CANN
8.0.RC2.alpha001

AscendCL 应用软件开发指南(Python)

文档版本	01
发布日期	2024-04-23

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

安全声明

产品生命周期政策

华为公司对产品生命周期的规定以“产品生命周期终止政策”为准，该政策的详细内容请参见如下网址：
https://support.huawei.com/ecolumnsweb/zh/warranty-policy

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

华为预置证书权责说明

华为公司对随设备出厂的预置数字证书，发布了“华为设备预置数字证书权责说明”，该说明的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-service/ENEWS2000015766

华为企业业务最终用户许可协议(EULA)

本最终用户许可协议是最终用户（个人、公司或其他任何实体）与华为公司就华为软件的使用所缔结的协议。最终用户对华为软件的使用受本协议约束，该协议的详细内容请参见如下网址：
https://e.huawei.com/cn/about/eula

产品资料生命周期策略

华为公司针对随产品版本发布的售后客户资料（产品资料），发布了“产品资料生命周期策略”，该策略的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-website/ENEWS2000017760

目 录

1 学习向导

概述

本文用于指导开发人员基于现有模型使用pyACL（Python Ascend Computing Language）提供的Python语言API库开发深度神经网络应用，用于实现目标识别、图像分类等功能。

通过本文档您可以达成：

- 了解pyACL的功能架构、基本概念以及接口的典型调用流程。
- 使用pyACL接口进行应用开发的基本流程和实现方法。
- 能够基于本文档中的样例，扩展进行其它应用的开发。

掌握以下经验和技能可以更好地理解本文档：

- 具备Python语言程序开发能力。
- 对机器学习、深度学习有一定的了解。

文档使用建议

如果您是第一次使用本文档，或者还不清楚以下问题时，建议先从**2 快速入门**了解下应用开发的整体过程，然后了解**3 概述**，再通过**开发基础推理应用**、**图像/视频数据处理**、**单算子调用**等章节的接口调用流程与示例代码来深入学习。

- pyACL在整体架构的什么位置?
- pyACL中的Device、Stream、Context是用来做什么的?
- 使用pyACL接口开发应用时，包含哪几个基本步骤?

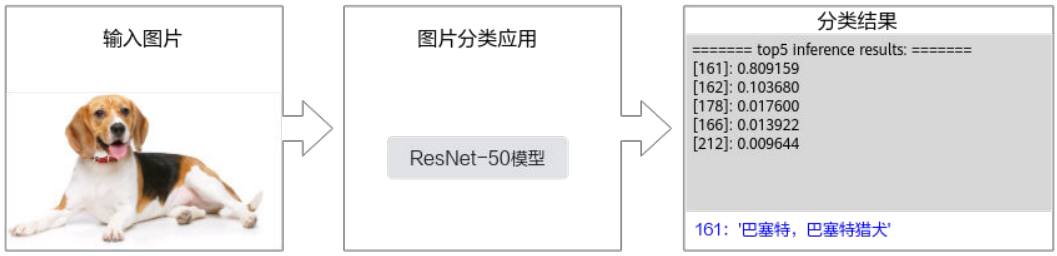
2 快速入门

在本章节中，您可以通过一个简单的图片分类应用代码示例了解使用pyACL接口（Python语言接口）开发应用的基本过程以及开发过程中涉及的关键概念。

什么是图片分类应用?

“图片分类应用”顾名思义标识图片所属的分类。

图 2-1 图片分类应用



“图片分类应用”是怎么做到这一点的呢？首先，需要有一个能做到图片分类的模型，我们可以直接使用一些训练好的开源模型，也可以基于开源模型的源码进行修改、重新训练，还可以自己基于算法、框架构建适合自己的模型。

本次快速入门样例，我们直接获取已训练好的开源模型，这种方式相对简单，此处我们选择的是ONNX框架的ResNet-50模型。

ResNet-50模型的基本介绍如下：

- 输入数据：RGB格式、224*224分辨率的输入图片。
- 输出数据：图片的类别标签及其对应置信度。

说明

- 置信度是指图片所属某个类别可能性。
- 类别标签和类别的对应关系与训练模型时使用的数据集有关，需要查阅对应数据集的标签及类别的对应关系。

前提条件

- 已在环境上部署昇腾AI软件栈。
- 安装环境，请参见**3.4 应用开发环境准备**。
- 安装必要的Python软件依赖（ Pillow、numpy ）。
  ```
  pip3 install pillow numpy
  ```

了解基本概念

- Host

  Host指与Device相连接的X86服务器、ARM服务器，会利用Device提供的NN（ Neural-Network ）计算能力，完成业务。
- Device

  Device指安装了昇腾AI处理器的硬件设备，利用PCIe接口与Host侧连接，提供NN计算能力。
- 开发环境、运行环境

  开发环境指开发代码的环境，运行环境指运行算子、推理或训练等程序的环境，运行环境上必须带昇腾AI处理器。

  说明

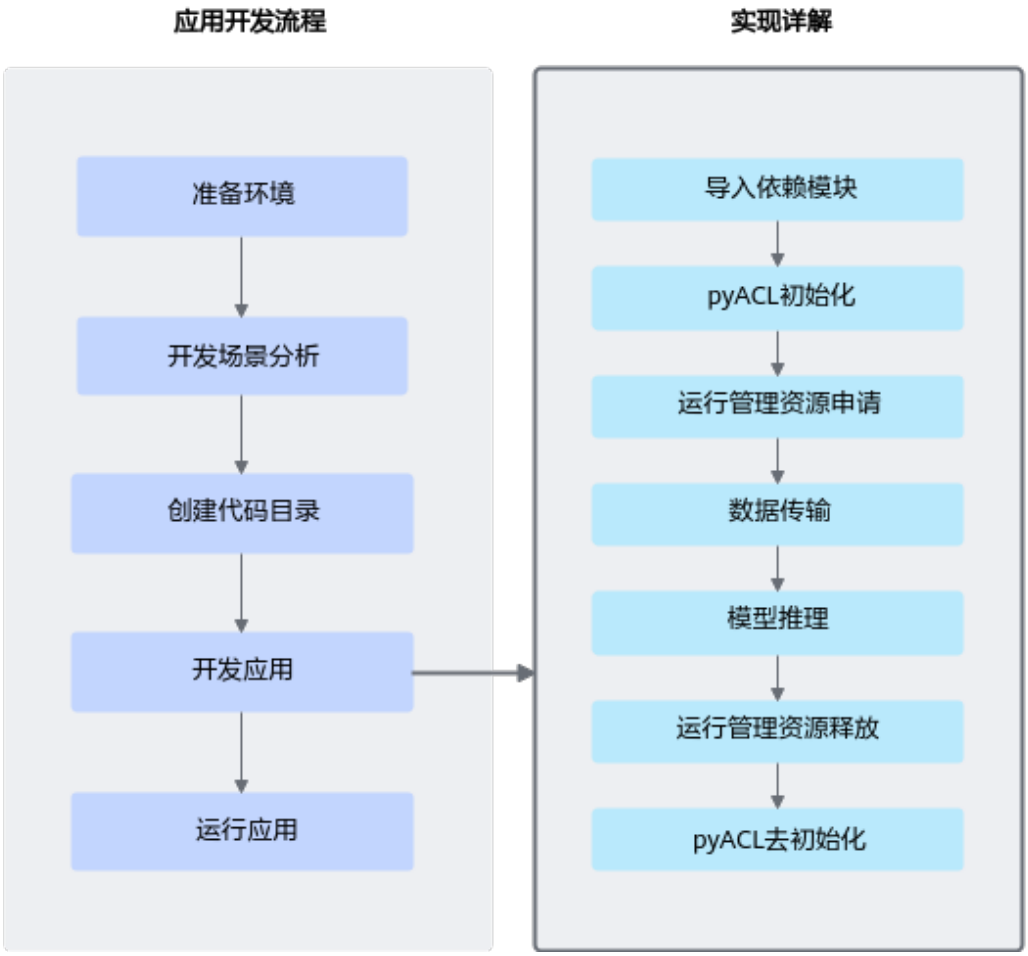
  您可以登录对应的环境，执行“arch”命令查询其操作系统的架构。
- 运行用户

  运行驱动进程、推理业务或执行训练的用户。

了解开发过程

pyACL（ Python Ascend Computing Language ）是一套在AscendCL的基础上使用CPython封装得到的Python API库，使用户可以通过Python进行昇腾AI处理器的运行管理、资源管理等，实现在昇腾CANN平台上进行深度学习推理计算、图形图像预处理、单算子加速计算等能力。

图 2-2 开发流程



了解了这些大步骤后，下面我们再展开来说明开发应用具体涉及哪些关键功能？各功能又使用哪些pyACL接口，这些pyACL接口怎么串联？

虽然此时您可能不理解所有细节，但这也不影响，通过快速入门旨在先了解整体的代码逻辑，后续再深入学习，了解其它细节。

创建代码目录

请参考以下目录结构，在开发环境中下创建“first_app”代码目录（例如“$HOME”目录）。

```
first_app
├── data
│   ├── dog1_1024_683.jpg       //测试图片1
│   └── dog2_1024_683.jpg       //测试图片2
└── model                       //用于存放ONNX ResNet-50模型文件
    └── resnet50.onnx
```

其中，需准备以下**数据**与**模型**。

- 准备测试数据，本次样例需要使用两张动物图片，请从以下链接获取，将下载好的图片上传至“first_app/data”目录。
  - **测试图片1**

    ```
    cd $HOME/first_app/data
    wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/models/aclsample/dog1_1024_683.jpg
    ```


  - **测试图片2**
    ```
    cd $HOME/first_app/data
    wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/models/aclsample/dog2_1024_683.jpg
    ```

- 准备模型数据，参考以下命令，将ONNX模型下载至“model”目录下或通过**模型获取链接**下载到本地后上传到运行环境。
  ```
  cd $HOME/first_app/model
  wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/003_Atc_Models/resnet50/resnet50.onnx
  ```
- 模型转换，对于开源框架的模型，不能直接在昇腾AI处理器上进行推理，需要使用ATC（Ascend Tensor Compiler）工具将开源框架的网络模型转换为适配昇腾AI处理器的离线模型（*.om文件）。

  执行以下命令（以Atlas 推理系列产品（Ascend 310P处理器）为例），将原始模型转换为昇腾AI处理器能识别的*.om模型文件。请注意，执行命令的用户需具有命令中相关路径的可读、可写权限。
  ```
  atc --model=resnet50.onnx --framework=5 --output=resnet50 --
  input_shape="actual_input_1:1,3,224,224"  --soc_version=Ascend310P3
  ```
  各参数的解释如下，详细约束说明请参见《ATC工具使用指南》。
  - --model：ResNet-50网络的模型文件的路径。
  - --framework：原始框架类型。5表示ONNX。
  - --output：resnet50.om模型文件的路径。请注意，记录保存该om模型文件的路径，后续开发应用时需要使用。
  - --input_shape：模型输入数据的shape。
  - --soc_version：昇腾AI处理器的版本。

    说明

    如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加“Ascend”信息，例如“Name”对应取值为“310P3”，实际配置的“soc_version”值为“Ascend310P3”。

开发应用

在“first_app”目录下创建“first_app.py”文件并依次写入以下内容。

步骤1 引入pyACL必要的模块，定义pyACL常量。
```
import os

import acl
import numpy as np
from PIL import Image

ACL_MEM_MALLOC_HUGE_FIRST = 0
ACL_MEMCPY_HOST_TO_DEVICE = 1
ACL_MEMCPY_DEVICE_TO_HOST = 2
```

步骤2 定义模型对象。

网络模型对象中应当包含以下函数。

- **初始化函数**。
- **执行推理任务函数**。
- **析构函数**。

对于后续的使用，用户只需要调用网络模型中的**forward**函数，传入对应的输入数据即可获得相应的输出。
```
class net:
```



```
#   def __init__(self, model_path):
        # 初始化函数，需要在后续步骤中实现。

#   def forward(self, inputs):
        # 执行推理任务，需要在后续步骤中实现。

#   def __del__(self):
        # 析构函数，按照初始化资源的相反顺序释放资源，需要在后续步骤中实现。
```

步骤3 实现初始化方法，具体涉及以下步骤（请在**net**类中实现）。

1. 调用**acl.init**接口进行初始化，在使用pyACL开发应用时，需要先初始化pyACL（在完成所有pyACL接口调用后，还需进行去初始化）。初始化时，也可通过JSON配置文件，向初始化接口传入配置参数（例如，传入性能相关的采集信息配置）。
2. 通过ID，调用**acl.rt.set_device**接口指定具体的计算设备（Device）。
3. 加载模型。
   a. 在此处样例选择调用**acl.mdl.load_from_file**接口加载om模型文件。
   b. 调用**acl.mdl.create_desc**接口创建模型描述信息。
   c. 根据加载成功的模型ID，调用**acl.mdl.get_desc**接口获取该模型的描述信息。
4. 创建输入数据集与输出数据集，对应方法在**步骤4**中实现。

```
def __init__(self, model_path):
    # 初始化函数
    self.device_id = 0

    # step1: 初始化
    ret = acl.init()
    # 指定运算的Device
    ret = acl.rt.set_device(self.device_id)

    # step2: 加载模型，本示例为ResNet-50模型
    # 加载离线模型文件，返回标识模型的ID
    self.model_id, ret = acl.mdl.load_from_file(model_path)
    # 创建空白模型描述信息，获取模型描述信息的指针地址
    self.model_desc = acl.mdl.create_desc()
    # 通过模型的ID，将模型的描述信息填充到model_desc
    ret = acl.mdl.get_desc(self.model_desc, self.model_id)

    # step3：创建输入输出数据集
    # 创建输入数据集
    self.input_dataset, self.input_data = self.prepare_dataset('input')
    # 创建输出数据集
    self.output_dataset, self.output_data = self.prepare_dataset('output')
```

步骤4 实现数据集创建方法（请在**net**类中实现）。

在调用pyACL接口进行模型推理时，模型推理有输入、输出数据，输入、输出数据需要按照pyACL规定的数据类型存放。相关数据类型如下：

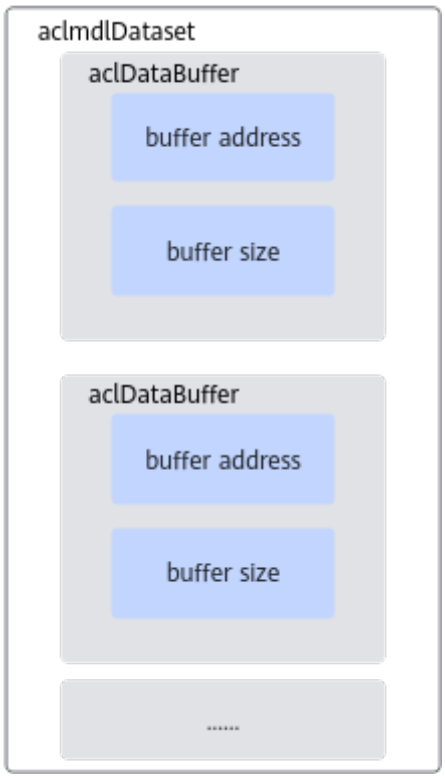
- 使用**aclmdlDesc**类型的数据描述模型基本信息（例如输入/输出的个数、名称、数据类型、Format、维度信息等）。

  模型加载成功后，用户可根据模型的ID，调用该数据类型下的操作接口获取该模型的描述信息，进而从模型的描述信息中获取模型输入/输出的个数、内存大小、维度信息、Format、数据类型等信息。
- 使用**aclDataBuffer**类型的数据来描述**每个输入/输出的内存地址、内存大小**。

  调用aclDataBuffer类型下的操作接口获取内存地址、内存大小等，便于向内存中存放输入数据、获取输出数据。

- 使用**aclmdlDataset**类型的数据描述模型的输入/输出数据。
  模型可能存在多个输入、多个输出，调用aclmdlDataset类型的操作接口添加多个aclDataBuffer类型的数据。

  图 2-3 aclmdlDataset 类型与 aclDataBuffer 类型的关系




```
def prepare_dataset(self, io_type):
    # 准备数据集
    if io_type == "input":
        # 获得模型输入的个数
        io_num = acl.mdl.get_num_inputs(self.model_desc)
        acl_mdl_get_size_by_index = acl.mdl.get_input_size_by_index
    else:
        # 获得模型输出的个数
        io_num = acl.mdl.get_num_outputs(self.model_desc)
        acl_mdl_get_size_by_index = acl.mdl.get_output_size_by_index
    # 创建aclmdlDataset类型的数据，描述模型推理的输入。
    dataset = acl.mdl.create_dataset()
    datas = []
    for i in range(io_num):
        # 获取所需的buffer内存大小
        buffer_size = acl_mdl_get_size_by_index(self.model_desc, i)
        # 申请buffer内存
        buffer, ret = acl.rt.malloc(buffer_size, ACL_MEM_MALLOC_HUGE_FIRST)
        # 从内存创建buffer数据
        data_buffer = acl.create_data_buffer(buffer, buffer_size)
        # 将buffer数据添加到数据集
        _, ret = acl.mdl.add_dataset_buffer(dataset, data_buffer)
        datas.append({"buffer": buffer, "data": data_buffer, "size": buffer_size})
    return dataset, datas
```

步骤5 实现同步推理方法（请在**net**类中实现）。

```
def forward(self, inputs):
    # 执行推理任务
    # 遍历所有输入，拷贝到对应的buffer内存中
    input_num = len(inputs)
```



```
    for i in range(input_num):
        bytes_data = inputs[i].tobytes()
        bytes_ptr = acl.util.bytes_to_ptr(bytes_data)
        # 将图片数据从Host传输到Device。
        ret = acl.rt.memcpy(self.input_data[i]["buffer"],  # 目标地址 device
                            self.input_data[i]["size"],    # 目标地址大小
                            bytes_ptr,                     # 源地址 host
                            len(bytes_data),               # 源地址大小
                            ACL_MEMCPY_HOST_TO_DEVICE)     # 模式:从host到device
    # 执行模型推理。
    ret = acl.mdl.execute(self.model_id, self.input_dataset, self.output_dataset)
    # 处理模型推理的输出数据，输出top5置信度的类别编号。
    inference_result = []
    for i, item in enumerate(self.output_data):
        buffer_host, ret = acl.rt.malloc_host(self.output_data[i]["size"])
        # 将推理输出数据从Device传输到Host。
        ret = acl.rt.memcpy(buffer_host,                   # 目标地址 host
                            self.output_data[i]["size"],   # 目标地址大小
                            self.output_data[i]["buffer"], # 源地址 device
                            self.output_data[i]["size"],   # 源地址大小
                            ACL_MEMCPY_DEVICE_TO_HOST)     # 模式：从device到host
        # 从内存地址获取bytes对象
        bytes_out = acl.util.ptr_to_bytes(buffer_host, self.output_data[i]["size"])
        # 按照float32格式将数据转为numpy数组
        data = np.frombuffer(bytes_out, dtype=np.float32)
        inference_result.append(data)
    vals = np.array(inference_result).flatten()
    # 对结果进行softmax转换
    vals = np.exp(vals)
    vals = vals / np.sum(vals)

    return vals
```

步骤6 实现析构方法（请在**net**类中实现）。

1. 销毁数据集资源（buffer数据、buffer内存、输入数据集、输出数据集）。
2. 销毁模型描述、卸载模型。
3. 释放计算资源。
4. 所有pyACL接口调用结束后（或在进程退出前），调用**acl.finalize**接口进行pyACL进行**去初始化**。

在推理过程中可能会抛出异常，请将资源释放步骤实现在析构方法中确保资源能够得到正确释放。以下内容仅供参考，实际情况下需要考虑更多情况下的资源释放问题。

```
def __del__(self):
    # 析构函数 按照初始化资源的相反顺序释放资源。
    # 销毁输入输出数据集
    for dataset in [self.input_data, self.output_data]:
        while dataset:
            item = dataset.pop()
            ret = acl.destroy_data_buffer(item["data"])    # 销毁buffer数据
            ret = acl.rt.free(item["buffer"])              # 释放buffer内存
    ret = acl.mdl.destroy_dataset(self.input_dataset)      # 销毁输入数据集
    ret = acl.mdl.destroy_dataset(self.output_dataset)     # 销毁输出数据集
    # 销毁模型描述
    ret = acl.mdl.destroy_desc(self.model_desc)
    # 卸载模型
    ret = acl.mdl.unload(self.model_id)
    # 释放device
    ret = acl.rt.reset_device(self.device_id)
    # acl去初始化
    ret = acl.finalize()
```

步骤7 实现图像预处理函数。

```
def transfer_pic(input_path):
    # 图像预处理
    input_path = os.path.abspath(input_path)
```



```
    with Image.open(input_path) as image_file:
        # 缩放为224*224
        img = image_file.resize((224, 224))
        # 转换为float32类型ndarray
        img = np.array(img).astype(np.float32)
    # 根据imageNet图片的均值和方差对图片像素进行归一化
    img -= [123.675, 116.28, 103.53]
    img /= [58.395, 57.12, 57.375]
    # RGB通道交换顺序为BGR
    img = img[:, :, ::-1]
    # resnet50为色彩通道在前
    img = img.transpose((2, 0, 1))
    # 返回并添加batch通道
    return np.array([img])
```

步骤8 调用**forward**函数（具体实现请参见**步骤5**），执行同步推理并在屏幕中打印**top5类别编号**及**置信度**。

```
def print_top_5(data):
    top_5 = data.argsort()[::-1][:5]
    print("======== top5 inference results: ==============")
    for j in top_5:
        print("[%d]: %f" % (j, data[j]))

if __name__ == "__main__":
    resnet50 = net('./model/resnet50.om')
    image_paths = ["./data/dog1_1024_683.jpg", "./data/dog2_1024_683.jpg"]
    for path in image_paths:
        # 图像预处理，此处仅供参考，用户按照自己需求进行预处理
        image = transfer_pic(path)
        # 将数据按照每个输入的顺序构造list传入，当前示例的ResNet-50模型只有一个输入
        result = resnet50.forward([image])
        # 输出top_5
        print_top_5(result)

    del resnet50
```

----结束

运行应用

将编写好的“first_app”文件夹及内容上传到运行环境，进入到代码目录下，检查环境变量配置是否正确，然后执行以下命令。

```
python3 first_app.py
```

可以得到如下输出，分别为两张测试图片的top5分类信息。

其中**[161]: 0.809159**表示的是类别标识索引“161”的置信度为“0.809159”。

```
======== top5 inference results: ==============
[161]: 0.809159
[162]: 0.103680
[178]: 0.017600
[166]: 0.013922
[212]: 0.009644
======== top5 inference results: ==============
[267]: 0.728299
[266]: 0.101693
[265]: 0.100117
[151]: 0.004214
[160]: 0.002721
```


说明

类别标签和类别的对应关系与训练模型时使用的数据集有关，本样例使用的模型是基于imagenet数据集进行训练的，您可以在互联网上查阅对应数据集的标签及类别的对应关系。

当前屏显信息中的类别标识与类别的对应关系如下：

"161": ["basset", "basset hound"]

"162": ["beagle"]

"163": ["bloodhound", "sleuthhound"]

"166": ["Walker hound", "Walker foxhound"]

"167": ["English foxhound"]

3 概述

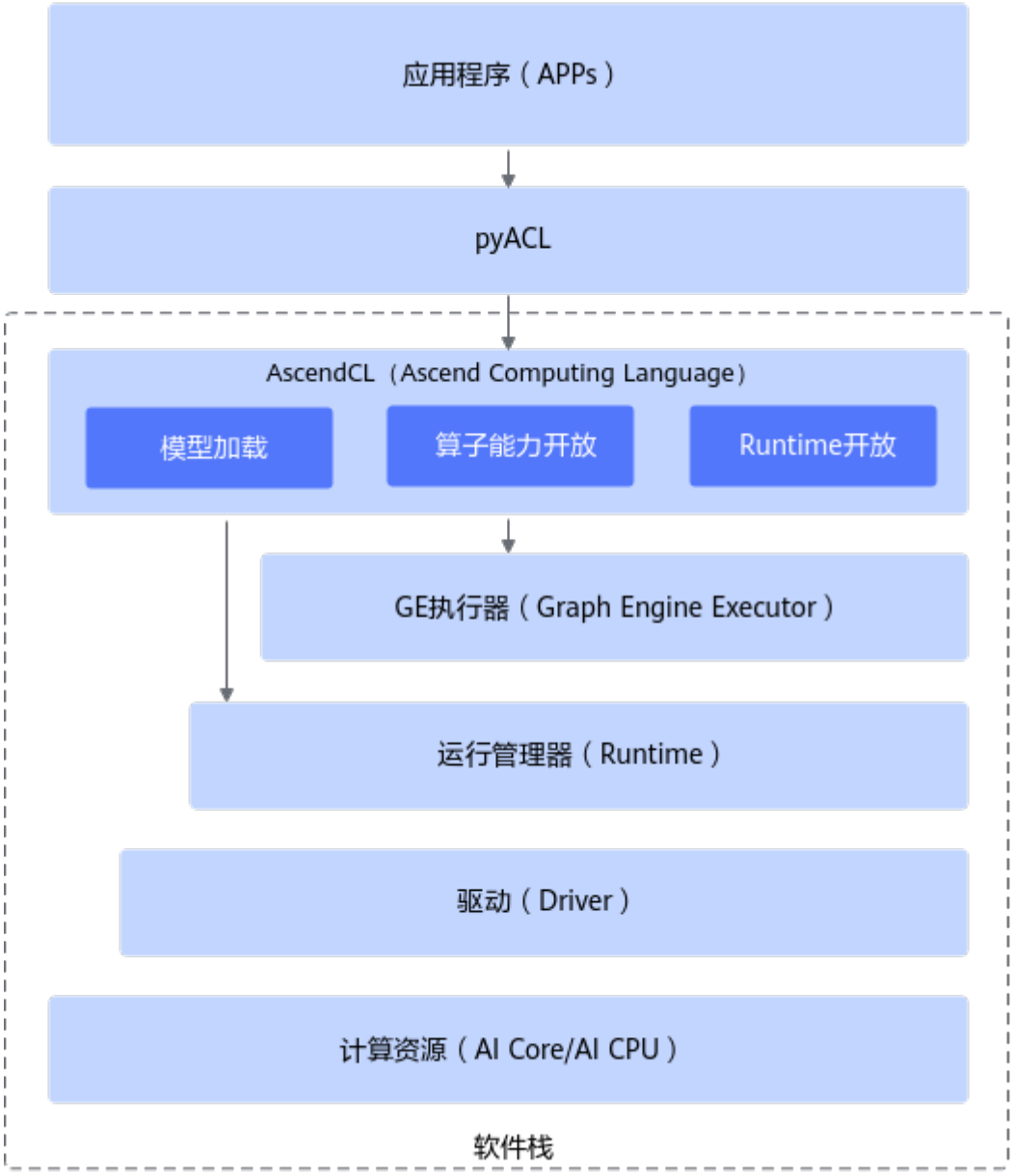
3.1 基于 AscendCL 的 pyACL

AscendCL（Ascend Computing Language）是一套用于在昇腾平台上开发深度神经网络应用的C语言API库，提供运行资源管理、内存管理、模型加载与执行、算子加载与执行、媒体数据处理等API，能够实现利用昇腾硬件计算资源、在昇腾CANN平台上进行**深度学习推理计算**、**图形图像预处理**、**单算子加速计算**等能力。简单来说，就是统一的API框架，实现对所有资源的调用。

计算资源层是昇腾AI处理器的硬件算力基础，主要完成神经网络的矩阵相关计算、完成控制算子/标量/向量等通用计算和执行控制功能、完成图像和视频数据的预处理，为深度神经网络计算提供了执行上的保障。

pyACL（Python Ascend Computing Language）就是在AscendCL的基础上使用CPython封装得到的Python API库，使用户可以通过Python进行昇腾AI处理器的运行管理、资源管理等。

图 3-1 逻辑架构图



3.2 基本概念

表 3-1 概念介绍

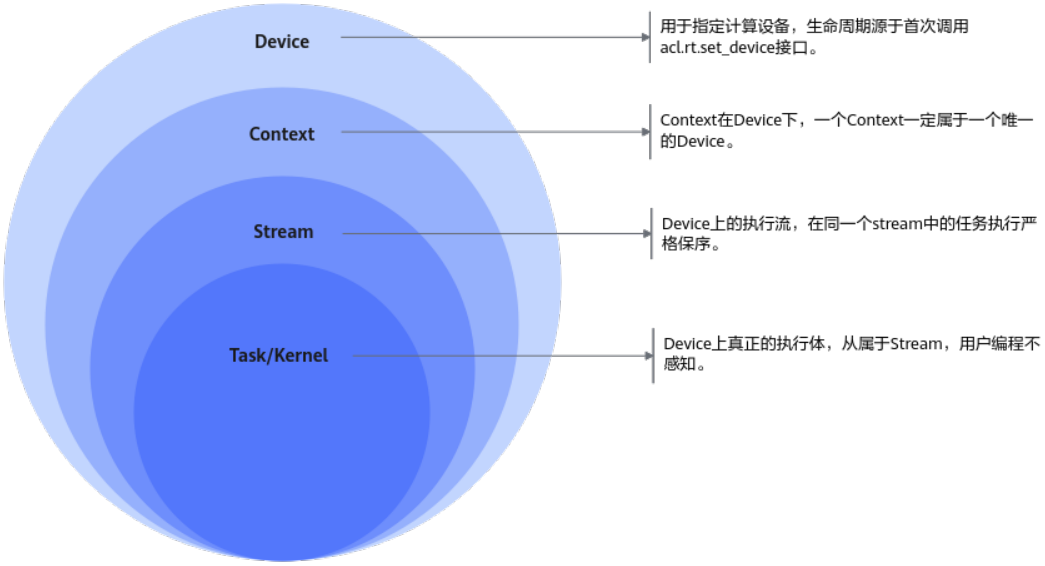
概念	描述
同步/异步	本文中提及的同步、异步是站在调用者和执行者的角度，在当前场景下，若在Host调用接口后不等待Device执行完成再返回，则表示Host的调度是异步的；若在Host调用接口后需等待Device执行完成再返回，则表示Host的调度是同步的。
进程/线程	本文中提及的进程、线程，若无特别注明，则表示Host上的进程、线程。

概念	描述
Host	Host指与Device相连接的X86服务器、ARM服务器，会利用Device提供的NN（ Neural-Network ）计算能力，完成业务。
Device	Device指安装了昇腾AI处理器的硬件设备，利用PCIe接口与Host侧连接，为Host提供NN计算能力。若存在多个Device，多个Device之间的内存资源不能共享。
Context	Context作为一个容器，管理了所有对象（包括Stream、Event、设备内存等）的生命周期。不同Context的Stream、不同Context的Event是完全隔离的，无法建立同步等待关系。 Context分为两种： ● 默认Context：调用acl.rt.set_device接口指定用于运算的Device时，系统会自动隐式创建一个默认Context，一个Device对应一个默认Context，默认Context不能通过acl.rt.destroy_context接口来释放。 ● 显式创建的Context：**推荐**，在进程或线程中调用acl.rt.create_context接口显式创建一个Context。
Stream	Stream用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在Device上执行。 基于Stream的kernel执行和数据传输能够实现Host运算操作、Host与Device间的数据传输、Device内的运算并行。 Stream分两种： ● 默认Stream：调用acl.rt.set_device接口指定用于运算的Device时，系统会自动隐式创建一个默认Stream，一个Device对应一个默认Stream，默认Stream不能通过acl.rt.destroy_stream接口来释放。 ● 显式创建的Stream：**推荐**，在进程或线程中调用acl.rt.create_stream接口显式创建一个Stream。
Event	支持调用pyACL接口同步Stream之间的任务，包括同步Host与Device之间的任务、同一个Device上的多个任务。 例如，若stream2的任务依赖stream1的任务，想保证stream1中的任务先完成，这时可创建一个Event，并将Event插入到stream1，在执行stream2的任务前，先同步等待Event完成。

概念	描述
AIPP	AIPP（Artificial Intelligence Pre-Processing）用于在AI Core上完成图像预处理，包括色域转换（转换图像格式）、图像归一化（减均值/乘系数）和抠图（指定抠图起始点，抠出神经网络需要大小的图片）等。 AIPP区分为静态AIPP和动态AIPP。您只能选择静态AIPP或动态AIPP其中一种方式来处理图片，不能同时配置静态AIPP和动态AIPP两种方式。 ● 静态AIPP：模型转换时设置AIPP模式为静态，同时设置AIPP参数，模型生成后，AIPP参数值被保存在离线模型（*.om）中，每次模型推理过程采用固定的AIPP预处理参数（无法修改）。 如果使用静态AIPP方式，多Batch情况下共用同一份AIPP参数。 ● 动态AIPP：模型转换时仅设置AIPP模式为动态，每次模型推理前，根据需求，在执行模型前设置动态AIPP参数值，然后在模型执行时可使用不同的AIPP参数。 如果使用动态AIPP方式，多Batch可使用不同的AIPP参数。
动态Batch/动态分辨率	在某些场景下，模型每次输入的batch size或分辨率是不固定的，如检测出目标后再执行目标识别网络，由于目标个数不固定导致目标识别网络输入BatchSize不固定。 ● 动态Batch：用户执行推理时，其batch size是动态可变的。 ● 动态分辨率：用户执行推理时，每张图片的分辨率H*W是动态可变的。
动态维度（ND格式）	为了支持Transformer等网络在输入格式的维度不确定的场景，需要支持ND格式下任意维度的动态设置。 ND表示支持任意格式，当前N ≤ 4。
通道	在RGB色彩模式下，图像通道就是指单独的红色R、绿色G、蓝色B部分。也就是说，一幅完整的图像，是由红色、绿色、蓝色三个通道组成的，它们共同作用产生了完整的图像。同样在HSV色系中指的是色调H、饱和度S、亮度V三个通道。
标准形态	指Device做为EP，通过PCIe配合主设备（X86、ARM等各种服务器）进行工作，此时Device上的CPU资源仅能通过Host调用，相关推理应用程序运行在Host。Device只为服务器提供NN计算能力。
EP模式	以昇腾 AI 处理器的PCIe的工作模式进行区分，如果PCIe工作在从模式，则称为EP模式。
RC模式	以昇腾 AI 处理器的PCIe的工作模式进行区分，如果PCIe工作在主模式，可以扩展外设，则称为RC模式。

Device、Context、Stream 之间的关系

图 3-2 Device、Context、Stream 之间的关系



- **Device**，用于指定计算设备。
  - Device的生命周期源于首次调用acl.rt.set_device接口。
  - 每次调用acl.rt.set_device接口，系统会进行引用计数**加1**；调用acl.rt.reset_device接口，系统会进行引用计数**减1**。
  - 当引用计数减为**零**时，在本进程中Device上的资源不可用。
- **Context**，在Device下，一个Context一定属于一个唯一的Device。
  - Context分**隐式创建**和**显式创建**。
  - **隐式创建**的Context（即默认Context），生命周期始于调用acl.rt.set_device接口，**终结于**调用acl.rt.reset_device接口使引用**计数为零**时。

    隐式Context只会被创建一次，调用acl.rt.set_device接口重复指定同一个Device，只增加隐式创建的Context的引用计数。
  - **显式创建**的Context，生命周期始于调用acl.rt.create_context接口，终结于调用acl.rt.destroy_context接口。
  - 若在某一进程内创建多个Context（Context的数量与Stream相关，Stream数量有限制，请参见acl.rt.create_stream），当前线程在同一时刻内只能使用其中一个Context，建议通过acl.rt.set_context接口明确指定当前线程的Context，增加程序的可维护性。
  - 进程内的Context是共享的，可以通过acl.rt.set_context进行切换。
- **Stream**，是Device上的执行流，在同一个Stream中的任务执行严格保序。
  - Stream分**隐式创建**和**显式创建**。
  - 每个Context都会包含一个默认Stream，属于隐式创建，隐式创建的Stream生命周期同归属的Context。
  - 用户可以显式创建Stream，显式创建的Stream生命周期始于调用acl.rt.create_stream，终结于调用acl.rt.destroy_stream接口。显式创建的Stream归属的Context被销毁或生命周期结束后，会影响该Stream的使用，虽然此时Stream没有被销毁，但不可再用。

- **Task/Kernel**，是Device上真正的任务执行体。

线程、Context、Stream 之间的关系

- 一个用户线程一定会绑定一个Context，所有Device的资源使用或调度，都必须基于Context。
- 一个线程中当前会有一个唯一的Context在用，Context中已经关联了本线程要使用的Device。
- 可以通过acl.rt.set_context进行Device的快速切换。示例代码如下，仅供参考，不可以直接拷贝运行：

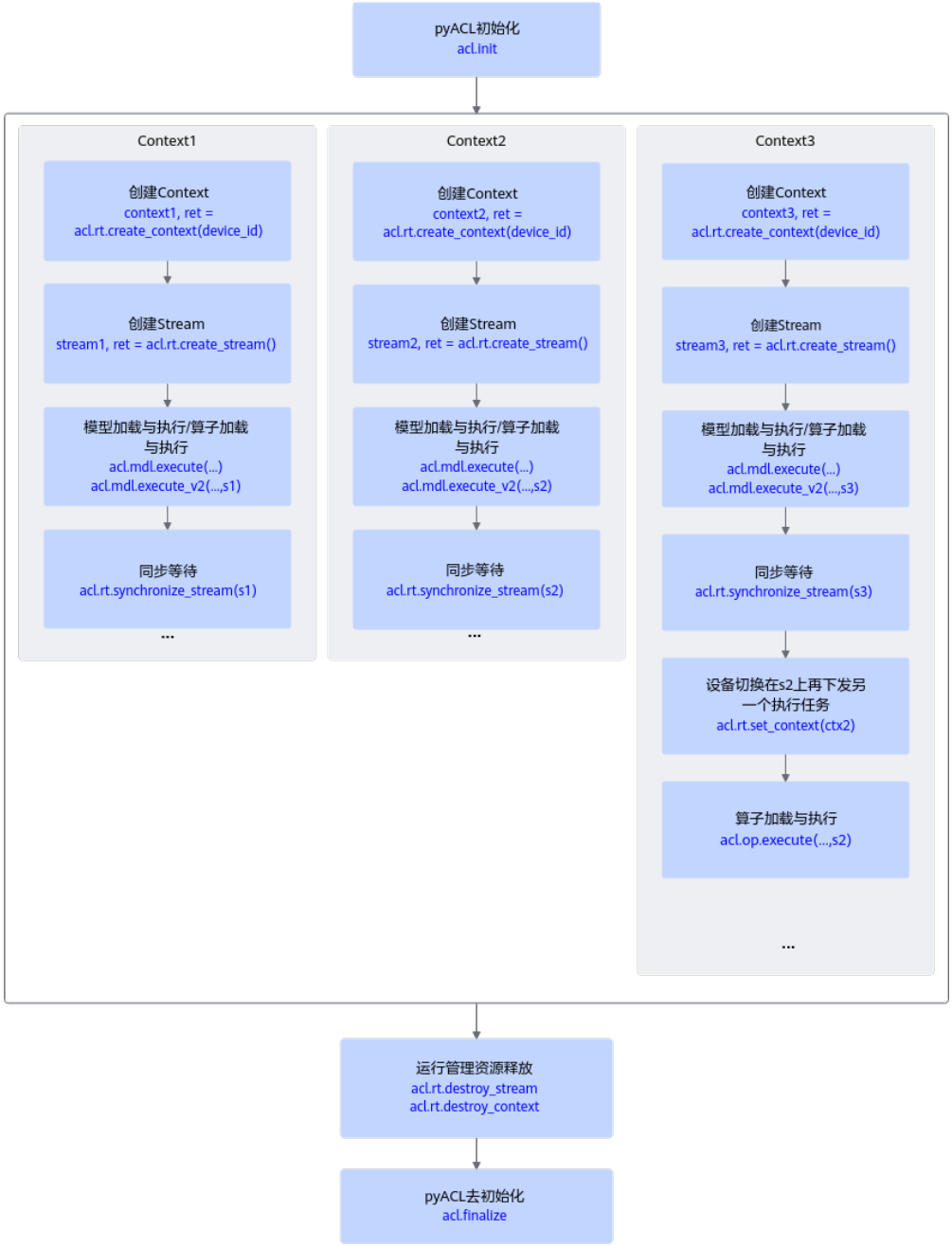
```
...
ctx1, ret = acl.rt.create_context(0)        #使用acl.rt.create_context接口通过传入Device Id创建Context。
stream, ret = acl.rt.create_stream()
ret = acl.op.execute_v2(op_type, input_desc, inputs, output_desc, outputs, attr, stream)
ctx2, ret = acl.rt.create_context(1)

# 在当前线程中，创建ctx2后，当前线程对应的Context切换为ctx2，对应在Device 1进行后续的计算任务，本例中将在Device 1上进行op2的执行调用。
stream2, ret = acl.rt.create_stream()
ret = acl.op.execute_v2(op_type2, input_desc, inputs, output_desc, outputs, attr, stream2)
ret = acl.rt.set_context(ctx1);

# 在当前线程中，通过Context切换，使后续计算任务在对应的Device 0上进行。
ret = acl.op.execute_v2(op3,...,s1)
...
```

- 一个线程中可以创建多个Stream，不同的Stream上计算任务是可以并行执行，多线程场景下，也可以每个线程创建一个Stream，线程之间的Stream在Device上相互独立，每个Stream内部的任务是按照Stream下发的顺序执行。
- 多线程的调度依赖于运行应用的操作系统调度，多Stream调度Device侧，由Device上调度组件进行调度。

一个进程内多个线程间的 Context 迁移

- 一个进程中可以创建多个Context，但一个线程同一时刻只能使用一个Context。
- 线程中创建的多个Context，线程缺省使用最后一次创建的Context。
- 进程内创建的多个Context，可以通过acl.rt.set_context设置当前需要使用的Context。

图 3-3 接口调用流程



默认 Context 和默认 Stream 的使用场景

- Device上执行操作下发前，必须有Context和Stream，这个Context、Stream可以显式创建，也可以隐式创建。**隐式创建**的Context、Stream就是默认Context、默认Stream。

  默认Stream作为接口入参时，直接传0。

- **默认Context**不允许用户执行acl.rt.get_context或acl.rt.set_context操作，也不允许执行acl.rt.destroy_context操作。
- **默认Context**、**默认Stream**一般适用于简单应用，用户仅仅需要一个Device的计算场景下。多线程应用程序建议全部使用显式创建的Context和Stream。

示例代码如下，仅供参考，不可以直接拷贝运行：

```
# …
ret = acl.init(config_path)
ret = acl.rt.set_device(device_id)

# 已经创建了一个default ctx，在default ctx中创建了一个default stream，并且在当前线程可用。
# …
ret = acl.op.execute_v2(op1, input_desc, inputs, output_desc, outputs, attr, 0)  # 最后一个0表示在default stream上执行算子op1。
ret = acl.op.execute_v2(op2, input_desc, inputs, output_desc, outputs, attr, 0)  # 最后一个0表示在default stream上执行算子op2。
ret = acl.rt.synchronize_stream(0)

# 等待计算任务全部完成（op1、op2执行结束），用户根据需要获取计算任务的输出结果。
# …
ret = acl.rt.reset_device(device_id)  # 释放计算设备0，对应的default ctx及default stream生命周期也终止。
```

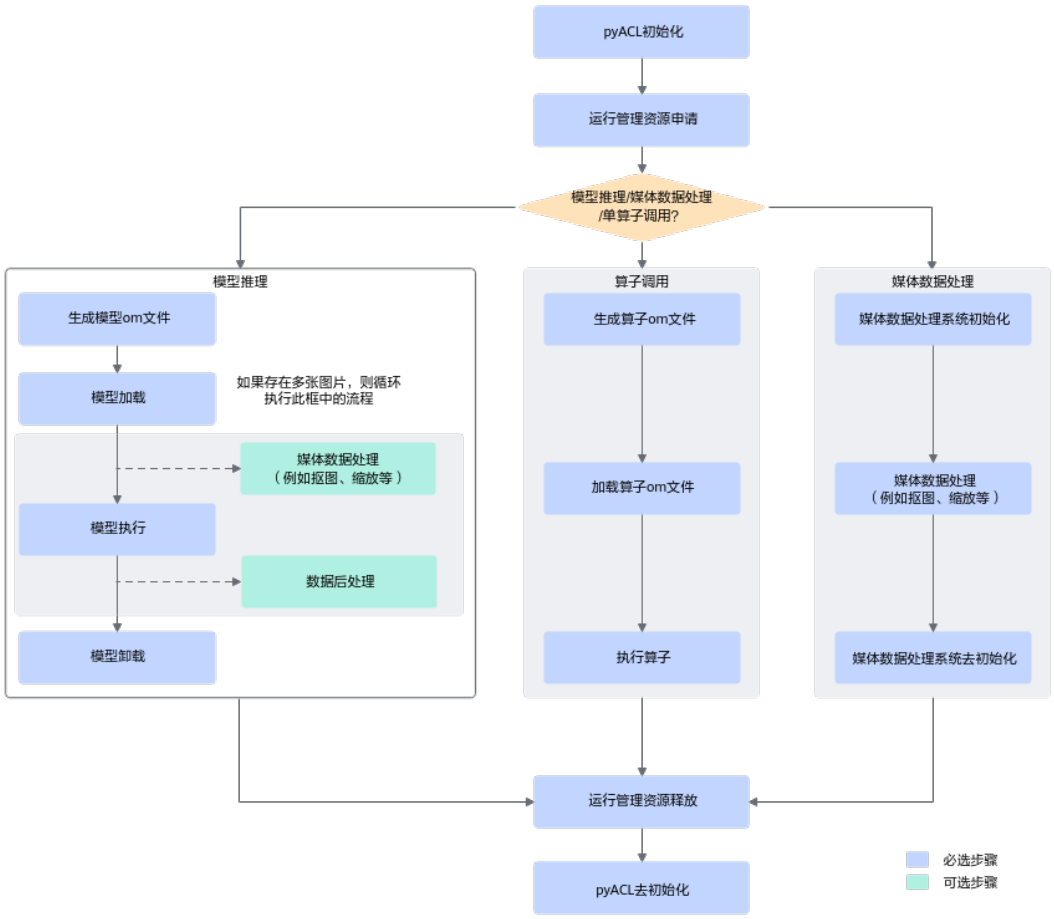
多线程、多 stream 的性能说明

- 线程调度依赖运行的操作系统，Stream上下发了任务后，Stream的调度由Device的调度单元调度，但如果一个进程内的多Stream上的任务在Device存在资源争抢的时候，性能可能会比单Stream低。
- 当前昇腾AI处理器有不同的执行部件，如AI Core、AI CPU、Vector Core等，对应使用不同执行部件的任务，建议多Stream的创建按照算子执行引擎划分。
- 单线程多Stream与多线程多Stream（一个进程中可以包含多个线程，每个线程中一个Stream）性能上哪个更优，具体取决于应用本身的逻辑实现，一般来说前者性能略好，原因是相对后者，应用层少了线程调度开销。

3.3 pyACL 接口调用流程

调用pyACL接口，可开发包含模型推理、媒体数据处理、单算子调用等功能的应用，这些功能可以独立存在，也可以组合存在。下图给出了使用pyACL接口开发AI应用的整体接口调用流程。

图 3-4 接口调用流程图



上图根据应用开发中的典型功能抽象出主要的接口调用流程，具体场景参考如下。

- 如果模型对输入图片的宽高要求与用户提供的源图不一致，则需要媒体数据处理，将源图裁剪成符合模型的要求。
- 如果需要实现模型推理的功能，则需要先加载模型，模型推理结束后，则需要卸载模型。
- 如果模型推理后，需要从推理结果中查找最大置信度的类别标识对图片分类，则需要数据后处理。

接口调用流程各步骤操作参见如下。

1. pyACL初始化。

   调用acl.init接口实现初始化pyACL。

2. 运行管理资源申请。

   依次申请运行管理资源：**Device**、**Context**、**Stream**。

   具体流程，请参见**运行管理资源申请流程**。

3. 模型推理/单算子调用/媒体数据处理。

   - **模型推理**。

     i. 生成模型om文件：模型推理场景下，必须要有适配昇腾AI处理器的离线模型，需提前构建模型，请参见**4.2 模型构建**。

ii. 模型加载：模型推理前，需要先将对应的模型加载到系统中。

接口调用流程，请参见**4.6.1 模型加载**。

iii. （可选）**媒体数据处理**：可实现JPEG图片解码、视频解码、抠图/图片缩放/格式转换、JPEG图片编码等功能。

接口调用流程，请参见**5 媒体数据处理**。

iv. 模型执行：使用模型实现图片分类、目标识别等功能。

接口调用流程，请参见**4.6.2 模型执行**。

v. （可选）数据后处理：处理模型推理的结果，此处根据用户的实际需求来处理推理结果，例如用户可以将获取到的推理结果写入文件、从推理结果中找到每张图片最大置信度的类别标识等。

vi. 模型卸载：调用acl.mdl.unload接口卸载模型。

- **算子调用**。

  如果AI应用中不仅仅包括模型推理，还有数学运算（例如BLAS基础线性代数运算）、数据类型转换等功能，也想使用昇腾的算力，直接通过pyACL接口加载并执行单个算子，省去模型构建、训练的过程，相对轻量级，又可以使用昇腾的算力。另外，自定义的算子，也可以通过单算子调用的方式来验证算子的功能。接口调用流程，请参见**6.3 接口调用流程**。

4. 运行管理资源释放。

   所有数据处理都结束后，需要依次释放运行管理资源：**Stream**、**Context**、**Device**。

   具体流程，请参见**运行管理资源释放流程**。

5. pyACL去初始化。

   调用acl.finalize接口实现pyACL去初始化。

说明

在应用开发过程中，各环节都涉及内存的申请与释放、数据传输（通过内存复制实现）、数据类型的创建与销毁，因此未在图中一一标识，关于内存申请与释放、内存复制的接口请参见内存管理。数据类型的创建与销毁的接口请参见数据类型及其操作接口。

3.4 应用开发环境准备

部署开发环境和运行环境，请参见《CANN软件安装指南》对应Atlas产品的描述。

- 部署开发环境后，才能获取调用接口所需的头文件、运行接口所需的库文件。

  对于昇腾设备，已安装驱动、固件场景下，该环境可直接作为运行环境，执行生成的应用可执行文件。

- 部署运行环境后，才能在运行环境上执行Python代码文件。

  说明

  - pyACL库文件路径：CANN软件安装后文件存储路径/lib64

    需要根据运行环境的安装包，确定引用的组件目录，否则会导致运行报错。安装方案请参见《CANN软件安装指南》。

    安装CANN软件后，需要以CANN运行用户登录环境，执行**source** *${install_path}*/**set_env.sh**命令设置环境变量，其中${install_path}为CANN软件的安装目录。

  - 本文中的操作步骤需以运行用户登录开发环境或运行环境后再执行，**请务必**获取各组件的运行用户，以便后续操作时使用。

- （可选）通过环境变量“ASCEND_RT_VISIBLE_DEVICES”设置Device ID，指定应用进程可用的Device。支持一次指定一个或多个Device ID。通过设置该环境变量，可以实现不修改应用程序、但调整Device的功能。

  示例场景：例如板端环境上的可用Device数量为8，Device ID分别为：0、1、2、3、4、5、6、7。

  - 指定一个Device ID，示例表示应用进程可使用Device ID为1的Device。
    ```
    # acl.rt.get_device_count接口获取到的可用Device数量为1，acl.rt.set_device(0)时，索引0对应的Device ID是1
    export ASCEND_RT_VISIBLE_DEVICES = 1
    ```
  - 指定多个Device ID，以下两个均示例表示应用进程可使用Device ID为2、3、4的Device， Device ID的顺序可以任意设置，但顺序会影响Device ID的索引值。
    ```
    # acl.rt.get_device_count接口获取到的可用Device数量为3，acl.rt.set_device(0)时，索引0对应的Device ID是2
    export ASCEND_RT_VISIBLE_DEVICES = 2,3,4
    # acl.rt.get_device_count接口获取到的可用Device数量为3，acl.rt.set_device(0)时，索引0对应的Device ID是4
    export ASCEND_RT_VISIBLE_DEVICES = 4,3,2
    ```
  - 指定多个Device ID，但出现无效值时，则仅无效值之前的Device可用。示例中仅“2”和“3”可用。
    ```
    # acl.rt.get_device_count接口获取到的可用Device数量为2，acl.rt.set_device(0)时，索引0对应的Device ID是2
    export ASCEND_RT_VISIBLE_DEVICES = 2,3,-1,5
    ```

  通过**export**命令，设置环境变量只在当前终端窗口生效，且只对设置环境变量之后启动的昇腾AI应用进程生效。

  若将**export**命令写入“~/.bashrc”文件，使环境变量永久生效，则环境变量对该用户下的所有昇腾AI应用进程都生效。这种方式，可能会影响其它不需要调整Device ID的应用进程，**请谨慎使用**。

  将**export**命令写入“~/.bashrc”文件的方法如下：

  a. 以安装用户在任意目录下执行**vi ~/.bashrc**，在该文件最后添加上述内容。

  b. 执行**:wq!**命令保存文件并退出。

  c. 执行**source ~/.bashrc**使环境变量生效。

- （可选）通过环境变量“ASCEND_CACHE_PATH”、“ASCEND_WORK_PATH”设置pyACL应用运行过程中产生的文件的落盘路径，涉及ATC模型转换、AOE模型智能调优、性能数据采集、日志采集等功能，落盘文件包括知识库文件、调优结果文件、性能数据文件、日志文件等。

  配置示例如下，详细配置说明请参见《环境变量参考》：

  ```
  export ASCEND_CACHE_PATH=/repo/task001/cache
  export ASCEND_WORK_PATH=/repo/task001/172.16.1.12_01_03
  ```

pyACL 的依赖

pyACL没有安装依赖，但是有运行依赖。详情请参见《CANN软件安装指南》完成对开发环境和运行环境的部署。

安装后的环境变量设置

在安装完CANN软件包之后，请务必自行配置以下环境变量，否则，将无法正常使用“import acl”。

- 若环境中安装了cann-toolkit软件包：
  ```
  # 以root用户安装toolkit包。
  . /usr/local/Ascend/ascend-toolkit/set_env.sh
  ```



```
# 以非root用户安装toolkit包。
. ${HOME}/Ascend/ascend-toolkit/set_env.sh
```

- 若环境中安装了cann-nnrt软件包：

```
# 以root用户安装nnrt包。
. /usr/local/Ascend/nnrt/set_env.sh
# 以非root用户安装nnrt包。
. ${HOME}/Ascend/nnrt/set_env.sh
```

- 若环境中安装了cann-nnae软件包：

```
# 以root用户安装nnae包。
. /usr/local/Ascend/nnae/set_env.sh
# 以非root用户安装nnae包。
. ${HOME}/Ascend/nnae/set_env.sh
```

设置完环境变量后，在Python脚本中加入“import acl”，就可以使用pyACL中的函数了。

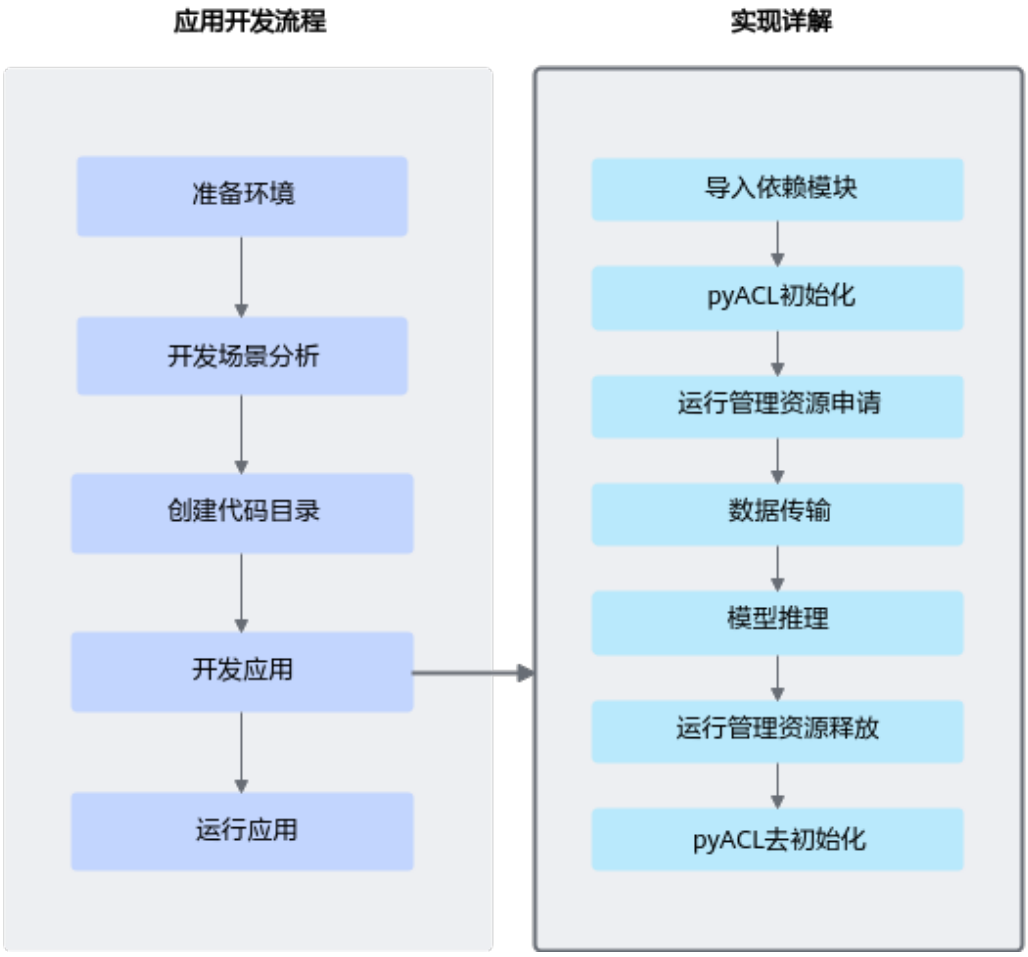
其它

- 关于日志的处理机制和日志级别设置等功能，请参见《日志参考》。
- pyACL部分功能可能会涉及到ATC工具来进行模型转换，请参见《ATC工具使用指南》。
- 生成pyACL中的“acl.so”依赖的Python版本范围为3.7.5~3.9.2。

4 开发基础推理应用

4.1 开发流程

图 4-1 开发流程



1. 准备环境，包括开发环境和运行环境。
2. 创建代码目录。

   在开发应用前，您需要先创建目录，存放代码文件、编译脚本、测试图片数据、模型文件等。如下仅是示例，可参考：

   ```
   ├App名称
   ├── caffe_model      # 该目录下存放模型转换相关的配置文件、模型文件。
   │   ├── xxx.cfg
   │   ├── xxx.prototxt
   ├── data
   │   ├── xxx.jpg     # 测试数据。
   │
   ├── model
   │   ├── xxx.om      # 转换后的模型文件。
   │
   ├── xxx.py          # python脚本。
   ├── xxx.py
   ```

3. 开发应用。

   a. pyACL初始化，请参见**4.3 pyACL初始化与去初始化**。

使用pyACL接口开发应用时，必须先调用acl.init接口进行pyACL初始化，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常。

b. 运行管理资源申请，请参见**4.4 运行管理资源申请与释放**。

c. 数据传输，请参见**4.5 数据传输**。

d. 执行模型推理。请参见**4.6 模型推理基本场景**。

模型推理结束后，需及时释放相关资源。

若需要处理模型推理的结果，还需要进行数据后处理，例如对于图片分类应用，通过数据后处理从推理结果中查找最大置信度的类别标识。

e. 所有数据处理结束后，需及时释放运行管理资源，请参见**4.4 运行管理资源申请与释放**。

f. 执行pyACL去初始化，请参见**4.3 pyACL初始化与去初始化**。

4. 运行应用，包括模型转换、运行应用，请参见**8 应用调试**。

4.2 模型构建

对于开源框架的网络模型（如Caffe、TensorFlow等），不能直接在昇腾AI处理器上做推理，需要先使用ATC（Ascend Tensor Compiler）工具将开源框架的网络模型转换为适配昇腾AI处理器的离线模型（*.om文件）。

此处以ONNX框架的ResNet-50网络为例，说明如何使用ATC工具进行模型转换，详细说明请参见《ATC工具使用指南》。

步骤1 以运行用户登录开发环境。

步骤2 执行模型转换。

执行以下命令，将原始模型转换为昇腾AI处理器能识别的*.om模型文件。请注意，执行命令的用户需具有命令中相关路径的可读、可写权限。以下命令中的“***<SAMPLE_DIR>***”请根据实际样例包的存放目录替换。***<soc_version>***请根据实际昇腾AI处理器的版本进行替换。

```
cd <SAMPLE_DIR>/MyFirstApp_ONNX/model
wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/003_Atc_Models/resnet50/resnet50.onnx
atc --model=resnet50.onnx --framework=5 --output=resnet50 --input_shape="actual_input_1:1,3,224,224" --soc_version=<soc_version>
```

各参数的解释如下，详细约束说明请参见《ATC工具使用指南》。

- --model：ResNet-50网络的模型文件的路径。
- --framework：原始框架类型。5表示ONNX。
- --output：resnet50.om模型文件的路径。请注意，记录保存该om模型文件的路径，后续开发应用时需要使用。
- --input_shape：模型输入数据的shape。
- --soc_version：昇腾AI处理器的版本。

说明

如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤3 （后续处理）如果想快速体验直接使用转换后的om离线模型文件进行推理，请准备好环境、om模型文件、符合模型输入要求的*.bin格式的输入数据，单击**Link**，获取**msame工具**，参考该工具配套的README，进行体验。

----结束

说明

- 如果模型转换时，提示有不支持的算子，请先参见《TBE&AI CPU自定义算子开发指南》先完成自定义算子，再重新转换模型。
- 如果模型转换时，提示有算子编译相关问题，但根据报错信息无法定位问题、需要联系华为工程师时（单击**Link**后新建Issue），则需设置DUMP_GE_GRAPH、DUMP_GRAPH_LEVEL环境变量，再重新模型转换，收集模型转换过程中各个阶段的图描述信息，提供给华为工程师定位问题。关于环境变量以及图描述信息的说明，请参见《ATC工具使用指南》中的“参考>dump图详细信息”。
- 如果模型的输入Shape是动态，关于模型构建、模型推理的说明请参见**7.6 模型动态推理**。
- 如果现有网络不满足您的需求，您可以使用昇腾AI处理器支持的算子、调用Ascend Graph接口自行构建自己的网络，再编译成om离线模型文件。详细说明请参见《Ascend Graph开发指南》。

4.3 pyACL 初始化与去初始化

关于pyACL初始化与去初始化的接口调用流程，请参见**3.3 pyACL接口调用流程**。

基本原理

您必须调用acl.init接口**初始化pyACL**，配置文件内容为JSON格式。

如果当前的默认配置已满足需求，无需修改，acl.init接口中可不传入参数，或者可将配置文件配置为空JSON串（即配置文件中只有{}）。在acl.init接口中不传入参数，示例如下：

```
ret = acl.init()
```

有初始化就有去初始化，在确定完成了pyACL的所有调用之后，或者进程退出之前，需调用acl.finalize接口实现pyACL**去初始化**。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
# 初始化基本配置。
# 此处的“..”表示相对路径，相对可执行文件所在的目录。
acl_config_path = "../src/acl.json"
ret = acl.init(acl_config_path)
# ......

# 去初始化。
ret = acl.finalize()
# ......
```


4.4 运行管理资源申请与释放

开发应用时，应用程序中必须包含运行管理资源申请的代码逻辑，关于运行管理资源申请的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的资源申请、释放流程说明。

基本原理

您需要按顺序**依次申请**如下运行管理资源：Device、Context、Stream，确保可以使用这些资源执行运算、管理任务。所有数据处理都结束后，需要按顺序依次释放运行管理资源：Stream、Context、Device。

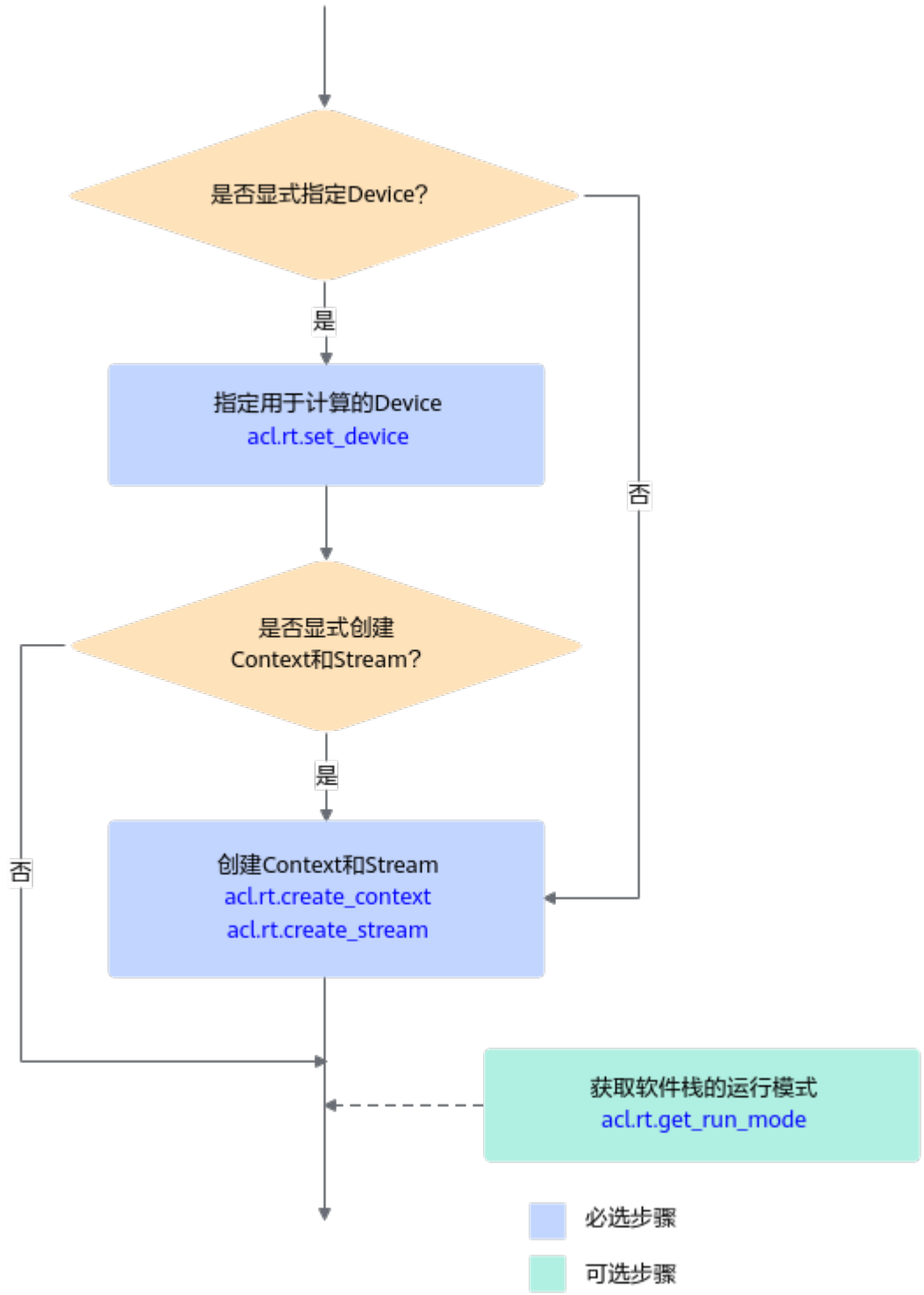
您需要按照Device、Context、Stream的顺序依次申请。其中，创建Context、Stream的方式分为隐式创建和显式创建，其适用场景有所不同：

- **隐式创建**Context和Stream：适合简单、无复杂交互逻辑的应用，但缺点在于，在多线程编程中，每个线程都使用默认Context或默认Stream，默认Stream中任务的执行顺序取决于操作系统线程调度的顺序。
- **显式创建**Context和Stream：**推荐显式**，适合大型、复杂交互逻辑的应用，且便于提高程序的可读性、可维护性。
- 关于单进程、单线程、单Stream场景如下所示：
  - 单进程：一个应用程序对应一个进程。
  - 单线程：不创建多个线程时，默认只有一个线程。
  - 单Stream：整个开发的过程中使用同一个Stream。

    对于同一个Stream中的异步任务，pyACL会按照应用程序中任务的顺序执行任务，确保异步任务执行的顺序。
- 关于多线程、多Stream的场景请参见**7.8 Stream管理**。

运行管理资源申请流程

图 4-2 运行管理资源申请流程



关键接口的说明如下：

1. **申请运行管理资源**时，需按顺序依次申请：Device、Context、Stream。

- 调用acl.rt.set_device接口**显式指定用于运算的Device**。
  - 调用acl.rt.create_context接口**显式创建Context**，调用acl.rt.create_stream接口**显式创建Stream**。
  - 如果不显式创建Context和Stream，您可以使用acl.rt.set_device接口**隐式创建的默认Context和默认Stream**，但默认Context和默认Stream存在如下限制：
    - 一个Device对应一个默认Context，默认Context不能通过acl.rt.destroy_context接口来释放。
    - 一个Device对应一个默认Stream，默认Stream不能通过acl.rt.destroy_stream接口来释放。默认Stream作为接口入参时，直接传0。
    - 默认Context、默认Stream，是在调用acl.rt.reset_device接口后自动释放。
  - **隐式指定用于运算的Device**。

    调用acl.rt.create_context接口显式创建Context，调用acl.rt.create_stream接口显式创建Stream。系统在显式创建Context时，系统内部会调用acl.rt.set_device接口指定运行的Device，Device ID通过acl.rt.create_context接口传入。

2. （可选）调用acl.rt.get_run_mode接口获取软件栈的运行模式，根据运行模式来判断后续的内存申请接口调用逻辑。

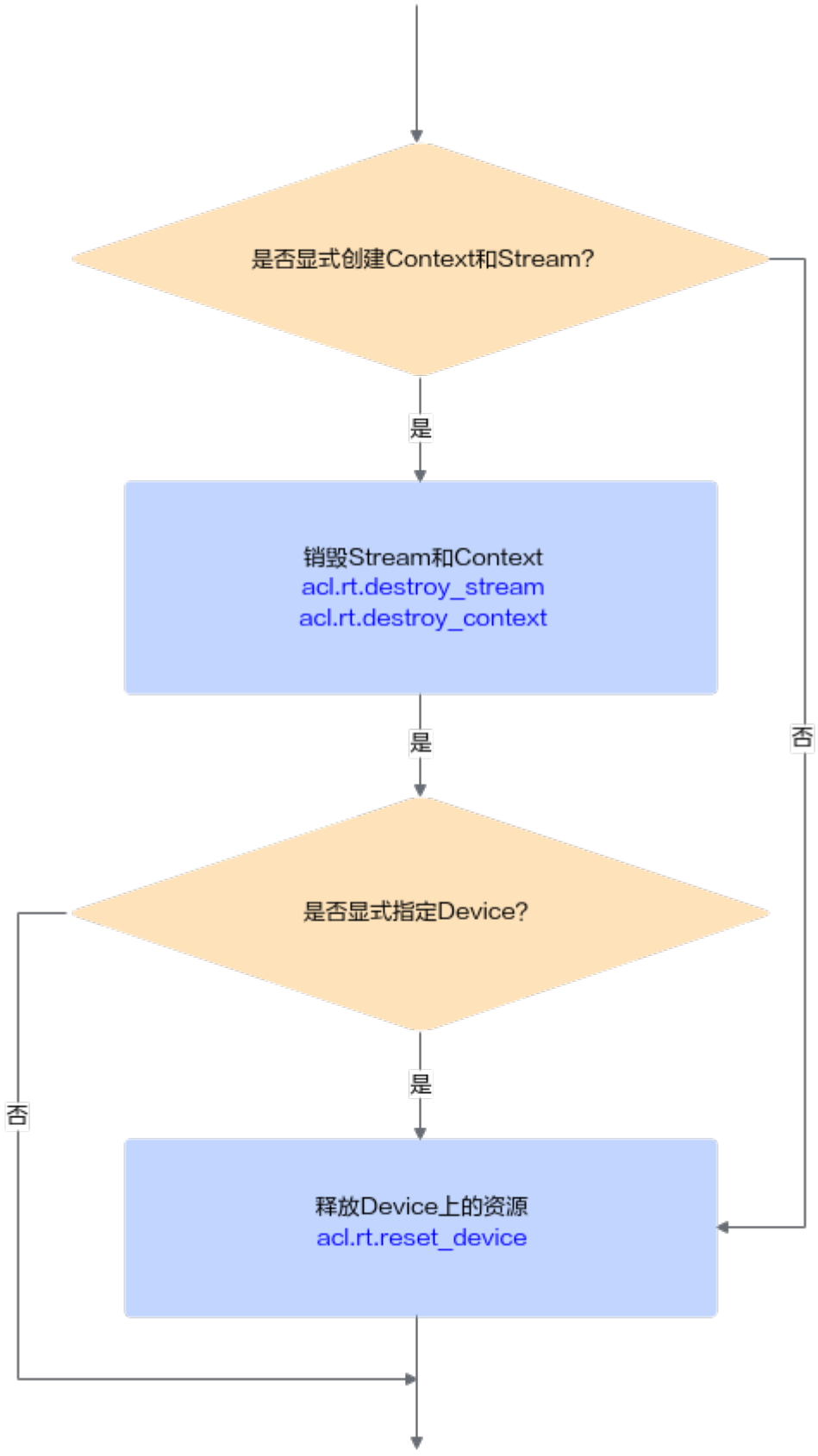
   如果查询结果为ACL_HOST，则数据传输时涉及申请Host上的内存。

   如果查询结果为ACL_DEVICE，则数据传输时仅需申请Device上的内存。

   数据传输的详细介绍请参见**4.5 数据传输**。

运行管理资源释放流程

图 4-3 运行管理资源释放流程



关键接口的说明如下：

- 释放运行管理资源时，需按顺序依次释放：Stream、Context、Device。
- 显式创建Context和Stream时，需调用acl.rt.destroy_stream接口释放Stream，再调用acl.rt.destroy_context接口释放Context。若显式调用acl.rt.set_device接口指定运算的Device时，还需调用acl.rt.reset_device接口释放Device上的资源。
- 不显式创建Context和Stream时，仅需调用acl.rt.reset_device接口释放Device上的资源。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
#......

# ======运行管理资源申请======
# 1.指定运算的Device。
ret = acl.rt.set_device(device_id)

# 2.显式创建一个Context，用于管理Stream对象。
context, ret = acl.rt.create_context(device_id)

# 3.显式创建一个Stream。
#用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序执行任务。
stream, ret = acl.rt.create_stream()
# ======运行管理资源申请======

#......

# ======运行管理资源释放======
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(device_id)
# ======运行管理资源释放======

#......
```

4.5 数据传输

4.5.1 接口调用流程

数据传输的关键接口调用流程如下：

1. **申请内存**。
   - Host上的内存，可以使用pyACL提供的acl.rt.malloc_host接口申请内存。
   - Device上的内存，使用pyACL提供的acl.rt.malloc或acl.rt.malloc_host接口申请内存。如果涉及数据预处理（例如，图片解码、缩放等）时，需使用acl.media.dvpp_malloc或acl.himpi.dvpp_malloc接口申请内存。
2. **将数据读入内存**。

   由用户自行管理数据读入内存的实现逻辑。
3. 通过内存复制实现**数据传输**。

   数据传输可以通过内存复制的方式实现，分为**同步内存复制**、**异步内存复制**：
   - 同步内存复制：调用acl.rt.memcpy接口。

- 异步内存复制：调用acl.rt.memcpy_async接口，再调用acl.rt.synchronize_stream接口实现Stream内任务的同步等待。
- 对于Host内的数据传输、Device内的数据传输、Host与Device之间的数据传输，可以调用内存复制的接口实现。
- 调用同步或异步内存复制接口时，支持以下类型的复制（可点击链接查看对应示例代码）：
  - **Host内的内存复制**
  - **Host到Device的内存复制**
  - **Device到Host的内存复制**
  - **一个Deviece内的内存复制**

说明

Ascend RC场景下，不涉及Host上的内存申请、Host内的数据传输、Host与Device之间的数据传输。

如果当前版本支持多种**运行形态**，在这种场景下，若想实现相同的应用程序可支持在多种形态下运行，申请内存的方式不同，会影响数据传输时调用的接口：

- 若应用程序中区分申请Host内存或Device内存的接口，例如使用acl.rt.malloc_host接口申请Host内存、使用acl.rt.malloc接口申请Device内存时：

  需先调用acl.rt.get_run_mode接口获取软件栈的运行模式。

  - 当查询结果为**ACL_HOST = 1**，则数据传输时涉及申请Host上的内存。
  - 当查询结果为**ACL_DEVICE = 0** ，则数据传输时不涉及申请Host上的内存，仅需申请Device上的内存。

  该种方式多一些代码逻辑的判断，不需要由用户处理Device上的内存对齐。在Device上运行应用的场景，该种方式少一些内存复制的步骤，性能较好。
- 若应用程序中不区分申请Host内存或Device内存的接口，统一使用acl.rt.malloc_host接口（该接口支持申请Host或Device内存），pyACL内部会根据软件栈的运行模式自行判断运行时申请的是Host内存还是Device内存：

  无需调用acl.rt.get_run_mode接口获取软件栈的运行模式。该种方式代码逻辑相比前一种简单，但需由用户处理Device上的内存对齐。

4.5.2 Host 内的数据传输

当前支持调用acl.rt.memcpy接口执行同步Host内的内存复制任务，不支持调用acl.rt.memcpy_async接口执行异步Host内的内存复制功能，若调用acl.rt.memcpy_async接口时选择**ACL_MEMCPY_HOST_TO_HOST**类型时，由于是异步接口，虽然接口调用成功，下发了内存复制任务，但在调用acl.rt.synchronize_stream接口等待该任务执行时会返回失败。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
host_ptr_a, ret = acl.rt.malloc_host(size)
host_ptr_b, ret = acl.rt.malloc_host(size)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, host_ptr_a, size)
```



```
# 3.同步内存复制。
# 同步内存复制，host_ptr_a表示Host上源内存地址的指针地址，host_ptr_b表示Host上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_HOST_TO_HOST = 0
ret = acl.rt.memcpy(host_ptr_b, size, host_ptr_a, size, ACL_MEMCPY_HOST_TO_HOST)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.free_host(host_ptr_a)
ret = acl.rt.free_host(host_ptr_b)

# ......
```

4.5.3 从 Host 到 Device 的数据传输

同步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
host_ptr_a, ret = acl.rt.malloc_host(size)
dev_ptr_b, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, host_ptr_a, size)

# 3.同步内存复制。
# 同步内存复制，host_ptr_a表示Host上源内存地址的指针地址，dev_ptr_b表示Device上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_HOST_TO_DEVICE = 1
ret = acl.rt.memcpy(dev_ptr_b, size, host_ptr_a, size, ACL_MEMCPY_HOST_TO_DEVICE)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.free_host(host_ptr_a)
ret = acl.rt.free(dev_ptr_b)

# ......
```

异步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
# 异步内存复制要求，内存首地址64字节对齐，使用acl.rt.malloc_host需多申请64字节。
host_ptr_a, ret = acl.rt.malloc_host(size + 64)
# host申请的内存需要用户自己64对齐处理。
host_align = host_ptr_a + 64 - host_ptr_a % 64
# acl.rt.malloc 申请的Device 侧内存系统保证已经符合64对齐。
dev_ptr_b, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, host_align, size)

# 3.异步内存复制。
# 异步内存复制，host_align表示Host上源内存地址的指针地址，dev_ptr_b表示Device上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_HOST_TO_DEVICE = 1。
stream = acl.rt_create_stream()
```



```
ret = acl.rt.memcpy_async(dev_ptr_b, size , host_align, size, ACL_MEMCPY_HOST_TO_DEVICE, stream)
ret = acl.rt.synchronize_stream(stream)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.free_host(host_ptr_a)
ret = acl.rt.free(dev_ptr_b)

# ......
```

4.5.4 Device 内的数据传输

同步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
dev_ptr_a, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_ptr_b, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, dev_ptr_a, size)

# 3.同步内存复制。
# 同步内存复制，dev_ptr_a表示Device上源内存地址的指针地址，dev_ptr_b表示Device上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_DEVICE_TO_DEVICE = 3。
ret = acl.rt.memcpy(dev_ptr_b, size, dev_ptr_a, size, ACL_MEMCPY_DEVICE_TO_DEVICE)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.free(dev_ptr_a)
ret = acl.rt.free(dev_ptr_b)

# ......
```

异步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
dev_ptr_a, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_ptr_b, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, dev_ptr_a, size)

# 3.异步内存复制。
# 异步内存复制，dev_ptr_a表示Device上源内存地址的指针地址，dev_ptr_b表示Device上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_DEVICE_TO_DEVICE = 3。
stream = acl.rt_create_stream()
ret = acl.rt.memcpy_async(dev_ptr_b, size, dev_ptr_a, size, ACL_MEMCPY_DEVICE_TO_DEVICE, stream)
ret = acl.rt.synchronize_stream(stream)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.destroy_stream(stream)
```



```
ret = acl.rt.free(dev_ptr_a)
ret = acl.rt.free(dev_ptr_b)

# ......
```

4.5.5 从 Device 到 Host 的数据传输

同步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
host_ptr_b, ret = acl.rt.malloc_host(size)
dev_ptr_a, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, dev_ptr_a, size)

# 3.同步内存复制。
# 同步内存复制，dev_ptr_a表示Device上源内存地址的指针地址，host_ptr_b表示Host上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_DEVICE_TO_HOST = 2。
ret = acl.rt.memcpy(host_ptr_b, size, dev_ptr_a, size, ACL_MEMCPY_DEVICE_TO_HOST)

# 4.使用完内存中的数据后，需及时释放资源。
ret  = acl.rt.free_host(host_ptr_b)
ret = acl.rt.free(dev_ptr_a)

# ......
```

异步内存复制

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，示例代码如下：

```
import acl
# ......

# 1.申请内存。
size = 1 * 1024 * 1024
# acl.rt.malloc 申请的Device侧内存系统保证已经符合64对齐。
dev_ptr_a, ret = acl.rt.malloc(size, ACL_MEM_MALLOC_NORMAL_ONLY)
# 异步内存复制要求，内存首地址64字节对齐，使用acl.rt.malloc_host 需多申请64字节。
host_ptr_b, ret = acl.rt.malloc_host(size + 64)
# host申请的内存需要用户自己64对齐处理。
host_align = host_ptr_b+ 64 - host_ptr_b % 64

# 2.申请内存后，可向内存中读入数据，该自定义函数read_file由用户实现。
read_file(fileName, dev_ptr_a, size)

# 3.异步内存复制。
# 异步内存复制，dev_ptr_a表示Device上源内存地址的指针地址，host_align 表示Host上目的内存地址的指针地址，size表示内存大小。
# ACL_MEMCPY_DEVICE_TO_HOST = 2。
stream = acl.rt_create_stream()
ret = acl.rt.memcpy_async(host_align, size , dev_ptr_a, size, ACL_MEMCPY_DEVICE_TO_HOST, stream)
ret = acl.rt.synchronize_stream(stream)

# 4.使用完内存中的数据后，需及时释放资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.free_host(host_ptr_b)
ret = acl.rt.free(dev_ptr_a)
```



```
# ......
```

4.6 模型推理基本场景

4.6.1 模型加载

从4.2 模型构建中的说明构建出模型后，需进行模型加载，为模型执行做准备。

接口调用流程

开发应用时，如果涉及整网模型推理，则应用程序中必须包含模型加载的代码逻辑，关于模型加载的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明。

本节描述的是**整网模型加载**的接口调用流程，对于算子模型加载与执行的详细说明请参见6 单算子调用。

pyACL提供**两套模型加载的接口**，用户可根据具体使用场景选择对应的模型加载接口：

- 如图4-4所示，根据不同的加载方式（从文件加载、从内存加载等）选择不同的接口，**操作相对简单，但需要记住各种方式的加载接口**。
- 如图4-5所示，针对不同的加载方式（从文件加载、从内存加载等），**只需设置接口中的配置参数**，适用各种加载方式，**但涉及多个接口配合使用**，分别用于创建配置对象、设置对象中的属性值、加载模型。

图 4-4 模型加载流程（通过不同接口区分加载方式）

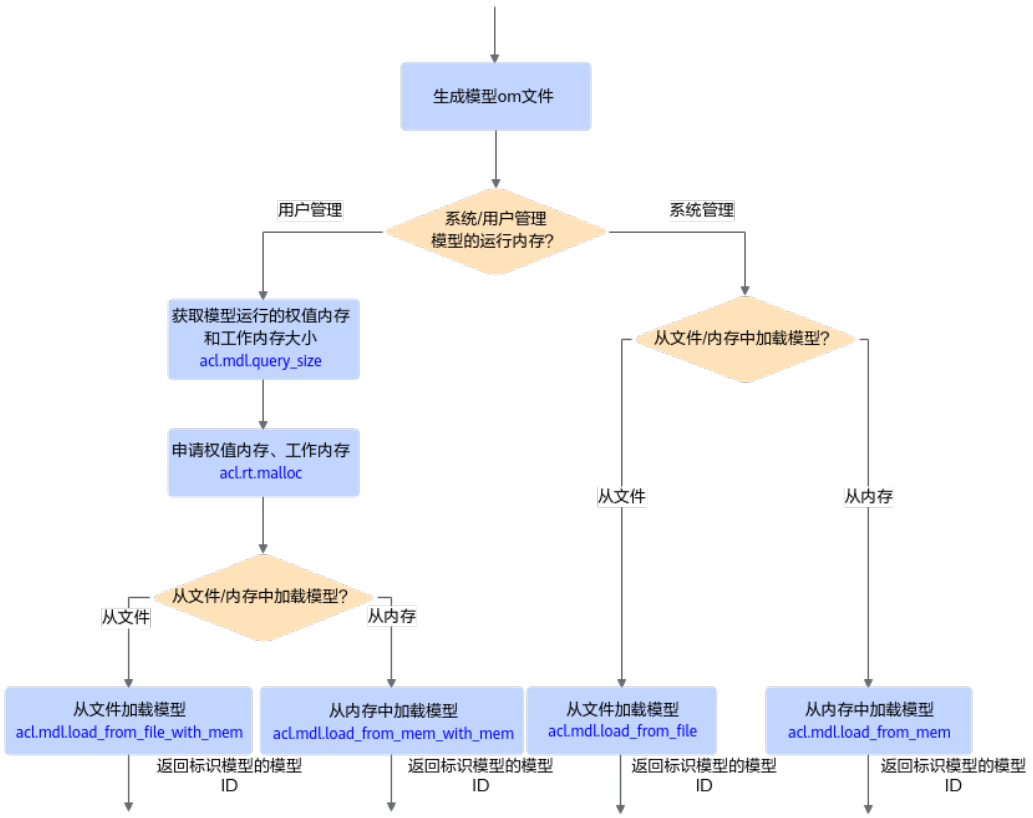
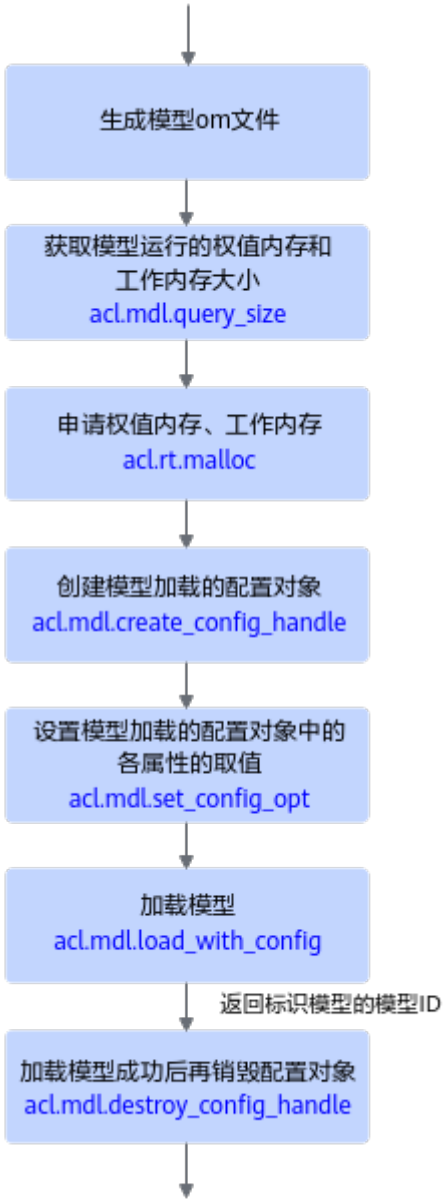


图 4-5 模型加载流程（通过接口中的配置参数区分加载方式）



关键接口的说明如下：

- **在模型加载前**，需要先构建出适配昇腾AI处理器的离线模型（*.om文件），构建方式请参见**4.2 模型构建**。
- **由用户管理内存**时，为确保内存不浪费，在申请工作内存、权值内存前，需要调用acl.mdl.query_size接口**查询模型运行时所需工作内存、权值内存的大小**。

  如果模型输入数据的Shape不确定，则不能调用acl.mdl.query_size接口查询内存大小，在加载模型时，就无法由用户管理内存，因此需选择由系统管理内存的模型加载接口（例如，acl.mdl.load_from_file、acl.mdl.load_from_mem）。
- 支持以下方式**加载模型**，模型加载成功后，返回标识模型的模型ID：
  - 通过不同接口区分加载方式，从使用的接口上区分从文件加载，还是从内存加载，以及内存是由系统内部管理，还是由用户管理。

  - acl.mdl.load_from_file：**从文件**加载离线模型数据，由系统内部管理内存。
  - acl.mdl.load_from_mem：**从内存**加载离线模型数据，由系统内部管理内存。
  - acl.mdl.load_from_file_with_mem：**从文件**加载离线模型数据，**由用户自行管理模型运行的内存**（包括工作内存和权值内存，工作内存用于模型执行过程中的临时数据，权值内存用于存放权值数据）。
  - acl.mdl.load_from_mem_with_mem：**从内存**加载离线模型数据，**由用户自行管理模型运行的内存**（包括工作内存和权值内存）。
- 通过接口中的配置参数区分加载方式，使用acl.mdl.set_config_opt接口、acl.mdl.load_with_config接口时，是**通过配置对象中的属性来区分**，在加载模型时是从文件加载还是从内存加载，以及内存是由系统内部管理还是由用户管理。

示例代码

模型加载成功，会返回标识模型的ID，在**4.6.2 模型执行**时需要使用该ID。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 初始化变量。
model_path = "./model/resnet50.om"
# ......

# 加载离线模型文件（适配昇腾AI处理器的离线模型），由系统管理模型运行的内存(包括权值内存、工作内存)。
# 模型加载成功，返回标识模型的ID。
model_id, ret = acl.mdl.load_from_file(model_path)

# ......
```

4.6.2 模型执行

基本原理

开发应用时，如果涉及整网模型推理，则应用程序中必须包含模型执行的代码逻辑，关于模型执行的接口调用流程。请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

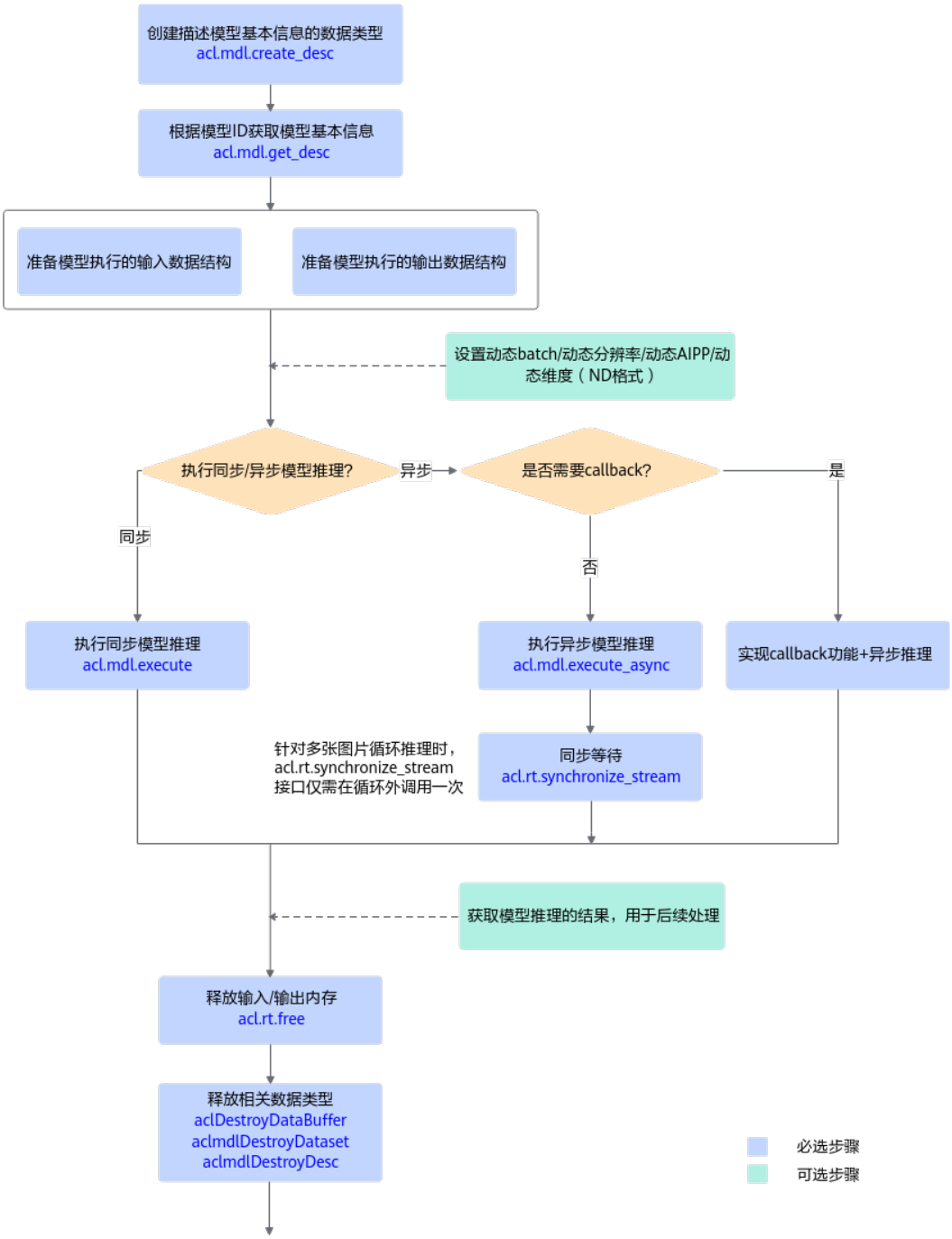
本节描述的是**整网模型执行**的接口调用流程，对于算子模型加载与执行的详细说明请参见**6 单算子调用**。

- **在模型加载之后，模型执行之前**，需要准备输入、输出数据结构，将输入数据传输到模型输入数据结构的对应内存中。
- **模型执行结束后**，若无需使用输入数据、aclmdlDesc类型、aclmdlDataset类型、aclDataBuffer类型等相关资源，需及时释放内存、销毁对应的数据类型，防止内存异常。

  模型可能存在多个输入、多个输出，每个输入/输出的内存地址、内存大小用aclDataBuffer类型的数据来描述，针对每个输入/输出，需调用acl.destroy_data_buffer接口销毁相应的aclDataBuffer类型，并调用acl.rt.free接口释放内存中的数据。

接口调用流程

图 4-6 基本的模型推理流程



关键接口的说明如下：

1. 调用acl.mdl.create_desc接口**创建描述模型基本信息**的数据类型。
2. 调用acl.mdl.get_desc接口根据**4.6.1 模型加载**中返回的模型ID**获取模型基本信息**。
3. **准备模型执行的输入、输出数据结构**，具体流程请参见**准备模型执行的输入/输出数据结构**。

如果模型的输入涉及**动态Batch**、**动态分辨率**、**动态AIPP**、**动态维度（ND格式）**等特性，请参见**7.6 模型动态推理**、**7.7 模型动态AIPP推理**。

4. **执行模型推理**。

   对于固定的多Batch场景，需要满足Batch数后，才能将输入数据发送给模型进行推理。不满足Batch数时，用户需根据自己的实际场景处理。

   当前系统支持模型的同步推理和异步推理：

   - 同步推理时调用acl.mdl.execute接口。
   - 异步推理时调用acl.mdl.execute_async接口。

     对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞应用程序运行，直到指定Stream中的所有任务都完成。

     异步推理的详细介绍，请参见**7.5 模型异步推理**。

5. **获取模型推理的结果**，用于后续处理。

   对于同步推理，直接获取模型推理的输出数据即可。

   对于异步推理，在实现Callback功能时，在回调函数内获取模型推理的结果，供后续使用。

6. **释放内存**。

   调用acl.rt.free接口释放Device上的内存。

7. **释放相关数据类型的数据**。

   在模型推理结束后，需及时依次调用acl.destroy_data_buffer接口和acl.mdl.destroy_dataset接口释放描述模型输入的数据。**如果存在多个输入、输出**，需调用多次acl.destroy_data_buffer接口。

准备模型执行的输入/输出数据结构

pyACL提供了以下数据类型来描述模型、模型输入、模型输出以及存放数据的内存，在模型执行前，需要构造好这些数据类型，作为模型执行的输入：

- 使用**aclmdlDesc**类型的数据描述模型基本信息（例如输入/输出的个数、名称、数据类型、Format、维度信息等）。

  模型加载成功后，用户可根据模型的ID，调用acl.mdl.get_desc接口获取该模型的描述信息，进而从模型的描述信息中获取模型输入/输出的个数、内存大小、维度信息、Format、数据类型等信息，可参见aclmdlDesc类型下的操作接口。

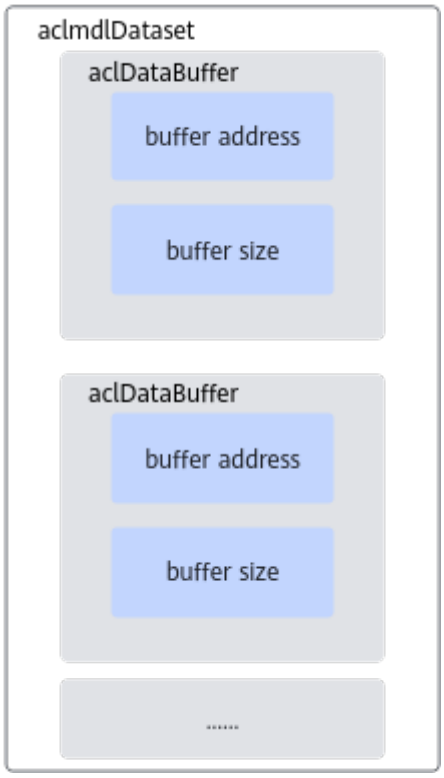
- 使用**aclmdlDataset**类型的数据描述模型的输入/输出数据，模型可能存在多个输入、多个输出。

  调用aclmdlDataset类型下的操作接口添加aclDataBuffer类型的数据、获取aclDataBuffer的个数等。

- 每个输入/输出的内存地址、内存大小用**aclDataBuffer**类型的数据来描述。

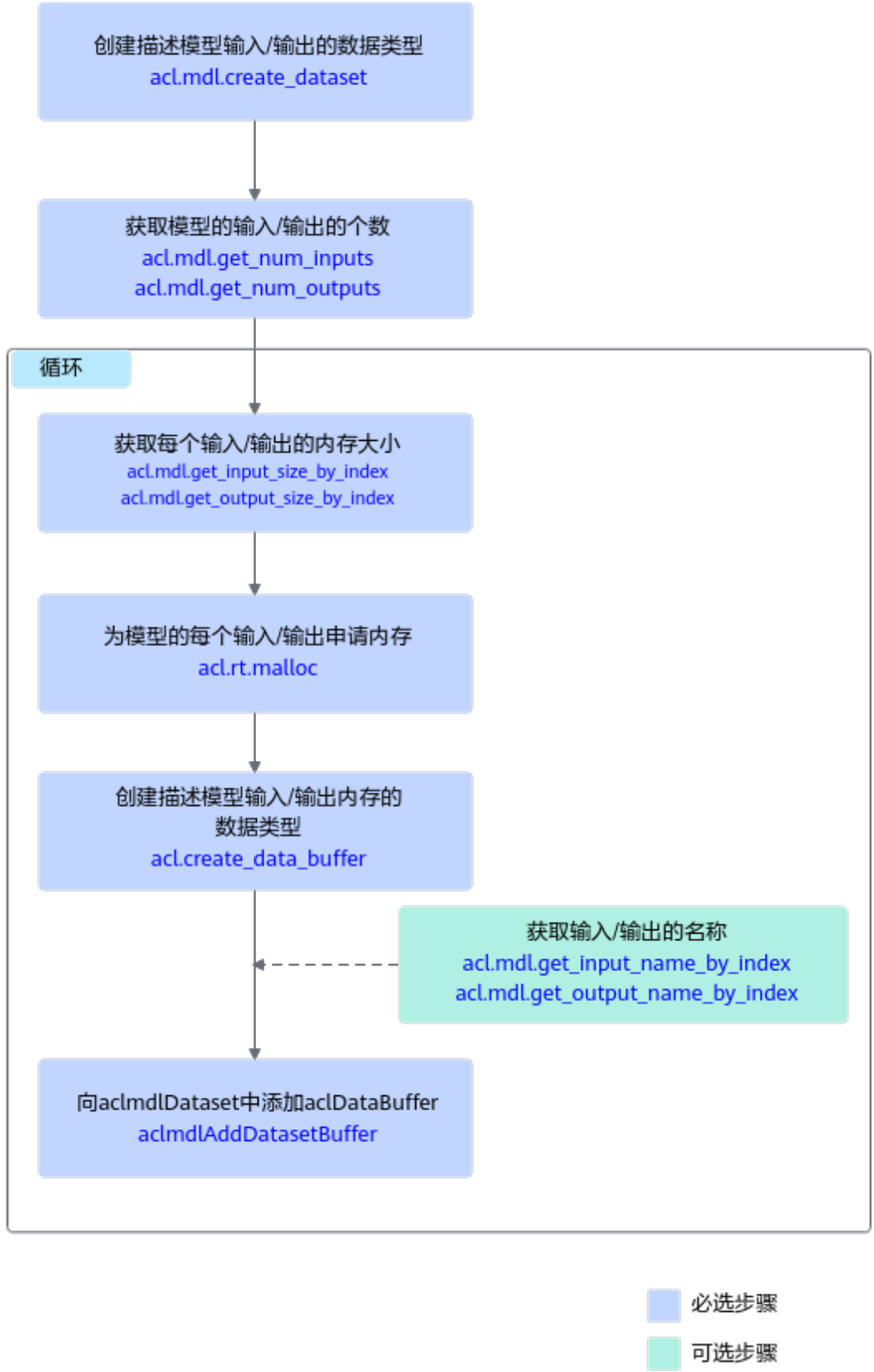
  调用aclDataBuffer类型下的操作接口获取内存地址、内存大小等。

图 4-7 aclmdlDataset 类型与 aclDataBuffer 类型的关系



了解相关的数据类型后，可以使用这些数据类型的操作接口准备模型的输入、输出数据结构，如下图所示。

图 4-8 模型执行的输入/输出数据结构的准备流程



关键说明如下：

- 模型存在多个输入、输出时，用户可调用acl.mdl.get_num_inputs、acl.mdl.get_num_outputs接口获取输入、输出的个数。
- 模型每个输入、输出所需的内存大小，用户可调用acl.mdl.get_input_size_by_index、acl.mdl.get_output_size_by_index接口获取。

  如果模型的输入涉及**动态Batch**、**动态分辨率**、**动态维度（ND格式）**等特性，输入Tensor数据的Shape支持多种档位，在模型执行前才能确定，因此该输入所需的内存大小建议用户调用acl.mdl.get_input_size_by_index接口获取，该接口获取的是最大档位的内存，确保内存够用。
- 模型存在多个输入、输出时，用户在向aclmdlDataset中添加aclDataBuffer时，为避免顺序出错，可以先调用acl.mdl.get_input_name_by_index、acl.mdl.get_output_name_by_index接口获取输入、输出的名称，根据输入、输出名称所对应的index的顺序添加。

示例代码（准备模型的输入和输出数据结构）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 初始化变量。
ACL_MEM_MALLOC_HUGE_FIRST = 0

# 1.根据模型的ID，获取该模型的描述信息。
# self.model_desc为aclmdlDesc类型。
self.model_desc = acl.mdl.create_desc()
ret = acl.mdl.get_desc(self.model_desc, self.model_id)

# 2.准备模型推理的输入数据集。
# 创建aclmdlDataset类型的数据，描述模型推理的输入。
self.load_input_dataset = acl.mdl.create_dataset()
# 获取模型输入的数量。
input_size = acl.mdl.get_num_inputs(self.model_desc)
self.input_data = []
# 循环为每个输入申请内存，并将每个输入添加到aclmdlDataset类型的数据中。
for i in range(input_size):
    buffer_size = acl.mdl.get_input_size_by_index(self.model_desc, i)
    # 申请输入内存。
    buffer, ret = acl.rt.malloc(buffer_size, ACL_MEM_MALLOC_HUGE_FIRST)
    data = acl.create_data_buffer(buffer, buffer_size)
    _, ret = acl.mdl.add_dataset_buffer(self.load_input_dataset, data)
    self.input_data.append({"buffer": buffer, "size": buffer_size})

# 3.准备模型推理的输出数据集。
# 创建aclmdlDataset类型的数据，描述模型推理的输出。
self.load_output_dataset = acl.mdl.create_dataset()
# 获取模型输出的数量。
output_size = acl.mdl.get_num_outputs(self.model_desc)
self.output_data = []
# 循环为每个输出申请内存，并将每个输出添加到aclmdlDataset类型的数据中。
for i in range(output_size):
    buffer_size = acl.mdl.get_output_size_by_index(self.model_desc, i)
    # 申请输出内存。
    buffer, ret = acl.rt.malloc(buffer_size, ACL_MEM_MALLOC_HUGE_FIRST)
    data = acl.create_data_buffer(buffer, buffer_size)
    _, ret = acl.mdl.add_dataset_buffer(self.load_output_dataset, data)
    self.output_data.append({"buffer": buffer, "size": buffer_size})

# ......
```

示例代码（执行模型）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。


```
ACL_MEMCPY_HOST_TO_DEVICE = 1
ACL_MEMCPY_DEVICE_TO_HOST = 2
NPY_BYTE = 1
images_list = ["./data/dog1_1024_683.jpg", "./data/dog2_1024_683.jpg"]

for image in images_list:
    # 1.自定义函数transfer_pic，使用Python库读取图片文件，并对图片进行缩放、剪裁等操作。
    # transfer_pic函数的实现请参考样例中源代码。
    img = transfer_pic(image)

    # 2.准备模型推理的输入数据，运行模式默认为运行模式为ACL_HOST，当前实例代码中模型只有一个输入。

    bytes_data = img.tobytes()
    np_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 将图片数据从Host传输到Device。
    ret = acl.rt.memcpy(self.input_data[0]["buffer"], self.input_data[0]["size"], np_ptr,
                        self.input_data[0]["size"], ACL_MEMCPY_HOST_TO_DEVICE)

    # 3.执行模型推理。
    # self.model_id表示模型ID，在模型加载成功后，会返回标识模型的ID。
    ret = acl.mdl.execute(self.model_id, self.load_input_dataset, self.load_output_dataset)

# ......
```

示例代码（处理推理结果：直接处理内存中的数据）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

以图片分类网络为例，模型执行结束后，需处理每一张图片的模型推理结果，直接输出top5置信度的类别编号。

```
# 处理模型推理的输出数据，输出top5置信度的类别编号。
inference_result = []
for i, item in enumerate(self.output_data):
    buffer_host, ret = acl.rt.malloc_host(self.output_data[i]["size"])
    # 将推理输出数据从Device传输到Host。
    ret = acl.rt.memcpy(buffer_host, self.output_data[i]["size"], self.output_data[i]["buffer"],
                        self.output_data[i]["size"], ACL_MEMCPY_DEVICE_TO_HOST)

    bytes_out = acl.util.ptr_to_bytes(buffer_host, self.output_data[i]["size"])
    data = np.frombuffer(bytes_out, dtype=np.byte)

    inference_result.append(data)
    tuple_st = struct.unpack("1000f", bytearray(inference_result[0]))
    vals = np.array(tuple_st).flatten()
    top_k = vals.argsort()[-1:-6:-1]
    print("======== top5 inference results: =============")
    for j in top_k:
        print("[%d]: %f" % (j, vals[j]))

# ......
```

示例代码（处理推理结果：调用单算子处理推理结果）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

以图片分类网络为例，模型执行结束后，需处理每一张图片的模型推理结果，直接输出最大置信度的类别编号。

当前示例中，使用Cast算子（将推理结果的数据类型从float32转成float16）和ArgMaxD算子（从推理结果中查找最大置信度的类别标识）实现数据后处理。单算子调用的流程请参见**6.3 接口调用流程**。

- Cast算子被封装成pyACL接口，因此在使用时，将算子的输入输出Tensor描述、算子输入输出数据的内存等信息作为acl.op.cast的入参，直接调用acl.op.cast接口加载并执行算子。
- ArgMaxD算子没有被封装成pyACL接口，因此在使用时，必须自行构造算子描述信息（输入输出Tensor描述、算子属性等）、申请存放算子输入输出数据的内存、明确算子类型名称、调用acl.op.execute_v2接口加载并执行算子。

```
ACL_MEMCPY_DEVICE_TO_HOST = 2

# 提前将Cast和ArgMaxD两个单算子的定义文件*.json编译成适配昇腾AI处理器的离线模型（*.om文件），用于验证单算子的运行。
# 设置单算子模型文件所在的目录，加载单算子模型。
ret = acl.op.set_model_dir("./op_models")
# ......

# 以下步骤需针对每一张图片的模型推理结果进行处理。
# 1.在数据后处理前，先获取模型推理的输出，dataset_ptr表示模型推理的输出。
self.input_buffer = acl.mdl.get_dataset_buffer(dataset_ptr, 0)

# 2.自定义函数_forward_op_cast，构造Cast算子的输入输出Tensor描述、申请存放算子输出数据的内存dev_buffer_cast、调用acl.op.cast接口加载并执行算子。
self._forward_op_cast()

# 3.自定义函数_forward_op_arg_max_d，构造ArgMaxD算子的输入输出Tensor、输入输出Tensor描述、算子属性、申请存放算子输出数据的内存dev_buffer_arg_max_d、调用acl.op.execute_v2接口加载并执行算子。
self._forward_op_arg_max_d()

# 4.将ArgMaxD算子的输出数据回传到Host。
# 4.1 根据ArgMaxD算子输出数据的大小，申请Host上的内存。
host_buffer, ret = acl.rt.malloc_host(self.tensor_size_arg_max_d)

# 4.2 将ArgMaxD算子的输出数据从Device复制到Host。
ret = acl.rt.memcpy(host_buffer,
                    self.tensor_size_arg_max_d,
                    self.dev_buffer_arg_max_d,
                    self.tensor_size_arg_max_d,
                    ACL_MEMCPY_DEVICE_TO_HOST)

# 4.3 在终端窗口显示最大置信度的类别编号。
bytes_out = acl.util.ptr_to_bytes(buffer_host, self.output_shape)
data = np.frombuffer(bytes_out, dtype=np.int32).reshape((self.output_shape,))
print("[SingleOP][ArgMaxOp] label of classification result is:{}"
      .format(data[0]))

# 5.释放资源。
# 5.1 释放Host的内存。
ret = acl.rt.free_host(host_buffer)

# 5.2 释放Device上存放算子输出数据的内存。
ret = acl.rt.free(self.dev_buffer_cast)
ret = acl.rt.free(self.dev_buffer_arg_max_d)

# 5.3 释放aclDataBuffer类型数据（用于描述算子输出数据）。
ret = acl.destroy_data_buffer(self.output_buffer_cast)
ret = acl.destroy_data_buffer(self.output_buffer_arg_max_d)

# ......
```

示例代码（释放模型的输入、输出资源）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 释放模型推理的输入、输出资源。
# 释放输入资源，包括数据结构和内存。
while self.input_data:
    item = self.input_data.pop()
```



```
    ret = acl.rt.free(item["buffer"])
input_number = acl.mdl.get_dataset_num_buffers(self.load_input_dataset)
for i in range(input_number):
    data_buf = acl.mdl.get_dataset_buffer(self.load_input_dataset, i)
    if data_buf:
        ret = acl.destroy_data_buffer(data_buf)
ret = acl.mdl.destroy_dataset(self.load_input_dataset)

# 释放输出资源，包括数据结构和内存。
while self.output_data:
    item = self.output_data.pop()
    ret = acl.rt.free(item["buffer"])
output_number = acl.mdl.get_dataset_num_buffers(self.load_output_dataset)
for i in range(output_number):
    data_buf = acl.mdl.get_dataset_buffer(self.load_output_dataset, i)
    if data_buf:
        ret = acl.destroy_data_buffer(data_buf)
ret = acl.mdl.destroy_dataset(self.load_output_dataset)
```

4.6.3 模型卸载

关于模型卸载的接口调用流程，请参见**3.3 pyACL接口调用流程**。

基本原理

在模型推理结束后，还需要通过acl.mdl.unload接口卸载模型，并销毁aclmdlDesc类型的模型描述信息。

示例代码

```
# 卸载模型。
ret = acl.mdl.unload(self.model_id)

# 释放模型描述信息。
if self.model_desc:
    ret = acl.mdl.destroy_desc(self.model_desc)
    self.model_desc = None
```


5 媒体数据处理

5.1 媒体数据处理基础知识

本章主要介绍图像/视频数据处理的具体功能、接口调用流程以及示例代码。

典型功能介绍

CANN提供了**AIPP**和**DVPP**两种处理图像/视频数据的方式，本章主要介绍基于DVPP的图像/视频数据处理。

AIPP、DVPP可以分开独立使用，也可以组合使用。组合使用场景下，一般先使用DVPP对图片/视频进行解码、抠图、缩放等基本处理，但由于DVPP硬件上的约束，DVPP处理后的图片格式、分辨率有可能不满足模型的要求，因此还需要再经过AIPP进一步做色域转换、抠图、填充等处理。

例如，在Atlas 200/300/500 推理产品和Atlas 训练系列产品上，由于DVPP视频解码仅支持输出YUV格式的图片，如果模型需要RGB格式的图片，则需要再经过AIPP做色域转换的处理。

处理方式	描述
AIPP（Artificial Intelligence Pre-Processing）	AIPP人工智能预处理，在AI Core上完成数据预处理，主要功能包括改变图像尺寸（抠图、填充等）、色域转换（转换图像格式）、减均值/乘系数（改变图像像素）等。 AIPP区分为静态AIPP和动态AIPP。您只能选择静态AIPP或动态AIPP中的一种来处理图片，不能同时配置静态AIPP和动态AIPP两种方式。 ● 静态AIPP：模型转换时设置AIPP模式为静态，同时设置AIPP参数，模型生成后，AIPP参数值被保存在离线模型（*.om）中，每次模型推理过程采用固定的AIPP预处理参数（无法修改）。 如果使用静态AIPP方式，多Batch情况下共用同一份AIPP参数，AIPP参数值在使用ATC工具进行模型转换时设置，ATC工具的详细说明请参见《ATC工具使用指南》。 ● 动态AIPP：模型转换时仅设置AIPP模式为动态，每次模型推理前，根据需求，在执行模型前设置动态AIPP参数值，然后在模型执行时可使用不同的AIPP参数。 如果使用动态AIPP方式，多Batch可使用不同的AIPP参数，各Batch所使用的AIPP参数值通过pyACL提供的接口来设置，请参见**7.7 模型动态AIPP推理**中的介绍。
DVPP（Digital Vision Pre-Processing）	DVPP是昇腾AI处理器内置的图像处理单元，通过pyACL媒体数据处理接口提供强大的媒体处理硬加速能力，主要功能包括以下功能： ● VPC（Vision Preprocessing Core）：处理YUV、RGB等格式的图片，包括缩放、抠图、色域转换等。 ● JPEGD（JPEG Decoder）：JPEG压缩格式-->YUV格式的图片解码。 ● JPEGE（JPEG Encoder）：YUV格式-->JPEG压缩格式的图片编码。 ● VDEC（Video Decoder）：H264/H265格式-->YUV/RGB格式的视频码流解码。 ● VENC（Video Encoder）：YUV420SP格式-->H264/H265格式的视频码流编码。 ● PNGD（PNG Decoder）：PNG格式-->RGB格式的图片解码。 **说明** AIPP、DVPP可以分开独立使用，也可以组合使用。组合使用场景下，一般先使用DVPP对图片/视频进行解码、抠图、缩放等基本处理，但由于DVPP硬件上的约束，DVPP处理后的图片格式、分辨率有可能不满足模型的要求，因此还需要再经过AIPP进一步做色域转换、抠图、填充等处理。 例如，在Atlas 200/300/500 推理产品和Atlas 训练系列产品上，由于DVPP视频解码仅支持输出YUV格式的图片，如果模型需要RGB格式的图片，则需要再经过AIPP做色域转换的处理。

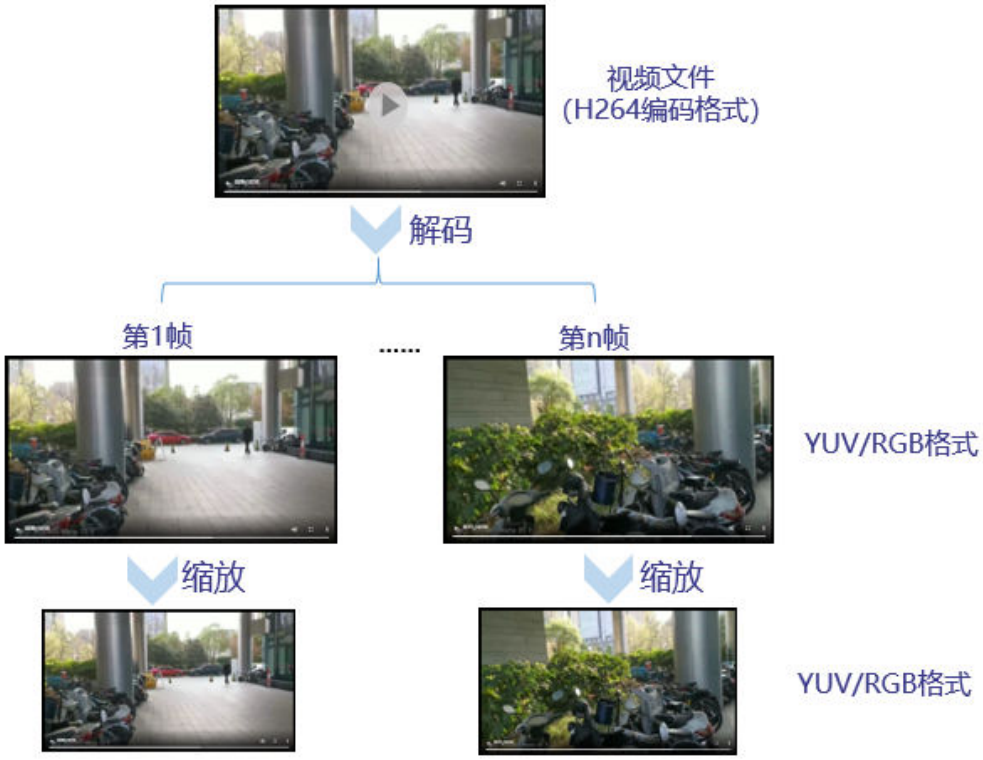
DVPP 图像/视频数据处理的典型使用场景

如果源图或视频的分辨率、格式等与模型的要求不一致时，我们可以将源图或视频处理成符合模型的要求。如下为**典型场景的举例**。

- **视频解码、缩放**。

  使用YOLOv3模型实现目标检测的场景下，用户提供的输入视频为H264/H265编码格式、分辨率为1920*1080，但YOLOv3模型要求的输入图片格式为RGB/YUV、分辨率为416*416，两者不一致，此时可对视频执行以下一系列处理。

  图 5-1 视频解码、缩放使用场景图

- **图片解码、缩放、格式转换**。

  使用ResNet-50模型实现图片分类的场景下，用户提供的输入图片为JPEG编码格式、分辨率为1280*720，但ResNet-50模型要求的输入图片格式为RGB、分辨率为224*224，两者不一致，此时可对图片执行以下一系列处理。

图 5-2 图片解码、缩放、格式转换使用场景图

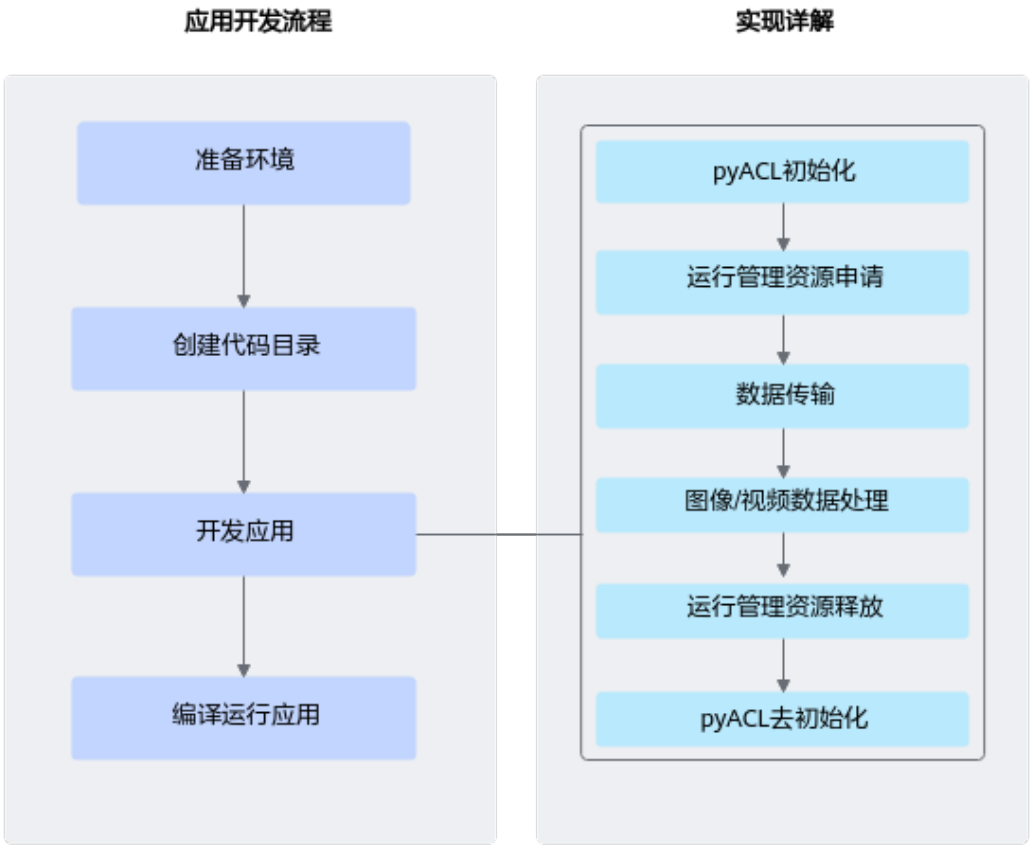
- **抠图、缩放、格式转换**。

  使用ResNet-50模型实现图片分类的场景下，用户提供的输入图片格式为YUV420SP、分辨率为1280*720，但ResNet-50模型要求的输入图片格式为RGB、分辨率为224*224，两者不一致，此时对图片执行以下一系列处理。

图 5-3 抠图、缩放、格式转换使用场景图

媒体数据处理功能开发流程

图 5-4 开发流程



1. **准备环境**。
   请参见**3.4 应用开发环境准备**。
2. **创建代码目录**。
   在开发应用前，您需要先创建目录，存放代码文件、脚本、测试图片数据、模型文件等。
3. **（可选）构建模型**。
   模型推理场景下，必须要有适配昇腾AI处理器的离线模型（*.om文件），请参见**4.2 模型构建**。

   说明

   如果应用中涉及模型推理，则需要构建模型。
4. **开发应用**。
   如果应用中涉及模型推理，请参见**4.6 模型推理基本场景**、**7 扩展更多特性**编写相应的代码。
5. **运行应用**，请参见**8 应用调试**。

5.2 V1 与 V2 版本的差别

本手册中媒体数据处理V1版本与媒体数据处理V2版本的接口都是描述处理媒体数据的接口，用于实现抠图、图片缩放、格式转换等功能，但**两套接口不能混用**。

V2版本的功能比V1版本更多，参见如下：

- JPEGE：V2版本接口支持高级的参数配置，如huffman表配置。
- VENC：V2版本接口支持更加细化的码控参数配置和效果调优，如I/P帧QP、宏块码控等。
- VDEC：V2版本接口支持更细化的内存控制，如设置输入码流缓存。

昇腾AI处理器对V1版本各功能的支持度，请参见**5.3.1 功能支持度说明**。

昇腾AI处理器对V2版本各功能的支持度，请参见**5.4.1 功能支持度说明**。

须知

Atlas 200/300/500 推理产品上，当前仅支持V1版本的媒体数据处理接口。

Atlas 训练系列产品上，当前仅支持V1版本的媒体数据处理接口。

Atlas 推理系列产品（Ascend 310P处理器）上，支持V1和V2两个版本的媒体数据处理接口，建议使用V2版本中的接口，保证后续版本接口功能以及业务的连续演进。

Atlas 200/500 A2推理产品上，支持V1和V2两个版本的媒体数据处理接口，建议使用V2版本中的接口，保证后续版本接口功能以及业务的连续演进。

Atlas A2训练系列产品上，支持V1和V2两个版本的媒体数据处理接口，建议使用V2版本中的接口，保证后续版本接口功能以及业务的连续演进。

5.3 媒体数据处理 V1

5.3.1 功能支持度说明

昇腾AI处理器对媒体数据处理V1版本各功能的支持度如下表所示。

昇腾AI处理器	VPC	JPEGD	JPEGE	PNGD	VDEC	VENC
Atlas 200/300/500 推理产品	√	√	√	√	√	√
Atlas 200/500 A2 推理产品	√	√	√	√	√	√
Atlas 训练系列产品	√	√	√	√	√	x

昇腾AI处理器	VPC	JPEGD	JPEGE	PNGD	VDEC	VENC
Atlas A2训练系列产品	√	√	√	√	√	x
Atlas 推理系列产品（Ascend 310P处理器）	√	√	√	√	√	√

5.3.2 VPC 图像处理典型功能

VPC（Vision Preprocessing Core）负责图像处理功能，支持对图片做抠图、缩放、格式转换等操作。关于VPC功能的详细介绍请参见功能说明，关于VPC功能对输入、输出的约束要求，请参见约束说明。

本节以抠图、缩放为例说明VPC图像处理时的接口调用流程，同时配合以下典型功能的示例代码辅助理解该接口调用流程：

- **图片缩放示例代码**
- **格式转换示例代码**
- **抠图（一图一框）示例代码**
- **抠图缩放（一图一框）示例代码**
- **抠图贴图（一图一框）示例代码**
- **抠图贴图缩放（一图一框）示例代码**
- **抠图贴图（一图多框）示例代码**

接口调用流程（以抠图、缩放为例）

开发应用时，如果涉及抠图、缩放等图片处理，则应用程序中必须包含图片处理的代码逻辑，**关于图片处理的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 5-5 抠图缩放流程

关键接口的说明如下（以抠图、缩放处理为例）：

1. 调用acl.media.dvpp_create_channel_desc接口**创建通道描述信息**。
2. 调用acl.media.dvpp_create_channel接口**创建图片数据处理的通道**。
3. 调用acl.media.dvpp_create_roi_config接口、acl.media.dvpp_create_resize_config接口分别**创建抠图区域位置的配置、缩放配置**。
4. 实现抠图、缩放功能前，**若需要申请Device上的内存**存放输入或输出数据，需调用acl.media.dvpp_malloc申请内存。
5. **执行抠图、缩放**。
   - 关于抠图：
     - 调用acl.media.dvpp_vpc_crop_async异步接口，按指定区域从输入图片中抠图，再将抠的图片存放到输出内存中，作为输出图片。

       输出图片区域与抠图区域“crop_area”不一致时会对图片再做一次缩放操作。
     - 当前系统还提供了acl.media.dvpp_vpc_crop_and_paste_async异步接口，支持按指定区域从输入图片中抠图，再将抠的图片贴到目标图片的指定位置，作为输出图片。
       - 抠图区域“crop_area”的宽高与贴图区域“paste_area”宽高不一致时会对图片再做一次缩放操作。
       - 如果用户需要将目标图片读入内存用于存放输出图片，将贴图区域叠加在目标图片上，则需要编写代码逻辑：**在申请输出内存后，将目标图片读入输出内存**。
   - 关于缩放：
     - 调用acl.media.dvpp_vpc_resize_async异步接口，将**输入图片缩放到输出图片大小**。
     - 缩放后输出图片内存根据YUV420SP格式计算，计算公式：**对齐后的宽 * 对齐后的高 * 3 / 2**。
   - 对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
6. 调用acl.media.dvpp_free接口**释放输入、输出内存**。
7. 调用acl.media.dvpp_destroy_roi_config接口、acl.media.dvpp_destroy_resize_config接口分别**销毁抠图区域位置的配置、缩放配置**。
8. 调用acl.media.dvpp_destroy_channel接口**销毁图片数据处理的通道**。

   销毁图片数据处理的通道后，再调用acl.media.dvpp_destroy_channel_desc接口销毁通道描述信息。

图片缩放示例代码

- 调用acl.media.dvpp_create_channel接口创建图片数据处理的通道、调用acl.media.dvpp_destroy_channel接口销毁图片数据处理的通道。
- 调用acl.media.dvpp_vpc_resize_async异步接口，将输入图片缩放到输出图片大小。对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。

- 调用acl.media.dvpp_create_resize_config接口创建图片缩放配置数据。
  如果不想使用默认缩放算法，也可以调用acl.media.dvpp_set_resize_config_interpolation接口指定缩放算法。

您可以从**获取样例**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.ACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.创建图片缩放配置数据，不支持指定缩放算法，默认缩放算法为“最近邻插值”。
# self.resize_config是acldvppResizeConfig类型。
self.resize_config = acl.media.dvpp_create_resize_config()

# 4.创建图片数据处理通道时的通道描述信息，self.dvpp_channel_desc是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 5.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 6.申请缩放输入内存，内存大小根据计算公式得出; w和h是输入图片的实际尺寸。
width_stride = ((w + 15) // 16) * 16
height_stride = ((h + 1) // 2) * 2
buffer_size = (width_stride * height_stride * 3) // 2
dev_in, ret = acl.media.dvpp_malloc(buffer_size )
self.dev_buffer['input_0'] = dev
# 7.申请缩放输出内存，内存大小根据计算公式得出; w和h是输入图片的缩放尺寸。
width_stride = ((w + 15) // 16) * 16
height_stride = ((h + 1) // 2) * 2
buffer_size = (width_stride * height_stride * 3) // 2
dev_out, ret = acl.media.dvpp_malloc(buffer_size )
self.dev_buffer['output_0'] = dev

# 8.创建缩放输入图片的描述信息，并设置各属性值。
# self.input_desc是acldvppPicDesc类型。
self.input_desc = acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(self.input_desc, buffer)
ret = acl.media.dvpp_set_pic_desc_format(self.input_desc, YUV420)
ret = acl.media.dvpp_set_pic_desc_width(self.input_desc, width)
ret = acl.media.dvpp_set_pic_desc_height(self.input_desc, height)
ret = acl.media.dvpp_set_pic_desc_width_stride(self.input_desc, wstride)
ret = acl.media.dvpp_set_pic_desc_height_stride(self.input_desc, hstride)
ret = acl.media.dvpp_set_pic_desc_size(self.input_desc, size)

# 9.创建缩放输出图片的描述信息，并设置各属性值。
# 如果缩放的输出图片作为模型推理的输入，则输出图片的宽高要与模型要求的宽高保持一致。
# self.output_desc是acldvppPicDesc类型。
self.output_desc= acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(self.output_desc, buffer)
ret = acl.media.dvpp_set_pic_desc_format(self.output_desc, YUV420)
ret = acl.media.dvpp_set_pic_desc_width(self.output_desc, width)
ret = acl.media.dvpp_set_pic_desc_height(self.output_desc, height)
ret = acl.media.dvpp_set_pic_desc_width_stride(self.output_desc, wstride)
ret = acl.media.dvpp_set_pic_desc_height_stride(self.output_desc, hstride)
ret = acl.media.dvpp_set_pic_desc_size(self.output_desc, size)

# 10.执行异步缩放，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完
```



```
成。
ret = acl.media.dvpp_vpc_resize_async(self.dvpp_channel_desc, self.input_desc,
                                      self.output_desc, self.resize_config,
                                      self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 11.缩放结束后，释放资源，包括缩放输入/输出图片的描述信息、缩放输入/输出内存。
ret = acl.media.dvpp_destroy_pic_desc(self.output_desc)
ret = acl.media.dvpp_destroy_pic_desc(self.input_desc)
ret = acl.media.dvpp_free(self.dev_buffer['input_0'] )
ret = acl.media.dvpp_free(self.dev_buffer['output_0'] )

# 12.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)
ret = acl.finalize()
# ......
```

格式转换示例代码

格式转换支持以下两种实现方式：

- 在实现抠图、缩放等功能时，调用对应的接口（例如acl.media.dvpp_vpc_crop_async接口）时，**通过将输入图片和输出图片的格式设置成不同的，达到转换图片格式的目的**。
- 如果**仅进行图片格式转换**，也可以直接调用acl.media.dvpp_vpc_convert_color_async接口。（Atlas 200/300/500 推理产品、.Atlas 训练系列产品不支持调用该接口。）

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

ACL_MEMCPY_HOST_TO_DEVICE = 1
ACL_MEMCPY_DEVICE_TO_HOST = 2
PIXEL_FORMAT_YUV_400 = 0
PIXEL_FORMAT_YUV_SEMIPLANAR_420 = 1

# 1.ACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
device_id = 0
ret = acl.rt.set_device(device_id)
context, ret = acl.rt.create_context(device_id)
stream, ret = acl.rt.create_stream()

# 3.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc是acldvppChannelDesc类型。
dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 4.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(dvpp_channel_desc)

# 5.申请输入内存。
input_width_stride = ((input_width + 15) // 16) * 16
input_height_stride = ((input_height + 1) // 2) * 2
# input_width_stride、input_height_stride分别表示图片的对齐后宽、对齐后高，此处以YUV420SP格式的图片为例。
in_buffer_size = (input_width_stride * input_height_stride * 3) // 2
in_dev_buffer, ret = acl.media.dvpp_malloc(in_buffer_size)
np_yuv = np.fromfile(path, dtype=np.byte)
buffer_size = np_yuv.itemsize * np_yuv.size
bytes_data = np_yuv.tobytes()
```



```
np_yuv_ptr = acl.util.bytes_to_ptr(bytes_data)
ret = acl.rt.memcpy(in_dev_buffer, buffer_size, np_yuv_ptr,
                    buffer_size, ACL_MEMCPY_HOST_TO_DEVICE)

# 6.申请色域转换输出内存。
output_width_stride = ((output_width + 15) // 16) * 16
output_height_stride = ((output_height + 1) // 2) * 2
# output_width_stride、output_height_stride分别表示图片的对齐后宽、对齐后高，此处以YUV420SP格式的图片为例。
out_buffer_size = (output_width_stride * output_height_stride * 3) // 2
out_dev_buffer, ret = acl.media.dvpp_malloc(out_buffer_size)

# 7.创建色域转换输入图片的描述信息，并设置各属性值。
input_desc = acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(input_desc, in_dev_buffer)
ret = acl.media.dvpp_set_pic_desc_format(input_desc, PIXEL_FORMAT_YUV_SEMIPLANAR_420)
ret = acl.media.dvpp_set_pic_desc_width(input_desc, input_width)
ret = acl.media.dvpp_set_pic_desc_height(input_desc, input_height)
ret = acl.media.dvpp_set_pic_desc_width_stride(input_desc, input_width_stride)
ret = acl.media.dvpp_set_pic_desc_height_stride(input_desc, input_height_stride)
ret = acl.media.dvpp_set_pic_desc_size(input_desc, in_buffer_size)

# 8.创建色域转换的输出图片的描述信息，并设置各属性值, 输出的宽和高要求和输入一致。
# 如果色域转换的输出图片作为模型推理的输入，则输出图片的宽高要与模型要求的宽高保持一致。
output_desc = acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(output_desc, out_dev_buffer)
ret = acl.media.dvpp_set_pic_desc_format(output_desc, PIXEL_FORMAT_YUV_400)
ret = acl.media.dvpp_set_pic_desc_width(output_desc, output_width)
ret = acl.media.dvpp_set_pic_desc_height(output_desc, output_height)
ret = acl.media.dvpp_set_pic_desc_width_stride(output_desc, output_width_stride)
ret = acl.media.dvpp_set_pic_desc_height_stride(output_desc, output_height_stride)
ret = acl.media.dvpp_set_pic_desc_size(output_desc, out_buffer_size)

# 9.执行异步色域转换，再调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
ret = acl.media.dvpp_vpc_convert_color_async(dvpp_channel_desc, input_desc,
                                             output_desc, stream)
ret = acl.rt.synchronize_stream(stream)

# 10.色域转换结束后，释放资源，包括输入/输出图片的描述信息、输入/输出内存。
ret = acl.media.dvpp_destroy_pic_desc(input_desc)
ret = acl.media.dvpp_destroy_pic_desc(output_desc)

np_output = np.zeros(out_buffer_size, dtype=np.byte)
bytes_data = np_output.tobytes()
np_output_ptr = acl.util.bytes_to_ptr(bytes_data)
# 将Device的处理结果数据传输到Host。
ret = acl.rt.memcpy(np_output_ptr, out_buffer_size, out_dev_buffer,
                    out_buffer_size, ACL_MEMCPY_DEVICE_TO_HOST)
ret = acl.media.dvpp_free(in_dev_buffer)
ret = acl.media.dvpp_free(out_dev_buffer)

# 11.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)
ret = acl.finalize()
# ......
```

抠图（一图一框）示例代码

调用acl.media.dvpp_vpc_crop_async异步接口，按指定区域从输入图片中抠图，再将抠的图片存放到输出内存中，作为输出图片。

您可以从**获取样例**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。


```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.指定抠图区域的位置、指定贴图区域的位置。
# 按左上角为原点做偏移。w，h为图片原始宽高。
self.crop_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)

# 4.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc 是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 5.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 6.创建输入输出图片的描述信息，并设置各属性值。
self.input_desc = acl.media.dvpp_create_pic_desc()
assert self.input_desc is not None
# 6.1 自定义方法 set_picture_desc 设置输入图片描述。
# 根据计算公式计算内存大小存储图片数据，并设置到图片描述，同时设置其他属性。
self.set_picture_desc(self.input_desc, w, h, "input", 0)

self.output_desc = acl.media.dvpp_create_pic_desc()
assert self.output_desc is not None
# 6.2 自定义方法 set_picture_desc 设置输出图片描述。
out_buffer_size = self.set_picture_desc(self.output_desc, w // 2, h // 2, "output", 0)

# 7.执行异步抠图，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完成。
ret = acl.media.dvpp_vpc_crop_async(self.dvpp_channel_desc, self.input_desc, self.output_desc,
self.crop_area, self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 8.释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(self.input_desc)
ret = acl.media.dvpp_destroy_pic_desc(self.output_desc)
# dev_buffer是字典对象，存储device侧申请内存。
for key in self.dev_buffer.keys():
    if self.dev_buffer[key]:
        ret = acl.media.dvpp_free(self.dev_buffer[key])
if self.dvpp_channel_desc:
    ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
    assert ret == 0
    ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)
    assert ret == 0

# 9.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)

# 10.pyACL去初始化。
ret = acl.finalize()
# ......
```

抠图缩放（一图一框）示例代码

- 调用acl.media.dvpp_vpc_crop_resize_async异步接口，按指定区域从输入图片中抠图，再将抠的图片存放到输出内存中，作为输出图片。对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。

调用acl.media.dvpp_vpc_crop_resize_async接口实现抠图缩放时，支持指定缩放算法。

- 输出图片区域与抠图区域“crop_area”不一致时会对图片再做一次缩放操作。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.指定抠图区域的位置、指定贴图区域的位置。
# 按左上角为原点做偏移。w，h为图片原始宽高。
self.crop_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)

# 4.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc 是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 5.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 6.创建输入输出图片的描述信息，并设置各属性值。
self.input_desc = acl.media.dvpp_create_pic_desc()
assert self.input_desc is not None
# 6.1 自定义方法 set_picture_desc 设置输入图片描述。
# 根据计算公式计算内存大小存储图片数据，并设置到图片描述，同时设置其他属性。
self.set_picture_desc(self.input_desc, w, h, "input", 0)

self.output_desc = acl.media.dvpp_create_pic_desc()
assert self.output_desc is not None
# 6.2 自定义方法 set_picture_desc 设置输出图片描述。
resize_width = w // 4
resize_height = h // 4
out_buffer_size = self.set_picture_desc(self.output_desc, resize_width, resize_height, "output", 0)

# 7.执行异步抠图，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完成。
self.resize_config = acl.media.dvpp_create_resize_config()
ret = acl.media.dvpp_vpc_crop_resize_async(self.dvpp_channel_desc, self.input_desc,
                                           self.output_desc, self.crop_area, self.resize_config,
                                           self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 8.释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(self.input_desc)
ret = acl.media.dvpp_destroy_pic_desc(self.output_desc)
# dev_buffer是字典对象，存储device侧申请内存。
for key in self.dev_buffer.keys():
    if self.dev_buffer[key]:
        ret = acl.media.dvpp_free(self.dev_buffer[key])
if self.dvpp_channel_desc:
    ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
    assert ret == 0
    ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)
    assert ret == 0

# 9.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)
```



```
# 10.pyACL去初始化。
ret = acl.finalize()
# ......
```

抠图贴图（一图一框）示例代码

- 调用acl.media.dvpp_vpc_crop_and_paste_async异步接口，按指定区域从输入图片中抠图，再将抠的图片贴到目标图片的指定位置，作为输出图片。对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
- 抠图区域“crop_area”的宽高与贴图区域“paste_area”宽高不一致时会对图片再做一次缩放操作。
- 如果用户需要将目标图片读入内存用于存放输出图片，将贴图区域叠加在目标图片上，则需要编写代码逻辑：在申请输出内存后，将目标图片读入输出内存。

您可以从**获取样例**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请，包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.指定抠图区域的位置、指定贴图区域的位置。
# 按左上角为原点做偏移。w，h为图片原始宽高。
self.crop_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)
self.paste_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)

# 4.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc 是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 5.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 6.创建输入输出图片的描述信息，并设置各属性值。
self.input_desc = acl.media.dvpp_create_pic_desc()
assert self.input_desc is not None
# 6.1 自定义方法 set_picture_desc 设置输入图片描述。
# 根据计算公式计算内存大小存储图片数据，并设置到图片描述，同时设置其他属性。
self.set_picture_desc(self.input_desc, w, h, "input", 0)

self.output_desc = acl.media.dvpp_create_pic_desc()
assert self.output_desc is not None
# 6.2 自定义方法 set_picture_desc 设置输出图片描述。
out_buffer_size = self.set_picture_desc(self.output_desc, w, h, "output", 0)

# 7.执行异步抠图粘贴，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完成。
ret = acl.media.dvpp_vpc_crop_and_paste_async(self.dvpp_channel_desc,
                                              self.input_desc, self.output_desc,
                                              self.crop_area, self.paste_area,
                                              self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 8.释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(self.input_desc)
```



```
ret = acl.media.dvpp_destroy_pic_desc(self.output_desc)
# dev_buffer是字典对象，存储device侧申请内存。
for key in self.dev_buffer.keys():
    if self.dev_buffer[key]:
        ret = acl.media.dvpp_free(self.dev_buffer[key])
if self.dvpp_channel_desc:
    ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
    assert ret == 0
    ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)
    assert ret == 0

# 9.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)

# 10.pyACL去初始化。
ret = acl.finalize()
# ......
```

抠图贴图缩放（一图一框）示例代码

- 调用acl.media.dvpp_vpc_crop_resize_paste_async异步接口，按指定区域从输入图片中抠图，再将抠的图片贴到目标图片的指定位置，作为输出图片。对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。

  调用acl.media.dvpp_vpc_crop_resize_paste_async实现抠图缩放贴图时，支持指定缩放算法。
- 抠图区域“crop_area”的宽高与贴图区域“paste_area”宽高不一致时会对图片再做一次缩放操作。
- 如果用户需要将目标图片读入内存用于存放输出图片，将贴图区域叠加在目标图片上，则需要编写代码逻辑：在申请输出内存后，将目标图片读入输出内存。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请，包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc 是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 4.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 5.创建输出图片的描述信息，并设置各属性值。
self.output_desc = acl.media.dvpp_create_pic_desc()
assert self.output_desc is not None
out_buffer_size = self.set_picture_desc(self.output_desc, w // 2, h // 2, "output", 0)

# 6.创建输入图片的描述信息，并设置各属性值。
self.input_desc = acl.media.dvpp_create_pic_desc()
assert self.input_desc is not None
self.set_picture_desc(self.input_desc, w, h, "input", 0)
```



```
# 7.加载图片数据，并拷贝到Device侧。
np_yuv = np.fromfile(path, dtype=np.byte)
in_buffer_size = np_yuv.itemsize * np_yuv.size
bytes_data = np_yuv.tobytes()
np_yuv_ptr = acl.util.bytes_to_ptr(bytes_data)
ret = acl.rt.memcpy(self.dev_buffer["input_0"], in_buffer_size, np_yuv_ptr,
                    in_buffer_size, ACL_MEMCPY_HOST_TO_DEVICE)
assert ret == 0

# 8.指定抠图、贴图区域的位置。
self.crop_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)
self.paste_area = acl.media.dvpp_create_roi_config(w // 4, w // 2 - 1, h // 4, h // 2 - 1)

# 9.执行异步抠图贴图缩放。
self.resize_config = acl.media.dvpp_create_resize_config()
ret = acl.media.dvpp_vpc_crop_resize_paste_async(self.dvpp_channel_desc,
                                                 self.input_desc, self.output_desc,
                                                 self.crop_area, self.paste_area,
                                                 self.resize_config, self.stream)
assert ret == 0
ret = acl.rt.synchronize_stream(self.stream)
assert ret == 0

# 10.将输出数据拷贝到Host侧。
np_output = np.zeros(out_buffer_size, dtype=np.byte)
bytes_data = np_output.tobytes()
np_output_ptr = acl.util.bytes_to_ptr(bytes_data)
ret = acl.rt.memcpy(np_output_ptr, out_buffer_size, self.dev_buffer["output_0"],
                    out_buffer_size, ACL_MEMCPY_DEVICE_TO_HOST)
assert ret == 0

# 11.释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(self.input_desc)
ret = acl.media.dvpp_destroy_pic_desc(self.output_desc)
# dev_buffer是字典对象，存储device侧申请内存。
for key in self.dev_buffer.keys():
    if self.dev_buffer[key]:
        ret = acl.media.dvpp_free(self.dev_buffer[key])
if self.dvpp_channel_desc:
    ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
    assert ret == 0
    ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)
    assert ret == 0

# 12.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)

# 13.pyACL去初始化。
ret = acl.finalize()
# ......
```

抠图贴图（一图多框）示例代码

- 调用dvpp_vpc_batch_crop_and_paste_async异步接口，按指定区域从输入图片中抠图，再将抠的图片贴到目标图片的指定位置，作为输出图片。对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
- 抠图区域“crop_area”的宽高与贴图区域“paste_area”宽高不一致时会对图片再做一次缩放操作。
- 如果用户需要将目标图片读入内存用于存放输出图片，将贴图区域叠加在目标图片上，则需要编写代码逻辑：在申请输出内存后，将目标图片读入输出内存。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.创建图片批处理输入输出描述。
self.out_batch_pic_desc = acl.media.dvpp_create_batch_pic_desc(self.out_batch_size)
self.in_batch_pic_desc = acl.media.dvpp_create_batch_pic_desc(self.in_batch_size)
# 3.1 指定批量抠图区域的位置、指定批量贴图区域的位置。
# 3.2 设置输入输出图片描述:如果抠图贴图的输出图片作为模型推理的输入，则输出图片的宽高要与模型要求的宽高保持一致。
for i in range(self.in_batch_size):
    input_desc = acl.media.dvpp_get_pic_desc(self.in_batch_pic_desc, i)
    assert input_desc is not None
    # 申请内存并设置输出图片描述。
    self.set_picture_desc(input_desc, w, h, "input", i)
    # copy from host to device.
    key = "input" + '_' + str(i)
    ret = acl.rt.memcpy(self.dev_buffer[key], in_buffer_size, np_yuv_ptr,
                        in_buffer_size, 1)
    assert ret == 0
    # 创建roiNums,每张图对应需要抠图和贴图的数量。
    roiList.append(self.out_batch_size // self.in_batch_size)
for i in range(self.out_batch_size):
    output_desc = acl.media.dvpp_get_pic_desc(self.out_batch_pic_desc, i)
    assert output_desc is not None
    # 自定义方法申请内存并设置输出图片描述。
    self.set_picture_desc(output_desc, w, h, "output", i)
    if i == 0:
        crop_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)
    else:
        crop_area = acl.media.dvpp_create_roi_config(0, w - 1, h // 2, h - 1)
    paste_area = acl.media.dvpp_create_roi_config(w // 2, w - 1, h // 2, h - 1)
    self.cropList.append(crop_area)
    self.pasteList.append(paste_area)
# 3.3 roiList,每张图对应需要抠图和贴图的数量。输出图片数相对输入多出来的，加到最后一张输入图片的输出。
total_num = 0
for i in range(self.in_batch_size):
    total_num += roiList[i]
if self.out_batch_size % self.in_batch_size != 0:
    roiList[-1] = self.out_batch_size - total_num + roiList[-1]

# 4.创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 5.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 6.执行异步缩放，再调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
_, ret = acl.media.dvpp_vpc_batch_crop_and_paste_async(self.dvpp_channel_desc, self.in_batch_pic_desc,
                                                       roiList, self.out_batch_pic_desc, self.cropList,
                                                       self.pasteList, self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 7.释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等。
# 7.1 释放图片描述和批量图片描述。
self._free_pic_desc()
for i in range(len(self.cropList)):
    ret = acl.media.dvpp_destroy_roi_config(self.cropList[i])
    assert ret == 0
for i in range(len(self.pasteList)):
```



```
    ret = acl.media.dvpp_destroy_roi_config(self.pasteList[i])
    assert ret == 0
for key in self.dev_buffer.keys():
    if self.dev_buffer[key]:
        ret = acl.media.dvpp_free(self.dev_buffer[key])
# 7.2 销毁通道和通道描述。
if self.dvpp_channel_desc:
    ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
    assert ret == 0
    ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)
    assert ret == 0

# 8.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)

9.pyACL去初始化。
ret = acl.finalize()
# ......
```

5.3.3 JPEGD 图片解码

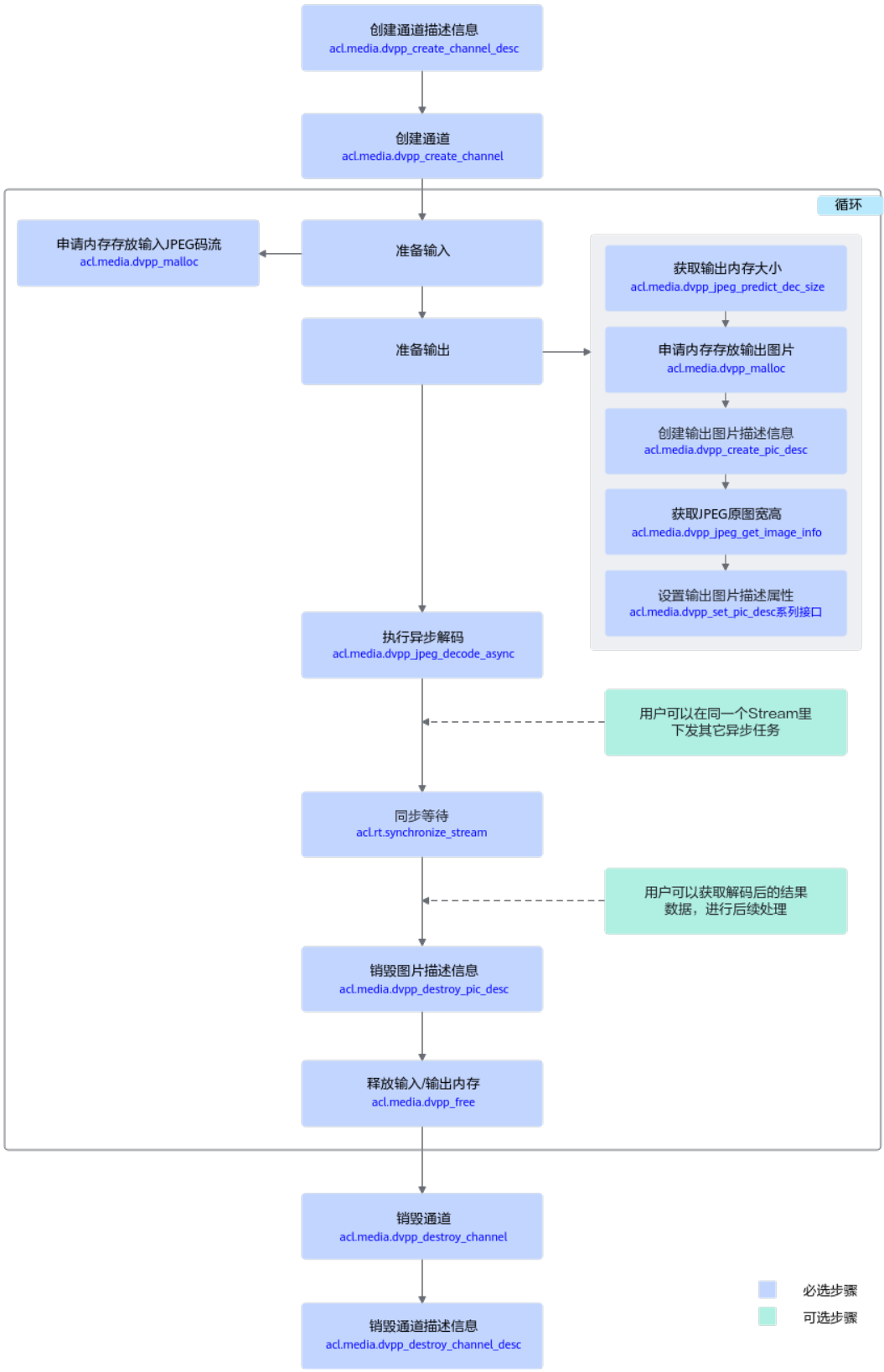
JPEGD（JPEG Decoder）负责完成图像解码功能，将.jpg、.jpeg、.JPG、.JPEG图片解码成YUV格式图片。关于JPEGD功能的详细介绍及约束请参见功能及约束说明。

本节介绍JPEGD图片解码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

接口调用流程

开发应用时，如果涉及对JPEG图片的解码，则应用程序中必须包含图片解码的代码逻辑，**关于图片解码的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 5-6 JPEG 图片解码



当前系统支持.jpg、.jpeg、.JPG、.JPEG图片的解码，针对不同的源图编码格式，输出不同编码格式的图片，关键接口的说明如下：

1. 调用acl.media.dvpp_create_channel_desc接口**创建通道描述信息**。
2. 调用acl.media.dvpp_create_channel接口**创建图片数据处理的通道**。
3. 实现JPEG图片解码功能前，**若需要申请Device上的内存**存放输入或输出数据，需调用acl.media.dvpp_malloc申请内存。

   在申请输出内存前，可根据存放JPEG图片数据的内存，调用acl.media.dvpp_jpeg_predict_dec_size接口**预估JPEG图片解码后所需的输出内存的大小**。
4. 调用acl.media.dvpp_jpeg_decode_async异步接口进行解码。

   对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
5. 在解码结束后，需及时调用acl.media.dvpp_free接口**释放输入、输出内存**。
6. 调用acl.media.dvpp_destroy_channel接口**销毁图片数据处理的通道**。

   销毁图片数据处理的通道后，再调用acl.media.dvpp_destroy_channel_desc接口销毁通道描述信息。

示例代码

您可以从**获取样例**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
context, ret = acl.rt.create_context(self.device_id)
stream, ret = acl.rt.create_stream()

# 3.创建图片数据处理通道时的通道描述信息，self.dvpp_channel_desc是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 4.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 5.申请内存。
# 加载 YUV格式的图片数据。
# 申请Device内存inputDevBuff。
# 将通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存。
np_jpg = np.fromfile(self.path, dtype=np.byte)
self.in_buffer_size = np_jpg.itemsize * np_jpg.size
bytes_data = np_jpg.tobytes()
np_jpg_ptr = acl.util.bytes_to_ptr(bytes_data)
self.in_buffer_dev, ret = acl.media.dvpp_malloc(self.in_buffer_size)
ret = acl.rt.memcpy(self.in_buffer_dev, self.in_buffer_size, np_jpg_ptr,
                    self.in_buffer_size, 1)

# 6 输出内存，按16对齐方式计算并申请Device内存self.out_buffer_dev,存放编码后的输出数据。
height_stride = ((h + 1) // 2) * 2
width_stride = ((w + 15) // 16) * 16
# malloc for output
self.out_buffer_size = (width_stride * height_stride * 3) // 2
self.out_buffer_dev, ret = acl.media.dvpp_malloc(self.out_buffer_size)

# 7. 创建解码输出图片的描述信息，设置各属性值。
# 解码设置输出内存和size到图片描述，用于数据占位。
```



```
self.pic_desc = acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(self.pic_desc, self.out_buffer_dev)
ret = acl.media.dvpp_set_pic_desc_size(self.pic_desc, self.out_buffer_size)
ret = acl.media.dvpp_set_pic_desc_format(
        self.pic_desc, PIXEL_FORMAT_YUV_SEMIPLANAR_420)
ret = acl.media.dvpp_set_pic_desc_width(self.pic_desc, w)
ret = acl.media.dvpp_set_pic_desc_height(self.pic_desc, h)
ret = acl.media.dvpp_set_pic_desc_width_stride(self.pic_desc, width_stride)
ret = acl.media.dvpp_set_pic_desc_height_stride(self.pic_desc, height_stride)

# 8. 执行异步解码，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完成。
ret = acl.media.dvpp_jpeg_decode_async(self.dvpp_channel_desc,
                                       self.in_buffer_dev,
                                       self.in_buffer_size,
                                       self.pic_desc, self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 9. 解码结束后，释放资源，包括解码输出图片的描述信息、解码输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(self.pic_desc)
ret = acl.media.dvpp_free(self.in_buffer_dev)
ret = acl.media.dvpp_free(self.out_buffer_dev)
ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)

# 10. 释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)
ret = acl.finalize()

# 11. pyACL去初始化。

# ......
```

5.3.4 JPEGE 图片编码

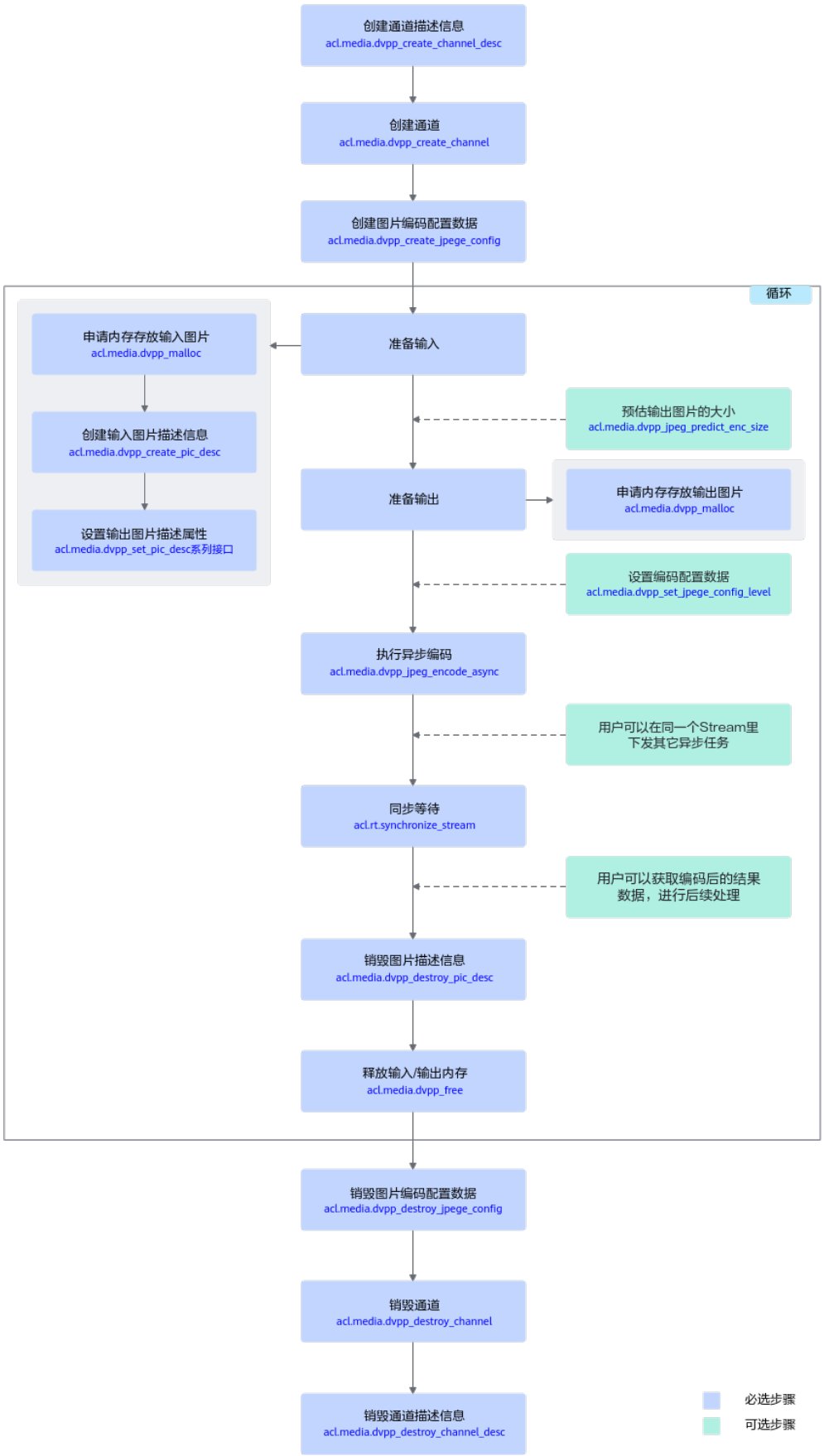
JPEGE（JPEG Encoder）负责完成图像编码功能，将YUV格式图片编码成.jpg图片。关于JPEGE功能的详细介绍及约束请参见功能及约束说明。

本节介绍JPEGE图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

接口调用流程

开发应用时，如果涉及将YUV格式图片编码成JPEG压缩格式的图片文件，则应用程序中必须包含图片编码的代码逻辑，关于图片编码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-7 JPEG 图片编码



当前系统支持将YUV格式图片编码成.jpg图片，关键接口的说明如下：

1. 调用acl.media.dvpp_create_channel_desc接口**创建通道描述信息**。
2. 调用acl.media.dvpp_create_channel接口**创建图片数据处理的通道**。
3. 调用acl.media.dvpp_create_jpege_config接口**创建图片编码配置数据**。
4. 实现JPEG图片编码功能前，**若需要申请Device上的内存存放输入或输出数据**，需调用acl.media.dvpp_malloc申请内存。

   在申请输出内存前，可调用acl.media.dvpp_jpeg_predict_enc_size接口根据输入图片描述信息、图片编码配置数据**预估图片编码后所需的输出内存的大小**。
5. 调用acl.media.dvpp_jpeg_encode_async异步接口进行编码。

   对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
6. 调用acl.media.dvpp_destroy_jpege_config接口**销毁图片编码配置数据**。
7. 在编码结束后，需及时调用acl.media.dvpp_free接口**释放输入、输出内存**。
8. 调用acl.media.dvpp_destroy_channel接口**销毁图片数据处理的通道**。

   销毁图片数据处理的通道后，再调用acl.media.dvpp_destroy_channel_desc接口**销毁通道描述信息**。

示例代码

您可以从**获取样例**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
self.device_id = 0
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.创建图片数据处理通道时的通道描述信息，self.dvpp_channel_desc是acldvppChannelDesc类型。
self.dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 4.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(self.dvpp_channel_desc)

# 5.申请内存。
# 5.1 输入内存。
#  加载 YUV格式的图片数据,将通过acl.rt.memcpy接口将Host的图片数据传输到Device。
np_jpg = np.fromfile(self.path, dtype=np.byte)
self.in_buffer_size = np_jpg.itemsize * np_jpg.size
bytes_data = np_jpg.tobytes()
np_jpg_ptr = acl.util.bytes_to_ptr(bytes_data)
self.in_buffer_dev, ret = acl.media.dvpp_malloc(self.in_buffer_size)
ret = acl.rt.memcpy(self.in_buffer_dev, self.in_buffer_size, np_jpg_ptr,
                    self.in_buffer_size, 1)

# 5.2 输出内存，按16对齐方式计算并申请Device内存self.out_buffer_dev,存放编码后的输出数据。
height_stride = ((h + 1) // 2) * 2
width_stride = ((w + 15) // 16) * 16
# malloc for output
self.out_buffer_size = (width_stride * height_stride * 3) // 2
```



```
self.out_buffer_dev, ret = acl.media.dvpp_malloc(self.out_buffer_size

# 6. 创建编码输入图片的描述信息，并设置各属性值。
# jpeg 编码设置输入yuv图片数据及size到图片描述。
self.pic_desc = acl.media.dvpp_create_pic_desc()
ret = acl.media.dvpp_set_pic_desc_data(self.pic_desc, self.in_buffer_dev)
ret = acl.media.dvpp_set_pic_desc_size(self.pic_desc, self.in_buffer_size)
ret = acl.media.dvpp_set_pic_desc_format(self.pic_desc, PIXEL_FORMAT_YUV_SEMIPLANAR_420)
ret = acl.media.dvpp_set_pic_desc_width(self.pic_desc, w)
ret = acl.media.dvpp_set_pic_desc_height(self.pic_desc, h)
ret = acl.media.dvpp_set_pic_desc_width_stride(self.pic_desc, width_stride)
ret = acl.media.dvpp_set_pic_desc_height_stride(self.pic_desc, height_stride)

# 7. 创建图片编码配置数据，设置编码质量, 预测输出的图片尺寸并分配内存。
# 编码质量范围[0, 100]，其中level 0编码质量与level 100相近，而在[1, 100]内数值越小输出图片质量越差。
self.jpege_config = acl.media.dvpp_create_jpege_config()
ret = acl.media.dvpp_set_jpege_config_level(self.jpege_config, 100)
self.out_buffer_size, ret = acl.media.dvpp_jpeg_predict_enc_size(
        self.pic_desc, self.jpege_config)
self.out_buffer_dev, ret = acl.media.dvpp_malloc(self.out_buffer_size)

# 8. 执行异步编码，再调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完
成。
acl.rt.set_context(self.context)
np_out_size = np.array([self.out_buffer_size], dtype=np.int32)
bytes_data = np_out_size.tobytes()
np_out_size_ptr = acl.util.bytes_to_ptr(bytes_data)
ret = acl.media.dvpp_jpeg_encode_async(self.dvpp_channel_desc, self.pic_desc,
                                       self.out_buffer_dev, np_out_size_ptr,
                                       self.jpege_config, self.stream)
ret = acl.rt.synchronize_stream(self.stream)

# 9.申请Host内存hostPtr。
# 9.1 将编码后的输出图片回传到Host，再将Host内存中的数据写入文件,写完文件后，需及时调用
acl.rt.free_host接口释放Host内存。
# 9.2 创建numpy对象并转成host侧数据存储输出内容。
np_output = np.zeros(size, dtype=np.byte)
bytes_data = np_output.tobytes()
np_output_ptr = acl.util.bytes_to_ptr(bytes_data)
size = int(np_out_size[0])
ret = acl.rt.memcpy(np_output_ptr, size, self.out_buffer_dev, size, 2)

# 10. 编码结束后，释放资源，包括编码输入/输出图片的描述信息、编码输入/输出内存、通道描述信息、通道
等。
ret = acl.media.dvpp_destroy_jpege_config(self.jpege_config)
ret = acl.media.dvpp_destroy_pic_desc(self.pic_desc)
ret = acl.media.dvpp_free(self.in_buffer_dev)
ret = acl.media.dvpp_free(self.out_buffer_dev)
ret = acl.media.dvpp_destroy_channel(self.dvpp_channel_desc)
ret = acl.media.dvpp_destroy_channel_desc(self.dvpp_channel_desc)

# 11. 释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id);
ret = acl.finalize()

# 12.pyACL去初始化。

# ....
```

5.3.5 PNGD 图片解码

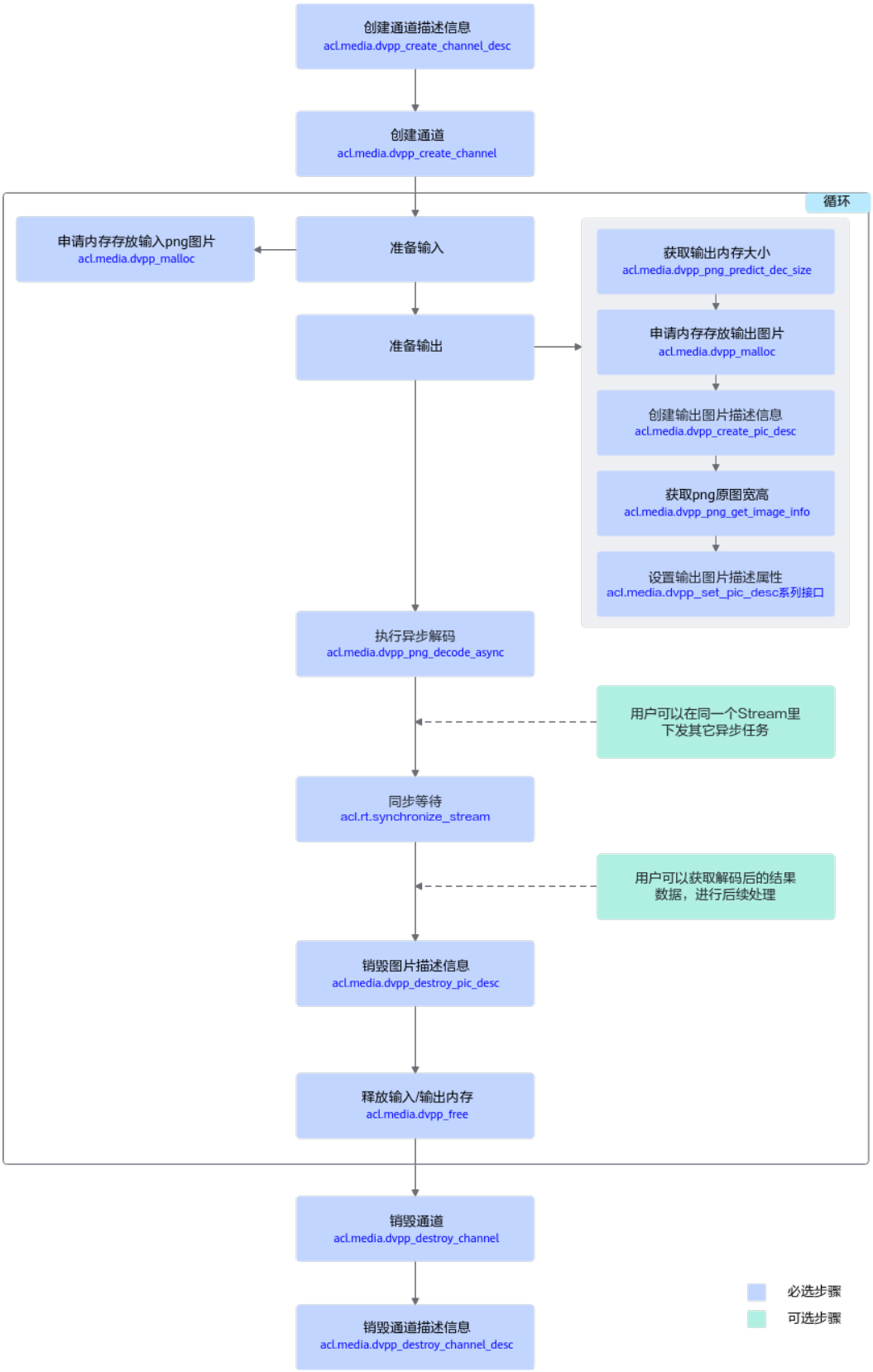
PNGD（PNG decoder）负责PNG格式图片的解码。关于PNGD功能的详细介绍及约束请参见功能及约束说明。

本节介绍PNGD图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

接口调用流程

开发应用时，如果涉及对PNG图片的解码，则应用程序中必须包含图片解码的代码逻辑，**关于图片解码的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 5-8 PNG 图片解码



当前系统支持PNG图片的解码，支持输出RGB、RGBA编码格式的图片，关键接口的说明如下：

1. 调用acl.media.dvpp_create_channel_desc接口**创建通道描述信息**。
2. 调用acl.media.dvpp_create_channel接口**创建图片数据处理的通道**。
3. 实现PNG图片解码功能前，**若需要申请Device上的内存**存放输入或输出数据，需调用acl.media.dvpp_malloc申请内存。

   在申请输出内存前，可调用acl.media.dvpp_png_predict_dec_size接口根据存放PNG图片数据的内存**预估**出PNG图片解码后所需的输出内存的大小。
4. 调用acl.media.dvpp_png_decode_async异步接口进行**解码**。

   对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
5. 在解码结束后，需及时调用acl.media.dvpp_free接口**释放输入、输出内存**。
6. 调用acl.media.dvpp_destroy_channel接口**销毁图片数据处理的通道**。

   销毁图片数据处理的通道后，再调用acl.media.dvpp_destroy_channel_desc接口**销毁通道描述信息**。

示例代码

调用接口后，需增加异常处理的分支，同时通过“ERROR_LOG”记录报错日志、通过“INFO_LOG”记录各动作的提示日志，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ACL_CONFIG_PATH = "../src/acl.json"
ret = acl.init(ACL_CONFIG_PATH)

# 2.运行管理资源申请,包括Device、Context、Stream。
device_id = 0
ret = acl.rt.set_device(device_id)
context, ret = acl.rt.create_context(device_id)
stream, ret = acl.rt.create_stream()

# 3.创建图片数据处理通道时的通道描述信息，dvpp_channel_desc是acldvppChannelDesc类型对象。
dvpp_channel_desc = acl.media.dvpp_create_channel_desc()

# 4.创建图片数据处理的通道。
ret = acl.media.dvpp_create_channel(dvpp_channel_desc)

# 5.申请输入内存。
pic_name = "../data/test_png_dec.png"
np_png = np.fromfile(pic_name, dtype=np.byte)
in_buffer_size = np_png.itemsize * np_png.size
bytes_data = np_png.tobytes()
np_png_ptr = acl.util.bytes_to_ptr(bytes_data)
in_dev_buffer, ret = acl.media.dvpp_malloc(in_buffer_size)
ACL_MEMCPY_HOST_TO_DEVICE = 1
ret = acl.rt.memcpy(in_dev_buffer, in_buffer_size, np_png_ptr, in_buffer_size,
    ACL_MEMCPY_HOST_TO_DEVICE)

# 6.申请解码输出内存。
PIXEL_FORMAT_RGB_888 = 12
out_buffer_size, ret = acl.media.dvpp_png_predict_dec_size(np_png_ptr, in_buffer_size,
    PIXEL_FORMAT_RGB_888)
out_dev_buffer, ret = acl.media.dvpp_malloc(out_buffer_size)

# 7.创建解码输出图片的描述信息，设置各属性值。
```



```
output_desc = acl.media.dvpp_create_pic_desc()
width, height, components, ret = acl.media.dvpp_png_get_image_info(np_png_ptr, in_buffer_size)
ret = acl.media.dvpp_set_pic_desc_data(output_desc, out_dev_buffer)
ret = acl.media.dvpp_set_pic_desc_size(output_desc, out_buffer_size)
ret = acl.media.dvpp_set_pic_desc_format(output_desc, PIXEL_FORMAT_RGB_888)

# 8.执行异步解码，再调用acl.rt.synchronize_stream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
ret = acl.media.dvpp_png_decode_async(dvpp_channel_desc, in_dev_buffer, in_buffer_size,
    output_desc, stream)
ret = acl.rt.synchronize_stream(stream)

# 9.解码结束后，释放资源，包括解码输出图片的描述信息、解码输出内存、通道描述信息、通道等。
ret = acl.media.dvpp_destroy_pic_desc(output_desc)
ret = acl.media.dvpp_free(in_dev_buffer)
ret = acl.media.dvpp_free(out_dev_buffer)
ret = acl.media.dvpp_destroy_channel(dvpp_channel_desc)
ret = acl.media.dvpp_destroy_channel_desc(dvpp_channel_desc)

# 10.释放运行管理资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(device_id)
ret = acl.finalize()

# 11.pyACL去初始化。

# ....
```

5.3.6 VDEC 视频解码

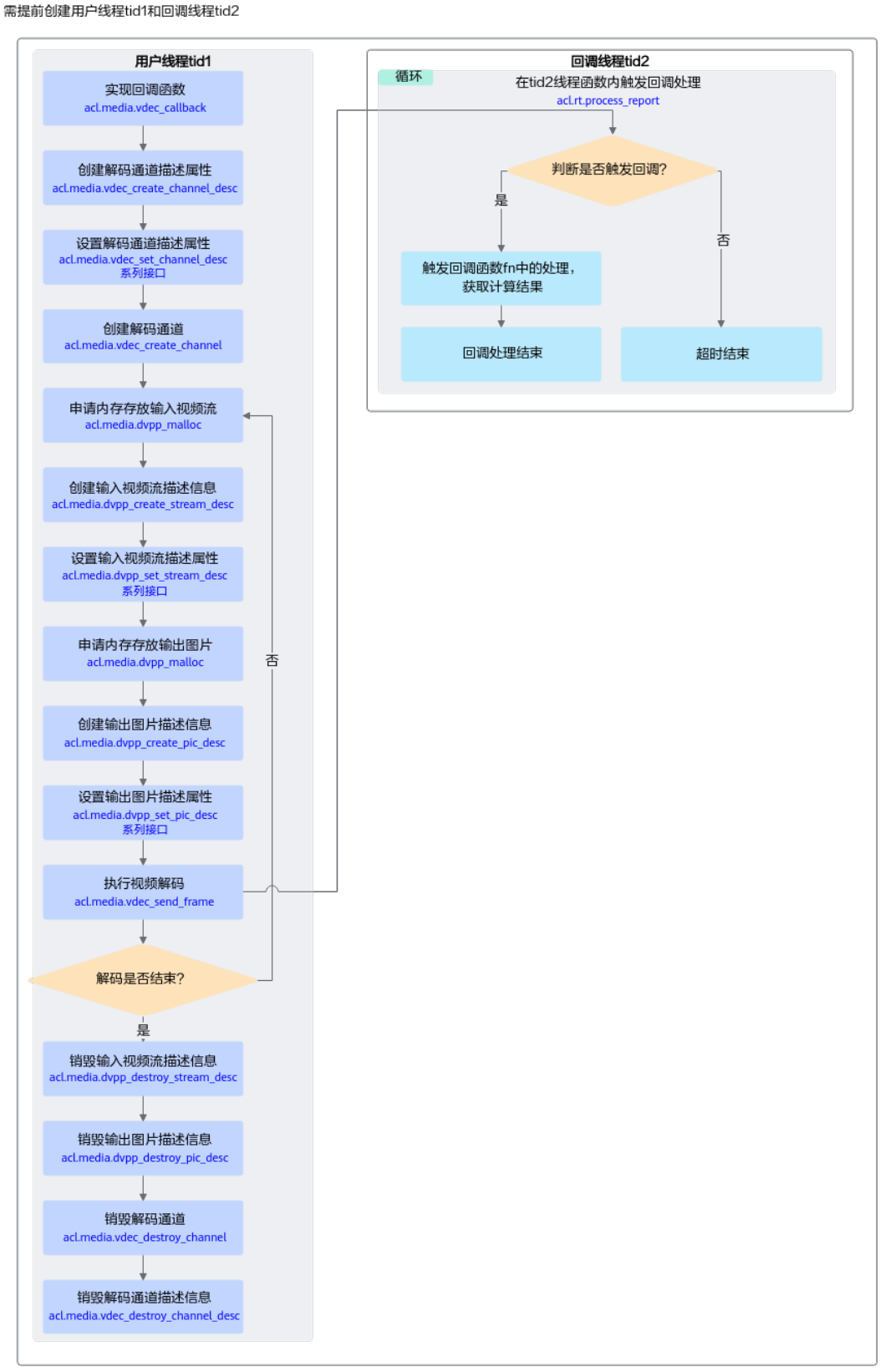
VDEC（Video Decoder）负责将H264/H265格式的视频码流解码为YUV/RGB格式的图片。关于VDEC功能的详细介绍及约束请参见功能及约束说明。

本节介绍VDEC视频编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

接口调用流程

开发应用时，如果涉及视频解码，则应用程序中必须包含视频解码的代码逻辑，**关于视频解码的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 5-9 视频解码流程



实现视频的解码，关键接口的说明如下：

1. **调用acl.media.vdec_create_channel接口创建视频解码处理的通道。**
   - 创建视频解码处理通道前，需先执行以下操作：
     i. 调用acl.media.vdec_create_channel_desc接口**创建通道描述信息**。
     ii. 调用acl.media.vdec_set_channel_desc系列接口**设置通道描述信息的属性，包括解码通道号、线程、回调函数、视频编码协议**等，其中：
        1) 回调函数需由用户提前创建，用于在视频解码后，获取解码数据，并及时释放相关资源，回调函数的原型前参见函数：vdec_set_channel_desc_callback。

           在回调函数内，用户需调用acl.media.dvpp_get_pic_desc_ret_code接口获取“ret_code”返回码判断是否解码成功，“ret_code”为“0”表示解码成功，为“1”表示解码失败。如果解码失败，需要根据日志中的返回码判断具体的问题，返回码请参见**返回码说明**。

           解码结束后，建议用户在回调函数内及时释放VDEC的输入码流内存、输出图片内存以及相应的视频码流描述信息、图片描述信息。
        2) 线程需由用户提前创建，并自定义线程函数，在线程函数内调用acl.rt.process_report接口，等待指定时间后，触发**1.ii.1)**中的回调函数。

           说明

           如果不调用acl.media.vdec_set_channel_desc_out_pic_format接口设置输出格式，则默认使用“YUV420SP NV12”。
   - acl.media.vdec_create_channel接口内部封装了如下接口，无需用户单独调用：
     i. acl.rt.create_stream接口：显式创建Stream，VDEC内部使用。
     ii. acl.rt.subscribe_report接口：指定处理Stream上回调函数的线程，回调函数和线程是由用户调用acl.media.vdec_set_channel_desc系列接口时指定的。
2. **调用acl.media.vdec_send_frame接口将视频码流解码成YUV420SP格式的图片。**
   - **视频解码前**，需先执行以下操作：
     - 调用acl.media.dvpp_create_stream_desc接口创建输入视频码流描述信息，并调用acl.media.dvpp_set_stream_desc系列接口设置输入视频的内存地址、内存大小、码流格式等属性。
     - 调用acl.media.dvpp_create_pic_desc接口创建输出图片描述信息，并调用acl.media.dvpp_set_pic_desc系列接口设置输出图片的内存地址、内存大小、图片格式等属性。
   - **视频解码时：**
     - acl.media.vdec_send_frame接口内部封装了acl.rt.launch_callback接口，用于在Stream的任务队列中增加一个需要执行的回调函数。用户无需单独调用acl.rt.launch_callback接口。
   - **视频解码后**，视频解码的结果数据通过回调函数获取：

     获取解码数据前，先获取“ret_code”的值，判断解码是否成功，0表示解码成功，1表示解码失败。如果解码失败，需要根据日志中的返回码判断具体的问题，返回码请参见**返回码说明**。

如果用户需要获取解码的帧序号，则可以在acl.media.vdec_send_frame接口的“user_data”参数处定义，然后解码的帧序号可以通过“user_data”参数传递给VDEC的回调函数，用于确定回调函数中处理的是第几帧数据。

如果不想获取某一帧的解码结果， 可以调用acl.media.vdec_send_skipped_frame接口， 将待解码的码流（输入内存）传到解码器进行解码，此时，解码结果最终不会输出，解码完成的回调函数中返回的“dvpp_pic_desc”为0。

3. **调用acl.media.vdec_destroy_channel接口销毁视频处理的通道。**
   - 系统会等待已发送帧解码完成且用户的回调函数处理完成后再销毁通道。
   - acl.media.vdec_destroy_channel接口内部封装了如下接口，无需用户单独调用。
     - acl.rt.unsubscribe_report接口：取消线程注册（Stream上的回调函数不再由指定线程处理）。
     - acl.rt.destroy_stream接口：销毁Stream。
   - 销毁通道后，需调用acl.media.vdec_destroy_channel_desc接口销毁通道描述信息。
   - 销毁通道描述信息后，用户才可以销毁**1.ii.2)**中创建的线程。

示例代码

您可以从**样例介绍**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.pyACL初始化。
ret = acl.init()

# 2.运行管理资源申请,包括Device、Context、Stream。
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()

# 3.创建回调函数。
def _callback(self, input_stream_desc, output_pic_desc, user_data):
    # 输入码流描述。
    if input_stream_desc:
        ret = acl.media.dvpp_destroy_stream_desc(input_stream_desc)
        if ret != 0:
            print("acl.media.dvpp_destroy_stream_desc failed")
    # 输出图片描述。
    if output_pic_desc:
        vdec_out_buffer = acl.media.dvpp_get_pic_desc_data(output_pic_desc)
        ret_code = acl.media.dvpp_get_pic_desc_ret_code(output_pic_desc)
        data_size = acl.media.dvpp_get_pic_desc_size(output_pic_desc)
        self.images_buffer.append(dict({"buffer": vdec_out_buffer,
                                        "size": data_size}))
        ret = acl.media.dvpp_destroy_pic_desc(output_pic_desc)
        if ret != 0:
            print("acl.media.dvpp_destroy_pic_desc failed")
    self.output_count += 1
    print("[Vdec] [_callback] _callback exit success")
    # 在此样例中，释放vdec_out_buffer在vdec流程之后处理。

# 4.创建回调处理线程，并提交注册由此线程执行回调函数。
timeout = 100
```



```
cb_thread_id, ret = acl.util.start_thread(self._thread_func, [timeout])
acl.rt.subscribe_report(cb_thread_id, self.stream)

# 5.创建视频码流处理通道时的通道描述信息，设置视频处理通道描述信息的属性，其中线程、callback回调函数需要用户提前创建。
def init_resource(self, cb_thread_id):
    print("[Vdec] class Vdec init resource stage:")
    self.vdec_channel_desc = acl.media.vdec_create_channel_desc()
    acl.media.vdec_set_channel_desc_channel_id(self.vdec_channel_desc, self._channel_id)
    acl.media.vdec_set_channel_desc_thread_id(self.vdec_channel_desc, cb_thread_id)
    acl.media.vdec_set_channel_desc_callback(self.vdec_channel_desc, self._callback)
    acl.media.vdec_set_channel_desc_entype(self.vdec_channel_desc, self._en_type)
    acl.media.vdec_set_channel_desc_out_pic_format(self.vdec_channel_desc, self._format)
    acl.media.vdec_create_channel(self.vdec_channel_desc)
    print("[Vdec] class Vdec init resource stage success")

# 6.创建视频码流处理的通道。
ret = acl.media.vdec_create_channel(self.vdec_channel_desc)
# ......
# 7.申请Device内存dataDev，存放视频解码的输入视频数据。
# 将通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时调用acl.rt.free_host接口释放Host内存。
# 输出图片尺寸对齐方式设置。
output_pic_size = (self.input_width * self.input_height * 3) // 2
img = np.fromfile(img_path, dtype=self.dtype)
input_stream_size = img.size
bytes_data = img.tobytes()
img_ptr = acl.util.bytes_to_ptr(bytes_data)
self.input_stream_mem, ret = acl.media.dvpp_malloc(input_stream_size)
ret = acl.rt.memcpy(self.input_stream_mem,
                    input_stream_size,
                    img_ptr,
                    input_stream_size,
                    ACL_MEMCPY_HOST_TO_DEVICE)

# 8.循环10次执行视频解码，输出10张YUV420SP NV12格式的图片。
def forward(self, output_pic_size, input_stream_size):
    self.frame_config = acl.media.vdec_create_frame_config()
    for i in range(self.rest_len):
        print("[Vdec] forward index:{}".format(i))
        # 8.1 创建输入视频码流描述信息，设置码流信息的属性。
        self._set_input(input_stream_size)
        # 8.2 创建输出图片描述信息，设置图片描述信息的属性。
        self._set_pic_output(output_pic_size)
        # 8.3 执行视频码流解码，解码每帧数据后，系统自动调用callback回调函数将解码后的数据写入文件，再及时释放相关资源。
        ret = acl.media.vdec_send_frame(self.vdec_channel_desc,
                                        self.dvpp_stream_desc,
                                        self.dvpp_pic_desc,
                                        self.frame_config,
                                        None)
        check_ret("acl.media.vdec_send_frame", ret)
        print('[Vdec] vdec_send_frame stage success')

# 9.释放图片处理通道、图片描述信息。
ret = acl.media.vdec_destroy_channel(self.vdec_channel_desc)
acl.media.vdec_destroy_channel_desc(self.vdec_channel_desc)

# 10. 释放运行管理资源。
ret = acl.rt.destroy_dtream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id)
ret = acl.finalize()

# 11.pyACL去初始化。

# ....
```


返回码说明

表 5-1 返回码列表

返回码	含义	可能原因及解决方法
AICPU_DVPP_KERNEL_STATE_SUCCESS = 0	解码成功。	-
AICPU_DVPP_KERNEL_STATE_FAILED = 1	其它错误。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_DVPP_ERROR = 2	AscendCL内部调用其它模块的接口失败。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_PARAM_INVALID = 3	参数校验失败。	请检查接口的参数是否符合接口要求。
AICPU_DVPP_KERNEL_STATE_OUTPUT_SIZE_INVALID = 4	输出内存大小校验失败。	请检查输出内存大小是否符合接口要求。
AICPU_DVPP_KERNEL_STATE_INTERNAL_ERROR = 5	系统内部错误。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_QUEUE_FULL = 6	系统内部队列满。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_QUEUE_EMPTY = 7	系统内部队列空。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_QUEUE_NOT_EXIST = 8	系统内部队列不存在。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_GET_CONTEX_FAILED = 9	获取系统内部上下文失败。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_SUBMIT_EVENT_FAILED = 10	提交系统内部事件失败。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_MEMORY_FAILED = 11	系统内部申请内存失败。	请检查系统是否有可用内存。
AICPU_DVPP_KERNEL_STATE_SEND_NOTIFY_FAILED = 12	发送系统内部通知失败。	日志的详细介绍，请参见《日志参考》。
AICPU_DVPP_KERNEL_STATE_VPC_OPERATE_FAILED = 13	系统内部接口处理失败。	日志的详细介绍，请参见《日志参考》。

返回码	含义	可能原因及解决方法
AICPU_DVPP_KERNEL_STATE_CHANNEL_ABNORMAL = 14	当前通道异常。	日志的详细介绍，请参见《日志参考》。
ERR_INVALID_STATE = 0x10001	VDEC解码器状态异常错误。	日志的详细介绍，请参见《日志参考》。
ERR_HARDWARE = 0x10002	硬件错误，包含解码器启动、执行、停止等异常。	日志的详细介绍，请参见《日志参考》。
ERR_SCD_CUT_FAIL = 0x10003	将视频码流分解成多帧图片异常。	请检查输入的视频流数据是否正确。
ERR_VDM_DECODE_FAIL = 0x10004	解码某一帧异常。	请检查输入的视频流数据是否正确。
ERR_ALLOC_MEM_FAIL = 0x10005	内部申请内存失败。	请检查系统是否有可用内存，**若忽略错误，则可能导致用户持续送入码流却无任何解码结果输出**。
ERR_ALLOC_DYNAMIC_MEM_FAIL = 0x10006	包括输入视频分辨率超范围、内部动态申请内存失败等异常。	请检查输入视频流的分辨率、系统是否有可用内存，**若忽略错误，则可能导致用户持续送入码流却无任何解码结果输出**。
ERR_ALLOC_IN_OR_OUT_PORT_MEM_FAIL = 0x10007	系统内部申请VDEC的输入、输出buffer异常。	请检查系统是否有可用内存，**若忽略错误，则可能导致用户持续送入码流却无任何解码结果输出**。
ERR_BITSTREAM = 0x10008	码流错误（如语法解析失败、重发eos或首帧即发送eos）。	请检查输入的视频流数据是否正确。
ERR_VIDEO_FORMAT = 0x10009	输入视频格式错误。	请检查输入视频的格式是否为h264或h265。
ERR_IMAGE_FORMAT = 0x1000a	输出格式配置错误。	请检查输出图像的格式是否为nv12或nv21。
ERR_CALLBACK = 0x1000b	回调函数为空。	请检查配置的回调函数是否为空。

返回码	含义	可能原因及解决方法
ERR_INPUT_BUFFER = 0x1000c	输入内存为空。	请检输入内存是否为空。
ERR_INBUF_SIZE = 0x1000d	输入内存大小<=0。	请检查输入内存大小是否小于等于0。
ERR_THREAD_CREATE_FBD_FAIL = 0x1000e	系统内部将解码结果通过回调函数返回给用户的线程异常。	请检查系统中资源（例如：线程、内存等）是否可用。
ERR_CREATE_INSTANCE_FAIL = 0x1000f	创建解码实例失败。	日志的详细介绍，请参见《日志参考》。
ERR_INIT_DECODER_FAIL = 0x10010	初始化解码器失败，例如解码实例个数超出范围（最大16）。	日志的详细介绍，请参见《日志参考》。
ERR_GET_CHANNEL_HANDLE_FAIL = 0x10011	系统内部获取某路视频流的解码句柄失败。	日志的详细介绍，请参见《日志参考》。
ERR_COMPONENT_SET_FAIL = 0x10012	系统内部设置解码实例异常。	请检查解码的入参值是否正确，例如输入视频格式video_format、输出帧格式image_format等。
ERR_COMPARE_NAME_FAIL = 0x10013	系统内部设置解码实例名称异常。	请检查解码的入参值是否正确，例如输入视频格式video_format、输出帧格式image_format等。
ERR_OTHER = 0x10014	其它错误。	请联系工程师。
ERR_DECODE_NOPIC = 0x20000	隔行码流场景下使用，隔行码流每帧发送两场，解码时其中一块无图像输出，属于正常现象，会返回该错误码；隔行码流的解码输出数据都在奇数场对应的输出buffer中。	-

返回码	含义	可能原因及解决方法
0x20001	参考帧个数设置错误。	请检查码流实际参考帧个数与用户设置的参考帧个数是否一致。Atlas 推理系列产品（Ascend 310P处理器），默认参考帧个数为8。
0x20002	VDEC解码帧存大小设置错误。	请检查输入码流实际宽、高与用户设置的宽、高是否一致。
0xA0058001	无效的Device ID。 暂未使用，预留。	-
0xA0058002	无效的channel ID。	请检查传入接口的通道号是否正确，或者检查通道总数是否达到上限。
0xA0058003	参数不合法，例如不合法的枚举值。	请根据日志检查出错的参数。 日志的详细介绍，请参见《日志参考》。
0xA0058004	资源已存在。	请检查是否重复创建通道。
0xA0058005	通道资源不存在。	请检查是否使用了不存在的通道号或通道句柄。
0xA0058006	函数参数中的指针地址为0。	请根据日志检查接口的入参。 日志的详细介绍，请参见《日志参考》。
0xA0058007	使能系统、Device或通道前未配置对应的参数。	请根据日志检查接口的入参。 日志的详细介绍，请参见《日志参考》。
0xA0058008	不支持的参数或者功能。	请根据日志检查接口的入参。 日志的详细介绍，请参见《日志参考》。
0xA0058009	该操作不允许，如试图修改静态配置参数。	日志的详细介绍，请参见《日志参考》。
0xA005800C	分配内存失败，如系统内存不足。	请检查系统是否有可用内存，**若忽略错误，则可能导致用户持续送入码流却无任何解码结果输出**。

返回码	含义	可能原因及解决方法
0xA005800D	分配缓存失败，如申请的数据缓冲区太大。暂未使用，预留。	-
0xA005800E	缓冲区中无数据。	系统未完成解码，缓冲区中无解码结果数据，需等待缓冲区中有数据后，再尝试获取数据。
0xA005800F	缓冲区中数据满。	用户发送输入码流数据的速度太快，导致输入缓冲区数据满，请尝试降低发送输入码流数据的速度，或者在创建通道时将缓冲区大小设置为较大值。
0xA0058010	系统没有初始化或者相关依赖的模块没有加载。 暂未使用，预留。	- 请检查是否调用系统初始化接口。
0xA0058011	地址错误。 暂未使用，预留。	- 日志的详细介绍，请参见《日志参考》。
0xA0058012	系统忙。	请检查VDEC解码总路数是否达到上限。 日志的详细介绍，请参见《日志参考》。
0xA0058013	缓存小于实际需要的大小。 暂未使用，预留。	-
0xA0058014	硬件或软件处理超时。 暂未使用，预留。	-
0xA0058015	内部系统错误。	日志的详细介绍，请参见《日志参考》。

返回码	含义	可能原因及解决方法
0xA005803F	最大的返回码，该模块的错误码必须小于该值。	-

5.3.7 VENC 视频编码

VENC（Video Encoder）将YUV420SP格式的图片编码成H264/H265格式的视频码流。关于VENC功能的详细介绍及约束请参见功能及约束说明。

本节介绍VENC视频编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

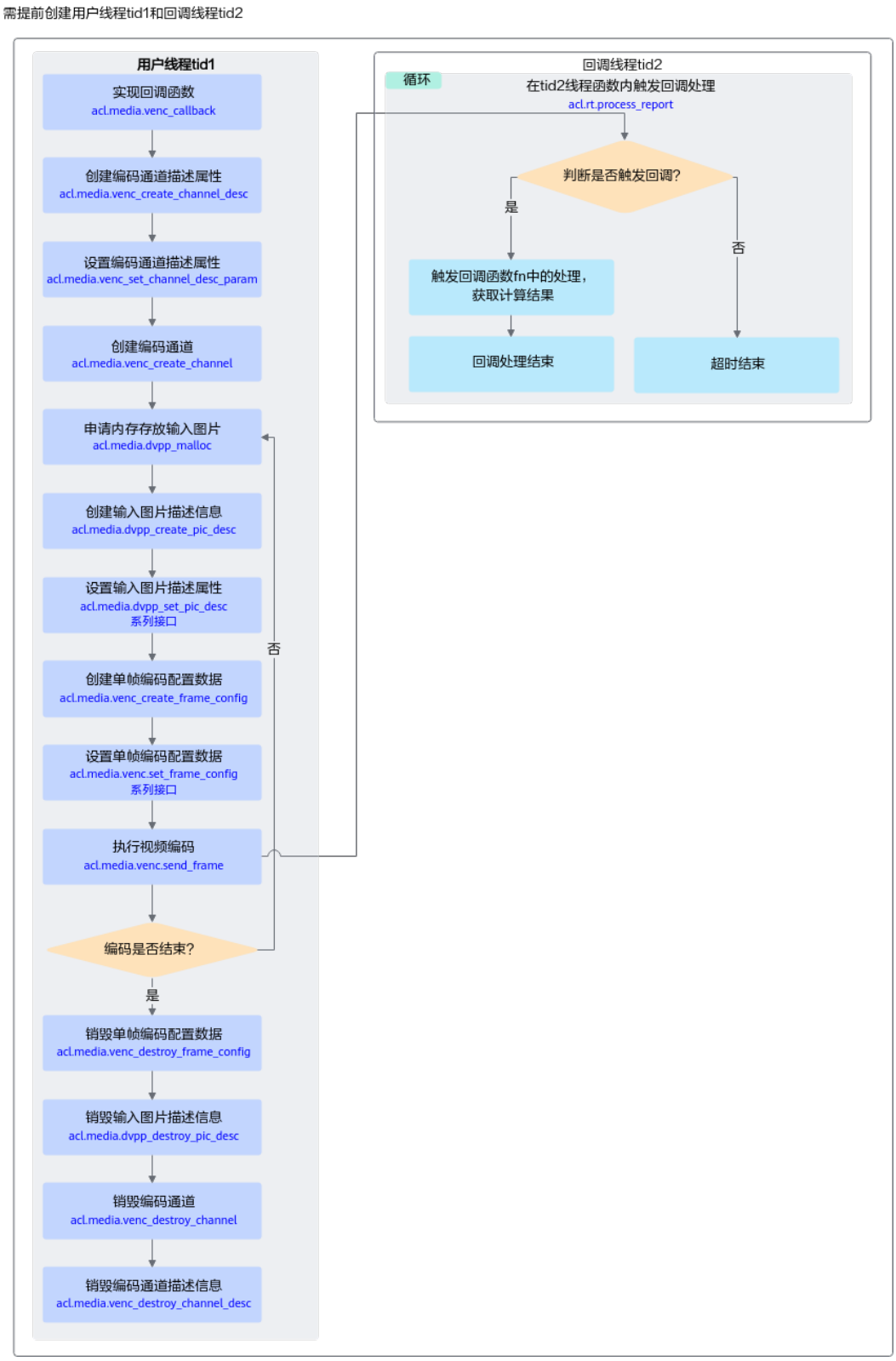
Atlas 训练系列产品上，不支持该功能。

Atlas A2训练系列产品上，不支持该功能。

接口调用流程

开发应用时，如果涉及视频编码，则应用程序中必须包含视频编码的代码逻辑，**关于视频编码的接口调用流程，请先参见3.3 pyACL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 5-10 视频编码流程



实现视频的编码，关键接口的说明如下：

1. **调用acl.media.venc_create_channel接口创建视频编码处理的通道。**

- 创建视频编码处理通道前，需先执行以下操作：
  i. 调用acl.media.venc_create_channel_desc接口创建通道描述信息。
  ii. 调用acl.media.venc_set_channel_desc_param接口设置通道描述信息的属性，包括线程、回调函数、视频编码协议、输入图片格式等，其中：
    1) 回调函数需由用户提前创建，用于在视频编码后，获取编码数据，并及时释放相关资源，回调函数的原型前参见函数：venc_set_channel_desc_callback。

       视频编码结束后，建议用户在回调函数内及时释放输入图片内存、以及相应的图片描述信息。视频编码的输出内存由系统管理，不由用户管理，因此无需用户释放。
    2) 线程需由用户提前创建，并自定义线程函数，在线程函数内调用acl.rt.process_report接口，等待指定时间后，触发**1.ii.1)**中的回调函数。

       说明

       推荐使用acl.media.venc_set_channel_desc_param接口设置通道描述信息的属性，通过枚举值来选择通过该接口设置某一个属性的值。

       但为兼容旧版本，也可以调用acl.media.venc.set_channel_desc系列接口设置通道描述信息的属性，每个属性的设置对应一个set接口。
- acl.media.venc_create_channel接口内部封装了如下接口，无需用户单独调用：
  i. acl.rt.create_stream接口：显式创建Stream，VENC内部使用。
  ii. acl.rt.subscribe_report接口：指定处理Stream上回调函数的线程，回调函数和线程是由用户调用acl.media.venc_set_channel_desc_param接口时指定的。

2. **调用acl.media.venc_send_frame接口将YUV420SP格式的图片编码成H264/H265格式的视频码流。**
   - **视频编码前**，需先执行以下操作：
     - 调用acl.media.dvpp_create_pic_desc接口创建输入图片描述信息，并调用acl.media.dvpp_set_pic_desc系列接口设置输入图片的内存地址、内存大小、图片格式等属性。
     - 调用acl.media.venc_create_frame_config接口创建单帧编码配置数据，并调用acl.media.venc_set_frame_config系列接口设置是否强制重新开始I帧间隔、是否结束帧。
   - **视频编码时**，acl.media.venc_send_frame接口内部封装了acl.rt.launch_callback接口，用于在Stream的任务队列中增加一个需要执行的回调函数。用户无需单独调用acl.rt.launch_callback接口。
   - **视频编码后**，视频编码的结果数据通过回调函数获取。
3. **调用acl.media.venc_destroy_channel接口销毁视频处理的通道。**
   - 系统会等待已发送帧编码完成且用户的回调函数处理完成后再销毁通道。
   - acl.media.venc_destroy_channel接口内部封装了如下接口，无需用户单独调用。
     - acl.rt.unsubscribe_report接口：取消线程注册（Stream上的回调函数不再由指定线程处理）。

    - acl.rt.destroy_stream接口：销毁Stream。
  - 销毁通道后，需调用acl.media.venc_destroy_channel_desc接口销毁通道描述信息。
  - 销毁通道描述信息后，用户才可以销毁**1.ii.2)**中创建的线程。

示例代码

您可以从**样例介绍**中获取完整样例代码。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.资源初始化：创建 AclVenc 对象进行初始化。
# 1.1 pyACL初始化。
ret = acl.init()
# 1.2 运行管理资源申请,包括Device、Context。
device_id = 0
ret = acl.rt.set_device(device_id)
self.context, ret = acl.rt.create_context(device_id)
# 调用acl.rt.get_run_mode接口获取软件栈的运行模式，根据运行模式来判断后续的内存申请接口调用逻辑。
runMode, ret = acl.rt.get_run_mode()

# 2.创建执行回调函数的线程及线程函数。
def cb_thread_func(self, args_list):
    ctx = args_list[0]
    timeout = args_list[1]
    print("[info] thread args_list = ", self.ctx, timeout, self.g_callbackRunFlag, "\n")
    ret = acl.rt.set_context(ctx)
    assert ret == 0

    while self.g_callbackRunFlag is True:
        print("[info] thread g_callbackRunFlag = ", self.g_callbackRunFlag, "\n")
        ret = acl.rt.process_report(timeout)
        print("[info] acl.rt.process_report =", ret)

timeout = 1000
g_callbackRunFlag = True
self.cb_thread_id, ret = acl.util.start_thread(self.cb_thread_func, [self.ctx, timeout])

# 3.创建回调函数。
# 3.1 获取流数据输出。
def get_stream_data(self, stream_desc):
    stream_data = acl.media.dvpp_get_stream_desc_data(stream_desc)
    assert stream_data is not None
    stream_data_size = acl.media.dvpp_get_stream_desc_size(stream_desc)
    print("[info] stream_data size", stream_data_size)
    # stream memcpy d2h
    np_data = np.zeros(stream_data_size, dtype=np.byte)

    bytes_data = np_data.tobytes()
    np_data_ptr = acl.util.bytes_to_ptr(bytes_data)
    ret = acl.rt.memcpy(np_data_ptr, stream_data_size,
                        stream_data, stream_data_size,
                        memcpy_kind.get("ACL_MEMCPY_DEVICE_TO_HOST"))
    assert ret == 0
    return np_data
# 3.2 回调函数将视频流保存文件。
def callback_func(self, input_pic_desc, output_stream_desc, user_data):
    # 获取视频编码结果数据, 转换成numpy对象。
    output_numpy = self.get_stream_data(output_stream_desc)
    with open('./data/output.h265', 'ab') as f:
        f.write(output_numpy)
    print("[INFO] [callback_func] stream size =",
```



```
acl.media.dvpp_get_stream_desc_size(output_stream_desc))

# 4.创建视频编码处理通道时的通道描述信息。
self.venc_channel_desc = acl.media.venc_create_channel_desc()

# 5.设置通道描述信息的属性，其中线程、callback回调函数需要用户提前创建。
# vencChannelDesc_是aclvdecChannelDesc类型。
venc_format = 1
venc_entype = 0
input_width = 1280
input_height = 720
ret = acl.media.venc_set_channel_desc_thread_id(self.venc_channel_desc, self.cb_thread_id)
ret = acl.media.venc_set_channel_desc_callback(self.venc_channel_desc, self.callback_func)
ret = acl.media.venc_set_channel_desc_entype(self.venc_channel_desc, venc_entype)
ret = acl.media.venc_set_channel_desc_pic_format(self.venc_channel_desc, venc_format)
ret = acl.media.venc_set_channel_desc_key_frame_interval(self.venc_channel_desc, 16)
ret = acl.media.venc_set_channel_desc_pic_height(self.venc_channel_desc, input_height)
ret = acl.media.venc_set_channel_desc_pic_width(self.venc_channel_desc, input_width )

# 6.创建视频码流处理的通道、单帧编码配置数据。
ret = acl.media.venc_create_channel(self.venc_channel_desc)
# vencFrameConfig_是aclvencFrameConfig类型。
frame_config = acl.media.venc_create_frame_config()

# 7.申请Device内存dataDev，存放视频编码的输入数据。
# 7.1 读入图片数据。
output_pic_size = (input_width * input_height * 3) // 2
print("[INFO] output_pic_size:", output_pic_size, " load vdec file:", venc_file_path)
file_context = np.fromfile(venc_file_path, dtype=np.byte)
file_size = file_context.size
bytes_data = file_context.tobytes()
file_mem = acl.util.bytes_to_ptr(bytes_data)
input_size = file_size
# 如果调用acl.rt.get_run_mode接口获取软件栈的运行模式为ACL_HOST，则需要通过acl.rt.memcpy接口将
Host的图片数据传输到Device，数据传输完成后，需及时调用acl.rt.free_host接口释放Host内存。
# 如果调用acl.rt.get_run_mode接口获取软件栈的运行模式为ACL_DEVICE，则直接申请Device内存，存放输入
图片数据。
# 此处是运行模式为 ACL_HOST。
input_mem, ret = acl.media.dvpp_malloc(input_size)
assert ret == 0
ret = acl.rt.memcpy(input_mem, input_size, file_mem,
                    file_size, memcpy_kind.get("ACL_MEMCPY_HOST_TO_DEVICE"))
assert ret == 0

# 8.执行视频编码。
def venc_set_frame_config(self, frame_confg, eos, iframe):
    ret = acl.media.venc_set_frame_config_eos(frame_confg, eos)
    assert ret == 0
    ret = acl.media.venc_set_frame_config_force_i_frame(frame_confg, iframe)
    assert ret == 0

def venc_process(self, venc_channel_desc, input_mem, input_size, frame_config):
   # 设置为非结束帧。
    self.venc_set_frame_config(frame_config, 0, 0)
    print("[INFO] set frame config")
    self.test_get_frame_config(frame_config)

    # set picture description
    dvpp_pic_desc = acl.media.dvpp_create_pic_desc()
    assert dvpp_pic_desc is not None
    ret = acl.media.dvpp_set_pic_desc_data(dvpp_pic_desc, input_mem)
    assert ret == 0
    ret = acl.media.dvpp_set_pic_desc_size(dvpp_pic_desc, input_size)
    assert ret == 0
    print("[INFO] set pic desc")

    # 发送1张图片到编码器进行编码。
    venc_cnt = 16
    while venc_cnt:
```



```
        ret = acl.media.venc_send_frame(venc_channel_desc, dvpp_pic_desc, 0, frame_config, 0)
        assert ret == 0
        print("[INFO] venc send frame")
        venc_cnt -= 1
    # 结束视频编码时可以发送eos为1的空图片，表示当前编码结束。
    self.venc_set_frame_config(frame_config, 1, 0)
    ret = acl.media.venc_send_frame(venc_channel_desc, 0, 0, frame_config, 0)
    assert ret == 0

# 9.释放资源。
ret = acl.media.dvpp_free(input_mem)
ret = acl.media.venc_destroy_frame_config(frame_config)
assert ret == 0
print("[INFO] free resources")
g_callbackRunFlag = False
ret = acl.util.stop_thread(self.cb_thread_id)
print("[INFO] thread join")

# 10. 释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.device_id);
ret = acl.finalize()

# 11.pyACL去初始化。

# ....
```

如果调用acl.media.venc_set_channel_desc_param接口设置通道描述信息的属性，调用acl.media.venc_get_channel_desc_param接口获取通道描述信息中的属性值，示例代码如下：

```
ACL_VENC_THREAD_ID_UINT64 = 0          #回调线程ID。
ACL_VENC_CALLBACK_PTR = 1             #回调函数。
ACL_VENC_PIXEL_FORMAT_UINT32 = 2       #输入图像格式。
ACL_VENC_ENCODE_TYPE_UINT32 = 3        #视频编码协议。
ACL_VENC_PIC_WIDTH_UINT32 = 4          #输入图片宽度。
ACL_VENC_PIC_HEIGHT_UINT32 = 5         #输入图片高度。
ACL_VENC_KEY_FRAME_INTERVAL_UINT32 = 6 #关键帧间隔。
ACL_VENC_BUF_ADDR_PTR = 7              #编码输出缓存地址。
ACL_VENC_BUF_SIZE_UINT32 = 8           #编码输出缓存大小。
ACL_VENC_RC_MODE_UINT32 = 9            #码率控制模式。
ACL_VENC_SRC_RATE_UINT32 = 10          #输入码流帧率。
ACL_VENC_MAX_BITRATE_UINT32 = 11       #输出码率。
ACL_VENC_MAX_IP_PROP_UINT32 = 12       #一个GOP内单个I帧bit数和单个P帧bit数的比例。

# 设置回调函数。
ret = acl.media.venc_set_channel_desc_param(self.venc_channel_desc, ACL_VENC_CALLBACK_PTR,
self.callback_func)
# 获取回调函数。
get_cb_func, ret = acl.media.venc_get_channel_desc_param(self.venc_channel_desc,
ACL_VENC_CALLBACK_PTR)

PIXEL_FORMAT_YUV_SEMIPLANAR_420 = 1
# 设置输入图片格式。
ret = acl.media.venc_set_channel_desc_param(self.venc_channel_desc,
ACL_VENC_PIXEL_FORMAT_UINT32, PIXEL_FORMAT_YUV_SEMIPLANAR_420)
# 获取输入图片格式。
get_pic_format, ret = acl.media.venc_get_channel_desc_param(self.venc_channel_desc,
ACL_VENC_PIXEL_FORMAT_UINT32)

width = 128
# 设置图片宽度。
ret = acl.media.venc_set_channel_desc_param(self.venc_channel_desc, ACL_VENC_PIC_WIDTH_UINT32,
width)
# 获取图片宽度。
get_width, ret = acl.media.venc_get_channel_desc_param(self.venc_channel_desc,
ACL_VENC_PIC_WIDTH_UINT32)
```



```
height = 128
# 设置图片高度。
ret = acl.media.venc_set_channel_desc_param(self.venc_channel_desc, ACL_VENC_PIC_HEIGHT_UINT32,
height)
# 获取图片高度。
get_height, ret = acl.media.venc_get_channel_desc_param(self.venc_channel_desc,
ACL_VENC_PIC_HEIGHT_UINT32)
```

5.4 媒体数据处理 V2

5.4.1 功能支持度说明

版本对媒体数据处理V2版本各功能的支持度如下表所示。

版本	VPC	JPEGD	JPEGE	PNGD	VDEC	VENC
Atlas 推理系列产品（Ascend 310P处理器）	√	√	√	√	√	√
Atlas 200/500 A2 推理产品	√	√	√	√	√	√
Atlas A2训练系列产品	√	√	√	√	√	x

5.4.2 VPC 图片处理典型功能

VPC（Vision Preprocessing Core）负责图像处理功能，支持对图片做抠图、缩放、格式转换等操作。关于VPC功能的详细介绍请参见VPC功能，关于VPC功能对输入、输出的约束要求，请参见约束说明。

本节以抠图、缩放为例说明VPC图像处理时的接口调用流程，同时配合以下典型功能的示例代码辅助理解该接口调用流程：

- 图片缩放示例代码
- 抠图示例代码

须知

Atlas 200/300/500 推理产品上，当前版本不支持该功能。

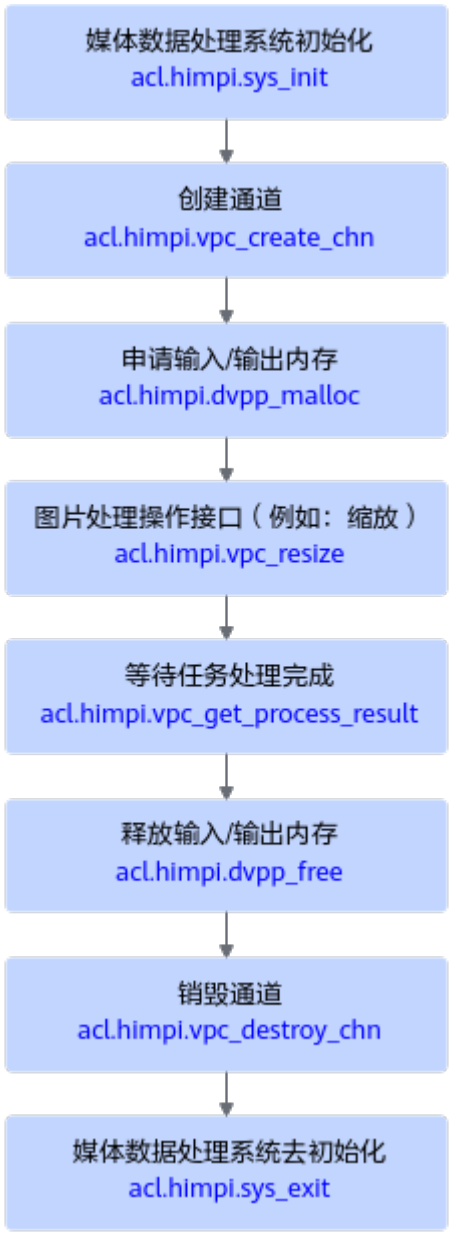
Atlas 训练系列产品上，当前版本不支持该功能。

典型功能接口调用流程（以缩放为例）

开发应用时，如果涉及抠图、缩放等图片处理，则应用程序中必须包含图片处理的代码逻辑，关于图片处理的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

如果在Host上调用DVPP接口，图像处理的结果数据都在Device的内存中，如果想访问结果数据，需要将结果数据传输回Host侧。

图 5-11 接口调用流程（以缩放为例）



当前系统支持对输入图片做抠图、缩放等处理，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.vpc_create_chn接口创建通道。

3. 调用acl.himpi.dvpp_malloc接口申请Device上的内存，存放输入或输出数据。
4. 执行抠图、缩放等，此步骤以缩放为例说明，其它功能（例如格式转换、金字塔等）请参见VPC功能下的接口说明。

   调用acl.himpi.vpc_resize接口，按指定区域从输入图片中抠图，抠出的图作为输出图片。acl.himpi.vpc_resize接口是异步接口，调用该接口成功仅表示任务下发成功，还需要调用acl.himpi.vpc_get_process_result接口等待任务完成。

   - 可以跟acl.himpi.vpc_resize接口在同一个线程中调用acl.himpi.vpc_get_process_result接口，也可以新起一个线程调用acl.himpi.vpc_get_process_result接口，后者多线程并行，提高效率，但用户需自行实现线程间同步。
   - 在实现抠图、缩放等功能时，通过将输入图片和输出图片的格式设置成不同的，达到转换图片格式的目的。
   - 调用acl.rt.get_run_mode接口获取软件栈的运行模式，如果运行模式为ACL_HOST，且Host上需要展示VPC输出的图片数据，则需要申请Host内存，通过acl.rt.memcpy接口将Device的输出图片数据传输到Host；如果Host上不需要展示VPC输出的图片数据，则VPC的输出图片数据可以直接作为模型推理的输入，模型推理的相关介绍请参见**4.6 模型推理基本场景**、**7.7 模型动态AIPP推理**。
5. 调用acl.himpi.dvpp_free接口释放输入、输出内存。
6. 调用acl.himpi.vpc_destroy_chn接口销毁通道。
7. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

图片缩放示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
channel_id = 0
chn_attr = {'attr': 0, 'pic_width': 0, 'pic_height': 0}
ret = acl.himpi.vpc_create_chn(channel_id, chn_attr)

# 6.执行缩放。
# 6.1 构造存放输入图片信息的字典。
input_pic = {'picture_width': 1920,
             'picture_height': 1080,
             'picture_width_stride': 1920,
             'picture_height_stride': 1080,
             'picture_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420}
input_pic["picture_buffer_size"] = input_pic["picture_width_stride"] * input_pic["picture_height_stride"] * 3 //
2
picture_address, ret = acl.himpi.dvpp_malloc(0, input_pic["picture_buffer_size"])
input_pic["picture_address"] = picture_address

# 如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device
```



```
内存。
# 直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 将输入图片读入内存中。
    vpc_file = np.fromfile(vpc_file_path, dtype=np.byte)
    vpc_file_size = vpc_file.itemsize * vpc_file.size

    bytes_data = vpc_file.tobytes()
    vpc_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_pic["picture_address"], input_pic["picture_buffer_size"],
                        vpc_file_ptr, vpc_file_size, ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    vpc_file = np.fromfile(vpc_file_path, dtype=np.byte)
    vpc_file_size = vpc_file.itemsize * vpc_file.size

    bytes_data = vpc_file.tobytes()
    vpc_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_pic["picture_address"], input_pic["picture_buffer_size"],
                        vpc_file_ptr, vpc_file_size, ACL_MEMCPY_DEVICE_TO_DEVICE)


# 6.3 构造存放输出图片信息的字典。
output_pic = {'picture_width': 960,
              'picture_height': 540,
              'picture_width_stride': 960,
              'picture_height_stride': 540,
              'picture_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420}
output_pic["picture_buffer_size"] = output_pic["picture_width_stride"] * output_pic["picture_height_stride"]
* 3 // 2
picture_address, ret = acl.himpi.dvpp_malloc(0, output_pic["picture_buffer_size"])
output_pic["picture_address"] = picture_address
# 初始化内存。
ret = acl.rt.memset(output_pic["picture_address"], output_pic["picture_buffer_size"], 0,
output_pic["picture_buffer_size"])

# 6.4 调用缩放接口。
testid, ret = acl.himpi.vpc_resize(channel_id, input_pic, output_pic, 0, 0, 0, -1)

# 6.5 等待任务处理结束，任务处理结束后，输出图片数据在outputPic.picture_address指向的内存中。
ret = acl.himpi.vpc_get_process_result(channel_id, task_id, -1)

# 6.6 如果运行模式为ACL_HOST，且Host上需要展示VPC输出的图片数据，则需要申请Host内存，通过
acl.rt.memcpy接口将Device的输出图片数据传输到Host；如果Host上不需要展示VPC输出的图片数据，则VPC的
输出图片数据可以直接作为模型推理的输入。
# 6.6 VPC的输出图片数据可以直接作为模型推理的输入。
if run_mode == ACL_HOST:
        np_output = np.zeros(output_pic.get('picture_buffer_size'), dtype=np.byte)

        bytes_data = np_output.tobytes()
        np_output_ptr = acl.util.bytes_to_ptr(bytes_data)
        ret = acl.rt.memcpy(np_output_ptr, output_pic.get('picture_buffer_size'),
output_pic.get("picture_address"),
                            output_pic.get('picture_buffer_size'), ACL_MEMCPY_DEVICE_TO_HOST)
        # ......
else
    # 可以直接使用VPC的输出图片数据，在outputPic.picture_address指向的内存中。
    # ......
}

# 6.7 释放输入、输出内存。
ret = acl.himpi.dvpp_free(input_pic['picture_address'])
ret = acl.himpi.dvpp_free(output_pic['picture_address'])

# 7.销毁通道。
ret = acl.himpi.vpc_destroy_chn(channel_id)
```



```
# 8. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 9. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 10.pyACL去初始化。
ret = acl.finalize()
```

抠图示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
channel_id = 0
chn_attr = {'attr': 0, 'pic_width': 0, 'pic_height': 0}
ret = acl.himpi.vpc_create_chn(channel_id, chn_attr)

# 6.执行抠图。
# 6.1 构造存放输入图片信息的字典。
input_pic = {'picture_width': 1920,
             'picture_height': 1080,
             'picture_width_stride': 1920,
             'picture_height_stride': 1080,
             'picture_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420}
input_pic["picture_buffer_size"] = input_pic["picture_width_stride"] * input_pic["picture_height_stride"] * 3 //
2
picture_address, ret = acl.himpi.dvpp_malloc(0, input_pic["picture_buffer_size"])
input_pic["picture_address"] = picture_address

# 如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口
将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device
内存。
# 直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 将输入图片读入内存中。
    vpc_file = np.fromfile(vpc_file_path, dtype=np.byte)
    vpc_file_size = vpc_file.itemsize * vpc_file.size

    bytes_data = vpc_file.tobytes()
    vpc_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_pic["picture_address"], input_pic["picture_buffer_size"],
                        vpc_file_ptr, vpc_file_size, ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    vpc_file = np.fromfile(vpc_file_path, dtype=np.byte)
    vpc_file_size = vpc_file.itemsize * vpc_file.size

    bytes_data = vpc_file.tobytes()
    vpc_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, vpc_file_ptr, vpc_file_size, ACL_MEMCPY_DEVICE_TO_DEVICE)
```



```
# 6.3 构造存放输出图片信息的字典。
multithreading_count = 1
crop_region_infos = []
for i in range(multithreading_count):
    output_pic = {'picture_width': 960,
                  'picture_height': 540,
                  'picture_width_stride': 960,
                  'picture_height_stride': 540,
                  'picture_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420}
    output_pic["picture_buffer_size"] = output_pic["picture_width_stride"] *
output_pic["picture_height_stride"] * 3 // 2
    picture_address, ret = acl.himpi.dvpp_malloc(0, output_pic["picture_buffer_size"])
    output_pic["picture_address"] = picture_address

    ret = acl.rt.memset(output_pic["picture_address"], output_pic["picture_buffer_size"], 0,
output_pic["picture_buffer_size"])
    crop_region = {'top_offset': 0, 'left_offset': 0, 'crop_width': 960, 'crop_height': 540}
    crop_region_info = {'dest_pic_info': output_pic, 'crop_region': crop_region}
    crop_region_infos.append(crop_region_info)

# 6.4 调用抠图接口。
task_id, ret = acl.himpi.vpc_crop(channel_id, input_pic, crop_region_infos, 1, -1)

# 6.5 等待任务处理结束，任务处理结束后，输出图片数据在outputPic.picture_address指向的内存中。
ret = acl.himpi.vpc_get_process_result(channel_id, task_id, -1)

# 6.6 如果运行模式为ACL_HOST，且Host上需要展示VPC输出的图片数据，则需要申请Host内存，通过
acl.rt.memcpy接口将Device的输出图片数据传输到Host；如果Host上不需要展示VPC输出的图片数据，则VPC的
输出图片数据可以直接作为模型推理的输入。
# 6.6 VPC的输出图片数据可以直接作为模型推理的输入。
if run_mode == ACL_HOST:
    np_output = np.zeros(output_pic.get('picture_buffer_size'), dtype=np.byte)

    bytes_data = np_output.tobytes()
    np_output_ptr = acl.util.bytes_to_ptr(bytes_data)
    ret = acl.rt.memcpy(np_output_ptr, output_pic.get('picture_buffer_size'), output_pic.get("picture_address"),
                        output_pic.get('picture_buffer_size'), ACL_MEMCPY_DEVICE_TO_HOST)
    # ......
else
    # 可以直接使用VPC的输出图片数据，在outputPic.picture_address指向的内存中。
    # ......
}

# 6.7 释放输入、输出内存。
ret = acl.himpi.dvpp_free(input_pic['picture_address'])
ret = acl.himpi.dvpp_free(output_pic['picture_address']))

# 7.销毁通道。
ret = acl.himpi.vpc_destroy_chn(channel_id)

# 8. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 9. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 10.pyACL去初始化。
ret = acl.finalize()
```

5.4.3 JPEGD 图片解码

JPEGD（JPEG Decoder）负责完成图像解码功能，将.jpg、.jpeg、.JPG、.JPEG图片解码成YUV格式图片。关于JPEGD功能的详细介绍及约束请参见JPEGD功能及约束说明。

本节介绍JPEGD图片解码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

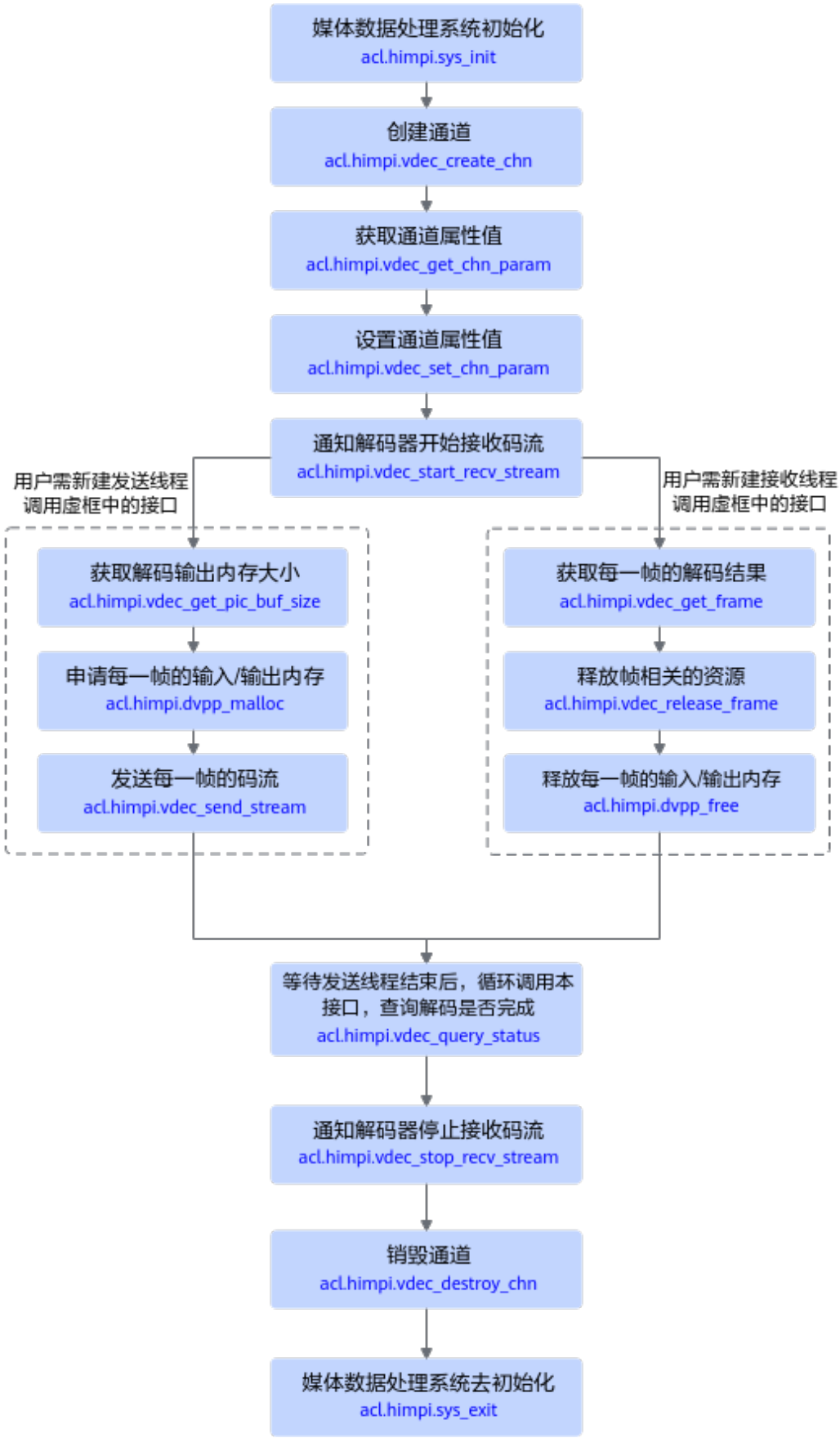
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

Atlas 训练系列产品上，当前版本不支持该功能。

接口调用流程

开发应用时，如果涉及对JPEG图片的解码，则应用程序中必须包含解码的代码逻辑，关于图片解码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-12 接口调用的流程



当前系统支持解码JPEG图片，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.vdec_create_chn接口创建通道。
3. 调用acl.himpi.dvpp_malloc接口申请Device上的内存，存放输入或输出数据。
4. 解码前，需调用acl.himpi.vdec_start_recv_stream接口通知解码器启动接收码流，再调用acl.himpi.vdec_send_stream接口发送解码码流，acl.himpi.vdec_send_stream接口是异步接口，调用该接口仅表示任务下发成功，还需要调acl.himpi.vdec_get_frame接口获取解码结果数据，成功获取解码数据后，可以调用acl.himpi.vdec_release_frame接口释放帧相关的资源。

   解码结束后，需调用acl.himpi.vdec_stop_recv_stream接口通知解码器停止接收码流。
5. 调用acl.himpi.dvpp_free接口释放输入、输出内存。
6. 调用acl.himpi.vdec_destroy_chn接口销毁通道。
7. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
channel_id = 0
attr = {'type': HI_PT_JPEG, 'mode': HI_VDEC_SEND_MODE_FRAME,
        'pic_width': 1920, 'pic_height': 1080,
        'stream_buf_size': 1920 * 1080, 'frame_buf_size': 0,
        'frame_buf_cnt': 9}

ret = acl.himpi.vdec_create_chn(channel_id, attr)
# 6.设置通道属性。
jpegd_param_dict, ret = acl.himpi.vdec_get_chn_param(channel_id)
jpegd_param_dict["pic_param"]["pixel_format"] = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420
jpegd_param_dict["pic_param"]["alpha"] = 255
jpegd_param_dict["display_frame_num"] = 0
ret = acl.himpi.vdec_set_chn_param(i, jpegd_param_dict)

# 7.解码器启动接收码流。
ret = acl.himpi.vdec_start_recv_stream(channel_id)

# 8.发送码流。
# 8.1 申请输入内存。
# input_size表示输入图片占用的内存大小，此处以1024 Byte为例，用户需根据实际情况计算内存大小。
input_size = 1024;
input_addr, ret = acl.himpi.dvpp_malloc(0, input_size);

#如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device内存。
```



```
#直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 申请Host内存。
    input_buffer, ret= acl.rt.malloc_host(input_size)
    # 将输入图片读入内存中。
    vdec_file = np.fromfile(vdec_file_path, dtype=np.byte)
    vdec_file_size = vdec_file.itemsize * vdec_file.size

    bytes_data = vdec_file.tobytes()
    vdec_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, vdec_file_ptr, vdec_file_size, ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    vdec_file = np.fromfile(vdec_file_path, dtype=np.byte)
    vdec_file_size = vdec_file.itemsize * vdec_file.size

    bytes_data = vdec_file.tobytes()
    vdec_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, vdec_file_ptr, vdec_file_size,
ACL_MEMCPY_DEVICE_TO_DEVICE)

# 8.2 构造存放输入图片信息的字典。
stream = {'end_of_frame': HI_TRUE, 'end_of_stream': HI_FALSE,
          'need_display': HI_TRUE, 'pts': 0,
          'len': input_size, 'addr': vdec_file_ptr}
img_info, ret = acl.himpi.dvpp_get_image_info(HI_PT_JPEG, stream)
if run_mode == ACL_HOST:
    # 如果不使用Host上的数据，需及时释放。
    ret = acl.rt.free_host(input_buffer)

stream['addr'] =  input_addr

# 8.3 构造存放输出图片信息的字典，并申请输出内存。
out_pic_info = {"width": img_info['width'],
                "height": img_info['height'],
                "width_stride": img_info['width_stride'],
                "height_stride": img_info['height_stride'],
                "pixel_format": HI_PIXEL_FORMAT_UNKNOWN,
                "buffer_size": img_info['img_buf_size']}
out_buffer, ret = acl.himpi.dvpp_malloc(0, out_pic_info['buffer_size'])
out_pic_info['vir_addr '] = out_buffer

# 8.4 发送需解码的输入图片。
ret = acl.himpi.vdec_send_stream(channel_id, stream, out_pic_info, 0)

# 9.接收解码结果。
# 9.1 获取解码结果。
frame_info, supplement, stream, ret = acl.himpi.vdec_get_frame(channel_id, 0)
if ret == 0:
    dec_result = frame_info['v_frame']['frame_flag']
    if dec_result == 0: #  0: Decode success
        print("Chn %u GetFrame Success, Decode Success \n"%channel_id)
    else: # Decode fail
        print("Chn %u GetFrame Success, Decode Fail \n"%channel_id)

# 9.2 如果运行模式为ACL_HOST，且Host上需要展示JPEGD输出的图片数据，则需要申请Host内存，通过
aclrtMemcpy接口将Device的输出图片数据传输到Host。
# 9.2 获取JPEGD的输出图片数据。

if run_mode == ACL_HOST:
    # 申请Host内存。
    output_buffer, ret= acl.rt.malloc_host(out_pic_info['buffer_size'])
    # 数据传输。
    ret = acl.rt.memcpy(output_buffer, out_pic_info['buffer_size'], frame_info['v_frame']['virt_addr'][0],
out_pic_info['buffer_size'], ACL_MEMCPY_DEVICE_TO_HOST)

    # ......
```



```
    # 数据使用完成后，及时释放不使用的内存。
    ret = acl.rt.free_host(output_buffer)
else:
    # 可以直接使用JPEGD的输出图片数据，在outputPic.picture_address指向的内存中。
    # ......

# 9.3 释放输入、输出内存。
ret = acl.himpi.dvpp_free(frame_info['v_frame']['virt_addr'][0])
ret = acl.himpi.dvpp_free(stream['addr'])

# 9.4 释放资源。
ret = acl.himpi.vdec_release_frame(channel_id, frame_info)

# 10.解码器停止接收码流。
ret = acl.himpi.vdec_stop_recv_stream(channel_id)

# 7.销毁通道。
ret = acl.himpi.vdec_destroy_chn(channel_id)

# 8. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 9. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 10.pyACL去初始化。
ret = acl.finalize()
```

5.4.4 JPEGE 图片编码

JPEGE（JPEG Encoder）负责完成图像编码功能，将YUV格式图片编码成.jpg图片。关于JPEGE功能的详细介绍请参见JPEGE功能及约束说明。

本节介绍JPEGE图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

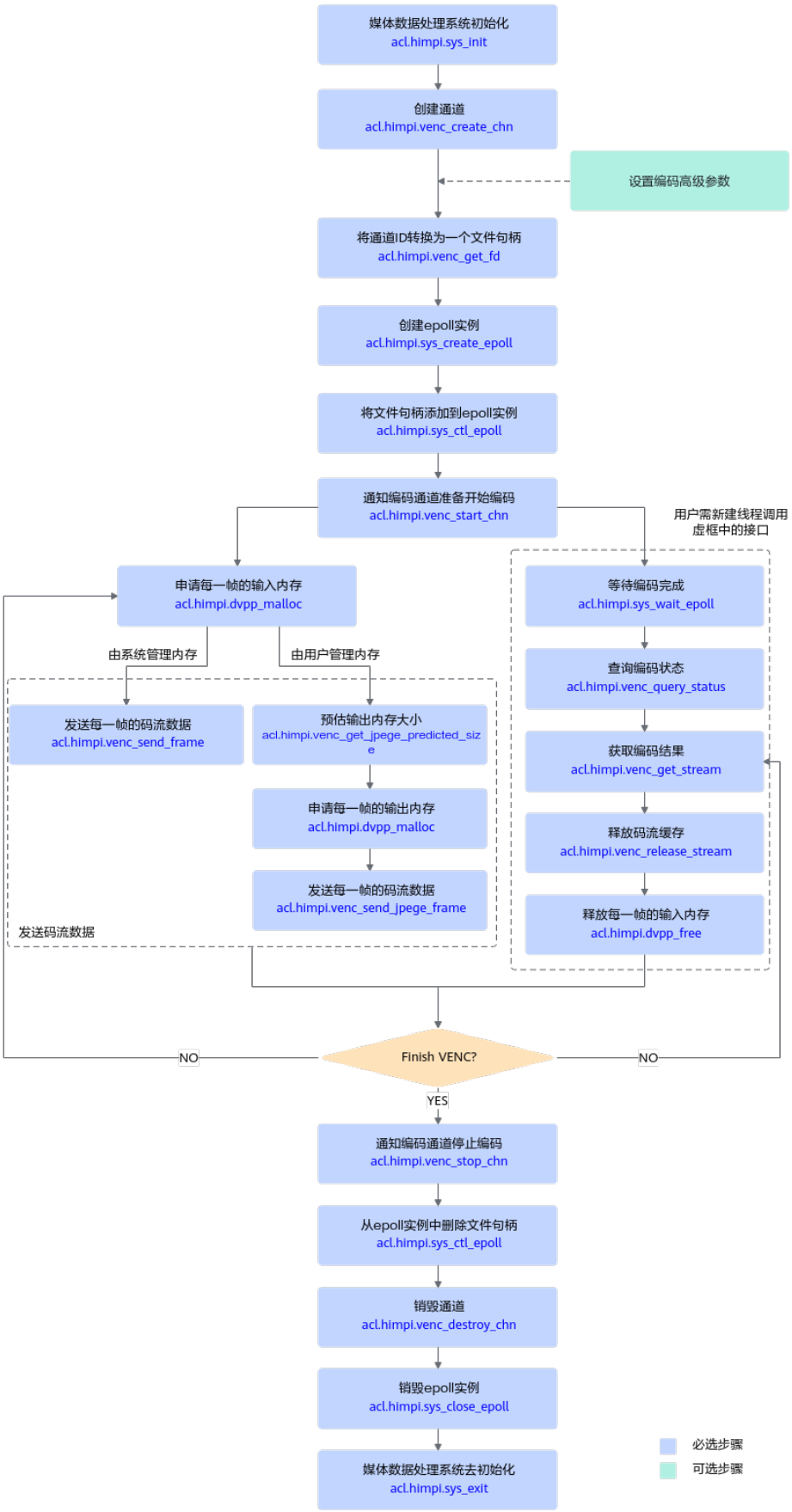
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

Atlas 训练系列产品上，当前版本不支持该功能。

接口调用流程

开发应用时，如果涉及将YUV格式图片编码成JPEG压缩格式的图片文件，则应用程序中必须包含编码的代码逻辑，关于编码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-13 接口调用流程



当前系统支持将YUV格式图片编码成JPEG压缩格式的图片文件，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.venc_create_chn函数创建完通道。

   成功创建通道之后，您可以根据实际需求设置编码的高级参数，例如场景模式、码流控制器的高级参数等，请参见acl.himpi.venc_set_jpeg_param~acl.himpi.venc_compact_jpeg_tables章节中的接口说明。
3. 调用acl.himpi.venc_get_fd将通道ID转换为一个文件句柄。
4. 调用acl.himpi.sys_create_epoll函数创建DVPP epoll实例。
5. 调用acl.himpi.sys_ctl_epoll函数将编码通道的文件句柄添加到epoll实例中，由epoll实例处理。

   select或者poll方式，不需要执行该步骤。
6. 调用acl.himpi.venc_start_chn函数通知通道准备开始编码。
7. 调用acl.himpi.dvpp_malloc接口申请存放Device上输入数据的内存。
8. 启动一个用户态线程，调用acl.himpi.sys_wait_epoll函数等待编码完成。
9. 之后用户就可以调用acl.himpi.venc_send_frame函数发送待编码的码流。
10. 一旦编码完成，acl.himpi.sys_wait_epoll函数或select函数或poll函数就会返回，用户就可以调用acl.himpi.venc_query_status接口查询编码状态，再调用acl.himpi.venc_get_stream函数获取编码结果。
11. 用户需要注意的是，编码结果数据使用完成之后，需要及时调用acl.himpi.venc_release_stream函数释放buffer。否则会因编码buffer用完导致后续编码无法进行。
12. 调用acl.himpi.dvpp_free接口释放输入内存。
13. 当用户不需发送图像到目的通道继续编码时，需要调用acl.himpi.venc_stop_chn函数通知该通道不再接收新的输入图片。
14. 调用acl.himpi.sys_ctl_epoll函数从epoll实例中删除编码通道的文件句柄。
15. 当用户完成所有编码之后，需要调用acl.himpi.venc_destroy_chn释放编码通道以及内部内存资源。
16. 调用acl.himpi.sys_close_epoll函数销毁DVPP epoll实例。
17. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

说明

支持DVPP内部管理输出内存，或用户自行管理输出内存两种方式：

- 不需要用户管理、由DVPP内部管理输出内存时，调用acl.himpi.venc_send_frame接口发送原始图像进行图像编码。

  在调用acl.himpi.venc_create_chn接口创建通道时，必须正确设置hi_venc_chn_attr["venc_attr"]["buf_size"]参数值（参数描述请参见hi_venc_attr）。

  该方式下，相比由用户管理内存，输出结果数据的JPEG头中不存在COM注释字段，数据长度会短一点，但需要用户从DVPP返回的内存中拷贝输出结果数据到指定内存。

- 由用户自行管理输出内存、管理内存的生命周期，调用acl.himpi.venc_send_jpege_frame接口发送原始图像进行图像编码。

  在调用acl.himpi.venc_create_chn接口创建通道时，需将hi_venc_chn_attr["venc_attr"]["buf_size"]参数值设置为0（参数描述请参见hi_venc_attr），然后调用acl.himpi.venc_get_jpege_predicted_size接口预估输出内存大小，调用acl.himpi.dvpp_malloc/acl.himpi.dvpp_free接口申请/释放输出内存。

  该方式下，直接在调用acl.himpi.venc_send_jpege_frame接口时，设置输出内存地址，输出结果数据直接存放到用户设置的内存中，**相比由系统管理内存的方式，用户可减少一次“从DVPP返回的内存中拷贝输出结果数据到指定内存”的操作，但输出结果数据的JPEG头中可能会存在COM注释字段（字段长度范围4~19Byte），数据长度会长一点。**

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
channel_id = 0
venc_attr = {'type': HI_PT_JPEG, 'profile': 0,
             'max_pic_width': 128, 'pic_width': 128,
             'max_pic_height': 128, 'pic_height': 128,
             'buf_size': 1024 * 1024 * 2, 'is_by_frame': 1}
jpeg_attr = {'recv_mode':HI_VENC_PIC_RECV_SINGLE,
           'mpf_cfg':{'large_thumbnail_num ': 0}}
attr = {'venc_attr':venc_attr, 'jpeg_attr':jpeg_attr}
ret = acl.himpi.venc_create_chn(channel_id, attr)

# 6.设置JPEGE参数。
param, ret = acl.himpi.venc_get_jpeg_param(channel_id)
param['qfactor'] = 100
ret = acl.himpi.venc_set_jpeg_param(channel_id, param)

# 7.通知编码器开始接收输入数据。
recv_param = {'recv_pic_num': -1}
ret = acl.himpi.venc_start_chn(channel_id, recv_param)

# 8.发送输入数据。
# 8.1 申请输入内存。
input_size = 128 * 128 * 3 // 2
input_addr, ret = acl.himpi.dvpp_malloc(0, input_size);
```



```
# 如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device内存。
# 直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_filee_path, dtype=np.byte)
    jpege_file_size = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_file_path, dtype=np.byte)
    jpege_file = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_DEVICE_TO_DEVICE)

# 8.2 发送输入数据，开始编码。
v_frame = {'width': 128,
           'height': 128,
           'field': HI_VIDEO_FIELD_FRAME,
           'pixel_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420,
           'video_format': HI_VIDEO_FORMAT_LINEAR,
           'compress_mode': HI_COMPRESS_MODE_NONE,
           'dynamic_range': HI_DYNAMIC_RANGE_SDR8,
           'color_gamut': HI_COLOR_GAMUT_BT709,
           'header_stride': [0, 0, 0],
           'width_stride': [128, 0, 0],
           'height_stride': [0, 0, 0],
           'header_phys_addr': [0, 0, 0],
           'phys_addr': [0, 0, 0],
           'header_virt_addr': [0, 0, 0],
           'virt_addr': [input_addr, 0, 0],
           'time_ref': 0,'pts': cur_time}
frame = {'v_frame':v_frame, 'pool_id':0, 'mod_id':HI_ID_VGS}
ret = acl.himpi.venc_send_frame(channel_id, frame, 0)

# 9.获取编码结果。
# 9.1 通过EPOLL处理编码完成事件。
fd = acl.himpi.venc_get_fd(channel_id)
epoll_fd, ret = acl.himpi.sys_create_epoll(10)

event['data'] = fd
event['events'] = HI_DVPP_EPOLL_IN
ret = acl.himpi.sys_ctl_epoll(epoll_fd, HI_DVPP_EPOLL_CTL_ADD, fd, event)

# 编码完成前，会超时阻塞在这里，一旦完成，才会往下执行。
events, eventCount, ret = acl.himpi.sys_wait_epoll(epoll_fd, 3, 1000);

# 9.2 获取编码结果。
status, ret = acl.himpi.venc_query_status(channel_id)
stream = {'pack_cnt': status['cur_packs']}
stream, ret = acl.himpi.venc_get_stream(self.channel_id, stream, 1000)
# 9.3 如果运行模式为ACL_HOST，且Host上需要使用编码输出的码流，则需要申请Host内存，通过acl.rt.memcpy接口将Device的输出码流传输到Host；否则直接使用编码输出码流数据。
# 9.3 获取编码输出码流数据。
if run_mode == ACL_HOST:
    # 申请Host内存。
    output_buffer, ret= acl.rt.malloc_host(output_ize)
    # 数据传输。
    ret = acl.rt.memcpy(output_buffer, output_ize, stream['pack'][0]['addr'], output_ize,
```



```
ACL_MEMCPY_DEVICE_TO_HOST)
    # ......
    # 数据使用完成后，及时释放不使用的内存。
    ret = acl.rt.free_host(output_buffer)
else:
    # 可以直接使用编码输出码流数据，在stream['pack'][0]['addr']指向的内存中。
    # ......

# 10.释放输入内存和输出码流。
ret = acl.himpi.dvpp_free(input_addr)
ret = acl.himpi.venc_release_stream(channel_id, stream)

# 11.通知编码器停止接收输入数据。
ret = acl.himpi.venc_stop_chn(channel_id)
ret = acl.himpi.sys_ctl_epoll(epoll_fd, HI_DVPP_EPOLL_CTL_DEL, fd, event)
ret = acl.himpi.sys_close_epoll(epoll_fd)

# 12.销毁通道。
ret = acl.himpi.venc_destroy_chn(channel_id)

# 13. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 14. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 15.pyACL去初始化。
ret = acl.finalize()
```

5.4.5 PNGD 图片解码

PNGD（PNG decoder）负责PNG格式图片的解码。关于PNGD功能的详细介绍请参见PNGD图像处理接口。

本节介绍PNGD图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

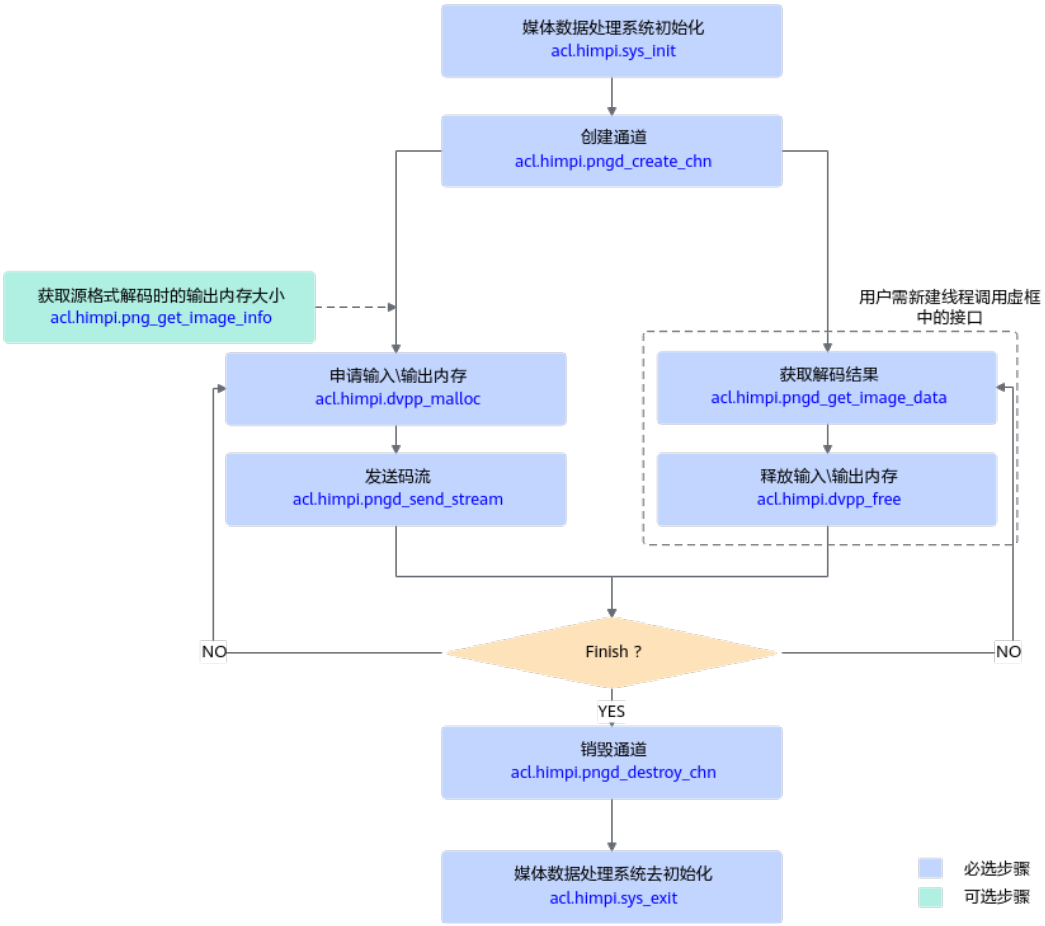
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

Atlas 训练系列产品上，当前版本不支持该功能。

接口调用流程

开发应用时，如果涉及对PNG图片的解码，则应用程序中必须包含解码的代码逻辑，关于图片解码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-14 接口调用流程



当前系统支持解码PNG图片，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.pngd_create_chn接口创建通道。
3. 调用acl.himpi.dvpp_malloc接口申请Device上的内存，存放输入或输出数据。
4. 调用acl.himpi.pngd_send_stream接口发送解码码流，acl.himpi.pngd_send_stream接口是异步接口，调用该接口仅表示任务下发成功，还需要调acl.himpi.pngd_get_image_data接口获取解码结果数据。
5. 调用acl.himpi.dvpp_free接口释放输入、输出内存。
6. 调用acl.himpi.pngd_destroy_chn接口销毁通道。
7. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()
```



```
# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
attr = {"stream_que_cnt": STREAM_QUE_CNT}
channel_id = 0
ret = acl.himpi.pngd_create_chn(channel_id, attr)

# 6.发送码流。
# 6.1 申请输入内存。
input_addr, ret = acl.himpi.dvpp_malloc(0, input_size);

# 如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device内存。
# 直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_filee_path, dtype=np.byte)
    jpege_file_size = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_file_path, dtype=np.byte)
    jpege_file = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_DEVICE_TO_DEVICE)

# 6.2 构造存放输入图片信息的字典。
stream = {"type": HI_PT_PNG, "addr": input_addr, "len": input_size, "pts": 0}
img_info, ret = acl.himpi.png_get_image_info(stream)

# 6.3 构造存放输出图片信息的字典，并申请输出内存。
output_addr, ret = acl.himpi.dvpp_malloc(0, img_info['img_buf_size']);
out_pic_info =  {"picture_address": output_addr,
                 "picture_buffer_size": img_info['img_buf_size'],
                 "picture_width": img_info['width'],
                 "picture_height": img_info['height'],
                 "picture_width_stride": img_info['width_stride'],
                 "picture_height_stride": img_info['height_stride'],
                 "picture_format": HI_PIXEL_FORMAT_UNKNOWN}
ret = acl.himpi.pngd_send_stream(channel_id, stream, out_pic_info, 0)

# 7.接收解码结果。
# 7.1 获取解码结果。
pic_info, get_stream, ret = acl.himpi.pngd_get_image_data(channel_id, -1)
if ret == 0: #  Decode success
    print("Chn %u GetFrame Success, Decode Success \n"%channel_id)
elif ret == HI_ERR_PNGD_BUF_EMPTY: # Decoding
    print("Chn %u Decoding, try again \n"%channel_id)
else: # Decode Fail
    print("Chn %u GetFrame Success, Decode Fail \n"%channel_id)

# 7.2 如果运行模式为ACL_HOST，且Host上需要展示PNGD输出的图片数据，则需要申请Host内存，通过acl.rt.memcpy接口将Device的输出图片数据传输到Host。
# 7.2 获取PNGD的输出图片数据。
```



```
if run_mode == ACL_HOST:
    # 申请Host内存。
    output_buffer, ret= acl.rt.malloc_host(output_ize)
    # 数据传输。
    ret = acl.rt.memcpy(output_buffer, output_ize, stream['pack'][0]['addr'], output_ize,
ACL_MEMCPY_DEVICE_TO_HOST)
    # ......
    # 数据使用完成后，及时释放不使用的内存。
    ret = acl.rt.free_host(output_buffer)
else:
    # 可以直接使用PNGD的输出图片数据，在outputPic.picture_address指向的内存中。
    # ......

# 7.3 释放输入内存和输出码流。
ret = acl.himpi.dvpp_free(input_addr)
ret = acl.himpi.dvpp_free(output_addr)

# 8.销毁通道。
ret = acl.himpi.pngd_destroy_chn(channel_id)

# 9. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 10. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 11.pyACL去初始化。
ret = acl.finalize()
```

5.4.6 VDEC 视频解码

VDEC（Video Decoder）负责将H264/H265格式的视频码流解码为YUV/RGB格式的图片。关于VDEC功能的详细介绍及约束请参见VDEC功能及约束说明。

本节介绍VDEC视频编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

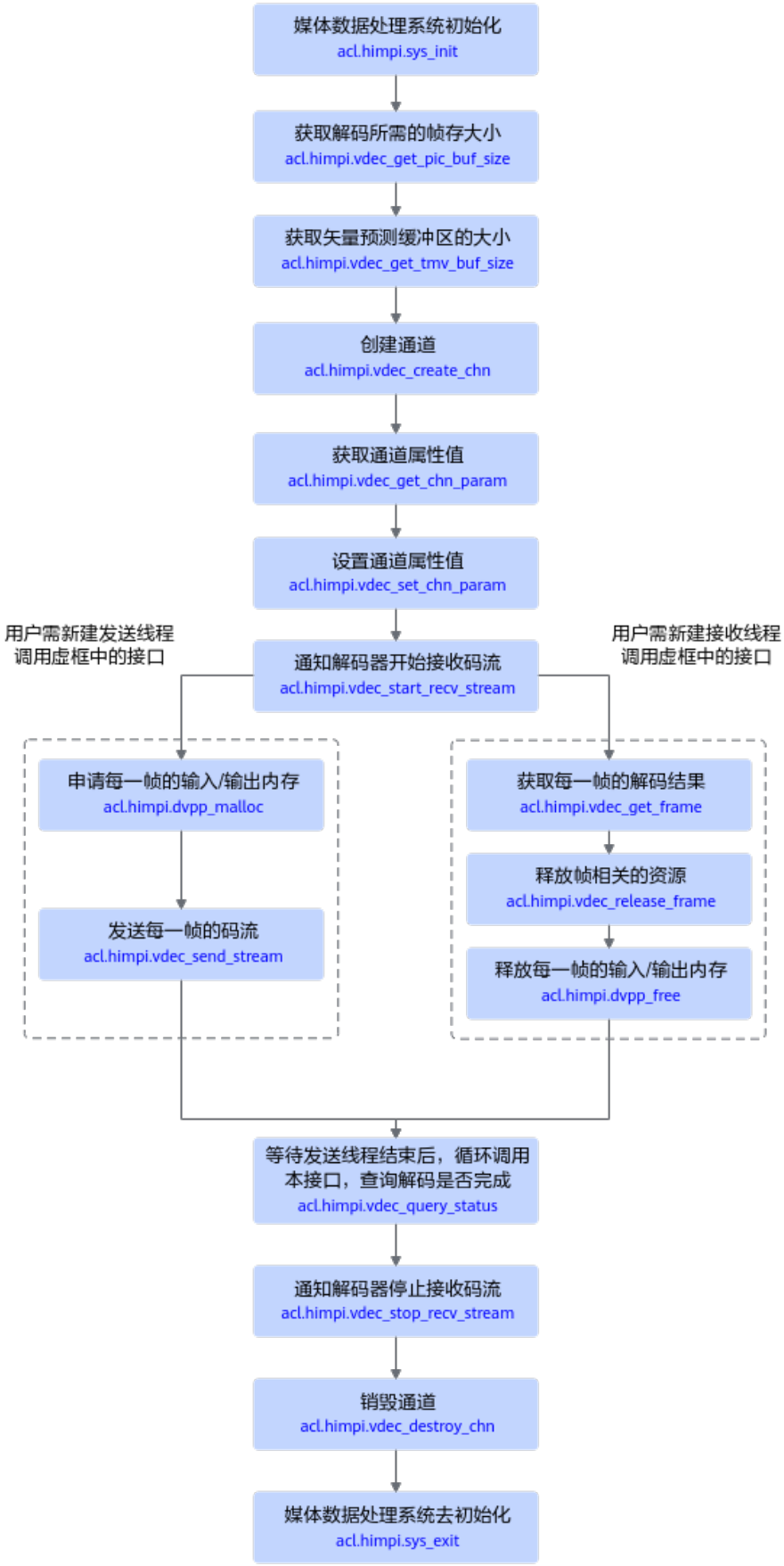
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

Atlas 训练系列产品上，当前版本不支持该功能。

接口调用流程

开发应用时，如果涉及视频解码，则应用程序中必须包含解码的代码逻辑，关于视频解码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-15 接口调用的流程



当前系统支持解码H264/H265的视频码流，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.vdec_get_pic_buf_size接口获取解码所需的帧存大小、调用acl.himpi.vdec_get_tmv_buf_size接口获取矢量预测缓冲区的大小，在创建通道时需要使用这些数据。
3. 调用acl.himpi.vdec_create_chn接口创建通道。
4. 调用acl.himpi.dvpp_malloc接口申请Device上的内存，存放输入或输出数据。
5. 解码前，需调用acl.himpi.vdec_start_recv_stream接口通知解码器启动接收码流，再调用acl.himpi.vdec_send_stream接口发送解码码流，acl.himpi.vdec_send_stream接口是异步接口，调用该接口仅表示任务下发成功，还需要调acl.himpi.vdec_get_frame接口获取解码结果数据，成功获取解码数据后，可以调用acl.himpi.vdec_release_frame接口释放帧相关的资源。

   解码结束后，需调用acl.himpi.vdec_stop_recv_stream接口通知解码器停止接收码流。
6. 调用acl.himpi.dvpp_free接口释放输入、输出内存。
7. 调用acl.himpi.vdec_destroy_chn接口销毁通道。
8. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.创建通道。
channel_id = 0
buf_attr = [1920, 1080, 0, 8, HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420, HI_COMPRESS_MODE_NONE]
tmv_buf_size = acl.himpi.vdec_get_tmv_buf_size(HI_PT_H264, 1920, 1080)
attr = {'type': HI_PT_H264, 'mode': HI_VDEC_SEND_MODE_FRAME,
        'pic_width': 1920, 'pic_height': 1080,
        'stream_buf_size': 1920 * 1080 * 3 // 2, 'frame_buf_size': 0,
        'frame_buf_cnt': 16, 'video_attr': {'ref_frame_num': 12, 'temporal_mvp_en': HI_TRUE,
                                            'tmv_buf_size': tmv_buf_size}}
ret = acl.himpi.vdec_create_chn(channel_id, attr)

# 6.设置通道属性。
video_param_dict, ret = acl.himpi.vdec_get_chn_param(channel_id)
video_param_dict["video_param"]["dec_mode "] = HI_VIDEO_DEC_MODE_IPB
video_param_dict["video_param"]["compress_mode "] = HI_COMPRESS_MODE_HFBC
video_param_dict["video_param"]["video_format "] = HI_VIDEO_FORMAT_TILE_64x16
video_param_dict["video_param"]["out_order"] = HI_VIDEO_OUT_ORDER_DISPLAY
video_param_dict["display_frame_num"] = 3
ret = acl.himpi.vdec_set_chn_param(i, video_param_dict)

# 7.解码器启动接收码流。
ret = acl.himpi.vdec_start_recv_stream(channel_id)
```



```
# 8.发送码流。
# 8.1 申请输入内存。
# input_size表示输入图片占用的内存大小，此处以1024 Byte为例，用户需根据实际情况计算内存大小。
input_size = 1024;
input_addr, ret = acl.himpi.dvpp_malloc(0, input_size);

#如果运行模式为ACL_HOST，则需要申请Host内存，将输入图片数据读入Host内存，再通过acl.rt.memcpy接口将Host的图片数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入图片数据读入Device内存。
#直接将输入图片数据读入Device内存。
if run_mode == ACL_HOST:
    # 申请Host内存。
    input_buffer, ret= acl.rt.malloc_host(input_size)
    # 将输入图片读入内存中。
    vdec_file = np.fromfile(vdec_file_path, dtype=np.byte)
    vdec_file_size = vdec_file.itemsize * vdec_file.size

    bytes_data = vdec_file.tobytes()
    vdec_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, vdec_file_ptr, vdec_file_size, ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    vdec_file = np.fromfile(vdec_file_path, dtype=np.byte)
    vdec_file_size = vdec_file.itemsize * vdec_file.size

    bytes_data = vdec_file.tobytes()
    vdec_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, vdec_file_ptr, vdec_file_size,
ACL_MEMCPY_DEVICE_TO_DEVICE)

# 8.2 申请输出内存。
output_size = 1920 * 1080 * 3 // 2
output_addr, ret = acl.himpi.dvpp_malloc(0, output_size);

# 8.3 构造存放一帧输入码流信息的字典。
stream = {'end_of_frame': HI_TRUE, 'end_of_stream': HI_FALSE,
          'need_display': HI_TRUE, 'pts': 0,
          'len': input_size, 'addr': input_addr}
# 8.4 构造存放一帧输出结果信息的字典。
out_pic_info = {"width": 1920,
                "height": 1080,
                "width_stride": 1920,
                "height_stride": 1080,
                "pixel_format": HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420,
                'vir_addr': output_addr,
                "buffer_size": output_size}

# 8.4 发送一帧码流。
ret = acl.himpi.vdec_send_stream(channel_id, stream, out_pic_info, 0)

# 9.接收解码结果。
# 9.1 获取解码结果。
frame_info, supplement, stream, ret = acl.himpi.vdec_get_frame(channel_id, 0)
if ret == 0:
    dec_result = frame_info['v_frame']['frame_flag']
    if dec_result == 0: #  0: Decode success
        print("Chn %u GetFrame Success, Decode Success \n"%channel_id)
    elif dec_result == 1: # 1:Decode fail
        print("Chn %u GetFrame Success, Decode Fail \n"%channel_id)
    elif dec_result == 2: # 2:This result is returned for the second field of
        print("Chn %u GetFrame Success, No Picture \n"%channel_id)
    elif dec_result == 3: # 3: Reference frame number set error
        print("Chn %u GetFrame Success, RefFrame Num Error \n"%channel_id)
    elif dec_result == 4: # 4: Reference frame size set error
        print("Chn %u GetFrame Success, RefFrame Size Error \n"% channel_id)
# 9.2 如果运行模式为ACL_HOST，且Host上需要展示VDEC输出的图片数据，则需要申请Host内存，通过acl.rt.memcpy接口将Device的输出图片数据传输到Host。
```



```
# 9.2 获取解码结果数据。
if run_mode == ACL_HOST:
    # 申请Host内存。
    output_buffer, ret= acl.rt.malloc_host(out_pic_info['buffer_size'])
    # 数据传输。
    ret = acl.rt.memcpy(output_buffer, out_pic_info['buffer_size'], frame_info['v_frame']['virt_addr'][0],
out_pic_info['buffer_size'], ACL_MEMCPY_DEVICE_TO_HOST)
    # ......
    # 数据使用完成后，及时释放不使用的内存。
    ret = acl.rt.free_host(output_buffer)
else:
    # 可以直接使用JPEGD的输出图片数据，在outputPic.picture_address指向的内存中。
    # ......

# 9.3 释放输入、输出内存。
ret = acl.himpi.dvpp_free(frame_info['v_frame']['virt_addr'][0])
ret = acl.himpi.dvpp_free(stream['addr'])

# 9.4 释放资源。
ret = acl.himpi.vdec_release_frame(channel_id, frame_info)

# 10.解码器停止接收码流。
ret = acl.himpi.vdec_stop_recv_stream(channel_id)

# 7.销毁通道。
ret = acl.himpi.vdec_destroy_chn(channel_id)

# 8. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 9. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 10.pyACL去初始化。
ret = acl.finalize()
```

5.4.7 VENC 视频编码

VENC（Video Encoder）将YUV420SP格式的图片编码成H264/H265格式的视频码流。关于VENC功能的详细介绍请参见VENC功能及约束说明。

本节介绍VENC视频编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

须知

Atlas 200/300/500 推理产品上，当前版本不支持该功能。

Atlas 训练系列产品上，当前版本不支持该功能。

Atlas A2训练系列产品上，不支持该功能。

接口调用流程

开发应用时，如果涉及视频编码，则应用程序中必须包含编码的代码逻辑，关于编码的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

图 5-16 接口调用流程

当前系统支持H264/H256格式的视频码流，关键接口的说明如下：

1. 调用acl.himpi.sys_init接口进行媒体数据处理系统初始化。
2. 调用acl.himpi.venc_create_chn函数创建完通道。

   成功创建通道之后，您可以根据实际需求设置编码的高级参数，例如场景模式、码流控制器的高级参数等，请参见acl.himpi.venc_set_jpeg_param~acl.himpi.venc_compact_jpeg_tables章节中的接口说明。
3. 调用acl.himpi.venc_get_fd将通道ID转换为一个文件句柄。
4. 调用acl.himpi.sys_create_epoll函数创建DVPP epoll实例。
5. 调用acl.himpi.sys_ctl_epoll函数将编码通道的文件句柄添加到epoll实例中，由epoll实例处理。

   select或者poll方式，不需要执行该步骤。
6. 调用acl.himpi.venc_start_chn函数通知通道准备开始编码。
7. 调用acl.himpi.dvpp_malloc接口申请存放Device上输入数据的内存。
8. 启动一个用户态线程，调用acl.himpi.sys_wait_epoll函数等待编码完成。
9. 之后用户就可以调用acl.himpi.venc_send_frame函数发送待编码的码流。
10. 一旦编码完成，acl.himpi.sys_wait_epoll函数或select函数或poll函数就会返回，用户就可以调用acl.himpi.venc_query_status接口查询编码状态，再调用acl.himpi.venc_get_stream函数获取编码结果。
11. 用户需要注意的是，编码结果数据使用完成之后，需要及时调用acl.himpi.venc_release_stream函数释放buffer。否则会因编码buffer用完导致后续编码无法进行。
12. 调用acl.himpi.dvpp_free接口释放输入内存。
13. 当用户不需发送图像到目的通道继续编码时，需要调用acl.himpi.venc_stop_chn函数通知该通道不再接收新的输入图片。
14. 调用acl.himpi.sys_ctl_epoll函数从epoll实例中删除编码通道的文件句柄。
15. 当用户完成所有编码之后，需要调用acl.himpi.venc_destroy_chn释放编码通道以及内部内存资源。
16. 调用acl.himpi.sys_close_epoll函数销毁DVPP epoll实例。
17. 调用acl.himpi.sys_exit接口进行媒体数据处理系统去初始化。

优化视频编码质量

在实现VENC视频编码功能时，可在创建通道时设置基本参数、或调用对应的set接口设置高级参数，优化视频编码质量，以下调整手段可以叠加使用，效果是叠加的，例如：

- H264视频数据获取场景，分辨率720P，gop = 60，帧率30fps，码率1M需要提升编码质量，可以使用如下优化手段组合：CBR模式、HI_VENC_SCENE_0、stats_time等于2、profile等于2、关闭宏块级码控。
- H265电影场景，分辨率1080P，gop=30，帧率25fps，码率2M需要提升编码质量，可以使用如下优化手段组合：CBR模式、HI_VENC_SCENE_1、stats_time等于1、关闭宏块级码控。

当前支持以下方式优化视频编码质量：

- **设置基本参数，优化视频编码质量**

不同分辨率的视频，其编码质量与视频的帧率、GOP（Group of pictures）、码率有关，在调用acl.himpi.venc_create_chn接口创建通道时，可设置编码的等级、设置H.264/H.265协议编码场景下CBR/VBR/AVBR/CVBR/QVBR模式的帧率、GOP、码率等参数，来调整视频编码质量：

- 编码等级，通过通过hi_venc_chn_attr["venc_attr"]字典内的“profile”属性来确定。
- 帧率，通过hi_venc_chn_attr["rc_attr"]字典内的“src_frame_rate”输入帧率参数、“dst_frame_rate”输出帧率属性来确定。
- GOP，通过hi_venc_chn_attr["rc_attr"]字典内的“gop”属性来确定。
- 码率，通过hi_venc_chn_attr["rc_attr"]字典内的“bit_rate”、“max_bit_rate”或“target_bit_rate”属性来确定。

表 5-2 典型场景下帧率、GOP、码率的取值

画质/分辨率	帧率	GOP	码率（mbps）
4K 3840*2160/4096*2160	25或30	50或60	• 视频数据获取场景 H264/H265码流，码率取值8~12。 • 秀场/主播/短视频场景 H265码流，码率取值6~12。 H264码流，不涉及。 • 游戏视频场景 H264/H265码流，码率取值10~16。
2K 2560*1440	25或30	50或60	• 视频数据获取场景 H264/H265码流，码率取值6~10。 • 秀场/主播/短视频场景 H265码流，码率取值4.8~8。 H264码流，不涉及。 • 游戏视频场景 H264/H265码流，码率取值6~10。
1080P（蓝光） 1920*1080	25或30	50或60	• 视频数据获取场景 H265码流，码率取值1~4。 H264码流，码率取值2~6。 • 秀场/主播/短视频场景 H265码流，码率取值1.4~3.6。 H264码流，码率取值2~4.8。 • 游戏视频场景 H264/H265码流，码率取值3~6。

画质/分辨率	帧率	GOP	码率（mbps）
720P（高清） 1280*720	25或30	50或60	• 视频数据获取场景 H265码流，码率取值0.8~2。 H264码流，码率取值1~3。 • 秀场/主播/短视频场景 H265码流，码率取值1~2。 H264码流，码率取值1~3。 • 游戏视频场景 H264/H265码流，码率取值2~4。
480P/D1_N（标清） 854*480/720*480	25或30	50或60	• 视频数据获取场景 H265码流，码率取值0.3~0.7。 H264码流，码率取值0.6~1.4。 • 秀场/主播/短视频场景 H265码流，码率取值0.25~0.6。 H264码流，码率取值0.3~0.7。 • 游戏视频场景 不涉及。
576P/D1（标清） 720*576	25或30	50或60	• 视频数据获取场景 H265码流，码率取值0.3~0.7。 H264码流，码率取值0.6~1.4。 • 秀场/主播/短视频场景 H265码流，码率取值0.25~0.6。 H264码流，码率取值0.3~0.7。 • 游戏视频场景 不涉及。
270P（流畅） 480*270	25或30	50或60	• 视频数据获取场景 不涉及。 • 秀场/主播/短视频场景 H265码流，码率取值0.2。 H264码流，码率取值0.3。 • 游戏视频场景 不涉及。
CIF P/N 352*288/320*240	25或30	50或60	• 视频数据获取场景 H264/H265码流，码率取值0.25。 • 秀场/主播/短视频场景 不涉及。 • 游戏视频场景 不涉及。

- **设置高级参数，调整视频编码细节**

您可以调用接口设置码控模式、宏块级码率控制参数、编码场景模式等，来调整视频编码的细节，进一步改善编码质量。

表 5-3 高级配置项列表

配置项	接口	参数名	说明
码控模式	acl.himpi.venc_create_chn	hi_venc_chn_attr["rc_attr"]字典内的“rc_mode”属性	追求码率平稳或追求PSNR大且码率符合目标值，配置为CBR。 追求节省码率，对主观编码质量有一定要求，配置为VBR。 追求节省码率，对主观编码质量有一定要求，且场景中有较多静止画面，配置为AVBR。 追求PSNR且对码率上浮没有严格要求，配置为QVBR。 追求节省码率，对主观编码质量有一定要求，且可以根据带宽、存储空间要求进行更多调整，配置为CVBR。
码率控制模型统计时间	acl.himpi.venc_create_chn	hi_venc_chn_attr["rc_attr"]字典内各模式属性值字典内的“stats_time”属性	关注长期码率稳定，短期波动不在意的可以设置大一些，例：DVR存盘。设大可以提高重编码判决的门槛，重编码次数会减少，但是码率波动会加大。
宏块级码率控制参数	acl.himpi.venc_set_rc_param	hi_venc_rc_param字典内的“threshold_i”、“threshold_p”、“threshold_b”、“direction”、“row_qp_delta”属性。	如果图像内容复杂、细节较多或用户关注PSNR等客观指标时，需关闭宏块级码率控制。

配置项	接口	参数名	说明
第一帧的起始Qp值	acl.himpi.venc_create_chn	hi_venc_rc_param字典内的“first_frame_start_qp”属性	典型场景下，用户配置的码率小于表5-2中给的参考值，且编码后的视频第一帧明显模糊，则建议配置“first_frame_start_qp”属性，参数值取[min_i_qp, max_i_qp]的中间值，例如，[min_i_qp, max_i_qp]为[30, 40]，则“first_frame_start_qp”配置为“35”，同时将“max_reencode_times”配置为“0”，会获得较好的编码质量。
编码场景模式	acl.himpi.venc_set_scene_mode	hi_venc_scene_mode类	安防场景配置为HI_VENC_SCENE_0；自动驾驶、直播、游戏、动画、电影配置为HI_VENC_SCENE_1。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）。
run_mode, ret = acl.rt.get_run_mode()

# 2.pyACL 初始化。
ret = acl.init()

# 3.运行管理资源申请(依次申请Device、Context)。
ret = acl.rt.set_device(0)
context, ret = acl.rt.create_context(0)

# 4.初始化媒体数据处理系统。
ret = acl.himpi.sys_init()

# 5.设置VENC模块参数。
param = {'mod_type':HI_VENC_MOD_H265}
param, ret = acl.himpi.venc_get_mod_param(param)
param['jpeg_mod_param']['one_stream_buf'] = 1
ret = acl.himpi.venc_set_mod_param(param)

# 6.创建通道。
channel_id = 0
venc_attr = {'type': HI_VENC_MOD_H265, 'profile': 0,
             'max_pic_width': 128, 'pic_width': 128,
             'max_pic_height': 128, 'pic_height': 128,
             'buf_size': 1024 * 1024 * 2, 'is_by_frame': 1}
rc_attr = {'rc_mode':HI_VENC_RC_MODE_H265_VBR,
           'h265_vbr':{'gop': 30, 'stats_time': 1,
                       'src_frame_rate': 30, 'dst_frame_rate': 30,
                       'max_bit_rate': 4000}}
gop_attr = {'gop_mode':0, 'normal_p':{'ip_qp_delta':3}}
attr = {'venc_attr':venc_attr, 'rc_attr':rc_attr, 'gop_attr':gop_attr}
ret = acl.himpi.venc_create_chn(channel_id, attr)

# 7.通知编码器开始接收输入数据。
recv_param = {'recv_pic_num':-1}
ret = acl.himpi.venc_start_chn(channel_id, recv_param)
```



```
# 8.发送输入数据。
# 8.1 申请输入内存。
input_size = 128 * 128 * 3 // 2
input_addr, ret = acl.himpi.dvpp_malloc(0, input_size);

# 如果运行模式为ACL_HOST，则需要申请Host内存，将输入数据读入Host内存，再通过acl.rt.memcpy接口将
Host的数据传输到Device，数据传输完成后，需及时释放Host内存；否则直接将输入数据读入Device内存。
# 直接将输入数据读入Device内存。
if run_mode == ACL_HOST:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_filee_path, dtype=np.byte)
    jpege_file_size = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_HOST_TO_DEVICE)
else:
    # 将输入图片读入内存中。
    jpege_file = np.fromfile(jpege_file_path, dtype=np.byte)
    jpege_file = jpege_file.itemsize * jpege_file.size

    bytes_data = jpege_file.tobytes()
    jpege_file_ptr = acl.util.bytes_to_ptr(bytes_data)
    # 数据传输。
    ret = acl.rt.memcpy(input_addr, input_size, jpege_file_ptr, jpege_file_size,
ACL_MEMCPY_DEVICE_TO_DEVICE)

# 8.2 发送输入数据，开始编码。
v_frame = {'width': 128,
           'height': 128,
           'field': HI_VIDEO_FIELD_FRAME,
           'pixel_format': HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420,
           'video_format': HI_VIDEO_FORMAT_LINEAR,
           'compress_mode': HI_COMPRESS_MODE_NONE,
           'dynamic_range': HI_DYNAMIC_RANGE_SDR8,
           'color_gamut': HI_COLOR_GAMUT_BT709,
           'header_stride': [0, 0, 0],
           'width_stride': [128, 0, 0],
           'height_stride': [0, 0, 0],
           'header_phys_addr': [0, 0, 0],
           'phys_addr': [0, 0, 0],
           'header_virt_addr': [0, 0, 0],
           'virt_addr': [input_addr, 0, 0],
           'time_ref': 0,'pts': cur_time}
frame = {'v_frame':v_frame, 'pool_id':0, 'mod_id':HI_ID_VENC}
ret = acl.himpi.venc_send_frame(channel_id, frame, 0)

# 9.获取编码结果。
# 9.1 通过EPOLL处理编码完成事件。
fd = acl.himpi.venc_get_fd(channel_id)
epoll_fd, ret = acl.himpi.sys_create_epoll(10)

event['data'] = fd
event['events'] = HI_DVPP_EPOLL_IN
ret = acl.himpi.sys_ctl_epoll(epoll_fd, HI_DVPP_EPOLL_CTL_ADD, fd, event)

# 编码完成前，会超时阻塞在这里，一旦完成，才会往下执行。
events, eventCount, ret = acl.himpi.sys_wait_epoll(epoll_fd, 3, 1000);

# 9.2 获取编码结果。
status, ret = acl.himpi.venc_query_status(channel_id)
stream = {'pack_cnt': status['cur_packs']}
stream, ret = acl.himpi.venc_get_stream(self.channel_id, stream, 1000)
# 9.3 如果运行模式为ACL_HOST，且Host上需要使用编码输出的码流，则需要申请Host内存，通过
acl.rt.memcpy接口将Device的输出码流传输到Host。
# 9.3 获取编码输出码流数据。
if run_mode == ACL_HOST:
```



```
    # 申请Host内存。
    output_buffer, ret= acl.rt.malloc_host(output_ize)
    # 数据传输。
    ret = acl.rt.memcpy(output_buffer, output_ize, stream['pack'][0]['addr'], output_ize,
ACL_MEMCPY_DEVICE_TO_HOST)
    # ......
    # 数据使用完成后，及时释放不使用的内存。
    ret = acl.rt.free_host(output_buffer)
else:
    # 可以直接使用编码输出码流数据，在stream['pack'][0]['addr']指向的内存中。
    # ......

# 10.释放输入内存和输出码流。
ret = acl.himpi.dvpp_free(input_addr)
ret = acl.himpi.venc_release_stream(channel_id, stream)

# 11.通知编码器停止接收输入数据。
ret = acl.himpi.venc_stop_chn(channel_id)
ret = acl.himpi.sys_ctl_epoll(epoll_fd, HI_DVPP_EPOLL_CTL_DEL, fd, event)
ret = acl.himpi.sys_close_epoll(epoll_fd)

# 12.销毁通道。
ret = acl.himpi.venc_destroy_chn(channel_id)

# 13. 媒体数据处理系统去初始化。
ret = acl.himpi.sys_exit()

# 14. 释放运行管理资源(依次释放Context、Device)。
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(0)

# 15.pyACL去初始化。
ret = acl.finalize()
```


6 单算子调用

6.1 简介

单算子调用的使用场景

如果AI应用中不仅仅包括模型推理，还有数学运算（例如BLAS基础线性代数运算）、数据类型转换等功能，也想使用昇腾的算力，昇腾CANN还能支持吗?

答案是肯定的，昇腾CANN提供了单算子调用的方式，直接通过pyACL接口加载并执行单个算子，省去模型构建、训练的过程，相对轻量级，又可以使用昇腾的算力。

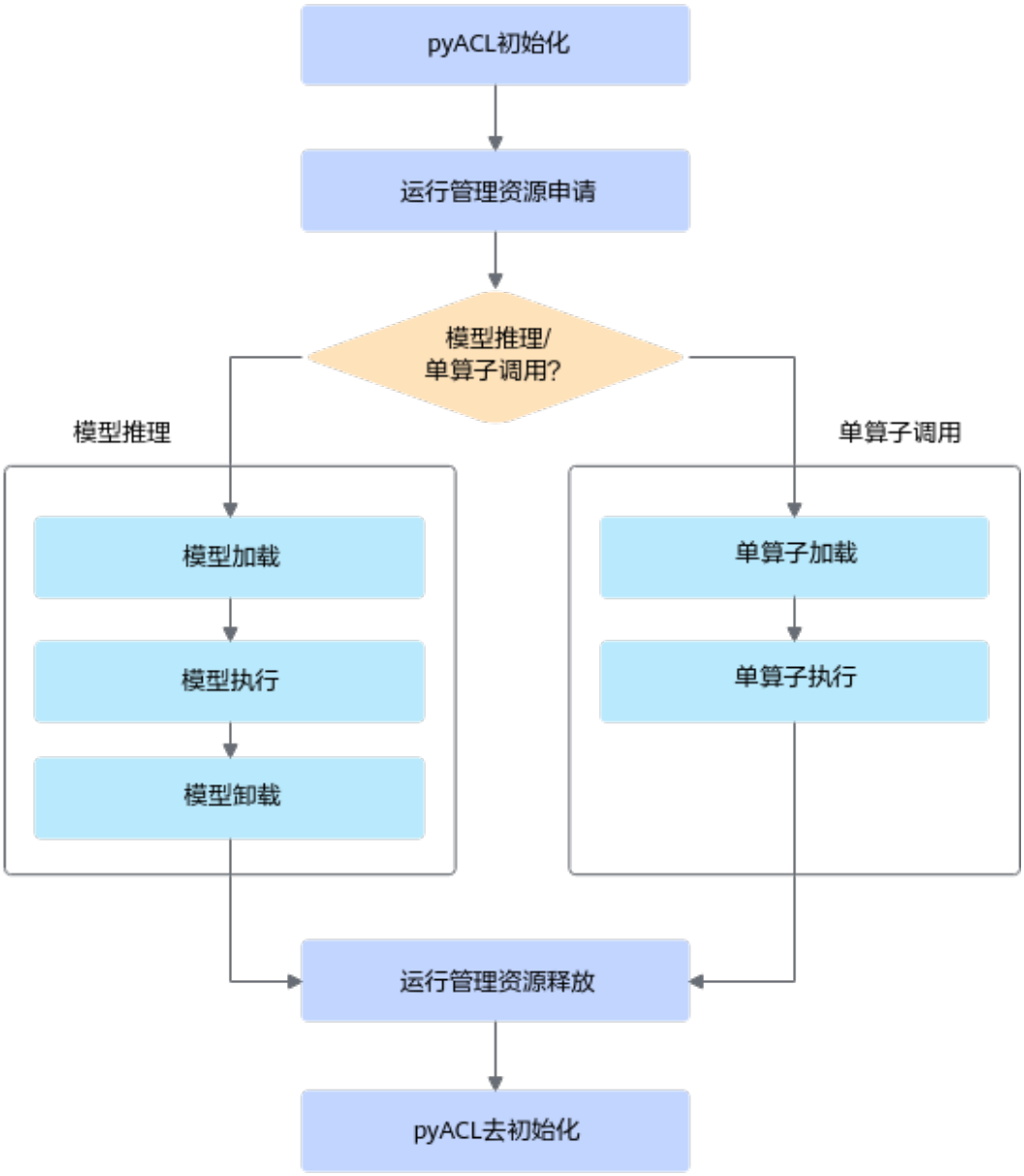
另外，自定义的算子，也可以通过单算子调用的方式来验证算子的功能。

单算子调用与模型推理的差别

在解释单算子调用与模型推理的差别前，我们先观察下面这个开发流程图，找出基本的共同点、不同点。

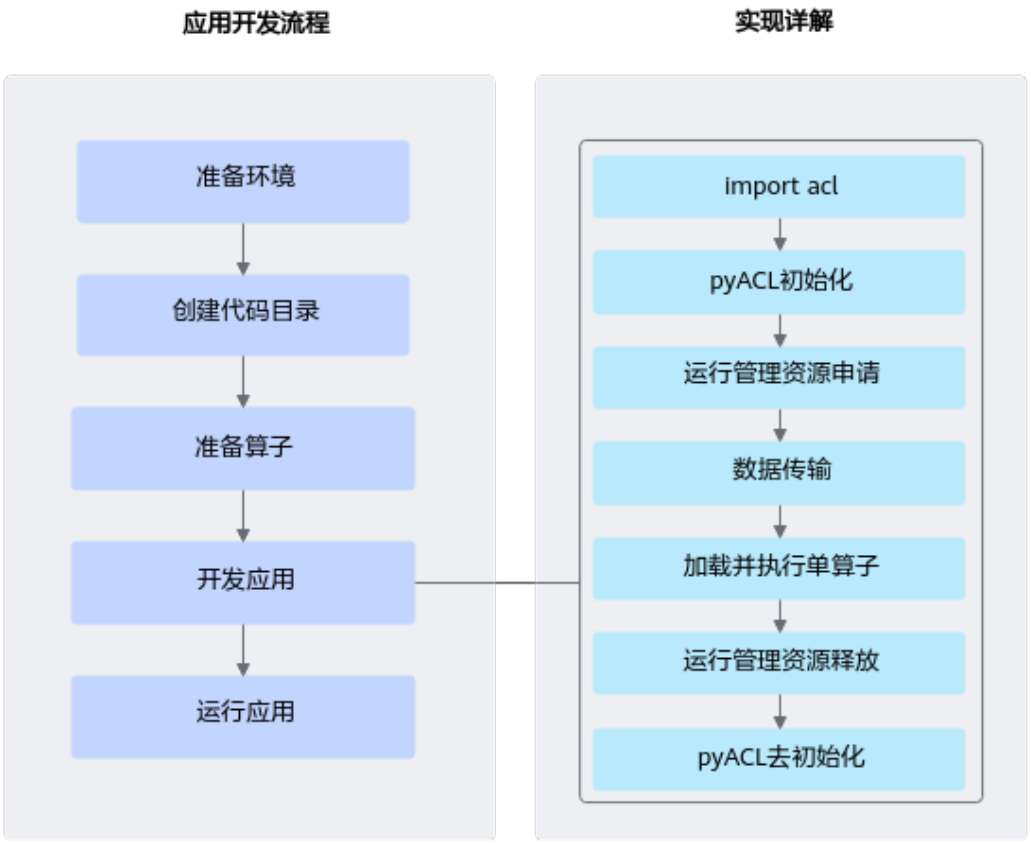
- 共同点：
  - 需要pyACL初始化和去初始化、运行管理资源申请和释放。
  - 涉及加载、执行的步骤，但是要注意，两者的加载、执行是调用不同的pyACL接口。
- 不同点：
  - 模型推理涉及模型卸载的步骤，单算子调用不涉及。

图 6-1 单算子调用与模型推理的流程对比



6.2 开发流程

图 6-2 开发流程



1. 准备环境。

   请参见**3.4 应用开发环境准备**。

2. 创建代码目录。

   在开发应用前，您需要先创建目录，存放代码文件、脚本、测试图片数据、模型文件等。

3. 准备算子。

   使用ATC工具编译算子生成om模型文件。

   该种方式，需要先构造*.json格式单算子描述文件（描述算子的输入、输出及属性等信息），借助ATC工具，将单算子描述文件编译成om模型文件，再分别调用pyACL接口加载om模型文件、执行算子。

   关于ATC工具的使用说明，请参见《ATC工具使用指南》。

4. 开发应用。

   单算子调用的流程请参见**6.3 接口调用流程**及相关的示例代码。

5. 运行应用，请参见**8 应用调试**。

6.3 接口调用流程

开发应用时，如果涉及执行单个算子，则应用程序中必须包含执行单个算子的代码逻辑。关于执行单个算子的接口调用流程，请先参见**3.3 pyACL接口调用流程**了解整体流程，再查看本节中的流程说明。

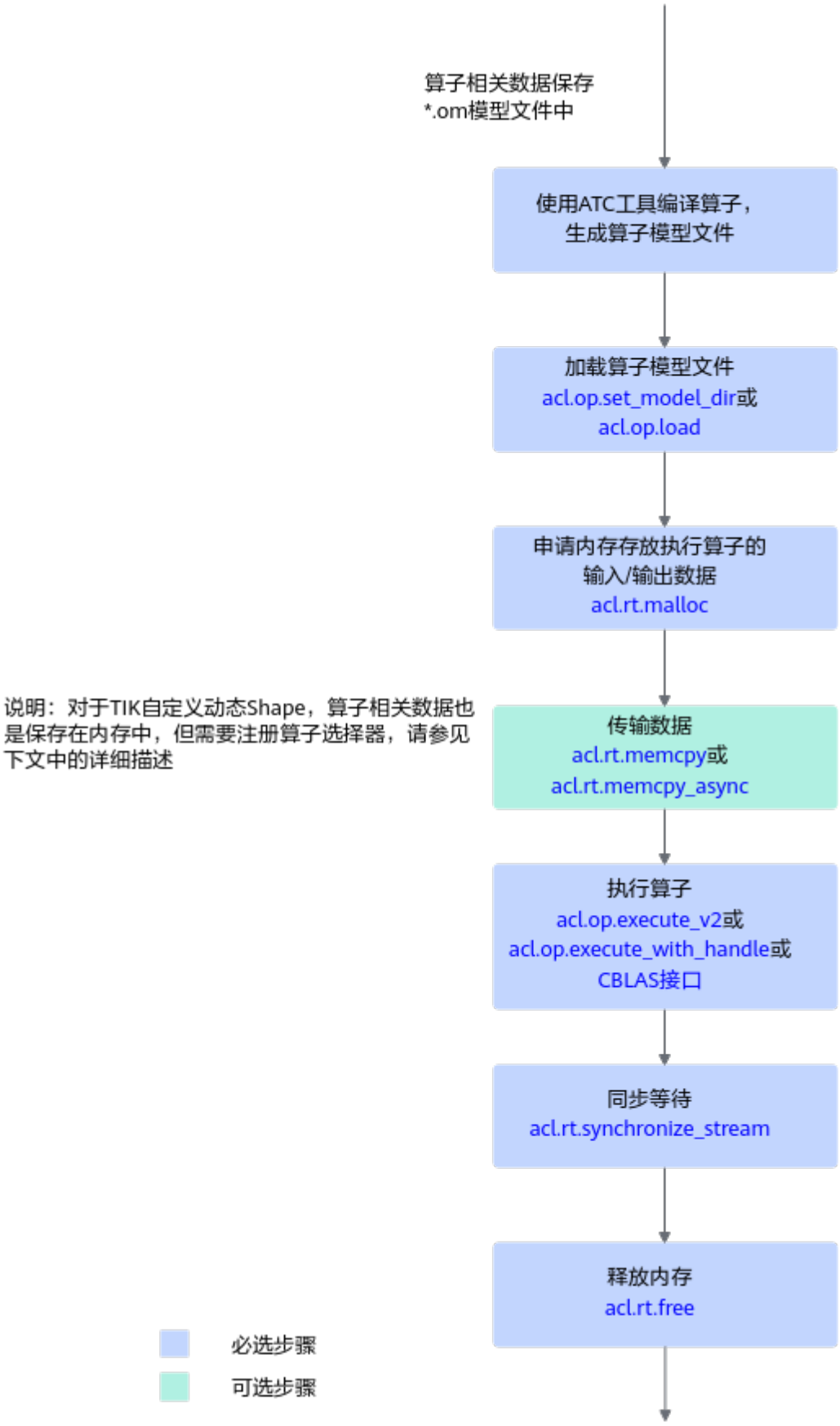
系统支持的算子请参见《 算子清单 》。

对于系统不支持的算子，用户需先参见《 TBE&AI CPU自定义算子开发指南 》完成自定义算子开发。

说明

对于TIK自定义动态Shape算子，需要先注册算子选择器，请参见**6.7 动态Shape算子（注册算子选择器）**。

图 6-3 算子调用流程



关键接口的说明如下：

1. 加载算子模型文件。

   支持以下2种方式中的一种加载单算子模型文件：

   - 调用acl.op.set_model_dir接口，设置加载模型文件的目录，目录下存放单算子模型文件（*.om文件）。
   - 调用acl.op.load接口，从内存中加载单算子模型数据，由用户管理内存。单算子模型数据是指“单算子编译成*.om文件后，再将om文件读取到内存中”的数据。

2. 调用acl.rt.malloc接口申请Device上的内存，存放执行算子的输入、输出数据。

   如果需要将Host上数据传输到Device，则需要调用acl.rt.memcpy接口（同步接口）或acl.rt.memcpy_async接口（异步接口）通过内存复制的方式实现数据传输。

3. 动态Shape场景，如果无法明确算子的输出Shape时，在执行算子前，还需推导或预估算子的输出Shape。

   需用户调用acl.op.infer_shape接口、acl.get_tensor_desc_num_dims接口、acl.get_tensor_desc_dim_v2接口、acl.get_tensor_desc_dim_range等接口，推导或预估算子的输出Shape，作为算子执行接口acl.op.execute_v2的输入。

4. 执行算子。

   - 对于被封装成pyACL接口的算子（参见**6.4 调用CBLAS接口**），包括GEMM算子、Cast算子，目前支持以下两种执行方式：

     - 不以handle方式执行算子，接口名称中不包含“Handle”关键字，例如，调用acl.blas.gemm_ex接口（封装GEMM算子）、acl.op.cast接口（封装Cast算子）等执行算子。
     - 以handle方式执行算子，接口名称中包含“Handle”关键字，例如，调用acl.blas.create_handle_for_gemm_ex接口、acl.op.create_handle_for_cast接口等创建handle后，还需要调用acl.op.execute_with_handle接口执行算子。

   - 对于未被封装成pyACL接口的算子，目前执行以下两种执行方式：

     - 不以handle方式执行算子，调用acl.op.execute_v2接口执行算子。
     - 以handle方式执行算子，调用acl.op.create_handle接口创建handle，再调用acl.op.execute_with_handle接口执行算子。

   说明

   不以handle方式执行算子时，每次执行算子时，系统内部都会根据算子描述信息匹配内存中的模型。

   以handle方式执行算子时，系统内部将算子描述信息匹配到内存中的模型，并缓存在Handle中，每次执行算子时，无需重复匹配算子与模型，因此在涉及多次执行同一个算子时，效率更高。但Handle使用结束后，需调用acl.op.destroy_handle接口释放。

5. 调用acl.rt.synchronize_stream接口阻塞应用运行，直到指定Stream中的所有任务都完成。

6. 调用acl.rt.free接口释放内存。

   如果需要将Device上的算子执行结果数据传输到Host，则需要调用acl.rt.memcpy接口（同步接口）或acl.rt.memcpy_async接口（异步接口）通过内存复制的方式实现数据传输，然后再释放内存。

6.4 调用 CBLAS 接口

基本原理

目前，pyACL已将GEMM算子（用于矩阵-向量乘、矩阵-矩阵乘）、Cast算子（用于转换数据类型）封装成pyACL接口，可参见CBLAS接口，目前支持以下两种执行方式：

- 不以handle方式执行算子，接口名称中不包含“handle”关键字，例如，调用acl.blas.gemm_ex接口（封装GEMM算子）、acl.op.cast接口（封装Cast算子）等执行算子。
- 以handle方式执行算子，接口名称中包含“handle”关键字，例如，调用acl.blas.create_handle_for_gemm_ex接口、acl.op.create_handle_for_cast接口等创建handle后，还需要调用acl.op.execute_with_handle接口执行算子。

说明

不以handle方式执行算子时，每次执行算子时，系统内部都会根据算子描述信息匹配内存中的模型。

以handle方式执行算子时，系统内部将算子描述信息匹配到内存中的模型，并缓存在Handle中，每次执行算子时，无需重复匹配算子与模型，因此在涉及多次执行同一个算子时，效率更高。但Handle使用结束后，需调用acl.op.destroy_handle接口释放。

示例代码

本章以acl.blas.gemm_ex接口为例，该示例中矩阵乘的计算公式为：C = αAB + βC。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

ACL_MEM_MALLOC_NORMAL_ONLY = 2
ACL_TRANS_N = 0
ACL_COMPUTE_HIGH_PRECISION = 0
ACL_MEMCPY_HOST_TO_DEVICE = 1
ACL_MEMCPY_DEVICE_TO_HOST = 2

# 1.pyACL初始化。
ret = acl.init("test_data/config/acl.json")

# 2.设置单算子模型文件所在的目录。
ret = acl.op.set_model_dir("op_models")

# 3.指定用于运算的设备。
device_id = 0
ret = acl.rt.set_device(device_id)

# 4.申请内存。
# 4.1 申请Device上的内存存放执行算子的输入数据。
# 对于该矩阵乘示例，size_a表示矩阵A数据的大小，size_b表示矩阵B数据的大小，size_c表示矩阵C数据的大小。
in_dtype, out_dtype = 1, 1
size_a = m * k * acl.data_type_size(acl_dtype)
size_b = m * k * acl.data_type_size(acl_dtype)
size_c = m * k * acl.data_type_size(acl_dtype)
dev_matrix_a, ret = acl.rt.malloc(size_a, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_matrix_b, ret = acl.rt.malloc(size_b, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_matrix_c, ret = acl.rt.malloc(size_c, ACL_MEM_MALLOC_NORMAL_ONLY)
# 4.2 申请Host上的内存。
```



```
# 对于该矩阵乘示例，m表示矩阵A的行数与矩阵C的行数，n表示矩阵B的列数与矩阵C的列数。
# k表示矩阵A的列数与矩阵B的行数。
host_matrix_a, ret = acl.rt.malloc_host(size_a)
host_matrix_b, ret = acl.rt.malloc_host(size_b)
host_matrix_c, ret = acl.rt.malloc_host(size_c)

# 5.准备输入数据。
# 从文件读入到host_matrix_a和host_matrix_b中。
# 对于该矩阵乘示例，将矩阵A和矩阵B的数据从Host复制到Device。
ret = acl.rt.memcpy(dev_matrix_a, size_a, host_matrix_a, size_a, ACL_MEMCPY_HOST_TO_DEVICE)
ret = acl.rt.memcpy(dev_matrix_b, size_b, host_matrix_b, size_b, ACL_MEMCPY_HOST_TO_DEVICE)

# 6.执行单算子。
stream, ret = acl.rt.create_stream()
# 对于该示例，调用acl.blas.gemm_ex接口（异步接口）实现矩阵-矩阵的乘法。
ret = acl.blas.gemm_ex(ACL_TRANS_N, ACL_TRANS_N, ACL_TRANS_N, m, n, k, dev_alpha, dev_matrix_a, k,
input_type, dev_matrix_b, n, input_type, dev_beta, dev_matrix_c, n, output_type,
ACL_COMPUTE_HIGH_PRECISION, stream)
# 调用acl.rt.synchronize_stream接口阻塞Host运行，直到指定Stream中的所有任务都完成。
ret = acl.rt.synchronize_stream(stream)

# 7.将算子的输出数据从Device复制到Host。
ret = acl.rt.memcpy(host_matrix_c, size_c, dev_matrix_c, size_c, ACL_MEMCPY_DEVICE_TO_HOST)

# 8.释放运行管理资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.reset_device(device_id)
ret = acl.finalize()
# ......
```

6.5 固定 Shape 算子

对于不支持的算子，用户需参考《TBE&AI CPU自定义算子开发指南》先完成自定义算子的开发，再参考如下内容执行单算子。

示例代码请参见**示例代码（处理推理结果：调用单算子处理推理结果）**。

6.6 动态 Shape 算子（不注册算子选择器）

基本原理

对于支持动态Shape的算子：

- 如果算子输出Shape明确，该类算子执行的基本流程与固定Shape算子执行类似，接口调用流程请参见**6.3 接口调用流程**，执行固定Shape算子的示例代码请参见**6.5 固定Shape算子**。
- 如果无法明确算子的输出Shape，在调用acl.op.execute_v2接口前，需用户调用acl.op.infer_shape接口、acl.get_tensor_desc_num_dims接口、acl.get_tensor_desc_dim_v2接口、acl.get_tensor_desc_dim_range等接口，推导或预估算子的输出Shape，作为算子执行接口acl.op.execute_v2的输入。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
```



```
# 此处算子输入Tensor数据的内存必须根据应用运行模式来确定，应用运行在Host时，此处需申请Host上的内存。
# 应用运行在Device时，此处需申请Device上的内存。
ret = acl.op.infer_shape(op_type, self.in_desc_list, self.in_list, num_outputs, self.out_desc_list, self.attr)

tensor_dims = []
# 循环算子的每一个输出，推导或预估Shape值：
for i in range(len(infer_desc_list)):
    dim_nums = acl.get_tensor_desc_num_dims(infer_desc_list[i])
    dim_size = []
    for j in range(dim_nums):
        dim, ret = acl.get_tensor_desc_dim_v2(infer_desc_list[i], j)
        # 表示动态Shape场景下维度值是动态的。
        if dim == -1:
            # 获取Shape范围，使用该范围的Shape最大值来构造输出tensor_desc，作为acl.op.execute_v2的输入。
            dim_range, ret = acl.get_tensor_desc_dim_range(infer_desc_list[i], j, 2)
            dim = dim_range[1]
        dim_size.append(dim)
    tensor_dims.append(dim_size)

# 以上给出了执行算子时输出的shape，根据tensor_dims中的dims构造输出tensor_desc（即self.out_desc_list参数值），用于调用acl.op.execute_v2。
ret = acl.op.execute_v2(op_type, self.in_desc_list, self.in_list, self.out_desc_list, self.out_list, self.attr, self.stream)

out_tensor_dims = []
# 针对上面用户预估Shape值以及使用Shape范围中的最大Shape的场景，设在算子执行结束后，需增加下面的调用，获取准确的shape。
for i in range(len(self.out_desc_list)):
    dim_nums = acl.get_tensor_desc_num_dims(self.out_desc_list[i])
    dim_size = []
    for j in range(dim_nums):
        dim, ret = acl.get_tensor_desc_dim_v2(self.out_desc_list[i], j)
        dim_size.append(dim)
    out_tensor_dims.append(dim_size)

# ......
```

6.7 动态 Shape 算子（注册算子选择器）

前提条件

在加载与执行动态Shape算子前，您需要参见《TBE&AI CPU自定义算子开发指南》中的“专题 > TIK自定义算子动态Shape专题”中的说明开发自定义算子以及生成对应的二进制文件。

基本原理

动态Shape场景下，算子加载与执行的流程如下：

1. 资源初始化，包括pyACL初始化、设置单算子模型文件的加载目录、指定用于运算的Device等。
   - 调用acl.init接口实现pyACL初始化。
   - 调用pyACL接口注册要编译的自定义算子：
     - 调用acl.op.register_compile_func接口注册算子选择器（即选择Tiling策略的函数），用于在算子执行时，能针对不同Shape，选择相应的Tiling策略。

       算子选择器需由用户提前定义并实现：
       - 函数原型：

```
op_selector(in_num, in_desc, out_num, out_desc, op_attr, op_kernel_desc)
    """
    算子选择器：函数和参数名称可自定义，参数个数和类型必须匹配。
    :param in_num: 输入Tensor描述数。
    :param in_desc: 输入Tensor描述list。
    :param out_num: 输出Tensor描述数。
    :param out_desc: 输出Tensor描述list。
    :param op_attr: 算子属性的地址对象，用于设置算子属性信息。
    :param op_kernel_desc:算子内核描述地址对象，用于动态Shape场景下，设置算子Workspace参数。
    :return:
    """
```

- 函数实现：

  用户自行编写代码逻辑实现Tiling策略选择、Tiling参数生成，并将调用acl.op.set_kernel_args接口，设置算子Tiling参数、执行并发数等。

- 调用acl.op.create_kernel接口将算子注册到系统内部，用于在算子执行时，查找到算子实现代码。

– 调用acl.rt.set_device接口指定运算的Device。

– 调用acl.rt.create_context接口显式创建一个Context，调用acl.rt.create_stream接口显式创建一个Stream。

  若没有显式创建Stream，则使用默认Stream，默认Stream是在调用acl.rt.set_device接口时隐式创建的，默认Stream作为接口入参时，直接传0。

2. 用户自行构造算子描述信息（输入输出Tensor描述、算子属性等）、申请存放算子输入输出数据的内存。
3. 将算子输入数据从Host复制到Device上。
   – 调用acl.rt.memcpy接口实现同步内存复制，内存使用结束后需及时释放。
   – 调用acl.rt.memcpy_async接口实现异步内存复制，内存使用结束后需及时释放。
4. 编译单算子。

   调用acl.op.update_params接口编译指定算子，触发算子选择器的调用逻辑。
5. 执行单算子。

   调用acl.op.execute_v2接口加载并执行算子。
6. 将算子运算的输出数据从Device上复制到Host上（提前申请Host上的内存）。
   – 调用acl.rt.memcpy接口实现同步内存复制，内存使用结束后需及时释放。
   – 调用acl.rt.memcpy_async接口实现异步内存复制，内存使用结束后需及时释放。
7. 按顺序先释放Stream资源，再释放Context资源，最后释放Device资源。
   – 调用acl.rt.destroy_stream接口释放Stream。

     不涉及显式创建Stream，使用默认Stream时，无需调用acl.rt.destroy_stream接口释放Stream。
   – 调用acl.rt.destroy_context接口释放Context。

     不涉及显式创建Context，使用默认Context时，无需调用acl.rt.destroy_context接口释放Context。
   – 调用acl.rt.reset_device接口释放Device。
8. 调用acl.finalize接口实现pyACL去初始化。

示例代码

在样例代码中，调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

动态Shape算子（注册算子选择器）样例的获取、编译运行请参见《TBE&AI CPU自定义算子开发指南》中的“专题 > TIK自定义算子动态Shape专题>样例使用”。

```
import acl
import numpy as np
# ......

ACL_ENGINE_AICORE = 1
ACL_FLOAT16 = 1
ACL_FORMAT_ND = 2
ACL_MEM_MALLOC_NORMAL_ONLY = 2
ACL_MEMCPY_HOST_TO_DEVICE = 1
ACL_MEMCPY_DEVICE_TO_HOST= 2

device_id = 0

# 1.资源初始化。
ret = acl.init()
ret = acl.rt.set_device(device_id)
stream, ret = acl.rt.create_stream()
ret = acl.op.register_compile_func("add", op_select)
# 将算子Kernel的*.o文件需要用户提前编译好，并调用numpy加载.o文件并转成地址对象，op_data_size_0表示第一个.o文件占用的内存大小。
# 如果有多个算子Kernel的*.o文件，需要多次调用该接口。
ret = acl.op.create_kernel("add", "cce_add_11_33_float16_11_33_float16__kernel0",
"cce_add_11_33_float16_11_33_float16__kernel0", np_op_0_ptr, op_data_size_0, ACL_ENGINE_AICORE, 0)

# 2.构造add算子的输入输出Tensor、输入输出Tensor描述，并申请存放算子输入数据、输出数据的内存。
# 输入参数。
a = np.random.rand(2, 1).astype(np.float16)
b = np.random.rand(2, 1).astype(np.float16)
bytes_data = a.tobytes()
a_ptr = acl.util.bytes_to_ptr(bytes_data)
bytes_data = b.tobytes()
b_ptr = acl.util.bytes_to_ptr(bytes_data)
input_desc_list = [acl.create_tensor_desc(ACL_FLOAT16, [2, 1], ACL_FORMAT_ND),
acl.create_tensor_desc(ACL_FLOAT16, [2, 1], ACL_FORMAT_ND)]
output_desc_list = [acl.create_tensor_desc(ACL_FLOAT16, [2, 1], ACL_FORMAT_ND)]
# 申请device侧内存。
size_a = acl.get_tensor_desc_size(input_desc_list[0])
size_b = acl.get_tensor_desc_size(input_desc_list[1])
size_c = acl.get_tensor_desc_size(output_desc_list[0])
dev_a, ret = acl.rt.malloc(size_a, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_b, ret = acl.rt.malloc(size_b, ACL_MEM_MALLOC_NORMAL_ONLY)
dev_c, ret = acl.rt.malloc(size_c, ACL_MEM_MALLOC_NORMAL_ONLY)

# 3.将算子输入数据从Host复制到Device上。
ret = acl.rt.memcpy(dev_a, size_a, a_ptr, size_a, ACL_MEMCPY_HOST_TO_DEVICE)
ret = acl.rt.memcpy(dev_b, size_b, b_ptr, size_b, ACL_MEMCPY_HOST_TO_DEVICE)

# 4.调用acl.op.update_params接口编译算子。
op_attr = acl.op.create_attr()
ret = acl.op.update_params("add", input_desc_list, output_desc_list, op_attr )

# 5.调用acl.op.execute接口加载并执行算子。
in_data_list = [acl.create_data_buffer(dev_a, size_a), acl.create_data_buffer(dev_b, size_b)]
out_data_list = [acl.create_data_buffer(dev_c, size_c)]
ret = acl.op.execute_v2("add", input_desc_list, in_data_list, output_desc_list, out_data_list, op_attr, stream)

# 6.将算子运算的输出数据从Device上复制到Host上（提前申请Host上的内存）。
host_ptr, ret = acl.rt.malloc_host(size_c)
ret = acl.rt.memcpy(host_ptr, size_c, dev_c, size_c, ACL_MEMCPY_DEVICE_TO_HOST)
bytes_out = acl.util.ptr_to_bytes(host_ptr, size_c)
out_np = np.frombuffer(bytes_out, dtype=np.byte).reshape((size_c,))
```



```
# 7.按顺序释放资源。
# 7.1 释放算子的输入输出Tensor描述。
# 7.2 释放Host上的内存。
# 7.3 释放Device上的内存。
# 7.4 释放设备管理资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.reset_device(device_id)
ret = acl.finalize()
# ......
```


7 扩展更多特性

7.1 内存二次分配管理

用户内存管理有两种管理方式：

- **独立**内存管理，根据需要**单独申请**所需的内存，内存**不做拆分或者二次分配**。
- **内存池**管理内存，用户**一次性申请**一块较大内存，并在使用时从这块较大内存中**二次分配**所需内存。

在内存二次分配时，请使用如下接口从内存池申请对应内存。由于各接口对申请的**内存地址、大小**有约束，在内存池管理时，需要对该情况关注处理，否则容易出现内存越界。

内存管理的总体说明请参见总体说明。

接口	用途	输入内存/输出内存
acl.rt.memcpy_async	实现内存复制，异步接口。	调用本接口进行内存复制时，源地址和目的地址都必须64字节对齐。
acl.rt.malloc	在Device上分配size大小的线性内存，并通过“dev_ptr”返回已分配内存的指针地址。本接口分配的内存会进行字节对齐，会对用户申请的size向上对齐成32字节整数倍后再多加32字节。	若用户使用本接口申请大块内存并自行划分、管理内存时，每段内存需同时满足以下需求： ● 内存大小向上对齐成32整数倍加32字节（m = ALIGN_UP[len, 32] + 32字节）。 ● 内存起始地址需满足64字节对齐（ALIGN_UP[m, 64]）。 **说明** **len**表示某段内存的大小，ALIGN_UP[len, k]表示向上按k字节对齐：((len - 1) / k + 1) * k。
acl.media.dvpp_malloc	该接口主要用于分配内存给Device侧媒体数据处理时使用，申请的大页内存满足数据处理的要求（例如，内存首地址128对齐）。 媒体数据处理各功能的详细介绍请参见媒体数据处理V1。	媒体数据处理的输出作为模型推理的输入时，若用户使用本接口申请大块内存并自行划分、管理内存时，每段内存需同时满足以下要求： ● 内存大小向上对齐成32整数倍+32字节（m=ALIGN_UP[len,32]+32字节）。 ● 内存起始地址需满足128字节对齐（ALIGN_UP[m,128]）。 **说明** len表示某段内存的大小，ALIGN_UP[len,k]表示向上按k字节对齐：((len-1)/k+1)*k。
acl.himpi.dvpp_malloc	申请Device上的内存，申请的内存满足媒体数据处理的要求（例如，内存首地址128字节对齐）。 媒体数据处理各功能的详细介绍请参见媒体数据处理V2。	媒体数据处理的输出作为模型推理的输入时，若用户使用本接口申请大块内存并自行划分、管理内存时，每段内存需同时满足以下要求： ● 内存大小向上对齐成32整数倍+32字节（m=ALIGN_UP[len,32]+32字节）； ● 内存起始地址需满足128字节对齐（ALIGN_UP[m,128]）。 **说明** len表示某段内存的大小，ALIGN_UP[len,k]表示向上按k字节对齐：((len-1)/k+1)*k。

接口	用途	输入内存/输出内存
acl.rt.malloc_host	AscendCL应用程序在Host上运行时，调用该接口申请的是Host内存（该内存是锁页内存），由系统保证内存首地址64字节对齐。 AscendCL应用程序在Device上运行时，调用该接口申请的是Device内存，且Device上的内存按普通页申请，如需首地址64字节对齐，需要用户自行处理对齐。	若用户使用本接口申请大块内存并自行划分、管理内存时，每段内存需同时满足以下需求： • 内存大小向上对齐成32整数倍加32字节（m = ALIGN_UP[len, 32] + 32字节）。 • 内存起始地址需满足64字节对齐（ALIGN_UP[m, 64]）。 **说明** **len**表示某段内存的大小，ALIGN_UP[len, k]表示向上按k字节对齐：((len - 1) / k + 1) * k。
acl.rt.malloc_cached	申请Device上的内存，该接口在任何场景下申请的内存都是支持cache缓存。在Device上申请size大小的线性内存，通过**dev_ptr**返回已分配内存的指针地址。	其它约束与acl.rt.malloc接口相同。

计算机视觉领域一般涉及使用媒体数据处理功能，因此会涉及以上多种内存申请接口。内存首地址涉及64字节或128字节对齐，为方便统一管理，内存首地址**对齐值建议选取较大项**，比如内存首地址128字节对齐。

关于媒体数据处理时自行管理内存时的典型场景如下，媒体数据处理的功能点介绍请参见**5.3 媒体数据处理V1**。

图 7-1 VDEC 场景

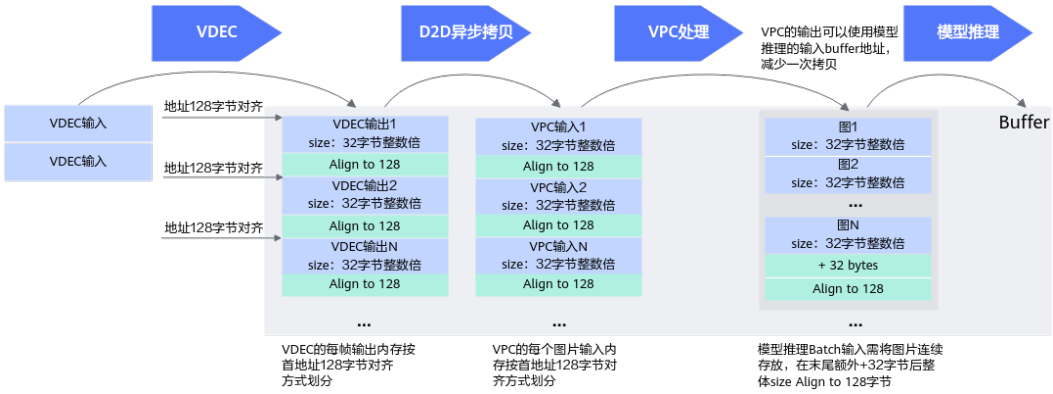
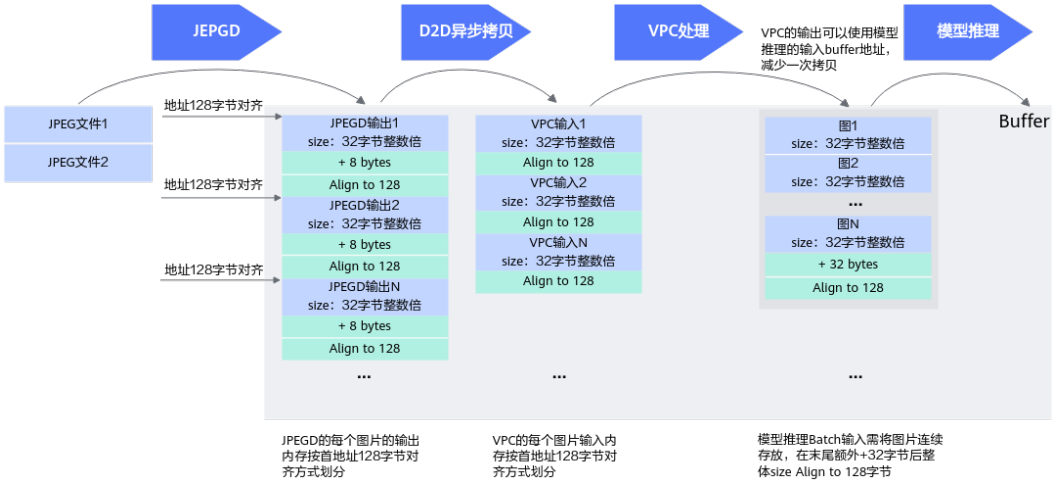


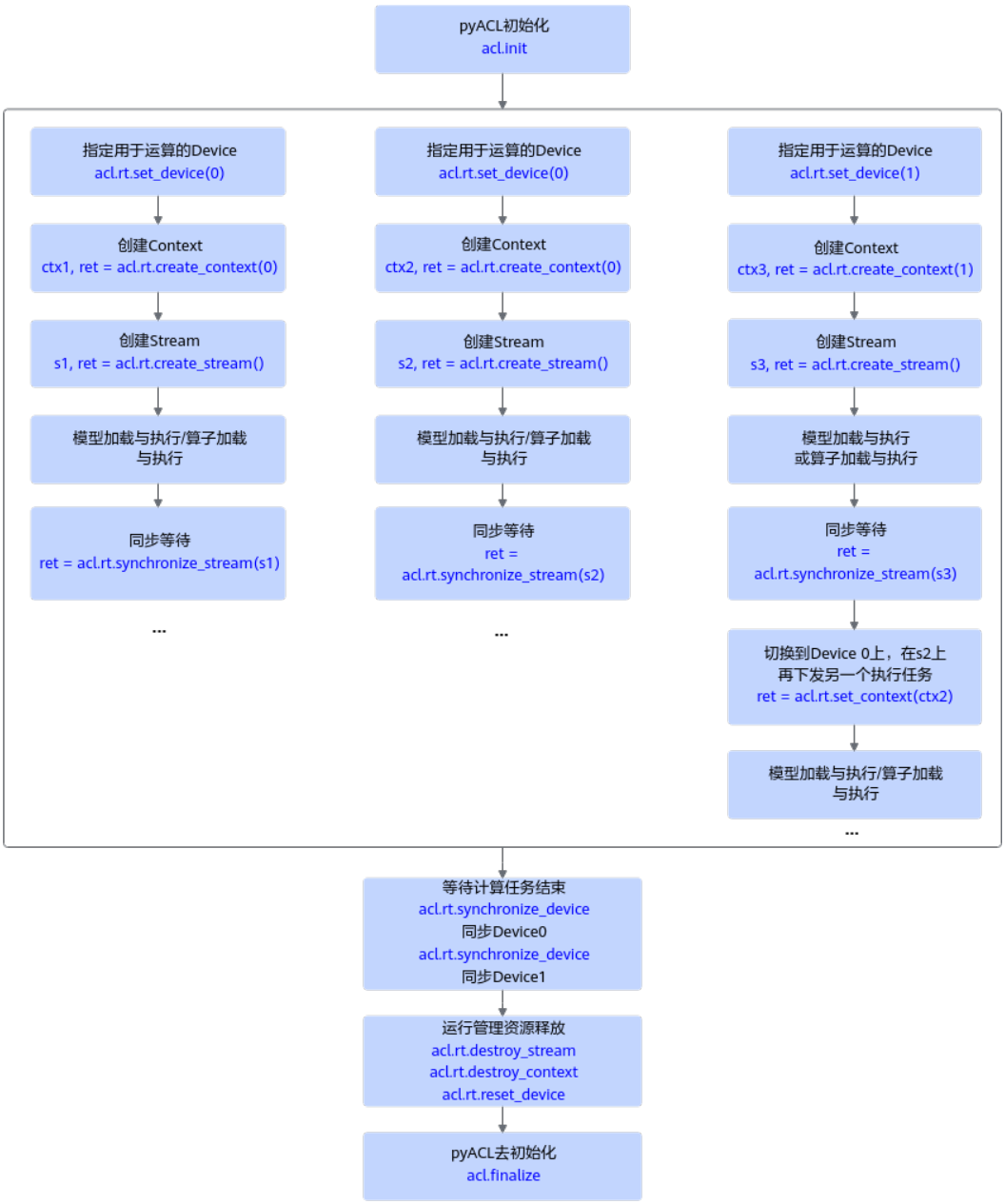
图 7-2 JPEGD 场景



7.2 多 Device 场景

开发应用时，如果涉及多Device之间的任务等待，则应用程序中必须包含相关的代码逻辑。关于该场景的接口调用流程，请依次参见**3.3 pyACL接口调用流程**以及本节中的说明。

图 7-3 同步等待流程_多 Device 场景



- 在多Device时，利用Context切换（调用acl.rt.set_context接口）来切换Device，比使用acl.rt.set_device接口效率高。
- 调用acl.rt.synchronize_device接口等待Device上的计算任务结束。
- 模型加载与执行的流程请参见**4 开发基础推理应用**。
- 算子加载与执行的流程请参见**6 单算子调用**。

7.3 多模型串联推理

多模型推理的基本流程与单模型类似，请参见**4.6 模型推理基本场景**。

多模型推理与单模型推理的不同点如下：

- 关于模型加载，如果涉及多个模型，需调用多次模型加载接口。模型加载请参见**4.6.1 模型加载**。
- 关于模型执行，如果涉及多个模型，需调用多次模型执行接口。模型执行请参见**4.6.2 模型执行**。

  例如，调用acl.mdl.execute接口实现同步模型推理。

7.4 多 Batch 模型推理

多Batch推理的基本流程与单Batch类似，请参见**4.6 模型推理基本场景**。

多Batch推理与单Batch推理的不同点在于：

- 多Batch场景下，在构建模型时，使用ATC工具的input_shape参数需设置具体的Batch数，详细说明请参见《ATC工具使用指南》。
- 在推理前，需要编写一段代码逻辑：等输入数据满足多Batch（例如：8Batch）的要求，申请Device上的内存存放多Batch的数据，作为模型推理的输入。如果最后循环遍历所有的输入数据后，仍不满足多Batch的要求，则直接将剩余数据作为模型推理的输入。

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考，此处以**8Batch**为例：

```
batch_size = 8
device_num = 1
device_id = 0

# 获取模型第一个输入的大小。
model_input_size = acl.mdl.get_input_size_by_index(model_desc, 0)
# 获取每个Batch输入数据的大小。
single_buff_size = model_input_size / batch_size

# 定义该变量，用于累加Batch数是否达到8Batch。
cnt = 0
# 定义该变量，用于描述每个文件读入内存时的位置偏移。
pos = 0
infer_file_vec = []

for i in len(files_list):
    # 每8个文件，申请一次Device上的内存，存放8Batch的输入数据。
    if cnt % batch_size == 0:
        pos = 0
        infer_file_vec.clear()
        # 申请Device上的内存。
        p_batch_dst, ret = acl.rt.malloc(model_input_size, ACL_MEM_MALLOC_NORMAL_ONLY)

    # TODO: 从某个目录下读入文件，计算文件大小file_size。

    # 根据文件大小，申请Host上的内存，存放文件数据。
    p_img_buf, ret = acl.rt.malloc_host(file_size)

    # 将Host上的文件数据复制到Device的内存中。
    ret = acl.rt.memcpy(p_batch_dst + pos, file_size, p_img_buf, file_size, ACL_MEMCPY_HOST_TO_DEVICE)
    pos += file_size
    # 释放Host上的内存。
    acl.rt.free_host(p_img_buf)
    # 将第i个文件存入vector中，同时cnt++。
    infer_file_vec.append(files_list[i])
    cnt++
    # 每8Batch的输入数据送给模型推理进行推理。
    if cnt % batch_size == 0:
        # TODO: 创建aclmdlDataset、aclDataBuffer类型的数据，用于描述模型的输入、输出数据。
        # TODO: 调用acl.mdl.execute接口执行模型推理。
```



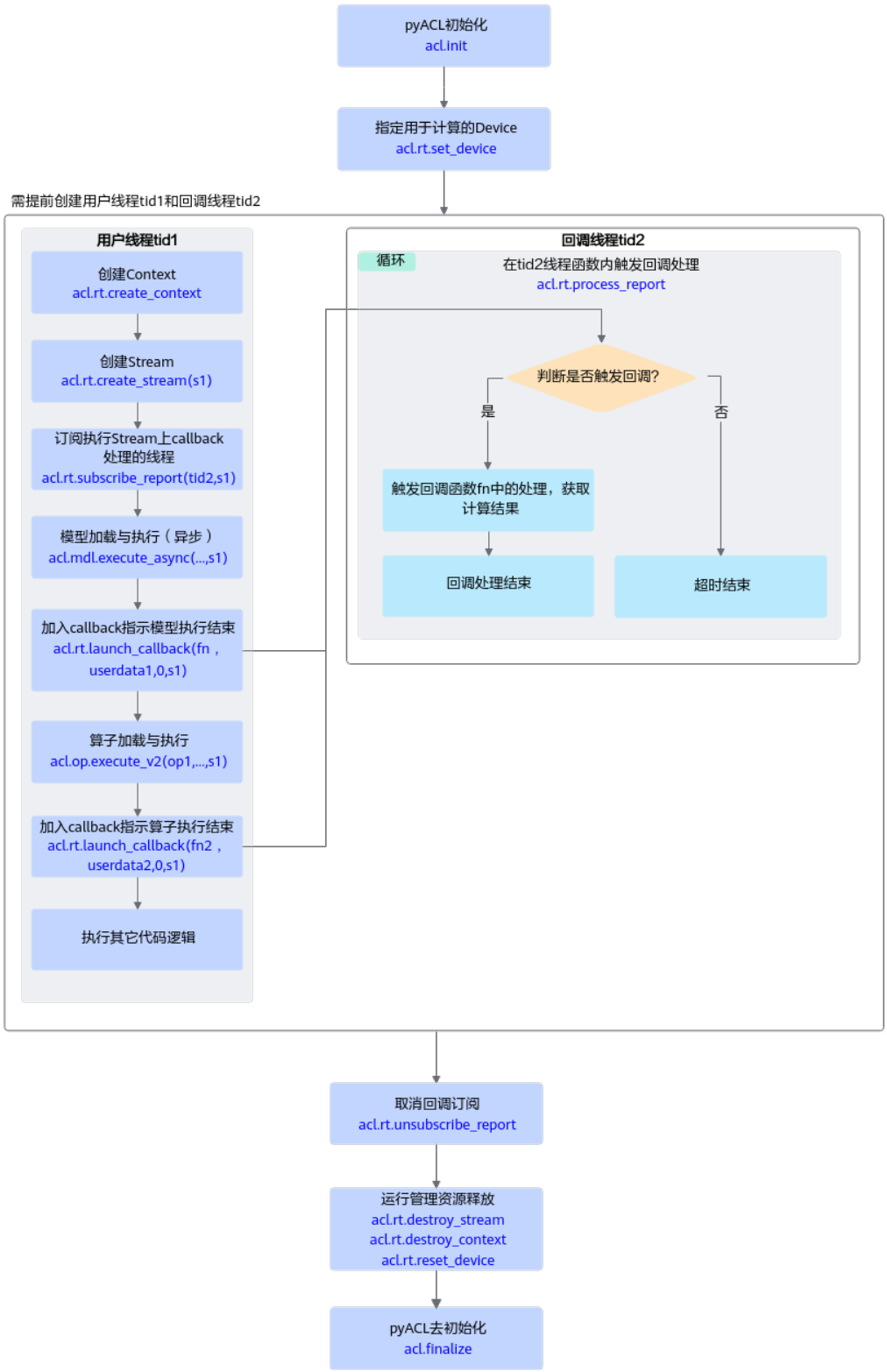
```
        # TODO: 推理结束后，调用acl.rt.free接口释放Device上的内存。

# 如果最后循环遍历所有的输入数据后，仍不满足多Batch的要求，则直接将剩余数据作为模型推理的输入。
if cnt % batch_size != 0:
    # TODO: 创建aclmdlDataset、aclDataBuffer类型的数据，用于描述模型的输入、输出数据。
    # TODO: 调用acl.mdl.execute接口执行模型推理。
    # TODO: 推理结束后，调用acl.rt.free接口释放Device上的内存。
# ......
```

7.5 模型异步推理

7.5.1 接口调用流程

开发应用时，如果涉及异步场景下的同步等待，则应用程序中必须包含相关的代码逻辑，关于该场景的接口调用流程，请参见下图。

图 7-4 同步等待流程_Callback 场景



关键接口说明如下：

1. 回调函数**需由用户提前创建**，用于获取并处理模型推理或算子执行的结果。

2. 线程需由用户提前创建，并自定义线程函数，在线程函数内调用acl.rt.process_report接口，设置超时时间，等待acl.rt.launch_callback接口下发的回调任务执行。
3. 调用acl.rt.subscribe_report接口：指定处理Stream上回调函数的线程，线程与**2**中创建的线程保持一致。
4. 异步推理时调用acl.mdl.execute_async接口。

   对于异步接口，还需调用acl.rt.synchronize_stream接口阻塞应用程序运行，直到指定Stream中的所有任务都完成。

   用户可以在acl.rt.synchronize_stream接口之后一次性获取所有图片的异步推理结果，但如果图片数据量较大的情况下，需要等待的时间比较长，这时可以使用Callback功能，每隔一段时间下发一次Callback任务，获取前一段时间内的异步推理结果。
5. 调用acl.rt.launch_callback接口：在Stream的任务队列中下发一个函数回调任务，系统内部在执行到该回调任务时，会在Stream上订阅的线程（通过acl.rt.subscribe_report接口订阅的线程）中执行回调函数，回调函数与**1**中保持一致。

   每调用一次acl.rt.launch_callback接口，就会下发一个回调函数任务。
6. 调用acl.rt.unsubscribe_report接口：取消线程订阅（Stream上的回调函数不再由指定线程处理）。

7.5.2 示例代码

您可以从**样例介绍**中查看完整样例代码。

在acl_resnet50_async样例中：

- 运行可执行文件，不带参数时：
  - 执行模型异步推理的次数默认为10次。
  - device_id默认使用编号为0的设备。
  - callback间隔默认为1，表示1次异步推理后，下发一次callback任务。
  - 内存池中的内存块的个数默认为10个。
- 运行可执行文件，带参数时：
  - 第一个参数表示使用设备ID。
  - 第二个参数表示执行模型异步推理的次数。
  - 第三个参数表示下发callback间隔，参数值为0时表示不下发callback任务，参数值为非0值（例如m）时表示m次异步推理后下发一次callback任务。
  - 第四个参数表示内存池中内存块的个数，内存块个数需大于等于模型异步推理的次数，用户可根据输入图片数量，来调整内存块的个数。
    - 例1：有2张输入图片、内存块个数为2时，则1张图片1个内存块。
    - 例2：有3张输入图片、内存块个数为10时，则执行10次循环，每（10/3取整）个内存块对应同一张图片，剩下1个内存块随机对应1张图片。
    - 内存块中存放是模型推理的输入数据、输出数据，若多个内存块对应的是同一张图片，则多个内存块中存放的是相同的输入数据、输出数据，用于输入图片少但又想模拟大量图片数据的场景。
  - 第五个参数表示要加载执行的模型的路径。

- 第六个参数表示输入的图片路径。支持图片格式如下：
  IMG_EXT = ['.jpg', '.JPG', '.png', '.PNG', '.bmp', '.BMP', '.jpeg', '.JPEG']

```
import acl
import argparse
# ......

images_list = [os.path.join(args.images_path, img) for img in os.listdir(args.images_path) \
               if os.path.splitext(img)[1] in IMG_EXT]
data_list = []
for image in images_list:
    # 自定义函数transfer_pic，完成以下功能：
    # 加载图片到内存，并resize成 224x224尺寸。
    transfer_pic(image)
    dst_im = np.fromfile(image.replace(".jpg", ".bin"), dtype=np.byte)
    data_list.append(dst_im)

# 1.资源初始化： 样例中通过Net类实现资源初始化操作。
# 1.1 pyACL初始化。
ret = acl.init()
# 1.2 运行管理资源申请。
ret = acl.rt.set_device(self.device_id)
self.context, ret = acl.rt.create_context(self.device_id)
self.stream, ret = acl.rt.create_stream()
# 获取当前昇腾AI软件栈的运行模式，根据不同的运行模式，后续的内存申请、内存复制等接口调用方式不同。
self.run_mode, ret = acl.rt.get_run_mode()
# 1.3 申请模型推理资源。
# 1.4 加载模型。
# 加载离线模型文件，模型加载成功，返回标识模型的ID。
self.model_id, ret = acl.mdl.load_from_file(self.model_path)
# 1.5 根据模型的ID，获取该模型的描述信息。
self.model_desc = acl.mdl.create_desc()
ret = acl.mdl.get_desc(self.model_desc, self.model_id)

# 2.模型推理。
# 2.1 根据输入参数内存池中内存块的个数申请内存，拷贝图片数据到device侧。
def _data_interaction(self, images_dataset_list):
    for idx in range(self.memory_pool):
        img_idx = idx % len(images_dataset_list)
        img_input = self._load_input_data(images_dataset_list[img_idx])
        infer_ouput = self._load_output_data()
        self.dataset_list.append([img_input, infer_ouput])
# 2.2 创建线程tid，并将该tid线程指定为处理Stream上回调函数的线程。
# 其中ProcessCallback为线程函数，在该函数内调用acl.rt.process_report接口，等待指定时间后，触发回调函数处理。
tid, ret = acl.util.start_thread(self._process_callback, [self.context, 50])
# 2.3 指定处理Stream上回调函数的线程。
ret = acl.rt.subscribe_report(tid, self.stream)
# 2.4 创建回调函数，用户处理模型推理的结果，由用户自行定义。
def callback_func(self, delete_list):
    for temp in delete_list:
        _, infer_output = temp
        # device to host
        num = acl.mdl.get_dataset_num_buffers(infer_output)
        for i in range(num):
            temp_output_buf = acl.mdl.get_dataset_buffer(infer_output, i)
            infer_output_ptr = acl.get_data_buffer_addr(temp_output_buf)
            infer_output_size = acl.get_data_buffer_size(temp_output_buf)
            output_host, ret = acl.rt.malloc_host(infer_output_size)
            ret = acl.rt.memcpy(output_host,
                                infer_output_size,
                                infer_output_ptr,
                                infer_output_size,
                                ACL_MEMCPY_DEVICE_TO_HOST)
            output_host_dict = ["buffer": output_host, "size": infer_output_size}]
            result = self.get_result(output_host_dict)
            st = struct.unpack("1000f", bytearray(result[0]))
            vals = np.array(st).flatten()
            top_k = vals.argsort()[-1:-6:-1]
            print("\n======== top5 inference results: =============")
```



```
            for n in top_k:
                print("[%d]: %f" % (n, vals[n]))
            ret = acl.rt.free_host(output_host)
# 2.5 自定义函数forward，执行模型推理。
def forward(self):
    self.excute_dataset = []
    for idx in range(self.excute_times):
        img_data, infer_output = self.dataset_list.pop(0)
        ret = acl.mdl.execute_async(self.model_id,
                                    img_data,
                                    infer_output,
                                    self.stream)
        if self.is_callback:
            self.excute_dataset.append([img_data, infer_output])
            self._get_callback(idx)
# 2.6 对于异步推理，需阻塞应用程序运行，直到指定Stream中的所有任务都完成。
ret = acl.rt.synchronize_stream(self.stream)
# 2.7 取消线程注册，Stream上的回调函数不再由指定线程处理。
ret = acl.rt.unsubscribe_report(tid, self.stream)
self.is_exist = True
ret = acl.util.stop_thread(tid)

# 3.释放运行管理资源。
ret = acl.rt.destroy_stream(self.stream)
ret = acl.rt.destroy_context(self.context)
ret = acl.rt.reset_device(self.model_id)

# 4.pyACL去初始化。
ret = acl.finalize()
# ......
```

7.6 模型动态推理

7.6.1 动态 Batch/动态分辨率/动态维度（设置多档维度值）

接口调用流程

动态Shape输入场景下模型推理与**4 开发基础推理应用**的流程类似，都涉及pyACL初始化与去初始化、运行管理资源申请与释放、模型构建、模型加载、模型执行、模型卸载等。

本节中重点描述动态Shape输入场景下模型推理与**4 开发基础推理应用**的不同之处：

1. **构建模型时**，需配置动态Batch、动态分辨率、动态维度（ND格式）相关的信息：

   若模型推理时包含动态Batch特性，在模型推理时，需调用pyACL提供的接口设置模型推理时需使用的“batch size”，模型支持的“batch size”已提前在构建模型时配置（使用ATC工具的“dynamic_batch_size”参数）。

   若模型推理时包含动态分辨率特性，在模型推理时，需调用pyACL提供的接口设置模型推理时需使用的分辨率，模型支持的分辨率已提前在构建模型时配置（使用ATC工具的“dynamic_image_size”参数）。

   若模型推理时包含动态维度（ND格式）特性，在模型推理时，需调用pyACL提供的接口设置模型推理时需使用的维度值，模型支持哪些维度值已提前在构建模型时配置（使用ATC工具的“dynamic_dims”参数）。

   构建模型成功后，在生成的om模型中，会新增相应的输入（下文简称动态Batch/动态分辨率/动态维度输入），在模型推理时通过该新增的输入提供具体的Batch值/分辨率/维度值。

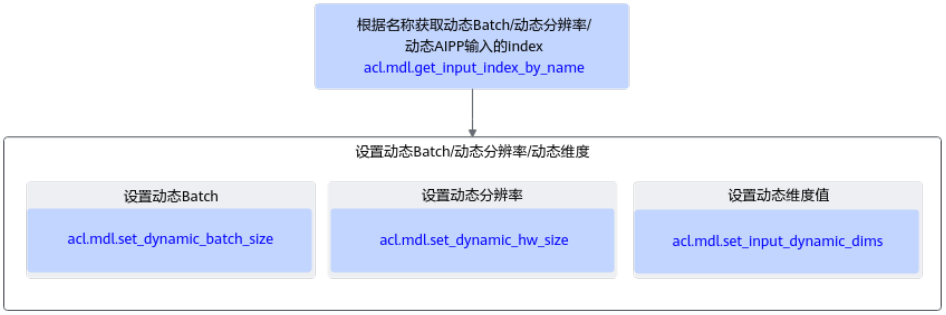
例如，a输入的“batch size”是动态的，在om模型中，会新增与a对应的b输入来描述a的batch信息。在模型执行时，准备a输入的数据结构请参见**准备模型执行的输入/输出数据结构**，准备b输入的数据结构、设置b输入的数据请参见**2**。

ATC工具的参数说明请参见《ATC工具使用指南》。

2. **在执行模型推理前**：
   - 需准备动态Batch/动态分辨率/动态维度输入的数据结构：
     i. 申请动态Batch/动态分辨率/动态维度输入对应的内存前，需要先调用acl.mdl.get_input_index_by_name接口根据输入名称（固定为“ascend_mbatch_shape_data”）获取模型中标识该输入的index。
     ii. 调用acl.mdl.get_input_size_by_index根据index获取输入内存大小。
     iii. 调用acl.rt.malloc接口根据**2.ii**中的大小申请内存。

     申请动态Batch/动态分辨率/动态AIPP/动态维度输入对应的内存后，无需用户设置该内存中的数据（否则可能会导致业务异常），用户调用**2.ii**中的接口后，系统会自动向该内存中填入数据。
     iv. 调用acl.create_data_buffer接口创建aclDataBuffer类型的数据，用于存放动态Batch/动态分辨率/动态维度输入数据的内存地址、内存大小。
     v. 调用acl.mdl.create_dataset接口创建aclmdlDataset类型的数据，并调用acl.mdl.add_dataset_buffer接口向aclmdlDataset类型的数据中增加aclDataBuffer类型的数据。
   - 需设置动态Batch/动态分辨率/动态维度参数值：

     图 7-5 接口调用流程



     i. 调用acl.mdl.get_input_index_by_name接口根据输入名称（固定为“ascend_mbatch_shape_data”）获取模型中标识该输入的index。
     ii. 设置动态Batch/动态分辨率/动态维度参数值。
       - 调用acl.mdl.set_dynamic_batch_size接口设置动态Batch。

         此处设置的“batch size”只能是构建模型时设置的Batch档位中的某一个。

         也可以调用acl.mdl.get_dynamic_batch接口获取指定模型支持的Batch档位数以及每一档中的“batch size”。
       - 调用acl.mdl.set_dynamic_hw_size接口设置动态分辨率。

         此处设置的分辨率只能是构建模型时设置的分辨率档位中的某一个。

         也可以调用acl.mdl.get_dynamic_hw接口获取指定模型支持的分辨率档位数以及每一档中的宽、高。

- 调用acl.mdl.set_input_dynamic_dims接口设置动态维度的维度值。
  此处设置的动态维度的值只能是构建模型时设置的档位中的某一档。
  也可以调用acl.mdl.get_input_dynamic_dims接口获取指定模型支持的动态维度档位数以及每一档中的值。

须知

- 对同一个模型，不能同时调用acl.mdl.set_dynamic_batch_size接口设置动态Batch、调用acl.mdl.set_dynamic_hw_size接口设置动态分辨率、调用acl.mdl.set_input_dynamic_dims接口设置动态维度的维度值，只能调用其中一种。
- 申请模型推理的输出内存时，可以按照各档位的实际大小申请内存，也可以调用acl.mdl.get_output_size_by_index接口获取内存大小后再申请内存（建议使用该方式，确保内存足够）。
- 动态AIPP和动态Batch同时使用时：
  - 调用acl.mdl.create_aipp接口设置“batchSize”时，“batchSize”要设置为最大Batch数。
  - 模型中需要进行动态AIPP处理的data节点，其对应的输入内存大小需按照最大Batch来申请。
- 动态AIPP和动态分辨率同时使用时：
  - 若在设置动态AIPP参数时，**开启了**抠图或缩放或补边功能，则不能与动态分辨率同时使用。
  - 若在设置动态AIPP参数时，**未开启**抠图或缩放或补边功能，在与动态分辨率同时使用时，需确保通过acl.mdl.set_aipp_src_image_size接口设置的宽和高，与通过acl.mdl.set_dynamic_hw_size接口设置的宽和高相等，都必须设置成模型转换时动态分辨率最大档位的宽和高。
  - 模型中需要进行动态AIPP处理的data节点，其对应的输入内存大小需按照最大分辨率（宽、高）来申请。
- 静态AIPP和动态分辨率同时使用时，由于动态分辨率场景下输入图片的宽和高不确定，因此在使用ATC工具的“insert_op_conf”参数传入AIPP配置文件时，AIPP配置文件中不能开启Crop和Padding功能，并且需要将配置文件中的“src_image_size_w”和“src_image_size_h”取值设置为0。
- 对同一个模型，**AIPP**（包括静态AIPP和动态AIPP）与动态维度（ND格式）不能同时使用。

动态 Batch 示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.模型加载，加载成功后，再设置动态Batch。
# ......

# 2.创建aclmdlDataset类型的数据，用于描述模型的输入数据input、输出数据output。
# ......

# 3.自定义函数，设置动态Batch。
def model_set_dynamicinfo():
```



```
    # 3.1 获取动态Batch输入的index，标识动态Batch输入的输入名称固定为“ascend_mbatch_shape_data”。
    index, ret = acl.mdl.get_input_index_by_name(model_desc, "ascend_mbatch_shape_data")
    # 3.2 设置Batch，model_id表示加载成功的模型的ID，input表示aclmdlDataset类型的数据，index表示标识动态Batch输入的输入index。
    batchSize = 8
    ret = acl.mdl.set_dynamic_batch_size(model_id_, input, index, batch_size)
    # ......

# 4.自定义函数，执行模型。
def model_execute(index):
    # 4.1 调用自定义函数，设置动态Batch。
    ret = model_set_dynamicinfo()
    # 4.2 执行模型，model_id表示加载成功的模型的ID，input和output分别表示模型的输入和输出。
    ret = acl.mdl.execute(model_id, input, output)
    # ......
}

# 5.处理模型推理结果。
# ......
```

动态分辨率示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.模型加载，加载成功后，再设置动态分辨率。
# ......

# 2.创建aclmdlDataset类型的数据，用于描述模型的输入数据input、输出数据output。
# ......

# 3.自定义函数，设置动态分辨率。
def model_set_dynamicInfo():
    # 3.1 获取动态分辨率输入的index，标识动态分辨率输入的输入名称固定为“ascend_mbatch_shape_data”。
    index, ret = acl.mdl.get_input_index_by_name(modelDesc, "ascend_mbatch_shape_data")
    # 3.2 设置输入图片分辨率，model_id表示加载成功的模型的ID，input表示aclmdlDataset类型的数据，index表示标识动态分辨率输入的输入index。
    height = 224
    width = 224
    ret = acl.mdl.set_dynamic_hw_size(model_id, input, index, height, width)
    # ......

# 4.自定义函数，执行模型。
def model_execute(index):
    # 4.1 调用自定义函数，设置动态分辨率。
    ret = model_set_dynamicInfo()
    # 4.2 执行模型，model_id表示加载成功的模型的ID，input和output分别表示模型的输入和输出。
    ret = acl.mdl.execute(model_id, input, output)
    # ......

# 5.处理模型推理结果。
# ......
```

ND 格式，动态维度示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 1.模型加载，加载成功后，再设置动态维度。
# ......

# 2.准备模型描述信息model_desc，准备模型的输入数据input_dataset和模型的输出数据output_dataset。
# ......
```



```
# 3.自定义函数，设置动态维度。
def model_set_dynamic_info():
    # 3.1 获取动态维度输入的index，标识动态维度输入的输入名称固定为“ascend_mbatch_shape_data”。
    index, ret = acl.mdl.get_input_index_by_name(model_desc, "ascend_mbatch_shape_data")
    # 3.2 设置具体档位信息，包括维度数dimCount和各个维度的数值，model_id表示加载成功的模型的ID，
input_dataset表示aclmdlDataset类型的数据，index表示标识动态维度输入的输入index。
    current_dims = {'name': '', 'dimCount': 4, 'dims': [8, 3, 224, 224]}
    ret = acl.mdl.set_input_dynamic_dims(mode_id, input_dataset, index, current_dims)
    # ......

# 4.自定义函数，执行模型。
def ModelExecute(int index):
    # 4.1 调用自定义函数，设置动态维度。
    ret = model_set_dynamic_info()
    # 4.2 执行模型，model_id表示加载成功的模型的ID，input_dataset和output_dataset分别表示模型的输入和
输出。
    ret = acl.mdl.execute(model_id, input_dataset, output_dataset)
    # ......

# 5.处理模型推理结果。
# ......
```

7.6.2 动态 Shape 输入（设置 Shape 范围）

Atlas 200/300/500 推理产品**不支持该特性**。

Atlas 200/500 A2推理产品**不支持该特性**。

接口调用流程

如果模型输入Shape是动态的，在模型执行之前调acl.mdl.set_dataset_tensor_desc设置该输入的Tensor描述信息（主要是设置Shape信息），在模型执行之后，调用acl.mdl.get_dataset_tensor_desc接口获取模型动态输出的Tensor描述信息，再进一步调用aclTensorDesc数据类型的操作接口获取输出Tensor数据占用的内存大小、Tensor的Format信息、Tensor的维度信息等。

关键原理说明如下：

1. 构建模型。

   模型推理场景下，对于动态Shape的输入数据，使用ATC工具转换模型时，通过“input_shape”参数设置输入Shape范围。

   ATC工具的参数说明请参见《ATC工具使用指南》。

2. 加载模型。

   模型加载的详细流程，请参见**4.6.1 模型加载**，模型加载成功后，返回标识模型的ID。

3. 创建aclmdlDataset类型的数据，用于描述模型执行的输入、输出。

   详细调用流程请参见**准备模型执行的输入/输出数据结构**。

   注意点如下：

   - 当调用acl.mdl.get_input_size_by_index获取到的size大小为0时，表示该**输入**的Shape是动态的，用户可根据实际情况预估一块较大的**输入**内存。
   - 当调用acl.mdl.get_output_size_by_index获取到的size大小为0时，表示该**输出**的Shape是动态的，用户可根据实际情况预估一块较大的**输出**内存。

4. 在成功加载模型之后，执行模型之前，调用acl.mdl.set_dataset_tensor_desc接口设置动态Shape输入的Tensor描述信息（主要是设置Shape信息）。

在调用acl.create_tensor_desc接口创建Tensor描述信息时，设置Shape信息，包括维度个数、维度大小，此处设置的维度个数、维度大小必须在模型构建时设置的输入Shape范围内，模型构建的详细说明请参见**4.2 模型构建**。

5. 执行模型。

   例如，调用acl.mdl.execute接口（同步接口）执行模型。

6. 获取模型执行的结果数据。

   调用acl.mdl.get_dataset_tensor_desc接口获取动态Shape输出的Tensor描述信息，再利用aclTensorDesc数据类型的操作接口获取Tensor描述信息的属性，此处以获取size（表示Tensor数据占用的空间大小）为例，然后从内存中读取对应size的数据。

须知

- 对同一个模型，acl.mdl.set_dataset_tensor_desc、acl.mdl.set_dynamic_batch_size接口、acl.mdl.set_dynamic_hw_size接口和acl.mdl.set_input_dynamic_dims接口，只能调用其中一个接口。
- 动态AIPP和动态Shape输入（设置Shape范围）同时使用时，动态AIPP的输出图片宽、高要在所设置的Shape范围内。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
// 此例中假设模型的第一个输入为动态输入，其index为0；模型的第一个输出为动态输出，其index为0。
# 1.模型加载，加载成功后，再设置动态输入的Tensor描述信息，主要设置动态输入的Shape信息。
# ......

# 2.准备模型描述信息modelDesc_，准备模型的输入数据input_和模型的输出数据output_。
# 此处需注意：
# 当利用acl.mdl.get_input_size_by_index获取到的size大小为0时，表示该输入的Shape是动态的，用户可根据实际情况预估一块较大的输入内存。
# 当利用acl.mdl.get_output_size_by_index获取到的size大小为0时，表示该输出的Shape是动态的，用户可根据实际情况预估一块较大的输出内存。
# ......

# 3.自定义函数，设置动态输入的Tensor描述信息。
def set_tensor_desc():
    # ......
    # 创建Tensor描述信息，其中dataType和format不需要设置，pyACL直接从模型中获取，此处使用的默认值。
    #shape需要和给定的输入数据的shape一致。
    shapes = [1, 3, 224, 224]
    inputDesc = acl.create_tensor_desc(0, shapes, 0)

    # 设置index为0的动态输入的Tensor描述信息。
    ret = acl.mdl.set_dataset_tensor_desc(input_, inputDesc, 0)
    # ......


def model_execute():
    # ......
    # 调用自定义接口，设置动态输入的Tensor描述信息。
    set_tensor_desc()

    # 执行模型。
    ret = acl.mdl.execute(modelId, input_, output_)

    # 获取index为0的动态输出的Tensor描述信息。
```



```
    outputDesc = acl.mdl.get_dataset_tensor_desc(output_, 0)

    # 利用aclTensorDesc数据类型的操作接口获取outputDesc的属性，此处需要获取size（表示Tensor数据占用的空间大小），然后从内存中读取对应size的数据。
    outputFileName = str(ss)

    outputDesc_size = acl.get_tensor_desc_size(outputDesc)
    dataBuffer = acl.mdl.get_dataset_buffer(output_, 0)
    data = acl.get_data_buffer_addr(dataBuffer)
    outHostData = None

    # 调用acl.rt.get_run_mode接口获取软件栈的运行模式，并根据运行模式判断是否进行数据传输。
    runMode = None
    ret = acl.rt.get_run_mode(runMode)
    if runMode == ACL_HOST:
        ret = acl.rt.malloc_host(outHostData, outputDesc_size)

        # 由于动态shape申请的内存比较大，而真实数据的大小是outputDesc_size，所以此处用真实数据大小去拷贝内存。
        ret = acl.rt.memcpy(outHostData, outputDesc_size, data, outputDesc_size, ACL_MEMCPY_DEVICE_TO_HOST)
        with open(outputFileName, "wb") as f:
            f.write(outHostData)
        ret = acl.rt.free_host(outHostData)
    else:
        with open(outputFileName, "wb") as f:
            f.write(data)
    # ......

# 5.处理模型推理结果
# TODO
```

7.7 模型动态 AIPP 推理

7.7.1 接口调用流程

动态AIPP场景下模型推理与**4 开发基础推理应用**的流程类似，都涉及pyACL初始化与去初始化、运行管理资源申请与释放、模型构建、模型加载、模型执行、模型卸载等。

本节中重点描述动态AIPP场景下模型推理与**4 开发基础推理应用**的不同之处：

- **模型构建时**，需配置动态AIPP相关参数：

  构建模型成功后，在生成的om模型中，会新增相应的输入（下文简称动态AIPP输入），在模型推理时通过该新增的输入提供具体的AIPP配置值。

  例如，a输入的AIPP配置是动态的，在om模型中，会有与a对应的b输入来描述a的AIPP配置信息。在模型执行时，准备a输入的数据结构请参见**准备模型执行的输入/输出数据结构**，准备b输入的数据结构、设置b输入的数据请参见以下内容。

- **在执行模型推理前**：
  - 准备动态AIPP输入的数据结构：
    i. 申请动态AIPP输入对应的内存前，需要先调用acl.mdl.get_input_index_by_name接口根据输入名称（固定为“ascend_dynamic_aipp_data”）获取模型中标识该输入的index。
    ii. 调用acl.mdl.get_input_size_by_index根据index获取输入内存大小。
    iii. 调用acl.rt.malloc接口根据**ii**中的大小申请内存。

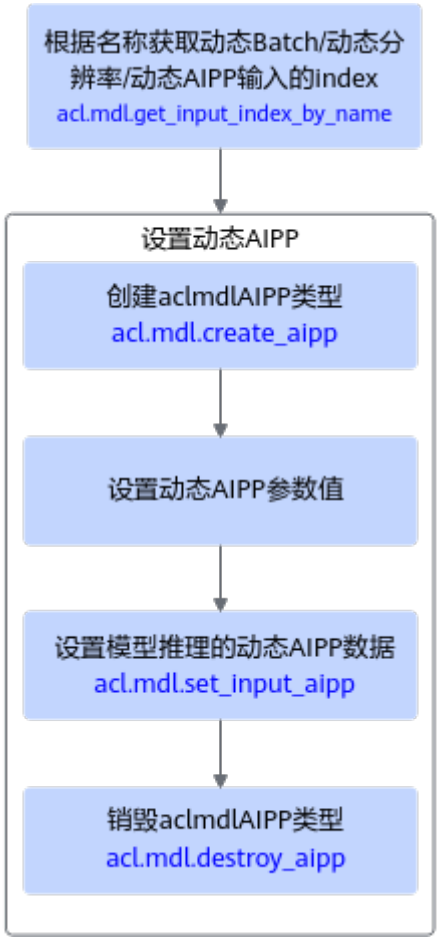
申请动态AIPP输入对应的内存后，无需用户设置该内存中的数据（否则可能会导致业务异常），用户调用ii中的接口后，系统会自动向该内存中填入数据。

iv. 调用acl.create_data_buffer接口创建aclDataBuffer类型的数据，用于存放动态AIPP输入数据的内存地址、内存大小。

v. 调用acl.mdl.create_dataset接口创建aclmdlDataset类型的数据，并调用acl.mdl.add_dataset_buffer接口向aclmdlDataset类型的数据中增加aclDataBuffer类型的数据。

- 设置动态AIPP参数值：

图 7-6 接口调用流程



i. 调用acl.mdl.get_input_index_by_name接口根据输入名称（固定为“ACL_DYNAMIC_AIPP_NAME”）获取模型中标识该输入的index。

ii. 设置动态AIPP参数值。

1) 调用acl.mdl.create_aipp接口创建aclmdlAIPP类型。

2) 根据实际需求，调用aclmdlAIPP数据类型下的操作接口设置动态AIPP参数值。

3) 动态AIPP场景下，acl.mdl.set_aipp_src_image_size接口（设置原始图片的宽和高）必须调用。

4) 调用acl.mdl.set_input_aipp接口设置模型推理时的动态AIPP数据。

5) 及时调用acl.mdl.destroy_aipp接口销毁aclmdlAIPP类型。

说明

pyACL还提供了基于DVPP（Digital Vision Pre-Processing）硬件进行媒体数据处理的功能，包括缩放、抠图、格式转换、图片编解码、视频编解码等，功能比AIPP丰富，但对于输入/输出图片、内存有一定的约束。

基于DVPP的媒体数据处理接口介绍，请参见**5 媒体数据处理**。

7.7.2 动态 AIPP（单个动态 AIPP 输入）

基本原理

若模型推理时包含**动态AIPP**特性，在模型推理时，需调用pyACL提供的接口设置模型推理时需使用的AIPP配置，模型支持的AIPP模式已提前在构建模型时配置（使用ATC工具的“insert_op_conf”参数）。

ATC工具的参数说明请参见《ATC工具使用指南》。

须知

- 动态AIPP和动态Batch同时使用时：
  - 调用acl.mdl.create_aipp接口设置“batchSize”时，“batchSize”要设置为最大Batch数。
  - 模型中需要进行动态AIPP处理的data节点，其对应的输入内存大小需按照最大Batch来申请。
- 动态AIPP和动态分辨率同时使用时：
  - 若在设置动态AIPP参数时，**开启了**抠图或缩放或补边功能，则不能与动态分辨率同时使用。
  - 若在设置动态AIPP参数时，**未开启**抠图或缩放或补边功能，在与动态分辨率同时使用时，需确保通过acl.mdl.set_aipp_src_image_size接口设置的宽和高，与通过acl.mdl.set_dynamic_hw_size接口设置的宽和高相等，都必须设置成模型转换时动态分辨率最大档位的宽和高。
  - 模型中需要进行动态AIPP处理的data节点，其对应的输入内存大小需按照最大分辨率（宽、高）来申请。
- 对同一个模型，**AIPP**（包括静态AIPP和动态AIPP）与动态维度（ND格式）不能同时使用。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

ACL_YUV420SP_U8 = 1

# 1.模型加载，加载成功后，再设置动态AIPP参数值。
# ......

# 2.创建aclmdlDataset类型的数据，用于描述模型的输入数据input_dataset和模型的输出数据output_dataset。
```



```
# ......

# 3.自定义函数，设置动态AIPP参数值。
def model_set_dynamic_aipp():
    # 3.1 获取标识动态AIPP输入的index。
    index, ret = acl.mdl.get_input_index_by_name(model_desc, "ascend_dynamic_aipp_data")

    # 3.2 设置动态AIPP参数值。
    batch_number = 1
    aipp_dynamic_set = acl.mdl.create_aipp(batch_number)
    ret = acl.mdl.set_aipp_src_image_size(aipp_dynamic_set, 256, 224)
    ret = acl.mdl.set_aipp_input_format(aipp_dynamic_set, ACL_YUV420SP_U8)
    ret = acl.mdl.set_aipp_csc_params(aipp_dynamic_set, 1, 256, 443, 0, 256, -86, -178, 256, 0, 350, 0, 0, 0,
0, 128, 128)
    ret = acl.mdl.set_aipp_rbuv_swap_switch(aipp_dynamic_set, 0)
    ret = acl.mdl.set_aipp_dtc_pixel_mean(aipp_dynamic_set, 0, 0, 0, 0, 0)
    ret = acl.mdl.set_aipp_dtc_pixel_min(aipp_dynamic_set, 0, 0, 0, 0, 0)
    ret = acl.mdl.set_aipp_pixel_var_reci(aipp_dynamic_set, 1, 1, 1, 1, 0)
    ret = acl.mdl.set_aipp_crop_params(aipp_dynamic_set, 1, 0, 0, 224, 224, 0)
    ret = acl.mdl.set_input_aipp(model_id, input_dataset, index, aipp_dynamic_set)
    ret = acl.mdl.destroy_aipp(aipp_dynamic_set)
    # ......

# 4.自定义函数，执行模型。
def model_execute(index)
    # 4.1 调用自定义函数，设置动态AIPP参数值。
    ret = model_set_dynamic_aipp()
    # 4.2 执行模型，modelId_表示加载成功的模型的ID，input_和output_分别表示模型的输入和输出。
    ret = acl.mdl.execute(model_id, input_dataset, output_dataset)
    # ......

# 5.处理模型推理结果。
# ......
```

7.7.3 动态 AIPP（多个动态 AIPP 输入）

基本原理

模型有多个动态AIPP输入时的推理基本流程与单个动态AIPP输入类似，请参见**动态AIPP（单个动态AIPP输入）**。

多个动态AIPP输入与单个动态AIPP输入的不同点如下：

- 需调用acl.mdl.get_aipp_type接口查询指定模型的指定输入是否有关联的动态AIPP输入，若有，则输出标识动态AIPP输入的index，该参数值可作为acl.mdl.set_aipp_by_input_index接口的入参之一，来设置对应输入上的动态AIPP参数值。
- 为避免在错误的输入上设置动态AIPP参数，用户可调用acl.mdl.get_input_name_by_index接口先获取指定输入index的输入名称，根据输入名称所对应的index设置动态AIPP参数。

示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

ACL_DATA_WITH_DYNAMIC_AIPP = 2
ACL_YUV420SP_U8 = 1

# 1.模型加载，加载成功后，再设置动态AIPP参数值。
# ......
```



```
# 2.准备模型描述信息model_desc，准备模型的输入数据input_dataset和模型的输出数据output_dataset。
# ......

# 3.自定义函数，设置动态AIPP参数值。
def model_set_dynamic_aipp():
    # 3.1 获取标识动态AIPP输入的index。
    need_dynamic_aipp = []
    input_num = acl.mdl.get_num_inputs(model_desc)
    for index in range(input_num):
        aipp_type, dynamic_attached_index, ret = acl.mdl.get_aipp_type(model_id, index)
        if aipp_type == ACL_DATA_WITH_DYNAMIC_AIPP:
            need_dynamic_aipp.append(index)

    # 3.2 当前示例中以2个动态AIPP输入为例，用户可根据实际情况修改。
    if len(need_dynamic_aipp) != 2:
        return 1
    # 创建第一个动态aipp配置参数。
    batch_number_first = 1
    aipp_dynamic_set_first = acl.mdl.create_aipp(batch_number_first)
    ret = acl.mdl.set_aipp_src_image_size(aipp_dynamic_set_first, 256, 224)
    ret = acl.mdl.set_aipp_input_format(aipp_dynamic_set_first, ACL_YUV420SP_U8)
    ret = acl.mdl.set_aipp_csc_params(aipp_dynamic_set_first, 1, 256, 443, 0, 256, -86,
                                      -178, 256, 0, 350, 0, 0, 0, 0, 128, 128)
    ret = acl.mdl.set_aipp_rbuv_swap_switch(aipp_dynamic_set_first, 0)
    ret = acl.mdl.set_aipp_dtc_pixel_mean(aipp_dynamic_set_first, 0, 0, 0, 0, 0)
    ret = acl.mdl.set_aipp_dtc_pixel_min(aipp_dynamic_set_first, 0, 0, 0, 0, 0)
    ret = acl.mdl.set_aipp_pixel_var_reci(aipp_dynamic_set_first, 1.0, 1.0, 1.0, 1.0, 0)
    ret = acl.mdl.set_aipp_crop_params(aipp_dynamic_set_first, 1, 2, 2, 224, 224, 0)
    # 设置模型推理时的动态aipp参数值。
    ret = acl.mdl.set_aipp_by_input_index(model_id, input_dataset, need_dynamic_aipp[0],
                                          aipp_dynamic_set_first)
    ret = acl.mdl.destroy_aipp(aipp_dynamic_set_first)
    # 创建第二个动态aipp配置参数。
    batch_number_second = 2
    aipp_dynamic_set_second = acl.mdl.create_aipp(batch_number_second)
    ret = acl.mdl.set_aipp_src_image_size(aipp_dynamic_set_second, 224, 224)
    # 此处可以继续调用其它AIPP参数设置接口。
    # 设置模型推理时的动态aipp参数值。
    ret = acl.mdl.set_aipp_by_input_index(model_id, input_dataset, need_dynamic_aipp[1],
                                          aipp_dynamic_set_second)
    ret = acl.mdl.destroy_aipp(aipp_dynamic_set_second)
    return ret

# 4.自定义函数，执行模型。
def ModelExecute(int index):
    # 4.1 调用自定义函数，设置动态AIPP参数值。
    ret = model_set_dynamic_aipp()
    # 4.2 执行模型，modelId_表示加载成功的模型的ID，input_dataset和output_dataset分别表示模型的输入和
输出。
    ret = acl.mdl.execute(model_id, input_dataset, output_dataset)
    # ......

# 5.处理模型推理结果。
# ......
```

7.8 Stream 管理

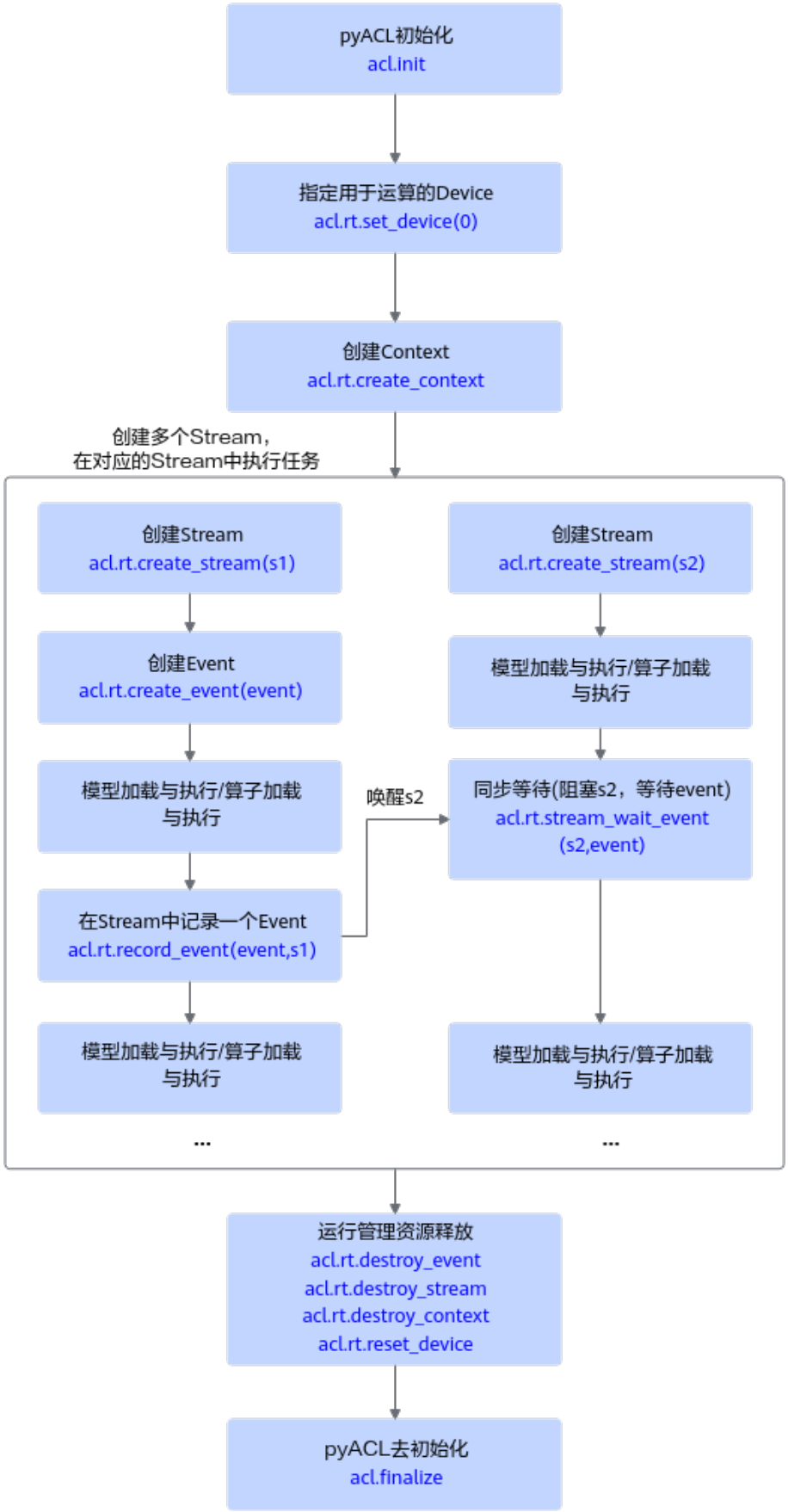
7.8.1 原理介绍

在pyACL中，Stream是一个任务队列，应用程序通过Stream来管理任务的并行，一个Stream内部的任务保序执行，即Stream根据发送过来的任务依次执行；不同Stream中的任务并行执行。一个默认Context下会挂一个默认Stream，如果不显式创建Stream，可使用默认Stream。默认Stream作为接口入参时，直接传0。

7.8.2 多 Stream 接口调用流程

开发应用时，如果涉及多Stream之间的任务等待，则应用程序中必须包含相关的代码逻辑，关于该场景的接口调用流程，请依次参见**3.3 pyACL接口调用流程**以及本节中的说明。

图 7-7 同步等待流程_多 Stream 场景



多Stream之间任务的同步等待可以利用Event实现，调用acl.rt.stream_wait_event接口阻塞指定Stream的运行，直到指定的Event完成。需在调用acl.rt.stream_wait_event接口前，先调用acl.rt.record_event接口。调用示例请参见**7.9.4 关于Stream间任务的同步等待**。

模型加载与执行的流程请参见**4 开发基础推理应用**。

算子加载与执行的流程请参见**6 单算子调用**。

7.8.3 单线程单 Stream

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
ACL_MEMCPY_HOST_TO_DEVICE = 1

# 显式创建一个Stream。
stream, ret = acl.rt.create_stream()

# 调用触发任务的接口，传入stream参数。
ret = acl.rt.memcpy_async(dev_ptr, dev_size, host_ptr, host_size, ACL_MEMCPY_HOST_TO_DEVICE, stream)
# 调用acl.rt.synchronize_stream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。
ret = acl.rt.synchronize_stream(stream)

# Stream使用结束后，显式销毁Stream。
ret = acl.rt.destroy_stream(stream)
# ......
```

7.8.4 单线程多 Stream

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

device_id = 0
model_id_1 = 0
model_id_2 = 1

# 如果只创建了一个Context，线程默认将这个Context作为线程当前的Context。
# 如果是多个Context，则需要调用acl.rt.create_context接口设置当前线程的Context。
context, ret = acl.rt.create_context(device_id)

stream1, ret = acl.rt.create_stream()
# 调用触发任务的接口，例如异步模型推理，任务下发在stream1。
ret = acl.mdl.execute_async(model_id_1, dataset_in_1, dataset_out_1, stream1)

stream2, ret = acl.rt.create_stream()
# 调用触发任务的接口，例如异步模型推理， 任务下发在stream2。
ret = acl.mdl.execute_async(model_id_2, dataset_in_2, dataset_out_2, stream2)

# 流同步。
ret = acl.rt.synchronize_stream(stream1)
ret = acl.rt.synchronize_stream(stream2)

# 释放资源。
ret = acl.rt.destroy_stream(stream2)
ret = acl.rt.destroy_stream(stream1)
ret = acl.rt.destroy_context(context)

# ....
```


7.8.5 多线程多 Stream

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
def run_thread(stream):
    device_id = 0

    # 如果只创建了一个Context，线程默认将这个Context作为线程当前的Context。
    # 如果是多个Context，则需要调用acl.rt.set_context接口设置当前线程的Context。
    context, ret = acl.rt.create_context(device_id)
    stream, ret = acl.rt.create_stream()

    # 调用触发任务的接口。
    # ....

    # 释放资源。
    ret = acl.rt.destroy_stream(stream)
    ret = acl.rt.destroy_context(context)


# 创建2个线程，每个线程对应一个Stream。
thread_id1, ret = acl.util.start_thread(run_thread, stream1)
thread_id2, ret = acl.util.start_thread(run_thread, stream2)
# 显式调用join函数确保结束线程。
ret = acl.util.stop_thread(thread_id1)
ret = acl.util.stop_thread(thread_id2)
```

7.9 同步等待

7.9.1 基本原理

同步机制

pyACL提供以下几种同步机制：

- Event的同步等待：调用acl.rt.synchronize_event接口，阻塞应用程序运行，等待Event完成。
- Stream内任务的同步等待：调用acl.rt.synchronize_stream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。
- Stream间任务的同步等待：调用acl.rt.stream_wait_event接口，阻塞指定Stream的运行，直到指定的Event完成。支持多个Stream等待同一个Event的场景。接口调用流程请参见**7.8.2 多Stream接口调用流程**。
- Device的同步等待：调用acl.rt.synchronize_device接口，阻塞应用程序运行，直到正在运算中的Device完成运算。多Device场景下，调用该接口等待的是当前Context对应的Device，接口调用流程请参见**7.2 多Device场景**。

7.9.2 关于 Event 的同步等待

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
# 创建一个Event。
```



```
event, ret = acl.rt.create_event()

# 创建一个Stream。
stream, ret = acl.rt.create_stream()

# stream末尾添加了一个event。
ret = acl.rt.record_event(event, stream)

# 阻塞应用程序运行，等待event发生，也就是stream执行完成。
# stream完成后产生event，唤醒执行应用程序的控制流，开始执行程序。
ret = acl.rt.synchronize_event(event)

# 显式销毁资源。
ret = acl.rt.destroy_stream(stream)
ret = acl.rt.destroy_event(event)
# ......
```

7.9.3 关于 Stream 内任务的同步等待

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

# 显式创建一个Stream。
stream, ret = acl.rt.create_stream()

# 调用触发任务的接口，传入stream参数。
ret = acl.rt.memcpy_async(dev_ptr, dev_size, host_ptr, host_size, ACL_MEMCPY_HOST_TO_DEVICE, stream)
# 调用acl.rt.synchronize_stream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。
ret = acl.rt.synchronize_stream(stream)

# Stream使用结束后，显式销毁Stream。
ret = acl.rt.destroy_stream(stream)
# ......
```

7.9.4 关于 Stream 间任务的同步等待

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......
# 创建一个Event。
event, ret = acl.rt.create_event()

# 创建两个Stream。
s1, ret = acl.rt.create_stream()
s2, ret = acl.rt.create_stream()

# 在s1末尾添加了一个event。
ret = acl.rt.record_event(event, s1)

# 阻塞s2运行，直到指定Event发生，也就是s1执行完成。
# s1完成后，唤醒s2，继续执行s2的任务。
ret = acl.rt.stream_wait_event(s2, event)

# 显式销毁资源。
ret = acl.rt.destroy_stream(s2)
ret = acl.rt.destroy_stream(s1)
ret = acl.rt.destroy_event(event)
# ......
```


7.9.5 关于 Device 的同步等待

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import acl
# ......

device_id = 0

# 指定Device。
ret = acl.rt.set_device(device_id)

# 创建context。
context, ret = acl.rt.create_context(device_id)

# 创建stream。
stream, ret = acl.rt.create_stream()

# 阻塞应用程序运行，直到正在运算中的Device完成运算。
ret = acl.rt.synchronize_device()

# 资源销毁。
ret = acl.rt.synchronize_stream(stream)
ret = acl.rt.destroy_context(context)
ret = acl.rt.reset_device(device_id)
```

7.10 AI Core 异常信息获取

基本原理

使用场景举例：执行整网模型推理时（**不支持动态Shape场景**），如果产生AI Core报错，可以调用本接口获取报错算子的描述信息，再做进一步错误排查。

推荐的接口调用顺序如下：

1. 定义并实现异常回调函数**fn(*aclrtExceptionInfoCallback类型*)**，回调函数原型请参见acl.rt.set_exception_info_callback。

   实现回调函数的关键步骤如下：

   a. 在异常回调函数**fn**内调用acl.rt.get_device_id_from_exception_info、acl.rt.get_stream_id_from_exception_info、acl.rt.get_task_id_from_exception_info接口分别获取Device ID、Stream ID、Task ID。

   b. 在异常回调函数**fn**内调用acl.mdl.create_and_get_op_desc接口获取算子的描述信息。

   c. 在异常回调函数**fn**内调用acl.get_tensor_desc_by_index接口获取指定算子输入/输出的Tensor描述。

   d. 在异常回调函数**fn**内如下接口获取Tensor描述中的数据，进行进一步分析。

      例如，调用acl.get_tensor_desc_address接口获取Tensor数据的内存地址（用户可从该内存地址中获取Tensor数据）、调用acl.get_tensor_desc_type接口获取Tensor描述中的数据类型、调用acl.get_tensor_desc_format接口获取Tensor描述中的Format、调用acl.get_tensor_desc_num_dims接口获取Tensor描述中的Shape维度个数、调用acl.get_tensor_desc_dim_v2接口获取Shape中指定维度的大小。

2. 调用acl.rt.set_exception_info_callback接口设置异常回调函数。

3. 执行模型推理。

   如果存在AI Core报错，则触发回调函数**fn**，获取算子的信息，进行进一步分析。

示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

示例中，运行管理资源申请与释放请参见**4.4 运行管理资源申请与释放**，模型加载的接口调用流程请参见**接口调用流程**，模型推理的接口调用流程、准备模型推理的输入/输出数据的接口调用流程请参见**准备模型执行的输入/输出数据结构**。

```
import acl
import numpy as np
# ......

# 1.申请运行管理资源，包括设置用于计算的Device、创建Context、创建Stream。
# ......

# 2.模型加载，加载成功后，返回标识模型的model_id。
# ......

# 3.创建aclmdlDataset类型的数据，用于描述模型的输入数据input、输出数据output。
# ......

# 4.实现异常回调函数。
def exception_callback(info):
    stream_id = acl.rt.get_stream_id_from_exception_info(info)
    thread_id = acl.rt.get_thread_id_from_exception_info(info)
    device_id = acl.rt.get_device_id_from_exception_info(info)
    task_id = acl.rt.get_task_id_from_exception_info(info)
    # 用户可以将获取的算子信息写入到文件，或者另起线程侦听异常回调，当发生异常回调时触发线程处理函数，在线程处理函数中将算子信息打屏。
    op_name, input_desc, num_inputs, output_desc, num_outputs, ret = \
        acl.mdl.create_and_get_op_desc(device_id, stream_id, task_id, 256)

    # 可以调用acl Tensor的相关接口，获取算子的相关信息，用户可以根据自己需要调用。
    for i in range(num_inputs):
        desc = acl.get_tensor_desc_by_index(input_desc, i)
        address = acl.get_tensor_desc_address(desc)
        num_dims = acl.get_tensor_desc_num_dims(desc)
        dim_0, ret = acl.get_tensor_desc_dim_v2(desc, 0)

    for i in range(num_outputs):
        desc = acl.get_tensor_desc_by_index(output_desc, i)
        address = acl.get_tensor_desc_address(desc)
        num_dims = acl.get_tensor_desc_num_dims(desc)
        dim_0, ret = acl.get_tensor_desc_dim_v2(desc, 0)

    acl.destroy_tensor_desc(input_desc)
    acl.destroy_tensor_desc(output_desc)

# 5.设置异常回调。
ret = acl.rt.set_exception_info_callback(exception_callback)

# 6.执行模型。
ret = acl.mdl.execute(model_id, input, output)

# 7.处理模型推理结果。
# ......

# 8.释放描述模型输入/输出信息、内存等资源，卸载模型。
# ......

# 9.释放运行管理资源。
# ......
```


7.11 Profiling 性能数据采集

基本原理

该章节下的接口用于Profiling采集性能数据，实现方式支持以下三种：

Profiling pyACL API（通过Profiling pyACL API采集并落盘性能数据）：实现将采集到的Profiling数据写入文件，再使用Profiling工具解析该文件（请参见《性能分析工具使用指南》下的“数据解析与导出”），并展示性能分析数据。

包括以下两种接口调用方式：

- acl.prof.init接口、acl.prof.start接口、acl.prof.stop接口、acl.prof.finalize接口配合使用，实现该方式的性能数据采集。该方式可获取pyACL的接口性能数据、AI Core上算子的执行时间、AI Core性能指标数据等。目前这些接口为进程级控制，表示在进程内任意线程调用该接口，其它线程都会生效。

  一个进程内，可以根据需求多次调用这些接口，基于不同的Profiling采集配置，采集数据。

- 调用acl.init接口，在pyACL初始化阶段，通过*.json 文件传入要采集的Profiling数据。该方式可获取pyACL的接口性能数据、AI Core上算子的执行时间、AI Core性能指标数据等。

  一个进程内，只能调用一次acl.init接口，如果要修改Profiling采集配置，需修改*.json文件中的配置。详细使用说明请参见acl.init接口处的说明，不在本章节描述。

Profiling pyACL API for Extension（Profiling pyACL API扩展接口）：当用户需要定位应用程序或上层框架程序的性能瓶颈时，可在Profiling采集进程内（acl.prof.start接口、acl.prof.stop接口之间）调用Profiling pyACL API扩展接口（统称为msproftx功能），开启记录应用程序执行期间特定事件发生的时间跨度，并将数据写入Profiling数据文件，再使用Profiling工具解析该文件，并导出展示性能分析数据。

Profiling工具解析导出操作请参见《性能分析工具使用指南》下的“Profiling数据解析”和“Profiling数据导出”。

一个进程内，可以根据需求多次调用这些接口。

接口调用方式：在acl.prof.start和acl.prof.stop接口之间调用acl.prof.create_stamp、acl.prof.push、acl.prof.pop、acl.prof.range_start、acl.prof.range_stop、acl.prof.destroy_stamp接口。该方式可获取应用程序执行期间特定时间发生的事件并记录事件发生的时间跨度。

一个进程内，可以根据需求多次调用这些接口。

Profiling pyACL API for Subscription（订阅算子信息的Profiling pyACL API）：实现将采集到的Profiling数据解析后写入管道，由用户读入内存，再由用户调用pyACL的接口获取性能数据。

接口调用方式：acl.prof.model_subscribe接口、acl.prof.get*接口、acl.prof.model_unsubscribe接口配合使用，实现该方式的性能数据采集，当前支持获取网络模型中算子的性能数据，包括算子名称、算子类型名称、算子执行时间等。

Profiling pyACL API 示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

示例中，运行管理资源申请与释放请参见**运行管理资源申请流程**与**运行管理资源释放流程**，模型加载的接口调用流程请参见**接口调用流程**，模型推理的接口调用流程、准备模型推理的输入/输出数据的接口调用流程请参见**准备模型执行的输入/输出数据结构**。

```
import acl
import numpy as np
# ......

# 1.申请运行管理资源，包括设置用于计算的Device、创建Context、创建Stream。
# ......

# 2.模型加载，加载成功后，返回标识模型的model_id。
# ......

# 3.创建aclmdlDataset类型的数据，用于描述模型的输入数据input、输出数据output。
# ......

# 4.profiling初始化。
# 设置数据落盘路径。
PROF_INIT_PATH='...'
ret = acl.prof.init(PROF_INIT_PATH)

# 5.进行profiling配置。
device_list = [0]
ACL_PROF_ACL_API = 0x0001
ACL_PROF_TASK_TIME = 0x0002
ACL_PROF_AICORE_METRICS = 0x0004
ACL_PROF_AICPU_TRACE = 0x0008
ACL_PROF_SYS_HARDWARE_MEM_FREQ = 3

# 创建配置类型指针地址。
prof_config = acl.prof.create_config(device_list, 0, 0, ACL_PROF_ACL_API | ACL_PROF_TASK_TIME |
ACL_PROF_AICPU | ACL_PROF_AICORE_METRICS | ACL_PROF_L2CACHE | ACL_PROF_HCCL_TRACE)
mem_freq = "15"
ret = acl.prof.set_config(ACL_PROF_SYS_HARDWARE_MEM_FREQ, mem_freq)
ret = acl.prof.start(prof_config)

# 6.执行模型。
ret = acl.mdl.execute(model_id, input, output)

# 7.处理模型推理结果。
# ......

# 8.释放描述模型输入/输出信息、内存等资源，卸载模型。
# ......

# 9.关闭profiling配置, 释放配置资源, 释放profiling组件资源。
ret = acl.prof.stop(prof_config)
ret = acl.prof.destroy_config(prof_config)
ret = acl.prof.finalize()

# 10.释放运行管理资源
# ......
```

Profiling pyACL API for Extension 示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

示例中，运行管理资源申请与释放请参见**运行管理资源申请流程**与**运行管理资源释放流程**，模型加载的接口调用流程请参见**接口调用流程**，模型推理的接口调用流程、准

备模型推理的输入/输出数据的接口调用流程请参见**准备模型执行的输入/输出数据结构**。

```
for i in range(200000):
    stamp = acl.prof.create_stamp()
     if stamp == 0:
        print("create stamp failed")
        return FAILED

    msg = "test msprof tx"
    msg_len = len(msg)
    ret = acl.prof.set_stamp_trace_message(stamp, msg, msg_len)
    ret = acl.prof.mark(stamp)

    ret = acl.prof.destroy_stamp(stamp)
```

或

```
for i in range(200000):
    stamp = acl.prof.create_stamp()
     if stamp == 0:
        print("create stamp failed")
        return FAILED

    msg = "test msprof tx"
    msg_len = len(msg)
    ret = acl.prof.set_stamp_trace_message(stamp, msg, msg_len)

# acl.prof.push 和 acl.prof.pop 接口配对使用，完成单线程采集
    ret = acl.prof.push(stamp)
    ret = acl.prof.pop()

    ret = acl.prof.destroy_stamp(stamp)
```

或

```
for i in range(200000):
    stamp = acl.prof.create_stamp()
     if stamp == 0:
        print("create stamp failed")
        return FAILED

    msg = "test msprof tx"
    msg_len = len(msg)
    ret = acl.prof.set_stamp_trace_message(stamp, msg, msg_len)

# acl.prof.range_start 和 acl.prof.range_stop 接口配对使用，可以完成多线程采集
    range_id = 0
    range_id, ret = acl.prof.range_start(stamp)
    ret = acl.prof.range_stop(range_id)

    ret = acl.prof.destroy_stamp(stamp)
```

Profiling pyACL API for Subscription 示例代码

调用接口后，需增加异常处理的分支，示例代码中不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

示例中，运行管理资源申请与释放请参见**运行管理资源申请流程**与**运行管理资源释放流程**，模型加载的接口调用流程请参见**接口调用流程**，模型推理的接口调用流程、准备模型推理的输入/输出数据的接口调用流程请参见**准备模型执行的输入/输出数据结构**。

```
import acl
import numpy as np
# ......

# 1.申请运行管理资源，包括设置用于计算的Device、创建Context、创建Stream。
```



```
# ......

# 2.模型加载，加载成功后，返回标识模型的model_id。
# ......

# 3.创建aclmdlDataset类型的数据，用于描述模型的输入数据input、输出数据output。
# ......

# 4.创建管道，用于读取以及写入模型订阅的数据。
r, w = os.pipe()

# 5.创建模型订阅的配置并且进行模型订阅。
ACL_AICORE_NONE = 0xFF
subscribe_config = acl.prof.create_subscribe_config(1, ACL_AICORE_NONE, w)
# 模型订阅需要传入模型的model_id。
ret = acl.prof.model_subscribe(model_id, subscribe_config)

# 6.实现管道读取订阅数据的函数。
# 6.1 自定义函数，实现从用户内存中读取订阅数据的函数。
def get_model_info(data, data_len):
    # 获取算子信息个数。
    op_number, ret = acl.prof.get_op_num(data, data_len)
    # 遍历用户内存的算子信息。
    for i in range(op_number):
        # 获取算子的模型id。
        model_id = acl.prof.get_model_id(data, data_len, i)
        # 获取算子的类型名称。
        op_type, ret = acl.prof.get_op_type(data, data_len, i, 65)
        # 获取算子的名称。
        op_name, ret = acl.prof.get_op_name(data, data_len, i, 275)
        # 获取算子的执行开始时间。
        op_start = acl.prof.get_op_start(data, data_len, i)
        # 获取算子的执行结束时间。
        op_end = acl.prof.get_op_end(data, data_len, i)
        # 获取算子执行的耗时时间。
        op_duration = acl.prof.get_op_duration(data, data_len, i)

# 6.2 自定义函数，实现从管道中读取数据到用户内存的函数。
def prof_data_read(args):
    fd, ctx = args
    ret = acl.rt.set_context(ctx)
    # 获取单位算子信息的大小（Byte）。
    buffer_size, ret = acl.prof.get_op_desc_size()
    # 设置每次从管道中读取的算子信息个数。
    N = 10
    # 计算存储算子信息的内存的大小。
    data_len = buffer_size * N
    # 从管道中读取数据到申请的内存中，读取到的实际数据大小可能小于buffer_size * N，如果管道中没有数
据，默认会阻塞直到读取到数据为止。
    while True:
        data = os.read(fd, data_len)
        if len(data) == 0:
            break
        np_data = np.array(data)

        bytes_data = np_data.tobytes()
        np_data_ptr = acl.util.bytes_to_ptr(bytes_data)
        size = np_data.itemsize * np_data.size
        # 调用6.1实现的函数解析内存中的数据。
        get_model_info(np_data_ptr, size)

# 7.启动线程读取管道数据并解析。
thr_id, ret = acl.util.start_thread(prof_data_read, [r, context])

# 8.执行模型。
ret = acl.mdl.execute(model_id, input, output)

# 9.处理模型推理结果。
# ......
```



```
# 10.释放描述模型输入/输出信息、内存等资源，卸载模型。
# ......

# 11.取消订阅，释放订阅相关资源。
ret = acl.prof.model_unsubscribe(model_id)
ret = acl.util.stop_thread(thr_id)
os.close(r)
ret = acl.prof.destroy_subscribe_config(subscribe_config)

# 12.释放运行管理资源。
# ......
```

7.12 溢出算子数据采集及分析

前提条件

使用ATC工具转换模型时，需在转换命令中增加“--status_check”参数，并将参数值设置为“1”，表示在编译算子时添加溢出检测逻辑。

关于ATC工具及其参数的详细说明，请参见《ATC工具使用指南》。

采集溢出算子信息

在调用调用acl.init接口初始化pyACL时，在JSON配置文件中增加溢出算子Dump配置。

JSON配置文件中的示例内容如下，示例中的“dump_path”以相对路径为例：

```
{
   "dump":{
      "dump_path":"output",
      "dump_debug":"on"
   }
}
```

当dump_path配置为相对路径时，您可以在“应用可执行文件的目录/{dump_path}”下查看导出的数据文件，针对每个溢出算子，会导出两个数据文件：

- 溢出算子的dump文件：命名规则如{op_type}.{op_name}.{taskid}.{stream_id}.{timestamp}，如果op_type、op_name出现了“.”、“/”、“\”、空格时，会转换为下划线表示。

  用户可通过该信息知道具体出现溢出错误的算子，并通过解析溢出算子的dump文件获取该算子的输入和输出信息。
- 算子溢出数据文件：命名规则如OpDebug.Node_Opdebug.{taskid}.{stream_id}.{timestamp}，其中taskid不是溢出算子的taskid，用户不需要关注taskid的实际含义。

  用户可通过解析算子溢出数据文件获取溢出相关信息，包括溢出算子所在的模型、AICore的status寄存器状态等。

解析溢出算子的 dump 文件

步骤1 请根据实际情况，将{op_type}.{op_name}.{taskid}.{stream_id}.{timestamp}上传到安装有Toolkit软件包的环境。

步骤2 进入解析脚本所在目录，例如Toolkit软件包安装目录为：/home/HwHiAiUser/Ascend/ascend-toolkit/latest。

```
cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/toolkit/tools/operator_cmp/compare
```


步骤3 执行msaccucmp.py脚本，转换dump文件为numpy文件。举例：

```
python3 msaccucmp.py convert -d /home/HwHiAiUser/dump -out /home/HwHiAiUser/dumptonumpy -v 2
```

说明

-d参数支持传入单个文件，对单个dump文件进行转换，也支持传入目录，对整个path下所有的dump文件进行转换。

步骤4 调用Python，转换numpy文件为txt文件。举例：

```
$ python3
>>> import numpy as np
>>> a = np.load("/home/HwHiAiUser/dumptonumpy/Pooling.pool1.1147.1589195081588018.output.0.npy")
>>> b = a.flatten()
>>> np.savetxt("/home/HwHiAiUser/dumptonumpy/Pooling.pool1.1147.1589195081588018.output.0.txt", b)
```

转换为.txt格式文件后，维度信息、Dtype均不存在。详细的使用方法请参考numpy官网介绍。

----结束

解析算子溢出数据文件

由于生成的溢出数据是二进制格式，可读性较差，需要通过工具将bin文件解析为用户可读性好的JSON文件。

步骤1 请根据实际情况，将溢出数据文件OpDebug.Node_Opdebug.{taskid}.{timestamp}上传到安装有Toolkit软件包的环境。

步骤2 进入解析脚本所在路径，例如Toolkit软件包安装目录为：/home/HwHiAiUser/Ascend/ascend-toolkit/latest。

```
cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/toolkit/tools/operator_cmp/compare
```

步骤3 执行解析命令，例如：

```
python3 msaccucmp.py convert -d /home/HwHiAiUser/opdebug/
Opdebug.Node_OpDebug.59.1597922031178434  -out /home/HwHiAiUser/result
```

关键参数：

- -d：溢出数据文件所在目录，包括文件名。
- -out：解析结果待存储目录，如果不指定，默认生成在当前目录下。

步骤4 解析结果文件内容如下所示。

```
{
   "DHA Atomic Add": {
      "model_id": 0,
      "stream_id": 0,
      "task_id": 0,
      "task_type": 0,
      "pc_start": "0x0",
      "para_base": "0x0",
      "status": 0
   },
   "L2 Atomic Add": {
      "model_id": 0,
      "stream_id": 0,
      "task_id": 0,
      "task_type": 0,
      "pc_start": "0x0",
      "para_base": "0x0",
      "status": 0
   },
   "AI Core": {
      "model_id": 514,
```



```
        "stream_id": 563,
        "task_id": 57,
        "task_type": 0,
        "pc_start": "0x1008005b0000",
        "para_base": "0x100800297000",
        "kernel_code": "0x1008005ae000",
        "block_idx": 1,
        "status": 32
    }
}
```

参数解释：

- model_id：标识溢出算子所在的模型id。
- stream_id：标识溢出算子所在的streamid。
- task_id：标识溢出算子的taskid。
- task_type：标识溢出算子的task类型。
- pc_start：标识溢出算子的代码程序的内存起始地址。
- para_base：标识溢出算子的参数的内存起始地址。
- kernel_code：标识溢出算子的代码程序的内存起始地址，和pc_start相同。
- block_idx：标识溢出算子的blockid参数。
- status：AICore的status寄存器状态，用户可以从status值分析得到具体溢出错误。status为10进制表示，需要转换成16进制，然后定位到具体错误。

  例如：status为272，转换成16进制为0x00000110，则可以判定出可能原因为0x00000010+0x00000100。
  - 0x00000008：符号整数最小负数NEG符号位取反溢出。
  - 0x00000010：整数加法、减法、乘法或乘加操作计算有溢出。
  - 0x00000020：浮点计算有溢出。
  - 0x00000080：浮点数转无符号数的输入是负数。
  - 0x00000100：FP32转FP16或32位符号整数转FP16中出现溢出。
  - 0x00000400：CUBE累加出现溢出。

----结束

7.13 特征向量检索

须知

Atlas 200/300/500 推理产品上，不支持该功能。

Atlas 200/500 A2推理产品上，不支持该功能。

Atlas 训练系列产品上，不支持该功能。

Atlas A2训练系列产品上，不支持该功能。

基本原理

该部分主要实现了对特征检索的功能验证，生成随机底库，随机生成特征数据进行特征检索（当前支持**1：N**、**M：N**两种检索模式，下文的示例代码以**1：N**为例）。大致

可分为初始化、添加特征到底库、底库搜索、精准修改或删除底库特征、去初始化几个主要步骤，具体接口调用方式如下：

- 初始化：调用acl.init接口进行初始化、运行管理资源申请，调用acl.fv.create_init_para接口创建aclfvInitPara类型的数据来指定特征向量检索的初始化参数。
- 添加特征到底库：主要调用acl.fv.create_feature_info接口创建aclfvFeatureInfo类型数据来表示创建特征的描述信息，然后调用acl.fv.repo_add添加底库。
- 底库搜索：调用acl.fv.search接口来实现检索。
- 精准修改或删除底库特征：调用acl.fv.delete和acl.fv.modify接口来实现删除或修改底库中某个特征。下文的代码以删除底库特征为例。
- 去初始化：主要包括释放运行管理资源、调用acl.fv.destroy_init_para接口销毁aclfvInitPara类型的数据、调用acl.fv.release接口特征检索模块去初始化，释放内存空间。

示例代码

调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。以下是关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
# 1.pyACL初始化。

# 2.申请运行管理资源，包括设置用于计算的Device、创建Context。

# 3.初始化。
# 3.1 初始化特征检索模块，此处以底库特征数100000为例。
fs_num = 100000
fv_init_para = acl.fv.create_init_para(fs_num)

# 3.2 指定特征向量检索的初始化参数。
ret = acl.fv.init(fv_init_para)

# 4.添加底库和特征向量。
# 4.1 增加第一个特征，创建特征描述信息时，偏移量offset参数值为0。
offset = 0
feature_count = 1000
feature_len = 36

# 创建内存。
feature_info_device, ret = acl.rt.malloc(base_data_len, ACL_MEM_MALLOC_NORMAL_ONLY)

# 此处的自定义函数BaseShortFeaAlloc用于生成特征随机数据，由用户自行实现。
feature_count = 1000
feature_type = SEARCH_1_N
feature_info_buffer = base_short_feature_data(feature_count, feature_type)


# 将随机特征数据转换为数组。
feature_info_list_arr = numpy.array(feature_info_list, dtype=numpy.uint8)
# 将随机特征数据转换为bytes对象，通过acl.util.bytes_to_ptr获取bytes对象的指针地址。
bytes_data = feature_info_list_arr.tobytes()
feature_info_ptr = acl.util.bytes_to_ptr(bytes_data)
# 将随机特征数据从Host侧拷贝到Device侧。
ret = acl.rt.memcpy(feature_info_device, feature_count * feature_len, feature_info_ptr, feature_count *
feature_len, ACL_MEMCPY_HOST_TO_DEVICE)


# 创建特征描述信息，inputData表示前一步的特征随机数据。
feature_info = acl.fv.create_feature_info(id0, id1, offset, feature_len, feature_count, feature_info_device,
feature_len * feature_count)

# 添加底库并向底库中添加特征，featureInfo表示前一步的特征描述信息。
ret = acl.fv.repo_add(SEARCH_1_N, feature_info)
```



```
# 销毁aclfvFeatureInfo特征描述信息。
ret = acl.fv.destroy_feature_info(feature_info)

# 4.2增加第二个特征，创建特征描述信息时，偏移值offset需要与库中已添加特征个数一致，并精确删除或修改底库中的某个特征。
offset += featureCount;

# 增加特征到底库的步骤，参考4.1中的代码。
# ....

feature_data_list = []
for i in range(36):
    feature_data_list.append(i)
# 将随机特征数据转换为数组。
feature_info_list_arr = numpy.array(feature_data_list, dtype=numpy.uint8)
# 将随机特征数据转换为bytes对象，通过acl.util.bytes_to_ptr获取bytes对象的指针地址。
bytes_data = feature_info_list_arr.tobytes()
feature_info_ptr = acl.util.bytes_to_ptr(bytes_data)
# 创建内存并传输特征数据。
feature_info_device, ret = acl.rt.malloc(36, ACL_MEM_MALLOC_NORMAL_ONLY)
kind = ACL_MEMCPY_DEVICE_TO_DEVICE

# 如果运行模式是ACL_HOST,将特征数据拷贝到Device侧，其中feature_len为feature_data_list数据集申请的内存长度。
ret = acl.rt.memcpy(feature_info_device, feature_count * feature_len, feature_info_ptr, feature_count * feature_len, ACL_MEMCPY_HOST_TO_DEVICE)

# 创建特征描述信息。
id0 = 0
id1 = 0
feature_info = acl.fv.create_feature_info(id0, id1, offset, 36, 1, feature_info_buffer, 36)

#  删除1个特征。
acl.fv.destroy_feature_info(feature_info)

# 4.3 增加特征到其它底库,其中一级底库为1，二级底库为1。
id0 = 1
id1 = 1
offset = 0

# 增加特征到底库步骤，参考4.1中的代码。
# ....

# 5 底库检索（以1：N检索为例），主要包括特征检索预处理，特征1:N检索，特征检索结果处理三个部分。
# 5.1 特征检索预处理，对于1:N来说, queryCnt必须为1。
query_cnt = 1
table_len = 32 * 1024
topK = 5
table_data_len = query_cnt * table_len

# 生成数据表，用户通过数据表进行检索比对，此处的自定义函数adc_table_init用于初始化特征检索输入Adc表，由用户自行实现。
table_data_tmp = adc_table_init(1000, query_cnt * 1024)

# 为数据表分配内存，table_data_dev用于创建检索输入表信息。
table_data,ret = acl.rt.malloc(table_data_len, ACL_MEM_MALLOC_NORMAL_ONLY)

# 为检索结果resultNumDev,id0Dev,id1Dev,resultOffsetDev,resultDistanceDev分配内存。
data_len = query_cnt * topk * type_size
result_num_data_len = query_cnt * type_size
result_num, ret = acl.rt.malloc(result_num_data_len, ACL_MEM_MALLOC_NORMAL_ONLY)
id_0, ret = acl.rt.malloc(data_len, ACL_MEM_MALLOC_NORMAL_ONLY)
id_1, ret = acl.rt.malloc(data_len, ACL_MEM_MALLOC_NORMAL_ONLY)
result_offert, ret = acl.rt.malloc(data_len, ACL_MEM_MALLOC_NORMAL_ONLY)
result_distance, ret = acl.rt.malloc(data_len, ACL_MEM_MALLOC_NORMAL_ONLY)

# 创建检索输入表信息，结果用于创建检索任务输入信息。
query_table = acl.fv.create_query_table(query_cnt, table_len, table_data, table_data_len)
```



```
# 创建特征库范围参数，结果用于创建检索任务输入信息。
repo_range = acl.fv.create_repo_range(0, 1023, 0, 1023)
# 创建检索任务输入信息，结果用于特征1：N检索。
search_input = acl.fv.create_search_input(query_table, repo_range, topK)

# 创建检索结果信息，结果用于特征1：N检索。
search_result = acl.fv.create_search_result(query_cnt, result_num, result_num_data_len, id_0, id_1,
result_offert, result_distance, data_len)

# 5.2 特征1：N检索。
ret = acl.fv.search(SEARCH_1_N, search_input, search_result)

# 6. 删除底库和数据。
# 创建特征库范围并删除指定范围内的底库。
id0Min = 0
id0Max = 1023
id1Min = 0
id1Max = 1023
repo_range = create_repo_range(id0Min, id0Max, id1Min, id1Max)
ret = acl.fv.repo_del(SEARCH_1_N, repo_range)

#  销毁aclfvInitPara类型的数据。
ret = acl.fv.destroy_init_para(fv_init_para)

# 7. 释放运行管理资源

# 8. pyACL去初始化
# ......
```


8 应用调试

运行应用

运行应用的步骤，请参考**基于Caffe ResNet-50网络实现图片分类（同步推理）**。

相关注意点如下：

1. 模型转换，详细说明请参见《ATC工具使用指南》。
2. 运行时，需将pyACL初始化配置文件（acl.json）所在的目录、测试图片所在的目录、*.om文件所在的目录都上传到Host的同一个目录下。

   如果在**pyACL初始化**阶段，在acl.init接口中不传入参数，则无需将pyACL初始化配置文件（acl.json）所在的目录上传到Host。
3. 运行代码时，直接运行对应的Python脚本即可。如：

```
python3 main.py
```

问题定位

运行应用时如果出错，您可以参见《日志参考》获取日志文件，以便查看日志文件中详细报错。根据报错初步定位后：

- 如果是接口约束导致接口调用逻辑不对，需查看总体的**B 使用约束**以及各接口本身的约束，再调整接口调用逻辑。
- 如果是算子在AI Core上运行报错，需要进一步定位算子报错的原因，可调用pyACL提供的接口，获取出错算子的描述信息，用于进一步分析时使用，可参见**7.10 AI Core异常信息获取**，查看原理及调用示例。
- 对于Atlas 200/300/500 推理产品，典型的案例及其解决方法请参见《故障处理》。

9 应用样例参考

9.1 获取更多样例（Atlas 200/300/500 推理产品）

当前pyACL提供的样例如下表所示。

表 9-1 Linux 操作系统场景下的 Sample 列表

Sample名称	Sample获取	运行指导	基本功能介绍
acl_operator_add	**获取样例**	**实现矩阵-矩阵相加运算**	实现了对自定义算子的功能验证，通过将自定义算子转换为单算子离线模型文件，然后通过pyACL加载单算子模型文件进行运行。
acl_dvpp_resnet50	**获取样例**	**基于Caffe ResNet-50网络实现图片分类（图片解码+缩放+同步推理）**	主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。具体请参见pyACL样例使用指导。

Sample名称	Sample获取	运行指导	基本功能介绍
acl_vdec_resnet50	获取样例	基于Caffe ResNet-50网络实现图片分类（视频解码+同步推理）	
acl_resnet_sync	获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	
acl_resnet_async	获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	
acl_vpc_jpege_resnet50	获取样例	9.5.2.3 基于Caffe ResNet-50网络实现图片分类（图片解码+抠图缩放+图片编码+同步推理）	该样例主要是从一张YUV420SP NV12格式的输入图片中按指定区域分别抠出一张或多张224 * 224子图（YUV420SP NV12)。
acl_venc	获取样例	媒体数据处理（视频编码）	将一张YUV420SP NV12格式的图片连续编码n次，生成一个H265格式的视频码流

9.2 获取更多样例（Atlas 推理系列产品（Ascend 310P 处理器））

当前pyACL提供的样例如下表所示。

表 9-2 Linux 操作系统场景下的 Sample 列表

Sample名称	Sample获取	运行指导	基本功能介绍
acl_operator_add	获取样例	实现矩阵-矩阵相加运算	实现了对自定义算子的功能验证，通过将自定义算子转换为单算子离线模型文件，然后通过pyACL加载单算子模型文件进行运行。
acl_dvpp_resnet50	获取样例	基于Caffe ResNet-50网络实现图片分类（图片解码+缩放+同步推理）	主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。具体请参见pyACL样例使用指导。
acl_vdec_resnet50	获取样例	基于Caffe ResNet-50网络实现图片分类（视频解码+同步推理）	
acl_resnet_sync	获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	
acl_resnet_async	获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	
acl_vpc_jpege_resnet50	获取样例	9.5.2.3 基于Caffe ResNet-50网络实现图片分类（图片解码+抠图缩放+图片编码+同步推理）	该样例主要是从一张YUV420SP NV12格式的输入图片中按指定区域分别抠出一张或多张224*224子图（YUV420SP NV12)。
acl_venc	获取样例	媒体数据处理（视频编码）	将一张YUV420SP NV12格式的图片连续编码n次，生成一个H265格式的视频码流

9.3 获取更多样例（Atlas 训练系列产品）

当前pyACL提供的样例如下表所示。

表 9-3 Linux 操作系统场景下的 Sample 列表

Sample名称	Sample获取	运行指导	基本功能介绍
acl_operator_add	获取样例	实现矩阵-矩阵相加运算	实现了对自定义算子的功能验证，通过将自定义算子转换为单算子离线模型文件，然后通过pyACL加载单算子模型文件进行运行。
acl_dvpp_resnet50	获取样例	基于Caffe ResNet-50网络实现图片分类（图片解码+缩放+同步推理）	主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。具体请参见pyACL样例使用指导。
acl_vdec_resnet50	获取样例	基于Caffe ResNet-50网络实现图片分类（视频解码+同步推理）	
acl_resnet_sync	获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	
acl_resnet_async	获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	
acl_vpc_jpege_resnet50	获取样例	9.5.2.3 基于Caffe ResNet-50网络实现图片分类（图片解码+抠图缩放+图片编码+同步推理）	该样例主要是从一张YUV420SP NV12格式的输入图片中按指定区域分别抠出一张或多张224*224子图（YUV420SP NV12）。

9.4 获取更多样例（Atlas A2 训练系列产品）

当前pyACL提供的样例如下表所示。

表 9-4 Linux 操作系统场景下的 Sample 列表

Sample名称	Sample获取	运行指导	基本功能介绍
acl_operator_add	获取样例	实现矩阵-矩阵相加运算	实现了对自定义算子的功能验证，通过将自定义算子转换为单算子离线模型文件，然后通过pyACL加载单算子模型文件进行运行。

9.5 样例使用指导

9.5.1 环境准备

9.5.1.1 Python 环境

前提条件

请参见表9-5完成环境准备。

表 9-5 安装软件

名称	版本	说明
Python	Python3.7.5	安装方式与Python依赖版本的限制请参见《CANN软件安装指南》。
	Python3.8.0 ~ Python3.8.10	
	Python3.9.2	
Python依赖	-	
Numpy	≥ 1.18.2	Python的一种开源的数值计算扩展。 使用**pip3 install numpy**安装Numpy。

名称	版本	说明
Pillow	7.2.0	Python的图像处理库（Python Imaging Library）。 使用**pip3 install Pillow**命令安装Pillow。

说明

本节所列的python、pip命令，实际命名与用户机器中软链接设置的命名一致，以对应python3.7.5版本为示例，请用户自行替换。

Atlas 500 智能小站配置说明

说明

- Atlas 500 智能小站出厂预安装的EulerOS未安装pip，需要参考以下步骤安装。
- 容器使用场景请参见《**Atlas 500智能小站用户指南**》中“制作和启动容器镜像”章节，请用户进入容器内自行安装Python3.7.5并参见以下步骤完成配置。

步骤1 在pip官网获取并上传**pip安装压缩包**，以pip-20.2.3.tar.gz为例。

步骤2 进入压缩文件包所在目录并解压压缩包：

```
tar -zvxf pip-20.2.3.tar.gz
```

步骤3 安装pip安装包。

依次执行如下命令，进入pip目录，编译并安装：

```
cd pip-20.2.3
python3 setup.py build
python3 setup.py install
```

安装完成后，执行如下命令查看pip版本：

```
pip -V
```

若出现如下提示，说明pip安装成功：

```
pip 20.2.3 from /usr/local/lib/python3.7/site-packages/pip-20.2.3-py3.7.egg/pip (python 3.7)
```

步骤4 执行以下命令安装numpy与pillow依赖，请注意版本限制，详见**表9-5**。

```
pip install numpy
pip install pillow
```

----结束

CentOS7 aarch64 环境下安装 Pillow 的说明

CentOS7 aarch64环境中通过**pip3.7 install Pillow**安装Pillow之后，在执行**from PIL import Image**的时候会报错：

```
ImportError: /usr/local/python3.7.5/lib/python3.7/site-packages/PIL/_imaging.cpython-37m-aarch64-linux-gnu.so: ELF load command alignment not page-aligned
```

这种情况下，需要用源码安装Pillow。若已经通过pip安装，需要运行**pip3.7 uninstall Pillow**进行卸载。

步骤1 在**链接**中获取Pillow的源码压缩包，以Pillow-7.2.0.tar.gz为例。

步骤2 检查下列Pillow的依赖是否安装，如果没有安装则进行安装。

```
sudo yum install python-devel
sudo yum install zlib-devel
sudo yum install libjpeg-turbo-devel
```

步骤3 解压Pillow源码压缩包并进行编译安装。

```
tar -xvf Pillow-7.2.0.tar.gz
cd Pillow-7.2.0/
python3.7 setup.py build
python3.7 setup.py install
```

----结束

9.5.1.2 准备环境

您需要部署开发环境和运行环境，请参见《CANN软件安装指南》。

- 本文以如下安装路径示例来说明操作步骤，实际操作前，**请务必**获取这些组件的实际安装路径，以便后续操作时使用：
  - 以**HwHiAiUser**用户安装Driver组件，Driver组件的默认安装路径为“${HOME}/Ascend”，在该路径下安装成功后，包括“driver”目录。
  - 以**HwHiAiUser**用户安装cann-toolkit软件包，cann-toolkit软件包的默认安装路径为“${HOME}/Ascend”，在该路径下安装成功后，包括“ascend-toolkit”目录。
  - 以**HwHiAiUser**用户安装cann-nnrt软件包，cann-nnrt软件包的默认安装路径为“${HOME}/Ascend”，在该路径下安装成功后，包括“nnrt”目录。
  - 以**HwHiAiUser**用户安装cann-nnae软件包，cann-nnae软件包的默认安装路径为“${HOME}/Ascend”，在该路径下安装成功后，包括“nnae”目录。
- pyACL在cann-toolkit软件包、cann-nnrt软件包和cann-nnae软件包中均有集成，用户可根据使用场景自行选择其中之一进行安装。cann-toolkit软件包为开发套件包，适用于开发环境。cann-nnrt软件包为离线推理引擎包、cann-nnae软件包为深度学习引擎包，适用于运行环境。
- 本文以**HwHiAiUser**用户作为开发环境、运行环境的运行用户为例来说明操作步骤，实际操作前，**请务必**获取开发环境、运行环境的运行用户，以便后续操作时使用。
- 用户使用**export**命令在当前终端窗口下声明环境变量，关闭Shell终端或切换用户时环境变量失效。

9.5.1.3 环境变量配置

在安装完CANN软件包之后，请务必自行配置pyACL相关的环境变量，否则，将无法正常“import acl”**。**

- 若环境中安装了cann-toolkit软件包：

```
# 以root用户安装toolkit包。
. /usr/local/Ascend/ascend-toolkit/set_env.sh
# 以非root用户安装toolkit包。
. ${HOME}/Ascend/ascend-toolkit/set_env.sh
```

- 若环境中安装了cann-nnrt软件包：

```
# 以root用户安装nnrt包。
. /usr/local/Ascend/nnrt/set_env.sh
# 以非root用户安装nnrt包。
. ${HOME}/Ascend/nnrt/set_env.sh
```


- 若环境中安装了cann-nnae软件包：

```
# 以root用户安装nnae包。
. /usr/local/Ascend/nnae/set_env.sh
# 以非root用户安装nnae包。
. ${HOME}/Ascend/nnae/set_env.sh
```

说明

Atlas 500 智能小站配置环境变量说明如下：

```
. /opt/ascend/nnrt/set_env.sh
```

（旧版本）Atlas 500 智能小站配置环境变量说明如下：

- 配置“PYTHONPATH”环境变量，将“/home/data/miniD/driver/lib64”加入到“PYTHONPATH”中。

  命令示例如下：

  ```
  export PYTHONPATH=/home/data/miniD/driver/lib64:$PYTHONPATH
  ```

- 配置“LD_LIBRARY_PATH”，将“/home/data/miniD/driver/lib64”加入到“LD_LIBRARY_PATH”中。

  命令示例如下：

  ```
  export LD_LIBRARY_PATH=/home/data/miniD/driver/lib64:$LD_LIBRARY_PATH
  ```

9.5.2 样例介绍

9.5.2.1 实现矩阵-矩阵相加运算

9.5.2.1.1 样例介绍

获取样例

单击Gitee，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level1_single_api/1_acl/4_blas/acl_operator_add”目录下获取acl_operator_add样例。

功能描述

此样例实现了对自定义算子的功能验证，通过将自定义算子转换为单算子离线模型文件，然后通过ACL加载单算子模型文件进行运行。

该实现矩阵-矩阵相加的运算示例为：C = A + B，其中A、B、C都是8*16的矩阵，类型为int32，矩阵加的结果是一个8 * 16的矩阵。

主要接口

主要接口如表9-6所示。

表 9-6 主要接口介绍

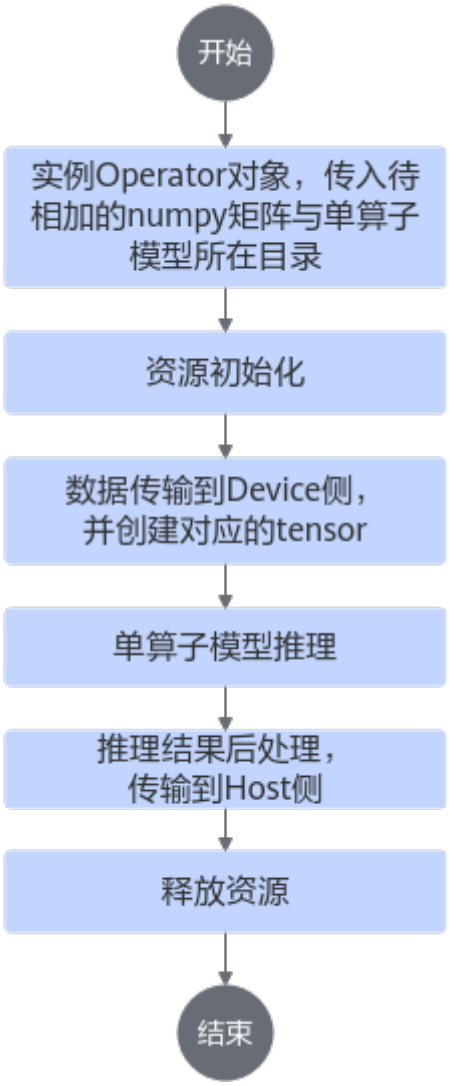
功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。

功能	对应ACL模块	ACL 接口函数	功能说明
	Context管理	acl.rt.create_context	创建Context。
	Stream管理	acl.rt.create_stream	创建Stream。
	算子加载与执行	acl.op.set_model_dir	加载模型文件的目录。
数据后处理	算子加载与执行	acl.op.create_attr	创建aclopAttr类型的数据。
	--	acl.create_tensor_desc	创建aclTensorDesc类型的数。
	--	acl.get_tensor_desc_size	获取Tensor描述占用的空间大小。
	--	acl.create_data_buffer	创建aclDataBuffer类型的数据。
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.rt.malloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。
单算子推理	算子加载与执行	acl.op.execute	异步加载并执行指定的算子。
公共模块	--	acl.util.ptr_to_numpy	通过指针地址获取numpy.ndarray对象。
	--	acl.util.numpy_to_ptr	获取numpy.ndarray对象的内存数据的指针地址。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。
	Stream管理	acl.rt.destroy_stream	销毁Stream。
	Context管理	acl.rt.destroy_context	销毁Context。
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

单算子矩阵相加流程图流程图

单算子矩阵相加流程图流程图如图9-1所示。

图 9-1 单算子矩阵相加流程图



目录结构

如下为模型文件转换后的示例目录结构，“op_models”文件夹是转换后生成的。

```
acl_operator_add
├──src
│   ├── acl_execute_add.py //运行文件。
│   └── constant.py //常量定义。
└── test_data
    ├── config
    │   ├── acl.json //系统初始化的配置文件。
    │   └── add_op.json //矩阵相加算子的描述信息。
    └── op_models
        └── 0_Add_3_2_8_16_3_2_8_16_3_2_8_16.om //矩阵相加算子的模型文件。
```


9.5.2.1.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品
- Atlas A2训练系列产品

模型转换

步骤1 以**HwHiAiUser**（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 在“acl_operator_add/test_data”路径下执行如下命令，生成单算子模型文件。*Ascendxxx*为使用的昇腾AI处理器的版本，请用户自行替换。

```
atc --singleop=config/add_op.json --soc_version=Ascendxxx --output=op_models
```

说明

- “--output”参数：生成的*.om文件存放在“./op_models”目录下。
- 使用**atc**命令时用户需保证对“acl_operator_add/test_data”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

模型转换成功后，会生成如下文件：

在当前目录的“op_model”目录下生成单算子的模型文件“0_Add_3_2_8_16_3_2_8_16_3_2_8_16.om”。

说明

算子模型文件的命名规范为：序号 + opType + 输入的描述(dateType_format_shape) + 输出的描述。

步骤4 以**HwHiAiUser**（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤3 在“acl_operator_add/test_data”路径下执行如下命令：

```
python3 ../src/acl_execute_add.py
```


步骤4 执行成功后，在屏幕上的关键提示信息示例如下。

```
factor_a:
[[85 19 20 27 74 27 47 73 69 82 81 78 25 45  1 84]
 [46 52 93 42  7 69 38 68 68 57 58 64 53 14 55 95]
 [35 87 81 50  5  8 97 58 14 18 92 91 97 86 91 11]
 [35 71 23  0 74 66  2 80 74 44 20 53 70 67 67 56]
 [ 6 52  5 66  9 86  8 14 60 65 16 80 82 56 90 59]
 [50 24 79 53 13 94  6 46 46 68 39 76 56 56  9 28]
 [92  7 67 43 39 20  2 52 17  5 46  0  6 98 31 25]
 [ 8 37 45 29 23 19 35 25 90 71 70 15 90 55 15 29]]
factor_b:
[[29  9 73 35 34  5 52 22 74 84 11 85 73 68 70 59]
 [57 73 78 90 75 46 28  4 69 81  2 15  9 21 76  7]
 [90 59 81 57 80 33 92 98 15 86 98 22  9 25 35 70]
 [94 54 15 65  2 62 32 55 32 91  3 38 96 51 24 86]
 [96 84 33 32 86 61 24 66 47 80 76  4 86 29 39 20]
 [48 97 68  4 42 19 60 43 72  0 68 65 80 43 38 54]
 [83 10 54 90 60 35 84 22 22  5 70 67 79 78 45 41]
 [51 19 66 74  4 72 54 23 32 42  9 49 93 52 50 83]]
init resource stage:
init resource success
gen input data stage:
gen input data success
gen output data stage:
gen output data success
execute stage:
execute success
get operator result stage:
shape: (8, 16)
ACL output:
 [[114  28  93  62 108  32  99  95 143 166  92 163  98 113  71 143]
 [103 125 171 132  82 115  66  72 137 138  60  79  62  35 131 102]
 [125 146 162 107  85  41 189 156  29 104 190 113 106 111 126  81]
 [129 125  38  65  76 128  34 135 106 135  23  91 166 118  91 142]
 [102 136  38  98  95 147  32  80 107 145  92  84 168  85 129  79]
 [ 98 121 147  57  55 113  66  89 118  68 107 141 136  99  47  82]
 [175  17 121 133  99  55  86  74  39  10 116  67  85 176  76  66]
 [ 59  56 111 103  27  91  89  48 122 113  79  64 183 107  65 112]]
get operator result success
release source stage:
release source success
```

----结束

9.5.2.2 基于 Caffe ResNet-50 网络实现图片分类（图片解码+缩放+同步推理）

9.5.2.2.1 样例介绍

获取样例

单击**Gitee**，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/1_classification/vpc_resnet50_imagenet_classification”目录下获取vpc_resnet50_imagenet_classification样例。

功能描述

该样例主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。

将Caffe ResNet-50网络的模型文件转换为适配昇腾AI处理器的离线模型（*.om文件），在样例中，加载该om文件，对2张*.jpg图片进行解码、缩放、推理，分别得到推理结果后，再对推理结果进行处理，输出最大置信度的类别标识。

转换模型时，需配置色域转换参数，用于将YUV420SP格式的图片转换为RGB格式的图片，才能符合模型的输入要求。

主要接口

主要接口如表9-7所示。

表 9-7 主要接口介绍

功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。
	Context管理	acl.rt.create_context	创建Context。
	Stream管理	acl.rt.create_stream	创建Stream。
	算子加载与执行	acl.op.set_model_dir	加载模型文件的目录。
模型初始化	模型加载与执行	acl.mdl.load_from_file	从*.om文件加载模型到device侧。
	数据类型及操作接口	acl.mdl.create_desc	创建模型描述数据类型。
	数据类型及操作接口	acl.mdl.get_desc	获取模型描述数据类型。
数据预处理	媒体数据模块	acl.media.dvpp_jpeg_decode_async	图形解码接口。
	数据类型及操作接口	acl.media.dvpp_vpc_resize_async	将输入图片缩放到输出图片大小。
	数据类型及操作接口	acl.media.dvpp_set_pic_desc系列接口	设置图片描述相关参数。
模型推理	模型加载与执行	acl.mdl.execute	执行模型同步推理。
数据后处理	数据类型及操作接口	acl.op.create_attr	创建aclopAttr类型的数据。
	数据类型及操作接口	acl.create_tensor_desc	创建aclTensorDesc类型的数。

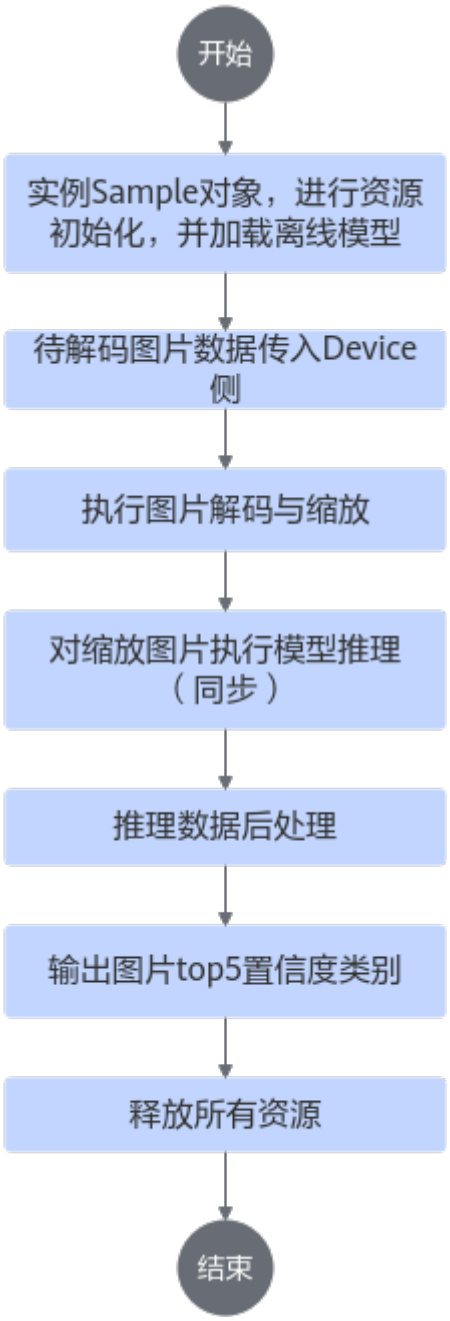
功能	对应ACL模块	ACL 接口函数	功能说明
	数据类型及操作接口	acl.get_tensor_desc_size	获取Tensor描述占用的空间大小。
	数据类型及操作接口	acl.create_data_buffer	创建aclDataBuffer类型的数据。
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.media.dvpp_malloc	分配内存给Device侧媒体数据处理时使用。
	内存管理	acl.rt.malloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。
单算子推理	算子加载与执行	acl.op.execute	异步加载并执行指定的算子。
公共模块	--	acl.util.ptr_to_numpy	通过指针地址获取numpy.ndarray对象。
	--	acl.util.numpy_to_ptr	获取numpy.ndarray对象的内存数据的指针地址。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.media.dvpp_free	通过acl.media.dvpp_malloc接口申请的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。
	模型加载与执行	acl.mdl.unload	卸载模型。
	Stream管理	acl.rt.destroy_stream	销毁Stream。
	Context管理	acl.rt.destroy_context	销毁Context。

功能	对应ACL模块	ACL 接口函数	功能说明
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

图片解码缩放流程图

图片解码缩放流程图如图9-2所示。

图 9-2 图片解码缩放流程图



目录结构

目录结构如下所示。

```
vpc_resnet50_imagenet_classification
├──src
│   ├── acl_dvpp.py //图片缩放实现文件。
│   ├── acl_model.py //模型推理实现文件。
│   ├── acl_op.py //单算子精度转换实现文件。
│   ├── acl_sample.py //运行文件。
│   ├── acl_util.py //工具类函数实现文件。
│   └── constant.py //常量定义。
├── data //测试数据。
│   ├── dog1_1024_683.jpg
```



```
└── dog2_1024_683.jpg
├── caffe_model
│   ├── aipp.cfg
│   ├── resnet50.caffemodel //resnet50模型。
│   └── resnet50.prototxt // resnet50模型的网络文件。
├── op_models
│   ├── 0_Cast_0_2_1000_1_2_1000.om //精度转换自定义算子。
│   ├── 1_ArgMaxD_1_2_1000_3_2_1.om //精度转换自定义算子。
│   └── op_list.json //精度转换算子配置文件。
└── model
    └── resnet50_aipp.om //推理模型。
```

9.5.2.2.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品

模型转换

步骤1 以HwHiAiUser（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 准备数据。

从以下链接获取ResNet-50网络的权重文件（*.caffemodel）、模型文件（resnet50.prototxt），并以HwHiAiUser（运行用户）将获取的文件上传至开发环境的“vpc_resnet50_imagenet_classification样例目录/caffe_model”目录下。

- 从gitee上获取：单击**Link**，查看README.md，查找获取原始模型的链接。
- 从GitHub上获取：单击**Link**，查看README.md，查找获取原始模型的链接。

步骤4 将ResNet-50网络转换为适配昇腾AI处理器的离线模型（*.om文件），转换模型时，需配置色域转换参数，用于将YUV420SP格式的图片转换为RGB格式的图片。

切换到“vpc_resnet50_imagenet_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --model=caffe_model/resnet50.prototxt --weight=caffe_model/resnet50.caffemodel --framework=0 --output=model/resnet50_aipp --soc_version=Ascendxxx --insert_op_conf=caffe_model/aipp.cfg
```

说明

- “--output”参数：生成的“resnet50_aipp.om”文件存放在“vpc_resnet50_imagenet_classification/model”目录下。
- 使用**atc**命令时用户需保证对“vpc_resnet50_imagenet_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤5 将Cast和ArgMaxD两个算子的算子描述信息（*.json文件）编译成适配昇腾AI处理器的离线模型（*.om文件），用于验证单算子的运行。

切换到“vpc_resnet50_imagenet_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --singleop=op_models/op_list.json --soc_version=Ascendxxx --output=op_models/
```

说明

- “--output”参数：生成的om文件必须放在“vpc_resnet50_imagenet_classification/op_models”目录下。
- 使用**atc**命令时用户需保证对“vpc_resnet50_imagenet_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

关于各参数的详细解释，请参见《ATC工具使用指南》中的参数说明。

步骤6 以HwHiAiUser（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 准备测试数据。 请从以下链接获取该样例的输入图片，并以运行用户将获取的文件上传至开发环境的“vpc_resnet50_imagenet_classification/data”目录下。如果目录不存在，需自行创建。

- **测试图片1**
- **测试图片2**

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“vpc_resnet50_imagenet_classification”路径下执行如下命令。

```
python3 ./src/acl_sample.py
```

步骤5 执行成功后，在屏幕上的关键提示信息示例如下。

```
Using device id:0
model path:./model/resnet50_aipp.om
images path:./data
[Sample] init resource stage:
[Sample] init resource stage success
[Model] The class Model initializes resources:
[Model] create output dataset:
[Model] create output dataset success
[Model] The class Model initializes resources successfully.
[Sample] width:1024 height:683
[Sample] image:./data/dog1_1024_683.jpg
[Dvpp] vpc decode stage:
[Dvpp] vpc decode stage success
[Dvpp] vpc resize stage:
[Dvpp] vpc resize stage success
[Model] create model input dataset:
[Model] create model input dataset success
[Model] execute stage:
[Model] execute stage success

======== top5 inference results: ========
label:161  prob: 0.712891
label:162  prob: 0.147095
label:167  prob: 0.051636
label:163  prob: 0.050476
label:166  prob: 0.030136
```



```
[SingleOP] single op cast success
[SingleOp] get top 1 label success
[SingleOP][ArgMaxOp] label of classification result is:161
[Sample] width:1024 height:683
[Sample] image:./data/dog2_1024_683.jpg
[Dvpp] vpc decode stage:
[Dvpp] vpc decode stage success
[Dvpp] vpc resize stage:
[Dvpp] vpc resize stage success
[Model] create model input dataset:
[Model] create model input dataset success
[Model] execute stage:
[Model] execute stage success

======== top5 inference results: ========
label:267  prob: 0.855469
label:266  prob: 0.049805
label:219  prob: 0.032654
label:265  prob: 0.013405
label:129  prob: 0.011024
[SingleOP] single op cast success
[SingleOp] get top 1 label success
[SingleOP][ArgMaxOp] label of classification result is:267
[Model] The class Model releases resources successfully.
[Dvpp] class Dvpp exit success
[SingOp] class SingOp release source success
[Sample] class Samle release source success
```

----结束

9.5.2.3 基于 Caffe ResNet-50 网络实现图片分类（图片解码+抠图缩放+图片编码+同步推理）

9.5.2.3.1 样例介绍

获取样例

单击**Gitee**，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/1_classification/vpc_jpeg_resnet50_imagenet_classification”目录下获取vpc_jpeg_resnet50_imagenet_classification样例。

功能描述

该样例主要是基于Caffe ResNet-50网络（单输入、单Batch）实现图片分类的功能。

根据运行应用的入参，该样例可实现以下功能：

- 将*.jpg格式的解码成YUV420SP NV12格式的输入图片，缩放，再进行模型推理，得到推理结果后，处理推理结果，输出top5置信度。
- 将YUV420SP NV12格式的输入图片，抠图，再进行模型推理，得到推理结果后，处理推理结果，输出top5置信度。
- 将YUV420SP NV12格式的输入图片，抠图贴图，再进行模型推理，得到推理结果后，处理推理结果，输出top5置信度。
- 将YUV420SP NV12格式的输入图片，编码为jpg格式的图片。
- 将YUV420SP NV12格式的输入图片（分辨率8192*8192）缩放，得到4000*4000。

- 将YUV420SP NV12格式的输入图片，进行批量抠图。
- 将YUV420SP NV12格式的输入图片，进行批量抠图粘贴。

主要接口

主要接口如表9-8所示。

表 9-8 主要接口介绍

功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。
	Context管理	acl.rt.create_context	创建Context。
	Stream管理	acl.rt.create_stream	创建Stream。
	算子加载与执行	acl.op.set_model_dir	加载模型文件的目录。
模型初始化	模型加载与执行	acl.mdl.load_from_file	从*.om文件加载模型到device侧。
	数据类型及操作接口	acl.mdl.create_desc	创建模型描述数据类型。
	数据类型及操作接口	acl.mdl.get_desc	获取模型描述数据类型。
数据预处理	媒体数据模块-JPEG	acl.media.dvpp_jpeg_decode_async	图形解码。
		acl.media.dvpp_jpeg_encode_async	图形编码。
	媒体数据模块-VPC	acl.media.dvpp_vpc_crop_async	抠一张子图。
		acl.media.dvpp_vpc_resize_async	将输入图片缩放到输出图片大小。
		acl.media.dvpp_vpc_crop_and_paste_async	抠图并粘贴到输出图片。
		acl.media.dvpp_vpc_batch_crop_async	抠多张子图。
		acl.media.dvpp_vpc_batch_crop_and_paste_async	抠多张子图并粘贴到输出图片。
	数据类型及操作接口	acl.media.dvpp_create_roi_config	创建图片框区域配置。

功能	对应ACL模块	ACL 接口函数	功能说明
		acl.media.dvpp_set_roi_config系列接口	设置图片框区域配置参数。
		acl.media.dvpp_set_pic_desc系列接口	设置图片描述相关参数。
模型推理	模型加载与执行	acl.mdl.execute	执行模型同步推理。
数据后处理	数据类型及操作接口	acl.op.create_attr	创建aclopAttr类型的数据。
		acl.create_tensor_desc	创建aclTensorDesc类型的数据。
		acl.get_tensor_desc_size	获取Tensor描述占用的空间大小。
		acl.create_data_buffer	创建aclDataBuffer类型的数据。
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.media.dvpp_malloc	分配内存给Device侧媒体数据处理时使用。
	内存管理	acl.rt.maclloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。
单算子推理	算子加载与执行	acl.op.execute	异步加载并执行指定的算子。
公共模块	--	acl.util.ptr_to_numpy	通过指针地址获取numpy.ndarray对象。
	--	acl.util.numpy_to_ptr	获取numpy.ndarray对象的内存数据的指针地址。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.media.dvpp_free	通过acl.media.dvpp_malloc接口申请的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。
	模型加载与执行	acl.mdl.unload	卸载模型。
	Stream管理	acl.rt.destroy_stream	销毁Stream。
	Context管理	acl.rt.destroy_context	销毁Context。
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

目录结构

目录结构如下所示。

```
resnet50_imagenet_classification
├──src
│  ├── acl_dvpp.py //图片缩放实现文件。
│  ├── acl_model.py //模型推理实现文件。
│  ├── acl_sample.py //运行文件。
│  ├── acl_util.py //工具类函数实现文件。
│  ├── acl_vdec.py //视频解码实现文件。
│  └── constant.py //常量定义。
├── data
│  └── vdec_h265_1frame_rabbit_1280x720.h265 //用户待处理的视频文件，由用户自行获取。
├── caffe_model
│  ├── aipp.cfg
│  ├── resnet50.caffemodel //ResNet-50模型。
│  └── resnet50.prototxt // ResNet-50模型的网络文件。
└── model
   └── resnet50_aipp.om //推理模型。
```

9.5.2.3.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品

模型转换

步骤1 以HwHiAiUser（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 准备数据。

从以下链接获取ResNet-50网络的权重文件（*.caffemodel）、模型文件（resnet50.prototxt），并以HwHiAiUser（运行用户）将获取的文件上传至开发环境的“vpc_jpeg_resnet50_imagenet_classification样例目录/caffe_model ”目录下。

- 从gitee上获取：单击**Link**，查看README.md，查找获取原始模型的链接。
- 从GitHub上获取：单击**Link**，查看README.md，查找获取原始模型的链接。

步骤4 将ResNet-50网络转换为适配昇腾AI处理器的离线模型（*.om文件），转换模型时，需配置色域转换参数，用于将YUV420SP格式的图片转换为RGB格式的图片。

切换到“vpc_jpeg_resnet50_imagenet_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --model=caffe_model/resnet50.prototxt --weight=caffe_model/resnet50.caffemodel --framework=0 --output=model/resnet50_aipp --soc_version=Ascendxxx --insert_op_conf=caffe_model/aipp.cfg
```


说明

- “--output”参数：生成的“resnet50_aipp.om”文件存放在“vpc_jpeg_resnet50_imagenet_classification/model”目录下。
- 使用**atc**命令时用户需保证对“vpc_jpeg_resnet50_imagenet_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤5 以HwHiAiUser（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 准备输入图片。 请从以下链接获取该样例的输入图片，并以运行用户将获取的文件上传至开发环境的“vpc_jpeg_resnet50_imagenet_classification/data”目录下。如果目录不存在，需自行创建。

- **dvpp_vpc_8192x8192_nv12.yuv**
- **persian_cat_1024_1536_283.jpg**
- **wood_rabbit_1024_1061_330.jpg**
- **wood_rabbit_1024_1068_nv12.yuv**

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“vpc_jpeg_resnet50_imagenet_classification”路径下请用户根据场景选择对应的命令执行：

- 执行解码+缩放+推理：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1061_330.jpg" --dvpp_type=0 --image_type='jpg'
  ```
- 执行抠图：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1068_nv12.yuv" --dvpp_type=1 --image_type='yuv'
  ```
- 执行抠图粘贴：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1068_nv12.yuv" --dvpp_type=2 --image_type='yuv'
  ```
- 执行编码：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1068_nv12.yuv" --dvpp_type=3 --image_type='yuv'
  ```
- 8k缩放：

  将“data”路径下“dvpp_vpc_8192x8192_nv12.yuv”文件名修改为“dvpp_vpc_8192_8192_nv12.yuv”。
  ```
  python3 ./src/main.py --images_path="./data/dvpp_vpc_8192_8192_nv12.yuv" --dvpp_type=4 --image_type='yuv'
  ```
- 执行批量抠图：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1068_nv12.yuv" --dvpp_type=5 --image_type='yuv' --in_batch_size=1 --out_batch_size=8
  ```
- 执行批量抠图粘贴：
  ```
  python3 ./src/main.py --images_path="./data/wood_rabbit_1024_1068_nv12.yuv" --dvpp_type=6 --image_type='yuv' --in_batch_size=1 --out_batch_size=8
  ```


步骤5 执行成功后，在屏幕上的关键提示信息示例如下（以执行批量抠图粘贴为例）。

```
Using device id:0
model path:./model/resnet50_aipp.om
images path:./data/wood_rabbit_1024_1068_nv12.yuv
result path:./vpc_out
dvpp type:6

[Sample] init resource stage:
[Sample] init resource stage success
[Model] class Model init resource stage:
[Model] create model output dataset:
[Model] create model output dataset success
[Model] class Model init resource stage success
[Dvpp] class Dvpp init resource stage:
[Dvpp] class Dvpp init resource stage:
[Sample] image:./data/wood_rabbit_1024_1068_nv12.yuv res_path:./vpc_out
[Sample] width:1024 height:1068
[Sample] in_batch_size:1 in_batch_size:8
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_0_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_1_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_2_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_3_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_4_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_5_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_6_.yuv success
[Sample]write out to file ./vpc_out/wood_rabbit_1024_1068_nv12_6crop_7_.yuv success
[Model] class Model release source success
[Dvpp] class Dvpp release source
[Dvpp] class Dvpp release source success
[Sample] class Sample release source success
```

----结束

9.5.2.4 基于 Caffe ResNet-50 网络实现图片分类（视频解码+同步推理）

9.5.2.4.1 样例介绍

获取样例

单击**Gitee**，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/1_classification/vdec_resnet50_classification”目录下获取vdec_resnet50_classification样例。

功能描述

该样例主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。

将Caffe ResNet-50网络的模型文件转换为适配昇腾AI处理器的离线模型（*.om文件）。该样例中加载该om文件，将1个h265格式（*.h265）的视频码流（仅包含一帧）循环10次解码出10张YUV420SP NV12格式的图片，对该10张图片做缩放，并对缩放的图片进行推理，分别得到推理结果后，再对推理结果进行处理，输出最大置信度的类别标识以及top5置信度的总和。

转换模型时，需配置色域转换参数，用于将YUV420SP格式的图片转换为RGB格式的图片，才能符合模型的输入要求。

主要接口

主要接口如**表9-9**所示。

表 9-9 主要接口介绍

功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。
	Context管理	acl.rt.create_context	创建Context。
	Stream管理	acl.rt.create_stream	创建Stream。
模型初始化	模型加载与执行	acl.mdl.load_from_file	从*.om文件加载模型到device侧。
	数据类型及操作接口	acl.mdl.create_desc	创建模型描述数据类型。
	数据类型及操作接口	acl.mdl.get_desc	获取模型描述数据类型。
数据预处理	媒体数据模块	acl.media.vdec_send_frame	视频解码接口。
	数据类型及操作接口	acl.media.vdec_set_channel_desc系列接口	设置视频处理通道描述信息。
	数据类型及操作接口	acl.media.dvpp_vpc_resize_async	将输入图片缩放到输出图片大小。
	数据类型及操作接口	acl.media.dvpp_set_pic_desc系列接口	设置图片描述相关参数。
模型推理	模型加载与执行	acl.mdl.execute	执行模型同步推理。
数据后处理	数据类型及操作接口	acl.op.create_attr	创建aclopAttr类型的数据。
	数据类型及操作接口	acl.create_tensor_desc	创建aclTensorDesc类型的数。

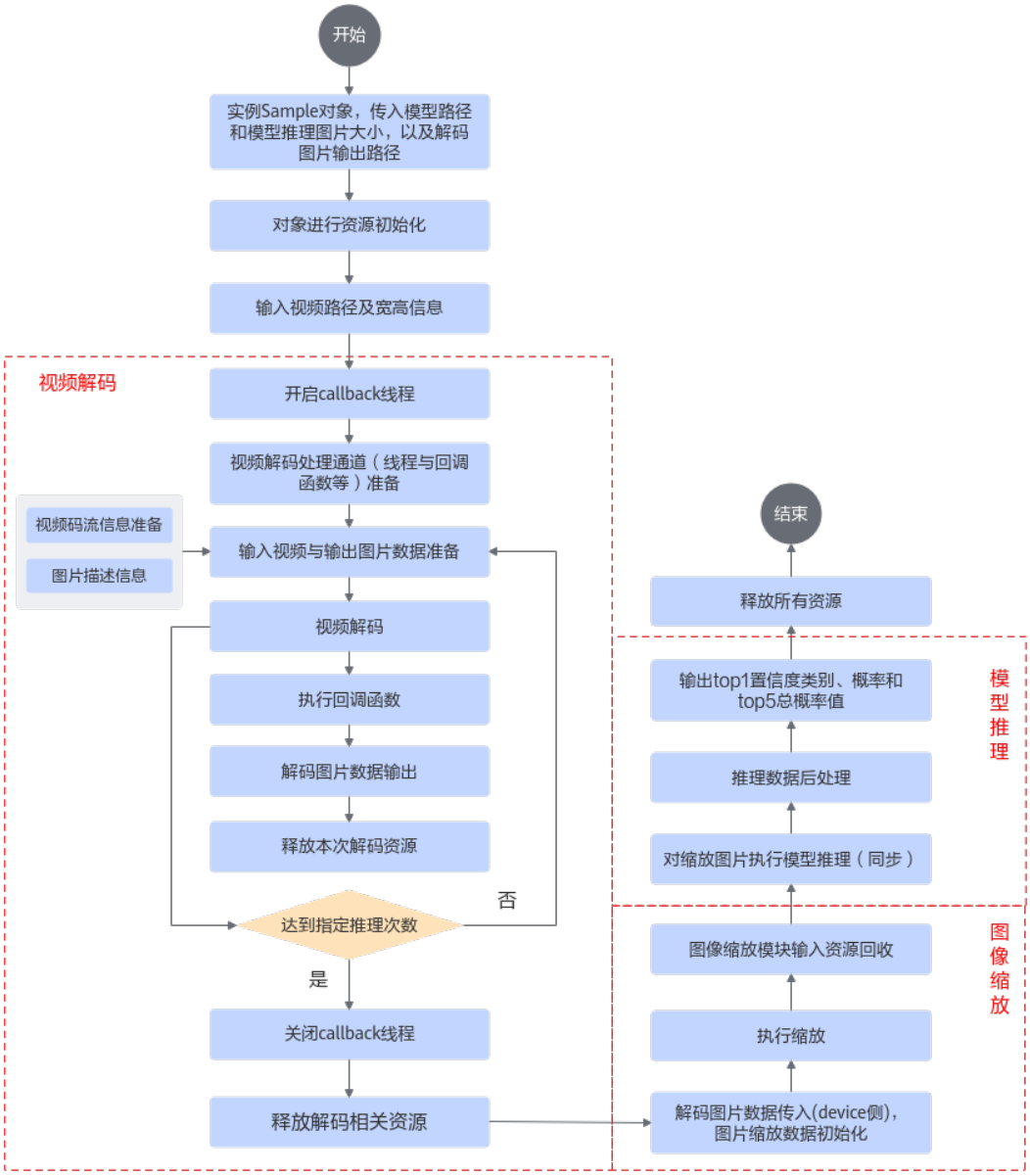
功能	对应ACL模块	ACL 接口函数	功能说明
	数据类型及操作接口	acl.get_tensor_desc_size	获取Tensor描述占用的空间大小。
	数据类型及操作接口	acl.create_data_buffer	创建aclDataBuffer类型的数据。
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.media.dvpp_malloc	分配内存给Device侧媒体数据处理时使用。
	内存管理	acl.rt.maclloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。
单算子推理	算子加载与执行	acl.op.execute	异步加载并执行指定的算子。
公共模块	--	acl.util.ptr_to_numpy	通过指针地址获取numpy.ndarray对象。
	--	acl.util.numpy_to_ptr	获取numpy.ndarray对象的内存数据的指针地址。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.media.dvpp_free	通过acl.media.dvpp_malloc接口申请的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。

功能	对应ACL模块	ACL 接口函数	功能说明
	模型加载与执行	acl.mdl.unload	卸载模型。
	Stream管理	acl.rt.destroy_stream	销毁Stream。
	Context管理	acl.rt.destroy_context	销毁Context。
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

视频解码及模型推理流程图

视频解码及模型推理流程图如图9-3所示。

图 9-3 视频解码及模型推理流程图



目录结构

目录结构如下所示。

```
vdec_resnet50_classification
├──src
│   ├── acl_dvpp.py //图片缩放实现文件。
│   ├── acl_model.py //模型推理实现文件。
│   ├── acl_sample.py //运行文件。
│   ├── acl_util.py //工具类函数实现文件。
│   ├── acl_vdec.py //视频解码实现文件。
│   └── constant.py //常量定义。
├── data
│   └── vdec_h265_1frame_rabbit_1280x720.h265 //用户待处理的视频文件，由用户自行获取。
├── caffe_model
│   ├── aipp.cfg
│   ├── resnet50.caffemodel //ResNet-50模型。
│   └── resnet50.prototxt // ResNet-50模型的网络文件。
```



```
└── model
    └── resnet50_aipp.om //推理模型。
```

9.5.2.4.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品

模型转换

步骤1 以HwHiAiUser（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 准备数据。

从以下链接获取ResNet-50网络的权重文件（*.caffemodel）、模型文件（resnet50.prototxt），并以HwHiAiUser（运行用户）将获取的文件上传至开发环境的“vdec_resnet50_classification/caffe_model”目录下。

- 从gitee上获取：单击**Link**，查看README.md，查找获取原始模型的链接。
- 从GitHub上获取：单击**Link**，查看README.md，查找获取原始模型的链接。

步骤4 将ResNet-50网络转换为适配昇腾AI处理器的离线模型（*.om文件），转换模型时，需配置色域转换参数，用于将YUV420SP格式的图片转换为RGB格式的图片。

切换到“vdec_resnet50_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --model=caffe_model/resnet50.prototxt --weight=caffe_model/resnet50.caffemodel --framework=0 --output=model/resnet50_aipp --soc_version=Ascendxxx --insert_op_conf=caffe_model/aipp.cfg
```

说明

- “--output”参数：生成的“resnet50_aipp.om”文件存放在“vdec_resnet50_classification/model”目录下。
- 使用**atc**命令时用户需保证对“vdec_resnet50_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤5 以HwHiAiUser（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 准备输入视频码流。

通过链接**vdec_h265_1frame_rabbit_1280x720.h265**获取输入视频码流文件“vdec_h265_1frame_rabbit_1280x720.h265”，并以HwHiAiUser（运行用户）上传至开发环境的“vdec_resnet50_classification/data”目录下。

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“vdec_resnet50_classification”路径下执行如下命令：

```
python3 ./src/acl_sample.py
```

步骤5 执行成功后，在屏幕上的关键提示信息示例如下（因打印信息过长，省略部分打印信息）。

```
init resource stage:
init resource stage success
[Model] class Model init resource stage:
[Model] create model output dataset:
[Model] create model output dataset success
[Model] class Model init resource stage success
[Dvpp] class Dvpp init resource stage:
[Dvpp] class Dvpp init resource stage success
[Vdec] class Vdec init resource stage:
[Vdec] class Vdec init resource stage success
[Vdec] forward index:0
[Vdec] create input stream desc success
[Vdec] create output pic desc success
[Vdec] vdec_send_frame stage success
......
[Vdec] [_callback] _callback exit success
[Vdec] [_thread_func] _thread_func out
[Vdec] vdec finish!!!

[Dvpp] vpc resize stage:
[Dvpp] vpc resize stage success
[Model] create model input dataset:
[Model] create model input dataset success
[Model] execute stage:
[Model] execute stage success

========== top5 inference results: ==========
label:331  prob: 0.910156
label:330  prob: 0.078308
label:104  prob: 0.009209
label:332  prob: 0.003283
label:350  prob: 0.000005
result: class_label[331],top1[0.910156],top5[1.000000]
......
[Sample] release source stage:
[Dvpp] class Dvpp release source success
[Model] class Model release source success
[Vdec] release resource:
[Vdec] release resource success
[Sample] release source stage success
```

----结束

9.5.2.5 基于 Caffe ResNet-50 网络实现图片分类（同步推理）

9.5.2.5.1 样例介绍

获取样例

单击Gitee，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/1_classification/resnet50_imagenet_classification”目录下获取resnet50_imagenet_classification样例

功能描述

该样例主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。

将Caffe ResNet-50网络的模型文件转换为适配昇腾AI处理器的离线模型（*.om文件），在样例中，加载该om文件，对2张jpg（*.jpg）图片进行同步推理，分别得到推理结果后，再对推理结果进行处理，输出top5置信度的类别标识。

主要接口

主要接口如表9-10所示。

表 9-10 主要接口介绍

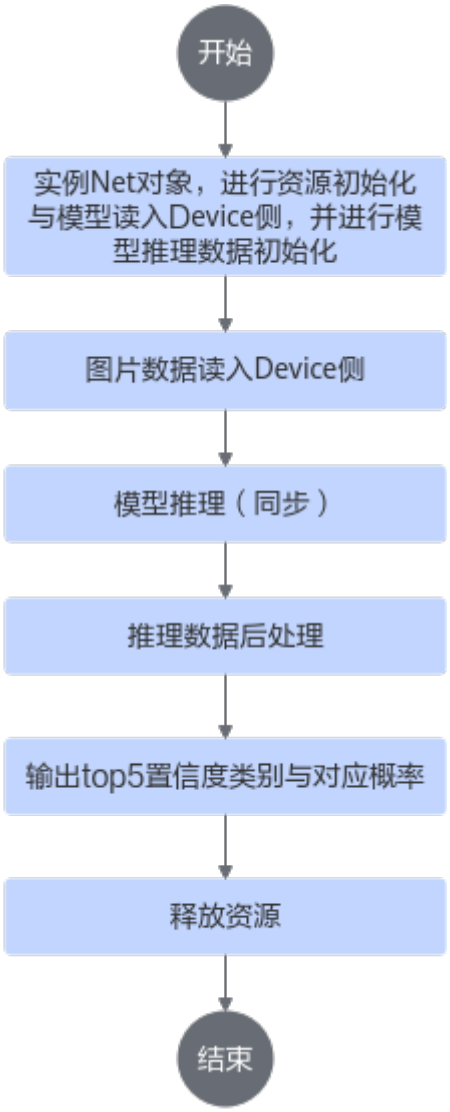
功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。
	Context管理	acl.rt.create_context	创建Context。
模型初始化	模型加载与执行	acl.mdl.load_from_file	从*.om文件加载模型到device侧。
	数据类型及操作接口	acl.mdl.create_desc	创建模型描述数据类型。
	数据类型及操作接口	acl.mdl.get_desc	获取模型描述数据类型。
模型推理	模型加载与执行	acl.mdl.execute	执行模型同步推理。
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.rt.malloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。

功能	对应ACL模块	ACL 接口函数	功能说明
公共模块	--	acl.util.ptr_to_numpy	void*数据转换为numpy类型数据。
	--	acl.util.numpy_to_ptr	numpy类型数据转换为void*数据。
数据后处理	数据类型及操作接口	acl.mdl.get_dataset_buffer	获取数据集中信息。
	数据类型及操作接口	acl.mdl.get_dataset_num_buffers	获取数据集中信息。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。
	模型加载与执行	acl.mdl.unload	卸载模型。
	Context管理	acl.rt.destroy_context	销毁Context。
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

模型同步推理流程图

模型同步推理流程图如图9-4所示。

图 9-4 模型同步推理流程图



目录结构

目录结构如下所示。

```
resnet50_imagenet_classification
├──src
│   ├── acl_net.py //运行文件。
│   └── constant.py //常量定义。
├── data
│   ├── dog1_1024_683.jpg //测试图片数据。
│   └── dog2_1024_683.jpg //测试图片数据。
├── caffe_model
│   ├── resnet50.caffemodel //ResNet-50模型。
│   └── resnet50.prototxt // ResNet-50模型的网络文件。
└── model
    └── resnet50.om //转换后的模型文件。
```


9.5.2.5.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品

模型转换

步骤1 以HwHiAiUser（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 准备数据。

从以下链接获取ResNet-50网络的权重文件（*.caffemodel）、模型文件（resnet50.prototxt），并以HwHiAiUser（运行用户）将获取的文件上传至开发环境的“resnet50_imagenet_classification样例目录/caffe_model”目录下。

- 从gitee上获取：单击**Link**，查看README.md，查找获取原始模型的链接。
- 从GitHub上获取：单击**Link**，查看README.md，查找获取原始模型的链接。

步骤4 将ResNet-50网络转换为适配昇腾AI处理器的离线模型（*.om文件）。

切换到“resnet50_imagenet_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --model=caffe_model/resnet50.prototxt --weight=caffe_model/resnet50.caffemodel --framework=0 --output=model/resnet50 --soc_version=Ascendxxx --input_format=NCHW --input_fp16_nodes=data --output_type=FP32 --out_nodes=prob:0
```

说明

- “--output”参数：生成的“resnet50.om”文件存放在“resnet50_imagenet_classification/model”目录下。
- 使用**atc**命令时用户需保证对“resnet50_imagenet_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤5 以HwHiAiUser（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 准备测试数据。 请从以下链接获取该样例的输入图片，并以运行用户将获取的文件上传至开发环境的“resnet50_imagenet_classification/data”目录下。如果目录不存在，需自行创建。

- **测试图片1**
- **测试图片2**

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“resnet50_imagenet_classification”路径下执行如下命令：

```
python3 ./src/acl_net.py
```

步骤5 执行成功后，在屏幕上的关键提示信息示例如下。

```
Using device id:0
model path:./model/resnet50.om
images path:./data
init resource stage:
model_id:1
init resource success
images:./data/dog1_1024_683.jpg
data interaction from host to device
data interaction from host to device success
execute stage:
execute stage success
data interaction from device to host
data interaction from device to host success
======== top5 inference results: =============
[161]: 0.767578
[162]: 0.154785
[167]: 0.038513
[163]: 0.021606
[166]: 0.011658
images:./data/dog2_1024_683.jpg
data interaction from host to device
data interaction from host to device success
execute stage:
execute stage success
data interaction from device to host
data interaction from device to host success
======== top5 inference results: =============
[267]: 0.935547
[266]: 0.041107
[265]: 0.018829
[219]: 0.002607
[160]: 0.000295
*****run finish******
Releasing resources stage:
Resources released successfully.
```

----结束

9.5.2.6 基于 Caffe ResNet-50 网络实现图片分类（异步推理）

9.5.2.6.1 样例介绍

获取样例

单击**Gitee**，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/1_classification/resnet50_async_imagenet_classification”目录下获取resnet50_async_imagenet_classification样例。

功能描述

该样例主要是基于Caffe ResNet-50网络（单输入、单batch）实现图片分类的功能。

将Caffe ResNet-50网络的模型文件转换为适配昇腾AI处理器的离线模型（*.om文件）。该样例中加载该om文件，对2张jpg（*.jpg）图片进行n次异步推理（n作为运行应用的参数，由用户配置，可通过“--execute_times”参数设置，默认为4次），分别得到n次推理结果后，再对推理结果进行处理，输出top5置信度的类别标识。

主要接口

主要接口如表9-11所示。

表 9-11 主要接口介绍

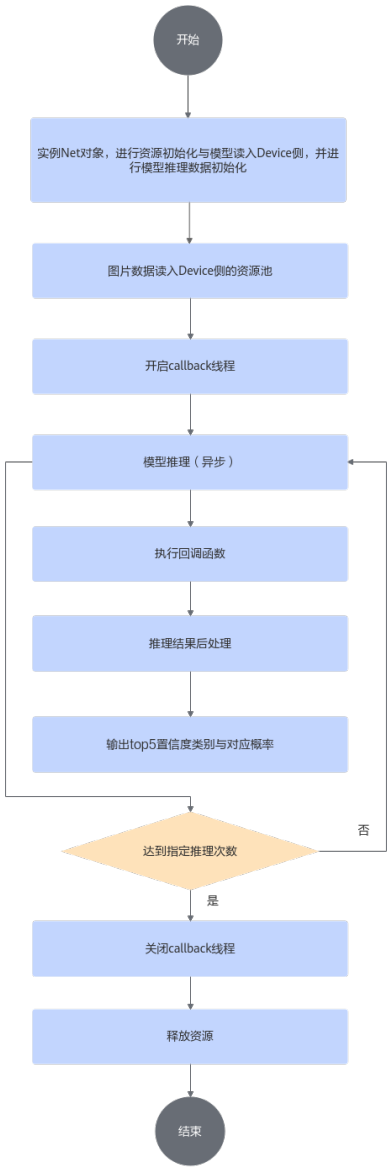
功能	对应ACL模块	ACL 接口函数	功能说明
资源初始化	初始化	acl.init	初始化ACL配置。
	Device管理	acl.rt.set_device	指定用于运算的Device。
	Context管理	acl.rt.create_context	创建Context。
模型初始化	模型加载与执行	acl.mdl.load_from_file	从*.om文件加载模型到device侧。
	数据类型及操作接口	acl.mdl.create_desc	创建模型描述数据类型。
	数据类型及操作接口	acl.mdl.get_desc	获取模型描述数据类型。
模型推理	模型加载与执行	acl.mdl.execute_async	执行模型异步推理。
同步等待	同步等待	acl.rt.subscribe_report	指定处理Stream上回调函数的线程。
	同步等待	acl.rt.unsubscribe_report	取消线程注册，Stream上的回调函数不再由指定线程处理。
	同步等待	acl.rt.launch_callback	在Stream的任务队列中增加一个需要在Host/Device上执行的回调函数。
	同步等待	acl.rt.process_report	等待指定时间后，触发回调处理。

功能	对应ACL模块	ACL 接口函数	功能说明
数据交互	内存管理	acl.rt.memcpy	数据传输，Host->Device或Device->Host。
	内存管理	acl.rt.malloc	申请Device上的内存。
	内存管理	acl.rt.malloc_host	申请Host上的内存。
公共模块	--	acl.util.ptr_to_numpy	void*数据转换为numpy类型数据。
	--	acl.util.numpy_to_ptr	numpy类型数据转换为void*数据。
	--	acl.util.start_thread	启动一个回调函数线程。
	--	acl.util.stop_thread	回收一个回调函数线程。
数据后处理	数据类型及操作接口	acl.mdl.get_dataset_buffer	获取数据集中信息。
	数据类型及操作接口	acl.mdl.get_dataset_num_buffers	获取数据集中信息。
资源释放	内存管理	acl.rt.free	释放Device上的内存。
	内存管理	acl.rt.free_host	释放Host上的内存。
	模型加载与执行	acl.mdl.unload	卸载模型。
	Context管理	acl.rt.destroy_context	销毁Context。
	Device管理	acl.rt.reset_device	复位当前运算的Device，回收Device上的资源。
	去初始化	acl.finalize	实现ACL去初始化。

模型异步推理流程图

模型异步推理流程图如图9-5所示。

图 9-5 模型异步推理流程图



目录结构

目录结构如下所示。

```
resnet50_async_imagenet_classification
├──src
│   ├── acl_net.py //运行文件。
│   └── constant.py //常量定义。
├── data
│   ├── dog1_1024_683.jpg //测试图片数据。
│   └── dog2_1024_683.jpg //测试图片数据。
├── caffe_model
│   ├── resnet50.caffemodel //ResNet-50模型。
│   └── resnet50.prototxt // ResNet-50模型的网络文件。
```



```
└── model
    └── resnet50.om //转换后的模型文件。
```

9.5.2.6.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）
- Atlas 训练系列产品

模型转换

步骤1 以HwHiAiUser（运行用户）登录开发环境。

步骤2 参见《ATC工具使用指南》中的ATC工具使用环境搭建，获取ATC工具并设置环境变量。

步骤3 准备数据。

从以下链接获取ResNet-50网络的权重文件（*.caffemodel）、模型文件（resnet50.prototxt），并以HwHiAiUser（运行用户）将获取的文件上传至开发环境的“resnet50_async_imagenet_classification样例目录/caffe_model”目录下。

- 从gitee上获取：单击**Link**，查看README.md，查找获取原始模型的链接。
- 从GitHub上获取：单击**Link**，查看README.md，查找获取原始模型的链接。

步骤4 将ResNet-50网络转换为适配昇腾AI处理器的离线模型（*.om文件）。

切换到“resnet50_async_imagenet_classification”目录，执行如下命令。*Ascendxxx*为使用的昇腾AI处理器版本，请用户自行替换。

```
atc --model=caffe_model/resnet50.prototxt --weight=caffe_model/resnet50.caffemodel --framework=0 --output=model/resnet50 --soc_version=Ascendxxx --input_format=NCHW --input_fp16_nodes=data --output_type=FP32 --out_nodes=prob:0
```

说明

- “--output”参数：生成的“resnet50.om”文件存放在“resnet50_async_imagenet_classification/model”目录下。
- 使用**atc**命令时用户需保证对“resnet50_async_imagenet_classification”目录有写权限。
- 如果无法确定当前设备的soc_version，则在安装NPU驱动包的服务器执行**npu-smi info**命令进行查询，在查询到的“Name”前增加Ascend信息，例如“Name”对应取值为*xxxyy*，实际配置的soc_version值为Ascend*xxxyy*。

步骤5 以HwHiAiUser（运行用户）将开发环境的样例目录及目录下的文件上传到运行环境。

----结束

运行应用

步骤1 登录运行环境。

步骤2 准备测试数据。 请从以下链接获取该样例的输入图片，并以运行用户将获取的文件上传至开发环境的“resnet50_async_imagenet_classification/data”目录下。如果目录不存在，需自行创建。

- **测试图片1**
- **测试图片2**

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“resnet50_async_imagenet_classification”路径下执行如下命令：

```
python3 ./src/acl_net.py
```

步骤5 执行成功后，在屏幕上的关键提示信息示例如下。

```
Using device id:0
execute_times:4
callback_interval:1
mem_pools:4
model path:../model/resnet50.om
images path:./data
init resource stage:
init resource stage success
data interaction from host to device
data interaction from host to device success
execute stage:
execute stage success
callback func stage:

======== top5 inference results: =============
[161]: 0.767578
[162]: 0.154785
[167]: 0.038513
[163]: 0.021606
[166]: 0.011658
callback func stage success
callback func stage:

======== top5 inference results: =============
[267]: 0.935547
[266]: 0.041107
[265]: 0.018829
[219]: 0.002607
[160]: 0.000295
callback func stage success
callback func stage:

======== top5 inference results: =============
[161]: 0.767578
[162]: 0.154785
[167]: 0.038513
[163]: 0.021606
[166]: 0.011658
callback func stage success
callback func stage:

======== top5 inference results: =============
[267]: 0.935547
[266]: 0.041107
[265]: 0.018829
[219]: 0.002607
[160]: 0.000295
callback func stage success
exit acl.rt.process_report
run finish!!!
release source stage:
release source stage success
```

----结束

9.5.2.7 媒体数据处理（视频编码）

9.5.2.7.1 样例介绍

获取样例

单击**Gitee**，进入Ascend samples开源仓，详细参见README中的“版本说明”下载配套版本的sample包，从“python/level2_simple_inference/0_data_process/venc_image”目录下获取venc_image样例。

功能描述

该样例将一张YUV420SP NV12格式的图片连续编码n次，生成一个H265格式的视频码流（n可配，通过运行应用时设置入参来配置，由代码中的“venc_cnt”参数来控制，默认为16次）。

目录结构

如下为模型文件转换后的示例目录结构，model文件夹是转换后生成的。

```
venc_image
├──src
│  └── acl_venc.py //视频编码实现文件
└── data
  └── dvpp_venc_128x128_nv12.yuv //用户待处理的图片文件
```

9.5.2.7.2 运行应用

须知

样例步骤适用于以下产品。

- Atlas 200/300/500 推理产品
- Atlas 推理系列产品（Ascend 310P处理器）

步骤1 登录运行环境。

步骤2 准备输入数据。 从**Link**上获取输入图片文件“dvpp_venc_128x128_nv12.yuv”，并以HwHiAiUser（运行用户）上传至开发环境的“venc_image样例目录/data”目录下。

步骤3 参照**9.5.1.3 环境变量配置**完成运行环境的配置。

步骤4 在“venc_image”路径下执行如下命令：

```
python3 ./src/acl_venc.py
```

步骤5 执行成功后，在屏幕上的关键提示信息示例如下。

```
[INFO] start_thread 46994427340544 0
[INFO] cb_thread_func args_list =  52200976 1000 True
[INFO] set venc channel desc
[INFO] create_frame_config
[INFO] set pic desc size
[INFO] set frame config
[INFO] [venc] stream_data size 5431
[INFO] [venc] stream_data size 704
[INFO] [venc] stream_data size 693
[INFO] [venc] stream_data size 922
```



```
[INFO] [venc] stream_data size 1153
[INFO] [venc] stream_data size 689
[INFO] [venc] stream_data size 857
[INFO] [venc] stream_data size 1273
[INFO] [venc] stream_data size 785
[INFO] [venc] stream_data size 976
[INFO] [venc] stream_data size 869
[INFO] [venc] stream_data size 1445
[INFO] venc send frame eos
[INFO] [venc] stream_data size 845
[INFO] [venc] stream_data size 1003
[INFO] [venc] stream_data size 922
[INFO] [venc] stream_data size 905
[INFO] acl.media.venc_send_frame ret= 0
[INFO] venc process success
[INFO] venc_process end
[INFO] cb_thread_func acl.rt.destroy_context ret= 0
[INFO] stop_thread 0
[INFO] thread join
[INFO] free resource
```

----结束

A pyACL 表达约定

接口命名规则

接口命名同时满足如下规则：

1. acl+*接口类别缩写+操作动词+对象*
2. 操作动词和对象均采用首字母大写。

接口类别

接口类别	缩写	描述
runtime	rt	表示运行管理类的接口
DVPP	media	表示媒体数据处理类的接口
AIPP	aipp	表示aipp（AI Preprocessing）类的接口
CBLAS	blas	表示blas类的接口
model	mdl	表示模型推理类的接口
OP	op	表示算子执行类的接口
fv	fv	表示特征向量检索的接口
Profiling	prof	表示Profiling配置类接口

说明

- 缩写原则上不超过4个字母。
- 在接口命名中，如果类别与操作对象重叠时，操作动词后的对象将省略。如：acl.mdl.load_from_file_with_mem，表示model类接口，这个接口表示含义是load model from file，因此在接口命名中Load后面mdl将被省略。

变量命名

本文代码示例中涉及的变量，其中，命名带下划线的变量（例如：_deviceId_）表示类的私有变量。

B 使用约束

- 进入系统休眠前，需要确保将正在运行的AI推理业务、媒体数据处理业务等进程退出。等待唤醒成功后，再继续执行业务。
- 不支持使用fork函数创建多个进程，且在进程中调用pyACL接口的场景，否则进程运行时会报错或者卡死。
- 不支持在acl.rt.memcpy_async、acl.rt.memset_async接口等异步操作内存过程中使用fork以及封装了fork的函数，如system、posix_spawnp等，否则会导致进程运行时会报错，甚至卡死等不可预期的错误。
- 对于创建类接口（例如：acl.rt.create_stream、acl.rt.create_event、acl.create_data_buffer等），用户调用该类接口创建对应的资源后，资源使用完成后，建议及时调用对应的销毁类接口（例如：acl.rt.destroy_stream、acl.rt.destroy_event、acl.destroy_data_buffer等），否则，程序可能会异常。
- 对于销毁类接口（例如：acl.rt.destroy_stream、acl.rt.destroy_event、acl.rt.free、acl.destroy_data_buffer等），用户调用该类接口后，不能继续使用已释放或销毁的资源，建议用户调用销毁类接口后，将相关资源设置为无效值（例如，设置为None）。
- 对于Atlas 200/300/500 推理产品，物理机场景下，一个Device上最多只能支持64个用户进程，Host最多只能支持Device个数*64个进程；虚拟机场景下，一个Device上最多只能支持32个用户进程，Host最多只能支持Device个数*32个进程。
- 对于Atlas 训练系列产品，物理机场景下，一个Device上最多只能支持64个用户进程，Host最多只能支持Device个数*64个进程；虚拟机场景下，一个Device上最多只能支持32个用户进程，Host最多只能支持Device个数*32个进程。
- 对于Atlas 推理系列产品（Ascend 310P处理器），物理机场景下，一个Device上最多只能支持64个用户进程，Host最多只能支持Device个数*64个进程；虚拟机场景下，一个Device上最多只能支持32个用户进程，Host最多只能支持Device个数*32个进程。
- 对于Atlas 200/500 A2推理产品，一个Device上最多只能支持64个用户进程，Host最多只能支持Device个数*64个进程。
- 对于Atlas A2训练系列产品，一个Device上最多只能支持63个用户进程，Host最多只能支持Device个数*63个进程。
- 使用pyACL提供的内存申请接口（例如acl.rt.malloc、acl.media.dvpp_malloc等）申请内存后，为确保内存中不会有脏数据，建议在使用内存前先调用acl.rt.memset或acl.rt.memset_async接口先清空内存，例如acl.rt.memset(dev_buffer_ptr, dev_buffer_size, 0, dev_buffer_size)。

- Ascend RC形态下，如果应用程序中涉及aclrtMalloc、acldvppMalloc、hi_mpi_dvpp_malloc等内存申请接口，应用程序在Device上运行时，当前默认在内存不足时，应用程序可能会挂起，等待内存资源，用户可以根据实际需求选择启用操作系统提供的一些配置（例如，enable_oom_killer），这样在内存不足时，应用程序会自动退出，不会一直等待。

  若启用enable_oom_killer，您需登录Device，在“/proc/sys/vm”目录下，以root用户启用enable_oom_killer，命令示例如下，1表示启用，0表示禁用：

  ```
  echo 1 > enable_oom_killer
  ```

