Cnstream

# 概述

Cnstream是一套基于寒武纪硬件平台的一套视频应用开发框架，采用模块化的设计。基于cnstream的应用都可以看作工厂的一条或多条流水线（Pipeline），在流水线中的每个模块各司其职，重复做同一件事情。Cnstream框架主要做了以下几件事情：

1. 基于cnstream的应用数据都由Tensor类承载。
2. 管理各模块的生命周期，包括初始化、开始、停止、释放资源。赋能类：Pipeline
3. 内存的申请和释放。主要是模块间流通的数据所使用的内存。Cnstream希望用户能够不必过多关心流水线（Pipeline）中流通的数据的内存。赋能类：Connector
4. 模块间数据传输。Cnstream希望用户不必过多的关注数据怎么在pipeline中传输。赋能类：Connector
5. 自带插件：基于寒武纪硬件平台的解码、推理模块。Cnstream希望用户只关心在寒武纪硬件平台上解码器的输出和推理的输出，而不必关心解码和推理的过程。
6. 支持自定义模块。

下面基于以上4点详细描述

# 基本数据类型Tensor

Tensor是cnstream中的基本数据类型，在插件中传输的数据都由Tensor来承载。使用过程中只需要把它当成void\*来使用即可。

TensorDesc类型用来描述Tensor的基本信息。主要包括Tensor中存储的数据量，数据类型，

**数据量**：Tensor::batch\_size() \* Tensor::shape().nhwc()即为Tensor中存储的数据量。一般batch\_size指该Tensor中存储着几帧数据，shape指数据规模。

**数据类型:** 支持uint8、int32、int64、uint32、float32。具体见TensorDesc类型声明。

**读取数据的接口：**

1. Void\* mutable\_cpu\_data()、const void\* cpu\_data() const。这两个接口可得到数据的首地址。
2. Void\* mutable\_cpu\_batch\_data(uint32\_t batch\_index)、const void\* cpu\_batch\_data(uint32\_t batch\_index) const。把Tensor看作二维数组，batch\_size作为第一维的大小，shape.n() \* shape.h() \* shape.w() \* shape.c()看作第二维的大小。这两个接口就相当于这个二维数组中第一维的下标运算符[]的作用。

综上，Tensor的使用方式较为简单。

通过上述4个接口取得数据指针，通过操作该指针实现对Tensor中数据的读写。

# 各模块生命周期

Cnstream中，所有模块都继承自Module类。Module类定义如下：

Class Module {

…

protected:

virtual void CheckAndInitialize();

virtual void Start() = 0;

virtual void Stop();

virtual void Destroy();

…

};

可见以上有4个可由Module派生类重写的虚接口。在cnstream中，建议用户在模块中重写这4个接口并分别做以下几件事情：

**CheckAndInitialize**：建议重写该接口并实现初始化资源的功能。在该接口被调用时，可以拿到当前模块所在的Pipeline正在处理的视频路数（Pipeline::channel\_count()）。例如Decoder插件在该函数被调用时，根据视频路数申请相应数量的解码器资源。

**Start**：建议重写该接口并根据该插件有多少个输入数据队列创建对应数据处理线程。当该接口被调用时（只有被Pipeline调用才生效），数据才开始流通。Module::running()返回true。

**Stop**：建议重写该接口并处理Start中创建的线程的退出工作。当该接口被调用时（只有被Pipeline调用才生效），数据停止流通Module::running()返回false。Module::GetInputData和Module::GetOutputBuf接口将不再阻塞并抛出BufferStop异常。

**Destroy**: 建议重写该接口并释放CheckAndInitialize中申请的资源。

Pipeline继承自Module类，且也拥有以上4个接口。而在Pipeline的4个接口中，将按模块注册（Pipeline::AddModule）顺序去依次调用每个模块的相应的接口。比如，Pipeline::Start将依次调用所有模块的Start函数。

如此，当Pipeline::CheckAndInitialize,Pipeline::Start,Pipeline::Stop, Pipeline::Destroy4个函数依次调用即组成了一个Pipeline的生命周期。

# 内存管理

当Connector创建完毕，就意味着在这个Connector中传输的数据会占用的最大内存和传输的数据长什么样就被确定好了，且不可改变。Connector的构造函数如下：

Connector(size\_t buf\_count,

std::vector<TensorDesc> vec\_tensor\_desc,

size\_t max\_buf\_size = 20,

size\_t init\_buf\_size = 0);

各参数详细解释：

**Buf\_count**: 在该connector中数据队列的个数（std::queue）。比如在sample ssd中，当推理模式采用MLU\_MODE\_BLOCK的时候，且处理32路视频，则P2pDecInf模块的输出Connector的buf\_count为32。

**Vec\_tensor\_desc**: TensorDesc数组，用来描述在当前Connector中流通的数据长什么样。Vec\_tensor\_desc.size() 为 2 代表每次通过调用GetInputData 和GetOutputBuf接口能得到两个Tensor。而TensorDesc则是描述了Tensor的形状、batch size、数据类型这些信息。

TensorDesc {

private:

//-----------------MEMBERS-----------------//

mrtcxx::Shape shape\_;

TensorType type\_;

mrtcxx::DataType dtype\_;（uint8, uint32, int32, int64,float32）

uint32\_t batch\_size\_ = 1;

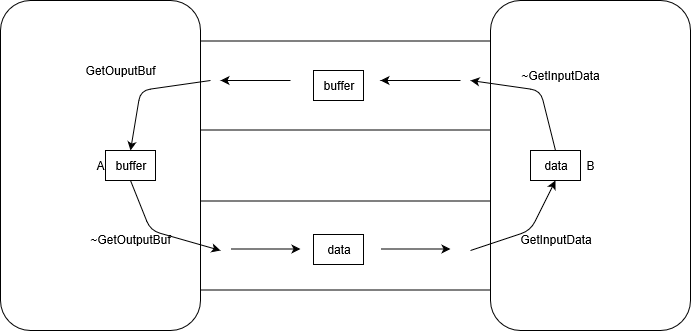
}；

**Max\_buf\_size**: 数据队列中最大缓存多少份数据。当缓存达到max\_buf\_size时GetOutputBuf将阻塞,直到数据队列中的数据被取出。

**Init\_buf\_size**: 数据队列创建时为多少份数据提前申请好内存。即创建Connector对象时就创建内存，max\_buf\_size – init\_buf\_size部分的内存在数据传输过程中视情况申请。

**内存复用机制：**

1. Module::GetInputData对象被构造时从数据队列中取得一份数据，当该对象被析构时，数据回收至数据队列中等待被复用。Module::GetInputData::next\_data接口可以取得下一份数据，同时把当前得数据内存回收至数据队列,等待其它模块调用GetOutputBuf接口复用该内存。
2. Module::GetOutputBuf对象被构造时从数据队列中取得一份数据内存。当该对象被析构时，数据被存放入数据队列中等待其它模块取用。Module::GetOutputBuf::next\_buf接口可以取得下一份数据内存，同时把当前数据放入数据队列中等待其它模块调用GetInputData接口取用。



# 模块间数据传输

Cnstream不希望用户过多关心模块间的数据是怎么传输的。在一个模块中，模块本身只需要关注模块本身的输入是什么，有几个输入、输出是什么，有几个输出等信息。而这些信息都被封装在Connector类中。通过Module的构造函数传递给模块：

Module(const Inputs& inputs, const Outputs& outputs);

Inputs 和Outputs是同一个类型，即Connector数组。从上面的描述中我们知道，对于模块来说，Connector完全可以看做是个队列数组（std::vector<std::queue>）。当Connector被放在inputs数组中传递给Module时，该模块对这个Connector中的队列只有pop数据的权限。当Connector被放在outputs数组中传递给Module时，该模块对这个Connector只有push数据的权限。当然，cnstream并不需要让用户真的去执行push或者pop。这些操作利用RALL机制隐藏在了GetInputData和GetOutputBuf接口中。

**两个存取数据的重要接口**：

1. Module::GetInputData

声明：Module:: GetInputData(const Module\* owner, int input\_id, int buf\_id);

参数说明：

**Owner**：模块指针，标识是哪个模块需要取输入数据。（没有特殊需求传this就好了）

**Input\_id**: 在Module构造函数中传入了一个Connector数组（inputs）。Input\_id用来标识从inputs[input\_id]这个Connector中取数据。

**Buf\_id**: 在Connector构造函数中第一个参数buf\_count标识了这个Connector中有几个数据队列。Buf\_id即标识了从Connector中的第几个数据队列中取数据。把Connector写成std::vector<std::queue> queues。即从queues[buf\_id]这个数据队列中取数据。

简单来说，模块要取得输入数据，只需要告诉GetInputData这个接口，我要从第几个Connector中的第几个数据队列中取数据。

下面使用代码来描述模块如何取输入数据：

首先，模块有一系列输入队列，可以把它写成std::vector<std::vector<std::queue>> queues。我们给出这个二维数组的两个下标：input\_id, buf\_id。GetInputData 即从queues[input\_id][buf\_id]这个数据队列中pop一个数据出来。

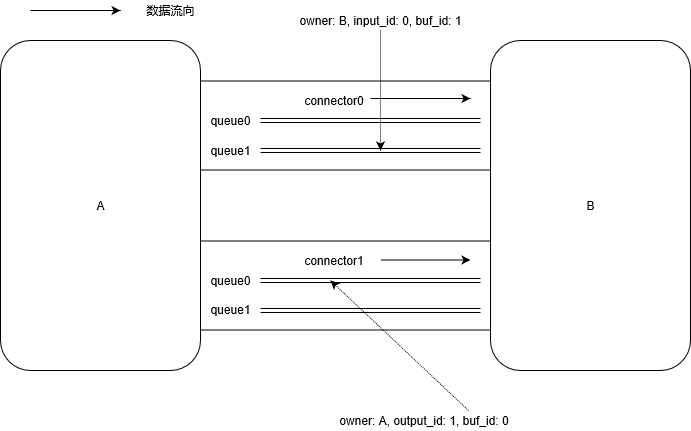
**那么，每次queues[input\_id][buf\_id]中pop出的数据长什么样呢？**

我们可以把每次pop出的数据类型看作std::vector<Tensor> tensors。

tensors有如下特性：

1. tensors.size() 与 创建Connector时给出的vec\_tensor\_desc的TensorDesc数组的size相等。
2. tensors中每个tensor的描述符TensorDesc与vec\_tensor\_desc中的TensorDesc一一对应（相等）。
3. Module::GetOutputBuf

参考GetInputData接口的描述。



# 自带插件

1. **Decoder插件。**

**Decoder插件是干什么的？**

Decoder插件提供在mlu上的解码功能。解码后的数据输出到host端。

**创建Decoder插件和各参数的意义**

创建Decoder插件的接口：static std::shared\_ptr<Decoder> Create(Attr attr);

参数：

struct Attr {

double drop\_rate;

uint32\_t src\_w;

uint32\_t src\_h;

uint32\_t target\_w;

uint32\_t target\_h;

uint32\_t buf\_cnt = 0;

uint32\_t batch\_size = 1;

uint32\_t nm\_copies = 1;

CN\_PIXEL\_FORMAT\_E output\_pixel\_format;

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

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

int device\_id = 0;

}; // struct DecoderAttr

Drop\_rate: 解码器的主动丢帧功能，取值范围[0, 1）。当为0时标识不进行丢帧。当不为0时，假设输入给解码器的帧率为20fps，当被动丢帧未触发时，解码器输出20 \* （1 – drop\_rate）fps。

Src\_w: 最大可处理的视频源分辨率的宽。

Src\_h: 最大可处理的视频源分辨率的高。

Target\_w: 输出分辨率的宽。

Target\_h: 输出分辨率的高。

Buf\_cnt: 输出Connector中的数据队列的个数。Decoder会把各通道数据尽量均匀分布在buf\_cnt个数据队列中。假设通道号为chn\_id, 则该通道的图像被放在下标为chn\_id % buf\_cnt的数据队列中。一个视频通道的数据不会被放在不同的输出数据队列中。

Batch\_size：一次输出的数据中包含batch\_size张图像数据，这些图像来自同一个视频通道。该插件输出Connector中的tensor desc中的batch size与它相等。

Nm\_copies: 把图像复制多份的功能。decoder->outputs().size() 与 nm\_copies相等。每个输出Connector中的数据是一样的图像。v7.2之前的版本该功能异常。

Output\_pixel\_format: 设置输出像数格式（bgr24或yuv420sp\_nv21）

Codec\_type\_getter: 回调函数，用来告诉Decoder每个通道处理的视频流的编码格式，该回调有一个参数channel\_id, 对应Pipeline::AddVideo中加入的视频流url。Pipeline中存储了所有视频流的url，想象Pipeline中有个成员变量std::vector<std::string> urls，Channel\_id 即为urls的下标，Pipeline::channel\_count（）返回urls.size()。用户编写的Codec\_type\_getter根据channel\_id返回给解码器该url对应的视频流的编码格式（支持的编码格式见cncodec.h中的枚举类型CN\_VIDEO\_CODEC\_TYPE\_E）。

Video\_mode\_getter: 回调函数，用来告诉Decoder每个通道处理的视频流是按帧发送数据给解码器还是按流的方式发送给解码器（见cncodec.h中的枚举类型CN\_VIDEO\_MODE\_E）。

Device\_id: mlu设备号，多卡支持。

**被动丢帧**

当Decoder模块的输出未被及时处理时，Decoder模块的GetOutputBuf操作被阻塞，造成解码器被动丢帧。均匀丢帧。

**如何发送数据给解码器？**

Decoder插件没有输入Connector，即Decoder::inputs().size()为0。Decoder提供相应接口方便用户发送数据给解码器：Decoder:: bool SendData(uint32\_t channel\_id, CN\_VIDEO\_PIC\_PARAM\_S \*st\_pic\_param)。其中channel\_id为解码器通道号，返回false标识发送数据失败。使用示例见example中的DataSrc模块。

**解码器的输出Connector**

1. ~~输出Connector个数为与nm\_copies参数相等，即Decoder::outputs().size() = nm\_copies。~~**~~目前有bug不要使用。~~**
2. 每个输出Connector的Connector::buf\_count()返回值与buf\_cnt参数相等，即每个Connector中有buf\_cnt个输出队列。
3. 每个输出Connector的Connector::tensor\_descs().size() 为1，即Decoder的每份输出数据std::vector<Tensor> tensors中有tensors.size() 为1。

当output\_pixel\_format 为rgb时Connector::tensor\_descs()[0].shape() 为 （n = 1, h = target\_h, w = target\_w, c = 3）。当output\_pixel\_format为yuv时，shape为（n = 1, h = target\_h \* 3 / 2, w = target\_w, c = 1）。

1. 当output\_pixel\_format为rgb时，Connector::tensor\_descs()[0].type() 返回TensorType::RGB, output\_pixel\_format为yuv时，Connector::tensor\_descs()[0].type()返回TensorType::YUV。
2. Connector::tensor\_descs()[0].dtype()返回mrtcxx::UINT8。即输出数据类型为uint8。

**限制：**

1. Src\_w 和 src\_h的大小决定能创建解码器的个数，若创建解码器失败，Decoder::CheckAndInitialize将抛出DecoderError异常。例如：当src\_w = 1280, src\_h = 720时，mlu100 最大支持创建64个解码通道，当Pipeline::AddVideo被调用次数大于64，即Pipeline::channel\_count()返回值大于64，则抛出异常。Src\_w = 1920, src\_h = 1080时，最大为54个解码器通道。
2. Target\_w 为16的倍数， target\_h 为2的倍数，否则解码器输出图像不正确。更多限制参考cncodec相关文档。
3. **Inferencer插件**

**Inferencer插件是干什么的？**

推理插件，输入数据来自host端。非p2p模式（模型输入为rgb，模型不需要增加yuv2rgb层）。用户只需提供离线模型路径（离线模型与框架无关，使用各种框架生成的离线模型都可以使用），告知输出数据在host端的数据类型（例如uint8，float32）和摆放顺序（NHWC 或者NCHW）和输入数据在host端的摆放顺序即可。输入数据类型写死为float32。

**创建Inferencer插件及各参数意义**

创建Inferencer插件的接口为：

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);

参数：

**Model\_path**: 离线模型路径。

**Function\_name**: 生成离线模型后在offline.cambricon\_twins文件中有function name 的相关描述，一般为subnet0。

**Input**: 输入Connector。在创建Inferencer插件时会对input做检查，其中input->tensor\_descs().size() 要与模型输入个数相等，input->tensor\_descs()[index].shape()要与对应的模型输入规模一致。

**Vec\_o\_dtype**: 离线模型每个输出数据在host端的数据类型。Vec\_o\_dtype.size()要与模型输出个数一致，一般都填FLOAT32即可。创建插件时做检查。

**Vec\_i\_data\_order**: 离线模型每个输入数据在host端的数据摆放顺序，vec\_i\_data\_order.size()要与模型输入个数一致。创建插件时做检查。

**Vec\_o\_data\_order**: 离线模型每个输出数据在host端的摆放顺序。Vec\_o\_data\_order.size()要与模型输出个数一致，一般填NCHW即可。创建插件时做检查。

**Func\_type**: 推理模式，当为MLU\_MODE\_BLOCK模式时，mlu每次推理一帧数据，此时,input->tensor\_descs()[index].batch\_size() = 1。当为MLU\_MODE\_UNION2模式时，mlu每次并行推理8帧数据，此时，input->tensor\_descs()[index].batch\_size() = 8。创建插件时做检查。

**Device\_id**: mlu设备号，多卡支持。

1. **P2pDecInfer插件**

集解码和推理为一体的插件，p2p模式，解码后数据不经过host端，直接拷贝到mlu上进行推理。相比于常规模式，如caffe中的example。主要优点为：图像不用从解码器拷出再考入mlu，减少了一次内存拷贝以及host to device拷贝时输入摆数的大量cpu占用和性能损耗。

**如何创建插件及各参数意义**

**创建插件的接口：Static std::shared\_ptr<P2pDecInf> Create(Attr attr);**

**Drop\_rate**: 见decoder插件描述。

**Src\_w**: 见decoder插件描述。

**Src\_h**: 见decoder插件描述。

**Target\_w**: 解码器输出到mlu上的图像数据的宽（用作推理的图像分辨率）。应与模型输入规模一致，创建插件时做检查。限制：16的倍数

**Target\_h**: 解码器输出到mlu上的图像数据的高（用作推理的图像分辨率）。应与模型输入规模一致，创建插件时做检查。限制：2的倍数

**Codec\_type\_getter**: 见decoder插件描述。

**Video\_mode\_getter**: 见decoder插件描述。

**Model\_path**: 离线模型路径（模型必须以yuv作为输入，即模型中第一层为yuv2rgb，且第二层为卷积，否则不支持）。

**Function\_name**: 一般填subnet0，见Inferencer插件描述。

**Inf\_output\_num**: 模型的输出个数，例如：ssd为1， fasterrcnn为3

**Inf\_output\_image**: 离线模型是否带有rgb0输出（即模型中yuv2rgb层参数external\_outputs是否为1）。

**Mode**: union2 和block，见Inferencer插件描述。

**Device\_id**: mlu设备号，多卡支持。

**Buf\_cnt**: 输出数据队列的个数。当采用BLOCK模式进行推理的时候，一般设置为与视频通道数一致即可。当采用UNION2模式进行推理的时候，一般设置为CNRT\_CHANNEL\_NUM(就是4)即可。可根据实际应用场景自行调整，主要取决于用来接收P2pDecInf模块输出的数据的模块希望开启的线程数量。Cnstream中自带插件都保证了每个视频通道的数据顺序，保证顺序的方式是：在一个模块中，保证同一路视频的数据都经过同一个输出队列。为了达到这个目的，自带插件有以下三个规则：

1. 每个模块针对输入Connector中数据队列的个数创建对应数量的线程去处理输入数据。
2. 当输入Connector中的数据队列数量与输出队列中的数据队列数量不相等时，通过tensor中的channel id()（视频通道号）对输出Connector中的queue\_num(数据队列的数量)进行取余计算（channel\_id % queue\_num）得到该视频通道号应该放入哪个输出数据队列中。
3. 在输入有多个Connector或输出有多个Connector的插件中，保证输入/输出的所有Connector中的数据队列的数量相等。

**Host\_target\_w**:双分辨率输出功能，输出到host端的图像的宽。

**Host\_target\_h**:双分辨率输出功能，输出到host端的图像的高。

**双分辨率输出功能：**

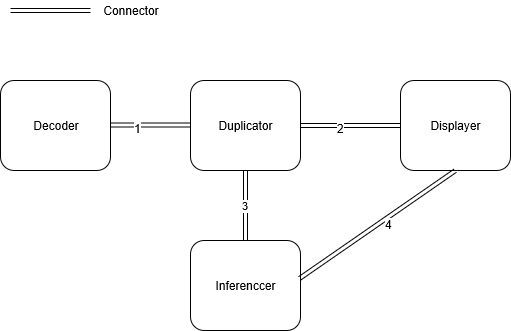
生效方式：当host\_target\_w和host\_target\_h都不为0。

表现：P2pDecInf::outputs().size()返回2即该插件有两个输出Connector。当不使用双分辨率输出功能时返回1。P2pDecInf::outputs()[0]中可取得离线模型的输出数据。P2pDecInf::outputs()[1]中可取得分辨率为host\_target\_w \* host\_target\_h 的yuv420sp\_nv21图像数据。

1. **Duplicator插件**

该插件提供复制数据的功能。可以理解为一个数据队列复制为多个数据队列，且每个复制的数据队列与原数据队列一模一样（数据相同且顺序一样）。

使用场景举例：非p2p模式的解码+目标检测+显示。在此场景下，Decoder插件输出的图像既要传递给Inferencer模块去推理，又要传递给显示插件。此时，需要把Decoder插件的输出复制一份。数据流程图如下：



其中，connector1、connector2、connector3中的数据完全一致。

# 自定义插件说明

#include <cnstream.hpp>

// 1.定以Module的派生类，并重写4个函数。

class Plugin : public Module {

public:

static shared\_ptr<Plugin> Create(const Inputs& inputs) {

// 2.实现创建插件的功能

// 一般情况下插件的功能决定了插件的输出，所以通常由插件创建输出Connector。这里在Create函数中创建输出Connector。

Outputs outputs; // vector<shared\_ptr<Connector>> outputs

// outputs.push\_back ...

// outputs.push\_back ...

return shared\_ptr<Plugin>(new Plugin(inputs, outputs));

}

void CheckAndInitialize() override {

// 3.对输入输出做一些必要的检查

// 4.申请需要的资源

}

void Start() override {

// 5.此函数被Pipeline调用时开启线程处理数据

// Start一般步骤

// 5.1取得输入的输入队列个数。

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

// 5.2创建线程

threads\_.clear();

for (decltype(queue\_num) qi = 0; qi < queue\_num; ++qi) {

threads\_.push\_back(thread(&Plugin::ThreadFunc, this, qi);

}

}

void Stop() override {

// 6.此函数被Pipeline调用时数据停止传输，等待数据处理线程结束。

for (auto& it : threads\_) {

if (it.joinable()) {

it.join();

}

}

}

void Destroy() override {

// 7.释放CheckAndInitialize中申请的资源。

}

private:

Plugin(const Inputs& inputs, const Outputs& outputs)

: Module("module name", inputs, outputs) {

}

void ThreadFunc(uint32\_t queue\_id) {

// 8.循环处理queue\_id对应的输入队列中的数据。

try {

while (running()) { // running()返回false表示Pipeline::Stop被调用。

// 8.1 取得输入数据

GetInputData tensors1(this, 0, queue\_id); // 从第一个输入Connector中的第queue\_id个数据队列中取得一份输入数据。

GetInputData tensors2(this, 1, queue\_id); // 从第二个输入Connector中的第queue\_id个数据队列中取得一份输出数据。

// ...

// tensors1、tensors2都可当作vector<shared\_ptr<Tensor>>来使用。GetInputData和GetOutputBuf都重载了下标运算符[]。

// 其中tensors1.size() = inputs()[0]->tensor\_descs().size()。

// tensors1[index]->tensor\_desc() = inputs()[0]->tensor\_descs()[index];

// 8.2 取得用来承载输出数据的Tensor

GetOutputBuf tensors3(this, 0, queue\_id);

// 8.3 取得数据指针。这里假设数据都是uint8类型。

auto input1 = reinterpret\_cast<const uint8\_t\*>(tensors1[0]->cpu\_data());

auto input2 = reinterpret\_cast<const uint8\_t\*>(tensors2[0]->cpu\_data());

auto output = reinterpret\_cast<uint8\_t\*>(tensors3[0]->mutable\_cpu\_data());

// 8.4 处理数据

for (;;) {

output[index] = input1[index] + input2[index];

}

// 8.5 传递channel id和frame id

tensors3[0]->set\_channel\_id(tensors1[0]->channel\_id()); // frame\_ids().size() = tensor->batch\_size()

tensors3[0]->set\_frame\_ids(tensors1[0]->frame\_ids());

}

} catch (BufferStop& bstop) {

// 当GetInputData 和 GetOutputBuf在阻塞状态中。Pipeline::Stop被调用，则BufferStop异常被抛出，结束数据处理即可。

}

}

vector<thread> threads\_;

};