寒武纪 CNStream 用户手册

版本 *2020-05-28 (Version 4.5.0)*

2020 年 06 月 11 日

Cambricon 寒武纪

目录

插图目录

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美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
- © 2020 中科寒武纪科技股份有限公司保留一切权利。

2 概述

2.1 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 DataFrame 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 MLU270 和 MLU220 M.2 平台上使用。

2.2 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、前后处理模块、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。高效支持低位宽 (INT8) 和稀疏权值运算。
- 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，在云端和边缘侧集成。

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。
- 由于 TensorFlow、Caffe 等深度学习框架，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。然而，CNStream 内置 CNInfer 模块直接基于寒武纪运行时库（CNRT）设计，能够高效利用底层硬件能力，快速完成组件的开发。
- 内置 Tracker 模块提供了经过寒武纪芯片加速优化后的 KCF 算法，使多路并行比运行在 CPU 上的 Deepsort 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。

2.3 CNStream 应用框架

使用 CNStream SDK 开发一个应用的典型框架如下图：

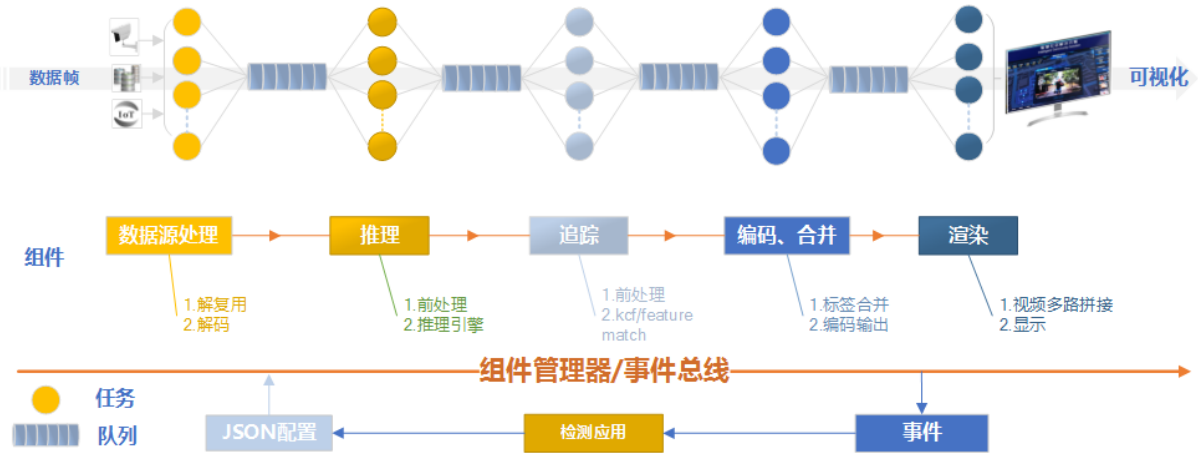

图 2.1: 典型视频结构化场景架构图

3 框架介绍

3.1 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------|
```


3.2 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。

> **注意:**
>
> Pipeline 的 source module 是没有输入队列，pipeline 中不会为 source module 启动线程，也就是说 pipeline 不会调度 source module。source module 通过 pipeline 的 `ProvideData` 接口向下游模块发送数据和启动内部线程。

cnstream::Pipeline 类在 `cnstream_pipeline.hpp` 文件内定义，主要接口如下。`cnstream_pipeline.hpp` 文件存放于 `modules/core/include` 目录下。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《CNStream Developer Guide》。

```
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);
```

(下页继续)

(续上页)

```
  ...
};
```

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

  "inference" : {
    "class_name" : "M220Inference",
    "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。
      "model_path" : "/data/models/resnet34_ssd.cambricon", //支持绝对路径和相对 JSON 文件路径。
      "func_name" : "subnet0",
      "device_id" : 0
    }
  }
}
```

3.3 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。

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 `modules/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《CNStream Developer Guide》。

```
class Module {
 public:

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

  // 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。
  //  1 -- 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) const;
};
```


3.4 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 `modules/core/include` 文件夹下。这个数据结构包括了 CNDataFrame 和 CNFrameInfo。

CNFrameInfo 用于数据和推理结果，并对 pipeline 中单路 stream 使用的 DataFrame 的数目进行限制，我们称之为 pipeline 的并发深度，接口如下：

```
cnstream::SetParallelism(int value);
```

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::shared_ptr<CNSyncedMemory> data[CN_MAX_PLANES];
```

CNDataFrame 中的 SyncedMem 支持 deep copy 或者复用已有的内存。当管理 CNDecoder 和 Inference 之间的 image buffer 时，可以进行 deep copy 和复用 decoder 的 buffer 内存。decoder 和后续的 inference 处理完全解耦，但是会带来 dev2dev copy 的代价。

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);
```

3.5 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 `modules/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《CNStream Developer Guide》。

```
class EventBus {
 public:
```

(下页继续)

(续上页)

```
  // 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, Module *watch_module);
  ......
};
```

3.6 cnstream::Event 类

cnstream::Event 类是模块和 piepline 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，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 `modules/core/include` 文件夹下。接口详情，查看《CNStream Developer Guide》。

4 内置模块

4.1 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 demux 和视频解码。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“Cambricon-CNCodec-Developer-Guide”文档手册。
- 神经网络推理模块：依赖于寒武纪实时运行库（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。
- 追踪模块：使用针对寒武纪平台优化的 FeatureMatch 和 KCF 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、多媒体显示模块和帧率统计模块等。

- OSD（On-Screen Display）模块：支持内容叠加和高亮物件处理。
- 编码模块：支持在 CPU 上编码。
- 多媒体显示模块：支持屏幕上显示视频。

4.2 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。通过 FFmpeg 来解封装，或解复用本地文件或者网络流，来得到码流。之后喂给 CNDecoder 或者 CPU Decoder 进行解码，得到图像，并把图像存到 CNDataFrame 的 syncedMem 中。另外，为了调试方便，该模块还支持读取 H264 和 H265 格式的裸码流文件。

> **注意：**
>
> 一个 pipeline 只支持定义一个数据源模块。

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 启动线程，也不会调度 source。source module 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 source 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 source module 的具体功能，而不是在编译时选择。

cnstream::DataSource 类在 `cnstream_source.hpp` 文件中定义。`cnstream_source.hpp` 文件存放在 `modules/core/include` 文件夹下。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 DataHandler;

class DataSource : public Module, public ModuleCreator<DataSource> {
 public:
  // 动态增加一路 stream 接口。
  int AddVideoSource(const std::string &stream_id, const std::string &filename, int framerate,␣
↪bool loop = false);
  // 动态减少一路 stream 接口。
  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 DataHandler 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<DataHandler>> source_map_;
};
```

4.2.1 使用说明

在 `detection_config.json` 配置文件中进行配置，如下所示。该文件位于 `cnstream/samples/demo` 目录下。

```
"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" : ["detector"],          // 下一个连接模块的名称。
```

(下页继续)

(续上页)

```
  "custom_params" : {                    // 特有参数。
    "source_type" : "ffmpeg",            // source 类型。
    "output_type" : "mlu",               // 输出类型。可设为 MLU 或 CPU。
    "decoder_type" : "mlu",              // decoder 类型。可设为 MLU 或 CPU。
    "reuse_cndec_buf" : "false"          // 是否复用 Codec 的 buffer。
    "device_id" : 0                      // 设备 id，用于标识多卡机器的设备唯一编号。
  }
}
```

其他配置字段可以参考 data_source.hpp 中详细注释。

4.3 神经网络推理模块

神经网络推理（Inferencer）模块是基于寒武纪实时运行库（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。

4.3.1 使用说明

在 detection_config.json 配置文件中进行配置，如下所示。该文件位于 cnstream/samples/demo 目录下。

```
"detector" : {
  "class_name" : "cnstream::Inferencer",    // 推理类名。
  "parallelism" : 32,                       // 并行度。
  "max_input_queue_size" : 20,              // 最大队列深度。
  "next_modules" : ["tracker"],             // 下一个连接模块的名称。
  "custom_params" : {                       // 特有参数 。
    // 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/MLU100/Primary_Detector/resnet34ssd/resnet34_ssd.cambricon",
    // 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",
    // 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",
    //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" : "PostprocSsd",
```

(下页继续)

(续上页)

```
    // 多 batch 推理支持。用于提高单位时间内吞吐量。该参数仅支持 MLU100。MLU100 生成离线时设置
batchsize 为 1。通过指定 batchsize 参数，来进行多 batch 推理。使用 MLU270 进行多 batch 推理时，需要
在生成离线模型时指定 batchsize。
    "batchsize" : 1,
    // 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" : 30,
    "device_id" : 0    // 设备 id，用于标识多卡机器的设备唯一编号。
  }
}
```

4.4 追踪模块

追踪（Tracker）模块用于对检测到的物体进行追踪并输出检查结果。主要应用于车辆等检测和追踪。目前支持 FeatureMatch 和 KCF 两种追踪方法。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。

4.4.1 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”,       // Track 的类名。
“parallelism” : 4,                        // 并行度。
“max_input_queue_size” : 20,              // 数据输入队列长度。
“next_modules” : [“osd”],                 // 下一个连接的模块名。
“custom_params” : {
   // 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”,    // 模型函数名。
    “track_name” : “KCF”        // 追踪方法。支持 FeatureMatch 和 KCF 两种追踪方法。
   }
}
```


4.5 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 4.1: RTSP Sink 模块数据处理流程

4.5.1 使用依赖

RTSP Sink 模块依赖于 Live555。用户需要先安装 Live555 后，才可以使用该模块。

在 CNStream github 仓库 `tools` 目录下，运行 `download_live.sh` 和 `build_live555.sh` 脚本，即可下载和安装 live555。

4.5.2 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：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：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：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：设备号。仅在使用 MLU 时生效。默认使用设备 0。
- view_mode：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：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：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：UDP 端口。格式为：
  `url=rtsp://本机 ip:9554/rtsp_live`。
  运行示例代码，URL 将保存在文件 `RTSP_url_names.txt` 中。默认值为 9554。
- http_port：RTSP-over-HTTP 隧道端口。默认值为 8080。
- dst_width：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：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：编码视频帧率。取值为大于 0。默认值为 25。
- kbit_rate：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：GOP（Group of Pictures），两个 I 帧之间的帧数。取值为大于 0。默认值为 30。

4.5.3 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",
```

(下页继续)

(续上页)

```
      "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
    }
  }
```

4.5.4 示例代码

CNStream 提供两个示例，位于 CNStream github 仓库 `samples/demo` 目录下：

- run_rtsp.sh：示例使用 single 模式。对应配置文件 `RTSP.json`。
- run_rtsp_mosaic.sh：示例使用 mosaic 模式。对应配置文件 `RTSP_mosaic.json`。

运行示例代码

执行下面步骤运行示例代码：

1. 运行 run_rtsp.sh 或 run_rtsp_mosaic.sh 脚本。
2. 使用 VLC Media Player 打开生成的 URL。例如：`rtsp://本机 ip:9554/rtsp_live`。URL 保存在 `samples/demo` 目录下的 `RTSP_url_names.txt` 文件中。

4.6 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 Decode 和 Inference 模块使用场景为例，配置 Decode 模块使用 MLU 卡 0，Inference 模块使用 MLU 卡 1。单进程中一般建议在 source module 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" : ["ipc"],                  // 下一个连接模块名称。
    "custom_params" : {                        // 特有参数。
      "source_type" : "ffmpeg",                // source 类型。
      "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。
}
  }
}
```

了解如何在多进程、单个 pipeline 下使用多个设备，查看多进程。

4.7 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。

4.7.1 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 1。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。
```

(下页继续)

(续上页)

```
    "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。
```

(下页继续)

(续上页)

```
    }
  },

  "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 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。

5.1 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。

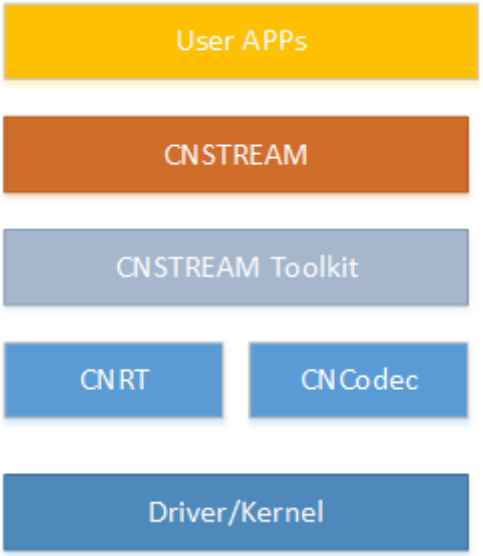

图 5.1: 寒武纪软件栈框图

5.2 文件目录

在 CNStream 源码目录下，主要由以下部分组成：

- 头文件：CNStream 头文件存放在 `modules` 文件夹下，包含所有的数据类型和接口。
- 动态库：编译成功后，CNStream 动态库存放在 `libcnstream.so` 文件中，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。

CNStream 文件位于 CNStream 仓库下，目录结构如下图所示：

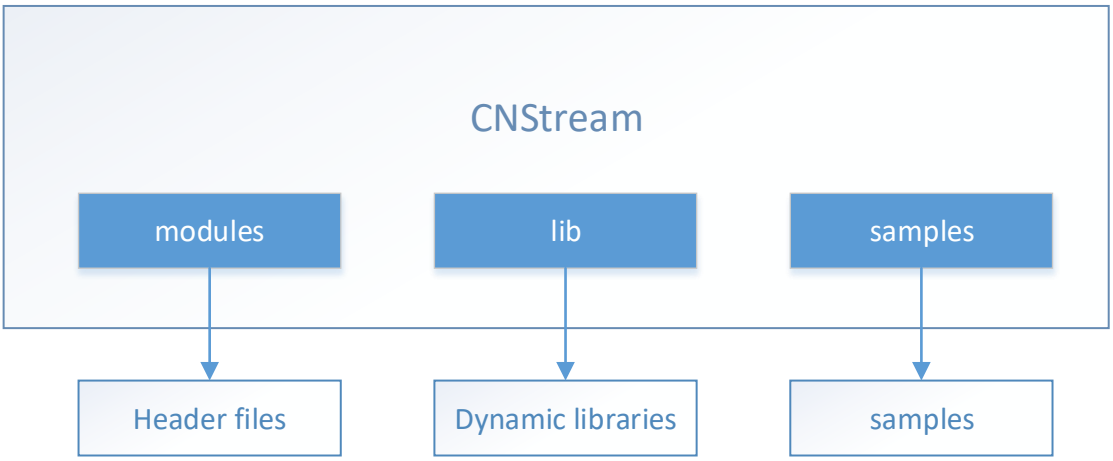

图 5.2: CNStream 文件目录结构图

5.3 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 注册多个预置模块或者自定义的模块。这些模块之间通过隐含的深度可控的队列连接，使用一个 JSON 的文本描述组件间的连接关系。

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，并设置到 pipeline。
4. 启动 pipeline。
5. 动态增加数据源。

更多详情，请查看 `Readme`。

6 环境配置

本章介绍了如何在 Debian、Ubuntu、CentOS 以及 Docker 环境下配置 CNStream。

6.1 依赖库

6.1.1 环境依赖

CNStream 有以下环境依赖。

- OpenCV2.4.9+
- GFlags2.1.2
- GLog0.3.4
- Cmake2.8.7+
- Live555
- SDL22.0.4+
- SQLite3

6.1.2 寒武纪安装包

CNStream 的使用依赖于寒武纪 Neuware 安装包中 CNRT 库和 CNCodec 库。Neuware 安装包是寒武纪公司发布的基于寒武纪硬件产品的神经网络开发工具包。用户需要在使用 CNStream 之前安装寒武纪 Neuware 安装包。发送邮件到 service@cambricon.com 联系寒武纪工程师获得 Neuware 安装包和安装指南。

6.2 Debian 或 Ubuntu 环境配置

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 Neuware 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
sudo apt-get install libfreetype6 ttf-wqy-zenhei libsdl2-dev lcov libsqlite3-dev sqlite3
cd ${CNSTREAM_DIR}/tools
./download_live.sh
```

4. 编译 CNStream。CNStream 使用 CMake 编译系统进行编译。
   - 针对 MLU270 平台：

   ```
   mkdir -p build; cd build
   cmake ${CNSTREAM_DIR} -DMLU=MLU270
   make
   ```

   - 针对 MLU220 SOC 平台：

   ```
   mkdir -p build; cd build
   cmake ${CNSTREAM_DIR} -DMLU=MLU220_SOC
   make
   ```

5. 运行示例程序。

```
cd ${CNSTREAM_DIR}/samples/demo
./run.sh
```

6.3 CentOS 环境配置

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 Neuware 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 lcov sqlite-devel
sudo yum install ffmpeg ffmpeg-devel
cd ${CNSTREAM_DIR}/tools
./download_live.sh
```

4. 编译 CNStream。CNStream 使用 CMake 编译系统进行编译。${CNSTREAM_DIR} 代表 CNStream 源码目录。
   - 针对 MLU270 平台：

```
mkdir -p build; cd build
cmake ${CNSTREAM_DIR} -DMLU=MLU270
make
```

   - 针对 MLU220 SOC 平台：

```
mkdir -p build; cd build
cmake ${CNSTREAM_DIR} -DMLU=MLU220_SOC
make
```

5. 运行示例程序。

```
cd ${CNSTREAM_DIR}/samples/demo
./run.sh
```

6.4 Docker 环境配置

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. 运行下面命令制作 Docker 镜像。
   其中 “${neuware_package}“ 为寒武纪 Neuware 安装包及其存放路径。${board_series} 为用户使用板卡的型号，即 MLU270 或 MLU220SOC。

```
git clone https://github.com/Cambricon/CNStream.git
cp ${neuware_package} CNStream   #copy your neuware package into CNStream
docker build -f Dockerfile --build-arg mlu_platform=${board_series} --build-arg neuware_
↪package=${neuware_package_name} -t ubuntu_cnstream:v1 .
```

   CNStream 提供以下 Dockerfile：

```
docker/Dockerfiler.16.04
```

(下页继续)

(续上页)

```
docker/Dockerfiler.18.04
docker/Dockerfiler.CentOS
```

3. 运行示例程序。

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=$DISPLAY --privileged -v /dev:/dev --
↪net=host --ipc=host --pid=host -v $HOME/.Xauthority -it --name container_name  -v $PWD:/
↪workspace ubuntu_cnstream:v1
./run.sh
```


Cambricon 寒武纪

7 创建自定义模块

7.1 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，用户只需根据 samples/example 目录下的 example.cpp 文件里给出的例子，即可方便快捷地实现自定义模块的轮廓。自定义模块需要多重继承 **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。

7.2 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 {
    std::cout << this->GetName() << " Open called" << std::endl;
    for (auto &v : paramSet) {
      std::cout << "\t" << v.first << " : " << v.second << std::endl;
    }
    return true;
  }
  void Close() override { std::cout << this->GetName() << " Close called" << std::endl; }
  int Process(std::shared_ptr<cnstream::CNFrameInfo> data) override {
    // do something ...
    std::unique_lock<std::mutex> lock(print_mutex);
    std::cout << this->GetName() << " process: " << data->frame.stream_id << "--" << data->
↪frame.frame_id;
    std::cout << " : " << std::this_thread::get_id() << std::endl;
```

(下页继续)

(续上页)

```
    /*continue by the framework*/
    return 0;
  }

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

7.3 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::Module, public cnstream::ModuleCreator
↪<ExampleModuleSource> {
  using super = cnstream::Module;

 public:
  explicit ExampleModuleSource(const std::string &name) : super(name) {}
  bool Open(cnstream::ModuleParamSet paramSet) override {
    std::cout << this->GetName() << " Open called" << std::endl;
    for (auto &v : paramSet) {
      std::cout << "\t" << v.first << " : " << v.second << std::endl;
    }
    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;
};
```


7.4 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 {
    std::cout << this->GetName() << " Open called" << std::endl;
    for (auto &v : paramSet) {
      std::cout << "\t" << v.first << " : " << v.second << std::endl;
    }
    running_.store(1);
    threads_.push_back(std::thread(&ExampleModuleEx::BackgroundProcess, this));
    return true;
  }
  void Close() override {
    running_.store(0);
    for (auto &thread : threads_) {
      thread.join();
    }
    std::cout << this->GetName() << " Close called" << std::endl;
  }
  int Process(FrameInfoPtr data) override {
    {
      std::unique_lock<std::mutex> lock(print_mutex);
      if (data->frame.flags & cnstream::CN_FRAME_FLAG_EOS) {
        std::cout << this->GetName() << " process: " << data->frame.stream_id << "--EOS";
      } else {
        std::cout << this->GetName() << " process: " << data->frame.stream_id << "--" << data->
↪frame.frame_id;
      }
      std::cout << " : " << std::this_thread::get_id() << std::endl;
    }
    // handle data in background threads...
```

(下页继续)

(续上页)

```
    q_.enqueue(data);

    /*notify that data handle by the module*/
    return 1;
  }

 private:
  void BackgroundProcess() {
    /*NOTE: EOS data has no invalid context,
     *    All data recevied including EOS must be forwarded.
     */
    std::vector<FrameInfoPtr> eos_datas;
    std::vector<FrameInfoPtr> datas;
    FrameInfoPtr data;
    while (running_.load()) {
      bool value = q_.wait_dequeue_timed(data, 1000 * 100);
      if (!value) continue;

      /*gather data*/
      if (!(data->frame.flags & cnstream::CN_FRAME_FLAG_EOS)) {
        datas.push_back(data);
      } else {
        eos_datas.push_back(data);
      }

      if (datas.size() == 4 || (data->frame.flags & cnstream::CN_FRAME_FLAG_EOS)) {
        /*process data...and then forward
         */
        for (auto &v : datas) {
          this->container_->ProvideData(this, v);
          std::unique_lock<std::mutex> lock(print_mutex);
          std::cout << this->GetName() << " forward: " << v->frame.stream_id << "--" << v->
↪frame.frame_id;
          std::cout << " : " << std::this_thread::get_id() << std::endl;
        }
        datas.clear();
      }

      /*forward EOS*/
      for (auto &v : eos_datas) {
        this->container_->ProvideData(this, v);
```

(下页继续)

(续上页)

```
        std::unique_lock<std::mutex> lock(print_mutex);
        std::cout << this->GetName() << " forward: " << v->frame.stream_id << "--EOS ";
        std::cout << " : " << std::this_thread::get_id() << std::endl;
      }
      eos_datas.clear();
    }  // while
  }

 private:
  moodycamel::BlockingConcurrentQueue<FrameInfoPtr> q_;
  std::vector<std::thread> threads_;
  std::atomic<int> running_{0};

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


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 自有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 提供的自有模块 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：


```
{
"source" : {
  // 数据源模块。设置使用 ffmpeg 进行 demux，使用 MUL 解码，不单独启动线程。
  "class_name" : "cnstream::DataSource",
  "parallelism" : 0,
  "next_modules" : ["detector"],
  "show_perf_info" : true,
  "custom_params" : {
    "source_type" : "ffmpeg",
    "output_type" : "mlu",
    "decoder_type" : "mlu",
    "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 4，模块输入队列的 max_size 为 20。
  "class_name" : "cnstream::Inferencer",
  "parallelism" : 4,
  "max_input_queue_size" : 20,
  "next_modules" : ["tracker"],
  "show_perf_info" : true,
  "custom_params" : {
    "model_path" : "../data/models/resnet34ssd/resnet34_ssd.cambricon",
    "func_name" : "subnet0",
    "postproc_name" : "PostprocSsd",
    "device_id" : 0
  }
},

"tracker" : {
  // Tracker 模块。设置使用 track 离线模型，并行度为 4，模块输入队列的 max_size 为 20。
   "class_name" : "cnstream::Tracker",
  "parallelism" : 4,
  "max_input_queue_size" : 20,
  "next_modules" : ["osd"],
  "show_perf_info" : true,
  "custom_params" : {
    "model_path" : "../data/models/Track/track.cambricon",
    "func_name" : "subnet0"
```

(下页继续)

(续上页)

```
    }
  },

  "osd" : {
    // Osd 模块。配置解析 label 路径，设置并行度为 4，模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::Osd",
    "parallelism" : 4,
    "max_input_queue_size" : 20,
    "next_modules" : ["encoder"],
    "show_perf_info" : true,
    "custom_params" : {
      "chinese_label_flag" : "false",
      "label_path" : "../data/models/resnet34ssd/label_voc.txt"
    }
  },

  "encoder" : {
    // Encoder 模块。配置输出视频的 dump 路径，设置并行度为 4，模块输入队列的 max_size 为 20。
    "class_name" : "cnstream::Encoder",
    "parallelism" : 4,
    "max_input_queue_size" : 20,
    "show_perf_info" : true,
    "custom_params" : {
      "dump_dir" : "output"
    }
  }
}
```

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。
3. 调用 `pipeline.SetStreamMsgObserver`，设置事件监听处理机制。
4. 调用 `pipeline.CreatePerfManager`，创建性能统计管理器。
5. 调用 `pipeline.Start()`，启动 pipeline。
6. 调用 `pipeline.AddVideoSource()` 或 `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/demo/demo.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/example/example.cpp`。

9 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 。

9.1 工具命令的使用

CNStream 环境配置 完成后，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。

9.1.1 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
```


9.1.2 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.
  ...                                      ...
```

9.1.3 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.
...                           ...
```

9.1.4 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/demo/detection_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 DataSourc：

```
{
  "source" : {
    "class_name" : "cnstream::DataSourc",
```

(下页继续)

(续上页)

```
    ...
  },
}
```

命令返回示例如下：

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

9.1.5 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: v4.0.0
```

9.2 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;
```

(下页继续)

(续上页)

```
  }

  // 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;
}
```

10 性能统计

CNStream 提供性能统计机制，帮助用户统计各模块及整条 pipeline 的性能，其中包括时延及吞吐量等。用户也可以自定义想要统计的信息，详情查看注册信息类型 。

性能统计机制的实现主要在 **PerfManager** 类中定义。**PerfManager** 类实现了自身初始化、记录信息、注册信息类型、计算模块性能和计算 pipeline 性能等功能。类的声明在 `modules/core/include/perf_manager.hpp` 文件的源码中。此外，初始化时会创建数据库和实例化 **PerfCalculator** 类。**PerfCalculator** 类主要用于性能计算。

> **注意:**
>
> 统计性能依赖于 pts，需要保证视频流中每帧的 pts 的唯一性，否则 CNStream 不能保证提供信息的准确性。

10.1 实现机制

性能统计的实现机制如下图所示：

图 10.1: 性能统计实现机制

每个数据流都需要创建一个 PerfManager 进行性能统计。初始化 PerfManager 后，数据库文件将被创建。每一帧数据将会被记录下来，并保存到创建的数据库中。基于数据库中的数据，计算相应各模块和 pipeline 的性能数据。

10.1.1 数据库文件

CNStream 使用 SQLite3 数据库保存用户想要计算的性能数据。

用户在初始化 PerfManager 时，可以通过 `Init` 函数传入数据库文件名以及数据库中的字段名称。再通过调用 `Record` 函数，基于数据库中的字段，将相关数据记录到数据库中。最后 CNStream 调用该数据库的数据对模块和 pipeline 进行性能计算。

10.1.2 初始化 PerfManager

根据 perf 类型的不同，初始化 PerfManager 的方法也会有所不同。主要分为：

- 使用 CNStream 预定义的 perf 类型 `PROCESS` 做性能统计。
- 使用自定义的 perf 类型做性能统计。

使用 CNStream 预定义的 perf 类型

寒武纪 CNStream 提供 `PROCESS` perf 类型对模块及 pipeline 的性能做统计。用户需要调用 `Init` 函数传入数据库文件名（包括数据库文件的所在路径）、pipeline 中所有节点（模块）、开始节点和所有结束节点的名字，初始化 PerfManager。

初始化 PerfManager 后，CNStream 将以 `PROCESS` 作为表格名生成数据库表格。使用 CNStream 预定义的 `PROCESS` 类型生成的数据库表格，主索引为 pts，其他索引为节点名字加 `_stime` 和 `_etime` 后缀，分别代表开始和结束时间，开始节点索引始终紧随主索引 pts。

用户也可以自定义想要统计的其他方面的信息，详情查看注册信息类型 。

以下面 pipeline 为例：

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

PerfManager 初始化调用 `Init` 函数，示例如下：

```
PerfManager perf_manager;

// 初始化数据库名字、所有节点名字、开始节点、所有结束节点。
perf_manager.Init("db_name.db", {"ModuleA", "ModuleB", "ModuleC"}, ModuleA, {ModuleC});
```

生成数据库表格如下：

```
TABLE PROCESS

  pts      ModuleA_stime    ModuleA_etime    ModuleB_stime    ModuleB_etime    ModuleC_stime    ␣
↪ModuleC_etime
 -------  -------------    -------------    -------------    -------------    -------------    ---
↪----------
```

(下页继续)

(续上页)

使用自定义的 perf 类型

如果想要使用自定义的 perf 类型，需要创建数据库文件，并连接数据库。使用此方法，用户需要调用 Init 函数传入数据库文件名（包括数据库文件的所在路径），并调用 RegisterPerfType 函数自定义 per 类型。

调用 Init 函数初始化 PerfManager，示例如下：

```
PerfManager perf_manager;


// 初始化数据库名字
perf_manager.Init("db_name.db");
```

随后可以通过 RegisterPerfType 函数创建 perf 类型，并生成表格。以记录模块 ModuleA 的开始和结束时间为例，依次传入 perf 类型、主索引和其他索引（即 ModuleA 的开始时间和结束时间）。

```
perf_manager.RegisterPerfType("TEST", "pts", {"ModuleA_stime", "ModuleA_etime"})
```

生成表格如下：

```
TABLE TEST


  pts      ModuleA_stime    ModuleA_etime
 -------  -------------    -------------
```

10.1.3 记录相关数据

每帧数据在流过每个模块时相关数据都会分别被记录下来，并储存到数据库中。用户可以通过调用 Record 函数实现。根据用户需要，有以下几种方式：

- 仅记录模块的开始和结束时间。
- 记录模块的其他信息的时间戳。
- 记录模块除时间戳外的其他信息。

仅记录模块的开始和结束时间

使用这种方法，用户需调用 Record 函数来依次传入参数：是否为结束帧、perf 类型、模块名字以及 pts。

例如：记录 pts 为 100 的一帧数据进入 ModuleA 模块的开始时间戳，perf 类型是 PROCESS。

```
// 初始化数据库名字、所有节点名字、开始节点、所有结束节点。
PerfManager perf_manager;
```

(下页继续)

(续上页)

```
perf_manager.Init("db_name.db", {"ModuleA", "ModuleB", "ModuleC"}, ModuleA, {ModuleC});

//记录新信息。
perf_manager.Record(false, "PROCESS", "ModuleA", 100);
```

在数据库中记录情况如下，其中 xxxx 代表当前时间的时间戳。

```
TABLE PROCESS

  pts      ModuleA_stime    ModuleA_etime    ModuleB_stime    ModuleB_etime    ModuleC_stime    ␣
↪ModuleC_etime
 -------  --------------   --------------   --------------   --------------   --------------    ---
↪----------
  100       xxxx
```

随后，记录 pts 为 100 的一帧数据 ModuleA 模块的结束时间戳，perf 类型是 PROCESS。

```
perf_manager.Record(true, "PROCESS", "ModuleA", 100);
```

在数据库中记录情况如下：

```
TABLE PROCESS

  pts      ModuleA_stime    ModuleA_etime    ModuleB_stime    ModuleB_etime    ModuleC_stime    ␣
↪ModuleC_etime
 -------  --------------   --------------   --------------   --------------   --------------    ---
↪----------
  100       xxxx             xxxx
```

记录模块的其他信息的时间戳

使用这种方法，用来记录其他信息的时间戳。用户需调用 Record 函数来依次传入参数：perf 类型、主索引、主索引值、索引。

例如：某一帧的一个 log 信息的时间戳，记录 perf 类型是 LOG，主索引为 pts，其值 100，索引为 ModuleA_log。

```
// 初始化，注册 perf type LOG，主索引 pts，其他索引 ModuleA_log
PerfManager perf_manager;
perf_manager.Init("db_name.db");
perf_manager.RegisterPerfType("LOG", "pts", {"ModuleA_log"});
```

(下页继续)

(续上页)

```
// 记录信息
perf_manager.Record("LOG", "pts", "100", "ModuleA_log");
```

在数据库中记录情况如下：

```
TABLE LOG


  pts      ModuleA_log
 -------  -------------
  100       xxxx
```

记录模块除时间戳外的其他信息

使用这种方法，用来记录其他信息，不仅仅是当前时间的时间戳。用户需调用 Record 函数来依次传入参数：perf 类型、主索引、主索引值、索引、索引值。

例如：某一帧的 frame id 信息。记录 perf 类型是 INFO，主索引为 pts，其值 1000，索引为 frame_id，其值为 300。

```
// 初始化，注册 perf type INFO，主索引 pts，其他索引 frame_id
PerfManager perf_manager;
perf_manager.Init("db_name.db");
perf_manager.RegisterPerfType("INFO", "pts", {"frame_id"});

// 记录信息
perf_manager.Record("INFO", "pts", "1000", "frame_id",  "300");
```

在数据库中记录情况如下：

```
TABLE INFO


  pts      frame_id
 -------  ------------
  1000       300
```


10.1.4 计算模块和 Pipeline 的性能

每隔一段时间各模块及整条 pipeline 的性能就会被统计一次。性能指标主要包括时延和吞吐量。

10.1.4.1 模块的性能计算

每帧的时延是模块处理该帧的时间。性能统计时，我们将计算所有帧的平均时延，最大时延以及吞吐量。吞吐量是平均时延的倒数。通过调用 `CalculatePerfStats` 函数实现。例如：

```
PerfStats statsA = perf_manager.CalculatePerfStats("PROCESS", ModuleA);
PerfStats statsB = perf_manager.CalculatePerfStats("PROCESS", ModuleB);
PerfStats statsC = perf_manager.CalculatePerfStats("PROCESS", ModuleC);
```

如需打印模块性能信息，可以调用 **PerfCalculator** 类的 `PrintPerfStats` 函数实现。详情参见 `modules/core/include/perf_calculator.hpp` 文件。

```
PrintPerfStats(statsA);
PrintPerfStats(statsB);
PrintPerfStats(statsC);
```

10.1.4.2 Pipeline 的性能计算

每帧的时延是该帧走完整个 pipeline 的时间。如果 pipeline 有多个结束节点，则对于每个结束节点都有一组统计信息包括平均时延、最大时延和吞吐量。

吞吐量计算公式如下：

```
throughput = frame count / (结束节点时间戳最大值 - 开始节点时间戳最小值)
```

用户可以通过调用以下函数实现：

```
std::vector<PerfStats> stats = perf_manager.CalculatePipelinePerfStats("PROCESS");
```

如需打印模块性能信息，可以调用 **PerfCalculator** 类的 `PrintPerfStats` 函数实现，详情见 `modules/core/include/perf_calculator.hpp` 文件。

```
for (auto it : stats) {
  PrintPerfStats(it);
}
```

除此之外，如果只打印时延或吞吐量信息，用户可以调用 **PerfCalculator** 类的 `PrintLatency` 或 `PrintThroughput` 函数来实现。

10.2 开发样例介绍

用户可以直接使用 CNStream 提供的开发样例，无需修改任何设置，即可快速体验模块和 pipleline 的性能统计功能。

10.2.1 示例脚本说明

用户通过运行 `run.sh` 示例脚本来运行示例。示例位于 `${CNSTREAM_PATH}/samples/demo` 目录下，其中 `${CNSTREAM_DIR}` 是指 CNStream 源码目录。

数据库文件默认保存到 `perf_database` 文件夹下。如果希望更改生成的数据库文件的储存路径，只需设置示例脚本中的参数 `perf_db_dir` 即可。此外，CNStream 提供的示例默认开启性能统计功能。如需关闭，可在脚本中设置 `perf` 参数为 **false**。

```
./../bin/demo  \

    ...

    --config_fname "detection_config.json" \

    ...

    --perf=false   \            # 关闭性能统计功能，默认开启。
    --perf_db_dir="db_dir"      # 设置数据库文件保存路径到执行目录下的 db_dir 文件夹下，默认保存到 perf_database 文件夹下。
```

10.2.2 配置文件说明

示例脚本 `run.sh` 对应的 JSON 配置文件 `detection_config.json` 位于 `${CNSTREAM_PATH}/samples/demo` 目录下，其中 `${CNSTREAM_DIR}` 是指 CNStream 源码目录。模块参数 `show_perf_info` 表示是否显示模块性能。设为 **true** 时将显示该模块的性能，设为 **false** 时则不显示该模块的性能。

例如显示 source 模块的性能数据，JSON 配置文件配置如下：

```
{
  "source" : {
    // 数据源模块。设置使用 ffmpeg 进行 demux，使用 MUL 解码，不单独启动线程。
    "class_name" : "cnstream::DataSource",

    ...
```

(下页继续)

(续上页)

```
    "show_perf_info" : true,   //显示数据源模块的性能。
    "custom_params" : {
      ...
    }
  },

  ...
}
```

10.3 对自定义构建 pipeline 的性能统计

用户需要按照编程指南 的步骤构建 pipeline。但在动态增加数据源之前，需要调用 `CreatePerfManager` 函数创建 PerfManager，并在函数中传入所有数据流的唯一标识 `stream_id` 和希望保存数据库文件的路径。

创建 PerfManager 源代码示例如下，详情可参考 `samples/demo/demo.cpp` 文件的 CNStream 源码。

```
/*
   创建 perf manager。
*/
if (FLAGS_perf) {
  std::vector<std::string> stream_ids;
  for (int i = 0; i < static_cast<int>(video_urls.size()); i++) {
    stream_ids.push_back(std::to_string(i));
  }
  // 创建 PerfManager。
  pipeline.CreatePerfManager(stream_ids, FLAGS_perf_db_dir);  // 传入 stream_id 和数据库文件储存路径。
}
```

注意:

用户需要在 pipeline 开始之前，调用 `CreatePerfManager` 函数。

10.4 自定义性能统计

除了统计模块及整条 pipeline 的性能，用户也可以对其他方面的信息进行统计，如所有模块 open 的时间等。本节介绍了如何自定义性能统计的信息以及自定义模块如何统计性能。

10.4.1 自定义性能统计信息

如果想要对其他方面信息进行统计，用户需要调用 `RegisterPerfType` 函数注册一个 perf 类型。随后可通过调用 `Record` 函数记录信息。

例如，注册 TEST1 类型和 TEST2 类型。

```
PerfManager perf_manager;

// 初始化 PerfManager。
perf_manager.Init("db_nam.db", {"ModuleA", "ModuleB", "ModuleC"}, ModuleA, {ModuleC});

// 注册 TEST1 类型。
perf_manager.RegisterPerfType("TEST1");

// 注册 TEST2 类型。
perf_manager.RegisterPerfType("TEST2");

int64_t pts = 1;
perf_manager.Record(false, "TEST1", "ModuleA", pts);
perf_manager.Record(false, "TEST2", "ModuleB", pts);
```

10.4.2 自定义计时

用户需要在模块基类中声明如下变量来实现自定义计时。调用 `CreatePerfManager` 函数后，其他模块即可访问到各视频流的 PerfManager。

```
// 每个视频流的 PerfManager，key 为 stream_id。
std::unordered_map<std::string, std::shared_ptr<PerfManager>> perf_managers_;
```


10.4.3 自定义模块设置

如果不在 pipeline 中调用自定义模块的 `Process` 和 `TransmitData` 函数，则用户需要在模块的 Process 开始处记录开始时间戳，处理完毕后记录结束时间戳。

11 FAQ

11.1 file_list 文件是做什么用的?

文本文件 file_list 的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 cnstream/samples/demo 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，内容如下:

```
// 最外层 file list 内容如下:
/path/of/%d.jpg
/path/of/%d.jpg
...
```

(下页继续)

(续上页)

```
...
...
```

12 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。

12.1 2020-05-28 (Version 4.5.0)

12.1.1 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。

12.1.2 版本兼容

- 兼容寒武纪 M220 M.2 平台。

12.1.3 版本限制

- 基于寒武纪 Neuware 1.3.0 版本。

12.2 2020-04-16 (Version 4.4.0)

12.2.1 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。

12.2.2 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。

12.2.3 版本兼容

- 兼容寒武纪 M220 M.2 平台。

12.2.4 版本限制

- 基于寒武纪 Neuware 1.2.5 版本。

12.3 CNStream 2019-02-20

12.3.1 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。

12.3.2 版本限制

- 依赖寒武纪 Neuware 1.2.4 运行。

12.4 CNStream 2019-12-31

12.4.1 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。

