

Cambricon Neuware Stream Processor

Developer Guide

寒武纪流处理开发者手册

2018.03.11

版权声明

本指引以及所有本指引涉及的任何寒武纪公司提供的文档中所包括的信息,均是”按现状”提供。本指引提供的信息仅供说明，不做任何明示、暗示、法定或其他任何方面的保证，包括并不限于适销性、所有权、知识产权或者适合特定用途的保证。对于用户可能因为任何原因导致的任何损害，本指引中所涉及的寒武纪公司产品的累积赔偿责任应根据该产品的销售条款和销售条件进行限制。

在任何情况下,对于使用本指引或者无法使用本指引造成的任何直接的、继发的、偶发的损失(包括但不限于商业利益的损失、人身损害、信息丢失等)，寒武纪公司及本指引的提供者均不承担任何责任，即使寒武纪公司已经知晓存在发生上述损失的可能。

寒武纪公司以及本指引的提供者不对本指引中包含的信息、文本、图形、链接或者其他内容的完整性、正确性负责。请用户审慎使用，寒武纪公司保留随时更新本指引中的内容的权利，并恕不承担通知义务。

本指引中涉及的性能测试和额定值测试是利用了特定芯片系统和/或组件，并在特定计算机系统和/组件的环境下完成的，反映了大致性能，系统硬件、软件设计或配置的任何不同都会影响实际性能。对于本指引中所涉及的寒武纪产品无需进一步测试或修改即可适用于特定用途，寒武纪公司不做任何保证。对于本指引所涉及的每种产品的所有参数不一定全部由寒武纪公司执行测试，用户有责任确保该产品适用于用户计划的应用程序中，并进行必要的测试。

鉴于用户产品设计中的缺陷可能会影响寒武纪公司产品的质量及可靠性，并可能导致其需求或条件超出 和/或 异于 和/或 不满足本指引的需求或条件。因此，寒武纪公司不承担可能基于或归因于以下原因的任何违约责任、损害赔偿责任或其他任何责任：（i）以违反本指引的任何方式使用寒武纪公司的产品，或（ii）用户的产品设计。

本指引在全球范围内，受到版权法和条约所规定的保护。未经寒武纪公司事先的书面许可，任何集体或个人不得以任何方式复制、修改、发布、传输或分发。除本指引明确规定外，寒武纪公司及其供应商不会就任何专利、版权、商标、商业秘密或其他任何知识产权或所有权授予任何明示或暗示的权利。除了客户使用本指引中信息的权利之外，寒武纪公司未通过本指引授予其他任何明示或暗示的许可。

寒武纪和寒武纪徽标是寒武纪公司在美国和其他国家的商标和/或注册商标。其他公司或产品名称可能是各相关公司的商标和/或注册商标。

Copyright © 2018 Cambricon Corporation. All rights reserved.

版本记录

|  |  |  |  |
| --- | --- | --- | --- |
| **文档名称** | | 寒武纪流处理库开发者手册 | |
| **版本号** | | V3.4.0 | |
| **作者** | | Cambricon | |
| **创建日期** | | 2018.07.02 | |
| 更新历史 | | | |
| Order | Date | Version | Note |
| 0 | 2018.09.15 | V3.0.0 | 初始版本，支持P2P模式 |
| 1 | 2018.10.15 | V3.1.1 | 支持RTSP输入，用SDL2代替OpenCV显示 |
| 2 | 2018.11.15 | V3.2.0 | 支持处理超过32路数据流 |
| 3 | 2018.1.10 | V3.3.0 | 增加非P2P的解码和推理模块，增加batch\_size参数代替shape.n表示batch\_size |
| 4 | 2019.02.17 | V3.4.0 | 增加完整视频结构化示例，支持双码流，支持切换数据源，显示模块支持丢帧，支持RTMP输入 |

服务联系: service@cambricon.com

[www.cambricon.com](http://www.cambricon.com)

目录

[第1章 简介 7](#_Toc884482)

[关于P2P： 8](#_Toc884483)

[第2章 CNSTREAM编程模型 9](#_Toc884484)

[2.1. 依赖库 9](#_Toc884485)

[2.2. 文件目录 9](#_Toc884486)

[2.3. 编程模型 9](#_Toc884487)

[第3章 公用数据类型 11](#_Toc884488)

[3.1. Shape 11](#_Toc884489)

[3.1.1. 构造函数(0) 11](#_Toc884490)

[3.1.2. n setter 11](#_Toc884491)

[3.1.3. c setter 11](#_Toc884492)

[3.1.4. h setter 12](#_Toc884493)

[3.1.5. w setter 12](#_Toc884494)

[3.1.6. n getter 12](#_Toc884495)

[3.1.7. c getter 12](#_Toc884496)

[3.1.8. h getter 13](#_Toc884497)

[3.1.9. w getter 13](#_Toc884498)

[3.1.10. nhwc 13](#_Toc884499)

[3.1.11. operator<< 13](#_Toc884500)

[3.1.12.operator== 14](#_Toc884501)

[3.1.13.operator!= 14](#_Toc884502)

[3.1.14.hwc 14](#_Toc884503)

[3.1.15.hw 14](#_Toc884504)

[3.2. TensorType 14](#_Toc884505)

[3.3 TensorDesc 15](#_Toc884506)

[3.3.1. 构造函数(0) 15](#_Toc884507)

[3.3.2. 构造函数(3) 15](#_Toc884508)

[3.3.3. set\_shape 15](#_Toc884509)

[3.3.4. set\_type 16](#_Toc884510)

[3.3.5. set\_dtype 16](#_Toc884511)

[3.3.6. shape 16](#_Toc884512)

[3.3.7. type 16](#_Toc884513)

[3.3.8. UnitSize 16](#_Toc884514)

[3.3.9. DataCount 17](#_Toc884515)

[3.3.10. TypeStr 17](#_Toc884516)

[3.3.11. Dtypestr 17](#_Toc884517)

[3.3.12. operator<< 17](#_Toc884518)

[3.3.13. set\_batch\_size 18](#_Toc884519)

[3.3.14. batch\_size 18](#_Toc884520)

[3.4. Tensor 18](#_Toc884521)

[3.4.1. 构造函数(0) 18](#_Toc884522)

[3.4.2. 构造函数(2) 18](#_Toc884523)

[3.4.3. 构造函数(1) 18](#_Toc884524)

[3.4.4. head（暂不支持） 19](#_Toc884525)

[3.4.5. tensor\_desc 19](#_Toc884526)

[3.4.6. shape 19](#_Toc884527)

[3.4.7. type 19](#_Toc884528)

[3.4.8. dtype 19](#_Toc884529)

[3.4.9. capacity 20](#_Toc884530)

[3.4.10. channel\_id 20](#_Toc884531)

[3.4.11. cpu\_data 20](#_Toc884532)

[3.4.12. mlu\_data()(Unsupported) 20](#_Toc884533)

[3.4.13. mutable\_cpu\_data 20](#_Toc884534)

[3.4.14. mutable\_mlu\_data(Unsupported) 21](#_Toc884535)

[3.4.15. CopyFromCpu 21](#_Toc884536)

[3.4.16. CopyBatchFromCpu 21](#_Toc884537)

[3.4.17. CopyFromMlu(Unsupported) 21](#_Toc884538)

[3.4.18. CopyFromTensor 22](#_Toc884539)

[3.4.19. SyncedHead 22](#_Toc884540)

[3.4.20. Reshape 22](#_Toc884541)

[3.4.21. set\_channel\_id 22](#_Toc884542)

[3.4.22. frame\_ids 23](#_Toc884543)

[3.4.23. set\_frame\_ids 23](#_Toc884544)

[3.4.24. AppendFrameId 23](#_Toc884545)

[3.4.25. operator<< 23](#_Toc884546)

[3.4.26. mutable\_cpu\_batch\_data 24](#_Toc884547)

[3.4.27.cpu\_batch\_data 24](#_Toc884548)

[3.4.28. DumpData(0) 24](#_Toc884549)

[3.4.29. DumpData(1) 25](#_Toc884550)

[3.4.30.UnitSize 25](#_Toc884551)

[3.5. Connector 25](#_Toc884552)

[3.5.1. 构造函数(4) 25](#_Toc884553)

[3.5.2. buf\_count 26](#_Toc884554)

[3.5.3. tensor\_descs 26](#_Toc884555)

[第4章 接口 26](#_Toc884556)

[4.2. Module 26](#_Toc884557)

[4.2.1. 连接器机制 26](#_Toc884558)

[4.2.2. 构造函数（2） 27](#_Toc884559)

[4.2.3. 构造函数（3） 27](#_Toc884560)

[4.2.4. CheckAndInitialize 27](#_Toc884561)

[4.2.5. destory 27](#_Toc884562)

[4.2.6. start 28](#_Toc884563)

[4.2.7. stop 28](#_Toc884564)

[4.2.8. name 28](#_Toc884565)

[4.2.9. set\_name 28](#_Toc884566)

[4.2.10. container 28](#_Toc884567)

[4.2.11. inputs 28](#_Toc884568)

[4.2.12. outputs 29](#_Toc884569)

[4.2.13. GetInputData::GetInputData 29](#_Toc884570)

[4.2.14. GetInputData::data 29](#_Toc884571)

[4.2.15.GetInputData::operator[] 29](#_Toc884572)

[4.2.16. GetInputBuf::next\_buf 30](#_Toc884573)

[4.2.17. GetOutputBuf::GetOutputBuf 30](#_Toc884574)

[4.2.18. GetOutputBuf::buf 30](#_Toc884575)

[4.2.19. GetOutputBuf::operator[] 30](#_Toc884576)

[4.2.20. GetOutputBuf::next\_buf 31](#_Toc884577)

[4.2.21. running 31](#_Toc884578)

[4.2.22. SetPerformanceDispalyMode 31](#_Toc884579)

[4.3. Pipeline 31](#_Toc884580)

[4.3.1. AddModule 32](#_Toc884581)

[4.3.2. AddVideo 32](#_Toc884582)

[4.3.4. start 32](#_Toc884583)

[4.3.4. stop 32](#_Toc884584)

[4.3.5. channel\_count 32](#_Toc884585)

[4.3.6. input\_videos 32](#_Toc884586)

[4.3.7.构造函数(0) 33](#_Toc884587)

[4.3.8.构造函数(1) 33](#_Toc884588)

[4.3.9.复制构造函数 33](#_Toc884589)

[4.3.10.operator= 33](#_Toc884590)

[4.4. P2pDecInf 33](#_Toc884591)

[4.4.1.Attr 34](#_Toc884592)

[4.4.2.Create 34](#_Toc884593)

[4.4.3. con\_for\_inf 34](#_Toc884594)

[4.4.3. con\_for\_bgr 35](#_Toc884595)

[**4.5. Decoder** 35](#_Toc884596)

[**4.5.1.Attr** 35](#_Toc884597)

[**4.5.2.获取模块属性** 35](#_Toc884598)

[**4.5.3.Create** 36](#_Toc884599)

[**4.5.3.SendData** 36](#_Toc884600)

[**4.6. Inferencer** 36](#_Toc884601)

[**4.6.1.Create** 36](#_Toc884602)

[第5章 示例程序 38](#_Toc884603)

[5.1. 插件式设计（自定义Module） 38](#_Toc884604)

第1章 简介

CNSTEAM是面向MLU100的流处理SDK。CNSTREAM可以大大简化MLU100提供的推理能力和其他处理（如视频解码）的集成。CNSTREAM在兼顾灵活性的同时，可以充分发挥MLU的硬件解码和机器学习算法运算的性能。CNSTREAM提供了一套插件系统，用户可以根据CNSTREAM提供的API实现自己的组件，通过组件之间的互连灵活地实现自己的需求。

CNSTREAM基于模块化和流水线的思想，提供了一套C++语言的API，CNSTREAM提供了神经网络必须的推理模块，用于流处理的可多路并发的PIPELINE框架和可自定义的插件机制以及模块间数据传输的通用Tensor类型以满足性能和可伸缩性需求。并且，针对视频结构化领域，CNSTREAM提供了3个通用功能模块：视频解码模块、 机器学习算法运算模块和颜色域转换模块，其中视频解码模块可以对多种格式的视频压缩格式进行解码。机器学习算法模块可以使用多种神经网络离线模型对解码得到的图象数据进行神经网络推理。CNSTREAM的插件式设计提供给用户对视频流解码和推理之后，对数据进一步加工处理的办法。

视频解码模块依赖于CNCODEC SDK（MLU视频解码SDK），机器学习算法运算模块依赖于CNRT，CNSTREAM以用户的角度提供了3个自定义示例模块：数据前处理模块、数据后处理模块和多媒体显示模块。

CNSTREAM具有以下几个特点：

* 支持解码常见的视频压缩格式（H264、H265、MPEG）
* 支持硬解码JPEG图片格式(1080P,720P)
* 快速简单部署机器学习算法，高效支持常见分类以及目标检测神经网络（VGG、ResNet、Faster R-CNN）；高效支持低位宽（int8）及稀疏权值运算）；
* 灵活可变的流水线设计和插件集成机制。

## 关于P2P：

P2P模式是指解码器输出数据不经过CPU, 直接拷贝到MLU上做神经网推理。相对于非P2P模式，减少了一次数据拷贝，降低带宽占用，前处理在mlu上进行。另外，如果生成离线模式使用的模型文件带rgb0输出的话，还可减少在cpu上做颜色空间转换（yuv420sp nv21 转 rgb）带来的cpu资源消耗。

非P2P模式与P2P模式对比：

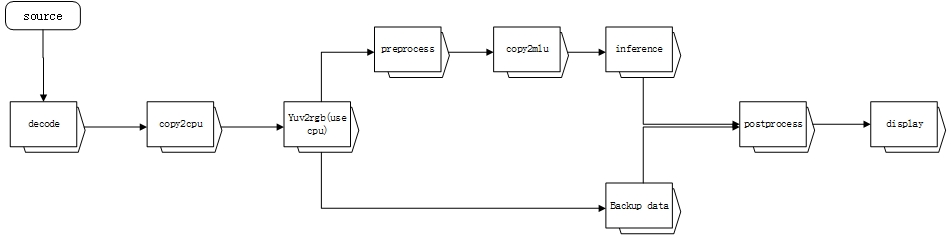


图1.1 cnstream非p2p流程



图1.2 cnstream p2p流程

**限制**：生成离线模型所用的模型文件必须支持yuv转rgb（带yuv2rgb层），yuv2rgb层作为第一个输出层。

**注意**：推理结果中，图像数据（rgb0）为NHWC顺序且数据类型为UIN8。其它输出为NCHW、FLOAT32。

第2章 CNSTREAM编程模型

## 依赖库

CNSTREAM有以下软件依赖：

**Linux OS**

**CNCODEC SDK**

**CNRT SDK**

**Glog**

除此之外，运行CNSTREAM还需要MLU驱动的支持。

## 文件目录

在CNSTREAM SDK目录下，主要有三个部分：头文件，动态库和示例程序源码。

include的文件夹下是CNSTREAM的头文件，所有的数据类型和接口都包含在其中。

lib的文件夹下是CNSTREAM动态库，文件名为libcnstream.so；

example下是一系列示例程序。

include

lib

example

CNStream头文件

libcnstream.so

示例程序

CNStream目录结构

## 编程模型

CNSTREAM是典型的基于pipeline和插件机制的编程模型，用户在pipeline注册多个预置插件或者用户自定义的插件，这些插件之间用过sdk提供的连接器(Connector)组件进行连接，每个插件（Module）可以连接多个输入连接器和多个输出连接器，使用同一个连接器分别作为输入连接器和输出连接器的两个插件即可自动连接在一起，模块间的数据通过连接器传输。

最常见的使用步骤如下：

1．创建一个pipeline对象。

2. 创建预置的decoder插件,该插件可以使用mlu解码视频和jpeg图片。

3．创建预置的inferencer插件，该插件可以使用mlu进行推理。

4. 创建其他用户自定义的插件，如数据的前处理，后处理，显示等插件

5. 将所有插件注册（Pipeline::AddModule）到pipline，启动pipeline。

第3章 公用数据类型

本章介绍CNSTREAM的公用数据类型。

## 3.1. Shape

用于描述Tensor的形状。

### 3.1.1. 构造函数(0)

Shape(uint32\_t n = 1, uint32\_t h = 1,

uint32\_t w = 1, uint32\_t c = 1);

参数

n

即BatchSize

h

高

w

宽

c

通道数

### 3.1.2. n setter

set\_n(uint32\_t n);

参数

n

shape中n的值

### 3.1.3. c setter

Set\_c(uint32\_t c);

参数

c

shape中c的值

### 3.1.4. h setter

Set\_h(uint32\_t h);

参数

h

设置的shape中的h

### 3.1.5. w setter

Set\_w(uint32\_t w);

参数

w

设置的shape中的w

### 3.1.6. n getter

n();

参数

无

返回shape的n

### 3.1.7. c getter

c();

参数

无

返回shape的c

### 3.1.8. h getter

h();

参数

无

返回shape的h

### 3.1.9. w getter

w();

参数

无

返回shape的w

### 3.1.10. nhwc

size\_t nhwc() const;

n \* h \* w \* c, shape描述的数据量。

返回值

返回该Shape描述的数据量。

### 3.1.11. operator<<

friend std::ostream &operator<<(std::ostream &os, const Shape &shape);

重写输出操作符 按照 n,c,h,w的顺序输出

参数

os

输出流

shape

被输出的Shape

### 3.1.12.operator==

bool operator==(const Shape& other);

判断两个shape是否相同

参数

other

相比较的shape

### 3.1.13.operator!=

bool operator!=(const Shape& other);

判断两个shape是否不同

参数

other

相比较的shape

### 3.1.14.hwc

uint32\_t chw() const;

返回c\*h\*w;

### 3.1.15.hw

uint32\_t hw() const;

返回h\*w

## 3.2. TensorType

Tensor的类型。

变量

RAW

自定义数据类型。

YUV

YUV图像数据。

RGB

RGB图像数据。

NEURON

神经网络输出数据。

COMMON

## 3.3 TensorDesc

描述Tensor信息

### 3.3.1. 构造函数(0)

TensorDesc();

默认构造函数。

参数

无

### 3.3.2. 构造函数(3)

TensorDesc(Shape shape, TensorType type, TensorDataType dType);

参数

shape

Tensor的形状。

type

Tensor的类型。

dType

Tensor内部数据的类型。

### 3.3.3. set\_shape

Void set\_shape(const Shape& shape);

设置Tensor的形状

参数

shape

Tensor的形状。

### 3.3.4. set\_type

Void set\_type(TensorType type);

设置Tensor的类型。

参数

type

Tensor的类型。

### 3.3.5. set\_dtype

Void set\_dtype(TensorDataType d\_type);

设置Tensor的数据类型。

参数

d\_type

Tensor的数据类型。

### 3.3.6. shape

const Shape& shape() const;

返回Tensor的形状

参数

无

### 3.3.7. type

TensorType type() const;

返回Tensor的类型

参数

无

### 3.3.8. UnitSize

size\_t UnitSize() const;

返回所描述Tensor的内部数据类型的字节数（exp. sizeof(uint8\_t)）

参数

无

### 3.3.9. DataCount

size\_t DataCount() const;

返回所描述的Tensor中存放的数据个数(batch size \* N \* C \* H \* W)

参数

无

### 3.3.10. TypeStr

std::sting TypeStr() const;

返回Tensor的类型字符串

参数

无

### 3.3.11. Dtypestr

std::string DtypeStr() const;

返回描述Tensor的内部数据类型的字符串

参数

无

### 3.3.12. operator<<

std::ostream &operator<<(std::ostream &os,const TensorDesc& tensor\_desc);

重载Tensor的输出符

参数

Os

输出流

**desc**

被输出的TensorDesc描述类

### 3.3.13. set\_batch\_size

void set\_batch\_size(uint32\_t bsize)

设置所描述的Tensor中所存放的数据的batch size。Tensor中所存放的数据量为batch\_size() \* shape.nhwc();

参数

bsize

### 3.3.14. batch\_size

uint32\_t batch\_size() const

获得所描述的Tensor中所存放的数据的batch size。

## 3.4. Tensor

cnstream数据包装器，cnstream的插件中流通的数据都由tensor来承载。Tensor支持uint8、float32、int32、uint32、int64这几种数据类型，具体见DataType。

### 3.4.1. 构造函数(0)

Tensor();

### 3.4.2. 构造函数(2)

Tensor(uint32\_t channel\_id, const TensorDesc& tensor\_desc);

参数

channel\_id

所属的视频通道（视频通道从0开始编号）。

tensor\_desc

Tensor的描述类。

### 3.4.3. 构造函数(1)

Tensor(const TensorDesc& tensor\_desc);

参数

tensor\_desc

Tensor的描述类。

### 3.4.4. head（暂不支持）

SyncedHead head() const;

返回数据同步状态

参数

无

### 3.4.5. tensor\_desc

const TensorDesc& tensor\_desc() const;

返回Tensor的描述符

参数

无

### 3.4.6. shape

const Shape& shape() const;

返回Tensor的形状

参数

无

### 3.4.7. type

TensorType type();

返回Tensor的类型

参数

无

### 3.4.8. dtype

TensorDataType dtype() const;

返回Tensor中存放的数据的类型

参数

无

### 3.4.9. capacity

size\_t capacity() const;

返回Tensor中数据所占的内存大小

参数

无

### 3.4.10. channel\_id

uint32\_T channel\_id() const;

返回Tensor所属的通道

参数

无

### 3.4.11. cpu\_data

const void\* cpu\_data();

返回host上的数据指针

返回值

host上的数据指针。

### 3.4.12. mlu\_data()(Unsupported)

const void\* mlu\_data();

返回mlu上的数据指针

返回值

mlu上的数据指针。

### 3.4.13. mutable\_cpu\_data

void\* mutable\_cpu\_data();

返回cpu上的数据指针

返回值

cpu上的数据指针。

### 3.4.14. mutable\_mlu\_data(Unsupported)

void\* mutable\_mlu\_data();

返回mlu上的数据指针

返回值

返回mlu上的数据指针。

### 3.4.15. CopyFromCpu

void CopyFromCpu(const void\* data);

从cpu上拷贝数据到Tensor。

参数

data

cpu上被拷贝的源数据指针。

### 3.4.16. CopyBatchFromCpu

void CopyBatchFromCpu(const void\* data, uint8\_t batch\_index);

从cpu上拷贝数据到Tensor的指定批次。

参数

data

cpu上被拷贝的源数据指针。

batch\_index

要拷贝的数据批次。

### 3.4.17. CopyFromMlu(Unsupported)

void CopyFromMlu(const void\* data);

从mlu上拷贝数据到Tensor。

参数

data

mlu上被拷贝的源数据指针。

### 3.4.18. CopyFromTensor

void CopyFromTensor(const std::shared\_ptr<Tensor>& src);

从源Tensor上拷贝数据到目标Tensor。

参数

src

Tensor上被拷贝的源数据指针。

### 3.4.19. SyncedHead

数据同步状态。

变量

UNINITIALIZED

未初始化状态。

HEAD\_AT\_CPU

数据被同步到cpu上。

HEAD\_AT\_MLU

数据被同步到mlu上。

SYNCED

暂不支持。。

### 3.4.20. Reshape

void Reshape(Shape shape);

改变Tensor形状。

参数

shape

目标形状。

### 3.4.21. set\_channel\_id

void set\_channel\_id(uint8\_t channel\_id);

设置当前的channel\_id

参数

channel\_id

被设置的Tensor的channel\_id

### 3.4.22. frame\_ids

const std::vector<uint64>&frame\_ids() const;

获取当前的Tensor的frame\_ids

返回值

Tensor的frame\_id的数组

### 3.4.23. set\_frame\_ids

void set\_framde\_ids(const std::vector<uint64\_t>& frame\_ids);

设置当前Tensor的各个frame\_id

参数

frame\_ids

一个包含多个frame\_id的vector数组

### 3.4.24. AppendFrameId

void AppendFrameId(uint64\_t frame\_id);

为当前Tensor添加一个frame\_id

参数

frame\_id

一个需要被添加到当前Tensor的frame\_id

### 3.4.25. operator<<

friend std::ostream &operator<<(std::ostream &os, const Tensor &tensor);

参数

os

输出流

tensor

被输出的tensor

### 3.4.26. mutable\_cpu\_batch\_data

void\* mutable\_cpu\_batch\_data(uint8\_t batch\_index);

返回cpu上指定批次的数据指针

参数

batch\_index

数据批次

返回值

cpu上的数据指针

### 3.4.27.cpu\_batch\_data

const void\*cpu\_batch\_data(uint32\_t batch\_index);

返回cpu上指定批次的数据

参数

batch\_index

数据批次

返回值

cpu上的数据

### 3.4.28. DumpData(0)

void DumpData(std::string fname);

将指定的文件输出

参数

fname

文件名

### 3.4.29. DumpData(1)

Void DumpData (std::ostream& os);

使用标准输出流输出数据

参数

os

标准输出流

### 3.4.30.UnitSize

size\_t UnitSize() const;

一个数据的大小(例如:sizeof(uint8\_t))

## 3.5. Connector

Connector是连接流水线上各个module之间的连接器。每个module可以有0到多个输入连接器和0-多个输出连接器。上游module的输出连接器通常作为下游module的输入连接器，上游module在自己输出连接器中写入数据，下游module从自己的输入连接器中读取数据。以此达到数据流通的目的。同时每个connector包含多个buffer对象 每个buffer对象有一个数据队列和一个回收队列。

模块之间传输的最基本的数据类型Tensor就存放在队列中。

### 3.5.1. 构造函数(4)

Connector(size\_t buf\_count,vector<TensorDesc> vec\_tensor\_desc,size\_t max\_buf\_size=0,size\_t init\_buf\_size=0);

参数

buf\_count

Connector中包含的缓冲区的数量

Vec\_tensor\_desc

Tensor的描述

Max\_buf\_size

最大缓冲区大小

Init\_buf\_size

缓冲区初始大小

### 3.5.2. buf\_count

size\_t buf\_count() const;

返回connector的buffer数量

参数

无

### 3.5.3. tensor\_descs

vector<TensorDesc>& tesnsor\_descs() const;

返回Tensor的描述类的数组

参数

无

第4章 接口

本章介绍模块接口，包括插件式接口Module，Inferencer接口和Decoder接口。

## 4.2. Module

Module是CNStream执行程序的基本单元，每个Module都有一个start()方法用来执行本模块流程。如果用户自定的模块start方法则要由用户实现。用户通过Connector连接件连接相邻的模块。通过GetInputData系列接口获取上一个模块的输出数据，通过GetOutpuData接口把本模块的输出数据传递给下一个模块。内部实现减少内存的申请和释放次数，提高程序运行效率。

### 4.2.1. 连接器机制

Module之间通过连接器进行连接，每个Module可以使用多个Connector连接多个下游Module：

Std::vector<std::shared\_ptr<Connector>> Inputs;

Std::vector<std::shared\_ptr<Connector>> Outputs;

Inputs作为当前模块和上游模块连接用的Connector。

Outputs作为当前模块和下游模块连接用的Connector

Module从多个Inputs中获取输入数据，同时将输出数据写入多个Outputs中，供下游module使用（上游module的Outputs作为下游module的Inputs）

### 4.2.2. 构造函数（2）

Module(Inputs inputs, Output outputs);

参数

inputs

本模块的输入，使用连接器从上游模块获取输入

outpus

本模块的输出，使用连接器输出数据给下游模块

### 4.2.3. 构造函数（3）

Module(std::string name,Inputs inputs, Output outputs);

参数

name

模块名

inputs

本模块的输入，使用连接器从上游模块获取输入

outpus

本模块的输出，使用连接器输出数据给下游模块

### 4.2.4. CheckAndInitialize

virtual void CheckAndInitialize()

检查各个参数是否设置无误，并初始化

### 4.2.5. destory

virtual void destory();

在Pipeline停止时被调用，可用于释放initialize里申请的资源。

### 4.2.6. start

virtual void start() = 0;

在initialize之后被调用，可用于实现模块主逻辑。

### 4.2.7. stop

virtual void stop();

在Pipeline停止时被调用，在destory之前。

### 4.2.8. name

Std::string name();

返回模块名

返回值

返回模块名。

### 4.2.9. set\_name

Void set\_name(std::string name)

设置模块名

参数

name

模块名

### 4.2.10. container

Pipeline\* container()

返回包含该module对象的pipeline的指针

参数

无

### 4.2.11. inputs

Inputs& inputs()

返回本模块的输入Connector数组

参数

无

### 4.2.12. outputs

Outputs& outputs()

返回本模块的输出Connector数组

参数

无

### 4.2.13. GetInputData::GetInputData

GetInputData(Module\* owner,int input\_id,int buf\_id)

GetInputData结构体的构造函数，该结构体用来访问输入连接器中数据

参数

owner

使用该结构体的Module对象

Input\_id

输入连接器的索引，用来指明该结构体访问的是输入连接器数组中的哪个连接器。

Buf\_id

连接器的buffer的索引，该结构体访问的是输入连接器的Buffer数组中的哪个Buffer。

### 4.2.14. GetInputData::data

std::shared\_ptr<Tensor> data(int id);

返回GetInputData结构体中索引为id的Tensor

返回值

GetInputData结构体中索引为id的Tensor

### 4.2.15.GetInputData::operator[]

std::shared\_ptr<Tensor> operator[](int id);

重载GetInputBuf结构体的下标操作符

参数

**id**

Tensor对象的索引

### 4.2.16. GetInputBuf::next\_buf

void next\_data() noexcept(false);

### 4.2.17. GetOutputBuf::GetOutputBuf

Getoutputbuf(Module\* owner,int output\_id,int buf\_id)

GetOutputBuf结构体的构造函数，初始化该结构体的buffer缓冲区

参数

owner

使用该结构体的Module对象

output\_id

输入连接器的索引，用来指明该结构体访问的是输出连接器数组中的哪个连接器。

Buf\_id

连接器的buffer的索引，该结构体访问的是输出连接器的Buffer数组中的哪一个buffer

### 4.2.18. GetOutputBuf::buf

Std::shared\_ptr<Tensor> buf(int id);

返回当前结构体的buffer数组中索引为id的Tensor

参数

**id**

Tensor对象的索引

### 4.2.19. GetOutputBuf::operator[]

Std::shared\_ptr<Tensor> operator[](int id)

重载GetOutputBuf结构体的下标操作符

参数

**id**

Tensor对象的索引

### 4.2.20. GetOutputBuf::next\_buf

Void next\_buf()

将结构体当前临时缓冲区的数据写入对应输出Connector对象的buffer对象的数据队列，以供下游module读取，并通过复用对应buffer对象的回收队列的空间来重置临时缓冲区。

参数

无

### 4.2.21. running

bool running() const;

获得模块的运行状态

返回值

当前模块正在运行返回true，否则返回false。

## 4.3. Pipeline

Pipeline是Module的管理单元，Pipeline会将成员Module按照Module的构造函数中使用的Connector连接器的队列机制连接成流水线。用户只需要往Pipeline里送入数据，之后各个Module会自动完成每一环节的处理，并将数据向下一个模块传递。

### 4.3.1. AddModule

void addModule(Module\* module);

添加一个模块到Pipeline。

参数

module

Module指针。

### 4.3.2. AddVideo

void AddData(std::string& video\_path);

往Pipeline中压入视频文件。（调用一次则代表该pipeline在处理一路视频流）

参数

Video\_path

视频文件的路径

### 4.3.4. start

void start();

开始执行Pipeline。

### 4.3.4. stop

void stop();

停止Pipeline。

### 4.3.5. channel\_count

Uint32\_t channel\_count()

返回Pipeline的通道数

### 4.3.6. input\_videos

Std::vector<std::string>& input\_videos();

返回Pipeline的视频文件数组。

### 4.3.7.构造函数(0)

Pipeline();

### 4.3.8.构造函数(1)

explicit Pipeline(const std::vector<std::string>& input\_videos);

参数

input\_videos

输入视频流的路径

### 4.3.9.复制构造函数

Pipeline(const Pipeline& other)

参数

other

被复制的pipeline

### 4.3.10.operator=

Pipeline& operator=(const Pipeline& other)

赋值运算符=的重定义

参数

other

赋值给新定义pipeline的pipeline

## 4.4. P2pDecInf

p2p模块，包括解码和神经网络推理。

### 4.4.1.Attr

P2p模块属性。

uint32\_t src\_frame\_rate 解码器数据输入帧率。

uint32\_t dst\_frame\_rate 解码器数据输出帧率。

uint32\_t target\_w 解码器输出图像宽度。

uint32\_t target\_h 解码器输出图像高度。

std::string model\_path 离线模型路径。

std::string function\_name （使用caffe生成离线模型时可在.twins文件中得到）

int inf\_output\_num 离线模型输出个数。

mrtcxx::MluMode mode 推理模式。

bool offmodel\_with\_image\_output 离线模型是否带由图像数据输出

bool input\_image 使用图像作为输入（image list）

### 4.4.2.Create

static std::shared\_ptr<P2pDecInf>& Create(Attr attr)

创建p2p模块。

参数

attr

用来配置p2p模块的属性。

返回值

返回p2p模块智能指针。

### 4.4.3. con\_for\_inf

static std::shared\_ptr<Connector>& con\_for\_inf() const

得到带有推理输出的connector智能指针。当配置p2p模块的attr.close\_inferencer为true时返回nullptr。

### 4.4.3. con\_for\_bgr

static std::shared\_ptr<Connector>& con\_for\_inf() const

得到带有bgr输出的connector智能指针。当配置p2p模块的attr.close\_bgr为true时返回nullptr。

**4.5. Decoder**

非p2p模式的解码模块，。

**4.5.1.Attr**

decoder模块属性。

double drop\_rate 解码器主动丢帧比率，range（0，1）。

uint32\_t src\_w 解码器可输入图像最大宽度。

uint32\_t src\_h 解码器可输入图像最大高度。

uint32\_t target\_w 解码器输出图像宽度。

uint32\_t target\_h 解码器输出图像高度。

uint32\_t buf\_cnt 缓冲区数量，应与inferencer线程数相等。

uint32\_t batch\_size 解码器输出数据batch\_size。

uint32\_t nm\_copies 解码器输出数拷贝份数（例：拷贝一份输出给显示）。

CN\_PIXEL\_FORMAT\_E output\_pixel\_format 输出图像格式（YUV、RGB等）。

std::function<CN­\_VIDEO\_CODEC\_TYPE\_E(int)> codec\_type\_getter

回调函数，获取每个通道视频编解码类型。

std::function<CN\_VIDEO\_MODE\_E(int)> video\_mode\_getter

回调函数，获取每个通道数据数据类型（帧解码、流解码）。

int device\_id 设备编号（使用多推理卡时需要设置，默认为0）。

**4.5.2.获取模块属性**

Attr attr();

得到模块属性结构体Attr。

**4.5.3.Create**

static std::shared\_ptr<Decoder> Create(Attr attr);

创建非p2p解码器模块。

参数

attr

用来配置Decoder模块的属性。

返回值

返回解码器模块智能指针。

**4.5.3.SendData**

bool SendData(uint32\_t channel\_id,

CN\_VIDEO\_PIC\_PARAM\_S \*st\_pic\_param);

由外部调用，输送解封装过的数据给解码模块解码

参数

channel\_id

通道号。

st\_pic\_param

图像解封装后的数据。

返回值

返回是否输送成功，当解码器未打开或已销毁时返回false，否则为true

**4.6. Inferencer**

非p2p神经网络推理模块。

**4.6.1.Create**

static std::shared\_ptr<Inferencer> Create(

const std::string& model\_path,

const std::string& function\_name,

std::shared\_ptr<Connector> input,

const std::vector<mrtcxx::DataType>& vec\_o\_dtype,

const std::vector<mrtcxx::DataOrder>& vec\_i\_data\_order,

const std::vector<mrtcxx::DataOrder>& vec\_o\_data\_order,

mrtcxx::MluMode func\_type = mrtcxx::MLU\_MODE\_UNION2,

int device\_id = 0);

创建非p2p神经网络推理模块。

参数

model\_path

离线模型路径。

function\_name

神经网络符号名称（一般为“subnet0”）

input

模块的输入连接器。

vec\_o\_dtype

神经网络输出数据类型（UINT8，FLOAT32等），vector的size应与神经网络输出个数相等。

vec\_i\_data\_order

神经网络输入数据格式（NCHW，NHWC），vector的size应与神经网络输入个数相等。

vec\_o\_data\_order

神经网络输出数据格式（NCHW，NHWC），vector的size应与神经网络输出个数相等。

func\_type

神经网络推理模式（UNION2、BLOCK）

device\_id

设备编号（使用多推理卡时需要设置，默认为0），需与相连接的Decoder模块保持一致。

第5章 示例程序

## 5.1. 插件式设计（自定义Module）

CNStream采用插件式设计，用户可以定制自己的Module。

示例1:下面是一个实现用户自定义功能(如:保存图片)的Module示例，具体示例可以参考sdk发布包中example目录下的cpp文件

class PicSaver : public cnstream::Module {

std::string save\_path\_;

std::vector<std::thread> vec\_thread\_;

//创建module

std::shared\_ptr<PicSaver> PicSaver::Create(std::shared\_ptr<cnstream::Connector> input, const std::string& image\_path) {

//略

//input连接器接收上游module的输出数据。该模块没有下游module所以没有输出的连接器

return std::shared\_ptr<PicSaver>(new PicSaver(input, image\_path));

}

//启动多个线程

void PicSaver::Start() {

auto buf\_count = inputs()[0]->buf\_count();

for (size\_t i = 0; i < buf\_count; ++i) {

vec\_thread\_.push\_back(std::move(std::thread(&PicSaver::ThreadFunc, this, i)));

}

}

//线程的执行函数

void PicSaver::ThreadFunc(int buf\_id) {

int cnt = 0;

try {

while (running()) {

//从输入连接器中不断获取新的数据

GetInputData input(this, 0, buf\_id);

auto image\_tensor = input.data(0);

auto shape = image\_tensor->shape();

IplImage\* image\_header = cvCreateImageHeader(cvSize(shape.w(), shape.h()), 8, 3);

for (uint32\_t i = 0; i < shape.n(); ++i) {

uint8\_t\* image\_data = static\_cast<uint8\_t\*>(image\_tensor->mutable\_cpu\_batch\_data(i));

cvSetData(image\_header, image\_data, shape.w() \* shape.c());

cv::Mat image(image\_header);

std::string path = save\_path\_ + std::to\_string(buf\_id) + "." + std::to\_string(cnt++) + ".jpg";

LOG(INFO) << path;

//将上游module输出的数据转变为图像格式并保存

cv::imwrite(path, image);

}

}

} catch (BufferStop& buf\_stop) {

DLOG(INFO) << "stopped";

}

}

}；