpenCV
- FFmpeg
- GFlags
- GLog
- Librdkafka

对于边缘端平台，还有以下环境依赖。

- FREETYPE

#### 2.1.2 寒武纪安装包

##### 2.1.2.1 Cloud

CNStream 支持云端 MLU370 平台，依赖于寒武纪 CNToolkit 安装包中 CNDEV、CNDRV、CNRT 和 CNCodec 库，以及 MagicMind、CNCV、Cambricon CNNL、CNL Extra 和 CNLight 库。

用户需要在使用 CNStream 之前安装以上寒武纪发布的软件包。发送邮件到 [service@cambricon.com](mailto:service@cambricon.com)，联系寒武纪工程师获得相关的软件包和安装指南。

MLU370 具体版本依赖如下:

CNStream	CNToolk	CNCV	Driver	MagicMir	Cambricon CNL	CNNL_EXTI	CNLigh
v7.0.0	v3.0.2	v1.0.0	v4.20.9	v0.13.0	v1.11.1	v0.17.0	v0.15.0
v6.3.0	v2.6.x	v0.9.0	>=v4.16.4	v0.8.2	v1.8.3	v0.11.0	v0.10.0
v6.2.0	v2.6.x	v0.9.0	>=v4.16.4	v0.8.2	v1.8.3	v0.11.0	v0.10.0

### 2.1.2.2 Edge

CNStream 支持边缘端 CE3226 平台。

在 CE3226 平台，CNStream 依赖 `ce3226v100-sdk-x.y.z.tar.gz` 软件包中的 inference 和 mps 部分，该软件包的安装和使用请参阅《寒武纪 CE3226V100&101 SDK 使用开发指南》。

用户需要在使用 CNStream 之前安装以上寒武纪发布的软件包。发送邮件到 [service@cambricon.com](mailto:service@cambricon.com)，联系寒武纪工程师获得相关的软件包和安装指南。

CE3226 具体版本依赖如下:

CNStream	ce3226v100-sdk
v7.0.0	v1.0.0

#### 注解:

1. CNToolkit 库由一系列组件打包组成，在安装 CNToolkit 后切记依次安装该库包含的单个组件，CNStream 要求安装 CNDEV、CNDRV、CNRT 和 CNCodec。具体安装方法可参考《寒武纪 CNToolkit 安装升级使用手册》。
2. CNToolkit 与 Driver，MagicMind 与 Cambricon CNL、CNL EXTRA、CNLight 之间的更详细的版本依赖关系请查看 CNToolkit 和 MagicMind 用户手册。

### 2.1.3 Ubuntu 环境下安装和配置

执行下面命令，在 Ubuntu 环境下安装和配置环境依赖和依赖库：

1. 运行下面指令从 github 仓库检出 CNStream 源码。`\${CNSTREAM\_DIR}` 代表 CNStream 源码目录。

```
git clone https://github.com/Cambricon/CNStream.git
```

2. 安装寒武纪 CNToolkit 安装包。详情查看[寒武纪安装包](#)。
3. 运行下面指令安装环境依赖。CNStream 依赖的环境详情，查看[环境依赖](#)。

用户可通过 `\${CNSTREAM\_DIR}/tools` 下的 **pre\_required\_helper.sh** 脚本进行安装：

```
cd ${CNSTREAM_DIR}/tools  
./pre_required_helper.sh
```

或者通过以下命令进行安装：

```
sudo apt-get install libopencv-dev libgflags-dev libgoogle-glog-dev cmake librdkafka-dev  
sudo apt-get install libfreetype6 ttf-wqy-zenhei curl libcurl4-openssl-dev
```

### 2.1.4 Docker 环境下安装和配置

CNStream 提供以下 Dockerfile，其中 “`\${CNSTREAM\_DIR}`” 代表 CNStream 源码目录。

- `\${CNSTREAM\_DIR}/docker/Dockerfile.devel.16.04`
- `\${CNSTREAM\_DIR}/docker/Dockerfile.devel.18.04`

执行下面步骤使用 Docker 镜像配置独立于宿主机的开发环境：

1. 安装 Docker。宿主机需要预先安装 Docker。详情请查看 Docker 官网主页：<https://docs.docker.com/>
2. 运行下面指令从 github 仓库检出 CNStream 源码。

```
git clone https://github.com/Cambricon/CNStream.git
```

3. 编译 Docker 镜像。

```
cd ${CNSTREAM_DIR}/docker  
docker build -f Dockerfile.devel.18.04 -t cnstream:latest .
```

4. 运行下面命令，开启容器：

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=$DISPLAY -e GDK_SCALE -e GDK_DPI_  
↪SCALE --privileged -v /dev:/dev --net=host --ipc=host --pid=host -it --name container_name_  
↪ cnstream:latest
```

5. 当前版本 Dockerfile 制作镜像时并没有安装 CNToolkit 等内容，用户需要在启动容器后手动安装。详情查看[寒武纪安装包](#)。如有需要，用户可以通过改写 Dockerfile 将 CNToolkit 等内容默认安装在镜

像中。

### 2.2 编译 CNStream 指令

完成环境依赖的部署以及依赖库的安装后，执行下面步骤编译 CNStream 指令：

- 1. 运行下面指令从 github 检出子仓 EasyDK 源码，EasyDK 面向寒武纪硬件设备提供一套接口，支持用户快速开发和部署人工智能应用。

```
git submodule update --init
```

- 2. 运行下面命令创建 build 目录用来保存输出结果。

```
mkdir build
```

- 3. 运行下面命令生成编译指令的脚本。CNSTREAM\_DIR 为 CNStream 源码目录，cmake option 为 CNStream 编译选项。

```
cd build
cmake ${CNSTREAM_DIR} -D{cmake option}={ON/OFF}
```

表 2.1: CNStream 编译选项

cmake Option	Range	Default	Description
PLATFORM	MLU370 / CE3226	MLU370	Set platform
BUILD_MODULES	ON / OFF	ON	Build modules
BUILD_VENCODE	ON / OFF	ON	Build encode module
BUILD_INFERENCE	ON / OFF	ON	Build inference module
BUILD_OSD	ON / OFF	ON	Build osd module
BUILD_SOURCE	ON / OFF	ON	Build source module
BUILD_TRACK	ON / OFF	ON	Build track module
BUILD_SELECTOR	ON / OFF	ON	Build selector module
BUILD_VOUT	ON / OFF	ON	Build vout module

下页继续

表 2.1 – 续上页

BUILD_KAFKA	ON / OFF	ON	Build Kafka module
BUILD_MODULES_CONTRIB	ON / OFF	OFF	Build contributed modules
BUILD_TESTS	ON / OFF	ON	Build tests
BUILD_SAMPLES	ON / OFF	ON	Build samples
WITH_FFMPEG	ON / OFF	ON	Build with FFmpeg
WITH_FREETYPE	ON / OFF	OFF	Build with FREETYPE.
WITH_FFMPEG_AVDEVICE	ON / OFF	OFF	Build with FFmpeg avdevice

**注解:**

PLATFORM 编译选项的值不是 ON/OFF，用户需要根据目标平台选择 PLATFORM，例如 -DPLATFORM=CE3226

## 4. 运行下面命令编译 CNStream 指令：

```
make
```

## 5. 如果想要安装 CNStream 头文件和库到某个目录的话，则需要通过 CMAKE\_INSTALL\_PREFIX 指定路径，示例如下：

```
cmake {CNSTREAM_DIR} -DCMAKE_INSTALL_PREFIX=/path/to/install
make
make install
```

## 6. 如果想要交叉编译 CNStream，则需要事先交叉编译并安装第三方依赖库，并配置 CMAKE\_TOOLCHAIN\_FILE 文件，以 CE3226 为例：

```
export MPS_HOME=/your/path/to/mps_package
export PATH=$PATH:/your/path/to/cross-compiler/bin
cmake ${CNSTREAM_DIR} -DCMAKE_FIND_ROOT_PATH=/your/path/to/3rdparty-libraries-install-path -
↪DCMAKE_TOOLCHAIN_FILE=${CNSTREAM_DIR}/cmake/cross-compile.cmake -DCNIS_WITH_CURL=OFF
```

更多信息可以参考[交叉编译](#)。

**注解：**

目前已知 Ubuntu16.04 使能 MagicMind 后编译过程中会出现 libmagicmind\_runtime.so: undefined reference to xxxx@GLIBCXX\_3.4.22 报错,这是系统库版本较低引起的。MagicMind 是由高版本 GCC 编译的,引入了高版本的 ABI(CXXABI\_1.3.11 和 GLIBCXX\_3.4.22),Ubuntu16.04 的系统库 libstdc++.so 不包含这些高版本 ABI 符号。使用 gcc>=7 工具链编译 CNStream,或替换至高版本的 libstdc++.so 系统库 (Ubuntu18.04 的 libstdc++.so.6.0.25 满足要求) 都可以解决该问题。

## 2.3 CNStream 开发样例

寒武纪 CNStream 开发样例为用户提供了物体分类、检测、追踪等场景的编程样例。另外还提供了前处理、后处理、自定义模块以及如何使用非配置文件方式创建应用程序的样例源码。帮助用户快速体验如何使用 CNStream 开发应用。用户只需直接通过脚本运行样例程序,无需修改任何配置。

### 2.3.1 样例介绍

CNStream 开发样例主要包括.json 文件和.sh 文件,其中.json 文件为样例的配置文件,用于声明 pipeline 中各个模块的上下游关系以及各模块的配置参数。用户可以根据自己的需求修改配置文件参数,完成应用开发。.sh 文件为样例的运行脚本,通过运行该脚本来运行样例。

开发样例中的模型在运行样例时被自动加载,并且会保存在 \${CNSTREAM\_DIR}/data/models 目录下。

下面重点介绍 CNStream 提供的样例。样例支持在 MLU370 和 CE3226 平台上使用。

#### 2.3.1.1 图像分类样例

##### 样例文件

- 配置文 件: `${CNSTREAM_DIR}/samples/cns_launcher/image_classification/config_template.json`
- 运行脚本: `${CNSTREAM_DIR}/samples/cns_launcher/image_classification/run.sh`
- 后处理源码: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_classification.cpp`

##### 使用模块

- DataSource
- Inferencer
- Osd
- VEncode



### 2.3.1.2 目标检测样例

#### 样例文件

- 配置文件: `${CNSTREAM_DIR}/samples/cns_launcher/object_detection/config_template.json`
- 运行脚本: `${CNSTREAM_DIR}/samples/cns_launcher/object_detection/run.sh`
- 后处理源码:
  - yolov3: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov3.cpp`
  - yolov5: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov5.cpp`

#### 使用模块

- DataSource
- Inferencer
- Osd
- VEncode

### 2.3.1.3 物体追踪样例

#### 样例文件

- 配置文件: `${CNSTREAM_DIR}/samples/cns_launcher/object_tracking/config_template.json`
- 运行脚本: `${CNSTREAM_DIR}/samples/cns_launcher/object_tracking/run.sh`
- 后处理源码:
  - yolov3: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov3.cpp`
  - yolov5: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov5.cpp`

#### 使用模块

- DataSource
- Inferencer
- Tracker
- Osd
- VEncode

### 2.3.1.4 车辆结构化样例

#### 样例文件

- 配 置 文 件: `${CNSTREAM_DIR}/samples/cns_launcher/vehicle_recognition/config_template.json`
- 运行脚本: `${CNSTREAM_DIR}/samples/cns_launcher/vehicle_recognition/run.sh`
- 后处理源码:
  - yolov3: `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov3.cpp`

- classification:

`${CNSTREAM_DIR}/samples/common/postprocess/postprocess_classification.cpp`

#### 使用模块

- DataSource
- Inferencer
- Osd
- VEncode

#### 2.3.1.5 姿态检测样例

##### 样例文件

- 配置文件: `${CNSTREAM_DIR}/samples/cns_launcher/body_pose/config_template.json`
- 运行脚本: `${CNSTREAM_DIR}/samples/cns_launcher/body_pose/run.sh`
- 后处理源码: `${CNSTREAM_DIR}/samples/common/cns_openpose/postprocess_body_pose.cpp`

#### 使用模块

- DataSource
- Inferencer
- PoseOsd
- VEncode

### 3.1 简介

CNStream 是面向寒武纪开发平台的数据流处理 SDK，基于模块化和流水线的思想，提供了一套基于 C++11 的类和接口来支持流处理多路并发的 Pipeline 框架。用户可以根据 CNStream 提供的接口，开发自定义模块，并通过模块之间相互连接，实现自己的业务处理流程。CNStream 能够大大简化寒武纪人工智能平台提供的推理和其他处理，如视频解码、图像前处理的集成。也能够在兼顾灵活性的同时，充分发挥寒武纪硬件解码和人工智能算法的运算性能。

### 3.2 CNStream 优势

CNStream 构建了一整套寒武纪硬件平台上的实时数据流分析框架，并具有多个基于寒武纪思元处理器的硬件加速模块，可将人工智能算法和其他复杂处理任务带入流处理管道。开发者只需专注于构建核心人工智能算法，无需从头开始设计端到端的解决方案。

CNStream 具有以下几个特点：

- 简单易用的模块化设计。内置模块及正在扩充的模块库可以让用户快速构建自己的业务应用，无需关心实现细节。
- 高效的流水线设计。区别于 GStreamer 等框架庞大的结构及传统的视频处理流水线结构，CNStream 设计了一套伸缩灵活的流水线框架，根据业务需求，通过配置 JSON 文件就可以很方便地进行调整。
- 丰富的原生模块。为提高推理效率，根据寒武纪推理芯片设计特点，内置了从数据源解码、前后处理及推理、追踪等模块。其中编解码模块、追踪模块和推理模块充分利用了寒武纪芯片设计特点，使用后极大的提高了系统整体吞吐效率。
- 支持常见分类以及常规目标检测网络，比如：YOLO、SSD、ResNet 等。
- 灵活布署。使用标准 C++11 开发，可以将一份代码根据设备能力，在云边边缘端集成。

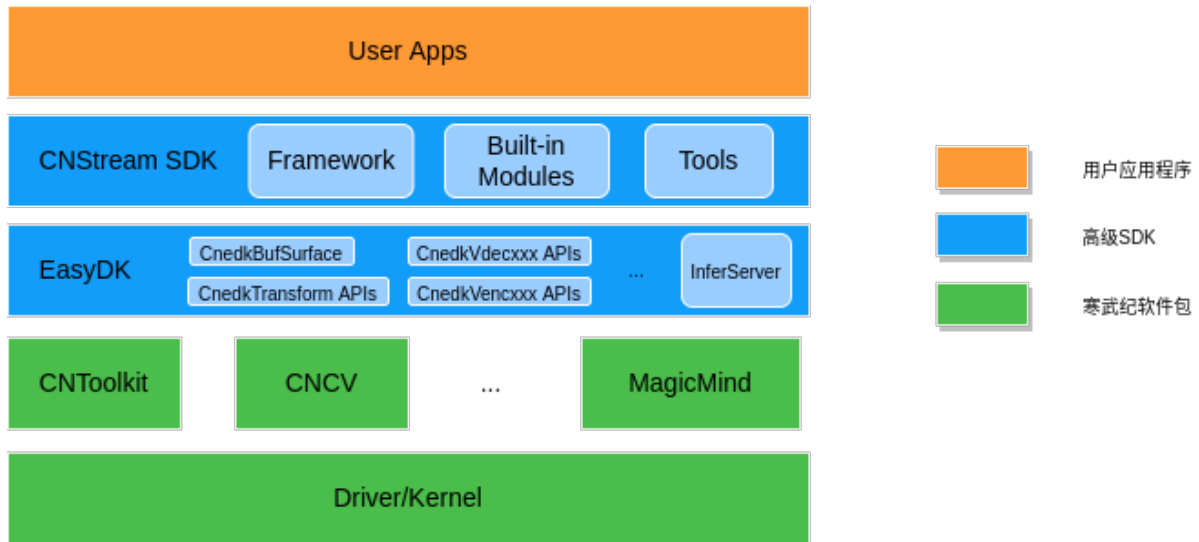
针对常规视频结构化分析（IVA）领域，使用 CNStream 开发应用可以带来如下便捷：

- 数据流处理能力。具有少冗余、高效率的特点。
- 硬件解码能力，用户只需提供原始数据即可。
- CNStream 内置 Inference 模块针对寒武纪 MagicMind 库设计，能够高效利用底层硬件能力，快速完成推理业务开发。

- 内置 Tracker 模块提供了经过寒武纪芯片加速优化后的 FeatureMatch 算法，使多路并行比运行在 CPU 上的效率更好。
- 良好的本地化支持团队、持续的迭代开发能力以及内建开发者社区提供了快速的客户响应服务。

### 3.3 CNStream 软件栈

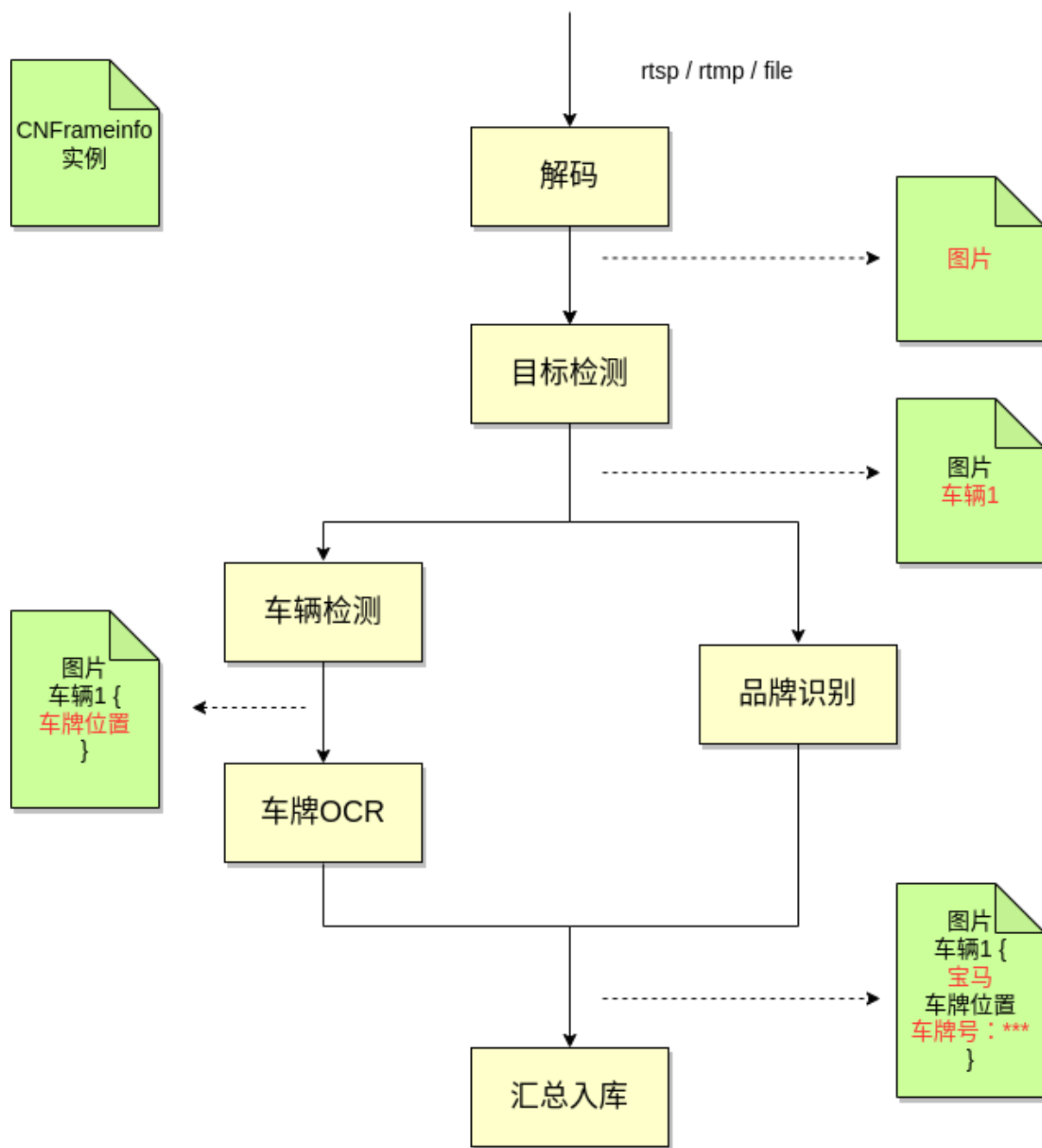
CNStream 依赖寒武纪软件包提供的 CNtoolkit、CNCV、Magicmind 等底层库和 EasyDK 库，后者实现了编解码的接口封装和推理功能的服务化接口封装（Inference Server）。



CNStream 内部主要由框架和内置模块两部分组成。框架基于有向无环图结构搭建业务流水线，并负责流水线上的节点调度和数据传输。每个节点作为一个单独的处理模块，既可处理设备端数据，也可以处理 CPU 数据。同时 CNStream 针对视频结构化的常用功能，如拉流解码、推理、追踪和推流等，以内置模块形式提供，用户可以直接使用，提升业务部署效率。

### 3.4 CNStream 数据流动

CNStream 模块间流动的数据实体为 `std::shared_ptr<CNFrameInfo>`，随着计算流的推进不断丰富结构体中的内容，CNFrameInfo 包含了一张图像所有的结构化信息，下图以车辆结构化为例，描述了 CNFrameInfo 中不断增加的信息。



流水线模块之间是一种生产者消费者的关系，通过模块的输入队列串联。上游模块生产数据后塞入到下游模块的输入队列，下游模块则从自身的输入队列中获取数据进行消费，如此迭代往后流动。当某个模块的输入数据队列被塞满，则会对上游模块造成阻塞，阻止内存占用无限上涨的同时阻止上游模块对计算资源的无意义占用。

## 4 框架介绍

### 4.1 核心框架

CNStream SDK 基于管道（Pipeline）和事件总线（EventBus）实现了模块式数据处理流程。

Pipeline 类似一个流水线，把复杂问题的解决方案分解成一个个处理阶段，然后依次处理。一个处理阶段的结果是下一个处理阶段的输入。Pipeline 模式的类模型由三部分组成：

- Pipeline：代表执行流。
- Module：代表执行流中的一个阶段。
- Context：是 Module 执行时的上下文信息。

EventBus 模式主要用来处理事件，包括三个部分：

- 事件源（Event Source）：将消息发布到事件总线上。
- 事件监听器（Observer/Listener）：监听器订阅事件。
- 事件总线（EventBus）：事件发布到总线上时被监听器接收。

Pipeline 和 EventBus 模式实现了 CNStream 框架。相关组成以及在 CNStream SDK 实现中对应关系如下：

- Pipeline：对应 **cnstream::Pipeline** 类。
- Module：Pipeline 的每个处理阶段是一个组件，对应 **cnstream::Module** 类。每一个具体的 module 都是 **cnstream::Module** 的派生类。
- FrameInfo：Pipeline 模式的 Context，对应 **cnstream::CNFrameInfo** 类。
- Event-bus 和 Event：分别对应 **cnstream::EventBus** 类和 **cnstream::Event** 类。

CNStream 既支持构造线性模式的 pipeline，也支持搭建非线性形状的 pipeline，例如 split、join 模式，如下所示：

```
ModuleA-----ModuleB-----ModuleC
```

```

      |-----ModuleB-----|
ModuleA---- |                  | ---- ModuleD
      |-----ModuleC-----|

```

## 4.2 cnstream::Pipeline 类

**cnstream::Pipeline** 类实现了 pipeline 的搭建、module 管理、以及 module 的调度执行。在 module 自身不传递数据时，负责 module 之间的数据传递。此外，该类集成事件总线，提供注册事件监听器的机制，使用户能够接收事件。例如 stream EOS（End of Stream）等。Pipeline 通过隐含的深度可控的队列来连接 module，使用 module 的输入队列连接上游的 module。CNStream 也提供了根据 JSON 配置文件来搭建 pipeline 的接口。在不重新编译源码的情况下，通过修改配置文件搭建不同的 pipeline。当用户程序想要获取 Pipeline 处理结果作进一步的处理时，可以从 **cnstream::IModuleObserver** 接口类继承实现一个观察者，并通过 RegisterFrameDoneCallBack 注册到 Pipeline 中，处理完的帧数据会通过 Notify 回调函数传出。

**cnstream::Pipeline** 类在 cnstream\_pipeline.hpp 文件内定义，cnstream\_pipeline.hpp 文件存放于 framework/core/include 目录下。源代码中有详细的注释，这里仅给出主要接口的必要说明。接口详情查看《寒武纪 CNStream 开发者手册》。

```
class Pipeline {
    ...
public:
    // 根据 ModuleConfigs 或者 JSON 配置文件来搭建 pipeline。
    // 实现这两者前提是能够根据类名字创建类实例即反射（reflection）机制。
    // 在 cnstream::Module 类介绍中会进行描述。
    int BuildPipeline(const std::vector<CNModuleConfig>& configs);
    int BuildPipelineByJSONFile(const std::string& config_file);

    ...

    // 向某个 module 发送 CNFrameInfo，比如向一个 pipeline 的 source module 发送图像数据。
    bool ProvideData(const Module* module, std::shared_ptr<CNFrameInfo> data);

    ...
    // 开始和结束 pipeline service。
    bool Start();
    bool Stop();

    ...
    // 根据 module name 获得 module instance。
    Module* GetModule(const std::string& name);
    ...
};
```

ModuleConfigs（JSON）的示例如下。JSON 配置文件支持 C 和 C++ 风格的注释。

```
{
{
    "source" : {
        "class_name" : "cnstream::DataSource", //指定 module 使用哪个类来创建。
        "parallelism" : 0, //框架创建的 module 线程数目。source module 不使用这个字段。
        "next_modules" : ["inference"], //下一个连接模块的名字，可以有多个。
        "custom_params" : { //当前 module 的参数。
            "device_id" : 0 //硬件设备 id。
        }
    },

    "inference" : {
        "class_name" : "cnstream::Inferencer",
        "parallelism" : 1, //框架创建的模块线程数，也是输入队列的数目。
        "max_input_queue_size" : 32, //输入队列的最大长度。
        "custom_params" : {
            // 使用寒武纪工具生成的离线模型，支持绝对路径和 JSON 文件的相对路径。
            "model_path" : "/data/models/yolov5_4b_rgb_uint8.magicmind",
            "device_id" : 0
        }
    }
}
}
```

### 4.3 cnstream::Module 类

CNStream SDK 要求所有的 Module 类使用统一接口和数据结构 **cnstream::CNFrameInfo**。从框架上要求了 module 的通用性，并简化了 module 的编写。实现具体 module 的方式如下：

- 从 **cnstream::Module** 派生：适合功能单一，内部不需要并发处理的场景。Module 实现只需要关注对 CNFrameInfo 的处理，由框架传递（transmit）CNFrameInfo。
- 从 **cnstream::ModuleEx** 派生：Module 除了处理 CNFrameInfo 之外，还负责 CNFrameInfo 的传递，以及保证数据顺序带来的灵活性，从而可以实现内部并发。

配置搭建 pipeline 的基础是实现根据 module 类名字创建 module 实例，因此具体 module 类还需要继承 **cnstream::ModuleCreator**。

一个 module 的实例，会使用一个或者多个线程对多路数据流进行处理，每一路数据流使用 pipeline 范围内唯一的 stream\_id 进行标识。此外从 **cnstream::IModuleObserver** 接口类继承实现一个观察者，并通过 SetObserver 注册到 module 中，应用程序就可以观察每个 module 处理结果。

**cnstream::Module** 类在 cnstream\_module.hpp 文件定义，主要接口如下。cnstream\_module.hpp 文件存放在 framework/core/include 文件夹下。源代码中有详细的注释，这里仅给出必要的说明。接口



详情，查看《寒武纪 CNStream 开发者手册》。

```
class Module {
public:

    // 一个 pipeline 中，每个 module 名字必须唯一。
    explicit Module(const std::string &name) : name_(name) { }
    ...

    // 必须实现 Open、Close 和 Process 接口。这三个接口会被 pipeline 调用。
    // 通过 Open 接口接收参数，分配资源。
    // 通过 Close 接口释放资源。
    // 通过 Process 接口接收需要处理的数据，并更新 CNFrameInfo。
    virtual bool Open(ModuleParamSet param_set) = 0;
    virtual void Close() = 0;

    // 特别注意：Process 处理多个 stream 的数据，由多线程调用。
    // 单路 stream 的 CNFrameInfo 会在一个线程中处理。
    // Process 的返回值：
    // 0    -- 表示成功。
    // < 0 -- 表示有错误产生。
    virtual int Process(std::shared_ptr<CNFrameInfo> data) = 0;

    ...
    // 向 pipeline 发送消息，如 Stream EOS。
    bool PostEvent(EventType type, const std::string &msg) ;

    // 注册一个观察者。
    void SetObserver(IModuleObserver *observer);
};
```

## 4.4 cnstream::CNFrameInfo 类

**cnstream::CNFrameInfo** 类是 module 之间传递的数据结构，即 pipeline 的 Context。该类在 `cnstream_frame.hpp` 文件中定义。`cnstream_frame.hpp` 文件存放在 `framework/core/include` 文件夹下。

比如 CNStream 内置插件库中针对智能视频分析场景专门定义了 `CNDataFrame` 和 `CNInferObject`，分别用于存放视频帧数据和推理结果：

```
auto frame = collection.Get<CNDataFramePtr>(kCNDataFrameTag);
collection.Add(kCNInferObjsTag, std::make_shared<CNInferObjs>());
```

CNDataFrame 中使用 cnedk::BufSurfaceWrapper 实例保存帧数据。接口如下：

```
cnedk::BufSurfWrapperPtr buf_surf;
```

```
// 获取 CnedkBufSurface。
buf_surf->GetBufSurface();
// 获取保存在 CnedkBufSurface 中的数据。
buf_surf->GetData();
// 获取 Host 侧数据。
buf_surf->GetHostData();
// 同步 Host 侧数据到设备侧。
buf_surf->SyncHostToDevice();
```

另外，CNInferObject 不仅提供对常规推理结果的数据存储机制，还提供用户自定义数据格式的接口 AddExtraAttribute，方便用户使用其他格式传递数据，如 JSON 格式。

```
bool AddExtraAttribute(const std::vector<std::pair<std::string, std::string>>& attributes);
std::string GetExtraAttribute(const std::string& key);
```

## 4.5 cnstream::EventBus 类

**cnstream::EventBus** 类是各个模块与 pipeline 通信的事件总线。各模块发布事件到总线上，由总线监听器接收。一条事件总线可以拥有多个监听器。

每条 pipeline 有一条事件总线及对应的一个默认事件监听器。pipeline 会对事件总线进行轮询，收到事件后分发给监听器。

**cnstream::EventBus** 类在 cnstream\_eventbus.hpp 文件中定义，主要接口如下。cnstream\_eventbus.hpp 文件存放在 framework/core/include 文件夹下。源代码中有详细的注释，这里仅给出必要的说明。接口详情，查看《寒武纪 CNStream 开发者手册》。

```
class EventBus {
public:

    // 向事件总线上发布一个事件。
    bool PostEvent(Event event);

    // 添加事件总线的监听器。
```

```
uint32_t AddBusWatch(BusWatcher func, Pipeline *watcher);
.....
};
```

## 4.6 cnstream::Event 类

**cnstream::Event** 类是模块和 pipeline 之间通信的基本单元，即事件。事件由四个部分组成：事件类型、消息、发布事件的模块、发布事件的线程号。消息类型包括：无效、错误、警告、EOS(End of Stream)、停止，数据流错误，以及一个预留类型。

**cnstream::Event** 类在 `cnstream_eventbus.hpp` 文件定义，`cnstream_eventbus.hpp` 文件存放在 `framework/core/include` 文件夹下。接口详情，查看《寒武纪 CNStream 开发者手册》。

## 4.7 cnstream::Collection 类

**cnstream::Collection** 是 CNStream 用于存放任意类型数据的类，通过该类添加与读取数据的操作都是线程安全的。方便用户使用自定义的类型存储推理结果。该类在 `cnstream_collection.hpp` 文件中定义。`cnstream_collection.hpp` 文件存放在 `framework/core/include` 文件夹下。数据最终以键值对形式存储于如下所示的 `data_` 中。

```
std::map<std::string, std::unique_ptr<cnstream::any>> data_;
```

用户通过 Add 接口添加数据，通过 Get 接口读取数据，如下代码所示，

```
//用户自定义类型数据
struct UserDefinedData {
    int id = 0;
}
//由用户自定义索引 UserDefinedData 类型的键值
constexpr std::string kTag = "Tag";

//创建 CNFrameInfo 和 UserDefinedData 数据
CNFrameInfoPtr data = CNFrameInfo::Create("stream_0"/*stream id*/, false/*eos*/);
std::shared_ptr<UserDefinedData> my_data = std::make_shared<UserDefinedData>();

//将 example_data 添加到 collection 中
data->collection.Add(kTag, my_data);

//在其他地方获取数据并对数据进行处理
```

```
auto ptr = data->collection.Get<std::shared_ptr<UserDefinedData>>(kTag);  
ptr->id++;
```

### 5.1 简介

针对常规的视频结构化领域，CNStream 提供了以下核心功能模块：

- 数据源处理模块：依赖于 CNCodec SDK（硬件视频解码 SDK），用于视频解码和 JPEG 解码。支持多种协议的解封装及对多种格式的视频压缩格式进行解码。视频压缩格式和图片硬解码支持详情，请参考《寒武纪 CNCodec 开发者手册》。
- 推理模块：基于 CNStream 子仓库 EasyDK（Cambricon Easy Development Kit）提供的寒武纪推理服务（CNIS），支持多种人工智能离线模型对图像数据进行智能推理。
- 追踪模块：使用针对寒武纪平台优化的 FeatureMatch 算法，在保证精度的前提下减少 CPU 使用率，提高了模块性能。

除以上核心模块外，CNStream 还提供了自定义示例模块：OSD 模块、编码模块、Kafka 模块和多媒体显示模块等。

- OSD（On-Screen Display）模块：支持内容叠加和高亮物件处理。
- 编码模块：支持硬件编码，并保存流到文件和 RTSP 推流。
- Kafka 模块：支持发送 Kafka 消息数据到集群中。
- 多媒体显示模块：支持屏幕上显示视频。

### 5.2 数据源模块

数据源（DataSource）模块是 pipeline 的起始模块，实现了视频图像获取功能。支持通过 FFMpeg 解封装、解复用本地文件或网络流来得到码流。之后喂给解码器解码得到图像，并把图像存到 CNDataFrame 的 `cnedk::BufSurfaceWrapper` 中。目前支持 H264、H265、MP4、JPEG、RTSP 等协议。另外，边缘端还支持从摄像头传感器获取视频图像。

数据源模块提供不同的 Handler，分别对应不同的输入形式，可以参考 `samples/cns_launcher/cns_launcher.cpp` 中的代码创建不同的输入句柄。

- FileHandler，本地文件作为输入源
- RtspHandler，RTSP 流作为输入源
- ESMemHandler，内存中的视频编码数据作为输入源

- ESJpegMemHandler, 内存中的原始 Jpeg 压缩数据作为输入源
- ImageFrameHandler, 解码后的图片作为输入源
- CameraHandler, 摄像头 Sensor 作为输入源 (仅边缘端支持)

数据源模块主要有以下特点:

- 作为 pipeline 的起始模块, 没有输入队列。因此 pipeline 不会为 DataSource 启动和调度线程。数据源模块需要内部启动线程, 通过 Module 的 TransmitData() 接口向下游发送数据。
- 每一路数据流由使用者指定唯一标识 stream\_id。
- 支持动态增加和减少数据流。
- 支持通过配置文件修改和选择数据源模块的具体功能, 而不是在编译时选择。

**cnstream::DataSource** 类在 data\_source.hpp 文件中定义。data\_source.hpp 文件存放在 modules/source/include 文件夹下。DataSource 主要功能继承自 SourceModule 类, 存放在 framework/core/include 目录下。主要接口如下, 源代码中有详细的注释, 这里仅给出必要的说明。

```
class SourceModule : public Module {
public:
    // 动态增加一路 stream 接口。
    int AddSource(std::shared_ptr<SourceHandler> handler);
    // 动态减少一路 stream 接口。
    int RemoveSource(std::shared_ptr<SourceHandler> handler, bool force = false);
    int RemoveSource(const std::string &stream_id, bool force = false);

    // 对 source module 来说, Process() 不会被调用。
    // 由于 Module::Process() 是纯虚函数, 这里提供一个缺省实现。
    int Process(std::shared_ptr<CNFrameInfo> data) override;
    ...

private:
    ...
    // 每一路 stream, 使用一个 SourceHandler 实例实现。
    // source module 维护 stream_id 和 source handler 的映射关系。
    // 用来实现动态的增加和删除某一路 stream。
    std::map<std::string /*stream_id*/, std::shared_ptr<SourceHandler>> source_map_;
};
```

### 5.2.1 使用说明及参数详解

例如在 decode\_config.json 配置文件，如下所示。该文件位于 samples/cns\_launcher/configs 目录下。

```
"source" : {  
  "class_name" : "cnstream::DataSource", // (必设参数) 数据源类名。  
  "custom_params" : { // 特有参数。  
    "bufpool_size" : 16, // (可选参数) 内存池大小。  
    "interval" : 1, // (可选参数) 抽帧。  
    "device_id" : 0 // (必设参数) 设备 id, 用于标识多卡机器的设备唯一编号。  
  }  
}
```

模块特有参数说明如下：

- bufpool\_size: (可选参数) 设置内存池大小。默认为 16。该值越大占用硬件设备侧内存越大。该值较小将可能阻塞整体 pipeline 性能。云端支持保持输入图片原始分辨率，此时不再采用内存池的方式存储数据，该参数无效。边缘端只能使用内存池的方式保存解码后的数据。
- interval: (可选参数) 插件丢帧策略。指定每 interval 帧数据帧输出一帧，剩余的帧将被丢弃。默认为 1 (即不丢帧)。最小值为 1，最大值为 size\_t 类型最大值。  
例如，interval 为 3。解码后输出 7 帧。则第 1 帧和第 4 帧和第 7 帧将被传递到后续模块，其余帧将被丢弃。
- device\_id: (必选参数) 设置使用的设备 id，决定硬件解码使用的设备及解码后数据存放在哪个硬件设备上。

### 5.2.2 SourceHandler 使用说明

#### 5.2.2.1 FileHandler

本地文件作为输入源。可以是视频 (xx.mp4) 或图片序列 (xx%d.jpg)。(xx 代表任意字符)。

创建 FileHandler，添加到 DataSource，以及移除：

```
DataSource *source;  
// Start Pipeline, Get DataSource module pointer here  
...  
  
std::shared_ptr<SourceHandler> handler;  
  
FileSourceParam param;  
param.filename = "your_input_path";  
param.framerate = 25;
```

```

param.loop = false;
param.max_res.width = 3840;
param.max_res.height = 2160;
param.out_res.width = 1920;
param.out_res.height = 1080;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);

...

source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除

```

### 5.2.2.2 RtspHandler

RTSP 流作为输入源作为输入源 (`rtsp://xx`)。(xx 代表任意字符)。

创建 RtspHandler，添加到 DataSource，以及移除：

```

DataSource *source;
// Start Pipeline, Get DataSource module pointer here
...
auto cb = [](ESPacket pkt, std::string stream_id) { ... };

std::shared_ptr<SourceHandler> handler;

RtspSourceParam param;
param.url_name = "rtsp://xx";
param.use_ffmpeg = false; // use live555 or ffmpeg.
param.reconnect = 5; // reconnect times, -1 means endless.
param.interval = 3; // Process a frame every 3 frames. 当不同路的抽帧策略不同时，使用
此参数。
param.only_key_frame = false; // Choose to process only key frame of the video.
param.callback = cb; // Pass a callback function to get h264/h265 packet.
param.max_res.width = 3840;
param.max_res.height = 2160;
param.out_res.width = 1920;
param.out_res.height = 1080;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);

```



```
...

source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除
```

### 5.2.2.3 ESMemHandler

内存中的视频编码数据作为输入源。

创建 ESMemHandler，添加到 DataSource，以及移除：

```
DataSource *source;
// Start Pipeline, Get DataSource module pointer here
...

std::shared_ptr<SourceHandler> handler;

ESMemSourceParam param;
param.max_res.width = 3840;
param.max_res.height = 2160;
param.out_res.width = 1920;
param.out_res.height = 1080;
param.data_type = ESMemSourceParam::DataType::H264;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);

// Start another thread and Write packet. See ``samples/cns_launcher/cns_launcher.cpp`` for
↪ details.
// ESPacket pkt;
// pkt.has_pts = false;
// pkt.data = ...;
// pkt.size = ...;
// Write(handler, &pkt)
...

source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除
```

#### 5.2.2.4 ESJpegMemHandler

内存中的原始 Jpeg 压缩数据作为输入源。

创建 ESJpegMemHandler，添加到 DataSource，以及移除：

```
DataSource *source;
// Start Pipeline, Get DataSource module pointer here
...

std::shared_ptr<SourceHandler> handler;

ESJpegMemSourceParam param;
param.max_res.width = 3840;
param.max_res.height = 2160;
param.out_res.width = 1920;
param.out_res.height = 1080;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);

// Start another thread and Write packet. See ``samples/cns_launcher/cns_launcher.cpp`` for
↪details.
// ESJpegPacket pkt;
// pkt.data = ...;
// pkt.size = ...;
// pkt.pts = ...;
// Write(handler, &pkt)
...

source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除
```

#### 5.2.2.5 ImageFrameHandler

解码后的图片作为输入源。

创建 ImageFrameHandler，添加到 DataSource，以及移除：

```
DataSource *source;
// Start Pipeline, Get DataSource module pointer here
...
```

```

std::shared_ptr<SourceHandler> handler;

ImageFrameSourceParam param;
param.out_res.width = 1920;
param.out_res.height = 1080;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);

// Start another thread and Write packet. See ``samples/cns_launcher/cns_launcher.cpp`` for
↪details.
// ImageFrame pkt;
// pkt.data = ...;
// Write(handler, &pkt)
...

source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除

```

### 5.2.2.6 CameraHandler

摄像头 Sensor 作为输入源 (仅边缘端支持)。

创建 CameraHandler，添加到 DataSource，以及移除：

```

// 配置 sensor here. See ``samples/cns_launcher/cns_launcher.cpp`` for details.
...

DataSource *source;
// Start Pipeline, Get DataSource module pointer here
...

std::shared_ptr<SourceHandler> handler;

SensorSourceParam param;
param.sensor_id = 0;
param.out_res.width = 1920;
param.out_res.height = 1080;

handler = CreateSource(source, stream_id, param);
source->AddSource(handler);
...

```

```
source->RemoveSource(stream_id);
// source->RemoveSource(handler); // 支持传入 stream_id 或 handler 移除
```

**cnstream::FileSourceParam**、**cnstream::RtspSourceParam**、**ESMemSourceParam**、**ESJpegMemSourceParam**、**ImageFrameSourceParam**、**cnstream::CameraHandler** 定义在 `modules/source/include/data_source.hpp` 文件中。

#### 注解：

视频解码时，在云端平台上，当设置 `max_res` 的宽高大于 0 时，代表使能对带有多个不同分辨率码流进行解码。若不设置，则以第一帧的分辨率作为最大分辨率。在边缘端平台上，当不设置 `max_res` 或宽高等于 0 时，默认使用 1920x1080 分辨率。

图片解码时，当不设置 `max_res` 或宽高等于 0 时，默认使用 8192x4320 分辨率。

当处理的码流中分辨率大于设置的最大分辨率时，将导致解码失败，并向 Pipeline 发送类型为 `cnstream::StreamMsgType::STREAM_ERR_MSG` 的事件。业务代码可通过 `cnstream::StreamMsgObserver` 来接收该事件并按业务需要做相应的处理。

## 5.3 推理模块

推理模块（Inferencer）是基于 EasyDK 推理服务实现的一个具有推理功能的模块，主要包括前处理、推理和后处理三个部分。用户根据业务需求，只需载入相应的模型，即可调用底层的推理，简化了开发推理相关插件的代码。

推理模块支持一级推理和二级推理，一级推理的输入是帧数据，而二级推理的输入是帧上的目标数据（一般情况下为一级推理的结果）。模块默认以帧数据作为输入，即进行一级推理。当我们希望模块对所有一级目标进行二级推理时，将参数 `filter` 设置为 `all (ALL)` 或 `-1`。

除此之外，推理模块还支持过滤目标功能。过滤目标功能则仅适用于二级推理，用户可以仅推理感兴趣的目标。被过滤的目标不会被丢弃。

- 通过设置 `categoies`，选择目标的类别。

例如：设置参数 `"filter" : "categoies=2"`，代表仅对类别为 2 的目标进行二级推理。

- 通过设置 `name`，自定义过滤逻辑。用户需要自定义一个类，继承 `cnstream::ObjectFilterVideo`，重载 `Filter` 函数。并将类名以字符串的形式通过参数传入。

例如：设置参数 `"filter" : "name=ObjectFilterLpr"`。

`ObjectFilterLpr` 定义在 `samples/common/filter/object_filter_vs.cpp` 文件中。

- 以上两种过滤方式也可联合使用。

例如：设置参数 `"filter" : "name=ObjectFilterLpr; categoies=1"`。

EasyDK 推理服务具有以下特性：

- 推理服务提供了一套类似服务器的推理接口，推理模块在推理服务的基础上实现了推理、前处理及后处理等功能，简化了模块内部的代码逻辑。
- 推理服务支持用户根据需求选择组 batch 的策略，包括 dynamic 和 static 两种策略。详情查看[batch 策略](#)。
- 推理服务可以创建多份推理实例，并且在推理服务内部实现了负载均衡。详情查看 [\\_ 推理引擎](#)。
- 当多个模块的推理流程完全一致时，即前处理、推理模型和后处理部分都相同时，支持多个模块共用处理资源。

### 5.3.1 前处理

前处理是将原始数据处理成适合网络输入的数据。前处理必须由用户指定，自定义前处理需继承 `cnstream::Preproc` 类，并重写 `Execute` 函数。

用户可通过在 json 配置文件中，将自定义的前处理类以字符串的形式设置给参数 `preproc` 选择相应的前处理。

例如，设置参数 `"preproc" : "name=PreprocYolov5;use_cpu=false"`。代表使用 `CNStream` 中提供的 `PreprocYolov5` 前处理，并且使用硬件进行前处理。`use_cpu` 的值可以通过前处理类的 `hw_accel_` 成员变量获取。

json 配置文件详情查看[inferencer 说明](#)。

`CNStream` 提供自定义前处理的方法如下：

#### 5.3.1.1 自定义前处理方法

1. 继承 `cnstream::Preproc` 类。
2. 重写 `Execute` 函数，自定义前处理，将输入转换到符合模型输入的格式，并填入到模型输入的内存中。

函数参数描述如下：

- `src`（输入参数）：前处理的输入。参数类型为 `CnedkBufSurface*`。获取方式如下：

```
uint32_t batch_size = src->num_filled;
cnedk::BufSurfWrapperPtr src_surf = std::make_shared<cnedk::BufSurfaceWrapper>(src,
↪false);
for (uint32_t batch_idx = 0; batch_idx < batch_size; ++batch_idx) {
    uint8_t *y_plane = static_cast<uint8_t *>(src_surf->GetHostData(0, batch_idx));
    uint8_t *uv_plane = static_cast<uint8_t *>(src_surf->GetHostData(1, batch_idx));
    // 前处理
    ...
}
```

- `dst`（输出参数）：前处理的输出。参数类型为 `CnedkBufSurface*`。填入方式如下：

```

cnek::BufSurfWrapperPtr dst_surf = std::make_shared<cnek::BufSurfaceWrapper>(dst,
↪false);
void *dst_img = dst_surf->GetHostData(0, batch_idx);
// 将前处理结果填入 dst_img 中
...
dst_surf->SyncHostToDevice();

```

- `src_rects`（输入参数）：前处理输入的有效区域。参数类型为 `const std::vector<CnekTransformRect> &`。该参数仅在二级推理时有效。

CNStream 提供自定义前处理公共代码，实现通用的前处理方式，包括缩放，颜色空间转换，保持宽高比，减均值除方差，截取有效区域等，代码位于 `samples/common/preprocess/preprocess_common.cpp`。

3. 重写 `OnTensorParams` 函数，获取模型输入相关参数。方式如下：

```

int OnTensorParams(const infer_server::CnPreprocTensorParams *params) override {
    // 定义 info_ 为类成员变量，类型为 cnstream::CnPreprocNetworkInfo
    if (GetNetworkInfo(params, &info_) < 0) {
        return -1;
    }
    // model input shape, data type, pixel format
    std::cout << "Model input : w = " << info_.w << ", h = " << info_.h << ", c = " <<
↪info_.c
        << ", dtype = " << static_cast<int>(info_.dtype)
        << ", pixel_format = " << info_.format;
    return 0;
}

```

### 5.3.2 推理

推理是该模块的核心功能，用户只需在 json 配置文件中设置模型路径即可。通过 `model_path` 参数设置离线模型的存放路径。详情查看[inferencer 说明](#)。

### 5.3.3 后处理

后处理是将模型的输出结果做进一步的加工处理，如筛选阈值等。后处理必须由用户指定，该模块不提供默认后处理。自定义后处理需继承 `cnstream::Postproc` 类，并重写 `Execute` 函数。

json 配置文件详情查看[inferencer 说明](#)。

CNStream 提供自定义后处理的方法如下：

### 5.3.3.1 自定义后处理方法

1. 继承 `cnstream::Postproc` 类。
2. 重写 `Execute` 函数，自定义处理推理输出结果，并填入到 `CNStream` 的数据结构中。

对于一级推理，函数参数描述如下：

- `net_outputs`（输入参数）：离线模型推理一个 batch 的结果，即后处理的输入。参数类型为 `const cnstream::NetOutputs&`。离线模型可以包含多个输出，每个输出都保存在 `net_outputs` 对应的元素中。获取方式如下：

```
cnedk::BufSurfWrapperPtr output0 = net_outputs[0].first; // output 0: data
cnedk::BufSurfWrapperPtr output1 = net_outputs[1].first; // output 1: bbox
output0->GetHostData(0);
output1->GetHostData(0);
CnedkBufSurfaceSyncForCpu(output0->GetBufSurface(), -1, -1);
CnedkBufSurfaceSyncForCpu(output1->GetBufSurface(), -1, -1);
float* data = static_cast<float*>(output0->GetHostData(0, batch_idx));
int box_num = static_cast<int*>(output1->GetHostData(0, batch_idx))[0];
...
```

- `model_info`（输入参数）：提供离线模型的详细信息，包括输入输出数量、形状、数据布局以及数据类型等。参数类型为 `const infer_server::ModelInfo&`。
- `packages`（输入参数/输出参数）：将一级推理的后处理结果填入。参数类型为 `const std::vector<cnstream::CNFrameInfoPtr>&`。示例如下：

```
for (size_t batch_idx = 0; batch_idx < packages.size(); batch_idx++) {
    cnstream::CNFrameInfoPtr package = packages[batch_idx];
    const auto frame = package->collection.Get<cnstream::CNDataFramePtr>
    (cnstream::kCNDataFrameTag);
    cnstream::CNInferObjsPtr objs_holder = nullptr;
    if (package->collection.HasValue(cnstream::kCNInferObjsTag)) {
        objs_holder = package->collection.Get<cnstream::CNInferObjsPtr>
        (cnstream::kCNInferObjsTag);
    }
    std::lock_guard<std::mutex> lk(objs_holder->mutex_);
    auto obj = std::make_shared<cnstream::CNInferObject>();
    // set object attribute
    ...

    objs_holder->objs_.push_back(obj);
}
```

- `labels`（输入参数）：目标 label。参数类型为 `const cnstream::LabelStrings&`。

对于二级推理，除了上述一级为例的参数外，还有以下参数：

- objects（输入参数/输出参数）：二级推理对应的一级输入目标。参数类型为 `const std::vector<cnstream::CNInferObjectPtr>&`。

如下是一个简单的一级推理分类网络后处理示例：

```
class PostprocClassification : public cnstream::Postproc {
public:
    int Execute(const cnstream::NetOutputs& net_outputs, const infer_server::ModelInfo& model_
↵info,
                const std::vector<cnstream::CNFrameInfoPtr>& packages,
                const cnstream::LabelStrings& labels) override;
private:
    DECLARE_REFLEX_OBJECT_EX(PostprocClassification, cnstream::Postproc);
}; // class PostprocClassification

IMPLEMENT_REFLEX_OBJECT_EX(PostprocClassification, cnstream::Postproc);

int PostprocClassification::Execute(const cnstream::NetOutputs& net_outputs, const infer_
↵server::ModelInfo& model_info,
                                    const std::vector<cnstream::CNFrameInfoPtr>& packages,
                                    const cnstream::LabelStrings& labels) {
    cnedk::BufSurfWrapperPtr output = net_outputs[0].first;
    output->GetHostData(0);
    CnedkBufSurfaceSyncForCpu(output->GetBufSurface(), -1, -1);

    auto len = model_info.OutputShape(0).DataCount();

    for (size_t batch_idx = 0; batch_idx < packages.size(); batch_idx++) {
        float* data = static_cast<float*>(output->GetHostData(0, batch_idx));
        auto score_ptr = data;

        float max_score = 0;
        uint32_t label = 0;
        for (decltype(len) i = 0; i < len; ++i) {
            auto score = *(score_ptr + i);
            if (score > max_score) {
                max_score = score;
                label = i;
            }
        }
        if (threshold_ > 0 && max_score < threshold_) continue;
    }
}
```



```

cnstream::CNFrameInfoPtr package = packages[batch_idx];
cnstream::CNInferObjsPtr objs_holder = nullptr;
if (package->collection.HasValue(cnstream::kCNInferObjsTag)) {
    objs_holder = package->collection.Get<cnstream::CNInferObjsPtr>
↪(cnstream::kCNInferObjsTag);
}

if (!objs_holder) {
    LOGE(PostprocClassification) << " object holder is nullptr.";
    return -1;
}

auto obj = std::make_shared<cnstream::CNInferObject>();
obj->id = std::to_string(label);
obj->score = max_score;

if (!labels.empty() && label < labels[0].size()) {
    obj->AddExtraAttribute("Category", labels[0][label]);
}

std::lock_guard<std::mutex> lk(objs_holder->mutex_);
objs_holder->objs_.push_back(obj);
} // for(batch_idx)

return 0;
}

```

### 5.3.4 batch 策略

通常我们会选择多 batch 的离线模型进行推理，一次执行一组 batch 数据，减少任务下发次数提升资源利用率，达到提高推理性能的目的。当使用的离线模型为多 batch 时，该模块支持用户根据需求选择组 batch 的策略，包括 dynamic 和 static 策略。

- dynamic 策略：总吞吐量较高。在推理服务内部进行组 batch，每次请求后不会立即执行，而是等到组满整个 batch 或超时后才开始执行任务，所以单个推理响应时延较长。
- static 策略：总吞吐量较低。每次请求后立刻执行任务，因此单个推理响应时延较短。

用户可以通过 json 配置文件中的 batch\_strategy 参数来选择 batch 策略。详情查看[inferencer 说明](#)。

### 5.3.5 推理引擎

推理引擎是推理服务中的核心部分，负责整个推理任务的调度执行等。用户可以通过增加推理引擎个数，增加推理并行度，从而提高推理性能。每增加一个推理引擎，便会 fork 一份推理模型的 Context，增加一定数量的线程数量以及申请一定大小的 MLU 内存用于存放模型的输入和输出数据。

MagicMind 模型默认占据 MLU 的所有核心，推理引擎数目设置为 1 或 2 最为合适，如果设置大于这个数目，性能可能不会提升，并且会占用更多的资源。

当两个及以上模块使用相同的推理任务时，如前处理、推理和后处理任务都使用相同的推理任务，将会共用相同的推理引擎。如果使用 dynamic 策略组 batch，这些模块的数据可能会在推理服务内部被组成一个 batch 进行推理任务。

#### 注解：

在两个及以上模块共用推理引擎时，推理引擎数目等于第一个接入推理服务的模块设置的推理引擎数目，其他模块的设置将无效。

### 5.3.6 使用说明及参数详解

例如在 vehicle\_recognition\_mlu370.json 配置文件，如下所示。该文件位于 samples/cns\_launcher/configs 目录下。

```
"classific" : {
  "class_name" : "cnstream::inferencer",    // （必设参数）推理类名。
  "parallelism" : 1,                        // （必设参数）并行度。
  "max_input_queue_size" : 20,              // （必设参数）最大队列深度。
  "next_modules" : ["osd"],                 // （可选参数）下一个连接模块的名称。
  "custom_params" : {                      // 特有参数。
    "model_path" : "../../../data/models/resnet50_v0.13.0_4b_rgb_uint8.magicmind",
    "preproc" : "name=PreprocClassification;use_cpu=false",
    "postproc" : "name=PostprocClassification;threshold=0.1",
    "filter" : "categoies=2,5,7",
    "batch_timeout" : 100,
    "engine_num" : 1,
    "model_input_pixel_format" : "RGB24",
    "device_id" : 0
  }
},
```

模块特有参数说明如下：

- model\_path：（必设参数）模型存放的路径。如设置为相对路径，则应该设置为相对于 JSON 配置文件的路径。

- **filter:** (二级推理时为必设参数) 指定目标种类或过滤器类名, 过滤器用来过滤在 `CNFrameInfo::collection` 中存储的检测目标, `Inferencer` 仅对过滤器指定的检测目标进行推理。自定义目标过滤器需要继承 `ObjectFilterVideo` 类, 并重写 `Filter` 接口。如果无需过滤, 可设置目标种类为 `all (ALL)` 或 `-1`。
- **postproc:** (必设参数) 后处理类名及阈值设定。详情参看[inferencer 后处理](#)。
- **preproc:** (必设参数) 前处理类名及是否使用 `cpu` 做前处理。详情参看[inferencer 前处理](#)。
- **device\_id:** (可选参数) 设备 id, 用于标识多卡机器的设备唯一编号。默认值为 0。
- **engine\_num:** (可选参数) 推理引擎个数。默认值 1。详情参看[推理引擎](#)。
- **batching\_timeout:** (可选参数) 组 batch 的超时时间, 单位为毫秒。只在 `batch_strategy` 为 `dynamic` 策略时生效。当超过指定的时间时, 该模块将直接进行推理不再继续等待, 未组满的部分数据则为随机值。一般应调整至大多数情况都能凑齐一组 batch 的时长, 以避免资源的浪费。默认值为 3000。
- **batch\_strategy:** (可选参数) 组 batch 的策略, 目前支持 `static (STATIC)` 和 `dynamic (DYNAMIC)` 两种。默认为 `dynamic` 策略。详情参看[batch 策略](#)。
- **priority:** (可选参数) 该模块在推理服务中的优先级。优先级只在同一设备上有效, 不同设备上的任务调度互不干扰。优先级限制为 0~9 的整数, 低于 0 的按 0 处理, 高于 9 的按 9 处理, 数值越大, 优先级越高。
- **show\_stats:** (可选参数) 是否显示推理服务内部的性能统计数据, 包括前后处理、推理的吞吐量、时延等。可设置为 `true` 或者 `false`。默认值为 `false`。
- **model\_input\_pixel\_format:** (可选参数) 模型输入的图像像素格式, 请根据模型进行选择。该参数可以设置为 `ARGB32`、`ABGR32`、`RGBA32`、`BGRA32`、`RGB24` 以及 `BGR24`。默认值为 `RGBA32`。用户可在自定义前处理类中通过 `OnTensorParams` 成员函数的参数 `params` 获得该值。获取方式请参看[inferencer 前处理](#)。

## 5.3.7 开发样例

### 5.3.7.1 自定义前处理开发样例

`CNStream` 中提供自定义前处理示例, 保存在 `samples/common/preprocess` 文件夹, 提供给用户参考。

前处理示例:

- **PreprocClassification** 类: 提供分类网络作为一级网络或者二级网络的前处理, 通过颜色空间转换及缩放以及补边 (输入保持宽高比), 将图片转换为适用离线网络的输入。定义在 `preprocess_classification.cpp` 文件中。
- **PreprocYolov3** 类: 提供 yolov3 网络的前处理。通过颜色空间转换, 缩放以及补边 (输入保持宽高比), 将图片转换为适用离线网络的输入。定义在 `preprocess_yolov3.cpp` 文件中。
- **PreprocYolov5** 类: 提供 yolov5 网络的前处理。通过颜色空间转换, 缩放以及补边 (输入保持宽高比), 将图片转换为适用离线网络的输入。定义在 `preprocess_yolov5.cpp` 文件中。
- **PreprocSSDLpd** 类: 提供 mobilenet ssd 网络的前处理。通过颜色空间转换, 缩放, 如模型输入数据类型为 `float` 则进行减均值除方差, 将图片转换为适用离线网络的输入。定义在 `preprocess_ssd_lpd.cpp`

文件中。

- **PreprocLprnet** 类：提供 lprnet 网络的前处理。通过颜色空间转换，缩放，将图片转换为适用离线网络的输入。定义在 preprocess\_lprnet.cpp 文件中。

### 5.3.7.2 自定义后处理开发样例

CNStream 中提供自定义后处理示例，保存在 samples/common/postprocess 文件夹，提供给用户参考：后处理示例：

- **PostprocClassification** 类：提供分类网络作为一级网络或者二级网络的后处理。定义在 postproc\_classification.cpp 文件中。
- **PostprocYolov3** 类：提供 yolov3 网络的后处理。定义在 postproc\_yolov3.cpp 文件中。
- **PostprocYolov5** 类：提供 yolov5 网络的后处理。定义在 postproc\_yolov5.cpp 文件中。
- **PostprocSSDLpd** 类：提供 mobilenet ssd 网络作为二级网络的后处理。定义在 postproc\_ssd\_lpd.cpp 文件中。
- **PostprocLprnet** 类：提供 lprnet 网络作为二级网络的后处理。定义在 postproc\_lprnet.cpp 文件中。

## 5.4 追踪模块

追踪模块（Tracker）用于对检测到的物体进行追踪。主要应用于车辆行人等检测物的追踪，需要连接在推理模块之后。目前支持 FeatureMatch 和 IoUMatch 追踪方法。

IoUMatch 通过计算 IoU（交并比）匹配检测目标，计算速度较 FeatureMatch 快。

FeatureMatch 主要通过特征值匹配检测目标，精度较 IoUMatch 高。在配置文件中指定提取特征使用的离线模型则使用推理服务在 MLU 提取特征值，若未配置模型相关参数则在 CPU 完成特征值提取。匹配算法可接收任意维度的特征值，仅要求一个 tracker 实例中使用的特征值维度保持一致。

### 5.4.1 特征值匹配方式

#### 5.4.1.1 FeatureMatch

FeatureMatchTrack 提供了基于检测结果的多目标追踪功能。输入是出现在某一帧上的全部待追踪检测目标，输出是与输入目标一一对应的追踪目标（DetectObject::detect\_id 标记与之对应的检测目标序号），顺序与输入目标不保证一致，未被确认为 CONFIRMED 的追踪目标 track\_id 标为-1。

追踪器对输入的多个检测目标与已知的追踪目标集进行匹配，未匹配到的检测目标将被加入到已知追踪目标集，并标记为 TENTATIVE，处于 TENTATIVE 状态的目标有一帧没有匹配到检测目标就会被删除，连续 n\_init 次匹配到就会转为 CONFIRMED 状态，并分配 track\_id，CONFIRMED 状态的目标连续 max\_age 帧未匹配到检测目标，则认为生命周期结束，被删除。

目标特征由一维 float 向量描述，长度一般是 128，也可以采用其他值，但一个 FeatureMatch 实例中必须使用统一长度的特征组。不同长度的特征向量无法进行相似度运算。

每一次追踪输入视为一次状态更新，只有每一帧都按顺序更新才能保证良好的追踪结果，当前帧无检测目标也需要送空的检测目标数组进行状态更新，跨帧或漏帧将会导致追踪精确度大幅下降。

追踪器可配置选项：

- 特征余弦距离阈值 `max_cosine_distance`，低于阈值的检测-追踪目标对可以被认为是同一目标。
- 特征集大小 `nn_budget`，每个缓存的追踪目标最多可保存 `nn_budget` 组特征，大特征集可稍微提高追踪精度，但也会增大内存占用和运算量。
- IoU 匹配阈值 `max_iou_distance`，低于阈值的检测-追踪目标对可以被认为是同一目标。
- 已确认追踪目标生命周期 `max_age`，CONFIRMED 状态的目标连续 `max_age` 次没有匹配到检测目标则生命周期结束。
- 目标确认次数 `n_init`，当目标连续 `n_init` 帧被追踪到时才会被标记为 CONFIRMED。

追踪效果优化方法：

1. 同一物体 `track_id` 跳变，可能由目标特征值变化大或两帧之间目标位置变化大导致，提升数据源帧率可以有效改善追踪效果，增大特征余弦值距离或 IoU 匹配阈值也可以一定程度上改善追踪效果，但过度增大阈值可能会导致不同物体匹配至同一个目标。`max_age` 默认值 30，有特征值时一般情况下不会因为偶尔漏检导致同一物体被识别为不同目标。
2. 不同物体匹配为同一个目标，可能由不同目标之间特征值差距过小或多个目标空间距离过近导致，降低特征余弦值距离或 IoU 阈值可以一定程度上改善追踪效果，但过度降低阈值可能会导致目标 `track_id` 跳变。特征值余弦距离与待追踪的数据集和采用的提取特征算法有关，建议针对待追踪目标数据集进行特征值余弦距离计算验证，设定合适的阈值。
3. 目标相遇时 `track_id` 交换，一般出现在无特征值输入，仅采用 IoU 匹配的情况下，给检测目标增加特征值可以有效降低 id 交换现象。
4. 物体出现在视界中的时间极短，可能仅有数帧，导致大部分情况下物体处于 TENTATIVE 状态，`track_id` 都是 -1，适当降低 `n_init` 可以有效改善追踪效果。但是当 `n_init` 小于 2 时，极有可能出现由检测错误导致生成无效追踪目标，具体取值应根据应用场景和检测精度而定。

### 5.4.2 使用说明及参数详解

配置追踪模块所需要的离线模型和追踪方法等。

例如在 `object_tracking_mlu370.json` 配置文件，如下所示。该文件位于 `samples/cns_launcher/configs` 目录下。

```
"tracker" : {
  "class_name" : "cnstream::Tracker",          // （必设参数）Track 的类名。
  "parallelism" : 1,                          // （必设参数）并行度。
  "max_input_queue_size" : 20,                // （必设参数）数据输入队列长度。
```

```
"custom_params" : {  
    "model_path" : "../../../data/models/feature_extract_v0.13.0_4b_rgb_uint8.  
↪magicmind",  
    "max_cosine_distance": "0.06",  
    "engine_num" : 2,  
    "device_id" : 0  
}  
}
```

模块特有参数说明如下：

- model\_path：（可选参数）模型存放的路径。如设置为相对路径，则应该设置为相对于 JSON 配置文件的路径。
- track\_name：（可选参数）追踪方法，目前支持 FeatureMatch 和 IoUMatch 追踪方法，默认值 FeatureMatch。
- max\_cosing\_distance：（可选参数）余弦距离匹配阈值，仅当两个目标特征值余弦距离不大于阈值时匹配生效。默认值为 0.2。
- model\_input\_pixel\_format：（可选参数）特征提取模型输入的颜色空间。默认值为 RGB24。
- priority：（可选参数）执行特征提取模型的优先级。默认值为 0。值越大优先级越高。
- engine\_num：（可选参数）提取特征的推理引擎个数。默认值 1。详情参看[推理引擎](#)。当追踪模块成为瓶颈时，可适当增加推理引擎数目。
- batching\_timeout：（可选参数）组 batch 的超时时间，单位为毫秒。当超过指定的时间时，该模块将直接进行推理不再继续等待，未组满的部分数据则为随机值。一般应调整至大多数情况都能凑齐一组 batch 的时长，以避免资源的浪费。默认值为 3000。
- show\_stats：（可选参数）是否显示推理服务内部的性能统计数据，包括前后处理、推理的吞吐量、时延等。可设置为 true 或者 false。默认值为 fasle。
- device\_id：（可选参数）设备 id，用于标识多卡机器的设备唯一编号。默认值为 0。仅在使能 MLU 提取特征时生效。

## 5.5 OSD 模块

OSD（On Screen Display）模块用于在图像上绘制对象。如使用硬件加速则将在原图上进行绘制，输出图像为 YUV 格式，如在 CPU 上进行绘制，则在 cv::Mat（可通过 CNDDataFrame::ImageBGR 接口获取）上绘制，并同步到原图。

OSD 模块可以连接在下面模块后面，绘制需要的推理结果：

- 推理模块（Inferencer）
- 追踪模块（Tracker）

OSD 模块后面可以连接下面模块，实现不同功能：



- 多媒体显示模块 (Vout)：对结果进行显示。(仅边缘端支持 Vout 模块)
- 编码模块 (Encode)：编码后保存成本地视频或者图片，也可进行 RTSP 推流。

### 5.5.1 使用说明及参数详解

例如在 `vehicle_recognition.json` 配置文件中配置一个包含二级网络的推理过程，即包含两个推理模块。该文件位于 `samples/cns_launcher/configs` 目录下。

```
"osd" : {  
  "class_name" : "cnstream::Osd",  
  "parallelism" : 4,  
  "max_input_queue_size" : 20,  
  "custom_params" : {  
    "label_path" : "../../../data/models/label_map_coco.txt",  
    "secondary_label_path" : "../../../data/models/synset_words.txt",  
    "attr_keys" : "classification"  
  }  
}
```

模块特有参数说明如下：

- `hw_accel`：(可选参数) 是否使用硬件加速，仅在边缘端生效。
- `label_path`：(可选参数) 标签路径。对应一级网络的标签路径。
- `font_path`：(可选参数) 自定义字体路径。支持显示中文字体。使用自定义字体依赖于 FreeType 开源库，可以参考 <https://www.freetype.org/> 安装。同时，需要打开编译选项 `WITH_FREETYPE`。示例中使用的 `wqy_zenhei.ttf` 字体文件需要自行下载。
- `label_size`：(可选参数) 标签大小，默认值为 `normal`。可设置的值包括：
  - `normal`：正常标签。
  - `large`：大标签。
  - `larger`：加大标签。
  - `small`：小标签。
  - `smaller`：较小标签。
  - 直接为数字，如 1、1.5、2 等。
- `text_scale`：(可选参数) 字体大小，默认值为 1。
- `text_thickness`：(可选参数) 字体宽度，默认值为 1。
- `box_thickness`：(可选参数) 标识框宽度，默认值为 1。设置 `label_size` 后可分别设置 `text_scale`、`text_thickness`、`box_thickness` 大小调节。也可以只设置 `label_size` 为其他缺省。
- `secondary_label_path`：(可选参数) 二级标签路径。对应二级网络的标签路径。
- `attr_keys`：(可选参数) 显示二级标签中某个关键特征。该属性必须结合二级网络的后处理过程。例如二级网络对车辆进行识别，识别出车的类别，车的颜色两个特征。同时在后处理时类别标记为 `classification`，颜色标记为 `color`。通过显示包含关键字 `classification` 可以输出车辆类别，也

可以同时包含 `classification` 和 `color` 输出类别和颜色两个标签。

- `logo`: (可选参数) 打印 logo 的名称。例如 `cambricon` 可以在每帧图像右下角添加名称为 `cambricon` 的水印。

`-osd_handler`: (可选参数) 支持自定义 `OsdHandler`，并通过此参数传入类名。可参考 `samples/common/osd_handler/osd_handler_vs.cpp` 文件中的 `OsdHandlerVS` 类。

## 5.6 VEncode 模块

VEncode 为编码模块，主要用于编码视频和图像。

编码模块可以连接在下面模块后面，对视频和图像进行编码：

- 数据源模块 (`cnstream::DataSource`)
- 推理模块 (`cnstream::Inferencer`)
- 追踪模块 (`cnstream::Tracker`)
- OSD 模块 (`cnstream::Osd`)

编码模块一般作为最后一个模块，后面不再连接其他模块。

### 5.6.1 使用说明及参数详解

例如在 `encode_video.json` 配置文件，如下所示。该文件位于 `samples/cns_launcher/configs/sinker_configs` 目录下。

```
"venc" : {  
  "class_name" : "cnstream::VEncode",  
  "parallelism" : 1,  
  "max_input_queue_size" : 10,  
  "custom_params" : {  
    "file_name" : "output/output.mp4",  
    "device_id" : 0  
  }  
}
```

模块特有参数说明如下：

- `dst_width`: (可选参数) 输出图像宽度，单位像素。输出图像宽度必须为偶数。
- `dst_height`: (可选参数) 输出图像高度，单位像素。输出图像高度必须为偶数。
- `frame_rate`: (可选参数) 编码后视频的帧率。默认值为 30。
- `bit_rate`: (可选参数) 单位时间内编码的数据量。默认值为 4000000bps，仅当在硬件上编码时才有效。较高的比特率表示视频质量较高，但相应编码速度较低。
- `gop_size`: (可选参数) 表示连续的画面组的大小。与两个关键帧 I-frame 相关。默认值 10。



- device\_id: (必选参数) 指定设备的 id, 默认值为 0。
- hw\_accel: (可选参数) 使用硬件或软件编码。默认使用硬件编码, 默认值为 true。
- view\_cols: (可选参数) 水平方向画布分割的区块数目, 默认值为 1。
- view\_rows: (可选参数) 竖直方向画布分割的区块数目, 默认值为 1。
- resample: (可选参数) 重采样, 默认值为 false。
- file\_name: (可选参数) 输出文件的路径和基础文件名。如设置文件名, 则将编码后的视频保存到本地文件。最终的文件名会加上 StreamID 形成 output\_StreamID.mp4。模块会依据文件名的后缀选择合适的编码器, 支持的后缀有 mp4, h264, h265, hevc, jpg, jpeg。
- rtsp\_port: (可选参数) RTSP 端口号。如设置端口号, 编码后的视频将被推流至互联网。URL 为 rtsp://xx.xx.xx.xx:port/live, xx.xx.xx.xx 代表服务器 ip, port 为端口号。推多路流时, port 递增。

## 5.7 Vout 模块

Vout 模块为多媒体展示模块。仅边缘端平台支持使用本模块。使用该模块可以略过视频流编码过程, 对 Pipeline 中处理完成的视频流进行实时的播放展示。

Vout 模块一般连接下面模块后面:

- 推理模块 (Inferencer)
- 追踪模块 (Tracker)
- OSD 模块 (Osd)

Vout 模块一般作为最后一个模块, 后面不再连接其他模块。

### 5.7.1 使用说明及参数详解

例如在 vout.json 配置文件, 如下所示。该文件位于 samples/cns\_launcher/configs/sinker\_configs 目录下。

```
"vout" : {  
  "class_name" : "cnstream::Vout",  
  "parallelism" : 4,  
  "max_input_queue_size" : 10,  
  "custom_params" : {  
  }  
}
```

## 5.8 Kafka 模块

Kafka 模块基于 librdkafka 将 CNFrameInfo 中结构化信息转换成 kafka 消息数据并发送到集群中供消费者消费。

Kafka 模块可以连接在下面模块后面：

- 数据源模块 (cnstream::DataSource)
- 推理模块 (cnstream::Inferencer)
- 追踪模块 (cnstream::Tracker)
- OSD 模块 (cnstream::Osd)

本模块一般作为最后一个模块，后面不再连接其他模块。

### 5.8.1 使用说明及参数详解

```
"kafka_producer" : {  
  "class_name" : "cnstream::Kafka",  
  "parallelism" : 1,  
  "max_input_queue_size" : 20,  
  "custom_params" : {  
    "handler" : "DefaultKafkaHandler",  
    "topic" : "CnstreamData",  
    "brokers" : "localhost:9092"  
  }  
}
```

模块特有参数说明如下：

- handler：（必选参数）将 CNFrameInfo 转换为 Kafka 消息数据的类的名称，模块内部提供示例类 DefaultKafkaHandler。
- brokers：（必选参数）消息代理服务:{SERVER\_IP}:{PORT}。如果 consumer 或 producer 与 broker 不处于同一台服务器，则需要在 kafka 配置文件中开启 server 的监听。
- topic：（可选参数）消息主题名称的前缀，与 streamID 拼接后形成真正的主题名称 topic\_streamID，默认值为 CnstreamData。

### 5.8.2 自定义 Handler

将 CNFrameInfo 转换为 Kafka 消息数据的过程非常定制化，每个使用者出于自身业务需求考虑都有特定的转换规则，所以 CNStream 提供了自定义处理接口。上文中提及的 DefaultKafkaHandler 是 CNStream 中自带的 Handler 供用户参考，它负责将数据流名称、帧号和推理所得的标签，检测框等信息序列化为如下所示的 JSON 字符串。

```
{
  "StreamName": "stream_0",
  "FrameCount": 119,
  "Objects": [
    {
      "Label": "2",
      "Score": 0.94384765625,
      "BBBox": [
        0.8779296875,
        0.4668387770652771,
        0.12060546875,
        0.5147867798805237
      ]
    },
    {
      "Label": "2",
      "Score": 0.62890625,
      "BBBox": [
        0.1103515625,
        0.8813707232475281,
        0.284423828125,
        0.11862927675247193
      ]
    }
  ]
}
```

用户自定义 Handler 通过继承 cnstream::KafkaHandler 实现，并重写 UpdateFrame 函数，代码模板如下：

```
class MyHandler : public cnstream::KafkaHandler {
public:
  ~MyHandler(){};

  int UpdateFrame(const cnstream::CNFrameInfoPtr &data) override;
```

```
DECLARE_REFLEX_OBJECT_EX(MyHandler, cnstream::KafkaHandler)
};

IMPLEMENT_REFLEX_OBJECT_EX(MyHandler, cnstream::KafkaHandler)
```

最后在 JSON 配置文件中将类名 `MyHandler` 与配置参数 `handler` 关联起来。



## 6 自定义模块

### 6.1 概述

CNStream 支持用户创建自定义模块。使用 CNStream 框架创建自定义模块非常简单，用户需要多重继承 **cnstream::Module** 和 **cnstream::ModuleCreator** 两个基类。其中 **cnstream::Module** 是所有模块的基类。**cnstream::ModuleCreator** 实现了反射机制，提供了 `CreateFunction`，并注册 `ModuleClassName` 和 `CreateFunction` 至 `ModuleFactory` 中。

### 6.2 自定义普通模块

这类模块支持多输入和多输出，数据由 pipeline 发送，并在模块的成员函数 `Process` 中处理。

```
class ExampleModule : public cnstream::Module, public cnstream::ModuleCreator<ExampleModule> {
    using super = cnstream::Module;

public:
    explicit ExampleModule(const std::string &name) : super(name) {}
    bool Open(cnstream::ModuleParamSet param_set) override {
        // Your codes.
        return true;
    }
    void Close() override { std::cout << this->GetName() << " Close called" << std::endl; }
    int Process(std::shared_ptr<cnstream::CNFrameInfo> data) override {
        // Your codes.
        return 0;
    }

private:
    ExampleModule(const ExampleModule &) = delete;
    ExampleModule &operator=(ExampleModule const &) = delete;
};
```

## 6.3 自定义数据源模块

数据源模块与普通模块基本类似，唯一不同的是这类模块没有输入只有输出，所以模块的成员函数 `Process` 不会被框架所调用。

```
class ExampleModuleSource : public cnstream::SourceModule, public cnstream::ModuleCreator
{
public:
    explicit ExampleModuleSource(const std::string &name) : super(name) {}
    bool Open(cnstream::ModuleParamSet param_set) override {
        // Your codes.
        return true;
    }
    void Close() override { std::cout << this->GetName() << " Close called" << std::endl; }
    int Process(std::shared_ptr<cnstream::CNFrameInfo> data) override {
        std::cout << "For a source module, Process() will not be invoked\n";
        return 0;
    }

private:
    ExampleModuleSource(const ExampleModuleSource &) = delete;
    ExampleModuleSource &operator=(ExampleModuleSource const &) = delete;
};
```

## 6.4 自定义扩展模块

这类模块支持多输入和多输出。继承自 `cnstream::ModuleEx` 和 `cnstream::ModuleCreator` 类。与普通模块不同，处理过的数据由模块自行送入下一级模块。此类模块的一个典型应用是在模块内部攒 batch，然后再进行批量处理。

```
class ExampleModuleEx : public cnstream::ModuleEx, public cnstream::ModuleCreator
{
public:
    explicit ExampleModuleEx(const std::string &name) : super(name) {}
    bool Open(cnstream::ModuleParamSet param_set) override {
```

```
// Your codes.
return true;
}
void Close() override {
    // Your codes.
}
int Process(FrameInfoPtr data) override {
    // Your codes.
    // Note that data transmitted by the module self
    TransmitData(data);
    return 1;
}

private:
    ExampleModuleEx(const ExampleModuleEx &) = delete;
    ExampleModuleEx &operator=(const ExampleModuleEx &) = delete;
};
```

## 7 创建应用程序

### 7.1 概述

基于 CNStream 创建应用程序，实际上是基于 CNStream 内置模块和用户自定义模块搭建业务流水线。用户可以选择使用配置文件方式或非配置文件方式创建应用程序。

配置文件方式与非配置文件方式的主要区别在于，配置文件使用 JSON 文件格式声明 pipeline 结构、模块上下游关系和模块参数等，而非配置文件则需要开发者创建模块对象，设置模块参数和模块上下游关系等。相对而言，配置文件方式更加灵活，推荐使用。开发者编写 pipeline 基本骨架后，仍可以灵活地调整配置文件中的模块参数甚至结构，而无需重新编译。

### 7.2 应用程序的创建

#### 7.2.1 配置文件方式

在配置文件方式下，用户开发应用时需要关注两部分：JSON 配置文件的编写和 pipeline 基本骨架的构建。

##### 7.2.1.1 JSON 配置文件的编写

JSON 配置文件主要用于声明 pipeline 中各个模块的上下游关系及其每个模块内部的参数配置。

下面示例展示了如何使用 CNStream 提供的内置模块 DataSource、Inferencer、Tracker、Osd、VEncode，以及 yolov3 和 track 离线模型，实现一个典型的 pipeline 操作。

典型的 pipeline 操作为：

1. 视频源解析和解码。
2. 物体检测。
3. 追踪。
4. 在视频帧上，叠加绘制的物体检测信息框。
5. 编码输出视频。

配置文件示例如下：



```

{
  "profiler_config" : {
    "enable_profiling" : true,
    "enable_tracing" : true
  },

  "source" : {
    // 数据源模块。设置使用 ffmpeg 进行 demux，使用硬件解码，pipeline 不负责该模块的线程启动。
    "class_name" : "cnstream::DataSource",
    "parallelism" : 0,
    "next_modules" : ["detector"],
    "custom_params" : {
      "interval" : 1,
      "device_id" : 0
    }
  },

  "detector" : {
    // Inferencer 模块。设置使用 yolov3 离线模型，使用 PostprocYolov3 进行网络输出数据后处理，推理引擎数为 4，并行度为 1，模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::Inferencer",
    "parallelism" : 1,
    "max_input_queue_size" : 20,
    "next_modules" : ["tracker"],
    "custom_params" : {
      "model_path" : "../data/models/yolov3_4b_rgb_uint8.magicmind",
      "preproc" : "name=PreprocYolov3;use_cpu=false",
      "postproc" : "name=PostprocYolov3;threshold=0.5",
      "batching_timeout" : 200,
      "engine_num" : 4,
      "model_input_pixel_format" : "RGB24",
      "device_id" : 0
    }
  },

  "tracker" : {
    // Tracker 模块。设置使用 track 离线模型，推理引擎数为 4，并行度为 1，模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::Tracker",
    "parallelism" : 1,
    "max_input_queue_size" : 20,
    "next_modules" : ["osd"],

```

```

    "custom_params" : {
        "model_path" : "../data/models/feature_extract_4b_rgb_uint8.magicmind",
        "engine_num" : 4,
        "device_id" : 0
    }
},

"osd" : {
    // Osd 模块。配置解析 label 路径, 设置并行度为 2, 模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::Osd",
    "parallelism" : 2,
    "max_input_queue_size" : 20,
    "next_modules" : ["encoder"],
    "custom_params" : {
        "hw_accel": "false",
        "label_path" : "../data/models/label_map_coco.txt"
    }
},

"encoder" : {
    // Encoder 模块。配置输出视频的 dump 路径, 设置并行度为 2, 模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::VEncode",
    "parallelism" : 2,
    "max_input_queue_size" : 20,
    "custom_params" : {
        "dst_width": 1280,
        "dst_height": 720,
        "frame_rate" : 25,
        "file_name" : "../output/output.mp4",
        "device_id": 0
    }
}
}
}

```

同时支持图嵌套结构创建业务流程, 每个子图以 subgraph 关键词标示, 配置文件示例如下:

```

{
    "profiler_config" : {
        "enable_profiling" : true,
        "enable_tracing" : true
    },
}

```

```
"subgraph:decode" : {
  "config_path" : "../configs/decode_config.json",
  "next_modules" : ["subgraph:object_detection"]
},

"subgraph:object_detection" : {
  "config_path" : "../configs/yolov3_object_detection.json",
  "next_modules" : ["subgraph:object_tracking"]
},

"subgraph:object_tracking" : {
  "config_path" : "../configs/object_tracking.json",
  "next_modules" : ["subgraph:osd_label_map_coco"]
},

"subgraph:osd_label_map_coco" : {
  "config_path" : "../configs/osd_configs/osd_label_map_coco.json",
  "next_modules" : ["subgraph:sinker"]
},

"subgraph:sinker" : {
  "config_path" : "../configs/sinker_configs/encode_video.json"
}
}
```

用户可以参考以上 JSON 的配置构建自己的配置文件。另外，CNStream 提供了 inspect 工具来查询每个模块支持的自定义参数。详情查看[Inspect 工具](#)。

### 7.2.1.2 Pipeline 基本骨架的构建

构建 pipeline 核心骨架包括：搭建整体业务流水线和设置事件监听处理机制。

在配置文件方式下，搭建整体的业务流水线实际是从预准备的 JSON 文件中获取 pipeline 结构、module 上下游关系和各个 module 的参数，并初始化各个任务执行环节，即模块。另外，用户可以通过设置事件监听获取 pipeline 的处理状态，添加对应的状态处理机制，如 eos 处理、错误处理等。

整个过程主要包括下面步骤：

1. 创建 pipeline 对象。
2. 调用 Pipeline::BuildPipelineByJSONFile，使用预准备的 JSON 配置文件构建整体业务流水线。
3. 调用 Pipeline::SetStreamMsgObserver，设置事件监听处理机制。
4. 调用 Pipeline::Start()，启动 pipeline。
5. 调用 DataSource::AddSource() 或 DataSource::RemoveSource()，动态添加或删除视频和图片源。

源代码示例，可参考 CNStream 源码中 `samples/cns_launcher/cns_launcher.cpp`。

## 7.3 用户侧 MessageHandle

用户程序可以通过注册的事件监听监测 Pipeline 的 Message 信息，目前定义的用户侧 Message 信息包括 EOS\_MSG、FRAME\_ERR\_MSG、STREAM\_ERR\_MSG、ERROR\_MSG(参见 StreamMsgType 定义)。

各消息处理示例可以参考 CNStream 源代码 `samples/cns_launcher/cns_launcher.cpp`。

### 7.3.1 1. EOS\_MSG

EOS\_MSG 表示 Pipeline 数据处理结束，接收到该消息时，可以正常结束 Pipeline 释放资源等。

### 7.3.2 2. FRAME\_ERR\_MSG

FRAME\_ERR\_MSG 表示帧解码失败消息，当前仅支持使用硬件解码 JPEG 图片场景：

(1) JPEG 图片文件形式时，用户侧接收到 FRAME\_ERR\_MSG 消息时，可以同时获取解码错误的图片帧信息，包含用户侧定义的 `stream_id` 和内部赋值定义的 `pts`、`frame_id` 信息；

接收到这些信息后，用户侧可以根据自己的业务逻辑处理解码失败的图片帧，比如丢弃、记录等。

### 7.3.3 3. STREAM\_ERR\_MSG

STREAM\_ERR\_MSG 表示某一路数据发生不可恢复错误，通常包括超过内存限制导致的解码器申请失败等。

用户侧接收到该信息时，若希望 Pipeline 继续进行，将出现错误的数据流移除掉即可（使用 Source 模块的 `RemoveSource` 方法进行特定数据流的卸载），该操作不影响其他正常处理的数据流。

### 7.3.4 4. ERROR\_MSG

ERROR\_MSG 表示普通的错误信息，目前表示不可恢复错误，建议直接停止 Pipeline，并根据 log 信息进行错误定位。

## 8.1 Inspect 工具

Inspect 工具是 CNStream 提供的一个用来扫描模块以及检查配置文件的工具。主要功能包括：

- 查看框架支持的所有模块。
- 查看某个模块在使用时需要用到的参数。
- 检查配置文件的合法性。
- 打印 CNStream 的版本信息。

如果使用自定义模块，用户需要先注册自定义的模块，才能使用该工具。详情请参照[配置 Inspect 工具](#)。

### 8.1.1 工具命令的使用

安装和配置环境依赖和依赖库后，输入下面命令进入工具所在目录：

```
cd $CNSTREAM_HOME/tools/bin
```

bin 目录是编译成功后创建的。

#### 8.1.1.1 打印工具帮助信息

输入下面命令打印工具帮助信息：

```
./cnstream_inspect -h
```

命令返回如下内容：

```
Usage:
  inspect-tool [OPTION...] [MODULE-NAME]
Options:
  -h, --help                Show usage
  -a, --all                  Print all modules
  -m, --module-name          List the module parameters
  -v, --version              Print version information
```

### 8.1.1.2 查看框架支持的所有模块

输入下面命令查看框架支持的所有模块：

```
./cnstream_inspect -a
```

命令返回示例如下：

Module Name	Description
cnstream::DataSource	DataSource is a module for handling input data.
...	...

### 8.1.1.3 查看某个模块的参数

输入下面命令查看某个模块的参数，以 DataSource 为例：

```
./cnstream_inspect -m DataSource
```

命令返回示例如下：

```
DataSource Details:
Common Parameter      Description
class_name            Module class name.
...                  ...
```

### 8.1.1.4 打印 CNStream 的版本信息

输入下面命令打印 CNStream 的版本信息：

```
./cnstream_inspect -v
```

命令返回示例如下，版本号为 CNStream 最新版本号：

```
CNStream: vx.y.z
```

### 8.1.2 配置 Inspect 工具

执行下面步骤完成自定义模块工具的配置。CNStream 内置模块无需配置，直接调用工具相关指令即可。

1. 每个自定义的模块在声明时，需要继承 **Module** 和 **ModuleCreator** 类，以 Encode 模块为例：

```
class VEncode: public Module, public ModuleCreator<VEncode> {
    ...
}
```

2. 添加自定义模块的描述信息。param\_register\_ 是 ParamRegister 类型的 **Module** 类的成员变量，以 VEncode 模块为例。

```
param_register_.SetModuleDesc("VEncode is a module for encoding video or images.");
```

3. 注册自定义模块所支持的参数。param\_register\_ 是 ParamRegister 类型的 **Module** 类的成员变量，以 VEncode 模块为例。

```
param_register_.Register("param_name", "param description");
```

4. 声明 **ParamRegister** 类。

```
class ParamRegister {
private:
    std::vector<std::pair<std::string /*key*/, std::string /*desc*/>> module_params_;
    std::string module_desc_;
public:
    void Register(const std::string &key, const std::string &desc); // 注册函数。
    // 通过该接口获取子模块已注册的参数。
    std::vector<std::pair<std::string, std::string>> GetParams();
    // 判断 key 是否是已注册的。也可以判断配置文件中是否配置了 module 不支持的参数。
    bool IsRegistered(const std::string& key);
    void SetModuleDesc(const std::string& desc); // 设置模块描述。
};
```

## 8.2 性能统计

CNStream 提供性能统计机制帮助用户分析程序的性能，主要包括 pipeline 各部分的时延和吞吐信息。

除此之外，CNStream 还提供自定义时间统计机制，帮助用户在自定义模块中，统计模块内部性能数据，并最后与整体性能数据汇总，帮助分析 pipeline 性能瓶颈。

性能统计机制的实现和定义在 CNStream 源代码目录下 framework/core/include/profiler 和 framework/core/src/profiler 目录中。

### 8.2.1 机制原理

性能统计通过在 pipeline 中各个处理过程的开始和结束点上打桩，记录打桩时间点，并基于打桩时间点来计算时延、吞吐等信息。

性能统计机制以某个处理过程为对象进行性能统计。一个处理过程可以是一个函数调用、一段代码，或是 pipeline 中两个处理节点之间的过程。在 CNStream 的性能统计机制中，每个处理过程通过字符串进行映射。

对于每个 pipeline 实例，通过创建一个 PipelineProfiler 实例来管理该 pipeline 中性能统计的运作。通过 PipelineProfiler 实例为 pipeline 中的每个模块创建一个 ModuleProfiler 实例来管理模块中性能统计。并通过 ModuleProfiler 的 RegisterProcessName 接口注册需要进行性能统计的处理过程，使用 ModuleProfiler::RecordProcessStart 和 ModuleProfiler::RecordProcessEnd 在各处理过程的开始和结束时间点打桩，供 CNStream 进行性能统计。

### 8.2.2 开启性能统计功能

性能统计功能默认是关闭状态。要打开性能统计功能，有以下两种方式：

- 通过 Pipeline::BuildPipeline 接口构建 pipeline，通过设置 enable\_profiling 参数为 **true** 打开性能统计功能。示例代码如下：

```
cnstream::Pipeline pipeline;  
cnstream::ProfilerConfig profiler_config;  
profiler_config.enable_profiling = true;  
pipeline.BuildPipeline(module_configs, profiler_config);
```

- 通过配置文件的方式构建 pipeline，即使用 Pipeline::BuildPipelineByJSONFile 接口构建 pipeline。在 json 格式的配置文件中设置 enable\_profiling 参数为 **true**，打开性能统计功能。  
示例配置文件如下。profiler\_config 项中 enable\_profiling 子项被置为 true，表示打开 pipeline 的性能统计功能。module1 和 module2 为两个模块的配置。pipeline 配置文件中 profiler\_config 项和模块配置项之间没有顺序要求。

```
{  
  "profiler_config" : {  
    "enable_profiling" : true  
  },  
  
  "module1" : {  
    ...  
  },  
  
  "module2" : {
```



```
...
}
}
```

**注意：**

打开性能统计功能仅仅表示后台在统计性能，至于如何在终端呈现性能数据则由 App 决定，可以参考[profiling 示例代码](#)。

### 8.2.3 内置处理过程

由于模块输入队列和模块 Process 函数的性能统计通常至关重要，CNStream 对 pipeline 中每个模块的输入队列和模块 Process 函数提供性能统计。CNStream 内部通过 PipelineProfiler 将这个两个处理过程注册为两个默认的处理过程。用户只需[开启性能统计功能](#)，即可查看模块输入队列的性能数据和 Process 函数的性能数据。

#### 8.2.3.1 获取模块输入队列性能数据

对于模块的输入队列，下面两个时间点会被打桩：

- 数据入队的时间点。
- 数据出队的时间点。

通过记录数据进出队列的时间节点，来统计模块输入队列的性能数据。

数据通过模块输入数据队列的过程作为一个处理过程，被 CNStream 默认注册在各模块的性能统计功能中。常量字符串 `cnstream::kINPUT_PROFILER_NAME` 作为这一处理过程的标识，可通过该字符串获取插件输入队列相关的性能数据。

使用该常量字符串，参考[获取指定处理过程的性能数据](#)即可获得模块输入数据队列的性能数据。

#### 8.2.3.2 获取模块 Process 函数性能数据

对于模块的 Process 函数，下面两个时间点会被打桩：

- 调用 Process 函数之前一刻记录开始时间。
- 数据经过 Process 处理，并且调用 TransmitData 接口时，记录时间作为 Process 结束时间。

通过记录这两个时间节点，来统计模块 Process 函数的性能数据。

数据通过模块 Process 函数的过程作为一个处理过程，被 CNStream 默认注册在各模块的性能统计功能中。常量字符串 `cnstream::kPROCESS_PROFILER_NAME` 作为这一处理过程的标识，可通过该字符串获取模块 Process 函数的性能数据。

使用该常量字符串，参考[获取指定处理过程的性能数据](#)即可获得 Process 函数的性能数据。

### 8.2.4 统计模块的性能

CNStream 通过 `ModuleProfiler::RegisterProcessName` 函数来自定义模块的性能统计。

通过 `ModuleProfiler::RegisterProcessName` 函数传入一个字符串，这个字符串用来标识某一个处理过程。在调用 `ModuleProfiler::RecordProcessStart` 和 `ModuleProfiler::RecordProcessEnd` 时，通过传入这个字符串，来标识当前是对哪个处理过程进行性能统计。

以下用自定义模块来模拟使用流程：

1. 开启性能统计功能。
2. 在自定义模块的 `Open` 函数中调用 `ModuleProfiler::RegisterProcessName` 注册一个自定义性能统计过程。示例代码如下：

```
static const std::string my_process_name = "AffineTransformation";

bool YourModule::Open(ModuleParamSet params) {
    ModuleProfiler* profiler = this->GetProfiler();
    if (profiler) {
        if (!profiler->RegisterProcessName(my_process_name)) {
            LOGE(YourModule) << "Register [" << my_process_name << "] failed.";
            return false;
        }
    }
    return true;
}
```

#### 注意：

`ModuleProfiler::RegisterProcessName` 函数中传递的字符串应保证唯一性，即已经注册使用过的字符串不能再次被注册使用，否则注册将失败，接口返回 `false`。

`cnstream::kPROCESS_PROFILER_NAME` 和 `cnstream::kINPUT_PROFILER_NAME` 两个字符串已经被 CNStream 作为模块 `Process` 函数和模块输入队列的性能统计标识注册使用，请不要再使用同名字符串。

3. 在需要进行性能统计的代码前后分别调用 `ModuleProfiler::RecordProcessStart` 和 `ModuleProfiler::RecordProcessEnd`。下面以统计 `AffineTransformation` 函数的性能数据为例，在 `AffineTransformation` 函数前后打桩。

```
void AffineTransformation(std::shared_ptr<cnstream::CNFrameInfo> frame_info);

int YourModule::Process(std::shared_ptr<cnstream::CNFrameInfo> frame_info) {
    ...

    cnstream::RecordKey key = std::make_pair(frame_info->stream_id, frame_info->timestamp);
```

```
if (this->GetProfiler()) {
    this->GetProfiler()->RecordProcessStart(my_process_name, key);
}

AffineTransformation(frame_info);

if (this->GetProfiler()) {
    this->GetProfiler()->RecordProcessEnd(my_process_name, key);
}

...

return 0;
}
```

代码中，key 为一帧数据的唯一标识，由 CNFrameInfo 结构中的 stream\_id 字段和 timestamp 字段构成。

4. 使用注册处理过程时的字符串，获取自定义处理过程的性能统计数据。详情参考[获取指定处理过程的性能数据](#)。

### 8.2.5 Pipeline 端到端的性能统计

pipeline 端到端的性能统计，在数据进入 pipeline 和数据离开 pipeline 两个时间点分别记录时间，来统计性能。不包括统计 pipeline 中各模块、各处理过程等。用户可以通过 PipelineProfiler 实例来完成性能统计。

pipeline 端到端的性能统计结果存放在 PipelineProfile::overall\_profile 中。详情查看[获取 Pipeline 整体性能数据](#)。

### 8.2.6 获取性能统计结果

#### 8.2.6.1 获取 Pipeline 整体性能数据

pipeline 整体性能数据的统计包括各模块、各处理过程、各数据流以及 pipeline 端到端的性能统计结果。从时间轴上可以分为：从开始到结束的性能数据和某一个时间段的性能数据。

通过 PipelineProfiler 提供的 GetProfile 重载函数、GetProfileBefore、GetProfileAfter 函数以获取 pipeline 的整体性能统计结果。这些函数都返回类型为 PipelineProfile 的数据。

### 获取从开始到结束的性能数据

通过 PipelineProfiler::GetProfile 的无参数版本函数用来获取从 pipeline 开始执行到 pipeline 停止执行这段时间内的性能数据。

使用示例:

```
cnstream::PipelineProfile profile = pipeline.GetProfile();
```

#### 注意:

- 要使用上述接口获取性能数据需要打开性能统计功能，性能统计功能打开方式请参阅[开启性能统计功能](#)。
- 若未正确打开性能统计功能，调用上述接口将返回空数据。

### 获取某一个时间段的性能数据

通过 PipelineProfiler::GetProfile 的两个参数版本函数和 PipelineProfiler::GetProfileBefore 以及 PipelineProfiler::GetProfileAfter 三个函数用来获取 pipeline 执行过程中某一段时间的性能数据。

以下提供使用两个参数版本的 PipelineProfiler::GetProfile 的使用示例，来获取 start 到 end 之间这段时间内的性能统计结果。其它两个接口的使用说明请参阅头 framework/core/include/profiler/pipeline\_profiler.hpp 文件声明或参考《寒武纪 CNStream 开发者手册》。

```
cnstream::Time start = cnstream::Clock::now();  
sleep(2);  
cnstream::Time end = cnstream::Clock::now();  
  
cnstream::PipelineProfile profile = pipeline.GetProfile(start, end);
```

#### 注意:

- 要使用上述三个接口获取指定时间段的性能数据，需要打开性能统计功能和数据流追踪功能。打开方式请参阅[开启性能统计功能](#)及[开启数据流追踪功能](#)。
- 若未正确打开性能统计功能，调用上述接口将返回空数据。
- 若未正确打开追踪功能，调用上述接口将返回空数据，并打印一条 WARNING 级别的日志。

### 8.2.6.2 获取 pipeline 端到端的性能数据

PipelineProfile 结构中的 overall\_profile 字段存储了数据从进入 pipeline 到离开 pipeline 这个过程的性能数据，被用来评估 pipeline 处理数据的能力。

overall\_profile 字段的类型为 ProcessProfile，其中带有吞吐、处理的数据帧数量、时延等一系列用来评估 pipeline 性能的数据。详情可参考 framework/core/include/profiler/profile.hpp 头文件或者《寒武纪 CNStream 开发者手册》中对 ProcessProfile 结构体的说明。

### 8.2.6.3 获取指定模块的性能数据

PipelineProfile 结构中的 module\_profiles 字段存储了所有模块的性能数据。

它的类型为 std::vector<ModuleProfile>。ModuleProfile::module\_name 中存储着模块名字，要获取指定模块的性能数据可通过模块名字从 module\_profiles 中查找。

示例代码如下：

```
cnstream::PipelineProfile pipeline_profile = pipeline.GetProfile();
const std::string my_module_name = "MyModule";
cnstream::ModuleProfile my_module_profile;
for (const cnstream::ModuleProfile& module_profile : pipeline_profile.module_profiles) {
    if (my_module_name == module_profile.module_name) {
        my_module_profile = module_profile;
        break;
    }
}
```

### 8.2.6.4 获取指定处理过程的性能数据

ModuleProfile 结构中的 process\_profiles 存放着模块注册的所有处理过程的性能数据，包括两个[内置处理过程](#)的性能统计结果和自定义处理过程的性能统计结果。

process\_profiles 的类型为 std::vector<ProcessProfile>。ProcessProfile::process\_name 为注册处理过程时提供的处理过程唯一标识字符串。

要获取指定处理过程的性能数据可通过处理过程的唯一标识字符串来查找。

示例代码如下：

```
cnstream::ModuleProfile module_profile;
const std::string my_process_name = "AffineTransformation";
cnstream::ProcessProfile my_process_profile;
for (const cnstream::ProcessProfile& process_profile : module_profile.process_profiles) {
    if (process_profile.process_name == my_process_name) {
```

```

    my_process_profile = process_profile;
    break;
}
}

```

ProcessProfile 结构中还存在有吞吐速度、时延、最大最小时延、处理的数据帧数目、丢弃的数据帧数目等性能参考数据。详情可查看 framework/core/include/profiler/profile.hpp 或参看《寒武纪 CNStream 开发者手册》中对该结构的说明。

#### 8.2.6.5 获取每一路数据流的性能数据

每个处理过程都包含经过这个处理过程的所有数据流的性能数据。存放于 ProcessProfile::stream\_profiles 中。

ProcessProfile::stream\_profiles 的类型为 std::vector<StreamProfile>。

StreamProfile::stream\_name 即往 pipeline 中加入数据流时指定的数据流名称。

StreamProfile 结构中还存在有吞吐速度、时延、最大最小时延、处理的数据帧数目、丢弃的数据帧数目等性能参考数据。详情可查看 framework/core/include/profiler/profile.hpp 或《寒武纪 CNStream 开发者手册》中对该结构的说明。

示例代码如下：

```

cnstream::ProcessProfile process_profile;
for (const cnstream::StreamProfile& stream_profile : process_profile.stream_profiles) {
    // stream_profile.stream_name : stream id.
    // stream_profile.fps : throughput.
    // stream_profile.latency : average latency
}

```

### 8.2.7 性能统计数据说明

性能统计功能的基本对象是一个处理过程。对于每个处理过程，会统计总体的性能数据并存放在 ProcessProfile 结构的各字段中。每个处理过程还会分别统计每路数据流经过该处理过程的性能数据，存放在 ProcessProfile 结构的 stream\_profiles 字段中。

每路数据流的性能由 StreamProfile 结构表示，内部的性能数据与 ProcessProfile 结构中表示性能数据的字段名与含义一致，ongoing 字段除外，它只存在于 ProcessProfile 结构中，StreamProfile 中不统计这个性能数据。

ProcessProfile 中各字段及其表示的含义如下：

表 8.1: 性能统计字段说明

字段名称	描述
completed	表示已经处理完毕的数据总量，不包括丢弃的数据帧。
dropped	表示被丢弃的数据总量。 当一个数据记录了开始时间，但是比它更后记录开始时间的数据已经结束了超过 16 个（取自 h.264、h.265 spec 中的 MaxDpbSize），则视为该数据帧已经丢弃。例如一个模块中存在丢帧逻辑，则会出现数据经过模块的 Process 函数，但是 TransmitData 不会被调用的情况，此时则会把这样的数据帧数量累加到 dropped 字段上。
counter	表示统计到的对应处理过程已经处理完毕的数据的总量。被丢弃的数据也视为处理完毕的数据，会被累加在到 counter 上。 $counter = completed + dropped$ 。
ongoing	表示正在处理，但是未被处理完毕的数据总量。即已经记录到开始时间但是未记录到结束时间的数据总量。
latency	平均时延，单位为毫秒。
maximum_latency	最大处理时延，单位为毫秒。
minimum_latency	最小处理时延，单位为毫秒。
fps	平均吞吐速度，单位为帧/秒。

### 8.2.8 示例代码

CNStream 提供示例代码存放在 `samples/cns_launcher/cns_launcher.cpp` 中。该示例展示了如何每隔两秒获取一次性能数据，并且打印完整的性能数据和最近两秒的性能数据。

`samples/bin/cns_launcher` 可执行文件中使用 `perf_level` 参数控制打印的性能数据的详细程度。

`perf_level` 可选值有 [0, 1, 2, 3]，默认值为 0：

- 当 `perf_level` 为 0 时，只打印各处理过程的 `counter` 统计值与 `fps`（吞吐）统计值。
- 当 `perf_level` 为 1 时，在 0 的基础上加上 `latency`、`maximum_latency`、`minimum_latency` 三个统计值的打印。
- 当 `perf_level` 为 2 时，打印 `ProcessProfile` 结构中的所有性能统计值。
- 当 `perf_level` 为 3 时，在 2 的基础上打印每路数据流的性能统计数据。

### 8.2.8.1 完整性能数据示例

完整性能打印示例如下：

```
***** Performance Print Start (Whole) *****
===== Pipeline: [MyPipeline] =====
----- Module: [displayer] -----
-----Process Name: [INPUT_QUEUE]
[Counter]: 592, [Throughput]: 35118.1fps
-----Process Name: [PROCESS]
[Counter]: 592, [Throughput]: 135526fps
----- Module: [osd] -----
-----Process Name: [INPUT_QUEUE]
[Counter]: 592, [Throughput]: 748.563fps
-----Process Name: [PROCESS]
[Counter]: 592, [Throughput]: 680.162fps
----- Module: [source] -----
-----Process Name: [PROCESS]
[Counter]: 597, [Throughput]: 59.7144fps
----- Module: [detector] ----- (slowest)
-----Process Name: [RUN MODEL]
[Counter]: 592, [Throughput]: 444.986fps
-----Process Name: [RESIZE CONVERT]
[Counter]: 592, [Throughput]: 6569.07fps
-----Process Name: [PROCESS]
[Counter]: 592, [Throughput]: 59.6681fps
-----Process Name: [INPUT_QUEUE]
[Counter]: 597, [Throughput]: 11810.1fps

----- Overall -----
[Counter]: 592, [Throughput]: 59.2285fps
***** Performance Print End (Whole) *****
```

### 8.2.8.2 最近两秒的性能数据打印示例

最近两秒的性能数据打印示例如下：

```
***** Performance Print Start (Last two seconds) *****
===== Pipeline: [MyPipeline] =====
----- Module: [displayer] -----
-----Process Name: [PROCESS]
[Counter]: 112, [Throughput]: 134805fps
```



```

-----Process Name: [INPUT_QUEUE]
[Counter]: 112, [Throughput]: 35815.7fps
-----Module: [osd] -----
-----Process Name: [PROCESS]
[Counter]: 112, [Throughput]: 686.523fps
-----Process Name: [INPUT_QUEUE]
[Counter]: 112, [Throughput]: 753.897fps
-----Module: [source] -----
-----Process Name: [PROCESS]
[Counter]: 119, [Throughput]: 61.1041fps
-----Module: [detector] ----- (slowest)
-----Process Name: [RUN MODEL]
[Counter]: 112, [Throughput]: 443.628fps
-----Process Name: [RESIZE CONVERT]
[Counter]: 112, [Throughput]: 6710.72fps
-----Process Name: [INPUT_QUEUE]
[Counter]: 119, [Throughput]: 20385.1fps
-----Process Name: [PROCESS]
[Counter]: 112, [Throughput]: 56.9688fps

-----Overall -----
[Counter]: 112, [Throughput]: 56.7687fps
***** Performance Print End (Last two seconds) *****

```

## 8.3 数据流的追踪

CNStream 提供数据流追踪机制用于帮助分析程序执行流程。CNStream 的追踪机制主要记录数据在 pipeline 中流过程中各节点发生的时间、类型（开始、结束）、事件级别（模块级别、pipeline 级别）、事件名称等信息。

### 8.3.1 开启数据流追踪功能

数据追踪功能默认是关闭状态。要打开数据追踪功能，有以下两种方式：

- 通过 Pipeline::BuildPipeline 函数构建 pipeline，将 profiler\_config.enable\_tracing 参数设为 **true** 开启数据追踪功能。并设置追踪数据占用的内存空间参数 trace\_event\_capacity。示例代码如下：

```

cnstream::Pipeline pipeline;
cnstream::ProfilerConfig profiler_config;
profiler_config.enable_tracing = true;

```

```
profiler_config.trace_event_capacity = 100000;
pipeline.BuildPipeline(module_configs, profiler_config);
```

- 通过配置文件的方式构建 pipeline,即使用 Pipeline::BuildPipelineByJSONFile 函数构建 pipeline。在 json 格式的配置文件中填入数据追踪相关参数, 打开数据追踪功能。

示例配置文件如下。在 profiler\_config 项中 enable\_tracing 子项被置为 **true**, 表示打开 pipeline 的数据追踪功能。并设置追踪数据占用的内存空间参数 trace\_event\_capacity。module1 和 module2 为两个模块的配置。pipeline 配置文件中 profiler\_config 项和模块配置项之间没有顺序要求。

```
{
  "profiler_config" : {
    "enable_tracing" : true,
    "trace_event_capacity" : 100000
  },

  "module1" : {
    ...
  },

  "module2" : {
    ...
  }
}
```

### 8.3.2 事件记录方式

在数据流追踪中, 事件的记录方式分为两种。

- 数据流追踪的同时, 对事件进行性能分析, 记录各处理过程的开始事件和结束事件。即在 ModuleProfiler::RecordProcessStart 和 ModuleProfiler::RecordProcessEnd 调用时, 会记录相关处理过程的追踪事件。详情可查看[性能统计](#)。
- 仅做数据流追踪, 不做性能分析。记录事件还可以使用 PipelineTracer::RecordEvent 函数进行。调用该函数需要传入一个名为 event 的参数, 类型为 TraceEvent。

TraceEvent 需要填入以下信息:

表 8.2: 数据流追踪字段说明

字段名称	描述
key	key 为一个数据的唯一标识，一般可通过 <code>std::make_pair(CNFrameInfo::stream_id, CNFrameInfo::timestamp)</code> 的方式构造。
module_name	事件发生的模块名。
process_name	事件名称。当追踪事件是伴随着性能统计发生的，那么 <code>process_name</code> 为注册在性能统计功能中的某一个处理过程的名称。
time	发生事件的事件点，一般可调用 <code>cnstream::Clock::now()</code> 获得。
level	可选值为 <code>cnstream::TraceEvent::Level::PIPELINE</code> 及 <code>cnstream::TraceEvent::Level::MODULE</code> 。分别表示 pipeline 端到端的事件和模块级别的事件。 该值决定了获取到的事件数据中这个事件的存放位置。详情请查看 <a href="#">获取追踪数据</a> 。
type	可选值为 <code>cnstream::TraceEvent::Type::START</code> 及 <code>cnstream::TraceEvent::Type::END</code> 。 若追踪事件是伴随着性能统计发生的，那么当此事件是某个处理过程的开始，则 <code>type</code> 为 <code>cnstream::TraceEvent::Type::START</code> ，否则为 <code>cnstream::TraceEvent::Type::END</code> 。

### 8.3.3 追踪数据占用的内存空间

追踪记录的数据存储在内存中，通过循环数组进行存储。循环数组的大小可在构建 pipeline 时设定。

通过 `Pipeline::BuildPipeline` 传入的 `profiler_config` 参数组中的 `trace_event_capacity` 参数指定最大存储的追踪事件个数。每个事件占用的内存大小是固定的，该参数可间接调整追踪功能的最大内存占用大小。

`trace_event_capacity` 默认值为 100000，即最大存储 100000 条记录。

### 8.3.4 获取追踪数据

PipelineTracer 提供 GetTrace 函数供获取追踪功能记录的事件。

GetTrace 函数可以获取某一段时间的事件信息。

该函数返回内存中指定时间段的事件，使用 PipelineTrace 结构存储。

PipelineTrace 结构中的两个字段说明：

#### 8.3.4.1 process\_traces

该字段存储 level 为 `cnstream::TraceEvent::Level::PIPELINE` 的事件。按 `process_name` 归类，分别存储为 `ProcessTrace` 结构。

#### 8.3.4.2 module\_traces

该字段存储 level 为 `cnstream::TraceEvent::Level::MODULE` 的事件。按 `module_name` 归类，分别存储为 `ModuleTrace` 结构。

`ModuleTrace` 中又按 `process_name` 归类，分别存储为 `ProcessTrace` 结构。

#### 注意：

CNStream 不保证追踪数据的完整性，即有可能丢失追踪数据。保证追踪数据的完整性需要配合调整 `trace_event_capacity` 参数和获取追踪数据的方式。

CNStream 中使用循环数组来存储事件，即随着事件的不断发生，新的事件将覆盖老的事件。任一时刻，能获取到最久远的事件即为最新发生的事件之前的第 `trace_event_capacity` 个事件，再之前的事件则被丢弃。

故想要获取完整的事件信息，应该保证事件获取速度大于事件被覆盖的速度。

### 8.3.5 追踪数据的处理

CNStream 提供简单的追踪事件的获取方式，参考[获取追踪数据](#)。追踪数据的处理属于一个开放性命题，用户获取到追踪数据后，可以以任何方式进行处理。CNStream 目前提供一个简单的追踪数据表现方式，参见[追踪数据可视化](#)。

### 8.3.6 追踪数据可视化

CNStream 中的追踪数据可视化依赖 Google Chrome 的 chrome tracing 功能实现。**TraceSerializeHelper** 类提供方法把追踪功能记录的数据存储为 chrome tracing 需要的 json 格式。

使用 **TraceSerializeHelper** 存储的追踪数据文件可以使用 chrome tracing 可视化。使用方法可参考 CNStream 源代码 `samples/cns_launcher/cns_launcher.cpp` 中的实现。

执行下面步骤追踪数据可视化:

1. 在配置文件中打开追踪功能。详情参考[打开数据流追踪功能](#)。
2. 指定 `trace_data_dir` 参数。为可执行文件 `samples/bin/cns_launcher` 指定参数 `trace_data_dir` 为追踪数据存放目录。其中, `samples/bin` 是在编译 CNStream 后自动生成的文件夹。
3. 生成可视化 JSON 文件。在运行完程序后, 在 `trace_data_dir` 参数指定的目录中即会生成 `cnstream_trace_data.json` 文件。
4. 可视化追踪的数据。在 chrome 浏览器地址栏输入 `chrome://tracing`, 把 `cnstream_trace_data.json` 拖入浏览器即可查看追踪数据的可视化结果。

可视化内容为每一帧数据在经过每一个处理过程的时间点和经历的时间长度。

可视化示例:

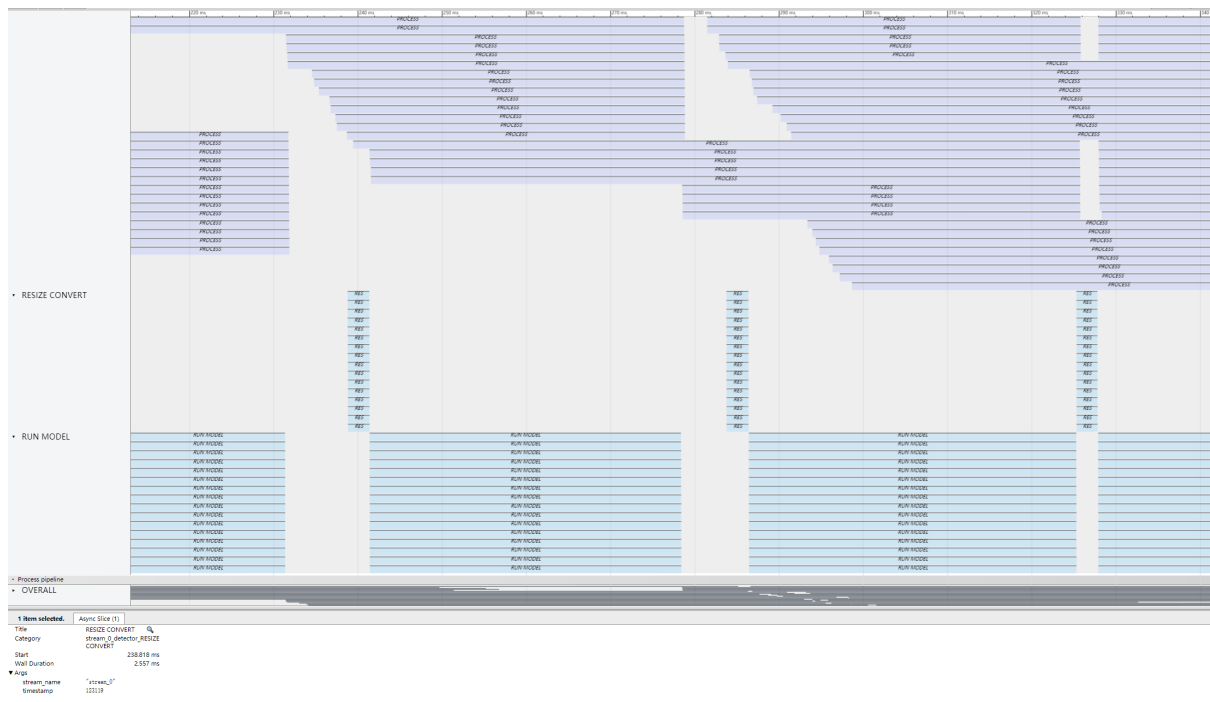


图 8.1: 追踪数据可视化

## 9.1 file\_list 文件是做什么用的？

文本文件 `file_list` 用于存储视频或图片的路径。文件中，每一行代表一路视频或者图片的 URL，可以是本地视频文件路径、RTSP 或 RTMP 地址等。执行 `run.sh` 脚本时，`file_list` 文件会被调用并传入应用程序。当首次执行 `run.sh` 脚本时，该脚本会自动生成 `files.list_image` 和 `files.list_video` 文件。`files.list_image` 文件用于存放一组 JPEG 图片路径。`files.list_video` 文件用于存放两路视频路径。

用户也可自己创建一个文件来存放存储视频或图片的路径。但是需要将文件名设置为 `run.sh` 脚本中 `data_path` 参数的值。该脚本存放于 `samples/cns_launcher/` 的各个子目录下。

常见几种 `file_list` 内容格式如下：

- file list 中存放本地视频文件，内容如下：

```
/path/of/videos/1.mp4
/path/of/videos/2.mp4
...
```

- file list 中存放 RTSP 视频流地址，内容如下：

```
rtsp://ip:port/1
rtsp://ip:port/2
...
```

- file list 中存放图片，每一行为一组 JPG 图片路径，通配符遵循 FFMpeg AVformat 匹配规则，如 `%02d.jpg`，`%.jpg` 等。内容如下：

```
/path/of/%d.jpg
/path/of/%d.jpg
...
```

## 9.2 怎么输入任意命名的图片？

目前 CNStream 支持 Jpeg 图片解码，可以通过 file\_list 中添加字段进行通配符匹配，这种使用方式对图片源输入来说并不灵活。CNStream 同时还支持输入任意名字的图片，通过 fopen 或者 cv::imread 事先将图片读入内存，然后再把该内存数据喂入 Pipeline 中进行后续处理。具体细节可以参考 samples/cns\_launcher/cns\_launcher.cpp 文件中的函数 FeedFrameAsync 和 FeedJpegAsync 内容。

输入任意名字图片时，file list 中存放图片目录，内容如下：

```
/path/of/images/  
...
```

/path/of/images/ 目录下保存待输入的图片。

## 9.3 有没有交叉编译 CNStream 的指导？

可下载 edge 编译压缩包 <http://video.cambricon.com/models/edge.tar.gz>，解压后按照 README 文档提供的步骤进行编译。

## 9.4 parallelism 参数该怎么配置？

parallelism 是模块内并行度，表明有多少个线程在同时运行 Module::Process 函数。总体上该值越大，并行度越高，流水线处理能力越强，占用资源也越多。

- parallelism 值应不大于数据流路数，否则会有线程空挂，造成资源浪费。同时增大该值时需要时刻关注系统资源是否够用。
- 对于数据源模块，parallelism 设置为 0 即可，因为数据源模块的并行度是由输入数据路数决定的。
- 对于 Inference 和 Tracker 模块，其内部使用 InferServer 进行异步推理，Process 函数主要实现将数据发送给 InferServer 的逻辑，建议 parallelism 设置为 1。
- 其余插件可以根据具体性能进行调整，比如某个插件性能较低，那么可以增大其 parallelism 值提高处理速率。如果当前性能已经远远超出 pipeline 上其他插件，那么可以减小该值以减少资源占用。

## 9.5 使用 module\_contrib 目录下的模块出现类未定义错误该怎么解决？

该问题是由于程序没有正确链接 libcnstream\_contrib.so 库导致的，module\_contrib 目录下的代码包含在 libcnstream\_contrib.so 库中。CNStream 初始化各模块使用的反射技术在程序运行时候才会依据类名字符串实例化一个类，这样的话不管编译时是否显式指定链接 libcnstream\_contrib.so，实际都不

会链接该库。我们可以通过 `ldd` 程序名称命令查看是否真正完成链接。为了保证找到类定义，可以给链接选项加上 `--no-as-needed` 参数，CMake 为例：

```
set(CMAKE_EXE_LINKER_FLAGS "-Wl,--no-as-needed")
```

## 9.6 怎么调整 Log 打印等级?

CNStream 采用 Glog 日志系统，所以日志默认行为和使用方式与 Glog 完全一致。例如可以通过环境变量 `GLOG_minloglevel` 调整日志输出等级，范围 [0-3]，数字越小输出的 log 内容越多。如果希望打印更多的日志信息，在 `GLOG_minloglevel=0` 的前提下，设置环境变量 `GLOG_v` 调整自定义日志等级，范围 [1-2]，数字越大输出的 log 内容越多。





## 10 Release Notes

本章介绍了 CNStream 各版本的新增功能、功能变更、废用功能、已修复问题以及已知问题。

### 10.1 CNStream 2022-10-17 (Version 7.0.0)

#### 10.1.1 新增功能及功能变更

- 支持云端 MLU370 平台。
- 支持边缘端 CE3226 平台。
- DataSource 模块 - 支持 Sensor (Camera) 形式的输入源 (仅边缘端支持)。- 支持将解码后的数据存放在内存池中, 避免内存的频繁申请释放。支持通过参数设置内存池个数。- RawImgMemHandler 更名为 ImageFrameHandler。- 创建 SourceHandler 方式变更。详情请查看[数据源模块](#)。
- 移除原 Inference 模块, Inferencer2 模块更名为 Inferencer。详情请查看[推理模块](#)。
- 自定义前后处理部分有较大改动。详情请查看[推理模块](#)。
- Encode 和 RtspSink 模块合并为 VEncode 模块。支持编码后同时保存文件到本地和 RTSP 推流。
- Osd 模块支持硬件加速 (仅边缘端支持)。
- 以下功能暂不支持, 将在后续版本中支持: - 不支持 Python API 接口。- 不支持 Web 可视化功能。- 未提供 simple pipeline 示例。

#### 10.1.2 废用功能

- 不再兼容 MLU200 系列 (MLU270/MLU220.M.2/MLU220.Edge/MLU220.SOM) 平台。
- DataSource 模块 - 不再支持 CPU 编码。- 不再支持复用 codec buffer 内存。- 不再支持设置输入输出 buffer 数目。- 不再支持输出内存类型的选择。
- 移除原 Inference 模块。Inferencer2 模块更名为 Inferencer。
- 不再提供内置前处理方法。
- 不再支持 cns\_launcher 中的 vehicle\_cts 示例。

### 10.1.3 版本兼容

- 兼容寒武纪 MLU370/CE3226 平台。
- 寒武纪 MLU370 平台兼容 CNToolkit 3.0.2 版本/CNCV 1.0.0 版本/MagicMind 0.13.0 版本。
- 寒武纪 CE3226 平台兼容 ce3226v100-sdk 1.0.0 版本。

### 10.1.4 版本限制

- 不再支持 MLU200 系列平台。
- Magicmind 运行时库引入对 Cambricon CNNL、CNL\_extra、CNLight 的依赖，版本要求见 MagicMind 用户手册。

## 10.2 CNStream 2022-8-26 (Version 6.3.0)

### 10.2.1 新增功能及功能变更

- 支持 MLU370 YOLOv5。
- 修复某场景下 RTSP client play 失败的问题。

### 10.2.2 版本兼容

- 兼容寒武纪 MLU270、MLU220 的 CNToolkit 1.7/1.8 版本。
- 兼容寒武纪 MLU370 的 CNToolkit 2.6 版本。

### 10.2.3 版本限制

- inference 插件仅支持 MLU200 系列推理，inference2 插件兼容 MLU200/MLU300 系列平台推理。
- 使用 Magicmind 后端，则要求其版本高于 0.5.0，Magicmind 运行时库引入对 Cambricon CNNL、CNL\_extra、CNLight 的依赖，版本要求见 Magicmind 用户手册。

## 10.3 CNStream 2022-1-6 (Version 6.2.0)

### 10.3.1 新增功能及功能变更

- 新增 Python API，详情请查看 python。
- 支持 MLU370 硬件编码。

### 10.3.2 废用功能

- 公开版本不再提供包含 CNToolkit 内容安装的 Dockerfile。

### 10.3.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 的 CNToolkit 1.7/1.8 版本。
- 兼容寒武纪 MLU370 的 CNToolkit 2.6 版本。

### 10.3.4 版本限制

- inference 插件仅支持 MLU200 系列推理，inference2 插件兼容 MLU200/MLU300 系列平台推理。
- 使用 Magicmind 后端，则要求其版本高于 0.5.0，Magicmind 运行时库引入对 Cambricon CNNL、CNL\_extra、CNLight 的依赖，版本要求见 Magicmind 用户手册。

## 10.4 CNStream 2021-10-11 (Version 6.1.0)

### 10.4.1 新增功能及功能变更

- 新增车牌检查识别示例。
- 调整 encode 插件部分配置参数，详情请查看[VEncode 模块](#)。
- 调整 rstp\_sink 插件部分配置参数，详情请查看 [rstp\\_sink](#)。

### 10.4.2 废用功能

- 移除 ModuleIPC 内置模块。
- cnstream\_inspect 工具移除配置文件检查功能。
- 删除 multi\_sources 示例。
- 废弃 readthedocs.io 网站发布文档，改由仓库自带 PDF 发布。

### 10.4.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 的 CNToolkit 1.7/1.8 版本。

#### 10.4.4 版本限制

- inference 插件仅支持 MLU200 系列推理，inference2 插件兼容 MLU200/MLU300 系列平台推理。

### 10.5 CNStream 2021-8-10 (Version 6.0.0)

#### 10.5.1 新增功能及功能变更

- 新增支持图嵌套结构，简化业务流程配置, 详情请参考[创建应用程序](#)。
- 新增 MLU300 系列硬件支持，包括 H264/JPEG 解码、推理等功能。
- 新增 vehicle\_cts/body\_pose/simple\_run\_pipeline 演示业务，并调整其余示例应用的目录结构。
- 新增 `cnstream::Collection` 类用于存储可变类型的结构化数据，`CNInferObject::collection` 和 `CNFrameInfo::collection` 分别替换 `CNInferObject::datas` 和 `CNFrameInfo::datas`。
- 支持 CNCV（Cambricon Computer Vision Library）算子进行预处理。
- 基于 librdkafka 新增 modules\_contrib/kafka 试验性插件，用于生产和消费消息数据, 详情请查看[Kafka 模块](#)。

#### 10.5.2 废用功能

- 废弃交叉编译卡型控制选项 MLU220\_SOC/MLU220EDGE，改由运行时动态判断硬件。
- 废弃 CMake 编译选项 RELEASE，改由 CMake 自带选项 CMAKE\_BUILD\_TYPE 控制。
- 废弃并发深度控制功能，即删除 `CNFrameInfo::flow_depth_` 等相关代码。
- 删除 `CNDataFrame::ptr_mlu/CNDataFrame::ptr_cpu`，建议通过 `CNSyncedMemory` 访问数据。
- 删除 samples/example 目录，目录中示例代码由 samples/simple\_run\_pipeline 替换。

#### 10.5.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 的 CNToolkit 1.6/1.7/1.8 版本。

#### 10.5.4 版本限制

- inference 插件仅支持 MLU200 系列推理，inference2 插件兼容 MLU200/MLU300 系列平台推理。

## 10.6 CNStream 2021-1-28 (Version 5.3.0)

### 10.6.1 新增功能及功能变更

- 支持快速移除某一路数据流。
- 支持分辨率动态变化的数据流。
- 单 Pipeline 支持多个输入源插件。
- 新增内置 inference2 插件，基于 Infer Server 实现推理功能。
- 优化内置输入源插件。
- 新增 Profiler 和 Tracing 性能统计功能。
- 修复 MLU220 M.2 和 MLU220 Edge 端部分缺陷。

### 10.6.2 废用功能

- 仓库中不再包含 EasyDK 源码，通过子仓形式依赖 EasyDK。
- 性能统计不再依赖 sqlite。

### 10.6.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 和 EDGE 平台。

### 10.6.4 版本限制

- 基于寒武纪 CNToolkit 1.5.0 版本使用 CNStream。

## 10.7 CNStream 2020-9-18 (Version 5.2.0)

### 10.7.1 新增功能及功能变更

- cnencode 移入内置模块，并支持 JPEG 和 H264 的 CPU 和 MLU 编码。
- 新增 web 可视化工具。
- sqlite 改用静态链接。

### 10.7.2 废用功能

- 由于存在某些特殊情况无法满足线程安全条件，`threadsafe_vector` 和 `threadsafe_map` 结构体改为使用 `vector` 和 `map`。用户需要在结构体外部去保证线程安全。

### 10.7.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 和 EDGE 平台。

### 10.7.4 版本限制

- 基于寒武纪基础软件平台 1.4.0 或者 1.5.0 版本使用 CNStream。

## 10.8 CNStream 2020-07-10 (Version 5.0.0)

### 10.8.1 新增功能及功能变更

- `cnstream-toolkit.so` 重命名为 `easydk.so`。
- 新增 `IModuleObserver`，支持应用程序获取每一个 Module 的输出。详情查看[cnstream::Module](#) 类。
- 在 `CNFrameInfo` 类中新增 `ThreadSafeUnorderedMap<int, cnstream::any> datas`，支持用户自定义任意类型数据。详情查看[cnstream::CNFrameInfo](#) 类。
- 新增 Classification、detection (YOLO v3)、track、secondary、rtsp、multi\_process 等示例程序。
- 性能统计功能变更。详情查看[性能统计](#)。

### 10.8.2 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

### 10.8.3 版本限制

- 基于寒武纪基础软件平台 1.3.0 或者 1.4.0 版本。

## 10.9 CNStream 2020-05-28 (Version 4.5.0)

### 10.9.1 新增功能及功能变更

- 支持多进程和单进程使用多个 MLU 卡。
- 新增 rtsp\_sink 模块。详情查看 [rtsp\\_sink](#)。
- 性能统计功能变更，修改相关接口介绍。详情查看[性能统计](#)。
- 支持 1.3.0 版本的寒武纪基础软件平台包。
- 部分算子更新。
- 修复一些已知问题。

### 10.9.2 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

### 10.9.3 版本限制

- 基于寒武纪基础软件平台 1.3.0 版本。

## 10.10 CNStream 2020-04-16 (Version 4.4.0)

### 10.10.1 新增功能及功能变更

- 支持性能统计功能，帮助用户统计各模块及整条 pipeline 的性能。详情查看[性能统计](#)。
- 支持多线程机制。
- 新增 Live555、SDL22.0.4+ 以及 SQLite3 环境依赖。
- 新增 CentOS 和 Ubuntu18.04 Dockerfile。
- 支持 1.2.5 版本的寒武纪基础软件平台。
- 修复汇聚插件随机性卡死、多线程并行推理异常等问题。

### 10.10.2 废用功能

下面功能已废弃：

- 废弃 fps\_stats 插件。
- 删除之前用于参考的 Apps 目录。

### 10.10.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

### 10.10.4 版本限制

- 基于寒武纪基础软件平台 1.2.5 版本。

## 10.11 CNStream 2019-02-20

### 10.11.1 新增功能及功能变更

- SyncedMemory 支持线程安全。
- 支持寒武纪 MLU220 M.2 平台。
- 修复部分缺陷。

### 10.11.2 版本限制

- 依赖寒武纪基础软件平台 1.2.4 运行。

## 10.12 CNStream 2019-12-31

### 10.12.1 新增功能及功能变更

- 新增 CNStream Inspect 工具。
- 不再依赖 toolkit 二进制文件。
- 优化 YoloV3 Demo 性能。