云天励飞DESDK Graph编程手册

正在修改[√] 正式发布[ ]	当前版本:	**V1.1.0**
	完稿日期:	**2020.9.22**
	审核日期:	**2020.9.22**

深圳云天励飞技术有限公司

声明

本文档是云天励飞提供接口使用文档，便于合作的企事业单位使用，云天励飞保留所有接口使用权利，未经许可不得公开接口使用信息给任何非授权企业和个人。

更新记录

版本	修改日期	修改说明
V0.1.0	2020.4.8	初始版本
V0.2.0	2020.5.29	补充模型推理算子
V1.0.0	2020.7.15	v1.0.0 版本更新接口
V1.0.1	2020.9.13	V1.0.1 增加 Graph 析构接口说明
V1.1.0	2020.9.22	增加 node 扩展功能说明

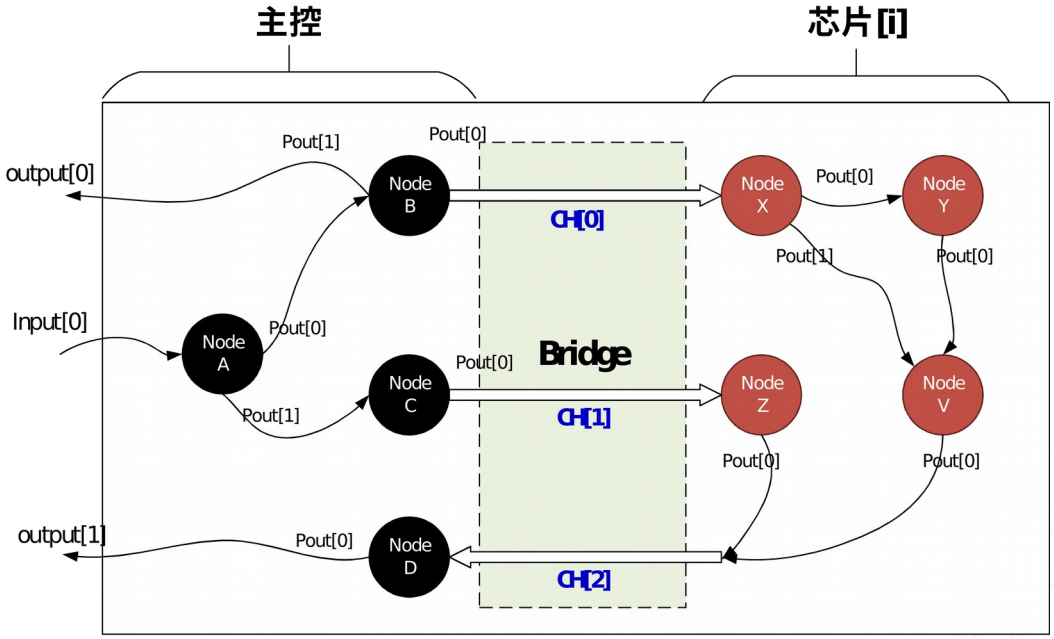
目录

1. 概述

本文档介绍如何使用 DESDK Graph 编程方法和标准 Node 算子开发应用程序。

2. Graph 编程简介

DeepEye1000 DESDK 提供 graph 功能，以数据流的方式，提供跨平台（host 和 device）的数据流水处理。Graph 提供直观的编程模式，支持 json 配置文件创建，在 host 端直接调用 device 的 NodeClass 算子，大大提高解决方案设计效率。



1. 解决方案 Graph 图

如上图，用户利用 graph 把 host 端的 Nodes 和 device 端的 Nodes 连接起来。

2.1. 使用 Graph API 创建 Graph

使用代码创建 Graph 的过程如下：

```
de::Graph graph("graph_demo");


// step 1: create bridge
```

```
graph.CreateBridge(dev_id);
graph.AddBridgeChan(0, "User_CH0", true);
graph.AddBridgeChan(1, "User_CH1", true);
graph.AddBridgeChan(2, "User_CH2", false);

//step 2: create node
graph.CreateHostNode("NodeAClass",  "nodeA"); // nodeA in host
graph.CreateHostNode("NodeBClass ", "nodeB");
graph.CreateHostNode("NodeCClass ", "nodeC");

graph.CreateDevNode("NodeZClass ", "nodeZ"); // nodeZ in device
graph.CreateDevNode("NodeVClass ", "nodeV");

// step 3: link node
graph.LinkNode("nodeA", 0, "nodeB");   //nodeA.pout[0] □nodeC.pin
graph.LinkNode("nodeA", 1, "nodeC");   //nodeA.pout[1] □nodeC.pin
……
//nodeB.pout[0] □ bridge channel 0 □nodeX.pin
graph.LinkNode("nodeB", 0, "nodeX", 0);

// step 4: set graph input and output
graph.SetInputNode(0, "nodeA");        //graph.pin[0] is nodeA's pin
……
graph.SetOutputNode(1, "nodeD", 0);    //graph.pout[1] is nodeC's pout[0]

// step 5: data process
Graph.pin[0].SendTask(task); // data input
task = graph.pout[1].RecvTask(task_type); // data output
```

2.2. 通过读取 json 配置文件创建 Graph

一个 Graph json 配置文件示例如下：

```
{
  "graph_demo": {
    "bridges": [
      {
        "dev_id": 0,
        "url": "",
        "port": -1
      }
    ],
    "chans": [
```

```
    {
      "direct": "H2D",
      "ch_id": 0,
      "name": "User_CH0",
      "attr": {}
    },
     {
      "direct": "H2D",
      "ch_id": 1,
      "name": "User_CH1",
      "attr": {}
    },
    {
      "direct": "D2H",
      "ch_id": 2,
      "name": "User_CH2",
      "attr": {}
    }
  ],
  "nodes": [
    {
      "location": "HOST",
      "class": "NodeAClass",
      "name": "nodeA",
      "attr": [],
      "link": [
        {
          "dest": "nodeB",
          "pout_idx": 0,
          "bridge": -1
        },
           {
          "dest": "nodeC",
          "pout_idx": 1,
          "bridge": -1
        }
      ]
    },
     {
      "location": "HOST",
      "class": "NodeBClass",
      "name": "nodeB",
      "attr": [],
      "link": [
```

```
          {
         "dest": "nodeX",
         "pout_idx": 0,
         "bridge": 0
        }
         ]
    },
     {
     "location": "DEV",
     "class": "NodeXClass",
     "name": "nodeX",
     "attr": [],
     "link": []
    },
     {
     "location": "DEV",
     "class": "NodeYClass",
     "name": "nodeY",
     "attr": [],
     "link": []
    }
    ],
   "inputs": [
      {
       "idx": 0,
         "node": "nodeA"
      }
    ],
   "outputs": [
      {
       "idx": 1,
         "node": "nodeD",
         "node_pout", 0
      }
    ]
  }
}
```

通过 json 文件创建 Graph 的代码如下：

```
de::Graph* pgraph = de::Graph::FromFile(“graph_demo.cfg”);
```

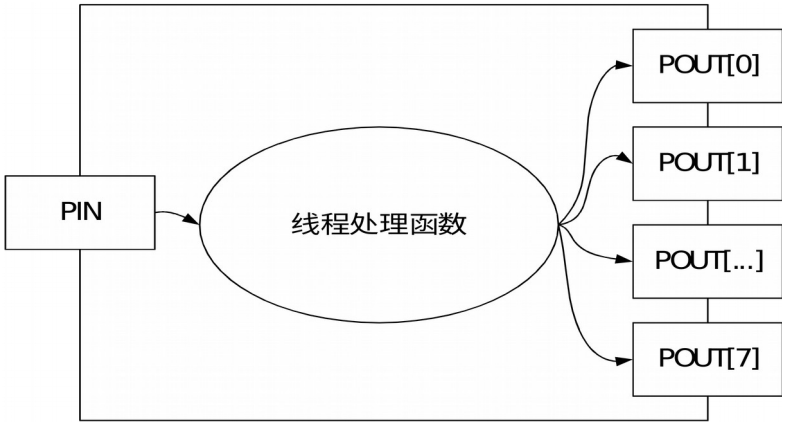
3. 编程接口以及约束

3.1. Node 算子

3.1.1. 概述

NodeClass 算子类利用注册机制，支持动态生成 node 对象，比如芯片视频解码器提供名称为 ” dp1000::VideoDec” 的算子类，那么用户可以直接多次调用 graph.CreateDevNode(“dp1000::VideoDec”, “nodeXXX”)创建多个 dec 对象，dec 对象可以通过下图中 pin 接收任务，处理完毕后，通过 pout 发送任务给下一级算子对象处理。

NodeClass 算子类派生于 de::Thread 类，支持单输入 Queue（定义为 PIN），多输出 Queue（定义为 POUT）。



2. Node 算子

假设 NodeX 支持 PIN 的 TypeA 和 TypeB 的任务处理，并分别通过 POUT[0]发送任务 TypeC 和 POUT[1]发送任务 TypeD，那么 NodeX 算子类的线程处理函数设计如下：

```
virtual void NodeXClass:：Proc(void* rx_task, int32_t task_type) {
 if(task_type == 0) {
    auto task = static_cast<TypeA*>(rx_task);
    …… // TypeA task process
    Auto tx_task = new TypeC();
    …….
    Pout[0].SendTask(tx_task);
 }
  if(task_type == 1) {
    auto task = static_cast<TypeB*>(rx_task);
```

```
    …… // TypeB task process
    Auto tx_task = new TypeD();
    …….
    Pout[1].SendTask(tx_task);
 }
 pin.DelTask(rx_task, task_type);
}
```

3.1.2. PIN 和 POUT

Node 算子本质上是线程，线程利用 Queue 进行任务传递通信，而 Queue 就是封装在线程内部的 PIN 和 POUTs 内部。

PIN 提供算子的输入相关信息，包括 Queue 信息、类型信息、是否通过 bridge 转接等，用户通过 pin.RcvTask 获取消息。

POUT 提供算子的输出相关信息，包括 Queue 信息、类型信息、Bridge 信息等，用户通过 pout[i].SendTask 发送消息。

3.1.3. 静态属性设置

Node 算子类在构造函数的时候设置静态属性，静态属性分为：

1) 设置属性的缺省参数：比如，attr_.SetDefault("ratio", 0.2, 0, 1.0); 表示 Node 算子的参数（key,value）缺省值为 0.2，范围为[0,1]
2) 设置任务类型信息，任务类型用于检测两个 Node 对象之间传递的 task 类型是否匹配，比如概述的例子 NodeX 算子使用了 NodeA 类定义的任务类型 TypeA 和 NodeB 类定义的任务类型 TypeB，且线程处理的 task type 分别对应为 0,1，那么：pin.SetTypeInfo(0, "NodeA::TypeA", nullptr);
   pin.SetTypeInfo(1, "NodeB::TypeB", nullptr);
3) 如果 Node 算子类支持跨平台通信，比如 NodeA 在 Host 端，NodeX 在 Device 端，那么 NodeA□NodeB 需要定义 NodeA 算子的 Pout 序列化函数，NodeB 算子 Pin 反序列化函数。对于概述的例子，NodeX 算子要支持如下序列化和反序列化定义：

```
pin.SetTypeInfo(0, "NodeA::TypeA", TypeADeserFunc);
pin.SetTypeInfo(1, "NodeB::TypeB", TypeBDeserFunc);
pout[0].SetTypeInfo(“NodeX::TypeC”, TypeCSerFunc);
pout[1].SetTypeInfo(“NodeX::TypeD”, TypeDSerFunc);
```

说明

- ✓ Node 算子的属性值设置非法（key 不存在或者 value 超过范围）会报错；
- ✓ Node 算子的支持的类型信息，必须统一，这里建议采用类型的 struct/class 名称字符（包括命名空间）来区别；
- ✓ Node 算子 Pin 输入支持的 type id 和 type name 必须和 process 保持一致；

✓ Node 算子类 Pout[15]的输出类型为 de::NodeProfileTask，用于调试信息、profile 信息的字符串输出；

3.1.4. 动态属性设置

每个算子提供 Key，value（支持 int、double、以及 string）方式设置算子的属性，比如 FaceDet 算子类生成的对象 Node[“face_det”]，通过 key = “min_face”, value = 48，设置 face detect 对象的最小人脸为 48。

具体支持的属性，请参考算子的 API 文档。

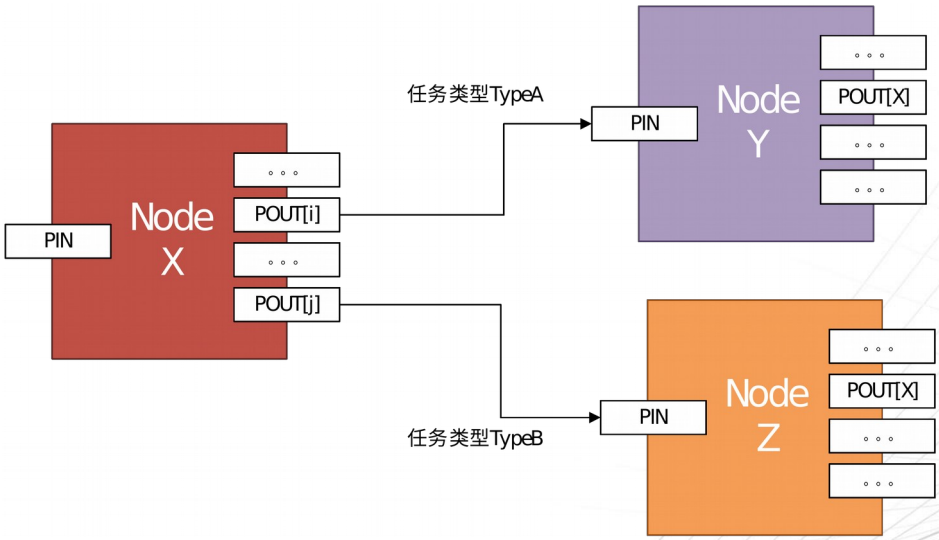
说明

✓ 用户使用 de::Thread 的 UpdatePara()函数来刷新动态属性参数；

3.1.5. Node 之间级联约束

Node 支持如下几种方式级联：

1) Source 算子在 host 端，Destination 算子在 host 端，本地级联；
2) Source 算子在 host 端，Destination 算子在 device 端，通过 bridge 跨平台级联，见 Node 通过 Bridge 级联约束;
3) Source 算子在 device 端，Destination 算子在 host 端，通过 bridge 跨平台级联，见 Node 通过 Bridge 级联约束;
4) Source 算子在 device 端，Destination 算子在 device 端，远程控制 device 内部级联；



3. Node 之间的级联（一对多）

如上图所示，Node X 算子使用 POUT[i]和 Node Y 算子 PIN 连接，同时也使用 POUT[j]和 Node Z 算子 PIN 连接。

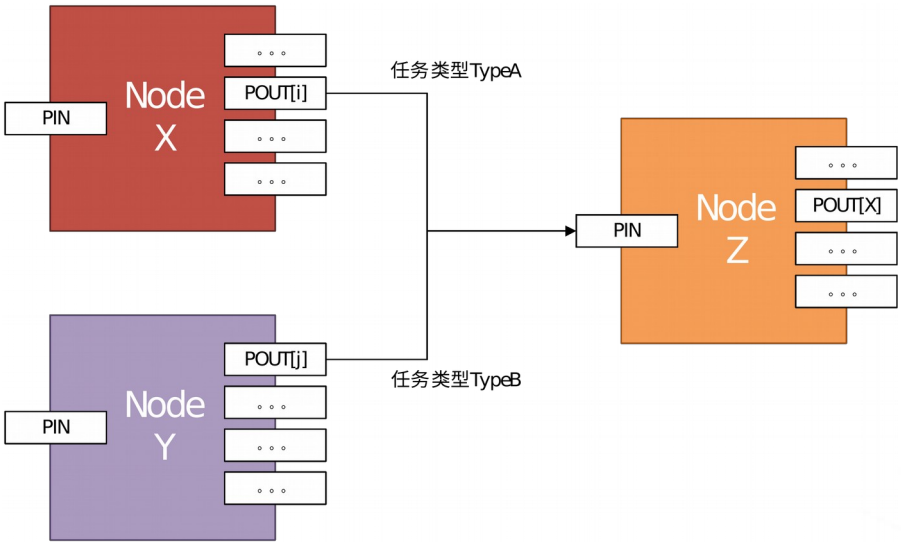
算子之间的级联唯一约束条件，就是 source 算子通过 POUT 发送的 task 任务类型和 destination 算子 PIN 接收的 task 任务类型匹配，这里我们约定使用任务的类型字符串做为标记，比如 JPEG 的 Dec

算子输出任务类型定义为 de::mm::JpegDecTask，而 Arctern 的 FaceDet 算子输入任务要支持 Dec 算子的输出，那么在各自的类构造函数上表明自己支持的类型，代码见下，那么 JpegDecoder 算子的 Pout[0]和 Pout[1]允许和 FaceDet 的 Pin 链接。如果名称不匹配，那么算子间 link 会报错。

```
JpegDecoder::JpegDecoder() {
…
 POUT[0].SetTypeInfo(“de::mm::JpegDecTask”, nullptr);
 POUT[1].SetTypeInfo(“de::mm::JpegDecTask”, nullptr);
 POUT[7].SetTypeInfo(“de::DebugStringTask”, nullptr);


…
}
Arctern::FaceDet::FaceDet() {
…
Pin.SetTypeInfo(0, “de::mm::JpegDecTask”, nullptr);
…
}
```

需要注意：Destination 算子是支持多个 Source 算子输入的，唯一条件就是 Destination 算子的输入要支持所连接的 Source 算子输出类型。



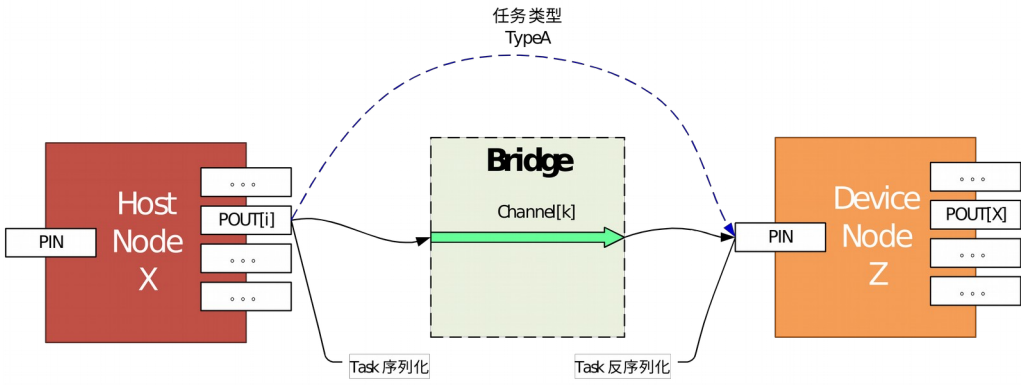
4. Node 之间的级联（多对一）

3.2. Bridge 开发

3.2.1. 概述

Bridge 用于连接 Host 和 Device 端通信（不支持 Device 之间互连），Bridge 提供 1 到多个端到端通信。见下图，Host 的算子 Node X 和 Device 的算子 Node Z 利用 bridge 的 channel k 来互连，

对于 Node X 和 Node Z 的算子类设计而言是透明传递的。



5. Node 之间 Bridge 级联

说明

✓ NodeClass 算子类设计时候，如果要支持跨平台部署，那么需要提供任务类型的序列化和反序列化，否则会报错；

3.2.2. Node 通过 Bridge 级联约束

Bridge 提供多个单向逻辑信道通信，而实际每个 Device 支持的物理信道有限（usb 端点），如果 graph 占用信道资源不够，就会出现 bridge 创建失败，此时需要复用物理信道。

如果用户提供的任务序列化的码流长度超过物理信道传输上限，那么会在 graph 运行时候出错。

说明

✓ Bridge 在创建逻辑信道时候，提供 channel name 字符串属性，如果多个逻辑信道的名称同名，那么占用的同一个物理信道。注意，不同方向的信道不能同名，为了效率考虑，数据通道和控制通道建议名称区别开来，分配不同的物理信道。

✓ Bridge 逻辑信道支持的最大数据传输不超过 3.9Mbyte。

3.2.3. 序列化和反序列化

3.2.3.1 通用序列化和反序列化

DeSDK 提供轻量级的任务序列化和反序列化的模板，以上图的例子来说明如何使用。上图中 Host Node X 的 POUT[i]和 Device 的 Node Z 的 PIN 传输的任务类型为 TypeA，见下面代码：

```
// 步骤 1：类型定义以及序列化模板
struct TypeA {
    int format;
    int pts;
    NDArray data;
    Std: : string name;
```

```
};
namespace serializer {
    STRUCT_SERIALIZE_4(TypeA,
        int, format,
        int, pts,
        NDArray, data,
        Std::string name);
};

// 步骤 2：Node X 类设计,构造函数中申明 pout[i]的序列化
NodeX::NodeX() {
pout[i].SetTypeInfo("TypeA ", de::TaskSerialize<TypeA>);
};

// 步骤 3：Node Z 类设计,构造函数中申明 pin 的反序列化
NodeZ::NodeZ() {
pin.SetTypeInfo(0, "TypeA ", de:: TaskDeSerialize<TypeA>);
};
```

说明

- ✓ 序列化不支持 union 类型、bit field、指针类型、数组；
- ✓ 序列化支持 STL 模板，包括 std pair,vector,map 等；
- ✓ 解决方案设计注意：数据 NDarray 的序列化代价非常大，需要分析系统的性能瓶颈；

3.2.3.2 特殊序列化和反序列化

对于数据 NDarray 的序列化开销大问题，DeSDK 提供一种特殊的任务序列化和反序列化的模板，以上图的例子来说明如何使用。上图中 Host Node X 的 POUT[i]和 Device 的 Node Z 的 PIN 传输的任务类型为 TypeB，见下面代码：

```
// 步骤 1：类型定义以及序列化模板
struct TypeBCtrl {
    int format;
    int pts;
    Std：：string name;
};

struct TypeB {
     TypeBCtrlctrl_info;
    NDArray data;
inline de::NDArray Serialize()
      {
            de::WriteDataToBearArea<TypeBCtrl>(img, ctrl_info);
```

```
return img;
      }

inline void DeSerialize(de::NDArray data)
       {
            track_result = de::GetDataFromBearArea<TypeBCtrl>(data);
img = data;
       }
 };

// 步骤 2：Node X 类设计,构造函数中申明 pout[i]的序列化
NodeX::NodeX() {
pout[i].SetTypeInfo("TypeA ", de::TaskSerializeCustom<TypeB>);
};

// 步骤 3：Node Z 类设计,构造函数中申明 pin 的反序列化
NodeZ::NodeZ() {
pin.SetTypeInfo(0, "TypeA ", de:: TaskDeSerializeCustom<TypeB>);
};
```

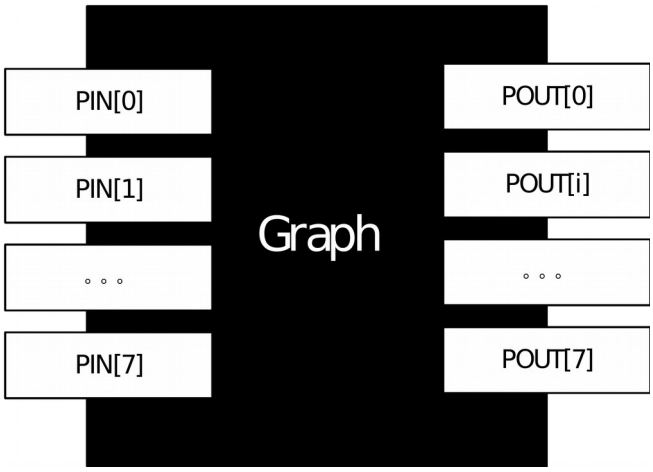
说明

- ✓ 该序列化模板需要用户实现 Serialize 和 DeSerialize 接口；
- ✓ 默认控制信息域 256 字节，可根据需要在构建数据时申请更大的空间；

3.3. Graph 开发

3.3.1. 概述

用户使用 graph 实现业务逻辑的 Node 算子串接，Graph 提供多输入、多输出的业务数据流处理模型。用户利用 graph 的 PINs 输入 task 进行业务处理，使用 POUTs 获取业务处理的结果。



6. Graph 业务模型

说明

✓ Graph 的输入和输出接口仅限于本地调用，不支持跨平台调用；

3.3.2. Graph 之间的级联

Graphs 之间也可以利用 graph 的 PINs 和 POUTs 级联，形成业务逻辑功能更强大的 graph。

说明

✓ 不支持两个 graph 内部的 Node 算子对象级联；

3.3.3. Graph 析构

Graph 析构包括以下几个步骤：

1．按照创建 Node 的相反顺序，逐一停止每个 Node，等待 Node 处理主线程结束；
2．等待 Graph 输入结束（即 Graph::SetInput）
3．等待 Graph 输出结束（即 Graph::GetOutput）
4．删除 bridge
5．按照创建 Node 的相反顺序，逐一删除每个 Node
6．卸载 Graph 启动过程中额外加载的动态库（包括 Host 侧和 Device 侧）

Graph 析构时需要注意的事项：

1．GetOutput 和 StopOutput 的使用

GetOutput 获取不到输出结果的时候，会阻塞当前线程。所以通常不应在主线程里直接调用 GetOutput，而是创建一个新线程，在线程里调用 GetOutput 去获取 Graph 的输出。当在控制线程（例如主线程）中析构 Graph 对象时，必须要保证没有线程阻塞在 GetOutput 里。

Graph 提供了一个接口 StopOutput，用来使阻塞在 GetOutput 中的线程从阻塞状态中退出。使用方法如下：

```
pgraph->StopOutput<SimpleTask1>(1);
delete pgraph;
```

其中，函数参数 1，表示 graph pout 1；模板参数 SimpleTask1，是 graph pout 1 中输出的 Task 类型。

另外也存在一种情况：用户虽然调用 GetOutput，但是明确知道何时可以停止 GetOutput，并不会阻塞在 GetOutput 中。例如：

```
std::thread([](de::Graph* pgraph) {
    void* p = pgraph->GetOutput(1);
    //其他操作
    return;
}, &graph).detach();
```

对于这种情况，析构 Graph 前不需要调用 StopOutput。但调用 StopOutput 也不会有问题，所以如果不确定 GetOutput 是否会阻塞，可以在 Graph 析构前统一调用一次 StopOutput。

4. 算子类型定义

4.1. NodeProfileTask

类型名称	NodeProfileTask	
功能描述	用于传输 Node profile 信息	
成员	**类型**	**作用**
str_node_name	string	算子名称，用以区分 profile 数据来源
rx_task_num	int	算子接收的任务总数量
tx_task_num	int	算子发送的任务总数量
max_queue_size	int	在一个 profile 周期内，PIN 的队列最大深度
average_queue_size	int	在一个 profile 周期内，PIN 的队列平均深度
proc_average_time	int	处理一条任务平均所需时间
proc_max_time	int	处理一条任务最长所需时间
frame_no	int64_t	frame number，帧号
total_cpu_occupy	int64_t	cpu 总使用时间
proc_cpu_occupy	int64_t	算子任务处理线程的 cpu 使用时间
proc_mem_usage	int	片子任务处理线程的 memory 使用量
time_stap	int64_t	profile 数据生成时的当前时间戳
备注		

4.2. NodeTimerTask

类型名称	NodeTimerTask	
功能描述	用于传输 Timer 信息	
成员	类型	作用
time_stap	int64_t	NodeTimerTask 携带的时间戳
备注		

5. 标准算子

DESDK 提供大量的标准 Node 算子供用户直接使用，能够解决大部分的应用问题。

5.1. 算子的通用属性

算子名称	de::Thread		支持平台	所有平台
功能描述	通用的算子父类			
类型	接口/关键字名称	类型	说明	
属性(Prop)	pin.que.size	int	pin 队列深度，1~128，默认为 16	
	que.size.warn.thres	int	pin 队列深度警告门限。当 pin 队列中堆积的任务个数大于该门限时，打印警告信息，1~128，默认为 10	
	poutX.que.size	int	X 为 0~15，表示 pout0~pout15 的队列尝试，取值 1~128，默认为 32	
	debug.pin.grab	int	是否抓取 pin 的输入任务数据	
	debug.pin.grab.file	string	抓取数据后保存目标文件路径，默认”/tmp/1.bin”	
	debug.pin.grab.max.mbsize	int	抓取的最大数据量，默认为 4（单位：MB）	
	profile.enable	int	是否开启算子的 profile 功能，取值 0 或者 1，默认为 0	
	profile.name	string	profile 结果携带的名称，用以区分 profile 数据来源，默认为”Anonynode”	
	profile.stat.period	int	输出 profile 结果的周期，默认 1000，最小 500（单位：ms）	
	debug.level	int	LOG 打印等级，默认为 WARNNING	
	intpu.filter.callback	string	任务过滤回调函数名称	

输出(Pout)	pout[15]		调试、Profile 输出端口
备注			

5.2. 媒体算子

5.2.1. 视频输入算子

算子名称	VideoInput		支持平台	X86/DP1000/ARM
功能描述	获取基于 RTSP 协议的视频流，获取 v4l2 设备中的视频流和图像，获取取本地文件中的视频和图像			
类型	**接口/关键字名称**	**类型**	**说明**	
输入(Pin)	无			
属性(Prop)	stream-id	int	当前流 ID	
	uri	String	Uri 格式的视频流或图像地址，例如： rtsp://login:passwd@the ipcipaddr" v4l2:///dev/videoX file:///data/your_video.h264	
	width	int	图像宽度（16-4096）	
	height	int	图像高度（16-4096）	
	framerate	double	帧率（0.1-60）	
	lower-trans-type	int	RTSP 传输层协议（RTSP_TRANS_UDP/RTSP_TRANS_UDP_MCAST/RTSP_TRANS_TCP）	
	stream-type	int	视频流编码格式或原始图像格式 (DE_PIX_FMT_H264/DE_PIX_FMT_H265/DE_PIX_FMT_MJPEG/ DE_PIX_FMT_YUV422_YUYV/DE_PIX_FMT_YUV422_UYVY/DE_PIX_FMT_NV16)	
	timeout	int	获取流超时时间（<=0: 永远等待，其他：超时时间）	
	test-pattern	int	测试模式，可显示动态彩条，只针对 v4l2 设备	
输出(Pout)	Pout[0]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[1]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[2]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[3]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
备注	1. 对于 X86 平台，支持 Linux 操作系统，依赖 gstreamer 第三方库； 2. YUV 格式的图像获取需要依据源的类型来定 3. 输出的对象是一样的			

5.2.2. 视频解码算子

算子名称	VideoDecoder		支持平台	X86/DP1000
功能描述	H264/H265 视频流解码器			
类型	**接口/关键字名称**	**类型**	**说明**	
输入(Pin)	Pin	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
属性(Prop)	output-buffers	int	解码器输 buffer 数目（4-64）	
	stream-type	int	视频流编码格式(DE_PIX_FMT_H264/DE_PIX_FMT_H265)	
	out-timeout	Int	获取解码图像超时时间（<=0: 永远等待，其他：超时时间）	
输出(Pout)	Pout[0]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[1]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[2]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[3]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[14]	MediaTask	媒体数据流对象，只传输超时或数据流结束标志	
备注	1. 对于 X86 平台，支持 Linux 操作系统，依赖 gstreamer 第三方库；			

5.2.3. JPEG 解码算子

算子名称	JpegDecoder		支持平台	X86/DP1000
功能描述	Jpeg 图像解码器			
类型	**接口/关键字名称**	**类型**	**说明**	
输入(Pin)	Pin	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
属性(Prop)				
输出(Pout)	Pout[0]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[1]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[2]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
	Pout[3]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据	
备注	1. 编码图像尺寸范围:48x48 - 16386x16386 2. 尺寸变化步进：水平和垂直均为 8 像素 3. 编码模式支持：baseline 4. 若原图的宽高不满足 8pixel 对齐，则软件会对输出 buffer 的尺寸进行对齐，使其输出图像的宽高满足 8 pixel 对齐，此时输出图像会有花边			

5.2.4. JPEG 编码算子

算子名称	JpegEncoder			支持平台	X86/DP1000
功能描述	JPEG 编码器				
类型	**接口/关键字名称**	**类型**	**说明**		
输入(Pin)	Pin	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据		
属性(Prop)	quality	int	编码器量化等级（0-10），值越大图像编码损失越小		
	out-size	Int	编码器输出 buffer 大小设置（128KB-48MB）		
输出(Pout)	Pout[0]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据		
	Pout[1]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据		
	Pout[2]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据		
	Pout[3]	MediaTask	媒体数据流对象，包含流序号，帧数据、帧号、标志、时间戳和自定义数据		
备注	1. 编码模式支持：baseline 2. 原图像格式支持：DE_PIX_FMT_YUV422_YUYV/DE_PIX_FMT_YUV422_UYVY/DE_PIX_FMT_NV12/ DE_PIX_FMT_NV21/ DE_PIX_FMT_YUV420P 3. 原图像尺寸范围：96x32 – 8192x8192 水平步进 16 像素，垂直步进不限 4. 编码图像尺寸范围：96x32 – 8192x8192 水平步进 4 像素，垂直步进 2 像素				

5.3. 模型推理算子

5.3.1 AiEngine 算子

算子名称	AiEngine			支持平台	DP1000
功能描述	模型推理 Ai 引擎				
类型	**接口/关键字名称**	**类型**	**说明**		
输入(Pin)	Pin	NNTask	输入对象类型		
属性(Prop)	batch_num	int	最大批处理 batch 数目（1~32），默认为 1		
	tensor_in_one_batch	int	一个 batch 所含输入 tensor 数目（1~256），默认为 1		
	encrypt	int	模型是否加密（0-不加密，1-加密），默认为 0		
	resize_type	int	模型 resize 类型（0-正常,1-等比例），默认为 0		
	model_net_path	string	模型网络文件（net.bin）路径		
	model_par	string	模型参数文件（model.bin）路径		

	am_path		
	period	int	执行周期（1~128），默认为 1
	copy_flag	int	模型结果是否拷贝后再上报（0-不拷贝，1-拷贝），默认为 0
输出(Pout)	Pout[0]	NNTask	输出对象类型
备注	无前后处理		

5.3.2 AiEngineExt 算子

算子名称	**AiEngineExt**		**支持平台**	**DP1000**
功能描述	模型推理 Ai 扩展引擎			
类型	**接口/关键字名称**	**类型**	**说明**	
输入(Pin)	Pin	JpegDecTask	输入对象类型	
		VideoDecTask	输入对象类型	
属性(Prop)	batch_num	int	最大批处理 batch 数目（1~32），默认为 1	
	tensor_in_one_batch	int	一个 batch 所含输入 tensor 数目（1~256），默认为 1	
	encrypt	int	模型是否加密（0-不加密，1-加密），默认为 0	
	resize_type	int	模型 resize 类型（0-正常,1-等比例），默认为 0	
	model_net_path	string	模型网络文件（net.bin）路径	
	model_param_path	string	模型参数文件（model.bin）路径	
	period	int	执行周期（1~128），默认为 1	
	copy_flag	int	模型结果是否拷贝后再上报（0-不拷贝，1-拷贝），默认为 0	
输出(Pout)	Pout[0]	NnJpegRslt	输出对象类型	
	Pout[1]	NnVideoRslt	输出对象类型	
备注	默认前后处理，可处理解码器输出			

5.4. 其他算子

5.4.1. Profile 算子

算子名称	ProfileNode		支持平台	所有平台
功能描述	用于接收其他算子的 Profile 结果			
类型	接口/关键字名称	类型	说明	
输入(Pin)	Pin	NodeProfileTask	Node Profile 结果任务，参见 de::NodeProfileTask。	
属性(Prop)	无			
输出(Pout)	无			
备注				

6. 自定义算子

DESDK 中提供了一些标准算子（即 Node）。但有时需要在标准算子（后文用 Node 代替算子）的基础上，扩展其功能。例如，标准 NodeBase 接收 TaskA 类型的输入，处理后输出 TaskB 类型。希望在此基础上扩展出 NodeExt，接收 TaskAExt 类型的输入，复用 NodeBase 的处理过程，最后输出 TaskBExt 类型。
本章介绍 Node 扩展原理和使用。

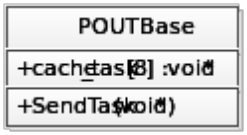
6.1. Node 扩展问题

在通过继承已有 Node 来扩展 Node 功能时，需要复用基类 Node 的 Proc 函数，来处理现有类型的 Task。但调用基类 Node 的 Proc 函数，会在最后通过 POUT::SendTask，将产生的输出发送给下一级 Node，而且这里的输出只能是基类 Node 支持的类型。这种情况不能满足扩展 Node 功能的需求，需要提供一种方法，可以取出基类 Node 的输出，对原输出做进一步加工处理，或者产生新类型的输出。

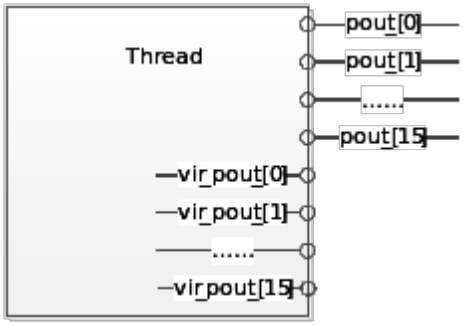
6.2. 解决方法

新增 POUT 类的基类 POUTBase。POUTBase 类内部有一个 void*类型的数组，SendTask 为

POUTBase 类的虚函数。POUTBase::SendTask 只是把输出暂存到内部的 void*数组中，而不像 POUT::SendTask 直接把输出发送给与其相连的下一个 Node 的 pin。



为 Thread 类增加成员 POUTBase* vir_pouts_[16]。Node 的 Proc 函数可以在最后通过调用 POUT::SendTask 把输出发送给下一个 Node 的 pin，也可以通过调用 POUTBase::SendTask，把输出暂存到某个 vir_pouts_的内部。后面的代码可以取出缓存的输出，做进一步处理。



Thread::Proc 函数增加第三个参数 POutType，以指示 Proc 在最后发送输出时，应该使用 pouts_，还是使用 vir_pouts_。函数原型如下：

```
virtual void Proc(void* rx_task, int32_t task_type, POutType pout_type = DEFAULT_POUT) = 0;
```

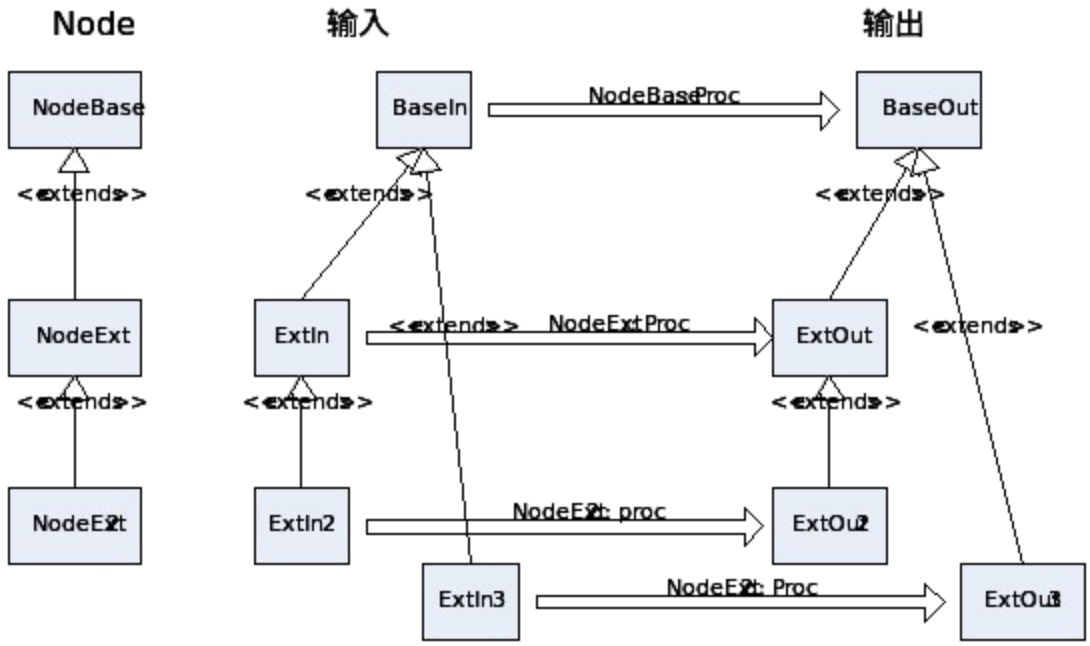
其中 POutType 类型定义如下：

```
enum POutType
{
   DEFAULT_POUT,
   USER_POUT
};
```

当 Proc 函数的参数 3 为 DEFAULT_POUT 时，表示应使用 pouts_发送输出；为 USER_POUT 时，表示应使用 vir_pouts_发送输出。该参数默认为 DEFAULT_POUT。

6.3. Node 扩展原理

以下图中的类型为例。



其中：

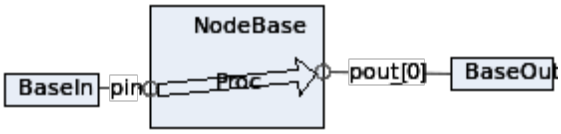
NodeBase 类可以处理 BaseIn 类型的输入任务，输出 BaseOut 类型的结果。

NodeExt 类继承于 NodeBase，新增处理 ExtIn 类型的输入任务，输出 ExtOut 类型的结果。其中，ExtIn、ExtOut 分别是对 BaseIn、BaseOut 的扩展。

NodeExt2 类继承于 NodeExt，新增处理 ExtIn2 类型的输入任务，输出 ExtOut2 类型的结果。以及新增处理 ExtIn3 类型的输入任务，输出 ExtOut3 类型的结果。其中，ExtIn2、ExtOut2 分别是对 ExtIn、ExtOut 的扩展，而 ExtIn3、ExtOut3 则是直接扩展自 BaseIn、BaseOut。

6.3.1. NodeBase 处理过程

NodeBase 的处理过程很简单，从 pin 中取出 BaseIn 类型的输出，处理得到 BaseOut 类型的输出，通过 pouts_[0]输出。



6.3.2. NodeExt 处理过程

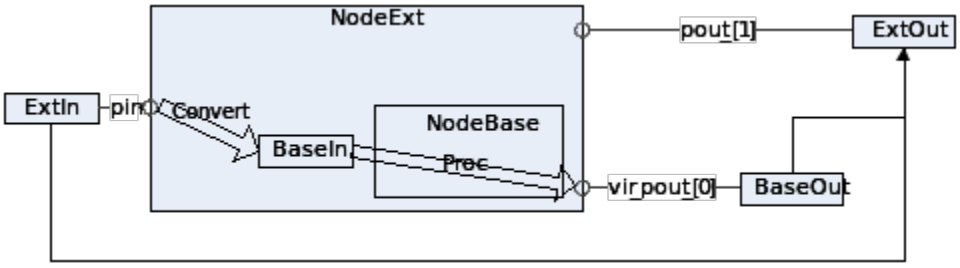
NodeExt 收到 BaseIn 类型的输入，直接交给基类处理，调用 NodeBase::Proc 即可。若收到 ExtIn 类型的输入，则按以下步骤进行处理：

1. 把 ExtIn 类型的输入，转换为 BaseIn 类型。在这个转换过程中，会丢失 ExtIn 中新增加的扩展信息
2. 把转换得到的 BaseIn 类型输入，调用 NodeBase::Proc，交由 NodeBase 处理，得到 BaseOut 类型的输出
3. 取出 NodeBase 中的 BaseOut 类型的输出，结合源输入 ExtIn，构造 ExtOut 类型的输出

4. 通过 pouts_[1]，把 ExtOut 类型的输出发送出去

上述过程中，关键在于第 2 步，调用 NodeBase::Proc 时，参数 3 使用 USER_POUT，指示 NodeBase 应将输出暂时存放于 vir_pouts_中，而不是直接通过 pouts_发送给下一个 Node 的 pin。

处理过程示意图如下：

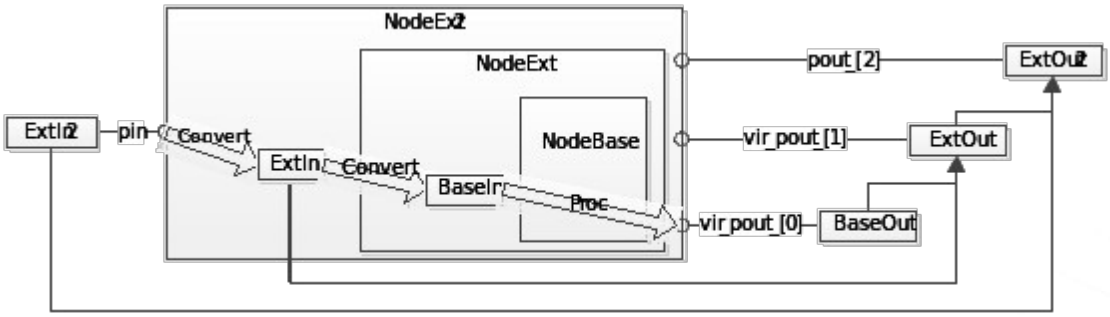


6.3.3. NodeExt2 处理过程

NodeExt2 收到 BaseIn、ExtIn 类型的输入，直接交给基类处理，调用 NodeExt::Proc 即可。若收到 ExtIn2 类型的输入，处理过程如下：

1. 把 ExtIn2 类型的输入，转换为 ExtIn 类型
2. 调用 NodeExt::Proc，处理转换后得到的 ExtIn 类型任务，得到 ExtOut 类型的输出结果。其中，参数 3 使用 USER_POUT，指示 NodeExt 应将 ExtOut 输出结果暂时存放于 vir_pouts_中
3. 取出 NodeExt 中的 ExtOut 类型的输出，结合源输入 ExtIn2，构造 ExtOut2 类型的输出
4. 通过 pouts_[2]，把 ExtOut2 类型的输出发送出去

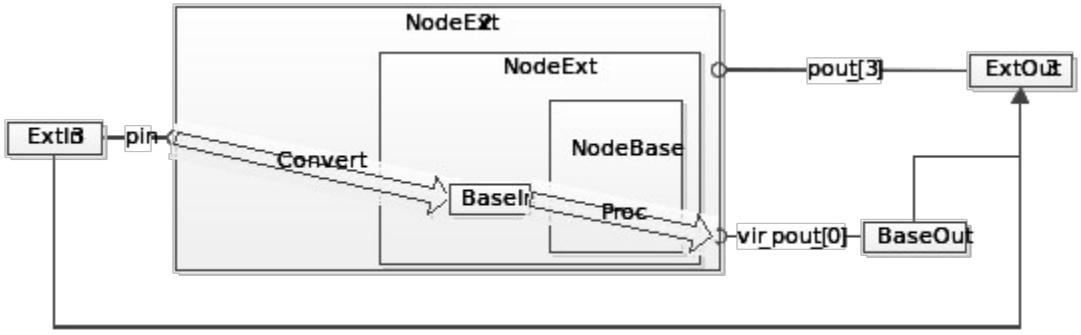
一个完整的处理过程示意图如下：



若收到 Ext3 类型的输入，处理过程如下：

1. 把 Ext3 类型的输入，转换为 BaseIn 类型
2. 调用 NodeBase::Proc，得到 BaseOut 类型的输出。其中，参数 3 使得 USER_POUT，指示 NodeBase 应将 BaseOut 输出结果暂时存放于 vir_pouts 中。
3. 取出 NodeBase 中的 BaseOut 类型的输出，结果源输入 ExtIn3，构造 ExtOut3 类型的输出。
4. 通过 pouts_[3]，把 ExtOut3 类型的输出发送出去。

一个完整的处理过程如下图所示：



6.4. 代码修改影响

修改了 Thread::Proc 函数原型，增加了第三个 POutType 类型的参数，指示 Proc 在输出结果时是使用 pouts_还是 vir_pouts_。所以目前所有 Node 的 Proc 函数声明和定义都需要修改。

6.4.1. 不需要支持扩展功能的基类 Node

Proc 处理代码不需要修改。

6.4.2. 需要支持扩展功能的基类 Node

对这类 Node，所有调用 SendTask 的地方，不能再直接使用成员 pouts_[x]。而需要根据 Proc 函数的入参是 DEFAULT_POUT 还是 USER_POUT，选择使用 pouts_[x]还是 vir_pouts_[x]。

Thread 类提供了一个接口 SelectPout，根据 POutType 类型的入参，选择使用 pouts_还是 vir_pouts_。例如：

```
virtual void Proc(void* task, int task_type, POutType pout_type = DEFAULT_POUT) override
{
   de::POUTBase** pout = SelectPout(pout_type);
     //其他操作
   BaseTaskOut* p_out_task = new BaseTaskOut;
     //如果某个类不需要被继承、扩展，此处可以直接使用成员 pouts_
   //但如果需要被继承、扩展，此处不能直接使用成员 pouts_，而应该使用局部变量 pout
     pout[0]->SendTask(p_out_task);
}
```

6.4.3. 扩展的派生 Node

建议扩展 Node 定义以下函数，按以下步骤实现扩展功能。

6.4.3.1. 定义输入 Task 转换函数

这类函数用于将扩展输入，转换为基类可以处理的输入任务。例如：

```
/**
 * @brief 把派生类的输入，转换成基类可以处理的输入
 * @param rx_task 派生类的源输入 task
 * @param rx_task_type 派生类的输入 task 类型
 * @return 基类可处理的任务指针
 */
void* PinConvert(void* rx_task, int rx_task_type)
{
    switch (rx_task_type)
   {
         case 2:
        {
           //type 2 是对 type 1 的直接扩展。转换为 type 1
           ExtTaskIn* p_ext_task = new ExtTaskIn;
           //其他操作
               return p_ext_task;
               break;
        }
         case 3:
        {
           //type 3 是对 type 0 的直接扩展。转换为 type 0
           BaseTaskIn* p_base_task = new BaseTaskIn;
           //其他操作
               return p_base_task;
               break;
        }
   }

    return nullptr;
}
```

6.4.3.2. 调用基类的 Proc 函数

可以交由基类直接处理，并且不需要对基类的输出做额外处理的任务，则直接调用基类的 Proc 函数

即可。但是如果需要对基类的输出，做进一步加工处理，则在调用基类的 Proc 函数时，参数 3 要使用 USER_POUT，指示基类的 Proc 函数将结果暂存到 vir_pouts_中，而不是直接通过 pouts_输出到下一 Node。例如：

```
virtual void Proc(void* task, int task_type, POutType pout_type = DEFAULT_POUT) override
{
   de::POUTBase** pout = SelectPout(pout_type);

    switch(task_type)
   {
        case 0:
        case 1:
    {
        //NodeExtThread 可处理 type 0 和 1 的 task
        NodeExtThread::Proc(task, task_type);
            break;
    }
    case 2: //type 2 是对 type 1 的扩展。转成 type 1 类型的任务，交由基类处理
    {
        //先转成基类可识别的类型
            void* p_base_task = PinConvert(task, task_type);
        //交由基类处理，参数 3 传参使用 USER_POUT
        NodeExtThread::Proc(p_base_task, 1, USER_POUT);

        //其他操作
            break;
    }
   }
}
```

调用基类的 Proc 函数时，参数 3 是使用默认参数还是使用 USER_POUT，关键在于需不需要对基类的输出做二次处理。例如，NodeExt 收到 ExtIn 类型的输入后，也只是想得到 BaseOut 类型的结果并直接输出，此时就可以转成 BaseIn 类型的输入，直接调用 NodeBase::Proc。而如果 NodeExt 收到 BaseIn 类型的输入后，想对 BaseOut 类型的输出做进一步处理而不是直接输出，这种情况就需要以参数 USER_POUT 作为参数 3 调用 NodeBase::Proc。

6.4.3.3. 定义并注册 vir_pouts_输出结果转换函数

示例代码如下：

```
/**
* @brief 处理基类存放在虚拟 pout0 中的输出
* @param rx_task 源输入 task
* @param rx_task_type 源输入 task 类型
* @param out_task 基类产生的输出 task
* @param pout 由父函数传递
```

```
*/
void VirPout0TaskConvert(void* rx_task, int rx_task_type, void* out_task, de::POUTBase**
pout)
{
   //代码实现
}

/**
 * @brief 处理基类存放在虚拟 pout1 中的输出
 * @param rx_task 源输入 task
 * @param rx_task_type 源输入 task 类型
 * @param out_task 基类产生的输出 task
 * @param pout 由父函数传递
 */
void VirPout1TaskConvert(void* rx_task, int rx_task_type, void* out_task, de::POUTBase**
pout)
{
   //代码实现
}

NodeExtThread2(void)
{
   VPOUT_CONVERCT_FUNC(0, NodeExtThread2, VirPout0TaskConvert);
   VPOUT_CONVERCT_FUNC(1, NodeExtThread2, VirPout1TaskConvert);
}
```

6.4.3.4. 调用 Thread 类的统一处理函数

定义并注册好 vir_pouts_输出转换函数后，可调用 Thread 类的统一函数 VirPoutTaskProc 处理。代码示例如下：

```
virtual void Proc(void* task, int task_type, POutType pout_type = DEFAULT_POUT) override
{
   de::POUTBase** pout = SelectPout(pout_type);

    switch(task_type)
   {
      //其他操作
      case 2: //type 2 是对 type 1 的扩展。转成 type 1 类型的任务，交由基类处理
      {
         //先转成基类可识别的类型
               void* p_base_task = PinConvert(task, task_type);
         //交由基类处理，参数 3 传参使用 vir_pout
         NodeExtThread::Proc(p_base_task, 1, USER_POUT);
```

```
            //调用 Thread 类的统一函数处理
                VirPoutTaskProc(task, task_type, pout);

            //根据实际使用场景决定是否要 DelTask
                pin.DelTask(task, task_type);
            break;
        }
        //其他代码
    }
}
```

6.5. 注意事项

1． 支持扩展功能的 Node，要定义 pouts_的 Deleter 函数。例如：

```
NodeBaseThread(void)
{
    pouts_[0]->SetTypeInfo("BaseTaskOut", de::TaskSerialize<BaseTaskOut>,
de::TaskDeleter<BaseTaskOut>);
}
```

2． 派生类不要覆盖基类的 pin、pouts_设置，而只应该扩展。例如：

```
NodeExtThread2(void)
{
    //注意：只能扩展，不能覆盖
    pin.SetTypeInfo(2, "ExtTaskIn2", de::TaskDeSerialize<ExtTaskIn2>,
de::TaskDeleter<ExtTaskIn2>);
    pin.SetTypeInfo(3, "ExtTaskIn3", de::TaskDeSerialize<ExtTaskIn3>,
de::TaskDeleter<ExtTaskIn3>);
    pouts_[2]->SetTypeInfo("ExtTaskOut2", de::TaskSerialize<ExtTaskOut2>,
de::TaskDeleter<ExtTaskOut2>);
    pouts_[3]->SetTypeInfo("ExtTaskOut3", de::TaskSerialize<ExtTaskOut3>,
de::TaskDeleter<ExtTaskOut3>);
}
```

3. 支持扩展功能的 Node，调用 SendTask 的地方不能直接使用成员 pouts_，而应该根据 Proc 函数入参 POutType 选择使用 pouts_还是 vir_pouts_。

4．对于非常了解基类处理过程的开发者来说，在扩展基类功能时，可以自己实现输入任务转换、vir_pouts_输出结果转换，不需要定义调用 PinConvert、VirPoutTaskProc 等。只需注意调用基类的 Proc 函数时，传递正确的 POutType 类型参数。

5．扩展 Node 的层次不要太多。因为输入、输出转换不可避免地要有 new、delete、copy 等操作，扩展层次越多越影响效率。

6.6. 不支持扩展的情况

如前所述，Node 扩展功能需要派生类的输入、基类的 Proc 函数、基类的输出三个要素。这三个要素获取不全，是无法扩展的。
例如，以下场景，无法对基类 Node 做扩展：

1. 基类的 Proc 函数为空
   这类 Node 通常不需要输入，只产生输出，重写了 Thread::Start 函数，在 Start 函数中自定义了处理函数。例如，一个 Node 负责周期性地向其连接的其他 Node 发送一个定时器消息 Task，不需要任何输入，也不需要实现、调用 Proc 函数。
   对于这类 Node，派生类即使调用它的 Proc 函数，也是一个空操作，获取不到基类输出，所以无法扩展。
2. 基类无输出
   例如，基类 Node 处理 BaseIn 类型的输入，有些条件下产生 BaseOut 的输出，有些条件下则不产生输出。派生类处理 ExtIn 的输入，希望总是产生 ExtOut 类型的输出，在输出中通过一个字段指示本次输出是否有效。
   若按照扩展的派生 Node 章节中描述的步骤实现，当基类无输出时，调用 Thread::VirPoutTaskProc 将没有效果。因为 vir_pouts_中没有输出 Task，无法完成派生类输入+基类输出的组装过程。
   但这种情况并不是绝对的无法扩展。按照前文中的模板实现，无法完成扩展；但若开发者对基类的处理过程很了解，自己实现 vir_pouts 到 pouts_ 的转换，而不是直接调用 Thread::VirPoutTaskProc，仍然能实现扩展。
3. 基类的 Proc 函数不是同步过程
   例如，在基类的 Proc 函数里，接收输入后，把中间结果暂存到一个 buffer 中。另启动了一个线程，从 buffer 中取出中间结果，再转换为输出。这种实现通常发生在需要调用硬件处理的 Node 中。
   如果对这样的 Node 做扩展，派生类调用了基类的 Proc 函数后，此时并不能保证 vir_pouts_中一定已经有输出了，所以无法取出基类的结果做进一步处理。

7. FAQ

1. Graph 析构出错，打印“Wait graph pout stop timeout”
   出现这条打印的原因是，有线程因为调用了 Graph::GetOutput 而阻塞。
   见 Graph 析构章节描述，需要在 Graph 析构前，调用 Graph::StopOutput 使阻塞线程从阻塞状态退出。