寒武纪 CNStream 用户手册

版本 *6.0.0*

2021 年 8 月 30 号

Cambricon 寒武纪

目录

插图目录

表格目录

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。寒武纪不应对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，以期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。其他公司和产品名称应为与其关联的各自公司的商标。

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
- © 2021 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。
- 编程指南详细介绍，参考编程指南。
- 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、CentOS 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 2.4.9+
- FFmpeg 2.8 3.4 4.2
- SDL2 2.0.4+
- GFlags
- GLog
- Librdkafka

2.1.2 寒武纪安装包

CNStream 还依赖于寒武纪 CNToolkit 安装包中 CNRT 库和 CNCodec 库。如果使用 MLU300 板卡，CNStream 则还需依赖 MagicMind、CNCV、CNNL、CNNLExtra 和 CNLight。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，联系寒武纪工程师获得相关的软件包和安装指南。

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 libsdl2-dev curl libcurl4-openssl-dev
```

2.1.4 CentOS 环境下安装和配置

执行下面步骤，在 CentOS 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 yum install opencv-devel.x86_64 gflags.x86_64 glog.x86_64 cmake3.x86_64
sudo yum install freetype-devel SDL2_gfx-devel.x86_64 wqy-zenhei-fonts
sudo yum install ffmpeg ffmpeg-devel curl libcurl-devel librdkafka-devel
```


2.1.5 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/Dockerfiler.16.04
- ${CNSTREAM_DIR}/docker/Dockerfiler.18.04
- ${CNSTREAM_DIR}/docker/Dockerfiler.CentOS

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。用户可以选择以下其中一种方式编译镜像。
   - 如果选择将寒武纪 CNToolkit 包安装进镜像中：
     1. 运行下面命令，拷贝寒武纪 CNToolkit 安装包到 CNStream 源码目录下。

     ```
     cp ${toolkit_package} CNStream
     ```

     2. 运行下面命令将寒武纪 CNToolkit 安装包安装到镜像中，其中 ${cntoolkit_package_name} 为寒武纪 CNToolkit 安装包及其存放路径。

     ```
     docker build -f Dockerfile.18.04 --build-arg toolkit_package=${cntoolkit_package_name}
     ↪ -t ubuntu1804_cnstream:v1 .
     ```

   - 如果选择不将寒武纪 CNToolkit 包安装进镜像中，运行下面命令编译镜像：

   ```
   docker build -f Dockerfile.18.04 -t ubuntu1804_cnstream:v1 .
   ```

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 --net=host --ipc=host --pid=host -it --name container_name  -v $PWD:/
↪workspace ubuntu1804_cnstream:v1
```

5. 如果之前制作的镜像没有安装寒武纪 CNToolkit 安装包，安装 CNToolkit 安装包。详情查看寒武纪安装包 。

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。

```
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
build_display	ON / OFF	ON	Build display module
build_ipc	ON / OFF	ON	Build ipc module
build_encode	ON / OFF	ON	Build encode module
build_inference	ON / OFF	ON	Build inference module
build_inference2	ON / OFF	ON	Build inference2 module
build_osd	ON / OFF	ON	Build osd module
build_rtsp_sink	ON / OFF	ON	Build rtsp_sink module
build_source	ON / OFF	ON	Build source module
build_track	ON / OFF	ON	Build track module
build_modules_contrib	ON / OFF	ON	Build contributed modules
build_tests	ON / OFF	ON	Build tests
build_samples	ON / OFF	ON	Build samples

下页继续

表 2.1 – 续上页

WITH_FFMPEG	ON / OFF	ON	Build with FFmpeg
WITH_FREETYPE	ON / OFF	OFF	Build with FREETYPE.
WITH_RTSP	ON / OFF	ON	Build with RTSP
WITH_FFMPEG_AVDEVICE	ON / OFF	OFF	Build with FFmpeg avdevice
CNIS_WITH_CNCV	ON / OFF	ON	Build with CNCV
CNIS_USE_MAGICMIND	ON / OFF	OFF	Use MagicMind as inference backend

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 MLU220-SOM 为例：

```
export NEUWARE_HOME=/your/path/to/neuware
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 -
↪Dbuild_display=OFF
```

更多信息可以参考交叉编译 。

7. 目前 CNStream 中有两个推理模块，Inferencer 和 Infernecer2，其中 Inferencer 使用 CNRT 后端，仅支持 MLU200 系列；Inferencer2 支持 CNRT 后端和 MagicMind 后端，MagicMind 后端支持 MLU200 系列平台和 MLU300 系列平台。Inferencer2 支持使用 CNCV（寒武纪计算机视觉库，详情参考《寒武纪计算机视觉库用户手册》）进行神经网络预处理。通过编译选项 CNIS_USE_MAGICMIND 使能 MagicMind 后端，CNIS_WITH_CNCV 使能 CNCV 预处理：


```
cmake {CNSTREAM_DIR} -DCNIS_USE_MAGICMIND=ON -DCNIS_WITH_CNCV=ON
```

注解：

目前已知 Ubuntu16.04 使能 MagmicMind 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配置模块的参数。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 MLU200 或者 MLU300 系列平台上使用。

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：
  - MLU200 `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_classification.cpp`
  - MLU300 `${CNSTREAM_DIR}/samples/common/postprocess/video_postprocess_classification.cpp`

使用模块

- DataSource

- Inferencer/Inferencer2
- Osd
- Encode

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：
  - MLU200 `${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov3.cpp`
  - MLU300 `${CNSTREAM_DIR}/samples/common/postprocess/video_postprocess_yolov3_mm.cpp`

使用模块

- DataSource
- Inferencer/Inferencer2
- Osd
- Encode/Displayer/RtspSink

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：`${CNSTREAM_DIR}/samples/common/postprocess/postprocess_yolov3.cpp`

使用模块

- DataSource
- Inferencer
- Tracker
- Osd
- Encode/Displayer/RtspSink

2.3.1.4 车辆结构化样例

样例文件

- 配置文件：${CNSTREAM_DIR}/samples/cns_launcher/vehicle_cts/config_template.json
- 运行脚本：${CNSTREAM_DIR}/samples/cns_launcher/vehicle_cts/run.sh
- 车辆筛选车的策略源码：${CNSTREAM_DIR}/samples/common/obj_filter/car_filter.cpp
- 后处理源码：
  - ${CNSTREAM_DIR}/samples/common/postprocess/postprocess_ssd.cpp
  - ${CNSTREAM_DIR}/samples/common/postprocess/postprocess_vehicle_cts.cpp

使用模块

- DataSource
- Inferencer
- Osd
- Encode/Displayer/RtspSink

2.3.1.5 车辆结构化样例 (MLU300)

样例文件

- 配置文件：${CNSTREAM_DIR}/samples/cns_launcher/configs/vehicle_recognization_mlu370.json
- 运行脚本：${CNSTREAM_DIR}/samples/cns_launcher/vehicle_recognization/run_370.sh
- 后处理源码：
  - ${CNSTREAM_DIR}/samples/common/postprocess/video_postprocess_classification.cpp
  - ${CNSTREAM_DIR}/samples/common/postprocess/video_postprocess_yolov3_mm.cpp

使用模块

- DataSource
- Inferencer2
- Osd
- Encode

2.3.1.6 姿态检测示例

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
- Encode/Displayer/RtspSink

Cambricon 寒武纪

3 概述

3.1 简介

CNStream 是面向寒武纪开发平台的数据流处理 SDK。用户可以根据 CNStream 提供的接口，开发实现自己的组件。还可以通过组件之间的互连，灵活地实现自己的业务需求。CNStream 能够大大简化寒武纪深度学习平台提供的推理和其他处理，如视频解码、神经网络图像前处理的集成。也能够在兼顾灵活性的同时，充分发挥寒武纪 MLU（Machine Learning Unit 机器学习处理器）的硬件解码和机器学习算法的运算性能。

CNStream 基于模块化和流水线的思想，提供了一套基于 C++ 语言的接口来支持流处理多路并发的 Pipeline 框架。为用户提供可自定义的模块机制以及通过高度抽象的 CNFrameInfo 类型进行模块间的数据传输，满足用户对性能和可伸缩性的需求。

CNStream 支持在 MLU200 和 MLU300 系列平台上使用。

3.2 CNStream 特点

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。框架具有多个基于寒武纪思元处理器的硬件加速模块，可将深层神经网络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深度学习网络和 IP（Intellectual Property），并用寒武纪特定模型转换器生成寒武纪平台能执行的模型，无需再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。每一个模块的并行度及队列深度可以根据时延要求及数据吞吐量而定制。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，根据寒武纪神经网络推理芯片设计特点，内置了从数据源解码、前后处理及推理、追踪等模块。其中推理模块、编解码模块和追踪模块充分利用了寒武纪芯片设计特点和内部 IP 核心间 DMA 的能力, 使用后极大的提高了系统整体吞吐效率。
- 支持分布式架构，内置 Kafka 消息模块，帮助客户快速接入分布式系统。

- 支持常见分类以及常规目标检测神经网络，比如：YOLO、SSD、ResNet、VGG 等。
- 图形化界面方式生成 pipeline。整个框架支持图形化拖拽方式生成 pipeline，帮助用户快速体验寒武纪数据流分析框架。
- 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，在云边端侧集成。

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。
- 由于深度学习框架强调框架算子的完整性及快速验证深度学习模型的能力，在生产系统中无法利用硬件性能达到最高效率。而 CNStream 内置 Inference 模块直接基于寒武纪运行时库（CNRT）设计，内置 Inference2 模块新增针对寒武纪 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 应用框架

使用 CNStream SDK 开发一个应用的典型框架如下图：

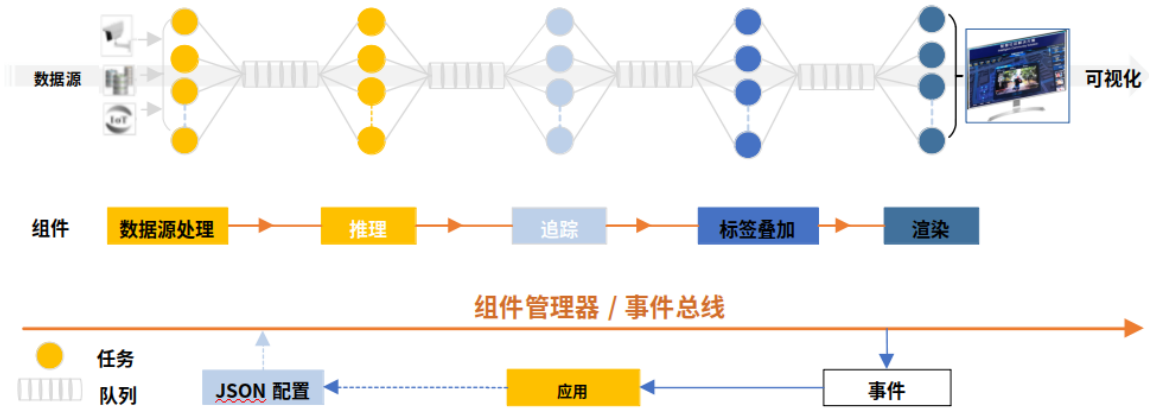

图 3.1: 典型视频结构化场景架构图

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。

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

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。
      "output_type" : "mlu",               //解码图像输出到 MLU 内存。
      "decoder_type" : "mlu",              //使用 MLU 解码。
      "device_id" : 0                      //MLU 设备 id。
    }
  },

  "inference" : {
    "class_name" : "cnstream::Inferencer",
    "parallelism" : 16,            //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/resnet34_ssd.cambricon",
      "func_name" : "subnet0",
      "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。详细代码编写结构，可参考 `samples/simple_run_pipeline/simple_run_pipeline.cpp`。

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   -- 表示已经处理完毕，传递数据操作由框架完成。
  //  > 0 -- 表示已经接收数据，在后台进行后续处理。传递数据操作由 module 自身完成。
  //  < 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神经网络推理结果：


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

CNDataFrame 中集成了 SyncedMemory。基于 MLU 平台的异构性，在应用程序中，当某个具体的 module 处理的数据可能需要在 CPU 上或者 MLU 上时，SyncedMem 实现了 CPU 和 MLU（Host 和 Device）之间的数据同步。通过 SyncedMem，module 可以自身决定访问保存在 MLU 或者 CPU 上的数据，从而简化 module 的编写，接口如下：

```
std::unique_ptr<CNSyncedMemory> data[CN_MAX_PLANES];
```

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 pipepline 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::unordered_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。

```
ValueT& Add(const std::string& tag, const ValueT& value);
ValueT& Get(const std::string& tag);
```


Cambricon
寒　武　纪

5 内置模块

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（MLU 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》。
- 神经网络推理模块：依赖于寒武纪运行时库（CNRT），支持多种神经网络离线模型对图像数据进行神经网络推理。CNStream 的模块式设计，为用户提供了在视频流解码和推理之后的进一步数据加工和处理。
- 神经网络推理模块 2：基于 CNStream 子仓库 Cambricon EasyDK（Cambricon Easy Development Kit）提供的寒武纪推理服务（CNIS），支持多种神经网络离线模型对图像数据进行神经网络推理。
- 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、多媒体显示模块和 RTSP 推流模块等。

- OSD（On-Screen Display）模块：支持内容叠加和高亮物件处理。
- 编码模块：同时支持 CPU/MLU 编码。
- 多媒体显示模块：支持屏幕上显示视频。
- RTSP 推流模块：将图像数据编码后推流至互联网。

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。不仅支持获取内存数据裸流，还可以通过 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 CNSyncedMemory 中。目前支持 H264、H265、MP4、JPEG、RTSP 等协议。

注意：

- 一个 pipeline 支持定义多个数据源模块, 不同的模块分别处理不同设备上的数据，可参考示例程序 multi_sources。
- MLU220 硬件平台上使用需要尤其注意 apply_stride_align_for_scaler 参数的使用。详情查看apply_stride_align_for_scaler。

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 pipeline 的 `ProvideData()` 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);
  int RemoveSource(const std::string &stream_id);

  // 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

在 decode_config.json 配置文件中进行配置，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。
    "output_type" : "mlu",                //  (可选参数) 输出类型。可设为 MLU 或 CPU。
    "decoder_type" : "mlu",               //  (可选参数) decoder 类型。可设为 MLU 或 CPU。
    "reuse_cndec_buf" : "false"           //  (可选参数) 是否复用 Codec 的 buffer。
    "device_id" : 0                       //  当使用 MLU 时为必设参数。设备 id，用于标识多卡机器的
设备唯一编号。
  }
}
```

5.2.1.1 decoder_type

字符串类型。指定解码器使用的硬件设备，可选值有 cpu、mlu。默认为 cpu。

- 当 decoder_type 值设置为 cpu 时表示使用 CPU 进行解码，CPU 解码器输出数据在主机内存上。
- 当 decoder_type 值设置为 mlu 时表示使用 MLU 进行解码，MLU 解码器输出数据在 MLU 设备内存上。

5.2.1.2 output_type

字符串类型。指定插件输出数据内存类型，可选值有 cpu、mlu。默认为 cpu。

- 当 output_type 值设置为 cpu 时表示插件将解码后的数据搬运到主机内存上。此时分为两种情况：
  - decoder_type 为 cpu，此时解码后原始数据在主机内存上，不进行数据搬运。
  - decoder_type 为 mlu，此时解码后原始数据在 MLU 设备内存上。在 MLU270 和 MLU220 M.2 平台将使用 PCIE 进行主机侧到设备侧的内存拷贝。在 MLU220 EDGE 平台将使用 CPU 进行内存拷贝。
- 当 output_type 值设置为 mlu 时表示插件将解码后的数据搬运到 MLU 设备内存上。此时分三种情况：
  - decoder_type 为 cpu，此时解码后原始数据在主机内存上。在 MLU270 和 MLU220 M.2 平台将使用 PCIE 进行主机测到设备侧的内存拷贝。在 MLU220 EDGE 平台将使用 CPU 进行内存拷贝。
  - decoder_type 为 mlu，且 reuse_cndec_buf 为 false。此时解码后原始数据在 MLU 设备内存上。将使用 MLU 计算核心（IPU）进行内存拷贝。
  - decoder_type 为 mlu，且 reuse_cndec_buf 为 true。此时解码后原始数据在 MLU 设备内存上，直接将解码后的数据指针向后传递使用，不进行内存拷贝。

5.2.1.3 reuse_cndec_buf

设置是否开启解码器内存复用功能。可选值有 true、false。默认为 false。

解码器内存复用即解码后内存直接向后传递使用，不进行内存拷贝。需要配合 decoder_type 和 output_type 参数使用。只有当两个参数都设置为 mlu 且该参数设置为 true 时，解码器内存复用功能生效。

注意：

使用解码器内存复用功能时需要注意 output_buf_number、input_buf_number 两个参数的设置。它们将影响 pipeline 性能、和 MLU 设备侧内存占用。

5.2.1.4 input_buf_number

设置 MLU 解码器输入队列中最大缓存帧数。当 decoder_type 为 mlu 时生效。该值越大占用 MLU 设备侧内存越大。推荐值为 4。

5.2.1.5 output_buf_number

设置 MLU 解码器输出队列中最大缓存帧数。当对视频流解码时该值最大为 32。当对 JPEG 进行解码时最大为 16。

该值越大占用 MLU 设备内存越大。pipeline 中数据流数越多，码流分辨率越大，图片分辨率越大，占用 MLU 设备内存越多。受限于内存大小，则 output_buf_number 上限值越小。

- 当复用解码器内存功能打开时，基于性能考虑，推荐把该值尽可能的设置大。
- 当复用解码器内存关闭时，该值设置为大于码流参考帧数量一般就不会影响性能。

若设置的值过大，会导致创建解码器失败。

5.2.1.6 interval

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。

5.2.1.7 apply_stride_align_for_scaler

指定解码后输出按 scaler 硬件的要求进行对齐。在使用 MLU 解码时，输出的 NV12/NV21 数据将按照 128 像素对齐，即解码后的 yuv 数据 stride 为 128 的倍数。

可选值为 true、false，默认值为 false。MLU220 硬件平台考虑使用该参数，其它硬件平台不推荐使用该参数为 true。

5.2.1.8 device_id

设置使用的设备 id，决定 MLU 解码使用的设备及解码后数据存放在哪张 MLU 卡上。

5.2.2 变分辨率解码支持

支持对带有多个不同分辨率码流进行解码。

5.2.2.1 使用限制

仅支持 MLU370 进行解码时处理变分辨率码流。

5.2.2.2 使用方式

在创建 **cnstream::FileHandler**、**cnstream::ESMemHandler**、**cnstream::RtspHandler** 处理视频流时，可传入类型为 **MaximumVideoResolution** 的参数来打开变分辨率解码功能并设置支持的最大分辨率。

cnstream::FileHandler、**cnstream::ESMemHandler**、**cnstream::RtspHandler** 定义在 `modules/source/include/data_source.hpp` 文件中定义。

5.2.2.3 异常处理

当处理的码流中分辨率大于设置的最大分辨时，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 神经网络推理模块

神经网络推理（Inferencer）模块是基于寒武纪运行时库（CNRT）的基础上，加入多线程并行处理及适应网络特定前后处理模块的总称。用户根据业务需求，只需载入客制化的模型，即可调用底层的推理。根据业务特点，该模块支持多 batch 及单 batch 推理，具体可参阅代码实现。

5.3.1 使用说明及参数详解

在 yolov3_object_detection_mlu270.json 配置文件中进行配置，如下所示。该文件位于 samples/cns_launcher/configs/ 目录下。

```
"detector" : {
  "class_name" : "cnstream::Inferencer",    // （必设参数）推理类名。
  "parallelism" : 2,                        // （必设参数）并行度。
  "max_input_queue_size" : 20,              // （必设参数）最大队列深度。
  "next_modules" : ["tracker"],             // （必设参数）下一个连接模块的名称。
  "custom_params" : {                       // 特有参数 。
    // （必设参数）模型路径。本例中的路径使用了代码示例的模型，用户需根据实际情况修改路径。该参数支持绝对路径和相对路径。相对路径是相对于 JSON 配置文件的路径。
    "model_path" : "../../../data/models/yolov3_b4c4_argb_mlu270.cambricon",
    // （必设参数）模型函数名。通过寒武纪神经网络框架生成离线模型时，通过生成的 twins 文件获取。
    "func_name" : "subnet0",
    // （可选参数）前处理类名。可继承 cnstream::Preproc 实现自定义前处理。在代码示例中，提供标准前处类 PreprocCpu 和 YOLOv3 的前处理类 PreprocYolov3。
    preproc_name" : "PreprocCpu",
    // （必设参数）后处理类名。可继承 cnstream::Postproc 实现自定义后处理操作。在代码示例中提供分类、SSD 以及 YOLOv3 后处理类。
    "postproc_name" : "PostprocYolov3",
    // （可选参数）攒 batch 的超时时间，单位为毫秒。即使用多 batch 进行推理时的超时机制。当超过指定的时间时，该模块将直接进行推理，不再继续等待上游数据。
    "batching_timeout" : 100,
    "device_id" : 0    // （可选参数）设备 id，用于标识多卡机器的设备唯一编号。
  }
}
```

注意：

preproc_name、model_input_pixel_format、use_scaler、keep_aspect_ratio 几个参数与图像预处理息息相关。请仔细查阅使用说明，避免预处理不符合模型要求，导致错误的推理结果。

postproc_name 指定的网络后处理实现尤其需要注意数据摆放顺序的问题，请仔细查阅 postproc_name、data_order 两个参数。

5.3.1.1 model_path

设置离线模型路径。

5.3.1.2 func_name

离线模型中函数名，一般为 subnet0，可以从生成离线模型时生成的 twins 文件中获得。

5.3.1.3 postproc_name

指定后处理类名，后处理类用来处理离线模型输出数据。后处理类应继承自 cnstream::Postproc。后处理类实现方法可参考 `samples/common/postprocess/postprocess_ssd.cpp` 中实现。

5.3.1.4 preproc_name

指定预处理类名，预处理类用来实现自定义图像预处理，经过指定预处理类处理过后的数据将作为离线模型的输入进行推理。自定义预处理实现方法可参考 `samples/common/preprocess/preprocess_standard.cpp`。默认值为空字符串。

当未设置 preproc_name 或者设置为空字符串，且 use_scaler 参数为 false 时。将使用默认的 MLU 预处理，默认在 MLU 上实现的预处理将使用 IPU 资源进行颜色空间转换以及缩放和抠图操作。

预处理类应继承自 cnstream::Preproc 或 cnstream::ObjPreproc，详情查看object_infer。

注意：

该参数在 Inferencer 插件预处理系列参数中拥有最高优先级。

自定义预处理需要注意最后需将预处理结果转为 float 类型，并按照 NHWC 的顺序摆放。详情查看 cnstream::Preproc 声明。

5.3.1.5 use_scaler

MLU220 平台（包括 M.2 和 SOM) 上可使用该参数。默认值为 false。

scaler 为 MLU220 平台上专门用于图像预处理的硬件单元。

当设置为 true 时，将使用 scaler 做网络预处理，包括颜色空间转换、图像缩放和抠图操作。该参数优先级比 preproc_name 参数低。

使用该参数应保证输入的 YUV420 SP NV12/NV21 为 128 像素对齐，可参考 DataSource 插件的apply_stride_align_for_scaler。且需要注意 scaler 的输出为 4 通道 ARGB 数据且宽度为 32 的倍数。所以如果要使用 scaler，离线模型的输入规模中宽必须为 32 的倍数。

当网络输入规模中宽度不是 32 的倍数，一般做法是在生成离线模型之前在网络首层加入 crop 层，并设置网络首层接收 ARGB 输入。

以 Cambricon Caffe 框架为例。如果有一个 300x300 的输入规模的网络，在网络的量化时设置 input_format 为 ARGB，并在量化后在第一层卷积上加入 ResizeCrop 层，ResizeCrop 层输入规模为 300x320，输出为 300x300。

5.3.1.6 batching_timeout

设置攒 batch 超时时间，单位 ms。默认为 3000ms。

每次进行离线推理会根据离线模型的 batchsize，一次送入一个 batch 的数据进行推理。在每次攒 batch 的过程中，如果超过 batching_timeout 指定的时间还未攒够一个 batch 的图，则不继续等待数据，直接进行推理。

5.3.1.7 data_order

指定网络输出数据按何种顺序排布，默认为 NHWC。可选值有 NHWC、NCHW。

注意：

该参数决定了 postproc_name 中，指定的后处理类接收到的网络的输出数据的排布方式。

当指定的数据排布顺序与模型输出的数据排布不同时将使用 CPU 进行数据重排。模型的输出数据顺序可通过查看生成模型时伴生的 twins 文件得到。

5.3.1.8 threshold

浮点数，透传给后处理类。

5.3.1.9 infer_interval

正整数，默认为 1。抽帧推理，每隔 infer_interval 个数据推理一次。剩余的数据不会进行计算，但是不被丢弃。

例如，解码后输出 7 帧，infer_interval 设置为 3，则第 1、4、7 帧将进行推理，其它数据不进行推理。

5.3.1.10 object_infer

表示是否以检测目标为单位进行推理。可选值为 1、true、TRUE、True、0、false、FALSE、False。默认值为 false。

当为 false 时，以 DataSource 插件解码后的数据帧作为输入进行推理。

当为 true 时，以 CNFrameInfo::collection.Get<CNInferObjsPtr>(kCNInferObjsTag) 中存储的检测目标作为输入进行推理。

当 object_infer 为 true 时，preproc_name 指定的预处理类应继承自 cnstream::ObjPreproc。

当 object_infer 为 false 时，preproc_name 指定的预处理类应继承自 cnstream::Preproc。

5.3.1.11 object_filer_name

指定过滤器类名，过滤器用来过滤检存放在 CNFrameInfo::collection.Get<CNInferObjsPtr>(kCNInferObjsTag) 中存储的检测目标，可用来过滤不需要进行推理的检测目标。当 obj_infer 为 true 时生效。

过滤器应继承自 cnstream::ObjFiler 类。可参考 `samples/common/obj_filter/car_filter.cpp` 中实现。

5.3.1.12 keep_aspect_ratio

指定当图像预处理在 MLU 上进行时，图像是否保持长宽比进行缩放。当使用保持长宽比的方式进行缩放，图像将保持长宽比不变缩放至网络的输入大小，并在左右或上下补 0。

可选值 1、true、TRUE、True、0、false、FALSE、False。默认值为 false。

当 preproc_name 为空字符串且 use_scaler 为 false 时生效。

5.3.1.13 dump_resized_image_dir

调试功能，用于保存离线模型执行前的图像数据，用于检查预处理是否符合网络要求。

指定保存图片的目录路径。

5.3.1.14 model_input_pixel_format

用于指定离线模型输入要求的 4 通道颜色顺序。可选值为 ARGB32、ABGR32、RGBA32、BGRA32。默认值为 RGBA32。

5.3.1.15 mem_on_mlu_for_postproc

指定网络输出数据的内存是否存放在 MLU 设备内存上。可选值为 1、true、TRUE、True、0、false、FALSE、False。默认值为 false。

- 当为 false，则网络输出数据将搬运至主机测，并根据 data_order 指定的数据顺序摆放数据。
- 当为 true 时，则网络推理后不进行内存拷贝，指针直接传递给 postproc_name 指定的后处理类中进行处理。

详情请查看 cnstream::PostProc 类中声明。

5.3.1.16 saving_infer_input

指定是否保存网络预处理结果。可选值为 1、true、TRUE、True、0、false、FALSE、False。默认值为 false。

若为 true，则将网络预处理结果保存至 CNFrameInfo::collection.Get<CNInferDataPtr>(kCNInferDataTag) 中向后传递。

5.3.1.17 device_id

设置使用的设备 id，决定 MLU 解码使用的设备及解码后数据存放在哪张 MLU 卡上。

5.4 基于推理服务的神经网络推理模块

神经网络推理模块 2（Inferencer2）是基于 Cambricon EasyDK 推理服务实现的一个神经网络推理模块，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。

Inferencer2 与神经网络推理模块在功能上基本一致，区别如下：

- 推理服务提供了一套类似服务器的推理接口，Inferencer2 在推理服务的基础上实现了推理模块功能，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。
- 推理服务实例化模型的多个实例时，相对于神经网络推理模块，减少了 MLU 内存开销。
- 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。
- 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。

5.4.1 前处理

前处理是将原始数据处理成适合网络输入的数据。

用户可通过在 json 配置文件中，设置参数 `preproc_name` 选择相应的前处理，取值如下:

- 通过 CNCV（寒武纪计算机视觉库，详情参考《寒武纪计算机视觉库用户手册》）算子实现在 MLU 平台上对图像进行缩放和颜色空间转换和减均值除方差的功能。可将 `preproc_name` 参数设置为 `CNCV` 或 `cncv` 。
- 实现在 scaler 硬件上对图像进行缩放和颜色空间转换功能。可将 `preproc_name` 参数设置为 `SCALER` 或 `scaler` 。仅在 MLU220 平台上支持。

- 使用 EasyDK 中的 ResizeConvert 算子实现对图像进行缩放和颜色空间转换功能。可将 preproc_name 参数设置为 RCOP 或 rcop 。由于该算子已不再维护，此选项即将废弃，请使用功能更为全面的 CNCV 算子实现的前处理。
- 自定义前处理，传入自定义前处理的类名。可以通过继承 cnstream::VideoPreproc 类，并重写 Execute 函数，实现自定义前处理。例如，使用 CNStream 中提供的 VideoPreprocCpu 前处理，可将 preproc_name 参数设置为 VideoPreprocCpu 。

  CNStream 中提供开发样例，通过定义 VideoPreprocCpu 以及 VideoPreprocYolov3 前处理的类来自定义前处理，示例位于 samples/common/preprocess 文件夹下的 video_preprocess_standard.cpp 和 video_preprocess_yolov3.cpp 文件中。
- 如果未指定 preproc_name 的值，默认选择使用 CNCV 算子。
- 不支持传入空字符串作为 preproc_name。

json 配置文件详情查看inferencer2 说明。

5.4.2 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。MLU200 系列以 CNRT 作为推理后端，通过 model_path 参数设置离线模型的存放路径以及通过 func_name 参数指定子网络名字（一般为 subnet0）；MLU300 系列以 MagicMind 作为推理后端，通过 model_graph 参数设置模型图结构的存放路径，通过 model_data 参数设置模型参数的存放路径。详情查看inferencer2 说明。

5.4.3 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::VideoPostproc 类。

推理结束后，在推理服务内部对推理结果进行后处理。在推理模块内部调用 cnstream::VideoPostproc 类的 Execute 函数。

注意：

CNStream 将整个 batch 的数据拆分为一份一份的数据传入后处理函数中进行处理。Execute 函数每次输入的数据为一份推理结果。例如，对于一级推理，每次输入一帧数据的结果，对于二级推理，每次输入一个 object 的结果。

CNStream 提供自定义后处理的方法如下：

5.4.3.1 自定义后处理方法

1. 继承 cnstream::VideoPostproc 类。
2. 重写 Execute 函数，自定义处理每一份推理输出结果，并填入到 CNStream 的数据结构中。函数参数描述如下：
   - output_data（输入参数）：后处理的结果。参数类型为 infer_server::InferData*。可以从参数中获得推理前设置的用户数据：
     - 对一级推理，模块的 object_infer 参数设为 false。用户数据类型是 cnstream::CNFrameInfoPtr ，CNStream 中流动在各个模块间的数据结构。将结果数据信息填入到该数据结构中对应的字段上。获取方式如:

```
cnstream::CNFrameInfoPtr info = output_data->GetUserData<cnstream::CNFrameInfoPtr>
↪()
```

     - 对二级推理，模块的 object_infer 参数设为 true。用户数据类型是 std::shared_ptr<cnstream::CNInferObject> ，后推理结果对应的 object。将结果数据信息填入到该数据结构中对应的字段上。获取方式如:

```
std::shared_ptr<cnstream::CNInferObject> obj = output_data->GetUserData
↪<std::shared_ptr<cnstream::CNInferObject>>()
```

   - model_output（输入参数）：离线模型推理的结果，即后处理的输入。参数类型为 const infer_server::ModelIO& 。可以通过 model_output.buffers 获得模型的输出。离线模型可以包含多个输出，每个输出都保存在 buffers 对应的元素中。例如，获得模型输出 0 的所有数据，数据类型为 float，const float* data = reinterpret_cast<const float*>(model_output.buffers[0].Data());
   - 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& 。

如下是一个简单的一级推理分类网络后处理示例：

```
class VideoPostprocClassification : public cnstream::VideoPostproc {
 public:
  bool Execute(infer_server::InferData* output_data,
               const infer_server::ModelIO& model_output,
               const infer_server::ModelInfo& model_info) override;

  DECLARE_REFLEX_OBJECT_EX(VideoPostprocClassification, cnstream::VideoPostproc)
};  // class VideoPostprocClassification

IMPLEMENT_REFLEX_OBJECT_EX(VideoPostprocClassification, cnstream::VideoPostproc)

bool VideoPostprocClassification::Execute(infer_server::InferData* output_data,
```



```
                                        const infer_server::ModelIO& model_output,
                                        const infer_server::ModelInfo& model_info)
↪{
  // 获取模型输出数据
  const float* data = reinterpret_cast<const float*>(model_output.buffers[0].Data());
  auto len = model_info.OutputShape(0).DataCount();

  // 找出最大值（score）和最大值坐标（label）
  float max_score = 0;
  int label = 0;
  for (decltype(len) i = 0; i < len; ++i) {
    auto score = *(data + i);
    if (score > max_score) {
      max_score = score;
      label = i;
    }
  }

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

  // 作为一级推理，user_data 是 cnstream::CNFrameInfoPtr
  // 将结果填入 cnstream::CNFrameInfoPtr
  cnstream::CNFrameInfoPtr frame =
      output_data->GetUserData<cnstream::CNFrameInfoPtr>();
  cnstream::CNInferObjsPtr objs_holder =
      frame->collection.Get<cnstream::CNInferObjsPtr>(cnstream::kCNInferObjsTag);
  std::lock_guard<std::mutex> objs_mutex(objs_holder->mutex_);
  objs_holder->objs_.push_back(obj);
  return true;
}
```

5.4.4 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inferencer2 说明。

5.4.5 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。

MLU200 系列（CNRT 后端）：一般来说推理引擎数目设置为 MLU 的 IPU 核心数目除以模型的核心数目最为合适。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。CNRT 后端仅支持 MLU200 系列平台。

MLU300 系列（MagicMind 后端）：MagicMind 模型默认占据 MLU 的所有 IPU 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。

5.4.6 使用说明及参数说明

以下为 `vehicle_recognization_config_mlu370.json` 配置文件示例。该示例文件位于 `cnstream/samples/cns_launcher/configs` 目录下。

```
"detector" : {
  "class_name" : "cnstream::Inferencer2",    // （必设参数）推理类名。
  "parallelism" : 2,                         // （必设参数）并行度。
  "max_input_queue_size" : 20,               // （必设参数）最大队列深度。
  "next_modules" : ["classifier"],           // （必设参数）下一个连接模块的名称。
  "custom_params" : {                        // 特有参数。
    "model_graph" : "../../../data/models/yolov3_nhwc.graph",
    "model_data" : "../../../data/models/yolov3_nhwc.data",
    "postproc_name" : "VideoPostprocYolov3MM",
    "preproc_name" : "CNCV",
    "normalize" : true,
    "threshold" : 0.5,
    "batching_timeout" : 100,
```



```
    "engine_num" : 1,
    "keep_aspect_ratio" : true,
    "model_input_pixel_format" : "RGB24",
    "device_id" : 0
  }
},
```

模块特有参数说明如下：

- model_path：（CNRT 后端必设参数）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。
- func_name：（CNRT 后端必设参数）模型函数名。模型加载时必须用到的参数。
- model_graph：（MagicMind 后端必设参数）模型结构文件路径。
- model_data：（MagicMind 后端必设参数）模型参数文件路径。
- obj_filter_name：（可选参数）指定过滤器类名，过滤器用来过滤在 `CNFrameInfo::collection` 中存储的检测目标，Infernecer2 仅对过滤器指定的检测目标进行推理。当 obj_infer 为 true 时生效。
- postproc_name：（必设参数）后处理类名。详情参看inferencer2 后处理。
- preproc_name：（可选参数）前处理类名。详情参看inferencer2 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。
- data_order：（可选参数）模型输出数据摆放顺序。可设置为 `NCHW` 或者 `NHWC`。默认值为 `NHWC`。
- threshold：（可选参数）后处理输出阈值。默认值为 0。
- 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。
- object_infer：（可选参数）是否为二级推理。可以设置为 `true`、`1`、`TRUE` 以及 `True` 具有相同效果，代表二级推理，以数据帧中的目标作为输入。可以设置为 `false`,`0`,`FALSE` 以及 `False` 具有相同效果，代表一级推理，以数据帧作为输入。默认值为 false。
- mean：（可选参数）仅当 preproc_name 为 CNCV 时生效。指定数据的分通道均值，参数形如 `"0.485, 0.456, 0.406"`。
- std：（可选参数）仅当 preproc_name 为 CNCV 时生效。指定数据的分通道方差，参数形如 `"0.229, 0.224, 0.225"`。
- normalize：（可选参数）仅当 preproc_name 为 CNCV 时生效。指定数据减均值除方差前是否需要归

一化（即除以 255）。

- keep_aspect_ratio：（可选参数）缩放时是否保持宽高比，请根据模型进行选择。可以设置为 `true`、`1`、`TRUE` 以及 `True` 具有相同效果，代表保持宽高比。可以设置为 `false`、`0`、`FALSE` 以及 `False` 具有相同效果，代表不保持宽高比。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。对于使用 `RCOP` 前处理，该参数可以设置为 `ARGB32`、`ABGR32`、`RGBA32`、`BGRA32`。对于用户自定义前处理，该参数可以设置为 `ARGB32`、`ABGR32`、`RGBA32`、`BGRA32`、`RGB24` 以及 `BGR24`。用户可在自定义前处理类中通过 `model_input_pixel_format_` 成员变量获得该值。默认值为 `RGBA32`。

5.4.7 开发样例

5.4.7.1 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。

一级推理前处理示例：

- **VideoPreprocCpu** 类：标准前处理，通过颜色空间转换及缩放，将图片转换为适用离线网络的输入。定义在 `video_preprocess_standard.cpp` 文件中。
- **VideoPreprocYolov3** 类：提供 yolov3 网络的前处理（输入保持宽高比）。通过颜色空间转换，缩放以及补边，将图片转换为适用离线网络的输入。定义在 `video_preprocess_yolov3.cpp` 文件中。

次级推理前处理示例：

- **VideoObjPreprocCpu** 类：标准次级网络前处理。将 object 所在的 roi 区域截取出来。并通过颜色空间转换及缩放，将图片转换为适用离线网络的输入。定义在 `video_preprocess_standard.cpp` 文件中。

5.4.7.2 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：

一级推理后处理示例：

- **VideoPostprocClassification** 类：分类网络作为一级网络的后处理。定义在 `video_postprocess_classification.cpp` 文件中。
- **VideoPostprocYolov3** 类：提供 CNRT 后端 yolov3 网络的后处理（输入保持宽高比）。定义在 `video_postprocess_yolov3.cpp` 文件中。
- **VideoPostprocYolov3MM** 类：提供 MagicMind 后端 yolov3 网络的后处理（输入保持宽高比）。定义在 `video_postprocess_yolov3_mm.cpp` 文件中。
- **VideoPostprocSsd** 类：提供 ssd 网络的后处理。定义在 `video_postprocess_ssd.cpp` 文件中。

次级推理后处理示例：

- **VideoObjPostprocClassification** 类：分类网络作为次级网络的后处理。定义在 `video_postprocess_classification.cpp` 文件中。

5.5 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。该模块连接在神经网络推理模块后，通过在配置文件中指定追踪使用的离线模型以及使用的追踪方法来配置模块。

5.5.1 使用说明

配置追踪模块所需要的离线模型和追踪方法等。

```
“tracker” : {
“class_name” : “cnstream::Tracker”,        // （必设参数）Track 的类名。
“parallelism” : 4,                         // （必设参数）并行度。
“max_input_queue_size” : 20,               // （必设参数）数据输入队列长度。
“next_modules” : [“osd”],                  // （必设参数）下一个连接的模块名。
“custom_params” : {
   // （必设参数）追踪使用的离线模型的路径。该参数支持绝对路径和相对路径。相对路径是相对于 JSON 配置文件的路径。
    “model_path” : “xxx.cambricon”,
    “func_name” : “subnet0”,    // 如果设置将使用 MLU，如果不设置将使用 CPU。模型函数名。
    “track_name” : “FeatureMatch”        // （可选参数）追踪方法。目前仅支持 FeatureMatch 追踪方法。
  }
}
```

5.6 OSD 模块

OSD（On Screen Display）模块用于在图像上绘制对象，并输出图像为 BGR24 格式。

OSD 模块可以连接在下面模块后面，绘制需要的推理结果：

- 神经网络推理模块（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：

- RTSP 模块（RtspSink）：进行编码和 RTSP 推流。
- 展示模块（Displayer）：对结果进行展示。
- 编码模块（Encode）：编码成视频或者图片。

5.6.1 使用说明

例如在 vehicle_recognization_config_mlu370.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,
    "next_modules" : ["encode"],
    "custom_params" : {
      "label_path" : "../../../data/models/label_map_coco.txt",
      "font_path" : "../../data/wqy_zenhei.ttf",
      "label_size" : "normal",
      "text_scale" : 1,
      "text_thickness" : 1,
      "box_thickness" : 1,
      "secondary_label_path" : "../../../data/models/synset_words.txt",
      "attr_keys" : "classification",
      "logo" : "cambricon"
    }
}
```

配置文件中参数说明如下：

- class_name：（必设参数）模块名称。
- parallelism：（必设参数）模块并行度。
- max_input_queue_size：（必设参数）数据输入队列长度。
- next_modules：（必设参数）下一个连接模块名称。
- 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、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。

5.7 Encode 模块

Encode 为编码模块，主要用于编码视频和图像。

注意：

Encode 插件 CPU 编码依赖 OpenCV，假如系统安装的 OpenCV 版本低于 3.x，那么在输入帧率过快时编码器内部有可能会出现丢帧并导致视频卡顿等现象。

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）
- 神经网络推理模块（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。

5.7.1 使用说明

例如 `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。

```
"encode" : {
  "class_name" : "cnstream::Encode",
  "parallelism" : 2,
  "max_input_queue_size" : 20,
  "custom_params" : {
    "encoder_type" : "mlu",
    "codec_type" : "h264",
    "preproc_type" : "cpu",
    "use_ffmpeg" : "false",
```



```
    "dst_width": 1280,
    "dst_height": 720,
    "frame_rate" : 25,
    "kbit_rate" : 3000,
    "gop_size" : 30,
    "output_dir" : "./output",
    "device_id": 0
  }
}
```

配置参数说明如下：

- class_name：（必设参数）模块名称。
- parallelism：（必设参数）模块并行度。
- max_input_queue_size：（必设参数）数据输入队列长度。
- encoder_type：（可选参数）编码类型。可设置的值包括：
  - cpu：使用 CPU 编码（默认值）。
  - mlu：使用 MLU 编码。
- codec_type：（可选参数）编码的格式。可设置的值包括：
  - h264（默认值）
  - h265
  - jpeg
- preproc_type：（可选参数）前处理使用的类型是 cpu 还是 mlu。可设置的值包括：
  - cpu(默认值)
  - mlu(目前不支持)
- use_ffmpeg：（可选参数）是否使用 ffmpeg 进行大小调整和色彩空间转换。可设置的值包括：
  - true
  - false(默认值)
- dst_width：（可选参数）输出图像宽度，单位像素。注意当使用 mlu 解码时，设置输出图像宽度不为奇数。
- dst_height：（可选参数）输出图像高度，单位像素。注意当使用 mlu 解码时，设置输出图像高度不为奇数。
- frame_rate：（可选参数）编码后视频的帧率。默认值为 25。
- kbit_rate：（可选参数）单位时间内编码的数据量。默认值为 1Mbps，仅当在 mlu 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 30。
- output_dir：（可选参数）视频解码后保存的地址。如果不指定，则不显示保存解码后视频或图片。默认值为 `{CURRENT_DIR}/output`.
- device_id：（可选参数）当 `encoder_type` 或者 `preproc_type` 设置为 `mlu` 时，必须指定设备的 id。

5.8 Display 模块

Display 模块是 CNStream 中基于 SDL 视频播放插件开发的多媒体展示模块。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。

Display 模块支持客户在播放视频过程中，选择播放窗口大小，并且支持全屏播放功能。对于不同路的视频流可以做到同时进行播放，可以通过设置 `max-channels` 对播放的视频流个数进行设置。该值若是小于输入视频个数，则只会展示前几路输入视频流。

Display 模块一般连接下面模块后面：

- 神经网络推理模块（Inferencer）
- 追踪模块（Tracker）
- OSD 模块（Osd）

Display 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

例如 `display.json` 配置文件如下，该文件位于 `samples/cns_launcher/configs/sinker_configs/` 目录下。配置 Display 模块所需要的窗口大小、视频流个数以及刷新帧率等参数。Display 模块和 Encode 模块一样，处于 Pipeline 最后位置的模块，所以不需要 `next_modules` 参数的设置。

```
“displayer” : {
“class_name” : “cnstream::Displayer”,
“parallelism” : 4,
“max_input_queue_size” : 20,
“custom_params” : {
   // 设置视频流播放时的窗口大小和刷新帧率等信息。
    “window-width” : “1920”,
    “window-height” : “1080”,
    “refresh-rate” : “25”,
   "max-channels" : "32",
   "show" : "false",
   "full-screen" : "false"
   }
}
```

配置参数说明如下：

- class_name：（必设参数）模块名称。
- parallelism：（必设参数）模块并行度。
- max_input_queue_size：（必设参数）数据输入队列长度。
- window-width：（必设参数）播放窗口的宽度，单位像素。

- window-height：（必设参数）播放窗口的高度，单位像素。
- refresh-rate：（必设参数）播放时的刷新帧率。
- max-channels：（必设参数）播放的视频流路数。建议大于等于输入视频流路数。
- show：（必设参数）是否播放视频流，设置为 `true` 之前需要确保顶层 `CMakeList.txt` 文件中 `build_display` 选项设置为 `ON`。
- full-screen：（可选参数）是否进行全屏播放。

5.9 RTSP Sink 模块

RTSP（Real Time Streaming Protocol）Sink 模块主要用于对每帧数据进行预处理，将图调整到需要的大小，并进行编码及 RTSP 推流。

RTSP Sink 模块提供 single 模式和 mosaic 模式来处理数据流。single 模式下，每个窗口仅显示一路视频，如 16 路视频会有 16 个端口，每个端口打开都是一个窗口，显示对应路的视频流。而 mosaic 模式下，多路视频仅有一个端口，所有路的视频都在一个窗口上显示。如 16 路视频只有一个端口，打开这个端口，显示的是 4×4 的拼图。

RTSP Sink 模块处理数据流程如下：

图 5.1: RTSP Sink 模块数据处理流程

5.9.1 使用说明

用户可以通过配置 JSON 文件方式设置和使用 RTSP Sink 模块。JSON 文件的配置参数说明如下：

- color_mode：（可选参数）颜色空间。可设置的值包括：
  - bgr：输入为 BGR。
  - nv：输入为 YUV420NV12 或 YUV420NV21。（默认值）
- preproc_type：（可选参数）预处理 (resize)。可设置的值包括：
  - cpu：在 CPU 上进行预处理。（默认值）
  - mlu：在 MLU 上进行预处理。(暂不支持)
- encoder_type：（可选参数）编码。可设置的值包括：
  - ffmpeg：在 CPU 上使用 ffmpeg 进行编码。
  - mlu：在 MLU 上进行编码。（默认值）
- device_id：（可选参数）设备号。仅在使用 MLU 时生效。默认使用设备 0。

- view_mode：（可选参数）显示界面。可设置的值包括：
  - single：single 模式，每个端口仅显示一路视频，不同路视频流会被推到不同的端口。（默认值）
  - mosaic：mosaic 模式，实现多路显示。根据参数 `view_cols` 和 `view_rows` 的值，将画面均等分割，默认为 4*4。

    注意：

    使用 mosaic 模式时，注意下面配置：

    * `view_cols` * `view_rows` 必须大于等于视频路数。以 2*3 为特例，画面将会分割成 1 个主窗口（左上角）和 5 个子窗口。
    * mosaic 模式仅支持 BGR 输入。

- view_cols：（可选参数）多路显示列数。仅在 mosaic 模式有效。取值应大于 0。默认值为 0。
- view_rows：（可选参数）多路显示行数。仅在 mosaic 模式有效。取值应大于 0。默认值为 0。
- udp_port：（可选参数）UDP 端口。格式为：`url=rtsp://本机 ip:9554/rtsp_live`。
- http_port：（可选参数）RTSP-over-HTTP 隧道端口。默认值为 8080。
- dst_width：（可选参数）输出帧的宽。取值为大于 0，小于原宽。只能向下改变大小。默认值为 0（原宽）。
- dst_height：（可选参数）输出帧的高。取值为大于 0，小于原高。只能向下改变大小。默认值为 0（原高）。
- frame_rate：（可选参数）编码视频帧率。取值为大于 0。默认值为 25。
- kbit_rate：（可选参数）编码比特率。单位为 kb，需要比特率/1000。取值为大于 0。默认值为 1000。
- gop_size：（可选参数）GOP（Group of Pictures），两个 I 帧之间的帧数。取值为大于 0。默认值为 30。

5.9.2 配置文件示例

Single 模式

```
"rtsp_sink" : {
    "class_name" : "cnstream::RtspSink",
    "parallelism" : 16,
    "max_input_queue_size" : 20,
    "custom_params" : {
      "http_port" : 8080,
      "udp_port" : 9554,
      "frame_rate" :25,
      "gop_size" : 30,
      "kbit_rate" : 3000,
      "view_mode" : "single",
      "dst_width" : 1920,
      "dst_height": 1080,
```



```
      "color_mode" : "bgr",
      "encoder_type" : "ffmpeg",
      "device_id": 0
    }
  }
```

Mosaic 模式

```
"rtsp_sink" : {
    "class_name" : "cnstream::RtspSink",
    "parallelism" : 1,
    "max_input_queue_size" : 20,
    "custom_params" : {
      "http_port" : 8080,
      "udp_port" : 9554,
      "frame_rate" :25,
      "gop_size" : 30,
      "kbit_rate" : 3000,
      "encoder_type" : "ffmpeg",
      "view_mode" : "mosaic",
      "view_rows": 2,
      "view_cols": 3,
      "dst_width" : 1920,
      "dst_height": 1080,
      "device_id": 0
    }
  }
```

5.10 单进程单 Pipeline 中使用多个设备

在单进程、单个 pipeline 场景下，CNStream 支持不同模块在不同的 MLU 卡上运行。用户可以通过设置模块的 `device_id` 参数指定使用的 MLU 卡。

下面以 DataSource 和 Inference 模块使用场景为例，配置 DataSource 模块使用 MLU 卡 0，Inference 模块使用 MLU 卡 1。单进程中建议在 DataSource 模块中复用 codec 的 buffer，即应设置 `reuse_codec_buf` 为 `true`。

```
{
  "source" : {
    "class_name" : "cnstream::DataSource",     // 数据源类名。
    "parallelism" : 0,                         // 并行度，无效值，设置为 0 即可。
```



```
    "next_modules" : ["infer"],                         // 下一个连接模块名称。
    "custom_params" : {                         // 特有参数。
      "reuse_cndec_buf" : "true",               // 是否复用 codec 的 buffer。
      "output_type" : "mlu",                    // 输出类型，可以设置为 MLU 或 CPU。
      "decoder_type" : "mlu",                   // decoder 类型，可以设置为 MLU 或 CPU。
      "device_id" : 0                           // 设备 id，用于标识多卡机器的设备唯一标号。
    }
  },

   "infer" : {
    "class_name" : "cnstream::Inferencer",      // 推理类名。
    "parallelism" : 1,                          // 并行度。
    "max_input_queue_size" : 20,                // 最大队列深度。
    "custom_params" : {                         // 特有参数。
      //模型路径。
      "model_path" : "../../data/models/MLU270/Classification/resnet50/resnet50_offline.
↪cambricon",
      "func_name" : "subnet0",                      // 模型函数名。
      "postproc_name" : "PostprocClassification",   // 后处理类名。
      "batching_timeout" : 60,                      // 攒 batch 的超时时间。
      "device_id" : 1                               // 设备 id，用于标识多卡机器的设备唯一标号。
}
  }
}
```

5.11 多进程操作

由于 pipeline 只能进行单进程操作，用户可以通过 ModuleIPC 模块将 pipeline 拆分成多个进程，并完成进程间数据传输和通信，例如最常见的解码和推理进程分离等。ModuleIPC 模块继承自 CNStream 中的 Module 类。两个 ModuleIPC 模块组成一个完整的进程间通信。此外，通过定义模块的 `memmap_type` 参数，可以选择进程间的内存共享方式。

CNStream 支持在单个 pipeline 中，不同的进程使用不同的 MLU 卡执行任务。用户可以通过设置模块的 `device_id` 参数指定使用的 MLU 设备。

5.11.1 使用示例

下面以进程 1 做解码，进程 2 做推理为例，展示了如何使用 ModuleIPC 模块完成多进程设置和通信，以及设置各进程使用不同的 MLU 卡。

1. 创建配置文件，如 `config_process1.json`。在配置文件中设置进程 1。第一个模块配置为解码模块，然后设置一个 ModuleIPC 模块。主要参数设置如下：
   - 设置 `ipc_type` 参数值为 **client**，做为多进程通信的客户端。
   - 设置 `memmap_type` 参数值为 **cpu**。当前仅支持 CPU 内存共享方式。后续会支持 MLU 内存共享方式。
   - 设置 `socket_address` 参数值为进程间通信地址。用户需定义一个字符串来表示通信地址。
   - 设置不同进程使用不同的 MLU 卡：设置 Decode 进程使用 MLU 卡 0。但配置 ModuleIPC 模块时，无需设置 `device_id`。另外，多进程使用中，不建议在 source module 中复用 codec 的 buffer，即应设置 `reuse_codec_buf` 设为 false。

   示例如下：

```
{
  "source" : {
    "class_name" : "cnstream::DataSource",  // 数据源类名。
    "parallelism" : 0,                      // 并行度。无效值，设置为 0 即可。
    "next_modules" : ["ipc"],               // 下一个连接模块的名称。
    "custom_params" : {                     // 特有参数设置。
      "source_type" : "ffmpeg",             // source 类型。
      "reuse_cndec_buf" : "false",          // 是否复用 codec 的 buffer。
      "output_type" : "mlu",                // 输出类型，可以设置为 MLU 或 CPU。
      "decoder_type" : "mlu",               // decoder 类型，可以设置为 MLU 或 CPU。
      "device_id" :0                        // 设备 id，用于标识多卡机器的设备唯一标号。
    }
  },

  "ipc" : {
    "class_name" : "cnstream::ModuleIPC",  // 进程间通信类名。
    "parallelism" :1 ,                     // 并行度，针对 client 端，设置为 1。
    "max_input_queue_size" : 20,           // 最大队列深度。
    "custom_params" : {                    // 特有参数设置。
      "ipc_type" : "client",               // 进程间通信类型，可设为 client 和 server。上游进程
设置为 client，下游进程设置为 server。
      "memmap_type" : "cpu",               // 进程间内存共享类型，可以设置为 CPU。
      "max_cachedframe_size" : "40",       // 最大缓存已处理帧队列深度，仅 client 端有该参数。
      "socket_address" : "test_ipc"        // 进程间通信地址，一对通信的进程，需要设置为相同的
通信地址。
    }
  }
```



```
}
```

2. 创建配置文件，如 config_process2.json。在配置文件中设置进程 2。第一个模块配置为 ModuleIPC 模块，然后设置一个推理模块。主要参数设置如下：
   - 在 ModuleIPC 模块中，设置 ipc_type 参数值为 **server**，做为多进程通信的服务器端。
   - 在 ModuleIPC 模块中，设置 memmap_type 参数值为 **cpu**。当前仅支持 CPU 内存共享方式。后续会支持 MLU 内存共享方式。
   - 在 ModuleIPC 模块中，设置 socket_address 参数值为进程间通信地址。用户需定义一个字符串来表示通信地址。
   - 设置不同进程使用不同的 MLU 卡：设置 Inference 进程使用 MLU 卡 1。但配置 ModuleIPC 模块时，需要指定 device_id。该 device_id 的值应与推理模块设置的 device_id 的值保持一致。

注意：

memmap_type 与 socket_address 的参数值设置需要与进程 1 中 ModuleIPC 模块的相关参数设置保持一致。

```
{
  "ipc" : {
    "class_name" : "cnstream::ModuleIPC",  // 进程间通信类名。
    "parallelism" : 0,                     // 并行度，无效值，针对 server 端，设置为 0 即可。
    "next_modules" : ["infer"],            // 下一个连接模块名称。
    "custom_params" : {                    // 特有参数设置。
      "ipc_type" : "server",               // 进程间通信类型，可设为 client 和 server。上游进程设置为 client，下游进程设置为 server。
      "memmap_type" : "cpu",               // 进程间内存共享类型，可以设置为 CPU。
      "socket_address" : "test_ipc",       // 进程间通信地址，一对通信的进程，需要设置为相同的通信地址。
      "device_id":1                        // 设备 id，用于标识多卡机器的设备唯一标号。
    }
  },

  "infer" : {
    "class_name" : "cnstream::Inferencer",   // 推理类名。
    "parallelism" : 1,                       // 并行度。
    "max_input_queue_size" : 20,             // 最大队列深度。
    "custom_params" : {                      // 特有参数设置。
      "model_path" : "../../data/models/MLU270/Classification/resnet50/resnet50_offline.cambricon",   // 模型路径。
      "func_name" : "subnet0",                        // 模型函数名。
      "postproc_name" : "PostprocClassification",     // 后处理类名。
      "batching_timeout" : 60,                        // 攒 batch 的超时时间。
      "device_id" : 1                                 // 设备 id，用于标识多卡机器的设备唯一标号。
```

```
    }
  }
}
```

5.12 kafka 模块

kafka 模块基于 librdkafka 将 CNFrameInfo 生成 kafka 消息数据。目前是一个试验性的内置模块，未充分考虑各平台的兼容情况，源码位于 modules_contrib 目录下。

kafka 模块可以连接在下面模块后面，将结构化信息广播到网络上：

- 数据源模块（cnstream::DataSource）
- 神经网络推理模块（cnstream::Inferencer）
- 神经网络推理模块 2 (cnstream::Inferencer2)
- 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。

5.12.1 使用说明

```
"kafka" : {
  "class_name" : "cnstream::KafkaClient",
    "parallelism" : 2,
    "max_input_queue_size" : 20,
    "custom_params" : {
      "handler" : "CnKafka",
      "broker" : "0",
      "topic" : "cnstream"
    }
}
```

配置文件中参数说明如下：

- class_name：（必设参数）模块名称。
- parallelism：（必设参数）模块并行度。
- max_input_queue_size：（必设参数）数据输入队列长度。
- 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 `CnKafka` 。
- broker：（必选参数）消息经纪人名称。

- 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`。

Cambricon 寒武纪

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 paramSet) 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
↪<ExampleModuleSource> {
  using super = cnstream::SourceModule;

 public:
  explicit ExampleModuleSource(const std::string &name) : super(name) {}
  bool Open(cnstream::ModuleParamSet paramSet) 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
↪<ExampleModuleEx> {
  using super = cnstream::ModuleEx;
  using FrameInfoPtr = std::shared_ptr<cnstream::CNFrameInfo>;

 public:
  explicit ExampleModuleEx(const std::string &name) : super(name) {}
  bool Open(cnstream::ModuleParamSet paramSet) 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=(ExampleModuleEx const &) = delete;
};
```


7 编程模型

本章重点介绍了 CNStream 在寒武纪软件栈是如何工作的，也介绍了文件目录以及如何快速开始使用内置模块进行编程。

7.1 寒武纪软件栈

CNStream 作为寒武纪视频结构化分析特定领域的框架，在整个寒武纪应用软件栈中起着承上启下的作用。CNStream 能快速构建自己的视频分析应用，并获得比较高的执行效率。用户无需花费精力在一些底层的细节上，从而有更多时间关注业务的发展。下图展示了 CNStream 在软件栈中的位置关系。

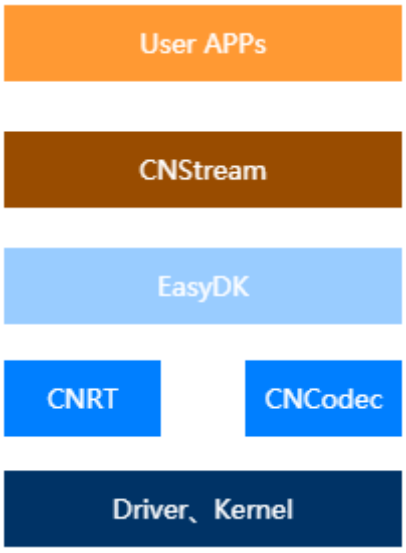

图 7.1: 寒武纪软件栈框图

7.2 文件目录

在 CNStream 源码目录下，主要由以下部分组成：

- 核心框架：在 `framework/core` 文件夹下，包含创建 Pipeline，Module 等基础核心源码。
- 内置模块：在 `modules` 文件夹下，内置一套标准的视频结构化模块组件。
- 动态库：编译成功后，CNStream 核心框架存放在 `libcnstream_core.so` 文件中，内置的视频结构化模块存放在 `libcnstream_va.so` 中。位于 `lib` 目录下。
- 示例程序源码：一系列示例程序存放在 `samples` 文件夹下。
- 样例数据：示例程序和测试程序所用到的数据文件，包括图片、短视频等。另外程序运行过程中从 Model Zoo 下载的离线模型文件也会存放在该文件夹下。

CNStream 仓库主要目录结构及功能如下图所示：

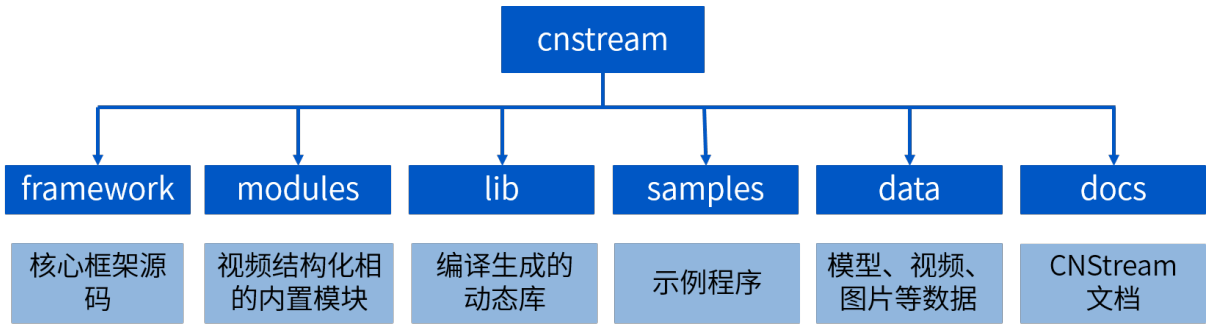

图 7.2: CNStream 文件目录结构图

7.3 编程指南

CNStream 是典型的基于 pipeline 和模块机制的编程模型。支持在 pipeline 注册多个内置模块 或者自定义模块。这些模块通过隐含的队列连接，使用一个 JSON 的文本描述模块间的连接关系。

应用开发的通用编程步骤如下：

1. 创建一个 pipeline 对象。
2. 读入预先编排的 JSON 文件构建数据流 pipeline。
3. 创建消息监测模块，并注册到 pipeline。
4. 启动 pipeline。
5. 动态增加数据源。

详情请查看创建应用程序。

8 创建应用程序

8.1 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。

8.2 应用程序的创建

8.2.1 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。

8.2.1.1 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、Encoder，以及 ssd 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 MLU 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" : {
    "output_type" : "mlu",
    "decoder_type" : "mlu",
    "output_buf_number" : 32,
    "device_id" : 0
  }
},

"detector" : {
  // Inferencer 模块。设置使用 resnet34ssd 离线模型，使用 PostprocSsd 进行网络输出数据后处理，并行度为 2，模块输入队列的 max_size 为 20。
  "class_name" : "cnstream::Inferencer",
  "parallelism" : 2,
  "max_input_queue_size" : 20,
  "next_modules" : ["tracker"],
  "custom_params" : {
    "model_path" : "../data/models/yolov3_b4c4_argb_mlu270.cambricon",
    "func_name" : "subnet0",
    "postproc_name" : "PostprocYolov3",
    "keep_aspect_ratio" : "true",
    "model_input_pixel_format" : "ARGB32",
    "batching_timeout" : 100,
    "device_id" : 0
  }
},

"tracker" : {
  // Tracker 模块。设置使用 track 离线模型，并行度为 2，模块输入队列的 max_size 为 20。
  "class_name" : "cnstream::Tracker",
  "parallelism" : 2,
  "max_input_queue_size" : 20,
```



```
    "next_modules" : ["osd"],
    "custom_params" : {
      "model_path" : "../data/models/feature_extract_for_tracker_b4c4_argb_mlu270.cambricon",
      "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" : {
      "chinese_label_flag" : "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::Encode",
    "parallelism" : 2,
    "max_input_queue_size" : 20,
    "custom_params" : {
      "dst_width": 1280,
      "dst_height": 720,
      "frame_rate" : 25,
      "kbit_rate" : 3000,
      "gop_size" : 30,
      "codec_type" : "h264",
      "preproc_type" : "cpu",
      "encoder_type" : "mlu",
      "use_ffmpeg" : "false",
      "output_dir" : "./output",
      "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_MLU270.json",
    "next_modules" : ["subgraph:object_tracking"]
  },

  "subgraph:object_tracking" : {
    "config_path" : "../configs/object_tracking_MLU270_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以及检查 JSON 配置文件的正确性。详情查看Inspect 工具 。

8.2.1.2 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` 。

8.2.2 非配置文件方式

CNStream 针对非配置文件方式提供了一些完整的、独立的应用程序开发示例。参见 CNStream 源代码中 `samples/simple_run_pipeline/simple_run_pipeline.cpp`。

8.3 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`。

8.3.1 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。

8.3.2 2. FRAME_ERR_MSG

FRAME_ERR_MSG 表示帧解码失败消息，当前仅支持使用 MLU 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；

（2）从内存中输入 JPEG 数据流时，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、pts 和内部赋值定义的 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。

8.3.3 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。

8.3.4 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。

9 web 可视化工具

Web 可视化工具提供了图形化界面设计、配置和运行 pipeline，并能够直观地查看运行输出结果。从而帮助用户快速体验寒武纪数据流分析框架。

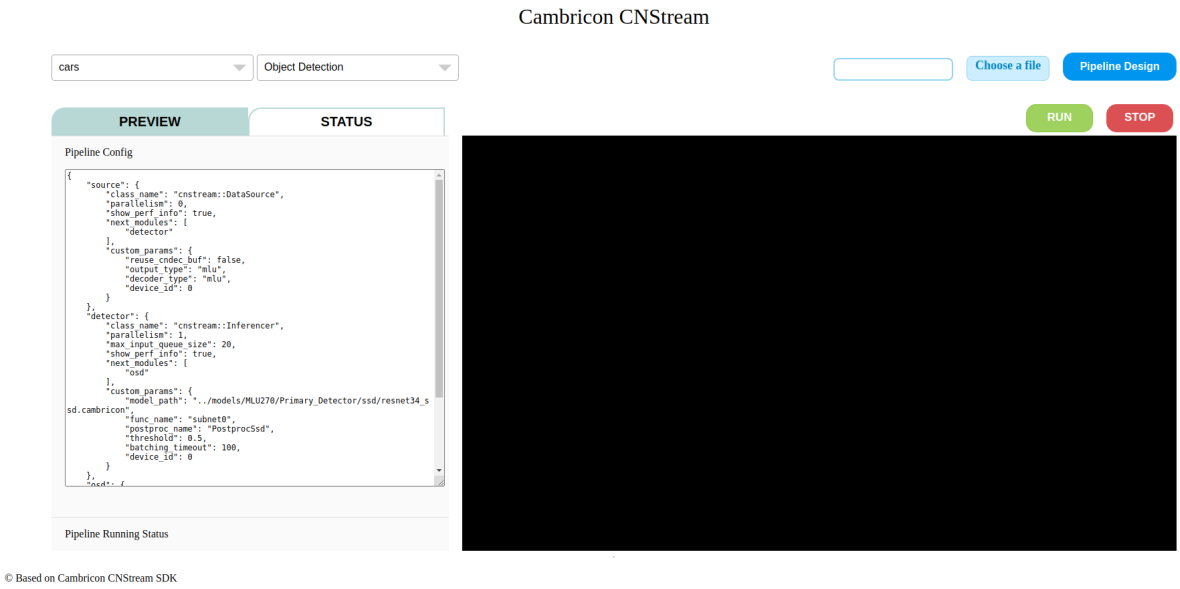

图 9.1: Web 可视化工具主页面

9.1 功能介绍

web 可视化工具主要提供以下功能：

- 图形化界面设计和配置 pipeline：
  - 提供内置的 pipeline 示例配置，用户可以直接运行示例，快速体验如何使用 CNStream 开发应用。
  - 支持在线设计和配置 pipeline。提供内置模块的流程块，支持像绘制流程图一样在 web 端绘制 pipeline，选择表示模块的流程块至设计框，并通过连线连接数据流向。
    - 支持修改模块参数配置。
    - 提供 pipeline 配置正确性自动检测，包括基本的模块参数配置和流程图的环自动检测。
    - 流程图绘制完成后，可以通过下载为 JSON 文件或者跳转至主页面运行或预览。
- 支持部分数据源选用和上传：
  - 默认支持 cars、people、images、objects 四种类型的数据源。
  - 支持上传视频文件为数据源。

- 支持 preview 和 status 两种模式运行 pipeline。
  - PREVIEW 模式下运行 pipeline，可以预览运行的视频结果。
  - Status 模式下运行 pipeline，会显示 pipeline 运行的状态，显示运行的 fps、latency 等信息。

9.2 首次使用前部署与设置

Web 可视化工具使用前，执行下面步骤完成部署和配置：

1. 在 ${CNSTREAM_DIR}/tools/web_visualize 目录下执行下面脚本，配置需要的环境。其中 ${CNSTREAM_DIR} 代表 CNStream 源码目录。

```
./prerequired.sh
```

2. 在 ${CNSTREAM_DIR}/tools/CMakeLists.txt 文件中打开 Web 可视化功能。

```
option(build_web_visualize "build web visualize" ON)
```

3. 编译 cnstream。在 ${CNSTREAM_DIR}/build 目录下运行下面命令：

```
cmake ..
make
```

4. 在 ${CNSTREAM_DIR}/tools/web_visualize 目录下执行下面脚本来运行 web 可视化工具。

```
./run_web_visualize.sh
```

5. web 可视化工具启动后，根据屏幕上会显示“Listening at:”字段。根据该字段的 IP 地址和端口号打开浏览器访问 web 可视化工具。例如：

```
[2020-09-08 17:39:31 +0800] [34621] [INFO] Listening at: http://0.0.0.0:9099 (34621)
```

9.3 Pipeline 的设计和配置

Web 可视化工具提供可视化的界面帮助用户快速搭建 pipeline。用户只需拖拽内置模块，并关联模块，即可完成 pipeline 的搭建。

Pipeline 的设计页面如下所示：

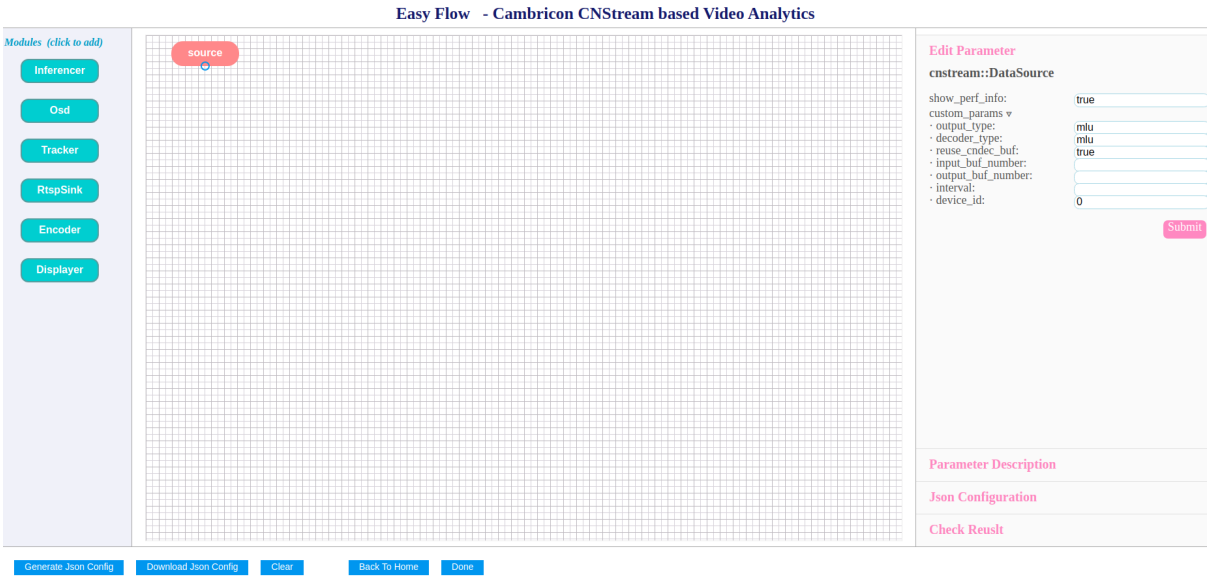

图 9.2: Pipeline 设计页面

9.3.1 设计和配置 pipeline

执行下面步骤完成 pipeline 设计：

1. 在 web 可视化工具主页面，点击 **Pipeline Design** 按钮。进入 Pipeline 设计页面。
2. 单击页面最左边想要添加的模块名，模块会添加到 pipeline 设计框。
3. 在 pipeline 设计框中单击上一步添加的模块，页面最右边“Edit Parameter”中，输入该模块的配置参数。在“Parameter Description”中可以查看该模块的参数设置。
4. 在 pipeline 设计框中连接各模块。

根据用户设计，相对应的 JSON 配置文件会自动生成。用户可以在页面最右边“Json Configuration”中可以查看 JSON 配置文件。Web 可视化工具会自动检测 pipeline 和各模块配置是否正确并在“Check Result”中查看配置是否成功。

配置完成后，用户可以运行自定义 pipeline 查看运行结果。

注意：

如果想要使用 Preview 模式运行 pipeline 后做数据的预览，由于数据预览需要同步数据，所以 pipeline 设计的末端只能有一个节点，即末端必须为汇聚节点。

9.3.2 生成和下载 JSON 配置文件

pipeline 设计完成后，Web 可视化工具会自动生成对应的 JSON 配置文件。用户通过点击 Pipeline 设计页面下的 **Generate Json Config** 按钮生成 JSON 配置文件。并点击 **Download Json Config** 按钮下载配置文件。

9.4 运行内置的 pipeline 示例

Web 可视化工具提供了 pipeline 示例，用户可以直接运行示例，并直接在页面查看运行结果。

1. 在 web 可视化工具主页面，从下拉菜单中选择数据源类型。目前支持 cars、people、images、objects 四种类型的数据源。
2. 在下拉菜单中选择任务类型类型，目前提供以下一种示例：Classification、Object Detection、Object Tracking、Secondary Classification。
3. 选择 Preview 或者 Status 标签。
4. 点击 **RUN** 按钮。

如果选择 Preview 模式，页面会在“Pipeline Config”中显示示例对应的 JSON 配置文件内容，并在右边显示输出的视频。如果选择 Status 模式，页面会在“Pipeline Config”中显示示例对应的 JSON 配置文件内容，并在“Pipeline Running Status”中显示运行的 fps、latency 等信息。

9.5 运行自定义 pipeline

用户可以通过 Web 可视化工具运行已设计的 pipeline。

1. pipeline 设计完成后，点击 Pipeline 设计页面下的 **Done** 按钮。返回到 Web 可视化工具主页面。有关如何设计 pipeline，查看设计 pipeline。
2. 上传数据源。点击 **Choose a file** 按钮选择数据源文件。目前只支持视频文件。
3. 选择 Preview 或者 Status 标签。
4. 点击 **RUN** 按钮。

如果选择 Preview 模式，页面会在“Pipeline Config”中显示示例对应的 JSON 配置文件内容，并在右边显示输出的视频。如果选择 Status 模式，页面会在“Pipeline Config”中显示示例对应的 JSON 配置文件内容，并在“Pipeline Running Status”中显示运行的 fps、latency 等信息。

10 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 。

10.1 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。

10.1.1 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
    -c, --check                         Check the config file
    -v, --version                       Print version information
```


10.1.2 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.
  ...                                      ...
```

10.1.3 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.
...                           ...
```

10.1.4 检查配置文件的合法性

输入下面命令检查配置文件合法性：

```
./cnstream_inspect -c $CNSTREAM_HOME/samples/cns_launcher/configs/decode_config.json
```

配置文件的检查包括模块、模块参数以及模块前后的连接。如果检查没有错误，则会显示如下信息：

```
Check module config file successfully!
```

否则，请根据提示信息修改配置文件。

例如，配置文件中模块名字写错，将 DataSource 写成 DataSource：

```
{
  "source" : {
    "class_name" : "cnstream::DataSource",
    ...
```



```
  },
}
```

命令返回示例如下：

```
Check module configuration failed, Module name : [source] class_name : [cnstream::DataSource]␣
↪non-existent.
```

10.1.5 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: v6.0.0
```

10.2 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 Encoder 模块为例：

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

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 Encoder 模块为例。

   ```
   param_register_.SetModuleDesc("Encoder is a module for encode the video or image.");
   ```

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 Encoder 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 IsRegisted(const std::string& key);
  void SetModuleDesc(const std::string& desc); // 设置模块描述。
};
```

5. 为了检查配置文件中参数的合法性，还需要实现父类 **cnstream::Module** 的 CheckParamSet 函数。

```
virtual bool CheckParamSet(ModuleParamSet paramSet) { return true; }
```

例如：

```
bool Inferencer::CheckParamSet(ModuleParamSet paramSet) {
  ParametersChecker checker;

  // 对配置文件中的配置项判断是否是已注册的，如不是，给出 WARNING 信息。
  for (auto& it : paramSet) {
    if (!param_register_.IsRegisted(it.first)) {
      LOG(WARNING) << "[Inferencer] Unknown param: " << it.first;
    }
  }

  // 对一些必要参数进行检查配置文件是否配置。
  if (paramSet.find("model_path") == paramSet.end()
      || paramSet.find("func_name") == paramSet.end()
      || paramSet.find("postproc_name") == paramSet.end()) {
    LOG(ERROR) << "Inferencer must specify [model_path], [func_name], [postproc_name].";
    return false;
  }

  // 检查模块路径是否存在。
  if (!checker.CheckPath(paramSet["model_path"], paramSet)) {
    LOG(ERROR) << "[Inferencer] [model_path] : " << paramSet["model_path"] << " non-
↪existence.";
    return false;
  }

  // 检查 batching_timeout 和 device_id 是否设为数字。
  std::string err_msg;
```

```
  if (!checker.IsNum({"batching_timeout", "device_id"}, paramSet, err_msg)) {
    LOG(ERROR) << "[Inferencer] " << err_msg;
    return false;
  }

  return true;
}
```


11 Log 工具

Log 工具是一个 C++ 流式日志工具，支持分级输出日志到终端或文件等功能，用户无需安装和启动该工具，通过 `LOGx(category)` 宏直接使用该工具。

11.1 使用说明

通过调用 `LOGx(category)` 宏，CNStream 将日志打印在屏幕上。`x` 代表日志等级，详情参看日志等级。`category` 用于区分日志信息的类别，由用户自定义，可以是模块名称，建议名字全大写，可以使用特殊符号。

日志语句调用示例：

```
LOGE(SOURCE) << "Error number: " << 326623;
```

示例中：

- 日志等级为 `LOGE`，1 级，即返回 FATAL 和 ERROR 级别的日志。
- category 为 `SOURCE`。
- 用户的日志信息为 `Error number: 326623`。

返回日志信息示例如下：

```
CNSTREAM SOURCE E0103 00:16:30.448275 30804] Error number: 326623
```

以上信息中：

- `CNSTREAM` 为关键字。
- `SOURCE` 代表该日志 category。
- `E` 代表该条日志的等级。
- `0103 00:16:30.448275` 是日志输出的时间。
- `30804` 为线程 ID（TID）。
- `Error number: 326623` 是用户的日志信息。

11.1.1 显示源码文件名和源码行号

如果想要显示源码文件名和源码行号字段，可在 CNStream 编译时，通过设置 -DRELEASE=OFF 编译开关开启 DEBUG 模式，返回信息示例如下：

```
CNSTREAM SOURCE E0103 00:16:30.448275 30804] filename.cpp:100 error messages:326623
```

filename.cpp 代表这条日志所在的源码文件，100 表示日志所在源码文件中的行号。文件名和行号由日志语句所处实际位置决定。

11.2 条件日志

条件日志 LOGx_IF(category, condition) 表示在满足给定的 condition 条件时日志才会输出。使用时仅需在普通日志宏后缀 _IF 即可。

示例如下：

```
LOGF_IF(CORE, IsFatal()) << "fatal error here, abort!!!";
```

该语句中：

- category 为 CORE，表示该条日志的 category 为 CORE。
- condition 为用户函数 IsFatal()。当 IsFatal() 返回 true 时，输出日志信息 “fatal error here, abort!!!”。当 IsFatal() 返回 false 时，不返回日志信息。

11.3 日志过滤

用户可以选择性输出不同 category 的日志和日志等级。使用方式如下：

```
./app --log_filter=category:log_level, category:log_level
```

或者：

```
export CNSTREAM_log_filter=category:log_level, category:log_level
```

其中 category 用于区分日志信息的类别，log_level 为日志级别。CNStream 会返回小于等于该 log_level 的日志。

例如，打印 category 为 SOURCE，日志等级不大于 2 的日志，以及 category 为 CORE，日志等级不大于 3 的日志：


```
./app  --log_filter=SOURCE:2, CORE:3

export CNSTREAM_log_filter=SOURCE:2, CORE:3
```

11.4 生成日志文件

CNStream 默认不保存日志文件。如果要将日志写入文件，可以通过如下方式开启:

```
./app --log_to_file=true  OR export  CNSTREAM_log_to_file=true
```

另外还需要在程序启动时调用 `InitCNStreamLogging(const char* log_dir)` 指定日志文件存放的路径，默认存储在 `/tmp` 目录下。并在结束时调用 `ShutdownCNStreamLogging()`。日志文件名称格式为“cnstream_ 年月日-时分秒. 微秒.log”。

11.5 日志等级

Log 工具基于用户设置的日志等级返回不同级别的日志信息。CNStream 日志等级说明如下表所示：

表 11.1: CNStream 日志等级

日志等级	对应宏	显示日志的级别
0	LOGF	显示 FATAL 级别的日志。
1	LOGE	显示 FATAL 和 ERROR 级别的日志。
2	LOGW	显示 FATAL、ERROR、WARNING 级别的日志。
3	LOGI	显示 FATAL、ERROR、WARNING、INFO 级别的日志。
4	LOGD	显示 FATAL、ERROR、WARNING、INFO、DEBUG 级别的日志.
5	LOGT	显示 FATAL、ERROR、WARNING、INFO、DEBUG、TRACE 级别的日志。
6	LOGA	显示 FATAL、ERROR、WARNING、INFO、DEBUG、TRACE、ALL 级别的日志。

用户可以通过环境变量 `CNSTREAM_min_log_level` 或者命令行参数 `min_log_level` 调整日志输出等级。使用示例如下：


```
./app --min_log_level=0   or export CNSTREAM_min_log_level=0  \\显示 FATAL 级别的日志
```


Cambricon
寒　武　纪

12 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。

12.1 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。

12.2 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 。

12.3 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。

12.3.1 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。

12.3.2 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。

12.4 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)) {
      LOG << "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);
  }

  AffineTransformation(frame_info);

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

  ...

  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。

12.5 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。

12.6 获取性能统计结果

12.6.1 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。

12.6.1.1 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。

12.6.1.2 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。

12.6.2 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 pipelne 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。

12.6.3 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;
  }
}
```


12.6.4 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。

12.6.5 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
}
```

12.7 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 12.1: 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。

12.8 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。

12.8.1 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)  **********
```

12.8.2 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)  ******************
```


13 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 中流转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。

13.1 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" : {
    ...
  }
}
```

13.2 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 13.1: 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 std::make_pair(CNFrameInfo::stream_id, CNFrameInfo::timestamp) 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 process_name 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 cnstream::Clock::now() 获得。
level	可选值为 cnstream::TraceEvent::Level::PIPELINE 及 cnstream::TraceEvent::Level::MODULE。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。详情请查看获取追踪数据 。
type	可选值为 cnstream::TraceEvent::Type::START 及 cnstream::TraceEvent::Type::END 。 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 type 为 cnstream::TraceEvent::Type::START ，否则为 cnstream::TraceEvent::Type::END 。

13.3 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。

13.4 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:

13.4.1 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。

13.4.2 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。

13.5 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 。

13.6 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:

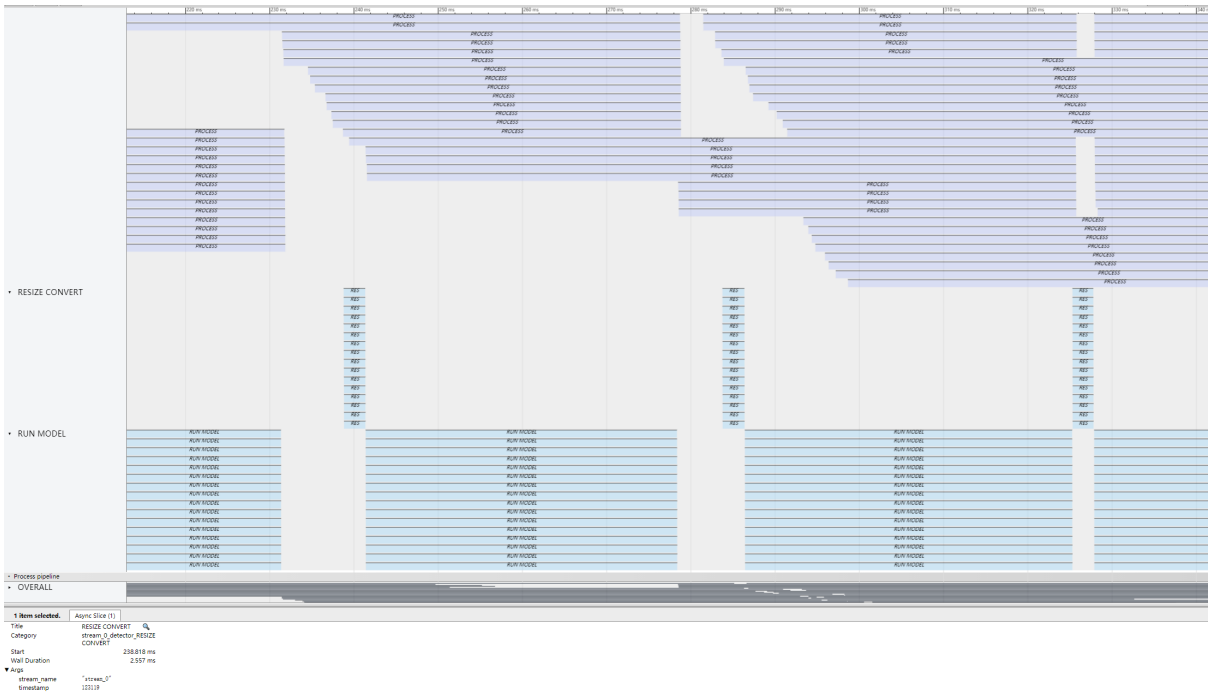

图 13.1: 追踪数据可视化

14 FAQ

14.1 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
...
```


14.2 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 `AddSourceForDecompressedImage` 和 `AddSourceForImageInMem` 内容。

14.3 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。

另外，也可以参考第三方开发人员提供步骤 https://github.com/CambriconKnight/mlu220-cross-compile-docker-image。

14.4 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。
- 对于数据源插件，parallelism 设置为 0 即可, 因为数据源插件的并行度是由输入数据路数决定的。
- 对于 Inference 插件，建议 parallelism 小于硬件核数与离线模型核数之比。
- 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。

14.5 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")
```

14.6 怎么调整 Log 打印等级?

CNStream 默认的 log 输出等级是 INFO，可以通过环境变量 CNSTREAM_min_log_level 或者命令行参数 min_log_level 调整日志输出等级，数字越大输出的 log 内容越多。更多细节可以参阅 CNStream 用户手册中关于Log 工具 的描述。

15 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。

15.1 CNStream 2021-8-10 (Version 6.0.0)

15.1.1 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 Neuware 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。

15.1.2 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。

15.1.3 版本兼容

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

15.1.4 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。

15.2 CNStream 2021-1-28 (Version 5.3.0)

15.2.1 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。

15.2.2 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。

15.2.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 和 EDGE 平台。

15.2.4 版本限制

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

15.3 CNStream 2020-9-18 (Version 5.2.0)

15.3.1 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。

15.3.2 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。

15.3.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 和 EDGE 平台。

15.3.4 版本限制

- 基于寒武纪 Neuware 1.4.0 或者 1.5.0 版本使用 CNStream。

15.4 CNStream 2020-07-10 (Version 5.0.0)

15.4.1 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。

15.4.2 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

15.4.3 版本限制

- 基于寒武纪 Neuware 1.3.0 或者 1.4.0 版本。

15.5 CNStream 2020-05-28 (Version 4.5.0)

15.5.1 新增功能及功能变更

- 支持多进程和单进程使用多个 MLU 卡。详情查看单进程单 Pipeline 中使用多个设备 和多进程操作。
- 新增 rtsp_sink 模块。详情查看RTSP Sink 模块。
- 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 Neuware 包。
- 部分算子更新。
- 修复一些已知问题。

15.5.2 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

15.5.3 版本限制

- 基于寒武纪 Neuware 1.3.0 版本。

15.6 CNStream 2020-04-16 (Version 4.4.0)

15.6.1 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。详情查看多进程操作。
- 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 Neuware。
- 修复汇聚插件随机性卡死、多线程并行推理异常等问题。

15.6.2 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。

15.6.3 版本兼容

- 兼容寒武纪 MLU270、MLU220 M.2 平台。

15.6.4 版本限制

- 基于寒武纪 Neuware 1.2.5 版本。

15.7 CNStream 2019-02-20

15.7.1 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。

15.7.2 版本限制

- 依赖寒武纪 Neuware 1.2.4 运行。

15.8 CNStream 2019-12-31

15.8.1 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。

