CANN
8.0.RC2.alpha001

AscendCL 应用软件开发指南(C&C++)

文档版本	01
发布日期	2024-05-13

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

本节介绍本文档的内容概要，并给出各知识点的学习顺序建议。

概述

本文用于指导开发人员基于现有模型、使用AscendCL（Ascend Computing Language）提供的C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。

通过本文档您可以学习以下内容：

- 了解AscendCL的功能架构、基本概念以及接口的典型调用流程。
- 使用AscendCL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。

具备C++/C语言程序开发能力、对机器学习或深度学习有一定了解的开发者，可以更好地理解本文档。

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**2 快速入门**了解下应用开发的整体过程，然后了解**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+示例代码来深入学习。

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

如果您在使用本文档时，已了解使用AscendCL接口开发应用的基本步骤，想进一步学习时，可参照下图的应用开发向导。

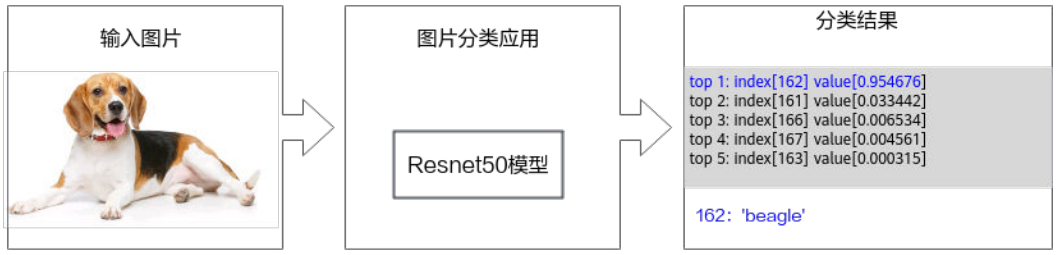
2 快速入门

在本节中，您可以通过一个简单的图片分类应用了解使用AscendCL接口（C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”，从名称上，我们也能直观地看出它的作用：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？当然得先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。

鉴于当前我们是入门内容，此处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：BGR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**4.3 准备开发和运行环境**。

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

  开发环境指编译开发代码的环境，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。

  开发环境和运行环境可以合设在同一台服务器上，也可以分设：

  - 合设场景下，登录到合设的服务器上，不用切换开发环境、运行环境。
  - 分设场景下，若开发环境和运行环境上的操作系统架构不同，在开发环境中需使用交叉编译，这样编译出来的可执行文件，才可以在运行环境中执行。

  说明

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

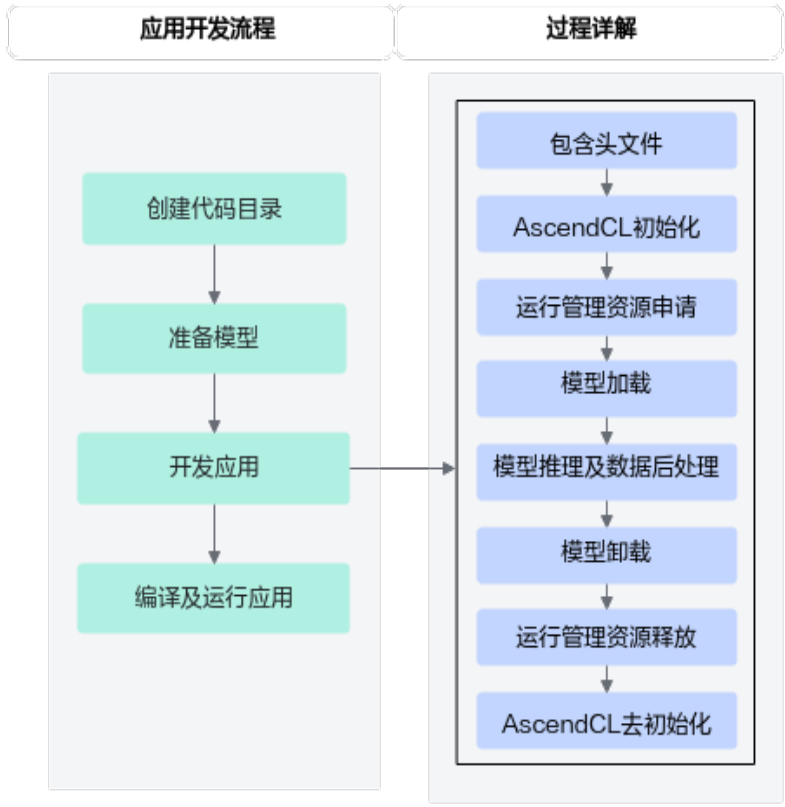
- 运行用户：

  运行驱动进程、推理业务或执行训练的用户。

了解开发过程

AscendCL（Ascend Computing Language）是一套用于在CANN（Compute Architecture for Neural Networks）上开发深度神经网络推理应用的C语言API库，提供模型加载与执行、媒体数据处理、算子加载与执行等API，能够实现在昇腾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AscendCL接口，这些AscendCL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

您可以单击**Link**，获取基础的样例包MyFirstApp_ONNX.zip，以运行用户将该样例包**上传**至开发环境（例如，存放在/root/demo目录下），先执行“**chmod +x MyFirstApp_ONNX.zip**”命令增加执行权限，再执行“**unzip MyFirstApp_ONNX.zip**”命令解压样例包。

该样例包中包含测试图片、测试图片预处理脚本、编译脚本等，如下所示。

```
MyFirstApp_ONNX
├── data
│   ├── dog1_1024_683.jpg          // 测试图片
├── model                          // 用于存放ResNet-50模型文件
├── script
│   ├── transferPic.py             // 将测试图片预处理为符合模型要求的图片
                                   // 包括将*.jpg转换为*.bin，同时将图片从1024*683的分辨率缩放为224*224
├── src
│   ├── CMakeLists.txt             // cmake编译脚本
│   ├── main.cpp                   // 主函数，图片分类功能的实现文件
├── sample_build.sh                // 编译代码的脚本
├── sample_run.sh                  // 运行应用的脚本
```


准备模型

对于开源框架的网络模型，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）。

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处理器版本替换。

```
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*。

----结束

开发应用

有了模型，接下来可以使用AscendCL接口加载、执行模型，开发AI应用了。

在开发环境中，打开“MyFirstApp_ONNX/src/main.cpp“文件，在该cpp中添加如下步骤中灰色底纹的代码。

步骤1 **定义一个main函数**，按照**了解开发过程**中的开发应用的过程详解，串联整个应用的代码逻辑。

```
int main()
{
    // 1.定义一个资源初始化的函数，用于AscendCL初始化、运行管理资源申请（指定计算设备）
  InitResource();

    // 2.定义一个模型加载的函数，加载图片分类的模型，用于后续推理使用
  const char *modelPath = "../model/resnet50.om";
  LoadModel(modelPath);

    // 3.定义一个读图片数据的函数，将测试图片数据读入内存，并传输到Device侧，用于后续推理使用
    const char *picturePath = "../data/dog1_1024_683.bin";
  LoadPicture(picturePath);

    // 4.定义一个推理的函数，用于执行推理
```



```
    Inference();

    // 5.定义一个推理结果数据处理的函数，用于在终端上屏显测试图片的top5置信度的类别编号
    PrintResult();

    // 6.定义一个模型卸载的函数，卸载图片分类的模型
    UnloadModel();

    // 7.定义一个函数，用于释放内存、销毁推理相关的数据类型，防止内存泄露
    UnloadPicture();

    // 8.定义一个资源去初始化的函数，用于AscendCL去初始化、运行管理资源释放（释放计算设备）
    DestroyResource();
}
```

了解总体的代码逻辑后，接下来开始写各自定义函数的实现，**以下函数的实现请按顺序添加在main函数之前**。

步骤2 include依赖的头文件，包括AscendCL、C或C++标准库的头文件。

在“MyFirstApp_ONNX/src/main.cpp”文件的开头增加如下代码。

```
#include "acl/acl.h"
#include <iostream>
#include <fstream>
#include <cstring>
#include <map>
#include <math.h>

using namespace std;
```

步骤3 资源初始化。

资源初始化包括2部分：

- 调用aclInit接口初始化AscendCL：

  使用AscendCL接口开发应用时，必须先初始化AscendCL ，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。

  在初始化时，还支持跟推理相关的可配置项（例如，性能相关的采集信息配置），以json格式的配置文件传入AscendCL初始化接口。如果当前的默认配置已满足需求（例如，默认不开启性能相关的采集信息配置），无需修改，可向AscendCL初始化接口中传入nullptr。

- 调用aclrtSetDevice接口指定计算设备。

```
int32_t deviceId = 0;
void InitResource()
{
    aclError ret = aclInit(nullptr);
    ret = aclrtSetDevice(deviceId);
}
```

有初始化就有去初始化，在确定完成了AscendCL的所有调用之后，或者进程退出之前，需执行资源去初始化，请参见**步骤10**。

步骤4 模型加载。

此处通过加载om模型文件来实现模型加载，如何获取om模型文件请参见**准备模型**。

```
uint32_t modelId;
void LoadModel(const char* modelPath)
{
    aclError ret = aclmdlLoadFromFile(modelPath, &modelId);
}
```

有加载就有卸载，模型推理结束后，需要卸载模型，请参见**步骤8**。

步骤5 将测试图片读入内存，再传输到Device侧，供推理使用。

```
size_t pictureDataSize = 0;
void *pictureHostData;
void *pictureDeviceData;

//申请内存，使用C/C++标准库的函数将测试图片读入内存
void ReadPictureTotHost(const char *picturePath)
{
    string fileName = picturePath;
    ifstream binFile(fileName, ifstream::binary);
    binFile.seekg(0, binFile.end);
    pictureDataSize = binFile.tellg();
    binFile.seekg(0, binFile.beg);
    aclError ret = aclrtMallocHost(&pictureHostData, pictureDataSize);
    binFile.read((char*)pictureHostData, pictureDataSize);
    binFile.close();
}

//申请Device侧的内存，再以内存复制的方式将内存中的图片数据传输到Device
void CopyDataFromHostToDevice()
{
    aclError ret = aclrtMalloc(&pictureDeviceData, pictureDataSize, ACL_MEM_MALLOC_HUGE_FIRST);
    ret = aclrtMemcpy(pictureDeviceData, pictureDataSize, pictureHostData, pictureDataSize,
ACL_MEMCPY_HOST_TO_DEVICE);
}

void LoadPicture(const char* picturePath)
{
    ReadPictureTotHost(picturePath);
    CopyDataFromHostToDevice();
}
```

步骤6 执行推理。

在调用AscendCL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AscendCL规定的数据类型存放。相关数据类型如下：

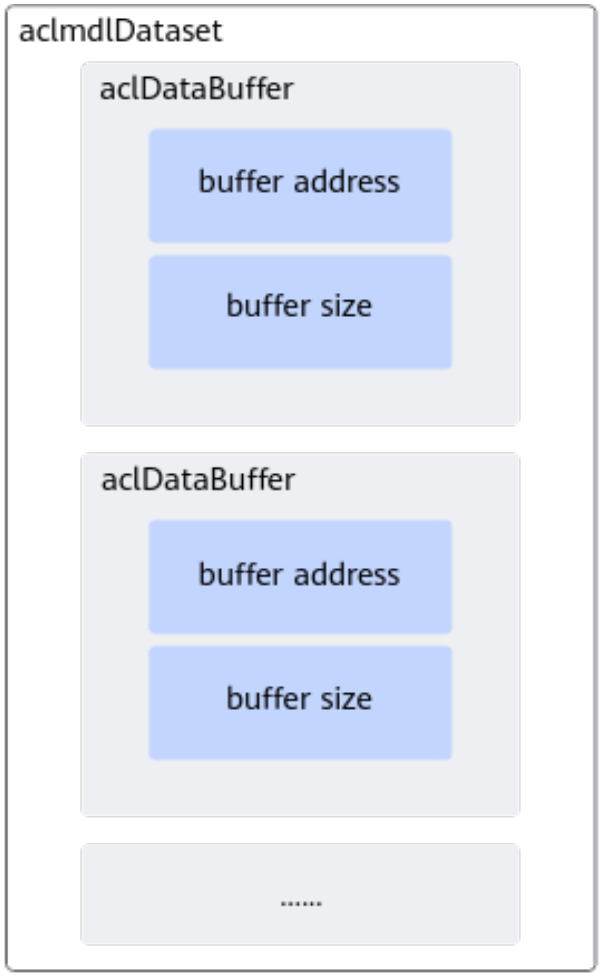
- 使用aclmdlDesc类型的数据描述模型基本信息（例如输入/输出的个数、名称、数据类型、Format、维度信息等）。

  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




```
aclmdlDataset *inputDataSet;
aclDataBuffer *inputDataBuffer;
aclmdlDataset *outputDataSet;
aclDataBuffer *outputDataBuffer;
aclmdlDesc *modelDesc;
size_t outputDataSize = 0;
void *outputDeviceData;

// 准备模型推理的输入数据结构
void CreateModelInput()
{
        // 创建aclmdlDataset类型的数据，描述模型推理的输入
    inputDataSet = aclmdlCreateDataset();
    inputDataBuffer = aclCreateDataBuffer(pictureDeviceData, pictureDataSize);
    aclError ret = aclmdlAddDatasetBuffer(inputDataSet, inputDataBuffer);
}

// 准备模型推理的输出数据结构
void CreateModelOutput()
{
      // 创建模型描述信息
    modelDesc =  aclmdlCreateDesc();
    aclError ret = aclmdlGetDesc(modelDesc, modelId);
        // 创建aclmdlDataset类型的数据，描述模型推理的输出
    outputDataSet = aclmdlCreateDataset();
        // 获取模型输出数据需占用的内存大小，单位为Byte
    outputDataSize = aclmdlGetOutputSizeByIndex(modelDesc, 0);
        // 申请输出内存
    ret = aclrtMalloc(&outputDeviceData, outputDataSize, ACL_MEM_MALLOC_HUGE_FIRST);
```



```
    outputDataBuffer = aclCreateDataBuffer(outputDeviceData, outputDataSize);
    ret = aclmdlAddDatasetBuffer(outputDataSet, outputDataBuffer);
}

// 执行模型
void Inference()
{
        CreateModelInput();
    CreateModelOutput();
    aclError ret = aclmdlExecute(modelId, inputDataSet, outputDataSet);
}
```

步骤7 处理模型推理的结果数据，在屏幕上显示图片的top5置信度的类别编号。

```
void *outputHostData;

void PrintResult()
{
        // 获取推理结果数据
        aclError ret = aclrtMallocHost(&outputHostData, outputDataSize);
        ret = aclrtMemcpy(outputHostData, outputDataSize, outputDeviceData, outputDataSize,
ACL_MEMCPY_DEVICE_TO_HOST);
        // 将内存中的数据转换为float类型
        float* outFloatData = reinterpret_cast<float *>(outputHostData);

        // 屏显测试图片的top5置信度的类别编号
        map<float, unsigned int, greater<float>> resultMap;
        for (unsigned int j = 0; j < outputDataSize / sizeof(float);++j)
        {
                resultMap[*outFloatData] = j;
                outFloatData++;
        }

        // do data processing with softmax and print top 5 classes
        double totalValue=0.0;
        for (auto it = resultMap.begin(); it != resultMap.end(); ++it) {
            totalValue += exp(it->first);
        }

        int cnt = 0;
        for (auto it = resultMap.begin();it != resultMap.end();++it)
        {
                if(++cnt > 5)
                {
                        break;
                }
                printf("top %d: index[%d] value[%lf] \n", cnt, it->second, exp(it->first) /totalValue);
        }
}
```

步骤8 卸载模型，并释放模型描述信息。

推理结束，需及时释放模型描述信息、卸载模型。

```
void UnloadModel()
{
        // 释放模型描述信息
    aclmdlDestroyDesc(modelDesc);
        // 卸载模型
    aclmdlUnload(modelId);
}
```

步骤9 释放内存、销毁推理相关的数据类型。

```
void UnloadPicture()
{
    aclError ret = aclrtFreeHost(pictureHostData);
    pictureHostData = nullptr;
    ret = aclrtFree(pictureDeviceData);
    pictureDeviceData = nullptr;
    aclDestroyDataBuffer(inputDataBuffer);
    inputDataBuffer = nullptr;
```



```
    aclmdlDestroyDataset(inputDataSet);
    inputDataSet = nullptr;

    ret = aclrtFreeHost(outputHostData);
    outputHostData = nullptr;
    ret = aclrtFree(outputDeviceData);
    outputDeviceData = nullptr;
    aclDestroyDataBuffer(outputDataBuffer);
    outputDataBuffer = nullptr;
    aclmdlDestroyDataset(outputDataSet);
    outputDataSet = nullptr;
}
```

步骤10 资源释放。

在确定完成了AscendCL的所有调用之后，或者进程退出之前，需调用如下接口实现计算设备释放，AscendCL去初始化。

```
void DestroyResource()
{
    aclError ret = aclrtResetDevice(deviceId);
    aclFinalize();
}
```

----结束

编译及运行应用

步骤1 编译代码。

以运行用户登录开发环境，切换到MyFirstApp_ONNX目录下，执行以下命令。

如下为设置环境变量的示例，<SAMPLE_DIR>表示样例所在的目录，*$HOME/Ascend/ascend-toolkit*表示CANN软件包的安装路径，*<arch-os>*中*arch*表示操作系统架构（需根据运行环境的架构选择），*os*表示操作系统（需根据运行环境的操作系统选择）。

```
export APP_SOURCE_PATH=<SAMPLE_DIR>/MyFirstApp_ONNX
export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest
export NPU_HOST_LIB=$HOME/Ascend/ascend-toolkit/latest/<arch-os>/devlib
chmod +x sample_build.sh
./sample_build.sh
```

说明

- 如果执行脚本报错“ModuleNotFoundError: No module named 'PIL'”，是由于开发环境中缺少Pillow库，需使用**pip3 install Pillow --user**命令安装Pillow库后，再执行脚本。
- 如果执行脚本报错“bash: ./sample_build.sh: /bin/bash^M: bad interpreter: No such file or directory”，是由于开发环境中没有安装dos2unix包，需使用**sudo apt-get install dos2unix**命令安装dos2unix包后，执行**dos2unix sample_build.sh**，然后再执行脚本。

步骤2 运行应用。

以运行用户将MyFirstApp_ONNX目录上传至运行环境，以运行用户登录运行环境，切换到MyFirstApp_ONNX目录下，执行以下命令。

```
chmod +x sample_run.sh
./sample_run.sh
```

终端上屏显的结果如下，index表示类别标识、value表示该分类的最大置信度：

```
top 1: index[162] value[0.954676]
top 2: index[161] value[0.033442]
top 3: index[166] value[0.006534]
top 4: index[167] value[0.004561]
top 5: index[163] value[0.000315]
```


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：

"162": ["beagle"]

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

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

"167": ["English foxhound"]

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

----结束

3 视频课程

表 3-1 课程列表

课程	简介
快速入门	本课介绍使用AscendCL接口开发AI推理应用的基本流程。
AscendCL概述	本课介绍AscendCL的架构、基本概念以及使用AscendCL接口开发AI应用的基本流程。
ATC模型转换	本课介绍ATC模型转换工具的基本使用方法。
AscendCL模型推理基础功能	本课结合代码介绍使用AscendCL接口（C语言接口）开发推理应用的基本过程和关键接口，包括运行资源管理、内存管理、数据传输、模型加载与执行。
AscendCL DVPP媒体数据处理	DVPP是昇腾AI处理器内置的图像处理单元，通过AscendCL媒体数据处理接口提供强大的媒体处理硬加速能力，本课结合代码介绍使用AscendCL接口处理视频、图片的基本过程。
AscendCL加载与执行算子	本课结合代码介绍使用AscendCL接口（C语言接口）加载并执行单个算子的关键接口、代码逻辑。
AscendCL模型推理动态特性	本课介绍使用AscendCL接口开发含动态特性（例如动态AIPP、动态Batch、动态分辨率等）的AI应用的基本方法。
AscendCL同步&异步特性	本课介绍AscendCL中同步&异步的概念，并介绍典型的同步等待流程及其关键接口。
AscendCL应用调试	本课介绍如何获取应用运行的日志，并结合典型日志错误说明如何调试AscendCL应用。

4 AscendCL 应用开发概述

4.1 AscendCL 架构及基本概念

本节介绍AscendCL的主要功能、Device/Stream/Context等基本概念以及这些概念之间的关系。

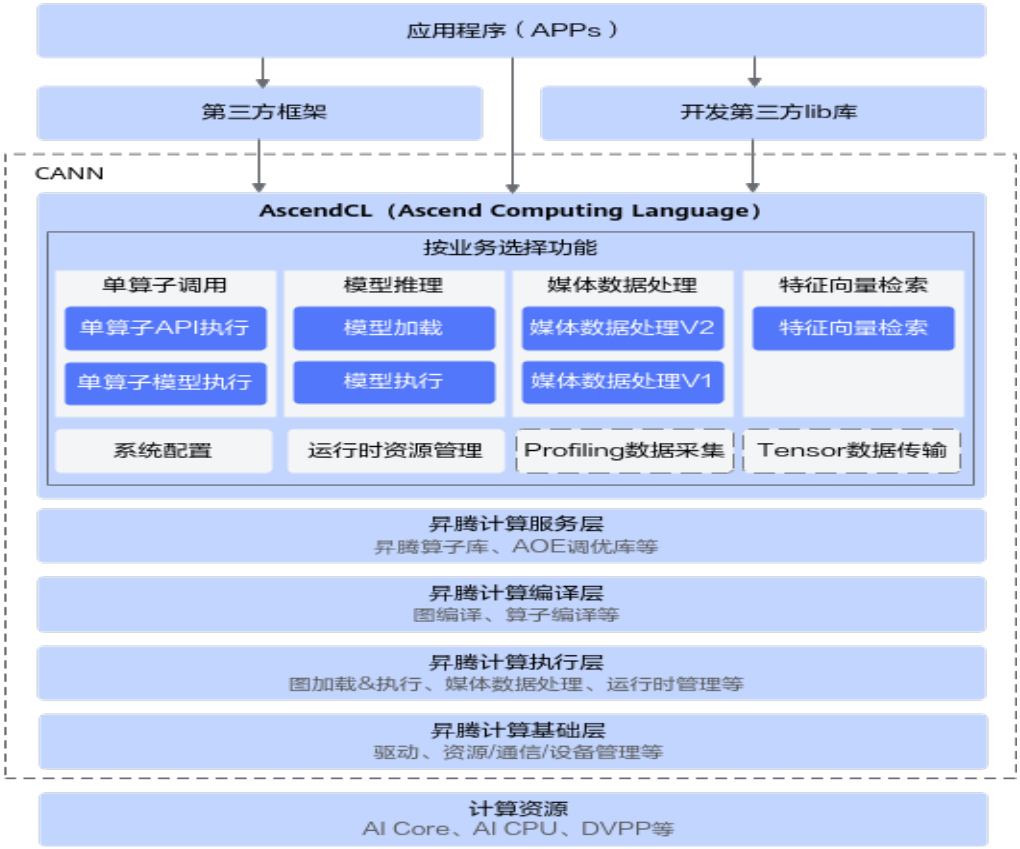
AscendCL 是什么?

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**。其中，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。

AscendCL的优势如下：

- **高度抽象**：算子编译、加载、执行的API归一，相比每个算子一个API，AscendCL大幅减少API数量，降低复杂度。
- **向后兼容**：AscendCL具备向后兼容，确保软件升级后，基于旧版本编译的程序依然可以在新版本上运行。
- **零感知芯片**：一套AscendCL接口可以实现应用代码统一，多款昇腾AI处理器无差异。

图 4-1 逻辑架构图



AscendCL的应用场景：

- **开发应用**：用户可以直接调用AscendCL提供的接口开发图片分类应用、目标识别应用等。
- **供第三方框架调用**：用户可以通过第三方框架调用AscendCL接口，以便使用昇腾AI处理器的计算能力。
- **供第三方开发lib库**：用户还可以使用AscendCL封装实现第三方lib库，以便提供昇腾AI处理器的运行管理、资源管理等能力。

基本概念

表 4-1 概念介绍

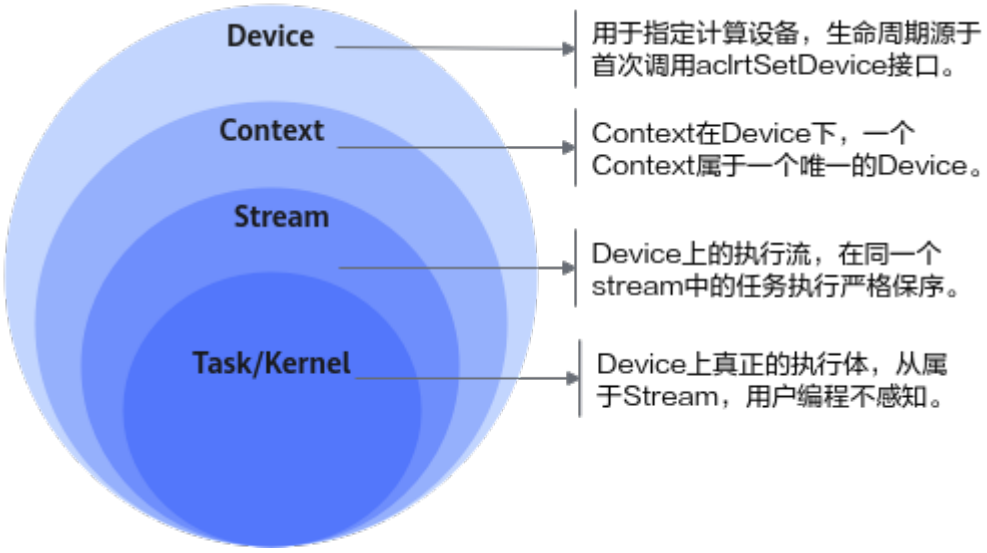
概念	描述
同步/异步	本文中提及的同步、异步是站在调用者和执行者的角度： • 若在调用AscendCL接口后**不等待**Device侧的任务执行完成再返回，则表示调度是**异步**的； • 若在调用AscendCL接口后**需等待**Device侧的任务执行完成再返回，则表示调度是**同步**的。
进程/线程	本文中提及的进程、线程，若无特别注明，则表示Host上的进程、线程。

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aclrtSetDevice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aclrtDestroyContext接口来释放。 ● 显式创建Context：**推荐，**在进程或线程中调用aclrtCreateContext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aclrtSetDevice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aclrtDestroyStream接口来释放。 ● 显式创建Stream：**推荐，**在进程或线程中调用aclrtCreateStream接口显式创建一个Stream。
Event	支持调用AscendCL接口同步Stream之间的任务，例如同一个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 4-2 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aclrtSetDevice接口。
  - 每次调用aclrtSetDevice接口，系统会进行引用计数加1；调用aclrtResetdevice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aclrtSetDevice接口，终结于调用aclrtResetdevice接口使引用计数为零时。隐式Context只会被创建一次，调用aclrtSetDevice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aclrtCreateContext接口，终结于调用aclrtDestroyContext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aclrtCreateStream），当前线程在同一时刻内只能使用其中一个Context，建议通过aclrtSetCurrentContext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aclrtSetCurrentContext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，这个属于隐式创建，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aclrtCreateStream，终结于调用aclrtDestroyStream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aclrtSetCurrentContext进行Device的快速切换。示例代码如下，仅供参考，不可以直接拷贝编译运行：

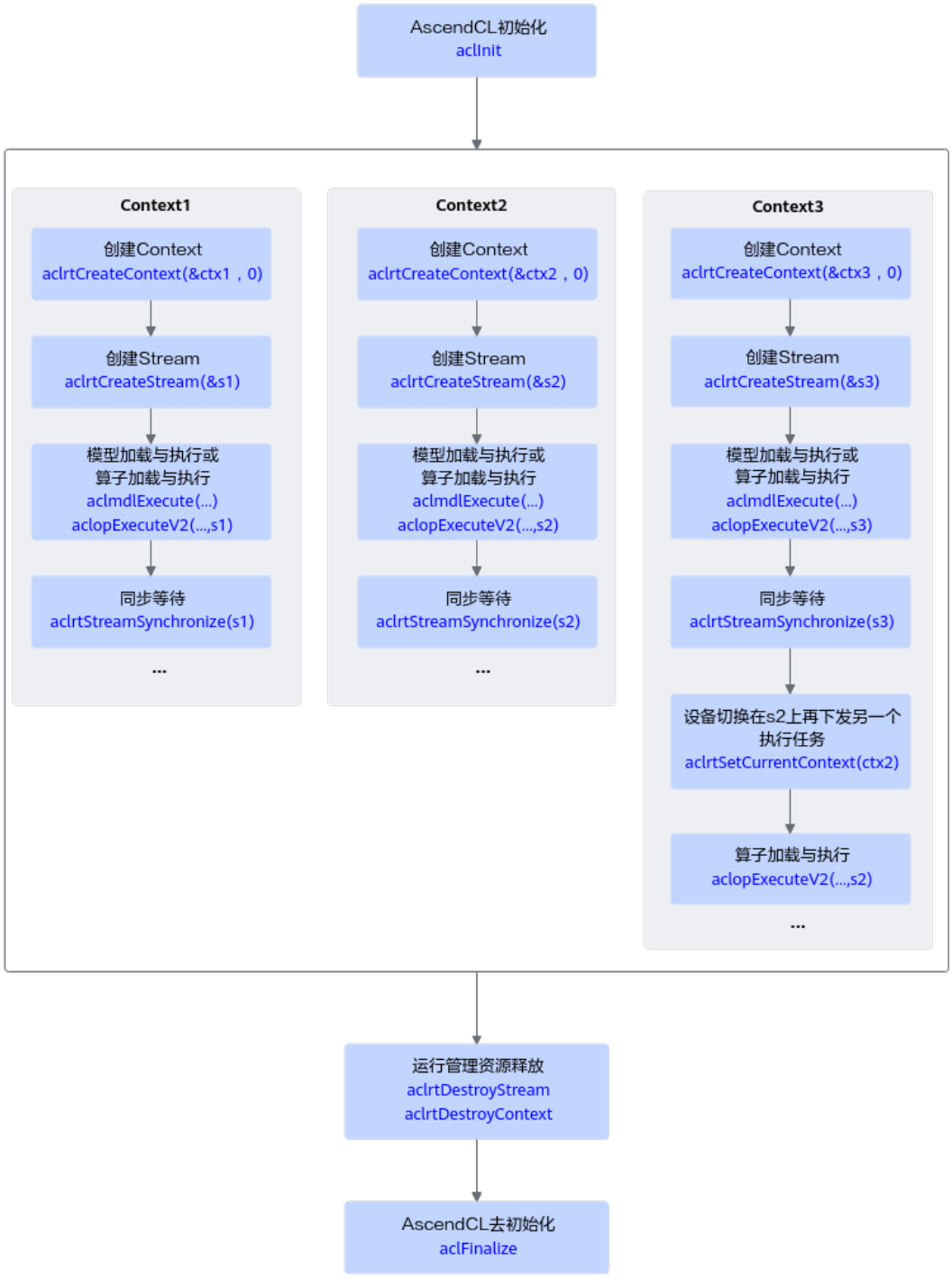
```
...
aclrtCreateContext(&ctx1，0);
aclrtCreateStream(&s1);
aclopExecuteV2(op1,...,s1);
aclrtCreateContext(&ctx2,1);

/*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 */
aclrtCreateStream(&s2);
aclopExecuteV2(op2,...,s2);
aclrtSetCurrentContext(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*/
aclopExecuteV2(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，推荐每个线程创建一个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在Device侧的调度，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aclrtSetCurrentContext设置当前需要使用的Context。

图 4-3 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NULL。

- **默认Context**不允许用户执行aclrtGetCurrentContext或aclrtSetCurrentContext操作，也不允许执行aclrtDestroyContext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，不可以直接拷贝编译运行：

```
...
aclInit(...);
aclrtSetDevice(0);

/*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*/
...
aclopExecuteV2(op1,...,NULL);  //最后一个NULL表示在default stream上执行算子op1
aclopExecuteV2(op2,...,NULL); //最后一个NULL表示在default stream上执行算子op2
aclrtSynchronizeStream(NULL);

/*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*/
...
aclrtResetDevice(0);  //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。

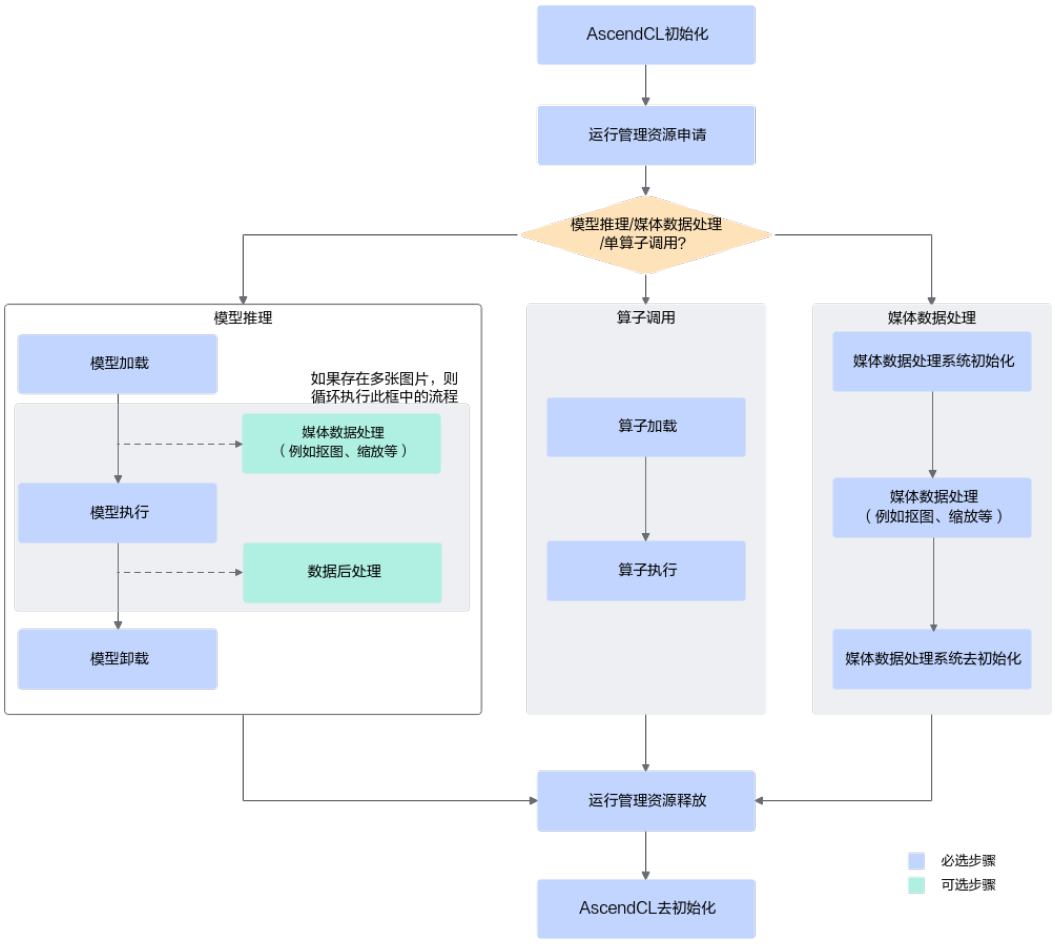
4.2 AscendCL 接口调用流程

本节介绍AscendCL接口调用流程以及调用AscendCL接口依赖的头文件和库文件。

接口调用流程

调用AscendCL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AscendCL接口开发AI应用的整体接口调用流程。

图 4-4 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，例如，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。

1. AscendCL初始化。

   调用aclInit接口实现初始化AscendCL。

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**6.1 运行管理资源申请与释放**。

3. 模型推理/单算子调用/媒体数据处理。

   - **模型推理**

     i. 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**8.3.1 模型加载**。

     但加载模型前，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**8.2 模型构建**。

     ii. （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**9 媒体数据处理（含图像/视频等）**

iii. 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**8.3.2 模型执行**。

iv. （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。

v. 模型卸载：调用aclmdlUnload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AscendCL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。

  算子调用的接口调用流程，请参见**7.2 单算子调用流程**。

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**。

   接口调用流程，请参见**6.1 运行管理资源申请与释放**。

5. AscendCL去初始化。

   调用aclFinalize接口实现AscendCL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。

调用接口依赖的头文件和库文件说明

您需要根据实际使用的接口来include依赖的文件，AscendCL中各头文件的用途如下表所示。

AscendCL头文件在“CANN软件安装后文件存储路径/include/”目录下，AscendCL库文件在“CANN软件安装后文件存储路径/lib64/”目录下。

须知

编译基于AscendCL接口的代码逻辑时，请按照include的头文件依赖对应的库文件，如果引用多余的so文件（例如libascendcl.a），可能导致版本功能异常或后续版本升级时存在兼容性问题。

表 4-2 头文件列表

定义接口的头文件	用途	对应的库文件
acl/acl_base.h	用于定义基本的数据类型（例如aclDataBuffer、aclTensorDesc等）及其操作接口、枚举值（例如aclFormat）、日志管理接口等。	libascendcl.so
acl/acl.h	该头文件中已包含acl/acl_mdl.h、acl/acl_rt.h、acl/acl_op.h。包含acl.h文件后，可以引用初始化/去初始化、Device管理、算力Group查询与设置、Context管理、Stream管理、同步等待、内存管理、模型加载与执行、算子编译（不包括aclopCompile接口）、算子加载与执行（不包括aclopCompileAndExecute接口）等接口。	libascendcl.so
acl/acl_prof.h	用于定义Profiling配置的接口。	libmsprofiler.so **说明** 为了兼容旧版本，旧版本中支持使用libascendcl.so，但后续版本这种方式会废弃，建议使用libmsprofiler.so，防止后续版本出现兼容性问题。
acl/ops/acl_cblas.h	用于定义CBLAS接口。	libacl_cblas.so
acl/ops/acl_dvpp.h	用于定义媒体数据处理V1版本的接口。	libacl_dvpp.so
acl/ops/acl_fv.h	用于定义特征向量检索的接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的接口。	libacl_retr.so
acl/acl_op_compiler.h	用于定义aclopCompile、aclopCompileAndExecute、aclSetCompileopt等算子在线编译相关的接口、数据类型、枚举值等。	libacl_op_compiler.so

定义接口的头文件	用途	对应的库文件
acl/acl_tdt.h	用于定义Tensor数据传输接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的接口。	libacl_tdt_channel.so
acl/acl_tdt_queue.h	用于定义共享队列管理、共享Buffer管理接口。	libacl_tdt_queue.so
acl/dvpp/hi_dvpp.h	用于定义媒体数据处理V2版本的接口。	libacl_dvpp_mpi.so
hi_mpi_vi.h hi_common_vi.h hi_common_dis.h hi_common_gdc.h hi_media_common.h hi_media_type.h hi_mpi_sys.h	用于定义VI（Vedio Input）视频数据获取功能的接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_vi_mpi.so libacl_dvpp_mpi.so
acl/media目录下： hi_mpi_isp.h hi_common_isp.h hi_common_3a.h hi_mpi_ae.h hi_common_ae.h hi_mpi_awb.h hi_common_awb.h hi_common_sns.h hi_media_common.h hi_media_type.h hi_mpi_sys.h	用于定义ISP（Image Signal Processing）系统控制功能的接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_isp_ae_mpi.so libacl_isp_awb_mpi.so libacl_isp_mpi.so libacl_dvpp_mpi.so

定义接口的头文件	用途	对应的库文件
acl/media目录下： hi_mpi_vpss.h hi_media_common.h hi_media_type.h hi_mpi_sys.h	用于定义VPSS（Video Process Sub-System）图像处理功能的接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_vpss_mpi.so libacl_dvpp_mpi.so
acl/media/hi_mipi_rx.h	用于定义MIPI Rx ioctl命令字。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	-
acl/media目录下： hi_mpi_audio.h hi_common_aio.h	用于定义频输入、音频输出功能的接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_audio_mpi.so
acl/media/hi_acodec.h	用于定义音量调整的命令字。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	-

定义接口的头文件	用途	对应的库文件
acl/media目录下： hi_common_vo.h hi_mpi_vo.h	用于定义视频输出接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_vo_mpi.so
acl/media/ hi_mpi_hdmi.h	用于定义对接外设的HDMI接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_hdmi_mpi.so
acl/media/ hi_mpi_tde.h	用于定义TDE图形绘制接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	libacl_tde_mpi.so
acl/media/hifb.h	用于定义叠加图形层管理接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas A2训练系列产品，不支持引用该头文件中的接口。	-

定义接口的头文件	用途	对应的库文件
aclnn/acl_meta.h	用于定义依赖的AscendCL meta接口，通过这些元接口可构建不同数据结构，如aclTensor、aclScalar、aclIntArray等。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的接口。	libnnopbase.so
aclnn/aclnn_base.h	用于定义NN类算子公共的base接口，即aclnnInit和aclnnFinalize。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的接口。	libnnopbase.so
aclnnop/aclnn_*.h （*表示具体的算子名称） **说明** 为兼容旧版本，旧版本NN类算子头文件路径是aclnnop/level2/aclnn_*.h，但后续版本该路径会废弃，建议使用新路径aclnnop/aclnn_*.h，防止后续版本出现兼容性问题。	用于定义NN类算子的功能接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的部分接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的部分接口。	libopapi.so
acldvppop/acldvpp_base.h acldvppop/acldvpp_op_api.h	用于定义DVPP媒体数据处理类算子的功能接口。 Atlas 200/300/500 推理产品，不支持引用该头文件中的接口。 Atlas 训练系列产品，不支持引用该头文件中的接口。 Atlas 推理系列产品（Ascend 310P处理器），不支持引用该头文件中的接口。 Atlas 200/500 A2推理产品，不支持引用该头文件中的接口。	libacl_dvpp_op.so

4.3 准备开发和运行环境

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、编译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，执行编译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编译生成的应用可执行文件。

  说明

  - 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，使用CANN运行用户编译、运行时，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。

    - 运行环境安装nnrt包，则开发过程中引用对应AscendCL目录。
      - 头文件路径：CANN软件安装后文件存储路径/nnrt/latest/include/acl
      - 库文件路径：CANN软件安装后文件存储路径/nnrt/latest/lib64
    - 运行环境安装nnae包，则开发过程中引用对应AscendCL目录。
      - 头文件路径：CANN软件安装后文件存储路径/nnae/latest/include/acl
      - 库文件路径：CANN软件安装后文件存储路径/nnae/latest/lib64
  - 本文中的操作步骤（包括模型转换、编译代码、运行应用等）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

    ```
    # aclrtGetDeviceCount接口获取到的可用Device数量为1，aclrtSetDevice(0)时，索引0对应的Device ID是1
    export ASCEND_RT_VISIBLE_DEVICES=1
    ```

  - 指定多个Device ID，以下两个示例均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。

    ```
    # aclrtGetDeviceCount接口获取到的可用Device数量为3，aclrtSetDevice(0)时，索引0对应的Device ID是2
    export ASCEND_RT_VISIBLE_DEVICES=2,3,4
    # aclrtGetDeviceCount接口获取到的可用Device数量为3，aclrtSetDevice(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。

    ```
    # aclrtGetDeviceCount接口获取到的可用Device数量为2，aclrtSetDevice(0)时，索引0对应的Device ID是2
    export ASCEND_RT_VISIBLE_DEVICES=2,3,-1,5
    ```

- （可选）**通过环境变量ASCEND_CACHE_PATH、ASCEND_WORK_PATH设置AscendCL应用编译运行过程中产生的文件的落盘路径**，涉及ATC模型转换、AscendCL应用编译配置、AOE模型智能调优、性能数据采集、日志采集等功能，

落盘文件包括算子编译缓存文件、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
```

说明

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：

1. 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。
2. 执行**:wq!**命令保存文件并退出。
3. 执行**source ~/.bashrc**使环境变量生效。

5 AscendCL 初始化

本节介绍AscendCL初始化与去初始化的相关接口、注意事项，并给出示例代码。

基本原理

您必须调用aclInit接口**初始化**AscendCL，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，可向aclInit接口中传入NULL，或者可将配置文件配置为空json串（即配置文件中只有{}）。向aclInit接口中传入空指针的示例如下：

```
aclError ret = aclInit(NULL);
```

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

示例代码

您可以从**13.10.1 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，不可以直接拷贝编译运行，仅供参考。

```
// 初始化
// 此处的..表示相对路径，相对可执行文件所在的目录
// 例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char *aclConfigPath = "../src/acl.json";
aclError ret = aclInit(aclConfigPath);

// ......

// 去初始化
ret = aclFinalize();
// ......
```


6 运行时管理

6.1 运行管理资源申请与释放

本节介绍运行管理资源包括哪些、如何申请&释放这些资源，并给出示例代码。

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**4.2 AscendCL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、示例代码。

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。

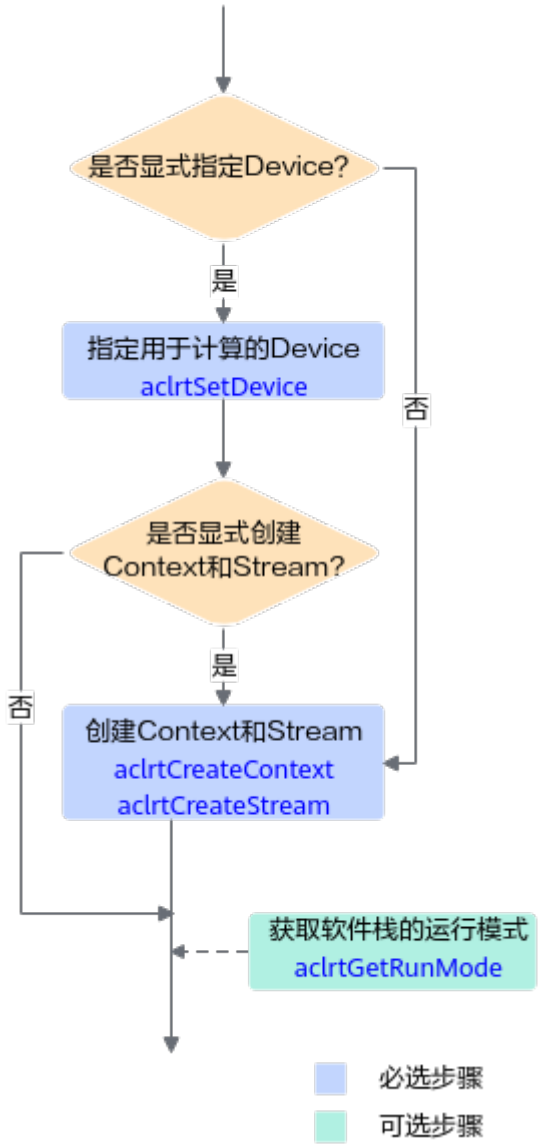
关于单进程、单线程、单Stream场景如下所示：

- 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，AscendCL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**6.3 Stream管理**。

运行管理资源申请流程

图 6-1 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aclrtSetDevice接口**显式指定用于运算的Device**。

- 调用aclrtCreateContext接口显式创建Context，调用aclrtCreateStream接口显式创建Stream。
- 如果不显式创建Context和Stream，您可以使用aclrtSetDevice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aclrtDestroyContext接口来释放。
  - 一个Device对应一个默认Stream，默认Stream不能通过aclrtDestroyStream接口来释放。默认Stream作为接口入参时，直接传NULL。
  - 默认Context、默认Stream，是在调用aclrtResetDevice接口后自动释放。

– **隐式指定用于运算的Device**。

调用aclrtCreateContext接口显式创建Context，调用aclrtCreateStream接口显式创建Stream。调用aclrtCreateContext接口显式创建Context时，传入Device ID，这时系统内部会根据该Device ID指定运行的Device。

2. （可选）调用aclrtGetRunMode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。

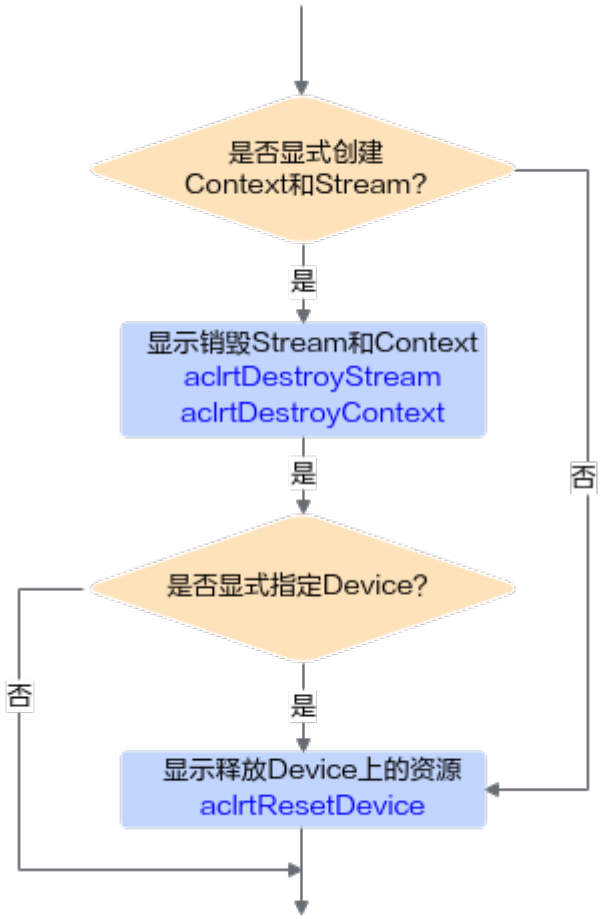
   如果查询结果为ACL_HOST，则数据传输时涉及申请Host上的内存。

   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**6.2 数据传输**。

运行管理资源释放流程

图 6-2 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aclrtDestroyStream接口释放Stream，再调用aclrtDestroyContext接口释放Context。若显式调用aclrtSetDevice接口指定运算的Device时，还需调用aclrtResetDevice接口释放Device上的资源。
- 不显式创建Context和Stream时，仅需调用aclrtResetDevice接口释放Device上的资源。

示例代码

您可以从**13.10.1 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，不可以直接拷贝编译运行，仅供参考。

```
// 初始化变量
int32_t deviceId=0 ;
aclrtContext context;
aclrtStream stream;
extern bool g_isDevice;
```



```
// =====运行管理资源申请=====
// 指定运算的Device
aclError ret = aclrtSetDevice(deviceId);

// 显式创建一个Context，用于管理Stream对象
ret = aclrtCreateContext(&context, deviceId);

// 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
ret = aclrtCreateStream(&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，后续的接口调用方式不同

aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);
// =====运行管理资源申请=====

// ......

// =====运行管理资源释放=====
ret = aclrtDestroyStream(stream);
ret = aclrtDestroyContext(context);
ret = aclrtResetDevice(deviceId);
// =====运行管理资源释放=====

// ......
```

6.2 数据传输

本节介绍数据传输的相关接口、注意事项，并给出示例代码。

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AscendCL提供的aclrtMallocHost接口申请内存，也可以用C++标准库中的new、malloc接口申请内存。
     - aclrtMallocHost会尝试申请物理地址连续的内存，后续在Host与Device数据交互时性能更优。调用aclrtMallocHost接口后、使用内存前，建议先调用aclrtMemset接口初始化内存，清除内存中的随机数。
     - 若调用malloc接口，在调用malloc接口后、使用内存前，需调用memset初始化内存，清除内存中的随机数。
   - Device上的内存，使用AscendCL提供的aclrtMalloc接口申请内存。如果涉及媒体数据处理（例如，图片解码、缩放等）时，需使用acldvppMalloc或hi_mpi_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aclrtMemcpy接口。
   - **异步**内存复制：调用aclrtMemcpyAsync接口，再调用aclrtSynchronizeStream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，也可以直接通过指针传递数据。

- 调用同步或异步内存复制接口时，支持以下类型的复制（可单击链接查看对应类型的内存复制示例代码）：
  - **Host内的数据传输**
  - **从Host到Device的数据传输**
  - **从Device到Host的数据传输**
  - **一个Device内的数据传输**
  - **两个Device间的数据传输**

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C++标准库的接口或aclrtMallocHost接口申请Host内存、使用aclrtMalloc接口申请Device内存时：

  需先调用aclrtGetRunMode接口获取软件栈的运行模式，当查询结果为ACL_HOST，则数据传输时涉及申请Host上的内存；当查询结果为ACL_DEVICE，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aclrtMallocHost接口（该接口支持申请Host或Device内存），AscendCL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aclrtGetRunMode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。

Host 内的数据传输

当前支持调用aclrtMemcpy接口执行同步Host内的内存复制任务，不支持调用aclrtMemcpyAsync接口执行异步Host内的内存复制功能，若调用aclrtMemcpyAsync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aclrtSynchronizeStream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，不可以直接拷贝编译运行，仅供参考。

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* hostPtrA = NULL;
void* hostPtrB = NULL;
aclrtMallocHost(&hostPtrA, size);
aclrtMallocHost(&hostPtrB, size);

// 2. 申请内存后，可向内存中读入数据，该自定义函数ReadFile由用户实现
ReadFile(fileName, hostPtrA, size);

// 3. 内存复制，可以选择同步
// 同步内存复制，hostPtrA表示Host上源内存地址指针，hostPtrB表示Host上目的内存地址指针，size表示内存大小
aclrtMemcpy(hostPtrB, size, hostPtrA, size, ACL_MEMCPY_HOST_TO_HOST);

// 4. 使用完内存中的数据后，需及时释放资源
aclrtFreeHost(hostPtrA);
```



```
aclrtFreeHost(hostPtrB);
// ......
```

从 Host 到 Device 的数据传输

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。

- 同步内存复制

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* hostPtrA = NULL;
void* devPtrB = NULL;
aclrtMallocHost(&hostPtrA, size);
aclrtMalloc(&devPtrB, 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ReadFile由用户实现
ReadFile(fileName, hostPtrA, size);

// 3. 内存复制，可以选择同步
// 同步内存复制，hostPtrA表示Host上源内存地址指针，devPtrB表示Device上目的内存地址指针，size表示内存大小
aclrtMemcpy(devPtrB, size, hostPtrA, size, ACL_MEMCPY_HOST_TO_DEVICE);

// 4. 使用完内存中的数据后，需及时释放资源
aclrtFreeHost(hostPtrA);
aclrtFree(devPtrB);

// ......
```

- 异步内存复制

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* hostAddr = NULL;
void* devAddr = NULL;
// 由于异步内存复制时，要求首地址64字节对齐，因此申请内存时，size需加64
aclrtMallocHost(&hostAddr, size + 64);
// 通过aclrtMalloc接口申请的内存，系统已保证内存地址64字节对齐，无需用户处理对齐的逻辑
aclrtMalloc(&devAddr, size, ACL_MEM_MALLOC_NORMAL_ONLY);

// 2. 异步内存复制
aclrtStream stream = NULL;
aclrtCreateStream(&stream);
// 获取到64字节对齐的地址
char *hostAlignAddr =(char *)hostAddr + 64 - ((uintptr_t)hostAddr % 64);
// 申请内存后，可向内存中读入数据，该自定义函数ReadFile由用户实现
ReadFile(fileName, hostAlignAddr, size);
aclrtMemcpyAsync(devAddr, size, hostAlignAddr, size, ACL_MEMCPY_HOST_TO_DEVICE, stream);
aclrtSynchronizeStream(stream);

// 3. 释放资源
aclrtDestroyStream(stream);
aclrtFreeHost(hostAddr);
aclrtFree(devAddr);

// ......
```

从 Device 到 Host 的数据传输

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。

- 同步内存复制

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* devPtrA = NULL;
void* hostPtrB = NULL;
```



```
aclrtMalloc(&devPtrA, size, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMallocHost(&hostPtrB, size);

// 2. 申请内存后，可向内存中读入数据，该自定义函数ReadFile由用户实现
ReadFile(fileName, devPtrA, size);

// 3. 内存复制，可以选择同步
// 同步内存复制，devPtrA表示Device上源内存地址指针，hostPtrB表示Host上目的内存地址指针，size表示内存大小
aclrtMemcpy(hostPtrB, size, devPtrA, size, ACL_MEMCPY_DEVICE_TO_HOST);

// 4. 使用完内存中的数据后，需及时释放资源
aclrtFree(devPtrA);
aclrtFreeHost(hostPtrB);

// ......
```

- 异步内存复制

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* hostAddr = NULL;
void* devAddr = NULL;
// 由于异步内存复制时，要求首地址64字节对齐，因此申请内存时，size需加64
aclrtMallocHost(&hostAddr, size + 64);
// 通过aclrtMalloc接口申请的内存，系统已保证内存地址64字节对齐，无需用户处理对齐的逻辑
aclrtMalloc(&devAddr, 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ReadFile由用户实现
ReadFile(fileName, devAddr, size);

// 3. 异步内存复制
aclrtStream stream = NULL;
aclrtCreateStream(&stream);
// 获取到64字节对齐的地址
char *hostAlignAddr =(char *)hostAddr + 64 - ((uintptr_t)hostAddr % 64);
aclrtMemcpyAsync(hostAlignAddr, size, devAddr, size, ACL_MEMCPY_DEVICE_TO_HOST, stream);
aclrtSynchronizeStream(stream);

// 4. 释放资源
aclrtDestroyStream(stream);
aclrtFreeHost(hostAddr);
aclrtFree(devAddr);

// ......
```

一个 Device 内的数据传输

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。

```
// 1. 申请内存
uint64_t size = 1 * 1024 * 1024;
void* devPtrA = NULL;
void* devPtrB = NULL;
aclrtMalloc(&devPtrA, size, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc(&devPtrB, 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ReadFile由用户实现
ReadFile(fileName, devPtrA, size);

// 3. 内存复制，可以选择同步或异步
// 同步内存复制，devPtrA表示Device上源内存地址指针，devPtrB表示Device上目的内存地址指针，size表示内存大小
aclrtMemcpy(devPtrB, size, devPtrA, size, ACL_MEMCPY_DEVICE_TO_DEVICE);

// 异步内存复制
// 显式创建一个Stream
aclrtStream stream;
aclrtCreateStream(&stream);
```



```
aclrtMemcpyAsync(devPtrB, size, devPtrA, size, ACL_MEMCPY_DEVICE_TO_DEVICE, stream);
aclrtSynchronizeStream(stream);

// 4. 使用完内存中的数据后，需及时释放资源
aclrtDestroyStream(stream);
aclrtFree(devPtrA);
aclrtFree(devPtrB);

// ......
```

两个 Device 间的数据传输

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，不支持该功能。

注意点说明：

- 可使用aclrtDeviceCanAccessPeer接口查询两个Device之间是否支持内存复制，若支持，需调用两次aclrtDeviceEnablePeerAccess接口使能两个Device之间的内存复制功能（例如，调用一次aclrtDeviceEnablePeerAccess接口使能Device 0到Device 1的内存复制，再调用一次aclrtDeviceEnablePeerAccess接口使能Device 1到Device 0的内存复制），再调用aclrtMemcpy接口（同步接口）或aclrtMemcpyAsync接口（异步接口）通过内存复制的方式实现数据传输。
- 当前仅支持同一个PCIe Switch内Device之间的内存复制。
- 仅支持同一个进程内的同一个线程或不同线程间的Device之间的内存复制，不支持不同进程间Device之间的内存复制。

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，不可以直接拷贝编译运行，仅供参考。

```
int main(int argc, const char *argv[])
{
    // AscendCL初始化
    auto ret = aclInit(NULL);

    int32_t canAccessPeer = 0;
    // 查询Device 0和Device 1之间是否支持内存复制
    ret = aclrtDeviceCanAccessPeer(&canAccessPeer, 0, 1);


    // 1表示支持内存复制
    if (canAccessPeer == 1) {
    // ************************************************************
    // Device 0下的操作
    ret = aclrtSetDevice(0);
    void *dev0;
    ret = aclrtMalloc(&dev0, 10, ACL_MEM_MALLOC_HUGE_FIRST_P2P);
    ret = aclrtMemset(dev0, 10, 1, 10);

    // ************************************************************
    // Device 1下的操作，使能当前Device（Device 1）到指定Device（Device 0）的内存复制，当前Device通过aclrtSetDevice接口设置，指定Device在aclrtDeviceEnablePeerAccess接口的第一个参数指定
    ret = aclrtSetDevice(1);
    ret = aclrtDeviceEnablePeerAccess(0, 0);
    void *dev1;
    ret = aclrtMalloc(&dev1, 10, ACL_MEM_MALLOC_HUGE_FIRST_P2P);
    ret = aclrtMemset(dev1, 10, 0, 10);
```



```
    // 执行复制，将Device 0上的内存数据复制到Device 1上
    ret = aclrtMemcpy(dev1, 10, dev0, 10, ACL_MEMCPY_DEVICE_TO_DEVICE);
    ret = aclrtResetDevice(1);
        // ************************************************************

    // ************************************************************
    // Device 0下的操作，等Device 1下的复制操作完成后，再调用aclrtResetDevice接口释放Device 0的资源
        ret = aclrtSetDevice(0);
    ret = aclrtResetDevice(0);
    // ************************************************************

    printf("P2P copy success\n");
    } else {
    printf("current device doesn't support p2p feature\n");
    }

    // AscendCL去初始化
    aclFinalize();
    return 0;
}
```

6.3 Stream 管理

本节介绍单Stream、多Stream的创建、销毁流程，以及多Stream同步等待的流程。

在AscendCL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NULL。

AscendCL提供以下几种Stream管理机制：

- **单线程单Stream**
- **单线程多Stream**
- **多线程多Stream**

其中，多Stream场景下的同步等待流程请参见**Stream间任务的同步等待接口调用流程及示例代码**。

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，不可以直接拷贝编译运行，仅供参考。

```
#include "acl/acl.h"
// ......
int32_t deviceId = 0 ;
aclrtContext context;

/ 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aclrtSetCurrentContext接口设置当前线程的Context
 aclrtCreateContext(&context, deviceId);

// 显式创建一个Stream
aclrtStream stream;
aclrtCreateStream(&stream);
// 调用触发任务的接口，传入stream参数
aclrtMemcpyAsync(dstPtr, dstSize, srcPtr, srcSize, ACL_MEMCPY_HOST_TO_DEVICE, stream);
// 调用aclrtSynchronizeStream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。
aclrtSynchronizeStream(stream);

// Stream使用结束后，显式销毁Stream
```



```
aclrtDestroyStream(stream);
aclrtDestroyContext(context);
// ......
```

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，不可以直接拷贝编译运行，仅供参考。

```
#include "acl/acl.h"
// ......
int32_t deviceId = 0 ;
uint32_t modelId1 = 0;
uint32_t modelId2 = 1;
aclrtContext context;
aclrtStream stream1;
aclrtStream stream2;

// 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aclrtSetCurrentContext接口设置当前线程的Context
 aclrtCreateContext(&context, deviceId);

// 创建stream1
aclrtCreateStream(&stream1);
// 调用触发任务的接口，例如异步模型推理，任务下发在stream1
aclmdlDataset *input1;
aclmdlDataset *output1;
aclmdlExecuteAsync(modelId1, input1, output1, stream1);

// 创建stream2
aclrtCreateStream(&stream2);
// 调用触发任务的接口，例如异步模型推理， 任务下发在stream2
aclmdlDataset *input2;
aclmdlDataset *output2;
aclmdlExecuteAsync(modelId2, input1, output2, stream2);

// 流同步
aclrtSynchronizeStream(stream1);
aclrtSynchronizeStream(stream2);

// 释放资源
aclrtDestroyStream(stream1);
aclrtDestroyStream(stream2);
aclrtDestroyContext(context);
// ....
```

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，不可以直接拷贝编译运行，仅供参考。

```
#include "acl/acl.h"
// ......
void runThread(aclrtStream stream) {
    int32_t deviceId =0 ;
    aclrtContext context;

    // 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aclrtSetCurrentContext接口设置当前线程的Context
    aclrtCreateContext(&context, deviceId);

    // 显式创建一个Stream
    aclrtStream stream;
    aclrtCreateStream(&stream);
    // 调用触发任务的接口
    // ....
```



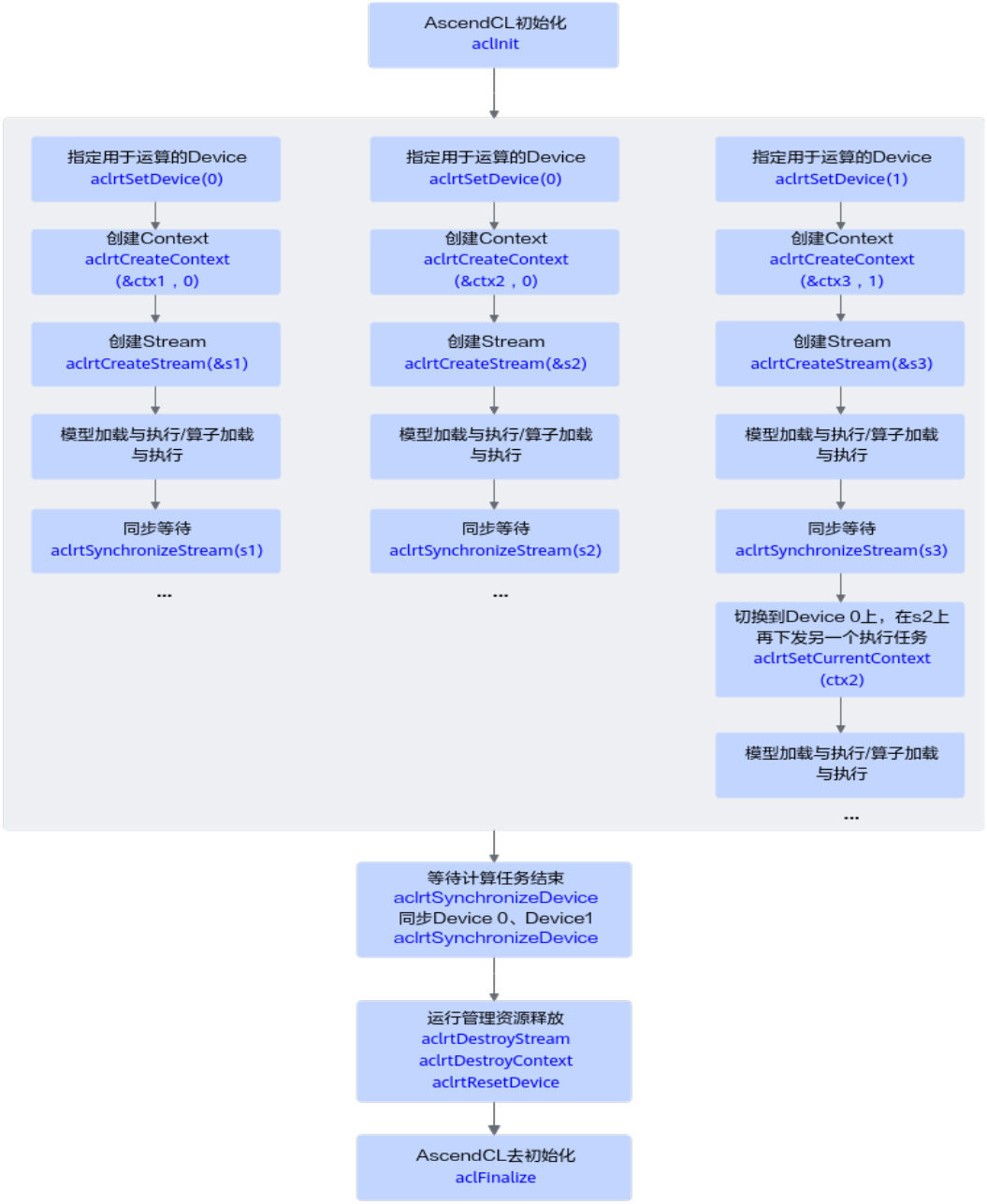
```
    // 释放资源
    aclrtDestroyStream(stream);
    aclrtDestroyContext(context);

}

aclrtStream stream1;
aclrtStream stream2;
// 创建2个线程，每个线程对应一个Stream
std::thread t1(runThread, stream1);
std::thread t2(runThread, stream2);
// 显式调用join函数确保结束线程
t1.join();
t2.join();
```

6.4 多 Device 切换

本节介绍多Device切换场景的关键接口及接口调用流程。

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**4.2 AscendCL接口调用流程**以及本节中的说明。

图 6-3 同步等待流程_多 Device 场景



- 在多Device时，利用Context切换（调用aclrtSetCurrentContext接口）来切换Device，比使用aclrtSetDevice接口效率高。
- 调用aclrtSynchronizeDevice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**8 模型推理**。
- 算子加载与执行的流程请参见**7 单算子调用**。

6.5 同步等待

本节介绍Device、Stream、Event在异步场景下的使用示例及关键接口。

同步机制

AscendCL提供以下同步机制：

- **Event的同步等待示例代码**：调用aclrtSynchronizeEvent接口，阻塞应用程序运行，等待Event完成。
- **Stream内任务的同步等待示例代码**：调用aclrtSynchronizeStream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间任务的同步等待示例代码**：调用aclrtStreamWaitEvent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**Stream间任务的同步等待接口调用流程及示例代码**。
- **Device的同步等待示例代码**：调用aclrtSynchronizeDevice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**6.4 多Device切换**。

Event 的同步等待示例代码

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。

```
#include "acl/acl.h"
// ......
// 创建一个Event
aclrtEvent event;
aclrtCreateEvent(&event);

// 显式创建一个Stream
aclrtStream stream;
aclrtCreateStream(&stream);

// stream末尾添加了一个event
aclrtRecordEvent(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
aclrtSynchronizeEvent(event);

// 显式销毁资源
aclrtDestroyStream(stream);
aclrtDestroyEvent(event);
// ......
```

Stream 内任务的同步等待示例代码

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。

```
#include "acl/acl.h"
// ......
// 显式创建一个Stream
aclrtStream stream;
aclrtCreateStream(&stream);
// 调用触发任务的接口，传入stream参数
aclrtMemcpyAsync(dstPtr, dstSize, srcPtr, srcSize, ACL_MEMCPY_HOST_TO_DEVICE, stream);
// 调用aclrtSynchronizeStream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。
aclrtSynchronizeStream(stream);

// Stream使用结束后，显式销毁Stream
aclrtDestroyStream(stream);
// ......
```

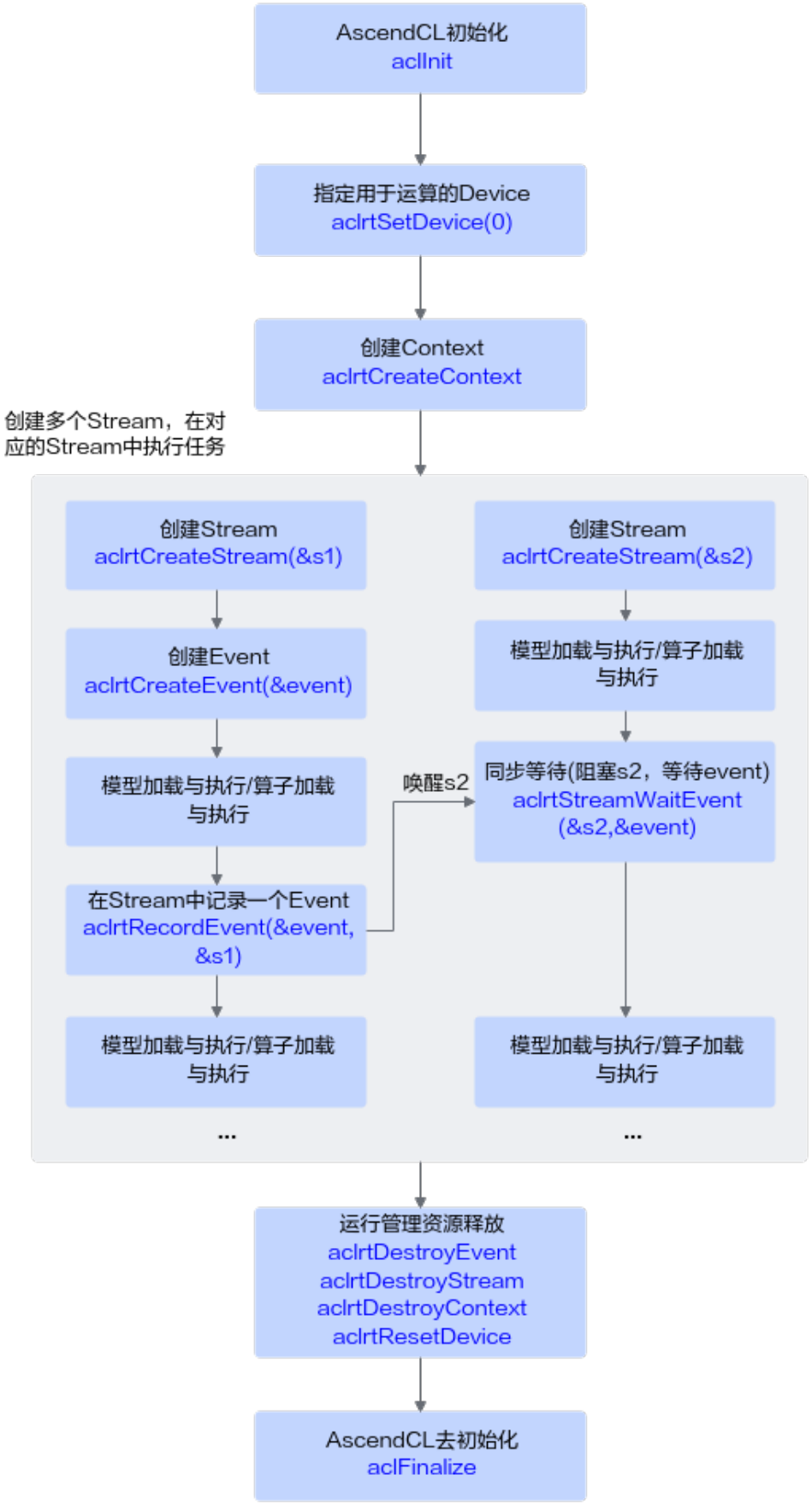

Stream 间任务的同步等待接口调用流程及示例代码

多Stream之间任务的同步等待可以利用Event实现，调用aclrtStreamWaitEvent接口阻塞指定Stream的运行，直到指定的Event完成。需在调用aclrtStreamWaitEvent接口前，先调用aclrtRecordEvent接口。

模型加载与执行的流程请参见**8 模型推理**。

算子加载与执行的流程请参见**7 单算子调用**。

图 6-4 同步等待流程_多 Stream 场景



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。

```
#include "acl/acl.h"
// ......
// 创建一个Event
aclrtEvent event;
aclrtCreateEvent(&event);

// 创建stream1
aclrtStream s1;
aclrtCreateStream(&s1);
// 创建stream2
aclrtStream s2;
aclrtCreateStream(&s2);
// 在s1末尾添加了一个event
aclrtRecordEvent(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
aclrtStreamWaitEvent(s2, event);

// 显式销毁资源
aclrtDestroyStream(s2);
aclrtDestroyStream(s1);
aclrtDestroyEvent(event);
// ......
```

Device 的同步等待示例代码

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。

```
#include "acl/acl.h"
// ......
// 指定device
aclrtSetDevice(0);

// 创建context
aclrtContext ctx;
aclrtCreateContext(&ctx, 0);

// 创建stream
aclrtStream stream;
aclrtCreateStream(&stream);

// 阻塞应用程序运行，直到正在运算中的Device完成运算
aclrtSynchronizeDevice();

// 资源销毁
aclrtDestroyStream(stream);
aclrtDestroyContext(ctx);
aclrtResetDevice(0);
```


7 单算子调用

7.1 单算子调用基础知识

本节介绍基于AscendCL接口调用单个算子的使用场景、开发流程等。

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AscendCL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。

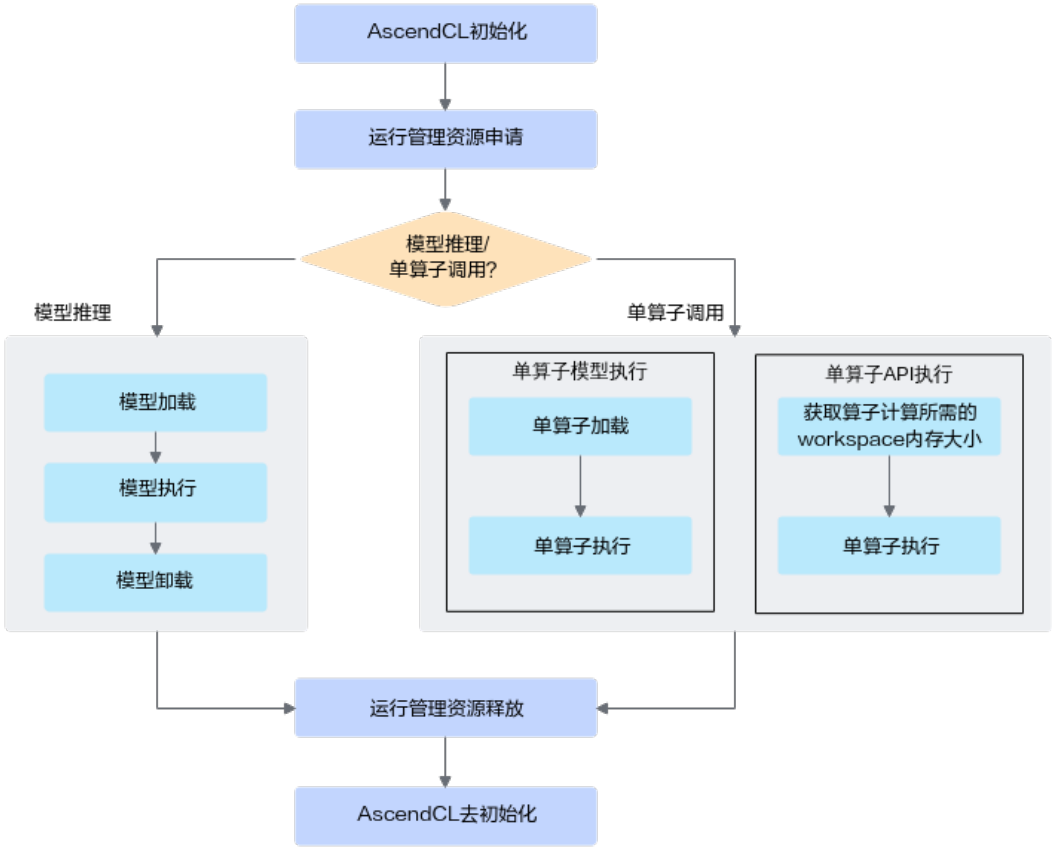
另外，自定义的算子，也可以通过单算子调用的方式来验证算子的功能。

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，先找出基本的共同点、不同点。

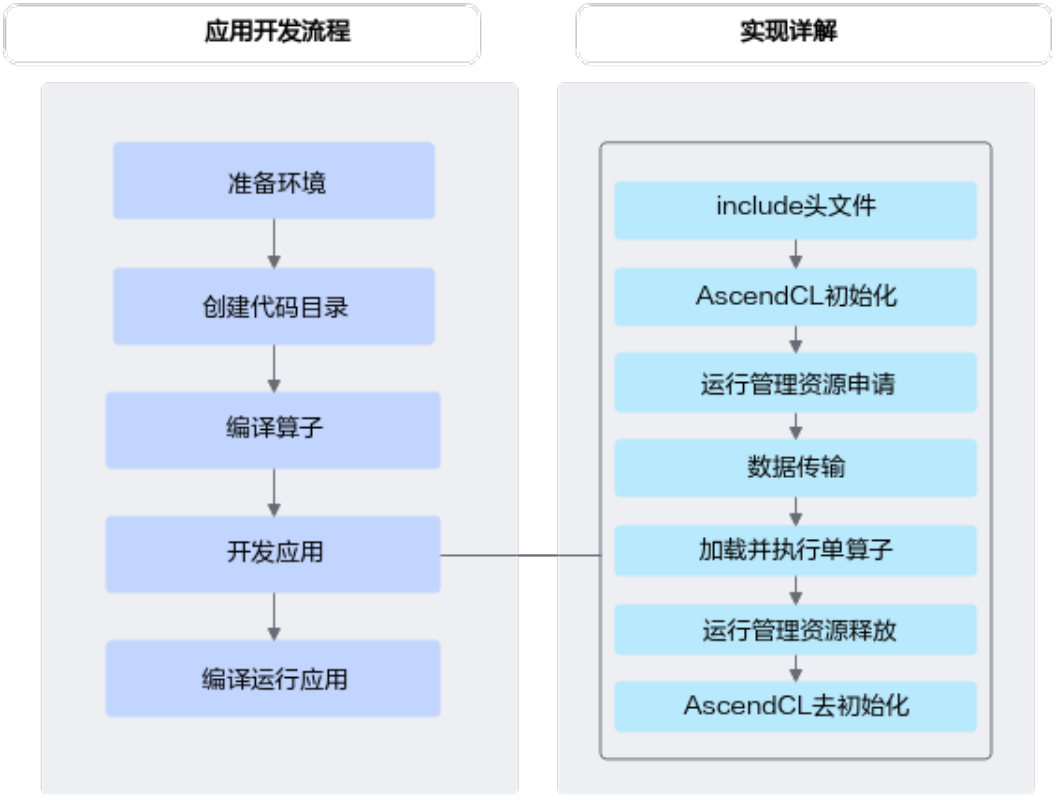
- **共同点**:
  - 不管是模型推理，还是单算子调用，**都需要**AscendCL初始化和去初始化、运行管理资源申请和释放。
  - 不管是模型推理，还是单算子调用，**都涉及**执行的步骤，但是要注意，两者执行是调用不同的AscendCL接口。
- **不同点**:
  - 模型推理涉及**模型卸载**的步骤，单算子调用不涉及。

图 7-1 单算子调用与模型推理的流程对比



单算子调用功能开发流程

图 7-2 开发流程



1. **准备环境**。
   请参见**4.3 准备开发和运行环境**。
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，供参考：

```
├App名称
├── op_model            // 该目录下存放编译算子的算子描述文件
│   ├── xxx.json

├── data
│   ├── xxxxxx          // 测试数据

├── inc                 // 该目录下存放声明函数的头文件
│   ├── xxx.h

├── out                 // 该目录下存放输出结果

├── src
│   ├── xxx.json        // 系统初始化的配置文件
│   ├── CMakeLists.txt  // 编译脚本
│   ├── xxx.cpp         // 实现文件
```

3. **（可选）编译算子**。
   若基于“**7.3 单算子API执行**”方式调用算子，则无需编译算子。
   若基于“**7.4 单算子模型执行**”方式调用算子，则需编译算子，编译算子有以下两种方式：

- 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AscendCL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》。
- 也可以调用AscendCL提供的编译算子接口

  该种方式，直接调用AscendCL接口编译、执行算子。

说明

关于单算子模型执行、单算子API执行的区别及详细的接口调用流程请参见**7.2 单算子调用流程**。

4. **开发应用**。

   依赖的头文件和库文件的说明请参见**调用接口依赖的头文件和库文件说明**。

   单算子调用的流程请参见**7.2 单算子调用流程**及相关的示例代码。
5. **编译运行应用**，请参见**11 应用编译&运行**。

7.2 单算子调用流程

本节介绍调用单算子的两种方式、以及这两种方下的接口调用流程。

开发应用时，如果涉及执行单个算子，请先参见**4.2 AscendCL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**7.4.4 执行动态Shape算子示例代码（注册算子选择器）**。

单算子调用方式：单算子模型执行、单算子 API 执行

- **单算子API执行**：**基于C语言的API执行算子**，无需提供IR（Intermediate Representation）定义，调用单算子API执行下的算子接口，针对每个算子，都需要依次调用aclnn*Xxx*GetWorkspaceSize接口获取算子执行需要的workspace内存大小、调用aclnn*Xxx*接口执行算子。
- **单算子模型执行**：**基于图IR执行算子**，先编译算子（例如，使用ATC工具将Ascend IR定义的单算子描述文件编译成算子om模型文件），再调用AscendCL接口加载算子模型（例如aclopSetModelDir接口），最后调用AscendCL接口执行算子（例如aclopExecuteV2接口）。

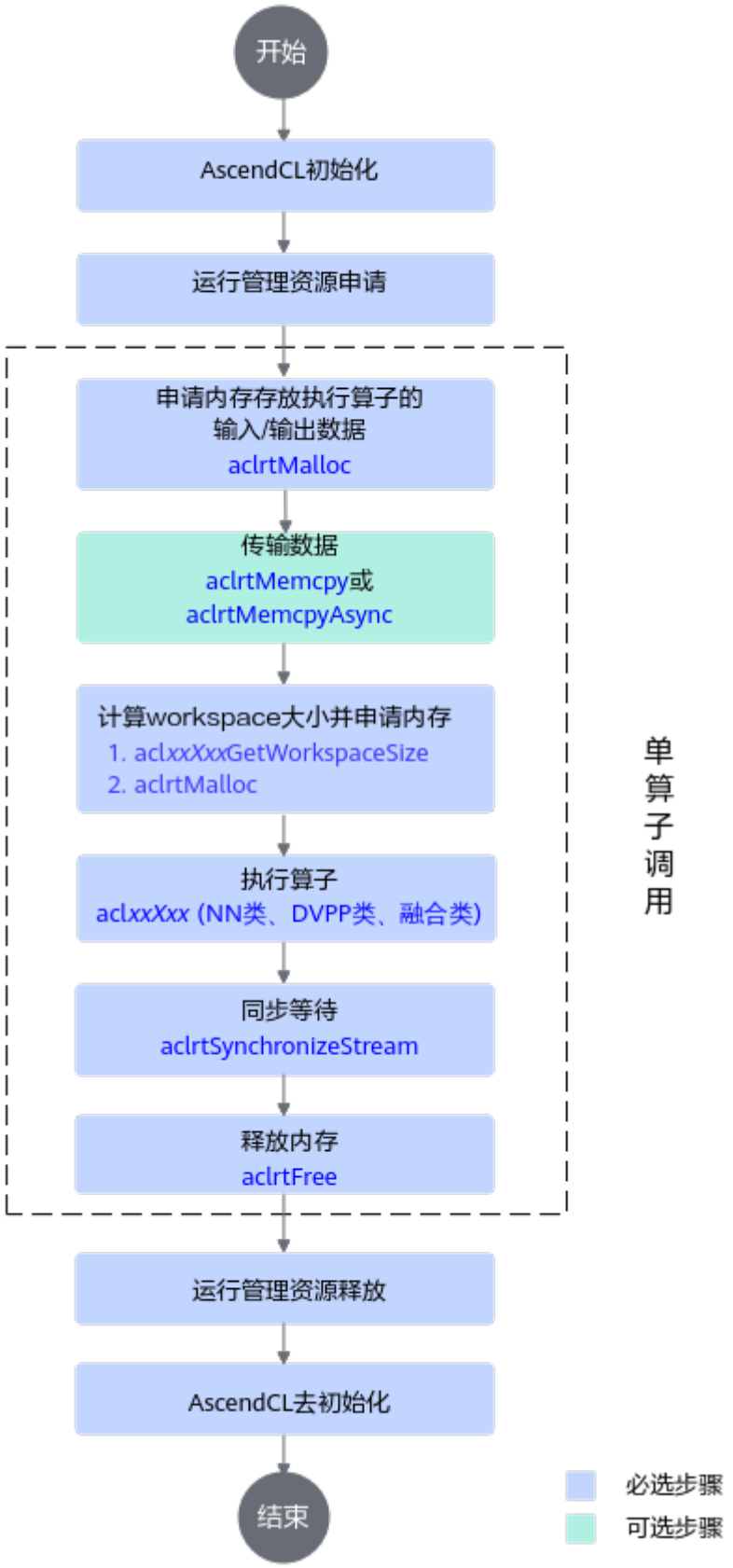
昇腾AI处理器对两种单算子调用方式的支持度如下表所示。

-	单算子API执行	单算子模型执行
Atlas 200/300/500 推理产品	x	√

-	**单算子API执行**	**单算子模型执行**
Atlas 200/500 A2推理产品	√（部分算子支持）	√
Atlas 训练系列产品	√	√
Atlas A2训练系列产品	√	√
Atlas 推理系列产品（Ascend 310P处理器）	√（部分算子支持）	√

单算子 API 执行接口调用流程

图 7-3 单算子 API 执行接口调用流程



关键接口的说明如下：

1. **AscendCL初始化**。

   调用aclInit接口实现初始化AscendCL。

2. **运行管理资源申请**。

   依次申请运行管理资源：**Stream**、**Context**、**Device**。具体流程，请参见**6.1 运行管理资源申请与释放**。

3. **数据内存申请和传输。**

   a. 调用aclrtMalloc接口申请Device上的内存，存放待执行算子的输入、输出数据。

   b. 调用aclCreateTensor、aclCreateIntArray等接口构造算子的输入、输出数据，如aclTensor、aclIntArray等，详细接口请参见公共接口。

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

4. **计算workspace并执行算子**。

   a. 调用acl*xxXxx*GetWorkspaceSize接口获取算子入参，并计算该算子执行流程需要多少的workspace内存。

   b. 调用aclrtMalloc接口，根据workspaceSize大小申请Device侧内存。

   c. 调用acl*xxXxx*接口执行计算并得到结果。

   说明

   单算子API执行的接口一般定义为“两段式接口”，即acl*xxXxx*GetWorkspaceSize和acl*xxXxx*，详细的接口功能和使用方法请参见单算子API执行章节。

5. 调用aclrtSynchronizeStream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aclrtFree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aclrtMemcpy接口（同步接口）或aclrtMemcpyAsync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。

7. **运行管理资源释放**。

   a. 调用aclDestroyTensor、aclDestroyIntArray等接口释放算子的输入、输出，相关接口请参见公共接口。

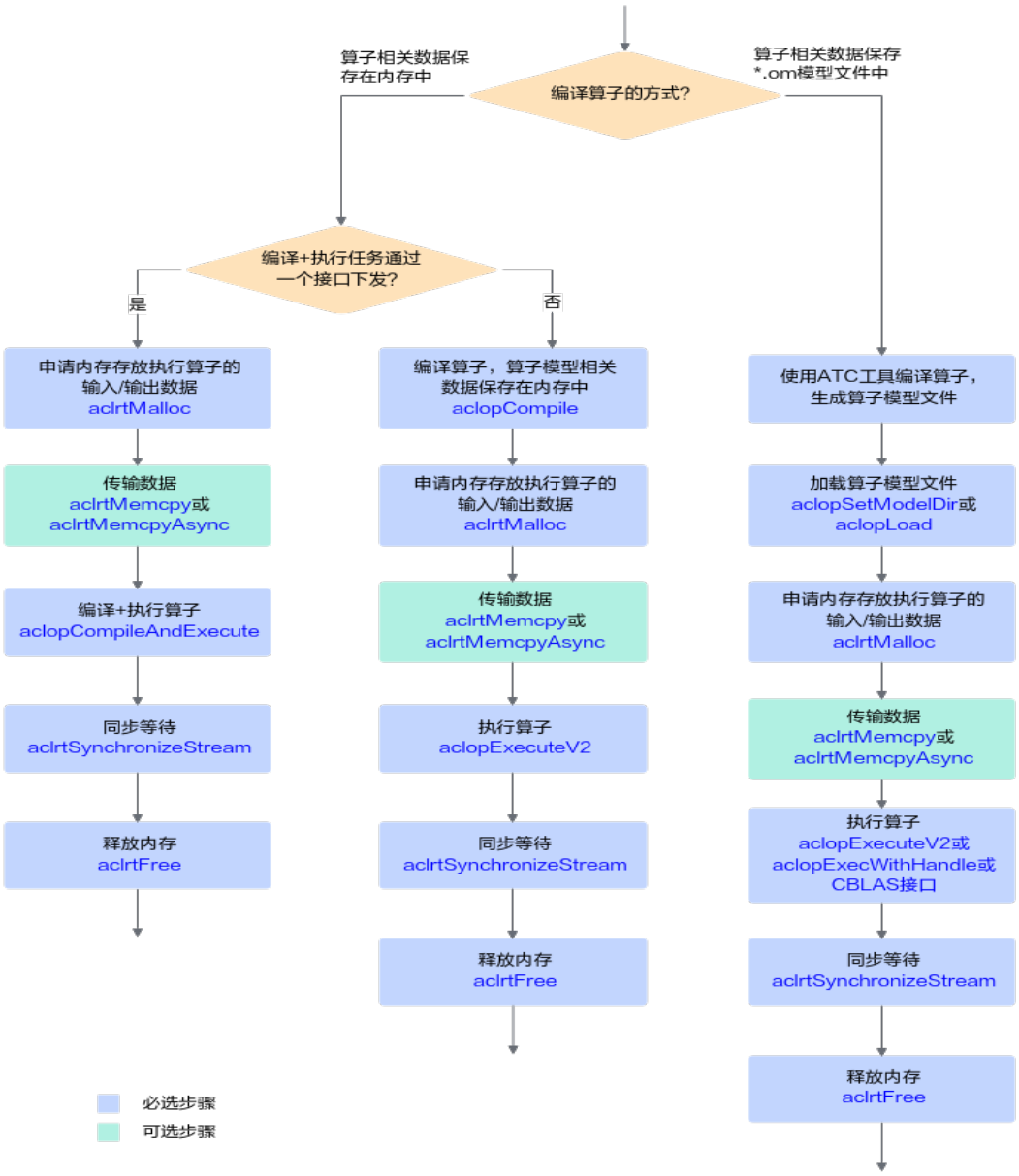
   b. 所有数据释放后，需要依次释放运行管理资源：**Stream**、**Context**、**Device**。具体流程，请参见**6.1 运行管理资源申请与释放**。

8. **AscendCL去初始化**。

   调用aclFinalize接口实现AscendCL去初始化。

单算子模型执行接口调用流程

图 7-4 单算子模型执行接口调用流程



关键接口的说明如下：

1. **编译算子**。

   根据算子编译的方式，可分为以下两种：

   - 编译算子后，算子相关数据保存在*.om模型文件中

     该种方式下编译算子，需使用ATC工具，详细描述请参见《ATC工具使用指南》，将单算子定义文件（*.json）编译成适配昇腾AI处理器的离线模型（*.om文件）。

     编译算子后，依次进行**2**、**3**、**4**、**5**、**6**、**7**。

   - 编译算子后，算子相关数据保存在内存中

该种方式下编译算子，需调用AscendCL提供的接口，根据不同场景调用不同的接口：

- 对于同一个算子，编译一次，多次执行的场景，建议调用aclopCompile接口编译算子。编译算子后，依次进行3、4、5、6、7。
- 对于编译算子、执行算子次数相同的场景，建议先执行3，再调用aclopCompileAndExecute接口编译算子。编译算子后，再依次进行6、7。

2. **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aclopSetModelDir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aclopLoad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。

3. 调用aclrtMalloc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aclrtMemcpy接口（同步接口）或aclrtMemcpyAsync接口（异步接口）通过内存复制的方式实现数据传输。

4. 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aclopInferShape接口、aclGetTensorDescNumDims接口、aclGetTensorDescDimV2接口、aclGetTensorDescDimRange等接口，推导或预估算子的输出Shape，作为算子执行接口aclopExecuteV2的输入。

5. **执行算子**。

   - 对于被封装成AscendCL接口的算子（参见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aclblasGemmEx接口（封装GEMM算子）、aclopCast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aclblasCreateHandleForGemmEx接口、aclopCreateHandleForCast接口等创建handle后，还需要调用aclopExecWithHandle接口执行算子。
   - 对于未被封装成AscendCL接口的算子，目前支持以下两种执行方式：
     - 不以handle方式执行算子，调用aclopExecuteV2接口执行算子。
     - 以handle方式执行算子，调用aclopCreateHandle接口创建handle，再调用aclopExecWithHandle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，但该方式不支持动态Shape算子，且Handle使用结束后，需调用aclopDestroyHandle接口释放。

6. 调用aclrtSynchronizeStream接口**阻塞应用运行**，直到指定Stream中的所有任务都完成。
7. 调用aclrtFree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aclrtMemcpy接口（同步接口）或aclrtMemcpyAsync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。

7.3 单算子 API 执行

7.3.1 调用 NN 类算子接口示例代码

本节介绍基于单算子API执行的方式调用NN类算子的示例代码。

基本原理

NN（Neural Network）类算子主要实现数学基础运算（如加、减、乘、除等）以及CNN相关的操作（如卷积、池化、激活函数）等，详细的算子API介绍参见单算子API执行，接口调用流程参见**单算子API执行接口调用流程**。

单算子API执行的算子接口一般定义为“两段式接口”，其中NN类算子接口示例如下：

```
aclnnStatus aclnnXxxGetWorkspaceSize(const aclTensor *src, ..., aclTensor *out, ..., uint64_t workspaceSize, aclOpExecutor **executor);
aclnnStatus aclnnXxx(void* workspace, int64 workspaceSize, aclOpExecutor* executor, aclrtStream stream);
```

其中aclnn*Xxx*GetWorkspaceSize为第一段接口，主要用于计算本次NN类算子API调用计算过程中需要多少的workspace内存。获取到本次计算需要的workspace大小后，按照workspaceSize大小申请Device侧内存，然后调用第二段接口aclnn*Xxx执行计算*。

示例代码

本章以**Add算子**调用为例，介绍编写算子调用的代码逻辑。其他NN类算子的调用逻辑与Add算子大致一样，请根据实际情况自行修改代码。

> **说明**
>
> Add算子暂不支持在Atlas 200/500 A2推理产品、Atlas 推理系列产品（Ascend 310P处理器）上调用，此处仅提供代码逻辑参考，请根据实际算子支持的产品型号进行调用，具体参见单算子API执行。

已知Add算子实现了张量加法运算，计算公式为：$y=x_1+\alpha x x_2$。您可以获取如下示例代码，并将文件命名为“**test_add.cpp**”，代码如下：

```
#include <iostream>
#include <vector>
#include "acl/acl.h"
#include "aclnnop/aclnn_add.h"

#define CHECK_RET(cond, return_expr) \
    do {                             \
        if (!(cond)) {               \
        return_expr;                 \
        }                            \
    } while (0)

#define LOG_PRINT(message, ...)     \
```

```
    do {                            \
        printf(message, ##__VA_ARGS__); \
    } while (0)

int64_t GetShapeSize(const std::vector<int64_t>& shape) {
    int64_t shapeSize = 1;
    for (auto i : shape) {
        shapeSize *= i;
  }
    return shapeSize;
}

int Init(int32_t deviceId, aclrtContext* context, aclrtStream* stream) {
    // 固定写法，acl初始化
    auto ret = aclrtSetDevice(deviceId);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtSetDevice failed. ERROR: %d\n", ret); return ret);
    ret = aclrtCreateContext(context, deviceId);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtCreateContext failed. ERROR: %d\n", ret); return ret);
    ret = aclrtSetCurrentContext(*context);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtSetCurrentContext failed. ERROR: %d\n", ret); return
ret);
    ret = aclrtCreateStream(stream);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtCreateStream failed. ERROR: %d\n", ret); return ret);

    ret = aclInit(nullptr);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclInit failed. ERROR: %d\n", ret); return ret);
    return 0;
}

template <typename T>
int CreateAclTensor(const std::vector<T>& hostData, const std::vector<int64_t>& shape, void** deviceAddr,
                    aclDataType dataType, aclTensor** tensor) {
    auto size = GetShapeSize(shape) * sizeof(T);
    // 调用aclrtMalloc申请device侧内存
    auto ret = aclrtMalloc(deviceAddr, size, ACL_MEM_MALLOC_HUGE_FIRST);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtMalloc failed. ERROR: %d\n", ret); return ret);

    // 调用aclrtMemcpy将host侧数据拷贝到device侧内存上
    ret = aclrtMemcpy(*deviceAddr, size, hostData.data(), size, ACL_MEMCPY_HOST_TO_DEVICE);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtMemcpy failed. ERROR: %d\n", ret); return ret);

    // 计算连续tensor的strides
    std::vector<int64_t> strides(shape.size(), 1);
    for (int64_t i = shape.size() - 2; i >= 0; i--) {
      strides[i] = shape[i + 1] * strides[i + 1];
    }

    // 调用aclCreateTensor接口创建aclTensor
    *tensor = aclCreateTensor(shape.data(), shape.size(), dataType, strides.data(), 0,
aclFormat::ACL_FORMAT_ND,
                              shape.data(), shape.size(), *deviceAddr);
    return 0;
}

int main() {
    // 1.(固定写法)device/context/stream初始化
    // 根据自己的实际device填写deviceId
    int32_t deviceId = 0;
    aclrtContext context;
    aclrtStream stream;
    auto ret = Init(deviceId, &context, &stream);
    // check根据自己的需要处理
    CHECK_RET(ret == 0, LOG_PRINT("Init acl failed. ERROR: %d\n", ret); return ret);
    // 2.构造输入与输出，需要根据API的接口自定义构造
    std::vector<int64_t> selfShape = {4, 2};
    std::vector<int64_t> otherShape = {4, 2};
    std::vector<int64_t> outShape = {4, 2};
    void* selfDeviceAddr = nullptr;
    void* otherDeviceAddr = nullptr;
```



```
    void* outDeviceAddr = nullptr;
    aclTensor* self = nullptr;
    aclTensor* other = nullptr;
    aclScalar* alpha = nullptr;
    aclTensor* out = nullptr;
    std::vector<float> selfHostData = {0, 1, 2, 3, 4, 5, 6, 7};
    std::vector<float> otherHostData = {1, 1, 1, 2, 2, 2, 3, 3};
    std::vector<float> outHostData = {0, 0, 0, 0, 0, 0, 0, 0};
    float alphaValue = 1.2f;
    // 创建self aclTensor
    ret = CreateAclTensor(selfHostData, selfShape, &selfDeviceAddr, aclDataType::ACL_FLOAT, &self);
    CHECK_RET(ret == ACL_SUCCESS, return ret);
    // 创建other aclTensor
    ret = CreateAclTensor(otherHostData, otherShape, &otherDeviceAddr, aclDataType::ACL_FLOAT, &other);
    CHECK_RET(ret == ACL_SUCCESS, return ret);
    // 创建alpha aclScalar
    alpha = aclCreateScalar(&alphaValue, aclDataType::ACL_FLOAT);
    CHECK_RET(alpha != nullptr, return ret);
    // 创建out aclTensor
    ret = CreateAclTensor(outHostData, outShape, &outDeviceAddr, aclDataType::ACL_FLOAT, &out);
    CHECK_RET(ret == ACL_SUCCESS, return ret);

    // 3.调用CANN算子库API
    uint64_t workspaceSize = 0;
    aclOpExecutor* executor;
    // 调用aclnnAdd第一段接口
    ret = aclnnAddGetWorkspaceSize(self, other, alpha, out, &workspaceSize, &executor);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclnnAddGetWorkspaceSize failed. ERROR: %d\n", ret);
return ret);
    // 根据第一段接口计算出的workspaceSize申请device内存
    void* workspaceAddr = nullptr;
    if (workspaceSize > 0) {
        ret = aclrtMalloc(&workspaceAddr, workspaceSize, ACL_MEM_MALLOC_HUGE_FIRST);
        CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("allocate workspace failed. ERROR: %d\n", ret); return
ret;);
    }
    // 调用aclnnAdd第二段接口
    ret = aclnnAdd(workspaceAddr, workspaceSize, executor, stream);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclnnAdd failed. ERROR: %d\n", ret); return ret);
    // 4.(固定写法)同步等待任务执行结束
    ret = aclrtSynchronizeStream(stream);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("aclrtSynchronizeStream failed. ERROR: %d\n", ret); return
ret);
    // 5.获取输出的值，将device侧内存上的结果拷贝至host侧，需要根据具体API的接口定义修改
    auto size = GetShapeSize(outShape);
    std::vector<float> resultData(size, 0);
    ret = aclrtMemcpy(resultData.data(), resultData.size() * sizeof(resultData[0]), outDeviceAddr, size *
sizeof(float),
                      ACL_MEMCPY_DEVICE_TO_HOST);
    CHECK_RET(ret == ACL_SUCCESS, LOG_PRINT("copy result from device to host failed. ERROR: %d\n",
ret); return ret);
    for (int64_t i = 0; i < size; i++) {
        LOG_PRINT("result[%ld] is: %f\n", i, resultData[i]);
    }

    // 6.释放aclTensor和aclScalar，需要根据具体API的接口定义修改
    aclDestroyTensor(self);
    aclDestroyTensor(other);
    aclDestroyScalar(alpha);
    aclDestroyTensor(out);

    // 7.释放device资源，需要根据具体API的接口定义修改
    aclrtFree(selfDeviceAddr);
    aclrtFree(otherDeviceAddr);
    aclrtFree(outDeviceAddr);
    if (workspaceSize > 0) {
        aclrtFree(workspaceAddr);
    }
    aclrtDestroyStream(stream);
```



```
    aclrtDestroyContext(context);
    aclrtResetDevice(deviceId);
    aclFinalize();
    return 0;
}
```

CMakeLists 文件

本章以**Add算子编译脚本**为例，介绍如何编写算子编译脚本CMakeLists.txt。其他NN类算子的编译脚本与Add算子大致一样，请根据实际情况自行修改脚本。

```
# Copyright (c) Huawei Technologies Co., Ltd. 2019. All rights reserved.

# CMake lowest version requirement
cmake_minimum_required(VERSION 3.14)

# 设置工程名
project(ACLNN_EXAMPLE)

# Compile options
add_compile_options(-std=c++11)

# 设置编译选项
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY  "./bin")
set(CMAKE_CXX_FLAGS_DEBUG "-fPIC -O0 -g -Wall")
set(CMAKE_CXX_FLAGS_RELEASE "-fPIC -O2 -Wall")

# 设置可执行文件名（如opapi_test），并指定待运行算子文件*.cpp所在目录
add_executable(opapi_test
               test_add.cpp)

# 设置ASCEND_PATH（CANN包目录）和INCLUDE_BASE_DIR（头文件目录）
if(NOT "$ENV{ASCEND_CUSTOM_PATH}" STREQUAL "")
    set(ASCEND_PATH $ENV{ASCEND_CUSTOM_PATH})
else()
    set(ASCEND_PATH "/usr/local/Ascend/")
endif()
set(INCLUDE_BASE_DIR "${ASCEND_PATH}/include")
include_directories(
    ${INCLUDE_BASE_DIR}
    ${INCLUDE_BASE_DIR}/aclnn
)

# 设置链接的库文件路径
target_link_libraries(opapi_test PRIVATE
                      ${ASCEND_PATH}/lib64/libascendcl.so
                      ${ASCEND_PATH}/lib64/libnnopbase.so
                      ${ASCEND_PATH}/lib64/libopapi.so)

# 可执行文件在CMakeLists文件所在目录的bin目录下
install(TARGETS opapi_test DESTINATION ${CMAKE_RUNTIME_OUTPUT_DIRECTORY})
```

编译与运行

说明

- 在编译与运行之前，请确保应用开发环境已就绪，具体请参见**4.3 准备开发和运行环境**。
- 更多关于编译和运行的详细操作，可参见**应用调试**章节中“编译及运行应用”内容。

步骤1 根据前文**示例代码**和**CMakeLists文件**，提前准备好算子的调用代码（*.cpp）和编译脚本（CMakeLists.txt）。

步骤2 配置环境变量。

安装CANN软件后，使用CANN运行用户（如HwHiAiUser）登录环境，执行如下命令设置环境变量。其中${install_path}为CANN软件安装后文件存储路径，请根据实际情况替换该路径。

```
source ${install_path}/set_env.sh
```

步骤3 编译并运行。

1. 进入CMakeLists.txt所在目录，执行如下命令，新建build目录存放生成的编译文件。
   ```
   mkdir -p build
   ```
2. 进入CMakeLists.txt所在目录，执行cmake命令编译，再执行make命令生成可执行文件。
   ```
   cmake ./ -DCMAKE_CXX_COMPILER=g++ -DCMAKE_SKIP_RPATH=TRUE
   make
   ```
   编译成功后，会在当前目录的bin目录下生成opapi_test可执行文件。
3. 进入bin目录，运行可执行文件opapi_test。
   ```
   ./opapi_test
   ```
   以Add算子的运行结果为例，运行后的结果如下：
   ```
   (base) [root@milan8p build]# ./opapi_test
   result[0] is: 1.200000
   result[1] is: 2.200000
   result[2] is: 3.200000
   result[3] is: 5.400000
   result[4] is: 6.400000
   result[5] is: 7.400000
   result[6] is: 9.600000
   result[7] is: 10.600000
   (base) [root@milan8p build]#
   ```

----结束

7.4 单算子模型执行

7.4.1 调用 CBLAS 接口执行算子示例代码

本节介绍基于单算子模型执行的方式调用CBLAS算子的关键接口、示例代码。

基本原理

接口调用流程，请参见**7.2 单算子调用流程**。

目前，AscendCL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AscendCL的CBALS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aclblasGemmEx接口（封装GEMM算子）、aclopCast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aclblasCreateHandleForGemmEx接口、aclopCreateHandleForCast接口等创建handle后，还需要调用aclopExecWithHandle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，但该方式不支持动态Shape算子，且Handle使用结束后，需调用aclopDestroyHandle接口释放。

示例代码

本章以aclblasGemmEx接口为例，该接口封装的是GEMM算子，该接口中矩阵乘的计算公式为：C = αAB + βC。您可以单击**acl_execute_gemm**，查看样例。

调用CBLAS接口（封装GEMM算子）分为以下几步：

1. 准备GEMM算子的模型文件。
   a. 构造GEMM算子的描述文件（*.json文件，描述输入输出Tensor描述、算子属性等）。

      GEMM算子的描述文件示例如下：

```
[
{
  "op": "GEMM",
  "input_desc": [
    {
      "format": "ND",
      "shape": [16, 16],
      "type": "float16"
    },
    {
      "format": "ND",
      "shape": [16, 16],
      "type": "float16"
    },
    {
      "format": "ND",
      "shape": [16, 16],
      "type": "float16"
    },
    {
      "format": "ND",
      "shape": [],
      "type": "float16"
    },
    {
      "format": "ND",
      "shape": [],
      "type": "float16"
    }
  ],
  "output_desc": [
    {
      "format": "ND",
      "shape": [16, 16],
      "type": "float16"
    }
  ],
  "attr": [
  {
    "name": "transpose_a",
    "type": "bool",
    "value": false
  },
  {
```



```
    "name": "transpose_b",
    "type": "bool",
    "value": false
    }
  ]
}
]
```

b. 借助ATC工具，将该算子描述文件编译成单算子模型文件（*.om文件），再分别调用AscendCL接口加载om模型文件、执行算子。

ATC工具的命令示例如下：

```
atc --singleop=$HOME/singleop/gemm.json --output=$HOME/singleop/out/op_model --soc_version=<soc_version>
```

关键参数解释如下（详细参数说明，请参见《ATC工具使用指南》。）：

- --singleop：单算子描述文件（json格式）的路径。
- --output：存放单算子模型文件的目录。
- --soc_version：昇腾AI处理器的版本。

  进入“CANN软件安装目录/compiler/data/platform_config”目录，".ini"文件的文件名即为昇腾AI处理器的版本，请根据实际情况选择。

2. 编写调用CBLAS的代码逻辑。

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，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。完整代码，您可以从**acl_execute_gemm**样例中查看。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（使用默认Context、默认Stream，默认Stream在作为其它接口入参时，可传空指针）
aclRet = aclrtSetDevice(0);
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）
aclrtRunMode runMode;
bool g_isDevice = false;
aclError aclRet = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3. 设置单算子模型文件所在的目录
// 该目录相对可执行文件所在的目录，例如，编译出来的可执行文件存放在run/out目录下，此处就表示run/out/op_models目录
aclopSetModelDir("op_models");

// 4. 申请内存
// 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，依次申请存放矩阵A数据、矩阵B数据、矩阵C数据、标量α数据、标量β数据的内存
aclrtMalloc((void **) &devMatrixA_, sizeA_, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc((void **) &devMatrixB_, sizeB_, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc((void **) &devMatrixC_, sizeC_, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc((void **) &devAlpha_, sizeAlphaBeta_, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc((void **) &devBeta_, sizeAlphaBeta_, ACL_MEM_MALLOC_NORMAL_ONLY);

// 申请Host上的内存，此处根据软件栈的运行模式判断是否需要申请Host上的内存
// 如果运行模式为ACL_DEVICE，则g_isDevice参数值为true，表示软件栈运行在Device侧，无需申请Host内存，无需传输图片数据或在Device内传输数据
// 如果运行模式为ACL_HOST，则g_isDevice参数值为false，表示软件栈运行在Host侧，需要申请Host内存，涉及Host和Device之间的数据传输
if (g_isDevice) {
        hostMatrixA_ = devMatrixA_;
        hostMatrixB_ = devMatrixB_;
        hostMatrixC_ = devMatrixC_;
    } else {
        aclrtMallocHost((void **) &hostMatrixA_, sizeA_);
        aclrtMallocHost((void **) &hostMatrixB_, sizeB_);
```



```
        aclrtMallocHost((void **) &hostMatrixC_, sizeC_);
    }

// 5. 准备输入数据，ReadFile为自定义函数，由用户自行管理，从文件中读入数据到内存中
size_t fileSize;
// Read matrix A
char *fileData = ReadFile("test_data/data/matrix_a.bin", fileSize, hostMatrixA_, sizeA_);
// Read matrix B
fileData = ReadFile("test_data/data/matrix_b.bin", fileSize, hostMatrixB_, sizeB_);
// Read matrix C
fileData = ReadFile("test_data/data/matrix_c.bin", fileSize, hostMatrixC_, sizeC_);
// 根据软件栈的运行模式判断是否涉及Host与Device之间的数据传输
if (!g_isDevice) {
    aclError ret = aclrtMemcpy(devMatrixA_, sizeA_, hostMatrixA_, sizeA_,
ACL_MEMCPY_HOST_TO_DEVICE);
    ret = aclrtMemcpy(devMatrixB_, sizeB_, hostMatrixB_, sizeB_, ACL_MEMCPY_HOST_TO_DEVICE);
    ret = aclrtMemcpy(devMatrixC_, sizeC_, hostMatrixC_, sizeC_, ACL_MEMCPY_HOST_TO_DEVICE);
}

aclrtMemcpyKind kind = g_isDevice ? ACL_MEMCPY_DEVICE_TO_DEVICE :
ACL_MEMCPY_HOST_TO_DEVICE;
ret = aclrtMemcpy(devAlpha_, sizeAlphaBeta_, hostAlpha_, sizeAlphaBeta_, kind);
ret = aclrtMemcpy(devBeta_, sizeAlphaBeta_, hostBeta_, sizeAlphaBeta_, kind);

// 6. 执行单算子
// 对于该示例，调用aclblasGemmEx接口（异步接口）实现矩阵-矩阵的乘法
aclblasGemmEx(ACL_TRANS_N, ACL_TRANS_N, ACL_TRANS_N, m_, n_, k_,
                devAlpha_, devMatrixA_, k_, inputType_, devMatrixB_, n_, inputType_,
                devBeta_, devMatrixC_, n_, outputType_, ACL_COMPUTE_HIGH_PRECISION,
                stream);
// 调用aclrtSynchronizeStream接口阻塞Host运行，直到指定Stream中的所有任务都完成
aclrtSynchronizeStream(nullptr);

// 7. 传输算子执行结果，根据软件栈的运行模式判断是否涉及Host与Device之间的数据传输
if (!g_isDevice) {
    auto ret = aclrtMemcpy(hostMatrixC_, sizeC_, devMatrixC_, sizeC_,
ACL_MEMCPY_DEVICE_TO_HOST);
}

// 8. 是否直接在终端屏幕上显示算子执行结果，由用户自行管理代码逻辑

// 9. 释放运行管理资源（默认Context、Stream无需用户释放，调用aclrtResetDevice接口后自动释放）
aclRet = aclrtResetDevice(0);

// 10.AscendCL去初始化
aclRet = aclFinalize();

// ......
```

相关资源

通过在线视频课程学习该功能，请参见**CANN应用开发初级**。

7.4.2 执行固定 Shape 算子示例代码

本节介绍基于单算子模型执行的方式调用固定Shape算子的关键接口、示例代码。

前提条件

在调用AscendCL接口执行固定Shape算子前，需提前编译算子。此处借助ATC工具编译Add算子的模型文件为例：

1. 先构造该算子的描述文件（*.json文件，描述输入输出Tensor描述、算子属性等）。

   Add算子的描述文件示例如下：


```
[
 {
  "op": "Add",
  "input_desc": [
   {
    "format": "ND",
    "shape": [8, 16],
    "type": "int32"
   },
   {
    "format": "ND",
    "shape": [8, 16],
    "type": "int32"
   }
  ],
  "output_desc": [
   {
    "format": "ND",
    "shape": [8, 16],
    "type": "int32"
   }
  ]
 }
]
```

2. 借助ATC工具，将该算子描述文件编译成单算子模型文件（*.om文件），再分别调用AscendCL接口加载om模型文件、执行算子。

   ATC工具的命令示例如下：

   ```
   atc --singleop=$HOME/singleop/add.json --output=$HOME/singleop/out/op_model --
   soc_version=<soc_version>
   ```

   关键参数解释如下（详细参数说明，请参见《ATC工具使用指南》。）：

   - --singleop：单算子描述文件（json格式）的路径。
   - --output：存放单算子模型文件的目录。
   - --soc_version：昇腾AI处理器的版本。

     进入“CANN软件安装目录/compiler/data/platform_config”目录，".ini"文件的文件名即为昇腾AI处理器的版本，请根据实际情况选择。

示例代码

以下是单算子加载、执行关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考，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。完整代码，您可以从**acl_execute_add**样例中查看。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（使用默认Context、默认Stream，默认Stream在作为其它接口入参时，可传空指针）
aclRet = aclrtSetDevice(0);
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）
aclrtRunMode runMode;
bool g_isDevice = false;
aclError aclRet = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3.加载单算子模型文件（*.om文件）
// 该目录相对可执行文件所在的目录，例如，编译出来的可执行文件存放在out目录下，此处就表示out/
op_models目录
aclRet = aclopSetModelDir("op_models");

// 4.执行算子
// opType表示算子类型名称，例如Add
// numInputs表示算子输入个数，例如Add算子是2个输入
// inputDesc表示算子输入tensor描述的数组，描述每个输入的format、shape、数据类型
```



```
// inputs表示算子输入tensor数据
// numOutputs表示算子输出个数，例如Add算子是1个输出
// outputDesc表示算子输出tensor描述的数组，描述每个输出的format、shape、数据类型
// outputs表示算子输出tensor数据
// attr表示算子属性，如果算子没有属性，也需要调用aclopCreateAttr接口创建aclopAttr类型的数据
// stream用于维护一些异步操作的执行顺序

aclopExecuteV2(opType, numInputs, inputDesc, inputs,
               numOutputs, outputDesc, outputs, attr, nullptr);

// 处理执行算子后的输出数据，例如在屏幕上显示、写入文件等，由用户根据实际情况自行实现
// ......

// 阻塞应用运行，直到指定Stream中的所有任务都完成
aclrtSynchronizeStream(nullptr);

// 5. 释放运行管理资源（默认Context、Stream无需用户释放，调用aclrtResetDevice接口后自动释放）
aclRet = aclrtResetDevice(0);

// 6.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

7.4.3 执行动态 Shape 算子示例代码（不注册算子选择器）

本节介绍基于单算子模型执行的方式调用动态Shape算子的关键接口、示例代码。

基本原理

对于支持动态Shape的算子：

- 如果算子输出Shape明确时，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**7.2 单算子调用流程**，执行固定Shape算子的示例代码请参见**7.4.2 执行固定Shape算子示例代码**。
- 如果无法明确算子的输出Shape时，在调用aclopExecuteV2接口前，需用户调用aclopInferShape接口、aclGetTensorDescNumDims接口、aclGetTensorDescDimV2接口、aclGetTensorDescDimRange等接口，推导或预估算子的输出Shape，作为算子执行接口aclopExecuteV2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，不可以直接拷贝编译运行，仅供参考。

```
// ......

const char *opType;
int numInputs;
aclTensorDesc *inputDesc[2];
aclDataBuffer *inputs[2];
int numOutputs;
aclTensorDesc *outputDesc[1];
aclopAttr *attr;

// 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。
aclError ret = aclopInferShape(opType, numInputs, inputDesc, inputs，numOutputs, outputDesc, attr);

std::vector<std::vector<int64_t>> tensorDims; // inferShape之后的输出tensor的Shape
// 循环算子的每一个输出，推导或预估Shape值：
for (int index = 0; index < numOutputs; ++index) {
    std::vector<int64_t> dimSize; // 表示执行算子时，输出的shape
```



```
    size_t dimNums = aclGetTensorDescNumDims(outputDesc[index]);
        // 表示动态Shape场景下维度个数未知，该场景预留
    if (dimNums == ACL_UNKNOWN_RANK)  {
        // 由用户预估最大Shape值max shape
        dimSize.push_back(max_shape);
    } else {
        for (size_t i = 0; i < dimNums; ++i) {
            int64_t dim;
            ret = aclGetTensorDescDimV2(outputDesc[index], i, &dim);

                        // 表示动态Shape场景下维度值是动态的
            if(dim == -1) {
                int64_t dimRange[2];
                            // 获取Shape范围，使用该范围中的Shape最大值来构造输出tensorDesc，作为
aclopExecuteV2的输入
                ret = aclGetTensorDescDimRange(outputDesc[index], i, 2, dimRange);
                dim = dimRange[1];
            }
            dimSize.push_back(dim);
        }
    }
    tensorDims.push_back(dimSize);
}

// 构造算子输入tensorDesc和输入tensor，作为aclOpExecuteV2的输入
aclTensorDesc *inputDescNew[2];
aclDataBuffer *inputsNew[2];
aclDataBuffer *outputsNew[1];
// 以上给出了执行算子时输出的shape, 根据tensorDims中的dims构造输出tensorDesc（即outputDescNew参数
值）, 用于调用aclopExecuteV2
ret = aclopExecuteV2(opType, numInputs, inputDescNew, inputsNew, numOutputs, outputDescNew,
outputsNew, attr, stream);

// 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，在算子执行结束后，需增加下面的调
用，获取准确的shape：
// for 循环每一个输出的tensorDesc
std::vector<std::vector<int64_t>> outTensorDims; // 准确的输出tensorShape
for (int index = 0; index < numOutputs; ++index) {
    std::vector<int64_t> dimSize;
    int dimNums = aclGetTensorDescNumDims(outputDescNew[index]);
    for (int i = 0; i < dimNums; i++){
        int64_t dim;
        ret = aclGetTensorDescDimV2(outputDescNew[index], i, &dim);
        dimSize.push_back(dim);
    }
    outTensorDims.push_back(dimSize);
}
// ......
```

7.4.4 执行动态 Shape 算子示例代码（注册算子选择器）

本节介绍基于单算子模型执行的方式调用动态Shape算子的关键接口、示例代码，注册算子选择器的使用场景有限，主要针对TIK自定义算子。

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、注册算子选择器场景下，算子加载与执行的流程如下：

1. 资源初始化，包括AscendCL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aclInit接口实现AscendCL初始化。
- 调用AscendCL接口注册要编译的自定义算子：
  - 调用aclopRegisterCompileFunc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：
      ```
      typedef aclError (*aclopCompileFunc)(int numInputs, const aclTensorDesc *const inputDesc[], int numOutputs, const aclTensorDesc *const outputDesc[], const aclopAttr *opAttr, aclopKernelDesc *aclopKernelDesc);
      ```
    - 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，并调用aclopSetKernelArgs接口，设置算子Tiling参数、执行并发数等。
  - 调用aclopCreateKernel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aclrtSetDevice接口指定运算的Device。
- 调用aclrtCreateContext接口显式创建一个Context，调用aclrtCreateStream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aclrtSetDevice接口时隐式创建的，默认Stream作为接口入参时，直接传NULL。

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. 将算子输入数据复制到Device上。
   - 调用aclrtMemcpy接口实现同步内存复制，内存使用结束后需及时释放。
   - 调用aclrtMemcpyAsync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aclopUpdateParams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aclopExecuteV2接口加载并执行算子。
6. 获取算子运算的输出数据。
   - 调用aclrtMemcpy接口实现同步内存复制，内存使用结束后需及时释放。
   - 调用aclrtMemcpyAsync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aclrtDestroyStream接口释放Stream。

     不涉及显式创建Stream，使用默认Stream时，无需调用aclrtDestroyStream接口释放Stream。
   - 调用aclrtDestroyContext接口释放Context。

     不涉及显式创建Context，使用默认Context时，无需调用aclrtDestroyContext接口释放Context。
   - 调用aclrtResetDevice接口释放Device。

8. 调用aclFinalize接口实现AscendCL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，不可以直接拷贝编译运行，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”。

```
#include "acl/acl.h"
// ......

// 1.资源初始化
// 此处是相对路径，相对可执行文件所在的目录
aclError ret = aclInit(NULL);
aclopRegisterCompileFunc("BatchNorm", SelectAclopBatchNorm);
// 将算子Kernel的*.o文件需要用户提前编译好，并调用用户自定义函数该文件加载到内存buffer中，length表示内存大小，如果有多个算子Kernel的*.o文件，需要多次调用该接口
aclopCreateKernel("BatchNorm", "tiling_mode_1__kernel0", "tiling_mode_1__kernel0",
                  buffer, length, ACL_ENGINE_AICORE, Deallocator);

// -----自定义函数BatchNormTest(n, c, h, w)，执行以下操作-----

// 2.构造BatchNorm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
aclTensorDesc *input_desc[3];
aclTensorDesc *output_desc[1];
input_desc[0] = aclCreateTensorDesc(ACL_FLOAT16, 4, shape_input, ACL_FORMAT_NCHW);
input_desc[1] = aclCreateTensorDesc(ACL_FLOAT16, 1, shape_gamma, ACL_FORMAT_ND);
input_desc[2] = aclCreateTensorDesc(ACL_FLOAT16, 1, shape_beta, ACL_FORMAT_ND);
output_desc[0] = aclCreateTensorDesc(ACL_FLOAT16, 4, shape_out, ACL_FORMAT_NCHW);

for (int i = 0; i < n * c * h * w; ++i) {
      input[i] = aclFloatToFloat16(1.0f);
   }

   for (int i = 0; i < c; ++i) {
      gamma[i] = aclFloatToFloat16(0.5f);
      beta[i] = aclFloatToFloat16(0.1f);
   }

aclrtMalloc(&devInput, size_input, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc(&devInput_gamma, size_gamma, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc(&devInput_beta, size_beta, ACL_MEM_MALLOC_NORMAL_ONLY);
aclrtMalloc(&devOutput, size_output, ACL_MEM_MALLOC_NORMAL_ONLY);

// 3.将算子输入数据复制到Device上
aclrtMemcpy(devInput, size_input, input, size_input, ACL_MEMCPY_HOST_TO_DEVICE);
aclrtMemcpy(devInput_gamma, size_gamma, gamma, size_gamma, ACL_MEMCPY_HOST_TO_DEVICE);
aclrtMemcpy(devInput_beta, size_beta, beta, size_beta, ACL_MEMCPY_HOST_TO_DEVICE);

// 4.调用aclopUpdateParams接口编译算子
aclopUpdateParams("BatchNorm", 3, input_desc, 1, output_desc, nullptr, ACL_ENGINE_AICORE,
ACL_COMPILE_UNREGISTERED, nullptr));

// 5.调用aclopExecuteV2接口加载并执行算子

aclopExecuteV2("BatchNorm", 3, input_desc, inputs, 1, output_desc, outputs, nullptr, stream);

// -----自定义函数BatchNormTest(n, c, h, w)，执行以上操作-----

// 6.获取算子运算的输出数据
aclrtMemcpy(output, size_output, devOutput, size_output, ACL_MEMCPY_DEVICE_TO_HOST);

// 7.按顺序释放资源
// 7.1 释放算子的输入输出Tensor描述
```



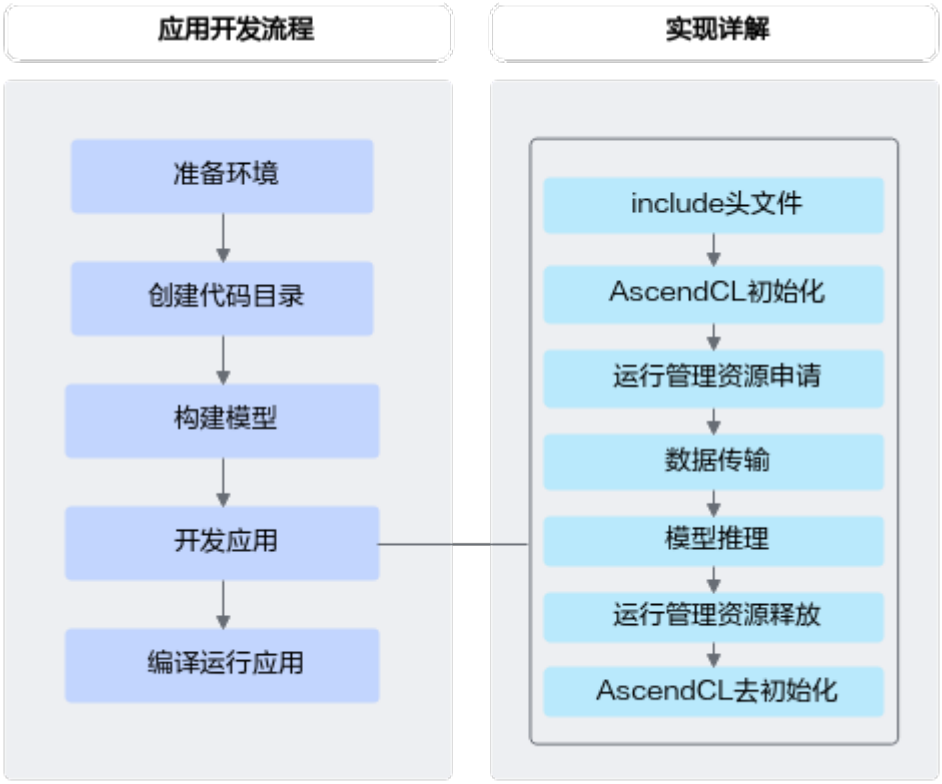
```
for (auto desc : input_desc) {
        aclDestroyTensorDesc(desc);
    }

for (auto desc : output_desc) {
        aclDestroyTensorDesc(desc);
    }
// 7.2 及时释放不使用的内存
delete[]input;
delete[]gamma;
delete[]beta;
delete[]output;
// 7.3 释放Device上的内存
aclrtFree(devInput);
aclrtFree(devInput_gamma);
aclrtFree(devInput_beta));
aclrtFree(devOutput);
// 7.4 依次释放Stream、Context、Device资源，如果未显式创建Stream、Context，则无需释放
aclrtDestroyStream(stream);
aclrtDestroyContext(context);
aclrtResetDevice(deviceId);
aclFinalize();
```


8 模型推理

8.1 推理应用开发流程

本节介绍基于AscendCL接口开发基础推理应用的开发流程。

图 8-1 开发流程



1. **准备环境**。
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，供参考：

   ```
   ├App名称
   ├── model                 // 该目录下存放模型文件
   │   ├── xxxxxx
   ├── data
   │   ├── xxx.jpg           // 测试数据
   ├── inc                   // 该目录下存放声明函数的头文件
   │   ├── xxx.h
   ├── out                   // 该目录下存放输出结果
   ├── src
   │   ├── xxx.json          // 系统初始化的配置文件
   │   ├── CMakeLists.txt    // 编译脚本
   │   ├── xxx.cpp           // 实现文件
   ```

3. **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**8.2 模型构建**。

4. **开发应用**。

   a. AscendCL初始化，请参见**5 AscendCL初始化**。

      使用AscendCL接口开发应用时，必须先调用aclInit接口进行AscendCL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**6.1 运行管理资源申请与释放**。
c. 数据传输，请参见**6.2 数据传输**。
d. 执行模型推理。请参见**8.3 单Batch&静态Shape输入推理**。

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。

模型推理结束后，需及时释放推理相关资源。

e. 所有数据处理结束后，需及时释放运行管理资源，请参见**6.1 运行管理资源申请与释放**。
f. 执行AscendCL去初始化，请参见**5 AscendCL初始化**。

5. **编译运行应用**，包括编译代码、运行应用，请参见**11 应用编译&运行**。

8.2 模型构建

对于开源框架的网络模型（如ONNX、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处理器版本替换。

```
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**8.9 模型动态Shape输入推理**。
- 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》。

8.3 单 Batch&静态 Shape 输入推理

8.3.1 模型加载

本节介绍如何加载模型，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**4.2 AscendCL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**7.2 单算子调用流程**。

AscendCL提供**两套模型加载的接口**，用户可根据编程习惯、使用场景选择对应的模型加载接口：

- 如**图8-2**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。
- 如**图8-3**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**。

图 8-2 模型加载流程（通过接口中的配置参数区分加载方式）

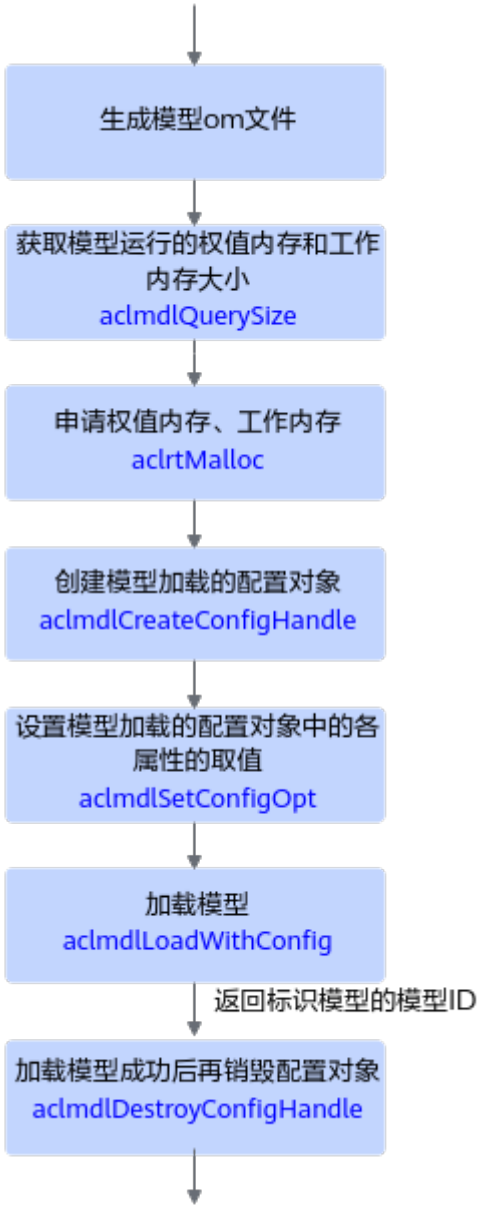
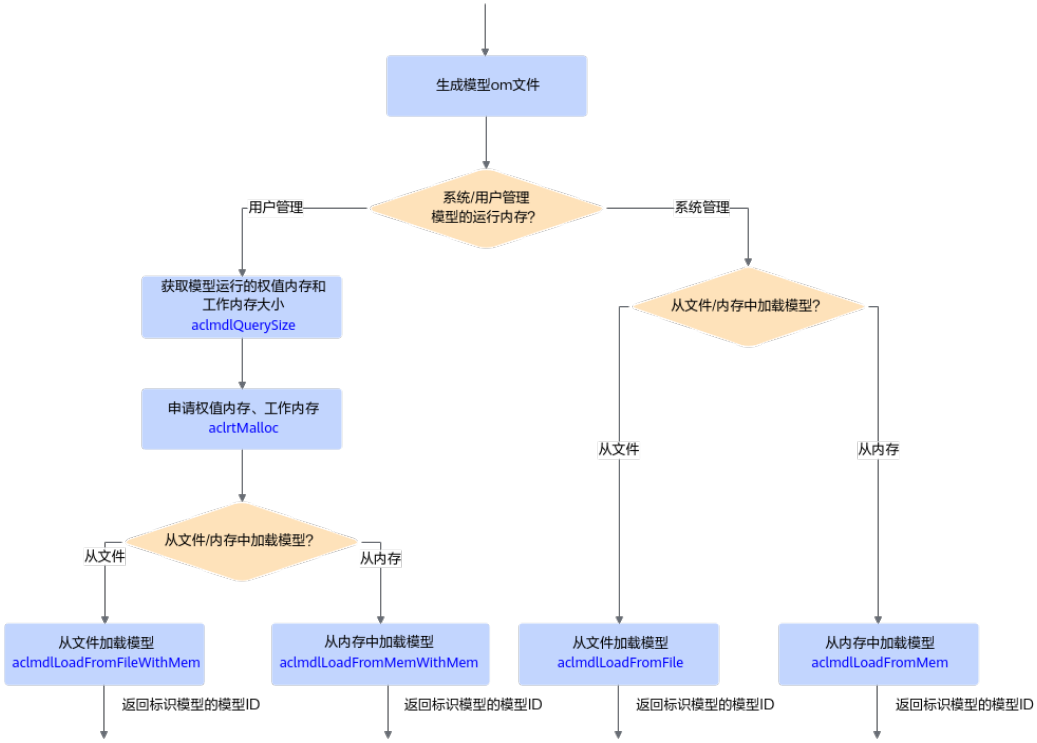


图 8-3 模型加载流程（通过不同接口区分加载方式）



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**8.2 模型构建**。
- 当由用户管理内存时，为确保内存不浪费，在申请工作内存、权值内存前，需要调用aclmdlQuerySize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aclmdlQuerySize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，aclmdlLoadFromFile、aclmdlLoadFromMem）。
- 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：
  - 使用aclmdlSetConfigOpt接口、aclmdlLoadWithConfig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。
  - 使用以下接口时，是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。
    - aclmdlLoadFromFile：从文件加载离线模型数据，由系统内部管理内存。
    - aclmdlLoadFromMem：从内存加载离线模型数据，由系统内部管理内存。
    - aclmdlLoadFromFileWithMem：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）。
    - aclmdlLoadFromMemWithMem：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）。

示例代码

模型加载成功，会返回标识模型的ID，在**8.3.2 模型执行**时需要使用该ID。

此处以从文件加载模型、由用户自行管理内存为例。您可以从**13.10.1 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，不可以直接拷贝编译运行，仅供参考。

```
// 1.初始化变量。
// 此处的..表示相对路径，相对可执行文件所在的目录
// 例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char* omModelPath = "../model/resnet50.om"
// ......

// 2.根据模型文件获取模型执行时所需的权值内存大小、工作内存大小。
aclError ret = aclmdlQuerySize(omModelPath, &modelMemSize_, &modelWeightSize_);

// 3.根据工作内存大小，申请Device上模型执行的工作内存。
ret = aclrtMalloc(&modelMemPtr_, modelMemSize_, ACL_MEM_MALLOC_HUGE_FIRST);

// 4.根据权值内存的大小，申请Device上模型执行的权值内存。
ret = aclrtMalloc(&modelWeightPtr_, modelWeightSize_, ACL_MEM_MALLOC_HUGE_FIRST);

// 5.加载离线模型文件，由用户自行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。
ret = aclmdlLoadFromFileWithMem(modelPath, &modelId_, modelMemPtr_, modelMemSize_,
modelWeightPtr_, modelWeightSize_);

// ......
```

8.3.2 模型执行

本节结合接口调用流程、示例代码介绍模型执行前需要准备哪些数据、模型执行接口以及模型执行之后需要释放哪些资源。

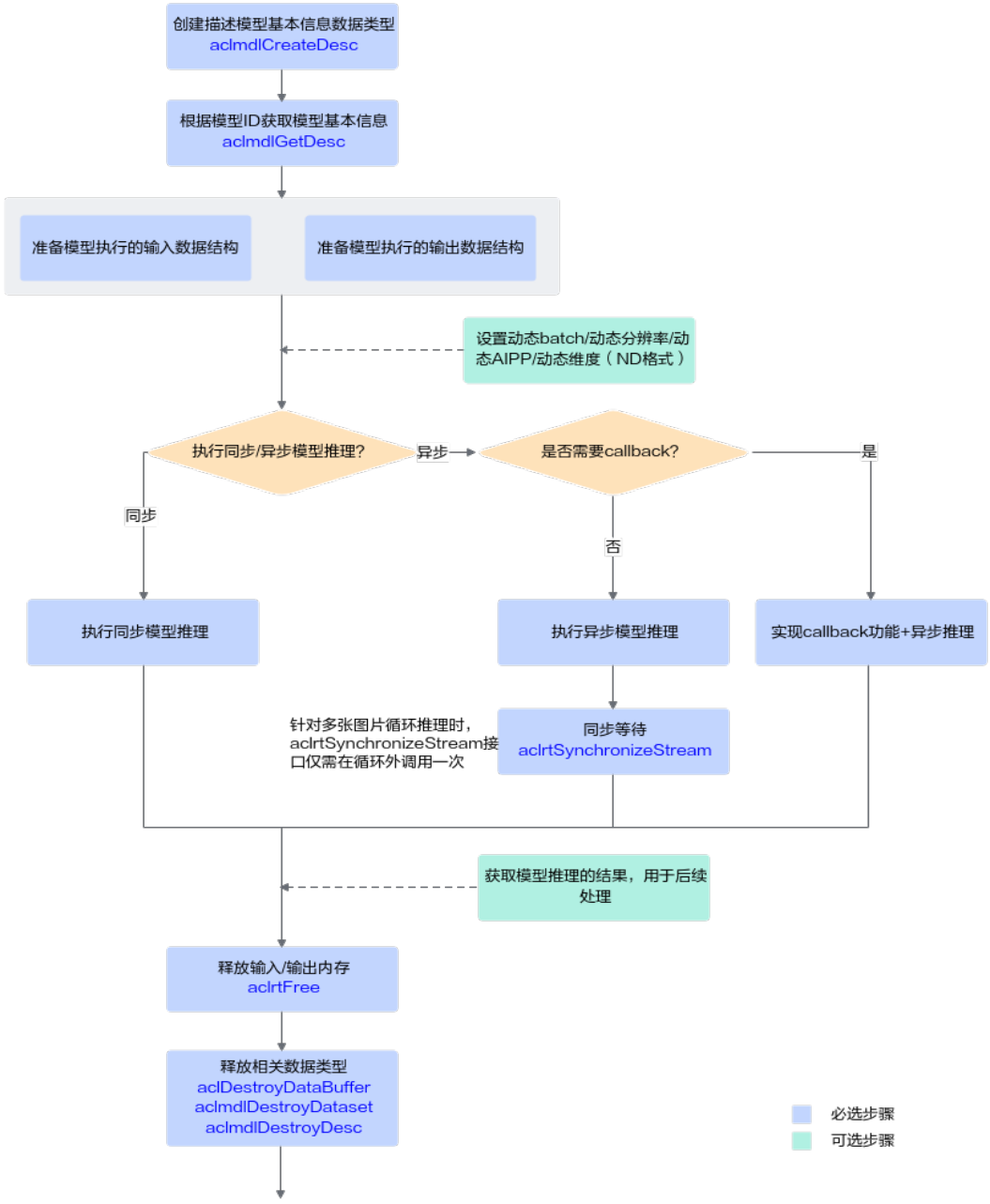
基本原理

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**4.2 AscendCL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**7 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aclDestroyDataBuffer接口销毁相应的aclDataBuffer类型，并调用aclrtFree接口释放内存中的数据。

模型执行流程

图 8-4 基本的模型推理流程



关键接口的说明如下：

1. 调用aclmdlCreateDesc接口**创建描述模型基本信息**的数据类型。
2. 调用aclmdlGetDesc接口根据**8.3.1 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**8.9 模型动态Shape输入推理**、**8.8 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 size后，才能将输入数据发送给模型进行推理。不满足batch size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

     调用aclmdlExecute接口执行同步推理。

   - 异步推理

     调用aclmdlExecuteAsync接口执行异步推理。

     但对于异步接口，还需调用aclrtSynchronizeStream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**8.5 异步模型推理**。

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aclrtFree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，需依次调用aclDestroyDataBuffer接口、aclmdlDestroyDataset接口及时释放描述模型输入、输出数据类型的数据。如果存在多个输入、输出，需调用多次aclDestroyDataBuffer接口。

准备模型执行的输入/输出数据结构

AscendCL提供了以下数据类型来描述模型、描述其输入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aclmdlGetDesc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。

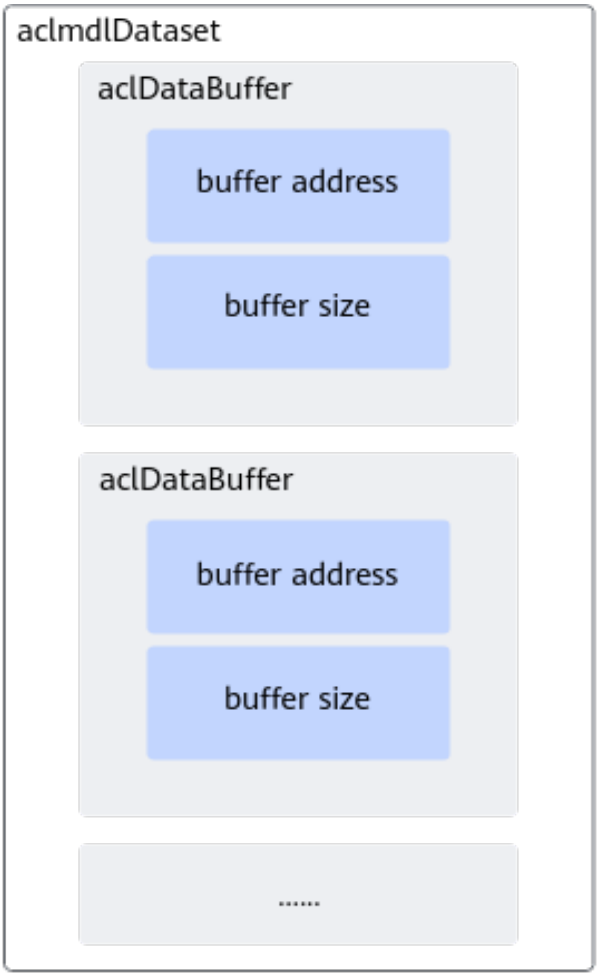
- 使用**aclmdlDataset**类型的数据描述模型的输入/输出数据，模型可能存在多个输入、多个输出。

  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 8-5 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 8-6 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aclmdlGetNumInputs、aclmdlGetNumOutputs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aclmdlGetInputSizeByIndex、aclmdlGetOutputSizeByIndex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aclmdlGetInputSizeByIndex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aclmdlGetInputNameByIndex、aclmdlGetOutputNameByIndex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

此处的示例代码是处理图片分类模型的输出结果，屏显每张图片的top5置信度的类别编号。用户可根据实际需求，自行实现模型推理输出数据的处理逻辑。

您可以从**13.10.1 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，不可以直接拷贝编译运行，仅供参考。

```
// 1 根据模型的ID，获取该模型的描述信息。
// modelDesc_为aclmdlDesc类型。
modelDesc_ = aclmdlCreateDesc();
ret = aclmdlGetDesc(modelDesc_, modelId_);

// 2 准备模型推理的输入数据结构
// (1)申请输入内存
size_t modelInputSize;
void *modelInputBuffer = nullptr;
// 当前示例代码中的模型只有一个输入，所以index为0，如果模型有多个输入，则需要先调用
aclmdlGetNumInputs接口获取模型输入的数量
modelInputSize = aclmdlGetInputSizeByIndex(modelDesc_, 0);
aclRet = aclrtMalloc(&modelInputBuffer, modelInputSize, ACL_MEM_MALLOC_NORMAL_ONLY);

// (2)准备模型的输入数据结构
// 创建aclmdlDataset类型的数据，描述模型推理的输入，input_为aclmdlDataset类型
input_ = aclmdlCreateDataset();
aclDataBuffer *inputData = aclCreateDataBuffer(modelInputBuffer, modelInputSize);
ret = aclmdlAddDatasetBuffer(input_, inputData);

// 3 准备模型推理的输出数据结构
// (1)创建aclmdlDataset类型的数据，描述模型推理的输出，output_为aclmdlDataset类型
output_ = aclmdlCreateDataset();

// (2)获取模型的输出个数.
size_t outputSize = aclmdlGetNumOutputs(modelDesc_);

// (3)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 (size_t i = 0; i < outputSize; ++i) {
   size_t buffer_size = aclmdlGetOutputSizeByIndex(modelDesc_, i);
   void *outputBuffer = nullptr;
   aclError ret = aclrtMalloc(&outputBuffer, buffer_size, ACL_MEM_MALLOC_NORMAL_ONLY);
   aclDataBuffer* outputData = aclCreateDataBuffer(outputBuffer, buffer_size);
   ret = aclmdlAddDatasetBuffer(output_, outputData);
   }

// 4 模型执行
string testFile[] = {
       "../data/dog1_1024_683.bin",
       "../data/dog2_1024_683.bin"
   };

for (size_t index = 0; index < sizeof(testFile) / sizeof(testFile[0]); ++index) {
   // 4.1 自定义函数ReadBinFile，调用C++标准库std::ifstream中的函数读取图片文件，输出图片文件占用的内存
大小inputBuffSize以及图片文件存放在内存中的地址inputBuff
   void *inputBuff = nullptr;
   uint32_t inputBuffSize = 0;
   auto ret = Utils::ReadBinFile(fileName, inputBuff, inputBuffSize);

   // 4.2 准备模型推理的输入数据
   // 在申请运行管理资源时调用aclrtGetRunMode接口获取软件栈的运行模式
   // 如果运行模式为ACL_DEVICE，则g_isDevice参数值为true，表示软件栈运行在Device侧，无需传输图片数据
```



```
或在Device内传输数据；否则，需要调用内存复制接口将数据传输到Device
    if (!g_isDevice) {
        // if app is running in host, need copy data from host to device
        // modelInputBuffer、modelInputSize分别表示模型推理输入数据的内存地址、内存大小，在输入/输出数据结构准备时申请该内存
        aclError aclRet = aclrtMemcpy(modelInputBuffer, modelInputSize, inputBuff, inputBuffSize, ACL_MEMCPY_HOST_TO_DEVICE);
        (void)aclrtFreeHost(inputBuff);
    } else { // app is running in device
        aclError aclRet = aclrtMemcpy(modelInputBuffer, modelInputSize, inputBuff, inputBuffSize, ACL_MEMCPY_DEVICE_TO_DEVICE);
        (void)aclrtFree(inputBuff);
    }

    // 4.3 执行模型推理
    // modelId_表示模型ID，在模型加载成功后，会返回标识模型的ID
    // input_、output_分别表示模型推理的输入、输出数据，在准备模型推理的输入、输出数据结构时已定义
    aclError ret = aclmdlExecute(modelId_, input_, output_)


    // 处理模型推理的输出数据，输出top5置信度的类别编号
    // output_表示模型执行的输出
    for (size_t i = 0; i < aclmdlGetDatasetNumBuffers(output_); ++i) {
    // 获取每个输出的内存地址和内存大小
        aclDataBuffer* dataBuffer = aclmdlGetDatasetBuffer(output_, i);
        void* data = aclGetDataBufferAddr(dataBuffer);

        size_t len = aclGetDataBufferSizeV2(dataBuffer);

        // 将内存中的数据转换为float类型
        float *outData = NULL;
        outData = reinterpret_cast<float*>(data);

        // 屏显每张图片的top5置信度的类别编号
        map<float, int, greater<float> > resultMap;
        for (int j = 0; j < len / sizeof(float); ++j) {
            resultMap[*outData] = j;
            outData++;
        }
        int cnt = 0;
        for (auto it = resultMap.begin(); it != resultMap.end(); ++it) {
            // print top 5
            if (++cnt > 5) {
                break;
            }

            INFO_LOG("top %d: index[%d] value[%lf]", cnt, it->second, it->first);
    }
}

// 5 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
for (size_t i = 0; i < aclmdlGetDatasetNumBuffers(input_); ++i) {
        aclDataBuffer *dataBuffer = aclmdlGetDatasetBuffer(input_, i);
        (void)aclDestroyDataBuffer(dataBuffer);
}
(void)aclmdlDestroyDataset(input_);
input_ = nullptr;
aclrtFree(modelInputBuffer);

// 释放输出资源，包括数据结构和内存
for (size_t i = 0; i < aclmdlGetDatasetNumBuffers(output_); ++i) {
    aclDataBuffer* dataBuffer = aclmdlGetDatasetBuffer(output_, i);
    void* data = aclGetDataBufferAddr(dataBuffer);
    (void)aclrtFree(data);
    (void)aclDestroyDataBuffer(dataBuffer);
}
```



```
(void)aclmdlDestroyDataset(output_);
output_ = nullptr;
```

8.3.3 模型卸载

模型执行结束后，需及时卸载模型，释放模型资源。

关于模型卸载的接口调用流程，请先参见**4.2 AscendCL接口调用流程**了解整体流程，再查看本节中的流程说明。

基本原理

在模型推理结束后，还需要通过aclmdlUnload接口卸载模型，并销毁aclmdlDesc类型的模型描述信息、释放模型运行的工作内存和权值内存。

示例代码

```
// 1. 卸载模型
aclError ret = aclmdlUnload(modelId_);

// 2. 释放模型描述信息
if (modelDesc_ != nullptr) {
    (void)aclmdlDestroyDesc(modelDesc_);
    modelDesc_ = nullptr;
}

// 3. 释放模型运行的工作内存
if (modelWorkPtr_ != nullptr) {
    (void)aclrtFree(modelWorkPtr_);
    modelWorkPtr_ = nullptr;
    modelWorkSize_ = 0;
}

// 4. 释放模型运行的权值内存
if (modelWeightPtr_ != nullptr) {
    (void)aclrtFree(modelWeightPtr_);
    modelWeightPtr_ = nullptr;
    modelWeightSize_ = 0;
}
```

8.4 多 Batch 模型推理

多Batch推理的基本流程与单Batch类似，请参见**8 模型推理**。

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参数需根据实际的Batch数设置batch size值，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，实现逻辑为：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，不可以直接拷贝编译运行，仅供参考，此处以8Batch为例：

```
uint32_t batchSize = 8;
uint32_t deviceNum = 1;
uint32_t deviceId = 0;
```



```
// 获取模型第一个输入的大小
uint32_t modelInputSize = aclmdlGetInputSizeByIndex(modelDesc, 0);
// 获取每个Batch输入数据的大小
uint32_t singleBuffSize = modelInputSize / batchSize;

// 定义该变量，用于累加batch size是否达到8Batch
uint32_t 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
uint32_t pos = 0;

void* p_batchDst = NULL;
std::vector<std::string>inferFile_vec;

for (int i = 0; i < files.size(); ++i)
      {
         // 每8个文件，申请一次Device上的内存，存放8Batch的输入数据
         if (cnt % batchSize == 0)
         {
            pos = 0;
            inferFile_vec.clear();
            // 申请Device上的内存
            ret = aclrtMalloc(&p_batchDst, modelInputSize, ACL_MEM_MALLOC_NORMAL_ONLY);
         }

         // TODO: 从某个目录下读入文件，计算文件大小fileSize

         // 根据文件大小，申请内存，存放文件数据
         ret = aclrtMallocHost(&p_imgBuf, fileSize);

         // 将数据传输到Device的内存
         ret = aclrtMemcpy((uint8_t *)p_batchDst + pos, fileSize, p_imgBuf, fileSize,
ACL_MEMCPY_HOST_TO_DEVICE);
         pos += fileSize;
         // 及时释放不使用的内存
         aclrtFreeHost(p_imgBuf);

         // 将第i个文件存入vector中，同时cnt+1
         inferFile_vec.push_back(files[i]);
         cnt++;

         // 每8Batch的输入数据送给模型推理进行推理
         if (cnt % batchSize == 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aclmdlExecute接口执行模型推理
            // TODO: 推理结束后，调用aclrtFree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 % batchSize != 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aclmdlExecute接口执行模型推理
         // TODO: 推理结束后，调用aclrtFree接口释放Device上的内存
   }
```

8.5 异步模型推理

本节介绍异步推理接口如何与Callback配合使用，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。

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 8-7 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aclrtProcessReport接口，设置超时时间，等待aclrtLaunchCallback接口下发的回调任务执行。

3. 调用aclrtSubscribeReport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aclmdlExecuteAsync接口。

   对于异步接口，还需调用aclrtSynchronizeStream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aclrtSynchronizeStream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aclrtLaunchCallback接口：在Stream的任务队列中下发一个回调任务，系统内部在执行到该回调任务时，会在Stream上注册的线程（通过aclrtSubscribeReport接口注册的线程）中执行回调函数，回调函数与1中的回调函数保持一致。

   每调用一次aclrtLaunchCallback接口，就会触发一次回调函数的执行。
6. 调用aclrtUnSubscribeReport接口：取消线程注册（Stream上的回调函数不再由指定线程处理）。

示例代码

您可以从13.11.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍异步模型推理的代码逻辑，AscendCL初始化和去初始化请参见5 AscendCL初始化，运行管理资源申请与释放请参见6.1 运行管理资源申请与释放。

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。

```
#include "acl/acl.h"
// ......
// 1. AscendCL初始化

// 2. 申请运行管理资源

// 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
aclrtRunMode runMode;
extern bool g_isDevice;
ret = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3. 申请模型推理资源

// 此处的..表示相对路径，相对可执行文件所在的目录
// 例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char* omModelPath = "../model/resnet50.om"

// 3.1 加载模型
// 根据模型文件获取模型执行时所需的权值内存大小、工作内存大小，并申请权值内存、工作内存
ret = aclmdlQuerySize(omModelPath, &modelMemSize_, &modelWeightSize_);
ret = aclrtMalloc(&modelMemPtr_, modelMemSize_, ACL_MEM_MALLOC_NORMAL_ONLY);
ret = aclrtMalloc(&modelWeightPtr_, modelWeightSize_, ACL_MEM_MALLOC_NORMAL_ONLY);

// 加载离线模型文件，模型加载成功，返回标识模型的ID。
ret = aclmdlLoadFromFileWithMem(modelPath, &modelId_, modelMemPtr_,
      modelMemSize_, modelWeightPtr_, modelWeightSize_);

// 3.2 根据模型的ID，获取该模型的描述信息
modelDesc_ = aclmdlCreateDesc();
ret = aclmdlGetDesc(modelDesc_, modelId_);

// 3.3 自定义函数InitMemPool，初始化内存池，存放模型推理的输入数据、输出数据
```



```
// -----自定义函数InitMemPool内部的关键实现-----
string testFile[] = {
        "../data/dog1_1024_683.bin",
        "../data/dog2_1024_683.bin"
    };
size_t fileNum = sizeof(testFile) / sizeof(testFile[0]);
// g_memoryPoolSize表示内存池中的内存块的个数默认为100个
for (size_t i = 0; i < g_memoryPoolSize; ++i) {
        size_t index = i % (sizeof(testFile) / sizeof(testFile[0]));
        // model process
        uint32_t devBufferSize;
        // 自定义函数GetDeviceBufferOfFile，完成以下功能：
        // 获取存放输入图片数据的内存及内存大小、将图片数据传输到Device
        void *picDevBuffer = Utils::GetDeviceBufferOfFile(testFile[index], devBufferSize);
        aclmdlDataset *input = nullptr;
        // 自定义函数CreateInput，创建aclmdlDataset类型的数据input，用于存放模型推理的输入数据
        Result ret = CreateInput(picDevBuffer, devBufferSize, input);
        aclmdlDataset *output = nullptr;
        // 自定义函数CreateOutput，创建aclmdlDataset类型的数据output，用于存放模型推理的输出数据，
modelDesc表示模型的描述信息
        CreateOutput(output, modelDesc);
        {
            std::lock_guard<std::recursive_mutex> lk(freePoolMutex_);
            freeMemoryPool_[input] = output;
        }
}
// -----自定义函数InitMemPool内部的关键实现-----

// 4 模型推理
// 4.1 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aclrtProcessReport接口，等待指定时间后，触发回调函数
处理
pthread_t tid;
(void)pthread_create(&tid, nullptr, ProcessCallback, &s_isExit);

// 4.2 指定处理Stream上回调函数的线程
aclError aclRt = aclrtSubscribeReport(tid, stream_);

// 4.2 创建回调函数，用户处理模型推理的结果，由用户自行定义
void ModelProcess::CallBackFunc(void *arg)
{
    std::map<aclmdlDataset *, aclmdlDataset *> *dataMap =
        (std::map<aclmdlDataset *, aclmdlDataset *> *)arg;

    aclmdlDataset *input = nullptr;
    aclmdlDataset *output = nullptr;
    MemoryPool *memPool = MemoryPool::Instance();

    for (auto& data : *dataMap) {
        ModelProcess::OutputModelResult(data.second);
        memPool->FreeMemory(data.first, data.second);
    }

    delete dataMap;
}

// 4.3 自定义函数ExecuteAsync，执行模型推理
// -----自定义函数ExecuteAsync内部的关键实现开始-----
// g_callbackInterval表示callback间隔，默认为1，表示1次异步推理后，下发一次callback任务
bool isCallback = (g_callbackInterval != 0);
    size_t callbackCnt = 0;
    std::map<aclmdlDataset *, aclmdlDataset *> *dataMap = nullptr;
    aclmdlDataset *input = nullptr;
    aclmdlDataset *output = nullptr;
    MemoryPool *memPool = MemoryPool::Instance();
    // g_executeTimes表示执行模型异步推理的次数，默认为100次
    for (uint32_t cnt = 0; cnt < g_executeTimes; ++cnt) {
        if (memPool->mallocMemory(input, output) != SUCCESS) {
            ERROR_LOG("get free memory failed");
```



```
            return FAILED;
        }
        // 执行异步推理
        aclError ret = aclmdlExecuteAsync(modelId_, input, output, stream_);

        if (isCallback) {
            if (dataMap == nullptr) {
                dataMap = new std::map<aclmdlDataset *, aclmdlDataset *>;
                if (dataMap == nullptr) {
                    ERROR_LOG("malloc list failed, modelId is %u", modelId_);
                    memPool->FreeMemory(input, output);
                    return FAILED;
                }
            }
            (*dataMap)[input] = output;
            callbackCnt++;
            if ((callbackCnt % g_callbackInterval) == 0) {
                // 在Stream的任务队列中增加一个需要执行的回调函数
                ret = aclrtLaunchCallback(CallBackFunc, (void *)dataMap, ACL_CALLBACK_BLOCK, stream_);
                if (ret != ACL_SUCCESS) {
                    ERROR_LOG("launch callback failed, index=%zu", callbackCnt);
                    memPool->FreeMemory(input, output);
                    delete dataMap;
                    return FAILED;
                }
                dataMap = nullptr;
            }

        }
    }
// -----自定义函数ExecuteAsync内部的关键实现结束-----

// 4.4 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
aclrtSynchronizeStream(stream_);

// 4.5 取消线程注册，Stream上的回调函数不再由指定线程处理
aclRt = aclrtUnSubscribeReport(static_cast<uint64_t>(tid), stream_);
s_isExit = true;
(void)pthread_join(tid, nullptr);

// 5 释放运行管理资源

// 6 AscendCL去初始化

// ......
```

8.6 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**8 模型推理**。

多模型推理与单模型推理在AscendCL接口使用上的不同点如下：

- 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**8.3.1 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**8.3.2 模型执行**。

  例如，调用aclmdlExecute接口实现同步模型推理。

8.7 队列方式模型推理

本节介绍如何基于队列加载模型、准备模型输入数据、获取模型推理输出数据。

基本原理

- 调用aclmdlLoadFromFileWithQ或aclmdlLoadFromMemWithQ接口以队列方式加载模型。
- 调用acltdtEnqueueData接口将模型的输入数据传入队列，由AscendCL内部根据队列中的输入数据进行推理，无需调用模型执行的接口。
- 调用acltdtDequeueData接口等待模型推理执行完毕，再由用户从输出内存中获取结果数据。

说明

如果涉及多线程，当模型有多个输入时，多个输入数据的入队任务（即调用acltdtEnqueueData接口）必须在同一个线程中；当模型有多个输出时，多个输出数据的出队任务（即调用acltdtDequeueData接口）必须在同一个线程中。

示例代码

本节中的示例重点介绍队列方式模型推理的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
#include "acl/acl.h"
// ......
// 1. AscendCL初始化

// 2. 申请运行管理资源

// 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
extern bool g_isDevice;
aclrtRunMode runMode;
aclError ret = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3. 加载并执行模型
// 此处的..表示相对路径，相对可执行文件所在的目录
// 例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char* omModelPath = "../model/resnet50.om"

// 3.1 创建模型的输入队列，如果模型有多个输入，则创建多个输入队列，此处以一个输入为例
acltdtQueueAttr *attr = acltdtCreateQueueAttr();
uint32_t *inputQueueList = new (nothrow) uint32_t[num];
int32_t inputNum = 1;

for (int n = 0; n < inputNum; n++) {
        uint32_t inputQid;
        ret = acltdtCreateQueue(attr, &inputQid);
        inputQueueList[n] = inputQid;
    }

// 3.2 创建模型的输出队列，如果模型有多个输出，则创建多个输出队列，此处以一个输出为例
uint32_t *outputQueueList = new (nothrow) uint32_t[num];
int32_t outputNum = 1;

for (int n = 0; n < outputNum; n++) {
        uint32_t outputQid;
        ret = acltdtCreateQueue(attr, &outputQid);
        outputQueueList[n] = outputQid;
    }

// 3.3 加载模型
uint32_t modelId;
ret= aclmdlLoadFromFileWithQ(modelPath, &modelId,
```

```
                        inputQueueList, inputNum, outputQueueList, outputNum);

// 3.4 根据模型的ID，获取该模型的描述信息
aclmdlDesc *modelDesc = aclmdlCreateDesc();
ret = aclmdlGetDesc(modelDesc, modelId);

// 3.5 获取模型的输入内存大小，如果模型有多个输入，则需要获取每个输入的内存大小，此处以一个输入为例
size_t inputSize = aclmdlGetInputSizeByIndex(modelDesc, 0);

// 3.6 加载测试图片数据，进行推理，并对推理结果数据进行后处理
string testFile[] = {
        "../data/dog1_1024_683.bin",
        "../data/dog2_1024_683.bin"
};

for (size_t index = 0; index < sizeof(testFile) / sizeof(testFile[0]); ++index) {
        uint32_t devBufferSize;
        void *picDevBuffer = nullptr;
        // 自定义函数ReadBinFile，根据昇腾AI软件栈的运行模式，申请对应的内存，再调用C++标准库中的函数将
图片数据读入内存
        ret = Utils::ReadBinFile(testFile[index], picDevBuffer, devBufferSize);

        // 将模型输入数据传入队列中，执行模型推理，-1是表示阻塞程序直到输入数据入队完成
        ret = acltdtEnqueueData(inputQid, picDevBuffer, devBufferSize, nullptr, 0, -1, 0);
        // 获取每个输出的大小
    size_t dataSize = aclmdlGetOutputSizeByIndex(modelDesc, 0);
    void *data = nullptr;
        size_t retDataSize = 0;
        // 为模型输出数据申请内存
    if (!g_isDevice) {
            aclError aclRet = aclrtMallocHost(&data, dataSize);
     } else {
        aclError aclRet = aclrtMalloc(&data, dataSize, ACL_MEM_MALLOC_NORMAL_ONLY);
     }
         // 等待模型推理执行完毕，从输出内存中获取结果数据，-1是表示阻塞程序直到推理输出数据入队完成
         ret = acltdtDequeueData(outputQid, data, dataSize, &retDataSize, nullptr, 0, -1);
         //将输出内存中的数据转换为float类型
         float *outData = NULL;
         outData = reinterpret_cast<float*>(data);

         //屏显每张图片的top5置信度的类别编号
         map<float, int, greater<float> > resultMap;
         for (int j = 0; j < len / sizeof(float); ++j) {
           resultMap[*outData] = j;
           outData++;
         }
         int cnt = 0;
         for (auto it = resultMap.begin(); it != resultMap.end(); ++it) {
           // print top 5
           if (++cnt > 5) {
             break;
           }
           INFO_LOG("top %d: index[%d] value[%lf]", cnt, it->second, it->first);
         }
         if (!g_isDevice) {
           aclError aclRet = aclrtFreeHost(picDevBuffer);
           aclrtFreeHost(data);
     } else {
        aclError aclRet = aclrtFree(picDevBuffer);
           aclrtFree(data);
     }
}

// 4. 卸载模型，并释放模型推理相关资源
aclmdlUnload(modelId);
aclmdlDestroyDesc(modelDesc);
acltdtDestroyQueue(inputQid);
acltdtDestroyQueue(outputQid);
acltdtDestroyQueueAttr(attr);
```



```
// 6. 释放运行管理资源
// 7. AscendCL去初始化
// ......
```

8.8 模型动态 AIPP 推理

8.8.1 使用约束

- **动态AIPP和动态Batch同时使用时**：
  - 调用aclmdlCreateAIPP接口设置batchSize时，batchSize要设置为最大batch size。
  - 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aclmdlSetAIPPSrcImageSize接口设置的宽、高与通过aclmdlSetDynamicHWSize接口设置的宽、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、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。
- **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。
- 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。
- AscendCL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**9 媒体数据处理（含图像/视频等）**。

8.8.2 动态 AIPP（单个动态 AIPP 输入）

本节介绍单个动态AIPP输入的模型，在执行模型推理时的关键接口、示例代码。

接口调用流程

动态AIPP场景下模型推理与**8 模型推理**的流程类似，都涉及AscendCL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**8 模型推理**的不同之处：

- **模型构建时**，需配置动态AIPP相关参数：

  构建模型时，需通过ATC工具的insert_op_conf参数配置动态AIPP模式。ATC工具的参数说明请参见《ATC工具使用指南》。

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aclmdlGetInputIndexByName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。

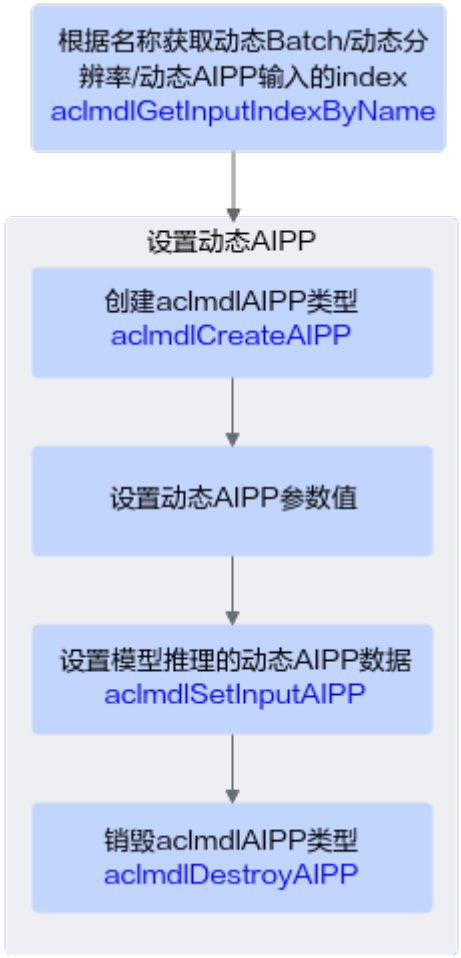
    说明

    ACL_DYNAMIC_AIPP_NAME是一个宏，宏的定义如下：
    ```
    #define ACL_DYNAMIC_AIPP_NAME "ascend_dynamic_aipp_data"
    ```

    ii. 调用aclmdlGetInputSizeByIndex根据index获取输入内存大小。
    iii. 调用aclrtMalloc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aclCreateDataBuffer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aclmdlCreateDataset接口创建aclmdlDataset类型的数据，并调用aclmdlAddDatasetBuffer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 8-8 接口调用流程



i. 调用aclmdlGetInputIndexByName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aclmdlCreateAIPP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，aclmdlSetAIPPSrcImageSize接口（设置原始图片的宽和高）必须调用。

4) 调用aclmdlSetInputAIPP接口设置模型推理时的动态AIPP数据。

5) 及时调用aclmdlDestroyAIPP接口销毁aclmdlAIPP类型。

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。

```
// 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modelDesc_，准备模型的输入数据input_和模型的输出数据output_

// 3.自定义函数，设置动态AIPP参数值
```



```
int  ModelSetDynamicAIPP()
{
    // 3.1 获取标识动态AIPP输入的index
    size_t index;
    // modelDesc_为aclmdlCreateDesc表示模型描述信息，根据1中加载成功的模型的ID，获取该模型的描述信息
    aclError ret = aclmdlGetInputIndexByName(modelDesc_, ACL_DYNAMIC_AIPP_NAME, &index);

    // 3.2 设置动态AIPP参数值
    uint64_t batchNumber = 1;
    aclmdlAIPP *aippDynamicSet = aclmdlCreateAIPP(batchNumber);
    ret = aclmdlSetAIPPSrcImageSize(aippDynamicSet, 256, 224);
    ret = aclmdlSetAIPPInputFormat(aippDynamicSet, ACL_YUV420SP_U8);
    ret = aclmdlSetAIPPCscParams(aippDynamicSet, 1, 256, 443, 0, 256, -86, -178, 256, 0, 350, 0, 0, 0, 0,
128, 128);
    ret = aclmdlSetAIPPRbuvSwapSwitch(aippDynamicSet, 0);
    ret = aclmdlSetAIPPDtcPixelMean(aippDynamicSet, 0, 0, 0, 0, 0);
    ret = aclmdlSetAIPPDtcPixelMin(aippDynamicSet, 0, 0, 0, 0, 0);
    ret = aclmdlSetAIPPPixelVarReci(aippDynamicSet, 1.0, 1.0, 1.0, 1.0, 0);
    ret = aclmdlSetAIPPCropParams(aippDynamicSet, 1, 2, 2, 224, 224, 0);
    ret = aclmdlSetInputAIPP(modelId_, input_, index, aippDynamicSet);
    ret = aclmdlDestroyAIPP(aippDynamicSet);

    // ......
}
// 4.自定义函数，执行模型
int ModelExecute(int index)
{
        aclError ret;
        // 4.1 调用自定义函数，设置动态AIPP参数值
    ret = ModelSetDynamicAIPP();
        // 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 = aclmdlExecute(modelId_, input_, output_);
        // ......
}

// 5.处理模型推理结果
// TODO
```

8.8.3 动态 AIPP（多个动态 AIPP 输入）

本节介绍多个动态AIPP输入的模型，在执行模型推理时的关键接口、示例代码。

接口调用流程

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**8.8.2 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aclmdlGetAippType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aclmdlSetAIPPByInputIndex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aclmdlGetInputNameByIndex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，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。

```
// 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modelDesc_，准备模型的输入数据input_和模型的输出数据output_
// ......

// 3.自定义函数，设置动态AIPP参数值
int  ModelSetDynamicAIPP()
{
    // 3.1 获取标识动态AIPP输入的index
    std::vector<size_t> dataNeedDynamicAipp;
    for (size_t index = 0; index < aclmdlGetNumInputs(modelDesc_); ++index) {
        aclmdlInputAippType aippType;
        size_t dynamicAttachedDataIndex;
        aclError ret = aclmdlGetAippType(modelId_, index, &aippType, &dynamicAttachedDataIndex);
        if (aippType == ACL_DATA_WITH_DYNAMIC_AIPP) {
            dataNeedDynamicAipp.push_back(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 (dataNeedDynamicAipp.size() != 2) {
        return -1;
    }
    // 创建第一个动态aipp配置参数
    uint64_t batchNumber1 = 1;
    aclmdlAIPP *aippDynamicSet1 = aclmdlCreateAIPP(batchNumber1);
    ret = aclmdlSetAIPPSrcImageSize(aippDynamicSet1, 256, 224);
    ret = aclmdlSetAIPPInputFormat(aippDynamicSet1, ACL_YUV420SP_U8);
    ret = aclmdlSetAIPPCscParams(aippDynamicSet1, 1, 256, 443, 0, 256, -86, -178, 256, 0, 350, 0, 0, 0, 0,
128, 128);
    ret = aclmdlSetAIPPRbuvSwapSwitch(aippDynamicSet1, 0);
    ret = aclmdlSetAIPPDtcPixelMean(aippDynamicSet1, 0, 0, 0, 0, 0);
    ret = aclmdlSetAIPPDtcPixelMin(aippDynamicSet1, 0, 0, 0, 0, 0);
    ret = aclmdlSetAIPPPixelVarReci(aippDynamicSet1, 1.0, 1.0, 1.0, 1.0, 0);
    ret = aclmdlSetAIPPCropParams(aippDynamicSet1, 1, 2, 2, 224, 224, 0);
    // 设置模型推理时的动态aipp参数值
    ret = aclmdlSetAIPPByInputIndex(modelId_, input_, dataNeedDynamicAipp[0], aippDynamicSet1);
    ret = aclmdlDestroyAIPP(aippDynamicSet1);

    // 创建第二个动态aipp配置参数
    uint64_t batchNumber2 = 2;
    aclmdlAIPP *aippDynamicSet2 = aclmdlCreateAIPP(batchNumber2);
    ret = aclmdlSetAIPPSrcImageSize(aippDynamicSet2, 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 = aclmdlSetAIPPByInputIndex(modelId_, input_, dataNeedDynamicAipp[1], aippDynamicSet2);
    ret = aclmdlDestroyAIPP(aippDynamicSet2);
}

// 4.自定义函数，执行模型
int ModelExecute(int index)
{
        aclError ret;
        // 4.1 调用自定义函数，设置动态AIPP参数值
    ret = ModelSetDynamicAIPP();
        // 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 = aclmdlExecute(modelId_, input_, output_);
        // ......
}

// 5.处理模型推理结果
// TODO
```

8.9 模型动态 Shape 输入推理

8.9.1 使用约束

- 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。
- 对同一个模型，以下方式，**只能选择其中一种**：
  - 调用aclmdlSetDatasetTensorDesc接口设置Shape范围
  - 调用aclmdlSetDynamicBatchSize接口设置动态Batch
  - 调用aclmdlSetDynamicHWSize接口设置动态分辨率
  - 调用aclmdlSetInputDynamicDims接口设置动态维度的维度值
- **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aclmdlGetOutputSizeByIndex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aclmdlCreateAIPP接口设置batchSize时，batchSize要设置为最大batch size。
  - 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aclmdlSetAIPPSrcImageSize接口设置的宽、高与通过aclmdlSetDynamicHWSize接口设置的宽、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、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。
- **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。
- **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。

8.9.2 设置 Shape 数据缓存，提升性能

可通过设置环境变量**HOST_CACHE_CAPACITY**配置动态Shape执行时的数据缓存功能，默认值为0，不开启数据缓存功能；配置为非零正整数时，例如**10**，系统会将最近出现较为频繁的**10**个输入Shape对应的部分执行数据缓存，已缓存Shape再次出现时，Host执行性能将得到提升，但Host内存占用会变多，具体涨幅与环境变量值、模型大小成正比。

环境变量配置示例如下：

```
export HOST_CACHE_CAPACITY=10
```

注意，HOST_CACHE_CAPACITY环境变量取值范围为：[1, INT32类型最大值]，超出INT32类型最大值（即2147483647），表示不开启数据缓存功能。

8.9.3 动态 Batch/动态分辨率/动态维度（设置多档维度值）

本节介绍动态Batch/动态分辨率/动态维度功能涉及的关键接口、接口调用流程及示例代码。

接口调用流程

动态Shape输入场景下模型推理与**8 模型推理**的流程类似，都涉及AscendCL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**8 模型推理**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AscendCL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AscendCL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AscendCL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aclmdlGetInputIndexByName接口根据输入名称（固定为ACL_DYNAMIC_TENSOR_NAME）获取模型中标识该输入的index。

        说明

        ACL_DYNAMIC_TENSOR_NAME是一个宏，宏的定义如下：
        ```
        #define ACL_DYNAMIC_TENSOR_NAME "ascend_mbatch_shape_data"
        ```

     ii. 调用aclmdlGetInputSizeByIndex根据index获取输入内存大小。

     iii. 调用aclrtMalloc接口根据**2.ii**中的大小申请内存。

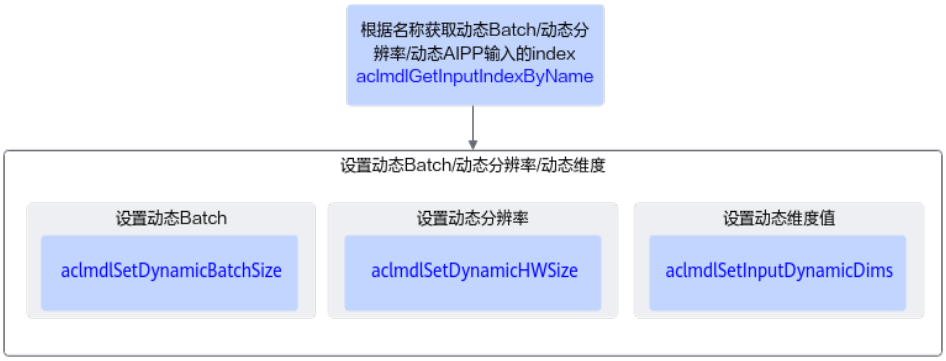
        申请动态Batch/动态分辨率/动态AIPP/动态维度输入对应的内存后，无需用户设置该内存中的数据（否则可能会导致业务异常），用户调用**2.ii**中的接口后，系统会自动向该内存中填入数据。

     iv. 调用aclCreateDataBuffer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aclmdlCreateDataset接口创建aclmdlDataset类型的数据，并调用aclmdlAddDatasetBuffer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 8-9 接口调用流程

  i. 调用aclmdlGetInputIndexByName接口根据输入名称（固定为ACL_DYNAMIC_TENSOR_NAME）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aclmdlSetDynamicBatchSize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aclmdlGetDynamicBatch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aclmdlSetDynamicHWSize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aclmdlGetDynamicHW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aclmdlSetInputDynamicDims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aclmdlGetInputDynamicDims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。

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，不可以直接拷贝编译运行，仅供参考。

```
// 1.模型加载，加载成功后，再设置动态Batch
// ......

// 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_
// ......

// 3.自定义函数，设置动态Batch
int  ModelSetDynamicInfo()
{
        size_t index;
        // 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ACL_DYNAMIC_TENSOR_NAME
        aclError ret = aclmdlGetInputIndexByName(modelDesc_, ACL_DYNAMIC_TENSOR_NAME, &index);
        // 3.2 设置Batch
        // modelId_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表示Batch数（此处以8为例）
        uint64_t batchSize = 8;
        ret = aclmdlSetDynamicBatchSize(modelId_, input_, index, batchSize);
        // ......
}

// 4.自定义函数，执行模型
int ModelExecute(int index)
{
        aclError ret;
        // 4.1 调用自定义函数，设置动态Batch
    ret = ModelSetDynamicInfo();
        // 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 = aclmdlExecute(modelId_, input_, output_);
        // ......
}
// 5.处理模型推理结果
// TODO
```

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，不可以直接拷贝编译运行，仅供参考。

```
// 1.模型加载，加载成功后，再设置动态分辨率
// ......

// 2.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_
// ......

// 3.自定义函数，设置动态分辨率
int  ModelSetDynamicInfo()
{
        size_t index;
                // 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
ACL_DYNAMIC_TENSOR_NAME
        aclError ret = aclmdlGetInputIndexByName(modelDesc_, ACL_DYNAMIC_TENSOR_NAME, &index);
                // 3.2 设置输入图片分辨率，modelId_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
                uint64_t height = 224;
        uint64_t width = 224;
        ret = aclmdlSetDynamicHWSize(modelId_, input_, index, height, width);
                // ......
}

// 4.自定义函数，执行模型
int ModelExecute(int index)
{
        aclError ret;
        // 4.1 调用自定义函数，设置动态分辨率
    ret = ModelSetDynamicInfo();
        // 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 = aclmdlExecute(modelId_, input_, output_);
        // ......
}

// 5.处理模型推理结果
// TODO
```

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，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。

```
// 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modelDesc_，准备模型的输入数据input_和模型的输出数据output_
// ......

// 3.自定义函数，设置动态维度
int  ModelSetDynamicInfo()
{
        size_t index;
        // 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ACL_DYNAMIC_TENSOR_NAME
        aclError ret = aclmdlGetInputIndexByName(modelDesc_, ACL_DYNAMIC_TENSOR_NAME, &index);
        // 3.2 设置具体档位信息，包括维度数dimCount和各个维度的数值，modelId_表示加载成功的模型的ID，
input_表示aclmdlDataset类型的数据，index表示标识动态维度输入的输入index
        aclmdlIODims currentDims;
        currentDims.dimCount = 4;
        currentDims.dims[0] = 8;
        currentDims.dims[1] = 3;
        currentDims.dims[2] = 224;
        currentDims.dims[3] = 224;
        ret = aclmdlSetInputDynamicDims(modelId_, input_, index, &currentDims);
        // ......
}

// 4.自定义函数，执行模型
int ModelExecute(int index)
{
        aclError ret;
        // 4.1 调用自定义函数，设置动态维度
    ret = ModelSetDynamicInfo();
        // 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 = aclmdlExecute(modelId_, input_, output_);
        // ......
}
// 5.处理模型推理结果
// TODO
```

8.9.4 动态 Shape 输入（设置 Shape 范围）

本节介绍动态Shape输入场景下，如何设置Shape范围，介绍其关键接口、接口调用流程及示例代码。

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aclmdlSetDatasetTensorDesc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aclmdlGetDatasetTensorDesc接口获取模型动态输出的Tensor描述信息，再进一步调用aclTensorDesc下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**8.3.1 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aclmdlGetInputSizeByIndex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aclmdlGetOutputSizeByIndex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aclmdlSetDatasetTensorDesc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aclCreateTensorDesc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**8.2 模型构建**。

5. 执行模型。

   例如，调用aclmdlExecute接口（同步接口）执行模型。

6. 获取模型执行的结果数据。

   调用aclmdlGetDatasetTensorDesc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。

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aclmdlGetInputSizeByIndex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aclmdlGetOutputSizeByIndex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
void SetTensorDesc()
{
    // ......
    // 创建Tensor描述信息
    // shape需要和给定的输入数据的shape一致
    int64_t shapes = {1，3，224，224};
    aclTensorDesc *inputDesc = aclCreateTensorDesc(ACL_FLOAT, 4, shapes, ACL_FORMAT_NCHW);
    // 设置index为0的动态输入的Tensor描述信息
    aclError ret = aclmdlSetDatasetTensorDesc(input_, inputDesc, 0);
    // ......
}
// 4.自定义函数，执行模型，并获取动态输出的Tensor描述信息
void ModelExecute()
{
    aclError ret;
    // 调用自定义接口，设置动态输入的Tensor描述信息
    SetTensorDesc()；
        // 执行模型
    ret = aclmdlExecute(modelId, input_, output_);
    // 获取index为0的动态输出的Tensor描述信息
        aclTensorDesc *outputDesc = aclmdlGetDatasetTensorDesc(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
    string outputFileName = ss.str();
        FILE *outputFile = fopen(outputFileName.c_str(), "wb");
    size_t outputDesc_size = aclGetTensorDescSize(outputDesc);
        aclDataBuffer *dataBuffer = aclmdlGetDatasetBuffer(output_, 0);
        void *data = aclGetDataBufferAddr(dataBuffer);
    void *outHostData = nullptr;
        // 调用aclrtGetRunMode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
        aclrtRunMode runMode;
        ret = aclrtGetRunMode(&runMode);
        if (runMode == ACL_HOST) {
            ret = aclrtMallocHost(&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 = aclrtMemcpy(outHostData, outputDesc_size, data, outputDesc_size,
ACL_MEMCPY_DEVICE_TO_HOST);
        fwrite(outHostData, outputDesc_size, sizeof(char), outputFile);
            ret = aclrtFreeHost(outHostData);
    } else {
        // if app is running in host, write model output data into result file
            fwrite(data, outputDesc_size, sizeof(char), outputFile);
    }
    fclose(outputFile);
        // ......
}

// 5.处理模型推理结果
// TODO
```

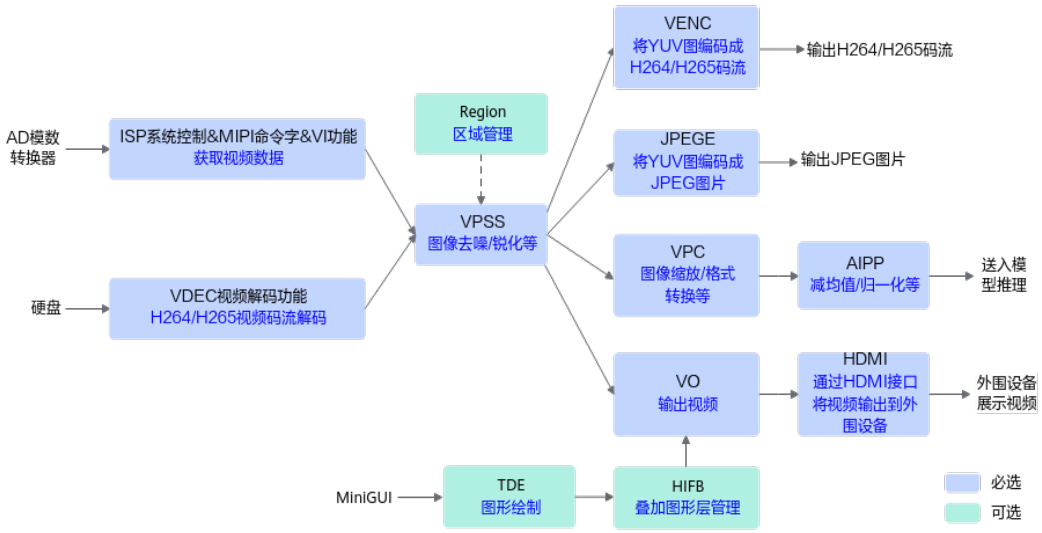

9 媒体数据处理（含图像/视频等）

9.1 媒体数据处理基础知识

本章主要介绍图像/视频/音频数据处理的具体功能、接口调用流程以及示例代码。

图像/视频/音频数据处理的典型功能介绍

图 9-1 图像/视频数据处理



各功能的介绍如下表所示，昇腾AI处理器对这些功能的支持度请参见**9.3 各版本功能支持度说明**（AIPP当前各版本均支持）。

功能	子功能模块	描述
获取视频数据	**ISP**（Image Signal Processing）系统控制	系统控制部分用于注册3A算法、注册Sensor驱动、初始化ISP firmware、运行ISP firmware、退出ISP firmware、配置ISP属性等功能。
	MIPI Rx ioctl命令字	MIPI Rx是一个支持多种差分视频输入接口的采集单元，通过combo-PHY接收MIPI/LVDS/sub-LVDS/HiSPi接口的数据，通过不同的功能模式配置，MIPI Rx可以支持多种速度和分辨率的数据传输需求，支持多种外部输入设备。
	VI（Vedio Input）	VI模块捕获视频图像，可对其做裁剪、防抖、颜色优化、亮度优化、噪声去除等处理，并输出YUV或RAW格式的图像数据。
展示视频数据	**VO**（Vedio Output）	VO模块接收VPSS处理后的输出图像，可进行播放控制等处理，最后按用户配置的输出协议（当前仅支持HDMI）输出给外围视频设备。 VO可配合TDE（Two Dimensional Engine）模块、HIFB（Hisilicon Framebuffer）模块，利用硬件分别进行图形绘制、叠加图形层管理。
	HDMI（High Definition Multimedia Interfac）	HDMI是全数字化影像和声音发送接口，可以发送未压缩的音频及视频信号。

功能	子功能模块	描述
	TDE（Two Dimensional Engine）	TDE是图形二维加速引擎，它利用硬件为 OSD（On Screen Display）和 GUI（Graphics User Interface）提供快速的图形绘制功能，主要有快速拷贝、快速色彩填充、模式填充（当前仅支持Alpha Blending操作）。
	HIFB（Hisilicon Framebuffer）	HIFB用于管理叠加图形层，它不仅提供Linux Framebuffer的基本功能，还在Linux Framebuffer的基础上增加图层显示起始位置修改、层间Alpha等扩展功能。
区域管理	Region	叠加在视频上的OSD (On Screen Display)和遮挡在视频上的色块统称为区域。**区域管理模块**，用于统一管理这些区域资源，用于在视频上显示一些特定信息（如通道号、时间戳等）、或在视频中填充色块用于遮挡，当前该功能需配合VPSS一起使用。
图像/视频数据处理	**VPSS**（Video Process Sub-System）	VPSS模块支持对输入图像进行统一预处理，如去噪、去隔行、裁剪等，然后再对各通道分别进行处理，如缩放、加边框等。
	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《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AscendCL提供的接口来设置，请参见**8.8 模型动态AIPP推理**中的介绍。

功能	子功能模块	描述
	DVPP（Digital Vision Pre-Processing）	DVPP是昇腾AI处理器内置的图像处理单元，通过AscendCL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、图像金字塔、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。
音频数据获取和输出	**AI**（Audio Input）	AI模块捕获音频数据 。
	AO（Audio Output）	通过ADEC模块解码后的音频数据，AO模块支持播放音频。
音频数据编解码	**AENC**（Audio Encoder）	通过AI模块获取的音频数据，AENC模块支持对其进行编码，输出音频码流。
	ADEC（Audio Decoder）	ADEC支持解码G.711a、G.711u等协议的音频码流，再通过AO模块播放音频。

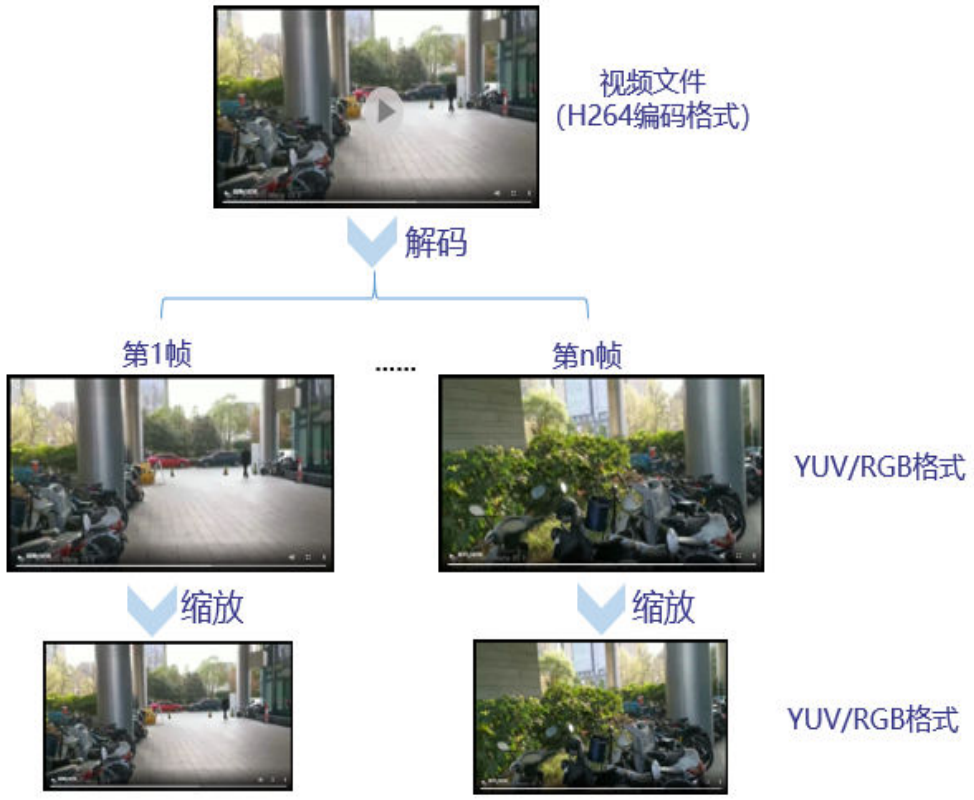
DVPP 图像/视频数据处理的典型使用场景

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。

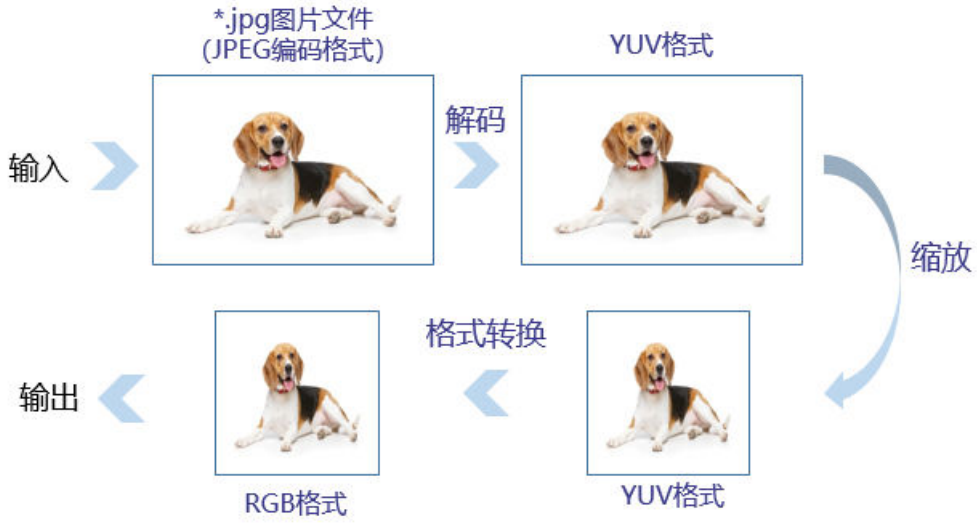
图 9-2 视频解码、缩放使用场景图



- **图片解码、缩放、格式转换**

  使用Resnet50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Resnet50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 9-3 图片解码、缩放、格式转换使用场景图



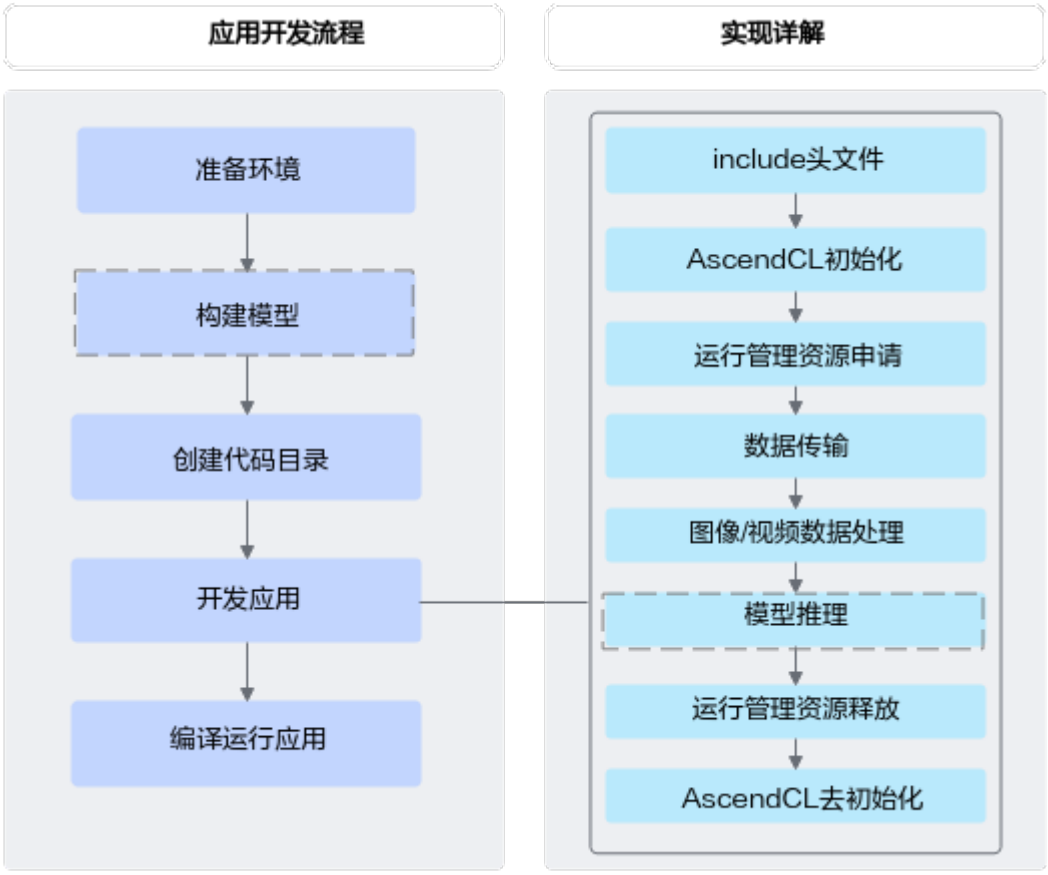
- **抠图、缩放、格式转换**

使用Resnet50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Resnet50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 9-4 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 9-5 开发流程



1. **准备环境**。

   请参见**4.3 准备开发和运行环境**。

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，供参考：

```
├App名称
├── model                // 该目录下存放模型文件
│   ├── xxx.json

├── data
│   ├── xxxxxx           // 测试数据

├── inc                 // 该目录下存放声明函数的头文件
│   ├── xxx.h

├── out                  // 该目录下存放输出结果

├── src
│   ├── xxx.json         // 系统初始化的配置文件
│   ├── CMakeLists.txt   // 编译脚本
│   ├── xxx.cpp          // 实现文件
```

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**8.2 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**。

   依赖的头文件和库文件的说明请参见**调用接口依赖的头文件和库文件说明**。

   如果应用中涉及模型推理，请参见**8.3 单Batch&静态Shape输入推理**、**10 更多特性**编写相应的代码。

5. **编译运行应用**，请参见**11 应用编译&运行**。

9.2 媒体数据处理 V1 与 V2 版本的差别

媒体数据处理接口有V1、V2两套，本节介绍两套接口的共同点、差异点。

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**V2版本的功能比V1版本更多，且两套接口不能混用**。

V2版本的功能比V1版本更多，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。
- 视频数据获取功能（ISP系统控制&MIPI命令字&VI功能）：仅V2版本接口支持。
- VPSS视频处理：仅V2版本接口支持。

- 音频相关功能，包括录音、播音、音量调节：仅V2版本接口支持。
- 视频数据展示功能（VO功能&HDMI外设）：仅V2版本接口支持。

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 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。

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 200/300/500 推理产品和Atlas 推理系列产品（Ascend 310P处理器）都支持V1版本接口，但部分使用约束存在差异，如果涉及Atlas 200/300/500 推理产品媒体数据处理V1->Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V1迁移的场景，请参见**15.4.2 Atlas 200/300/500 推理产品媒体数据处理V1->Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V1迁移指引**。

V1、V2这两套媒体数据处理接口，在功能约束、接口调用流程上存在差异，如果涉及Atlas 200/300/500 推理产品媒体数据处理V1->Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V2迁移的场景，请参见**15.4.3 Atlas 200/300/500 推理产品媒体数据处理V1->Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V2迁移指引**。

9.3 各版本功能支持度说明

昇腾AI处理器对媒体数据处理各功能的支持度如下表所示。

各标识的含义如下：

- √：支持。
- x：不支持。

版本	DVPP图像/视频处理（媒体数据处理V1）	DVPP图像/视频处理（媒体数据处理V2）	Camera场景视频数据获取和处理	NVR场景视频解码、处理和显示	NVR场景语音对讲	音频获取&音频播放
Atlas 200/300/500 推理产品	√	x	x	x	x	x

版本	DVPP图像/视频处理（媒体数据处理V1）	DVPP图像/视频处理（媒体数据处理V2）	Camera场景视频数据获取和处理	NVR场景视频解码、处理和显示	NVR场景语音对讲	音频获取&音频播放
Atlas 训练系列产品	VPC：√ JPEGD：√ JPEGE：√ PNGD：√ VDEC：√ VENC：x	x	x	x	x	x
Atlas 推理系列产品（Ascend 310P处理器）	√	√	x	x	x	x
Atlas 200/500 A2推理产品	√	√	√	√	√	√
Atlas A2 训练系列产品	VPC：√ JPEGD：√ JPEGE：√ PNGD：√ VDEC：√ VENC：x	VPC：√ JPEGD：√ JPEGE：√ PNGD：√ VDEC：√ VENC：x	x	x	x	x

9.4 DVPP 图像/视频处理（媒体数据处理 V1）

9.4.1 VPC 图像处理典型功能

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。

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。

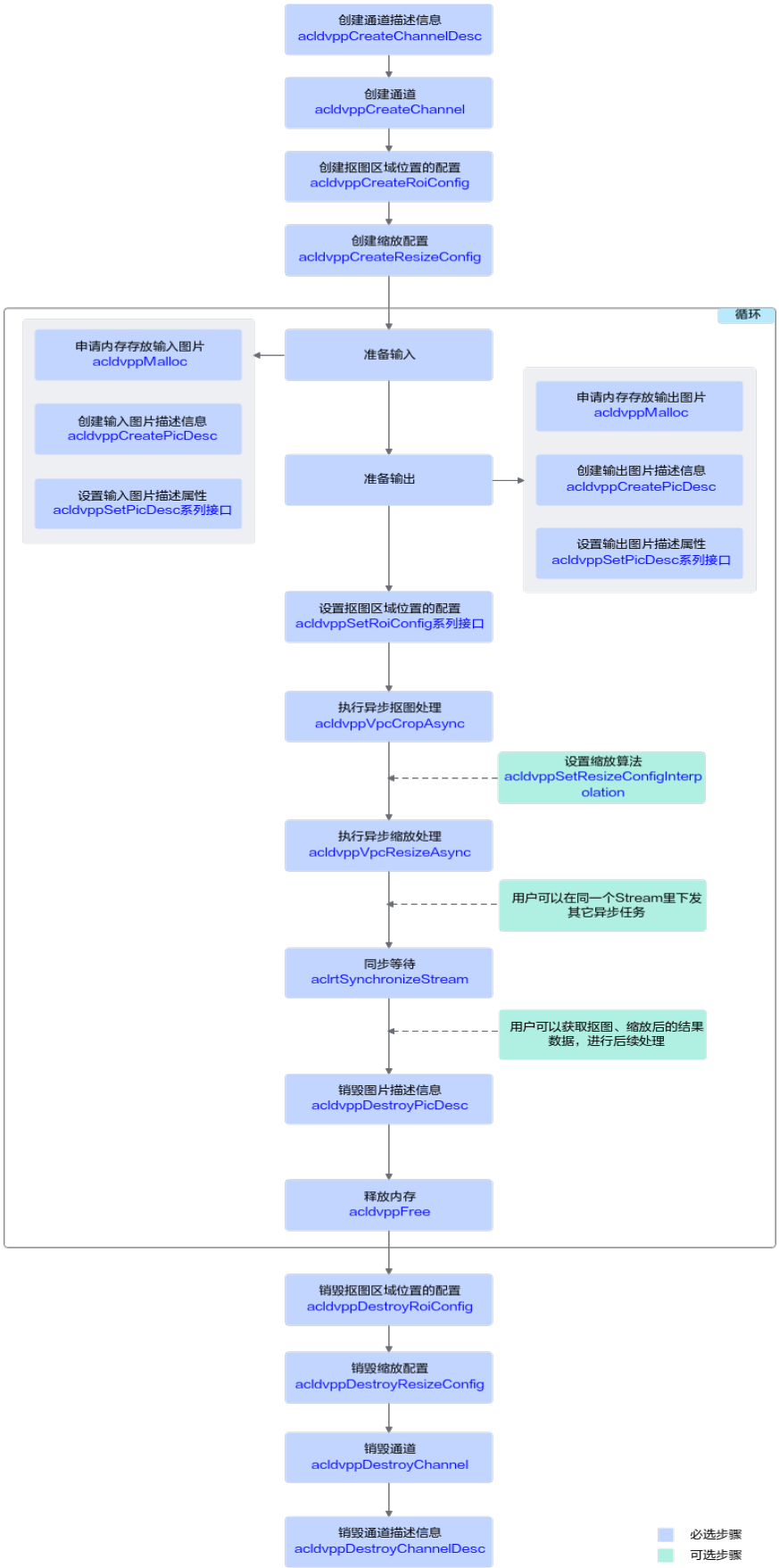
- **图片缩放示例代码**
- **格式转换示例代码**
- **抠图缩放（一图一框）示例代码**
- **抠图贴图缩放（一图一框）示例代码**

- **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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明。**

图 9-6 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acldvppCreateChannel接口**创建图片数据处理的通道**。

   创建图片数据处理的通道前，需先调用acldvppCreateChannelDesc接口创建通道描述信息。
2. 调用acldvppCreateRoiConfig接口、acldvppCreateResizeConfig接口分别**创建抠图区域位置的配置、缩放配置**，调用调用acldvppSetResizeConfigInterpolation接口指定缩放算法。
3. 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acldvppMalloc申请内存。
4. **执行抠图、缩放**。
   - 关于抠图：
     - 调用acldvppVpcCropAsync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cropArea不一致时会对图片再做一次缩放操作。
     - 当前系统还提供了acldvppVpcCropAndPasteAsync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cropArea的宽高与贴图区域pasteArea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acldvppVpcResizeAsync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aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
5. 调用acldvppFree接口**释放输入、输出内存**。
6. 调用acldvppDestroyRoiConfig接口、acldvppDestroyResizeConfig接口分别**销毁抠图区域位置的配置、缩放配置**。
7. 调用acldvppDestroyChannel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acldvppDestroyChannelDesc接口销毁通道描述信息。

图片缩放示例代码

您可以从13.7.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍图片缩放的代码逻辑，AscendCL初始化和去初始化请参见5 AscendCL初始化，运行管理资源申请与释放请参见6.1 运行管理资源申请与释放。

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。


```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3. 创建图片缩放配置数据、指定缩放算法
// resizeConfig_是acldvppResizeConfig类型
acldvppResizeConfig *resizeConfig_ = acldvppCreateResizeConfig();
aclError ret = acldvppSetResizeConfigInterpolation(resizeConfig, 0);

// 4. 创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 5. 创建图片数据处理的通道。
ret = acldvppCreateChannel(dvppChannelDesc_);

// 6. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t resizeInBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if（runMode == ACL_HOST）{
    // 申请Host内存vpcInHostBuffer
    void* vpcInHostBuffer = nullptr;
    vpcInHostBuffer = malloc(resizeInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInHostBuffer, resizeInBufferSize);
    // 申请Device内存resizeInDevBuffer_
    aclRet = acldvppMalloc(&resizeInDevBuffer_, resizeInBufferSize);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(resizeInDevBuffer_, resizeInBufferSize, vpcInHostBuffer, resizeInBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(vpcInHostBuffer);
} else {
    // 申请Device输入内存resizeInDevBuffer_
    ret = acldvppMalloc(&resizeInDevBuffer_, resizeInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, resizeInDevBuffer_, resizeInBufferSize);
}

// 7. 申请缩放输出内存resizeOutBufferDev_，内存大小\resizeOutBufferSize_根据计算公式得出
// outputPicWidth、outputPicHeight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
uint32_t resizeOutBufferSize_ = outputPicWidth * outputPicHeight * 3 / 2;
ret = acldvppMalloc(&resizeOutBufferDev_, resizeOutBufferSize_)

// 8. 创建缩放输入图片的描述信息，并设置各属性值
// resizeInputDesc_是acldvppPicDesc类型
resizeInputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(resizeInputDesc_, resizeInDevBuffer_);
acldvppSetPicDescFormat(resizeInputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(resizeInputDesc_, inputWidth_);
acldvppSetPicDescHeight(resizeInputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(resizeInputDesc_, inputWidthStride);
acldvppSetPicDescHeightStride(resizeInputDesc_, inputHeightStride);
acldvppSetPicDescSize(resizeInputDesc_, resizeInBufferSize);

// 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
// resizeOutputDesc_是acldvppPicDesc类型
resizeOutputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(resizeOutputDesc_, resizeOutBufferDev_);
acldvppSetPicDescFormat(resizeOutputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(resizeOutputDesc_, resizeOutputWidth_);
acldvppSetPicDescHeight(resizeOutputDesc_, resizeOutputHeight_);
acldvppSetPicDescWidthStride(resizeOutputDesc_, resizeOutputWidthStride);
acldvppSetPicDescHeightStride(resizeOutputDesc_, resizeOutputHeightStride);
```



```
acldvppSetPicDescSize(resizeOutputDesc_, resizeOutBufferSize_);

// 10. 执行异步缩放，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
ret = acldvppVpcResizeAsync(dvppChannelDesc_, resizeInputDesc_,
        resizeOutputDesc_, resizeConfig_, stream_);
ret = aclrtSynchronizeStream(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
acldvppDestroyPicDesc(resizeInputDesc_);
acldvppDestroyPicDesc(resizeOutputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存vpcOutHostBuffer
    void* vpcOutHostBuffer = nullptr;
    vpcOutHostBuffer = malloc(resizeOutBufferSize_);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(vpcOutHostBuffer, resizeOutBufferSize_, resizeOutBufferDev_,
resizeOutBufferSize_, ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(resizeInDevBuffer_);
    (void)acldvppFree(resizeOutBufferDev_);
    // 数据使用完成后，释放内存
    free(vpcOutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(resizeInDevBuffer_);
    (void)acldvppFree(resizeOutBufferDev_);
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 12. 释放运行管理资源（依次释放Stream、Context、Device）

// 13.AscendCL去初始化

// ....
```

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acldvppVpcCropAsync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。
- 如果仅仅做图片格式转换，也可以直接调用acldvppVpcConvertColorAsync接口。但Atlas 200/300/500 推理产品和Atlas 训练系列产品不支持调用该接口。

本节中的示例重点介绍格式的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3. 创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 4. 创建图片数据处理的通道。
aclError ret = acldvppCreateChannel(dvppChannelDesc_);

// 5. 申请输入内存（区分运行状态）
```



```
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t inBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if（runMode == ACL_HOST）{
    // 申请Host内存vpcInHostBuffer
    void* vpcInHostBuffer = nullptr;
    vpcInHostBuffer = malloc(inBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInHostBuffer, inBufferSize);
    // 申请Device内存inDevBuffer_
    aclRet = acldvppMalloc(&inDevBuffer_, inBufferSize);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(inDevBuffer_, inBufferSize, vpcInHostBuffer, inBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(vpcInHostBuffer);
} else {
    // 申请Device输入内存inDevBuffer_
    ret = acldvppMalloc(&inDevBuffer_, inBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inDevBuffer_, inBufferSize);
}

// 6. 申请色域转换输出内存outBufferDev_，内存大小outBufferSize_根据计算公式得出
// outputPicWidth、outputPicHeight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
uint32_t outBufferSize_ = outputPicWidth * outputPicHeight * 3 / 2
ret = acldvppMalloc(&outBufferDev_, outBufferSize_)

// 7. 创建色域转换输入图片的描述信息，并设置各属性值
// inputDesc_是acldvppPicDesc类型
inputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(inputDesc_, inDevBuffer_);
acldvppSetPicDescFormat(inputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(inputDesc_, inputWidth_);
acldvppSetPicDescHeight(inputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(inputDesc_, inputWidthStride);
acldvppSetPicDescHeightStride(inputDesc_, inputHeightStride);
acldvppSetPicDescSize(inputDesc_, inBufferSize);

// 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
// outputDesc_是acldvppPicDesc类型
outputDesc_= acldvppCreatePicDesc();
acldvppSetPicDescData(outputDesc_, outBufferDev_);
acldvppSetPicDescFormat(outputDesc_, PIXEL_FORMAT_YUV_400);
acldvppSetPicDescWidth(outputDesc_, outputWidth_);
acldvppSetPicDescHeight(outputDesc_, outputHeight_);
acldvppSetPicDescWidthStride(outputDesc_, outputWidthStride);
acldvppSetPicDescHeightStride(outputDesc_, outputHeightStride);
acldvppSetPicDescSize(outputDesc_, outBufferSize_);

// 9. 执行异步色域转换，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
ret = acldvppVpcConvertColorAsync(dvppChannelDesc_, inputDesc_, outputDesc_, stream_);
ret = aclrtSynchronizeStream(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
acldvppDestroyPicDesc(inputDesc_);
acldvppDestroyPicDesc(outputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存vpcOutHostBuffer
    void* vpcOutHostBuffer = nullptr;
    vpcOutHostBuffer = malloc(outBufferSize_);
```



```
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(vpcOutHostBuffer, outBufferSize_, outBufferDev_, outBufferSize_,
ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(outBufferDev_);
    // 数据使用完成后，释放内存
    free(vpcOutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(outBufferDev_);
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 11. 释放运行管理资源（依次释放Stream、Context、Device）

// 12.AscendCL去初始化

// ....
```

抠图缩放（一图一框）示例代码

调用acldvppVpcCropResizeAsync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，此外，还支持指定缩放算法。当输出图片区域与抠图区域cropArea不一致时会对图片再做一次缩放操作。

您可以从13.8.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍抠图缩放的代码逻辑，AscendCL初始化和去初始化请参见5 AscendCL初始化，运行管理资源申请与释放请参见6.1 运行管理资源申请与释放。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3. 创建缩放配置数据，并指定抠图区域的位置
// resizeConfig_是acldvppResizeConfig类型
resizeConfig_ = acldvppCreateResizeConfig();
aclError aclRet = acldvppSetResizeConfigInterpolation(resizeConfig_, 0);
// cropArea_是acldvppRoiConfig类型
cropArea_ = acldvppCreateRoiConfig(550, 749, 480, 679);

// 4. 创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 5. 创建图片数据处理的通道。
ret = acldvppCreateChannel(dvppChannelDesc_);

// 6. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t cropInBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if（runMode == ACL_HOST）{
    // 申请Host内存cropInHostBuffer
    void* cropInHostBuffer = nullptr;
    cropInHostBuffer = malloc(cropInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
```



```
    ReadPicFile(picName, cropInHostBuffer, cropInBufferSize);
    // 申请Device内存cropInDevBuffer_
    aclRet = acldvppMalloc(&cropInDevBuffer_, cropInBufferSize);
    // 通过aclrtMemcpy接口图片数据传输到Device
    aclRet = aclrtMemcpy(cropInDevBuffer_, cropInBufferSize, cropInHostBuffer, cropInBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(cropInHostBuffer);
} else {
    // 申请Device输入内存cropInDevBuffer_
    ret = acldvppMalloc(&cropInDevBuffer_, cropInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, cropInDevBuffer_, cropInBufferSize);
}

// 7. 申请Device输出内存cropOutBufferDev_，内存大小cropOutBufferSize_根据计算公式得出
// outputPicWidth、outputPicHeight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
uint32_t cropOutBufferSize_ = outputPicWidth * outputPicHeight * 3 / 2;
ret = acldvppMalloc(&cropOutBufferDev_, cropOutBufferSize_)

// 8. 创建输入图片的描述信息，并设置各属性值，cropInputDesc_是acldvppPicDesc类型
cropInputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(cropInputDesc_, cropInDevBuffer_);
acldvppSetPicDescFormat(cropInputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(cropInputDesc_, inputWidth_);
acldvppSetPicDescHeight(cropInputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(cropInputDesc_, inputWidthStride);
acldvppSetPicDescHeightStride(cropInputDesc_, inputHeightStride);
acldvppSetPicDescSize(cropInputDesc_, cropInBufferSize);

// 9. 创建输出图片的描述信息，并设置各属性值，cropOutputDesc_是acldvppPicDesc类型
// 如果抠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
cropOutputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(cropOutputDesc_, cropOutBufferDev_);
acldvppSetPicDescFormat(cropOutputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(cropOutputDesc_, OutputWidth_);
acldvppSetPicDescHeight(cropOutputDesc_, OutputHeight_);
acldvppSetPicDescWidthStride(cropOutputDesc_, OutputWidthStride);
acldvppSetPicDescHeightStride(cropOutputDesc_, OutputHeightStride);
acldvppSetPicDescSize(cropOutputDesc_, cropOutBufferSize_);

// 10. 执行异步抠图缩放，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都
完成
ret = acldvppVpcCropResizeAsync(dvppChannelDesc_, cropInputDesc_,
        cropOutputDesc_, cropArea_, resizeConfig_, stream_);
ret = aclrtSynchronizeStream(stream_);

// 11. 抠图贴图结束后，释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等
acldvppDestroyRoiConfig(cropArea_);
acldvppDestroyResizeConfig(resizeConfig_);
acldvppDestroyPicDesc(cropInputDesc_);
acldvppDestroyPicDesc(cropOutputDesc_);
if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存cropOutHostBuffer
    void* cropOutHostBuffer = nullptr;
    cropOutHostBuffer = malloc(cropOutBufferSize_);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(cropOutHostBuffer, cropOutBufferSize_, cropOutBufferDev_, cropOutBufferSize_,
ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(cropInDevBuffer_);
    (void)acldvppFree(cropOutBufferDev_);
    // 数据使用完成后，释放内存
    free(cropOutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作抠图结果后，释放Device侧内存
    (void)acldvppFree(cropInDevBuffer_);
    (void)acldvppFree(cropOutBufferDev_);
```



```
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 12. 释放运行管理资源（依次释放Stream、Context、Device）

// 13.AscendCL去初始化

// ....
```

抠图贴图缩放（一图一框）示例代码

调用acldvppVpcCropResizePasteAsync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，此外，还支持指定缩放算法。当抠图区域cropArea的宽高与贴图区域pasteArea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13.8.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍抠图贴图缩放的代码逻辑，AscendCL初始化和去初始化请参见5 AscendCL初始化，运行管理资源申请与释放请参见6.1 运行管理资源申请与释放。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3. 创建缩放配置数据，并指定抠图区域的位置、指定贴图区域的位置
// resizeConfig_是acldvppResizeConfig类型
resizeConfig_ = acldvppCreateResizeConfig();
aclError aclRet = acldvppSetResizeConfigInterpolation(resizeConfig_, 0);
// cropArea_和pasteArea_是acldvppRoiConfig类型
cropArea_ = acldvppCreateRoiConfig(512, 711, 512, 711);
pasteArea_ = acldvppCreateRoiConfig(16, 215, 16, 215);

// 4. 创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 5. 创建图片数据处理的通道。
ret = acldvppCreateChannel(dvppChannelDesc_);

// 6. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t vpcInBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if（runMode == ACL_HOST）{
    // 申请Host内存vpcInHostBuffer
    void* vpcInHostBuffer = nullptr;
    vpcInHostBuffer = malloc(vpcInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInHostBuffer, vpcInBufferSize);
    // 申请Device内存vpcInDevBuffer_
    aclRet = acldvppMalloc(&vpcInDevBuffer_, vpcInBufferSize);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(vpcInDevBuffer_, vpcInBufferSize, vpcInHostBuffer, vpcInBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(vpcInHostBuffer);
} else {
```



```
    // 申请Device输入内存vpcInDevBuffer_
    ret = acldvppMalloc(&vpcInDevBuffer_, vpcInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInDevBuffer_, vpcInBufferSize);
}

// 7. 申请输出内存vpcOutBufferDev_，内存大小vpcOutBufferSize_根据计算公式得出
// outputPicWidth、outputPicHeight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
uint32_t vpcOutBufferSize_ = outputPicWidth * outputPicHeight * 3 / 2;
ret = acldvppMalloc(&vpcOutBufferDev_, vpcOutBufferSize_)

// 8. 创建输入图片的描述信息，并设置各属性值
// 此处示例将解码的输出内存作为抠图贴图的输入，vpcInputDesc_是acldvppPicDesc类型
vpcInputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(vpcInputDesc_, decodeOutBufferDev_);
acldvppSetPicDescFormat(vpcInputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(vpcInputDesc_, inputWidth_);
acldvppSetPicDescHeight(vpcInputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(vpcInputDesc_, jpegOutWidthStride);
acldvppSetPicDescHeightStride(vpcInputDesc_, jpegOutHeightStride);
acldvppSetPicDescSize(vpcInputDesc_, jpegOutBufferSize);

// 9. 创建输出图片的描述信息，并设置各属性值
// 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
// vpcOutputDesc_是acldvppPicDesc类型
vpcOutputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(vpcOutputDesc_, vpcOutBufferDev_);
acldvppSetPicDescFormat(vpcOutputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(vpcOutputDesc_, dvppOutWidth);
acldvppSetPicDescHeight(vpcOutputDesc_, dvppOutHeight);
acldvppSetPicDescWidthStride(vpcOutputDesc_, dvppOutWidthStride);
acldvppSetPicDescHeightStride(vpcOutputDesc_, dvppOutHeightStride);
acldvppSetPicDescSize(vpcOutputDesc_, vpcOutBufferSize_);

// 10. 执行异步抠图贴图缩放，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
ret = acldvppVpcCropResizePasteAsync(dvppChannelDesc_, vpcInputDesc_,
        vpcOutputDesc_, cropArea_, pasteArea_, resizeConfig_, stream_);
ret = aclrtSynchronizeStream(stream_);

// 11. 抠图贴图结束后，释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等
acldvppDestroyRoiConfig(cropArea_);
acldvppDestroyRoiConfig(pasteArea_);
acldvppDestroyResizeConfig(resizeConfig_);
acldvppDestroyPicDesc(vpcInputDesc_);
acldvppDestroyPicDesc(vpcOutputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存vpcOutHostBuffer
    void* vpcOutHostBuffer = nullptr;
    vpcOutHostBuffer = malloc(vpcOutBufferSize_);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(vpcOutHostBuffer, vpcOutBufferSize_, vpcOutBufferDev_, vpcOutBufferSize_,
ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(vpcInDevBuffer_);
    (void)acldvppFree(vpcOutBufferDev_);
    // 数据使用完成后，释放内存
    free(vpcOutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(vpcInDevBuffer_);
    (void)acldvppFree(vpcOutBufferDev_);
}

acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;
```



```
// 12. 释放运行管理资源（依次释放Stream、Context、Device）

// 13.AscendCL去初始化

// ....
```

抠图贴图（一图多框）示例代码

调用acldvppVpcBatchCropAndPasteAsync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。当抠图区域cropArea的宽高与贴图区域pasteArea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。

本节中的示例重点介绍抠图贴图的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.指定批量抠图区域的位置、指定批量贴图区域的位置,cropAreas_和pasteAreas_是acldvppRoiConfig类型
acldvppRoiConfig *cropAreas_[2], pasteAreas_[2];
cropAreas_[0] = acldvppCreateRoiConfig(512, 711, 512, 711);
cropAreas_[1] = acldvppCreateRoiConfig(512, 711, 512, 711);
pasteAreas_[0] = acldvppCreateRoiConfig(16, 215, 16, 215);
pasteAreas_[1] = acldvppCreateRoiConfig(16, 215, 16, 215);

// 4. 创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 5. 创建图片数据处理的通道。
aclError ret = acldvppCreateChannel(dvppChannelDesc_);

// 6. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t vpcInBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if（runMode == ACL_HOST）{
    // 申请Host内存vpcInHostBuffer
    void* vpcInHostBuffer = nullptr;
    vpcInHostBuffer = malloc(vpcInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInHostBuffer, vpcInBufferSize);
    // 申请Device内存vpcInDevBuffer_
    aclRet = acldvppMalloc(&vpcInDevBuffer_, vpcInBufferSize);
    // 通过aclrtMemcpy接口将Host的图片数据传输到Device
    aclRet = aclrtMemcpy(vpcInDevBuffer_, vpcInBufferSize, vpcInHostBuffer, vpcInBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(vpcInHostBuffer);
} else {
    // 申请Device输入内存vpcInDevBuffer_
    ret = acldvppMalloc(&vpcInDevBuffer_, vpcInBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInDevBuffer_, vpcInBufferSize);
}
```



```
// 7. 申请输出内存vpcOutBufferDev_，内存大小vpcOutBufferSize_根据计算公式得出
// outputPicWidth、outputPicHeight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
uint32_t vpcOutBufferSize_ = outputPicWidth * outputPicHeight * 3 / 2;
ret = acldvppMalloc(&vpcOutBufferDev_, vpcOutBufferSize_)

// 8. 创建输入图片的描述信息，并设置各属性值
// 此处示例将解码的输出内存作为抠图贴图的输入，vpcInputDesc_是acldvppPicDesc类型
vpcInputBatchDesc_ = acldvppCreateBatchPicDesc(1);
vpcInputDesc_ = acldvppGetPicDesc(vpcInputBatchDesc_, 0);
acldvppSetPicDescData(vpcInputDesc_, decodeOutBufferDev_);
acldvppSetPicDescFormat(vpcInputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(vpcInputDesc_, inputWidth_);
acldvppSetPicDescHeight(vpcInputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(vpcInputDesc_, jpegOutWidthStride);
acldvppSetPicDescHeightStride(vpcInputDesc_, jpegOutHeightStride);
acldvppSetPicDescSize(vpcInputDesc_, jpegOutBufferSize);

// 9. 创建批量输出图片的描述信息，并设置各属性值
// 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
// vpcOutputDesc_是acldvppPicDesc类型
vpcOutputBatchDesc_ = acldvppCreateBatchPicDesc(2);
for (index=0; index<2; ++index){
    vecOutPtr_.push_back(vpcOutBufferDev_);
    vpcOutputDesc_ = acldvppGetPicDesc(vpcInputBatchDesc_, index);
   acldvppSetPicDescData(vpcOutputDesc_, vpcOutBufferDev_);
   acldvppSetPicDescFormat(vpcOutputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
   acldvppSetPicDescWidth(vpcOutputDesc_, dvppOutWidth);
   acldvppSetPicDescHeight(vpcOutputDesc_, dvppOutHeight);
   acldvppSetPicDescWidthStride(vpcOutputDesc_, dvppOutWidthStride);
   acldvppSetPicDescHeightStride(vpcOutputDesc_, dvppOutHeightStride);
   acldvppSetPicDescSize(vpcOutputDesc_, vpcOutBufferSize_);
}

// 10. 创建roiNums,每张图对应需要抠图和贴图的数量

uint32_ttotalNum = 0;
std::unique_ptr<uint32_t[]> roiNums(new (std::nothrow) uint32_t[1]);
roiNums[0]=2;
// 11. 执行异步抠图贴图，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都
完成
ret = acldvppVpcBatchCropAndPasteAsync(dvppChannelDesc_, vpcInputBatchDesc_, roiNums.get(), 1,
       vpcOutputBatchDesc_, cropAreas_, pasteAreas_, stream_);
ret = aclrtSynchronizeStream(stream_);

// 12. 抠图贴图结束后，释放资源，包括输入/输出图片的描述信息、输入/输出内存、通道描述信息、通道等
acldvppDestroyRoiConfig(cropAreas_[0]);
acldvppDestroyRoiConfig(cropAreas_[1]);
acldvppDestroyRoiConfig(pasteAreas_[0]);
acldvppDestroyRoiConfig(pasteAreas_[1]);
(void)acldvppFree(vpcInDevBuffer_);
for(index=0; index<2; ++index){
if（runMode == ACL_HOST）{
   // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
   // 申请Host内存vpcOutHostBuffer
   void* vpcOutHostBuffer = nullptr;
   vpcOutHostBuffer = malloc(vpcOutBufferSize_);
   // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
   aclRet = aclrtMemcpy(vpcOutHostBuffer, vpcOutBufferSize_, vpcOutBufferDev_, vpcOutBufferSize_,
ACL_MEMCPY_DEVICE_TO_HOST);
   // 释放掉输入输出的device内存
   (void)acldvppFree(vpcOutBufferDev_);
   // 数据使用完成后，释放内存
   free(vpcOutHostBuffer);
} else {
   // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存

   (void)acldvppFree(vpcOutBufferDev_);
}
}
```



```
acldvppDestroyBatchPicDesc(vpcInputDesc_);
acldvppDestroyBatchPicDesc(vpcOutputDesc_);
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 13. 释放运行管理资源（依次释放Stream、Context、Device）

// 14.AscendCL去初始化

// ....
```

9.4.2 JPEGD 图像解码

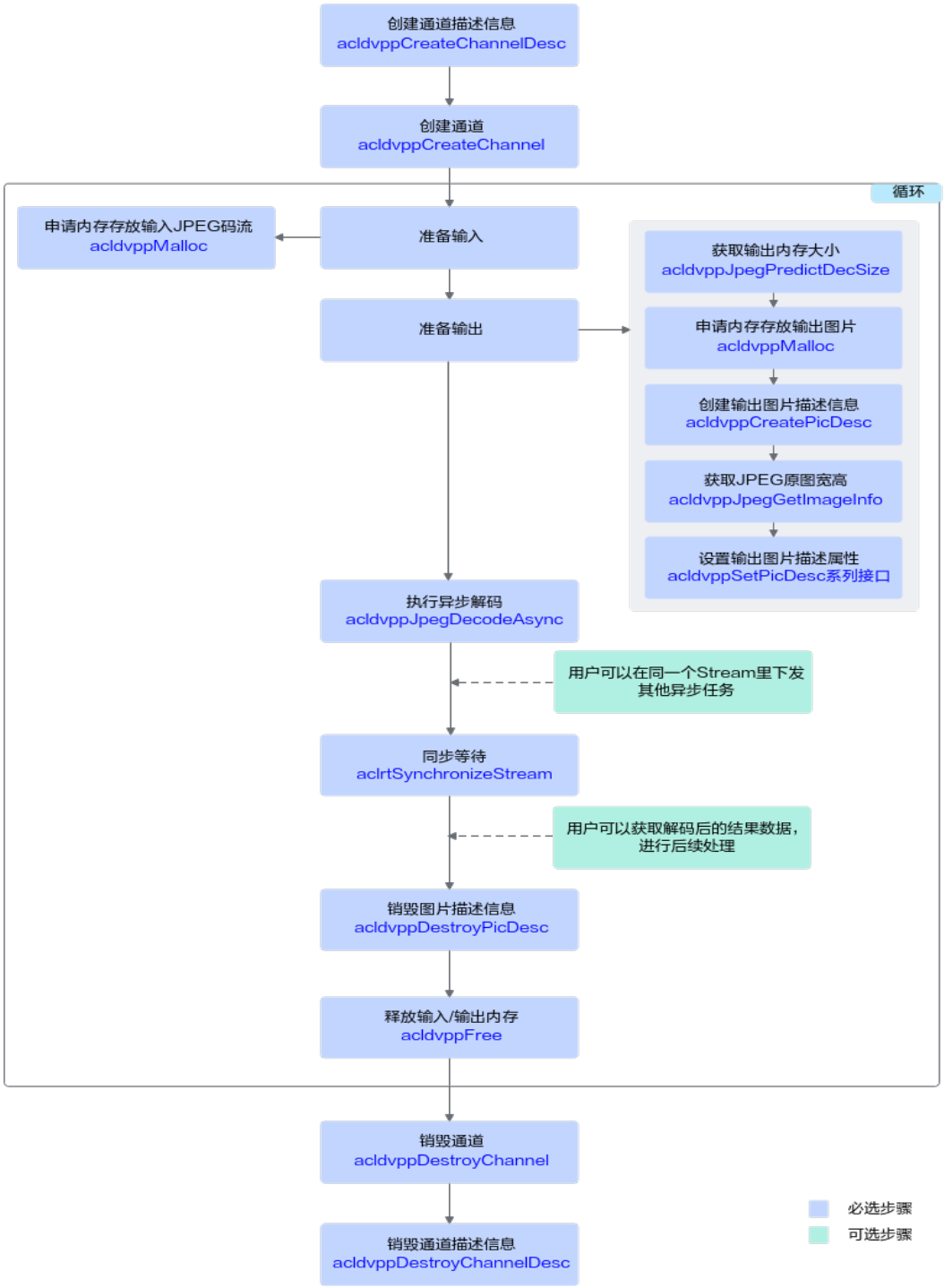
本节介绍JPEGD图片解码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

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。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-7 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acldvppCreateChannel接口**创建图片数据处理的通道**。

   创建图片数据处理的通道前，需先调用acldvppCreateChannelDesc接口创建通道描述信息。

2. 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acldvppMalloc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acldvppJpegPredictDecSize接口预估JPEG图片解码后所需的输出内存的大小。

实际输出内存大小可能与调用acldvppJpegPredictDecSize接口预估的内存大小存在差异，如果用户需要获取解码后的实际输出内存大小，需调用acldvppPicDesc类型下的acldvppGetPicDescSize接口获取。

3. 调用acldvppJpegDecodeAsync异步接口进行**解码**。

   对于异步接口，还需调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。

4. 在解码结束后，需及时调用acldvppFree接口**释放输入、输出内存**。
5. 调用acldvppDestroyChannel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acldvppDestroyChannelDesc接口销毁通道描述信息。

示例代码

您可以从13.8.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍JPEGD图片解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 4.创建图片数据处理的通道。
aclError ret = acldvppCreateChannel(dvppChannelDesc_);

// 5. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
if（runMode == ACL_HOST）{
    // 申请Host内存inputHostBuff，并将输入图片读入该地址，inDevBufferSize为读入图片大小
    void* inputHostBuff = nullptr;
    inputHostBuff = malloc(inDevBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputHostBuff, inDevBufferSize);
    // 申请Device内存inDevBuffer_
    aclRet = acldvppMalloc(&inDevBuffer_, inDevBufferSize);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(inDevBuffer_, inDevBufferSize, inputHostBuff, inDevBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);

} else {
    // 申请Device输入内存inDevBuffer_
    ret = acldvppMalloc(&inDevBuffer_, inBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inDevBuffer_, inBufferSize);
}

// 6. 申请解码输出内存decodeOutDevBuffer_
// 预估JPEGD处理结果所需的内存大小
```



```
uint32_t decodeOutBufferSize = 0;
ret = acldvppJpegPredictDecSize(inputHostBuff, inDevBufferSize, PIXEL_FORMAT_YVU_SEMIPLANAR_420，
&decodeOutBufferSize)
ret = acldvppMalloc(&decodeOutDevBuffer_, decodeOutBufferSize)
// 及时释放内存
free(inputHostBuff);

// 7. 创建解码输出图片的描述信息，设置各属性值
// decodeOutputDesc是acldvppPicDesc类型
decodeOutputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(decodeOutputDesc_, decodeOutDevBuffer_);
acldvppSetPicDescFormat(decodeOutputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(decodeOutputDesc_, inputWidth_);
acldvppSetPicDescHeight(decodeOutputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(decodeOutputDesc_, decodeOutWidthStride);
acldvppSetPicDescHeightStride(decodeOutputDesc_, decodeOutHeightStride);
acldvppSetPicDescSize(decodeOutputDesc_, decodeOutBufferSize);

// 8. 执行异步解码，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
ret = acldvppJpegDecodeAsync(dvppChannelDesc_, inDevBuffer_, inDevBufferSize, decodeOutputDesc_,
stream_);
ret = aclrtSynchronizeStream(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
acldvppDestroyPicDesc(decodeOutputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存vpcOutHostBuffer
    void* vpcOutHostBuffer = nullptr;
    vpcOutHostBuffer = malloc(decodeOutBufferSize);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(vpcOutHostBuffer, decodeOutBufferSize, decodeOutDevBuffer_,
decodeOutBufferSize, ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(decodeOutDevBuffer_);
    // 数据使用完成后，释放内存
    free(vpcOutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(decodeOutDevBuffer_);
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 10. 释放运行管理资源（依次释放Stream、Context、Device）

// 11.AscendCL去初始化

// ....
```

9.4.3 JPEGE 图片编码

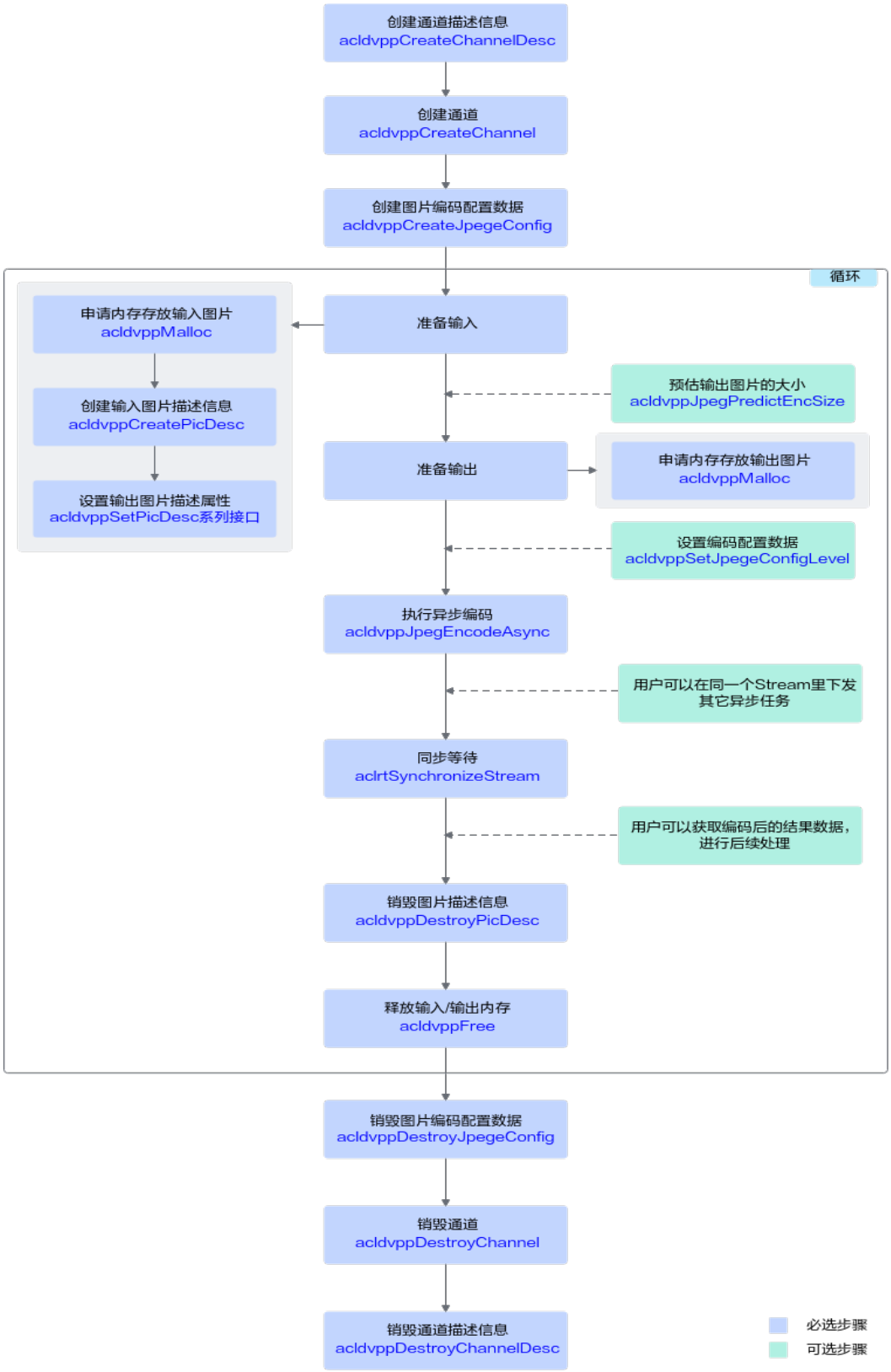
本节介绍JPEGE图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

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。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-8 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acldvppCreateChannel接口**创建图片数据处理的通道**。

   创建图片数据处理的通道前，需先调用acldvppCreateChannelDesc接口创建通道描述信息。
2. 调用acldvppCreateJpegeConfig接口**创建图片编码配置数据**。
3. 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acldvppMalloc申请内存。

   在申请输出内存前，可调用acldvppJpegPredictEncSize接口根据输入图片描述信息、图片编码配置数据可预估图片编码后所需的输出内存的大小。

   实际输出内存大小可能与调用acldvppJpegPredictEncSize接口预估的内存大小存在差异，如果用户需要获取编码后的实际输出内存大小，可通过acldvppJpegEncodeAsync接口的出参size获取。
4. 调用acldvppJpegEncodeAsync异步接口进行**编码**。

   对于异步接口，还需调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
5. 调用acldvppDestroyJpegeConfig接口**销毁图片编码配置数据**。
6. 在编码结束后，需及时调用acldvppFree接口**释放输入、输出内存**。
7. 调用acldvppDestroyChannel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acldvppDestroyChannelDesc接口销毁通道描述信息。

示例代码

您可以从13.8.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍JPEGE图片编码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 4.创建图片数据处理的通道
aclRet = acldvppCreateChannel(dvppChannelDesc_);

// 5.申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
// inputPicWidth、inputPicHeight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
uint32_t PicBufferSize = inputPicWidth * inputPicHeight * 3 / 2;
if (runMode == ACL_HOST) {
    // 申请Host内存vpcInHostBuffer
    void* vpcInHostBuffer = nullptr;
    vpcInHostBuffer = malloc(PicBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, vpcInHostBuffer, PicBufferSize);
    // 申请Device内存inDevBuffer_
    aclRet = acldvppMalloc(&inDevBuffer_, PicBufferSize);
```



```
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(inDevBuffer_, PicBufferSize, vpcInHostBuffer, PicBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(vpcInHostBuffer);
} else {
    // 申请Device输入内存inDevBuffer_
    ret = acldvppMalloc(&inDevBuffer_, PicBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inDevBuffer_, PicBufferSize);
}

// 6. 创建编码输入图片的描述信息，并设置各属性值
// encodeInputDesc_是acldvppPicDesc类型
encodeInputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(encodeInputDesc_, reinterpret_cast<void *>(inDevBuffer_));
acldvppSetPicDescFormat(encodeInputDesc_, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
acldvppSetPicDescWidth(encodeInputDesc_, inputWidth_);
acldvppSetPicDescHeight(encodeInputDesc_, inputHeight_);
acldvppSetPicDescWidthStride(encodeInputDesc_, encodeInWidthStride);
acldvppSetPicDescHeightStride(encodeInputDesc_, encodeInHeightStride);
acldvppSetPicDescSize(encodeInputDesc_, inDevBufferSizeE_);

// 7. 创建图片编码配置数据，设置编码质量
// 编码质量范围[0, 100]，其中level 0编码质量与level 100差不多，而在[1, 100]内数值越小输出图片质量越差。
jpegeConfig_ = acldvppCreateJpegeConfig();
acldvppSetJpegeConfigLevel(jpegeConfig_, 100);

// 8. 申请输出内存，申请Device内存encodeOutBufferDev_,存放编码后的输出数据
uint32_t outBufferSize= 0;
ret = acldvppJpegPredictEncSize(encodeInputDesc_, jpegeConfig_, &outBufferSize);
ret = acldvppMalloc(&encodeOutBufferDev_, outBufferSize)

// 9. 执行异步编码，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
aclRet = acldvppJpegEncodeAsync(dvppChannelDesc_, encodeInputDesc_, encodeOutBufferDev_,
          &outBufferSize, jpegeConfig_, stream_);
aclRet = aclrtSynchronizeStream(stream_);

// 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
acldvppDestroyPicDesc(encodeInputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存outputHostBuffer
    void* outputHostBuffer = nullptr;
    outputHostBuffer = malloc(outBufferSize_);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(outputHostBuffer, outBufferSize_, encodeOutBufferDev_, outBufferSize_,
ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(inputDevBuff);
    (void)acldvppFree(encodeOutBufferDev_);
    // 数据使用完成后，释放内存
    free(outputHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(inputDevBuff);
    (void)acldvppFree(encodeOutBufferDev_);
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 11. 释放运行管理资源（依次释放Stream、Context、Device）

// 12.AscendCL去初始化

// ....
```


9.4.4 PNGD 图片解码

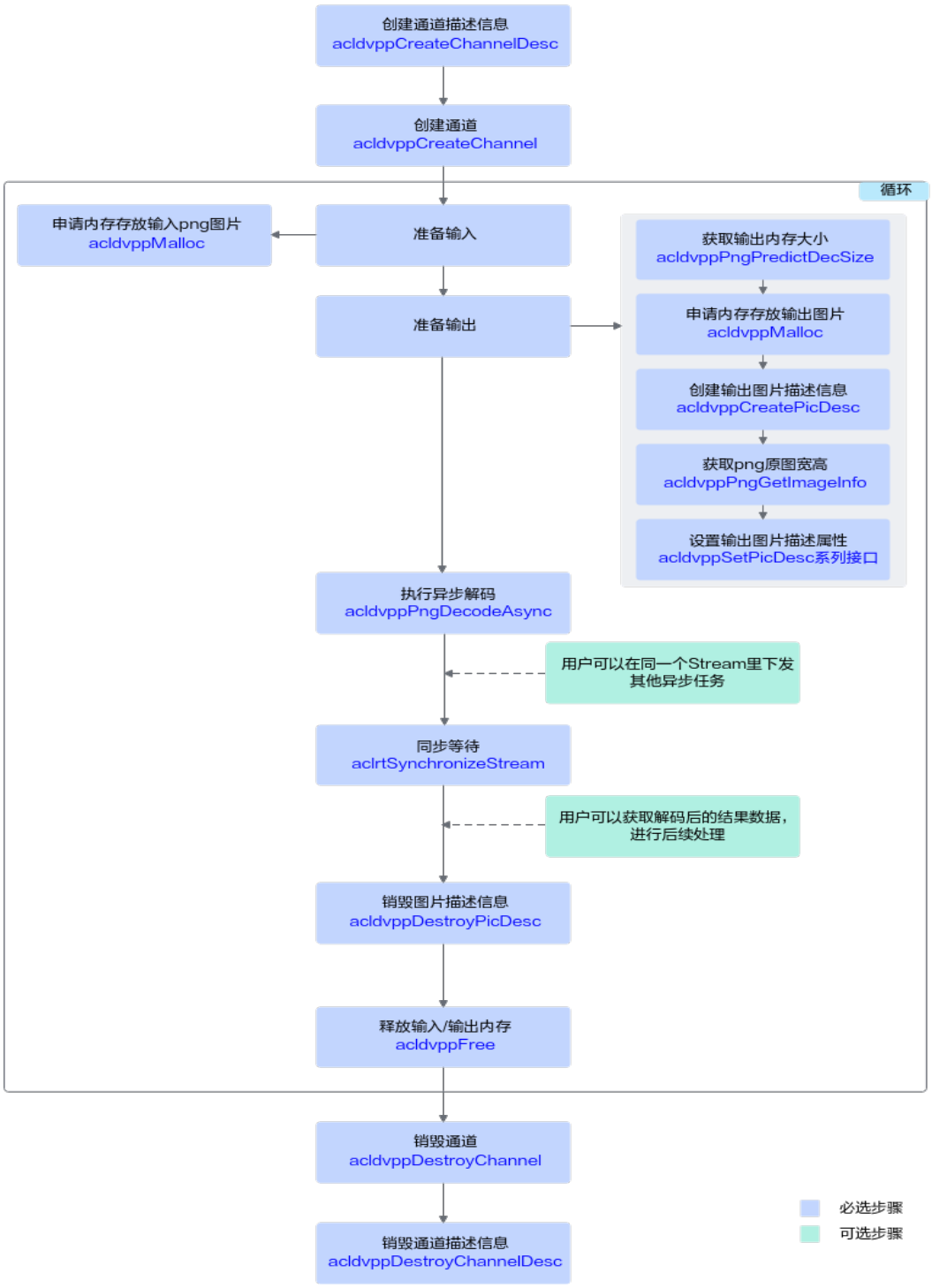
本节介绍PNGD图片编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

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。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-9 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acldvppCreateChannel接口**创建图片数据处理的通道**。

   创建图片数据处理的通道前，需先调用acldvppCreateChannelDesc接口创建通道描述信息。

2. 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acldvppMalloc申请内存。

   在申请输出内存前，可调用acldvppPngPredictDecSize接口根据存放png图片数据的内存计算出png图片解码后所需的输出内存的大小。

3. 调用acldvppPngDecodeAsync异步接口进行**解码**。

   对于异步接口，还需调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。

4. 在解码结束后，需及时调用acldvppFree接口**释放输入、输出内存**。

5. 调用acldvppDestroyChannel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acldvppDestroyChannelDesc接口销毁通道描述信息。

示例代码

本节中的示例重点介绍PNGD图片解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.创建图片数据处理通道时的通道描述信息，dvppChannelDesc_是acldvppChannelDesc类型
dvppChannelDesc_ = acldvppCreateChannelDesc();

// 4.创建图片数据处理的通道。
aclError ret = acldvppCreateChannel(dvppChannelDesc_);

// 5. 申请输入内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
if（runMode == ACL_HOST）{
    // 申请Host内存inputHostBuff，并将输入图片读入该地址，inDevBufferSize为读入图片大小
    void* inputHostBuff = nullptr;
    inputHostBuff = malloc(inDevBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputHostBuff, inDevBufferSize);
    // 申请Device内存inDevBuffer_
    aclRet = acldvppMalloc(&inDevBuffer_, inDevBufferSize);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(inDevBuffer_, inDevBufferSize, inputHostBuff, inDevBufferSize,
ACL_MEMCPY_HOST_TO_DEVICE);

} else {
    // 申请Device输入内存inDevBuffer_
    ret = acldvppMalloc(&inDevBuffer_, inDevBufferSize);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inDevBuffer_, inDevBufferSize);
}

// 6. 申请解码输出内存decodeOutDevBuffer_
// 计算PNGD处理结果所需的内存大小
uint32_t decodeOutBufferSize = 0;
ret = acldvppPngPredictDecSize(inputHostBuff, inDevBufferSize, PIXEL_FORMAT_RGB_888，
&decodeOutBufferSize)
ret = acldvppMalloc(&decodeOutDevBuffer_, decodeOutBufferSize)
```



```
// 及时释放内存
free(inputHostBuff);

// 7. 创建解码输出图片的描述信息，设置各属性值
// decodeOutputDesc是acldvppPicDesc类型
decodeOutputDesc_ = acldvppCreatePicDesc();
acldvppSetPicDescData(decodeOutputDesc_, decodeOutDevBuffer_);
acldvppSetPicDescFormat(decodeOutputDesc_, PIXEL_FORMAT_RGB_888);
acldvppSetPicDescSize(decodeOutputDesc_, decodeOutBufferSize);

// 8. 执行异步解码，再调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成
ret = acldvppPngDecodeAsync(dvppChannelDesc_, inDevBuffer_, inDevBufferSize, decodeOutputDesc_,
stream_);
ret = aclrtSynchronizeStream(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
acldvppDestroyPicDesc(decodeOutputDesc_);

if（runMode == ACL_HOST）{
    // 该模式下，由于处理结果在Device侧，因此需要调用内存复制接口传输结果数据后，再释放Device侧内存
    // 申请Host内存OutHostBuffer
    void* OutHostBuffer = nullptr;
    OutHostBuffer = malloc(decodeOutBufferSize);
    // 通过aclrtMemcpy接口将Device的处理结果数据传输到Host
    aclRet = aclrtMemcpy(OutHostBuffer, decodeOutBufferSize, decodeOutDevBuffer_,
decodeOutBufferSize, ACL_MEMCPY_DEVICE_TO_HOST);
    // 释放掉输入输出的device内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(decodeOutDevBuffer_);
    // 数据使用完成后，释放内存
    free(OutHostBuffer);
} else {
    // 此时运行在device侧，处理结果也在Device侧，可以根据需要操作处理结果后，释放Device侧内存
    (void)acldvppFree(inDevBuffer_);
    (void)acldvppFree(decodeOutDevBuffer_);
}
acldvppDestroyChannel(dvppChannelDesc_);
(void)acldvppDestroyChannelDesc(dvppChannelDesc_);
dvppChannelDesc_ = nullptr;

// 10. 释放运行管理资源（依次释放Stream、Context、Device）

// 11.AscendCL去初始化

// ....
```

9.4.5 VDEC 视频解码

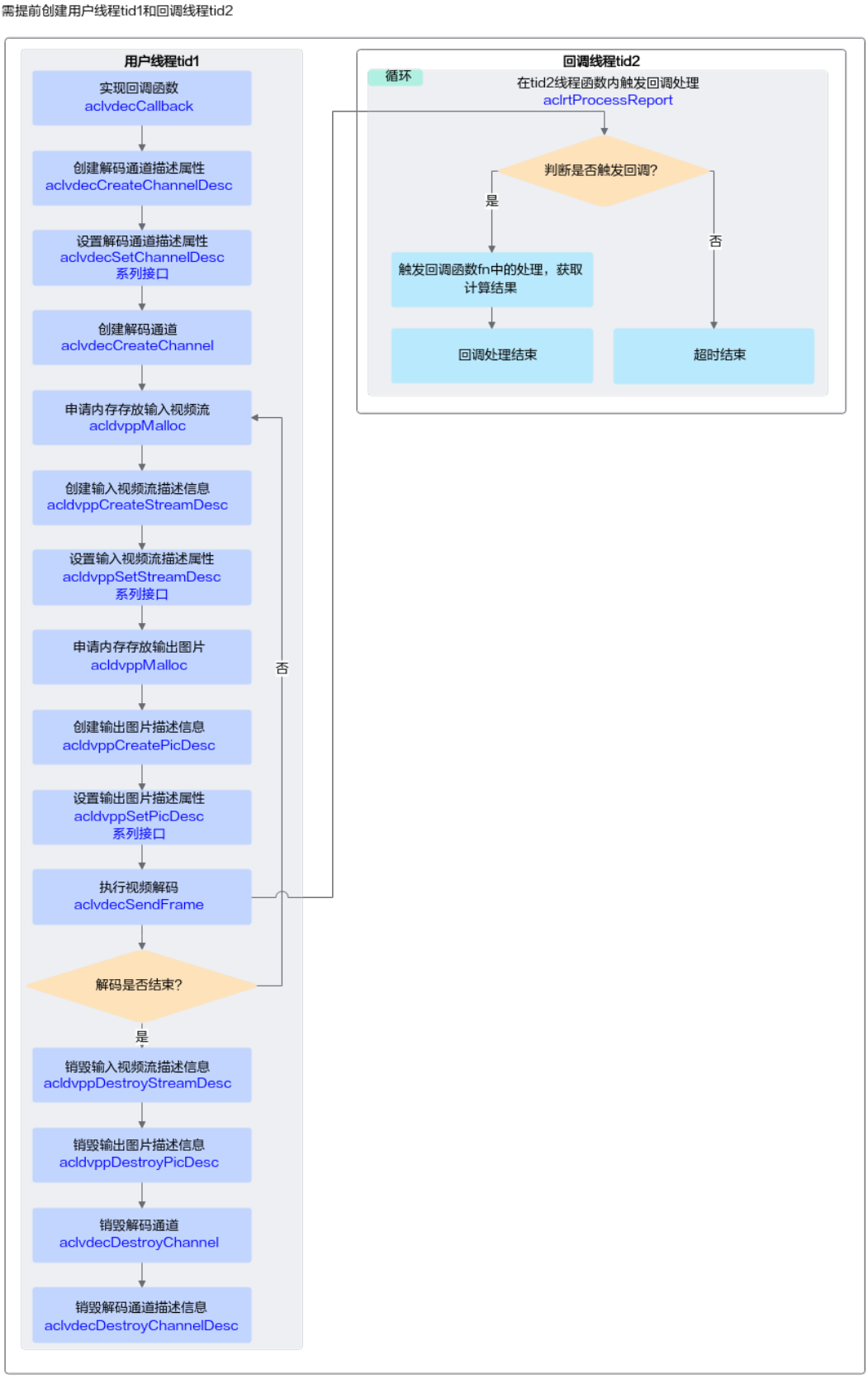
本节介绍VDEC视频编码的接口调用流程，同时配合示例代码辅助理解该接口调用流程。

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。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-10 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aclvdecCreateChannel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aclvdecCreateChannelDesc接口**创建通道描述信息**。
     ii. 调用aclvdecSetChannelDesc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aclvdecCallback。

           在回调函数内，用户需调用acldvppGetPicDescRetCode接口获取retCode返回码判断是否解码成功，retCode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aclrtProcessReport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aclvdecSetChannelDescOutPicFormat接口设置输出格式，则默认使用YUV420SP NV12。
   - aclvdecCreateChannel接口内部封装了如下接口，无需用户单独调用：
     i. aclrtCreateStream接口：显式创建Stream，VDEC内部使用。
     ii. aclrtSubscribeReport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aclvdecSetChannelDesc系列接口时指定的。
2. **调用aclvdecSendFrame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acldvppCreateStreamDesc接口创建输入视频码流描述信息，并调用acldvppSetStreamDesc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acldvppCreatePicDesc接口创建输出图片描述信息，并调用acldvppSetPicDesc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：**

     aclvdecSendFrame接口内部封装了aclrtLaunchCallback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aclrtLaunchCallback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retCode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aclvdecSendFrame接口的userData参数处定义，然后解码的帧序号可以通过userData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aclvdecSendSkippedFrame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output为nullptr。

3. **调用aclvdecDestroyChannel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。
   - aclvdecDestroyChannel接口内部封装了如下接口，无需用户单独调用：
     - aclrtUnSubscribeReport接口：取消线程注册（Stream上的回调函数不再由指定线程处理）。
     - aclrtDestroyStream接口：销毁Stream。
   - 销毁通道后，需调用aclvdecDestroyChannelDesc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13.9.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍VDEC视频解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.创建回调函数
void callback(acldvppStreamDesc *input, acldvppPicDesc *output, void *userdata)
{
    static int count = 1;
    if (output != nullptr) {
        // 获取VDEC解码的输出内存，调用自定义函数WriteToFile将输出内存中的数据写入文件后，再调用acldvppFree接口释放输出内存
        void *vdecOutBufferDev = acldvppGetPicDescData(output);
        if (vdecOutBufferDev != nullptr) {
            // 0: vdec success; others, vdec failed
            // retCode为0表示解码成功，为1表示解码失败
            int retCode = acldvppGetPicDescRetCode(output);
            if (retCode == 0) {
                // process task: write file
                uint32_t size = acldvppGetPicDescSize(output);
                std::string fileNameSave = "outdir/image" + std::to_string(count);
                // vdec输出结果在device侧，在WriteToFile方法中进行下述处理
                if (!Utils::WriteToFile(fileNameSave.c_str(), vdecOutBufferDev, size)) {
                    ERROR_LOG("write file failed.");
                }
            } else {
                ERROR_LOG("vdec decode frame failed.");
            }

            // free output vdecOutBufferDev
            aclError ret = acldvppFree(vdecOutBufferDev);
        }
        // 释放acldvppPicDesc类型的数据，表示解码后输出图片描述数据
        aclError ret = acldvppDestroyPicDesc(output);
    }

    // free input vdecInBufferDev and destroy stream desc
    if (input != nullptr) {
        void *vdecInBufferDev = acldvppGetStreamDescData(input);
        if (vdecInBufferDev != nullptr) {
            aclError ret = acldvppFree(vdecInBufferDev);
        }
```



```
        // 释放acldvppStreamDesc类型的数据，表示解码的输入码流描述数据
        aclError ret = acldvppDestroyStreamDesc(input);
    }

    INFO_LOG("success to callback %d.", count);
    count++;
}

// 4.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。
// vdecChannelDesc_是aclvdecChannelDesc类型
vdecChannelDesc_ = aclvdecCreateChannelDesc();
ret = aclvdecSetChannelDescChannelId(vdecChannelDesc_, 10);
ret = aclvdecSetChannelDescThreadId(vdecChannelDesc_, threadId_);
ret = aclvdecSetChannelDescCallback(vdecChannelDesc_, callback);
// 示例中使用的是H265_MAIN_LEVEL视频编码协议
ret = aclvdecSetChannelDescEnType(vdecChannelDesc_, static_cast<acldvppStreamFormat>(enType_));
// 示例中使用的是PIXEL_FORMAT_YVU_SEMIPLANAR_420
ret = aclvdecSetChannelDescOutPicFormat(vdecChannelDesc_, static_cast<acldvppPixelFormat>(format_));

// 5.创建视频码流处理的通道
ret = aclvdecCreateChannel(vdecChannelDesc_);

// 6.申请输入码流内存（区分运行状态）
// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
if（runMode == ACL_HOST）{
    // 申请Host内存inputHostBuff
    void* inputHostBuff= nullptr;
    // inBufferSize_为输入码流大小
    inputHostBuff= malloc(inBufferSize_);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputHostBuff, inBufferSize_);
    // 申请Device内存inBufferDev_
    aclRet = acldvppMalloc(&inBufferDev_, inBufferSize_);
    // 通过aclrtMemcpy接口将输入图片数据传输到Device
    aclRet = aclrtMemcpy(inBufferDev_, inBufferSize_, inputHostBuff, inBufferSize_,
ACL_MEMCPY_HOST_TO_DEVICE);
    // 数据传输完成后，及时释放内存
    free(inputHostBuff);
} else {
    // 申请Device输入内存dataDev， StreamBufferSize为输入码流大小
    ret = acldvppMalloc(&inBufferDev_, inBufferSize_);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inBufferDev_, inBufferSize_);
}

// 7.循环10次执行视频解码，输出10张YUV420SP NV12格式的图片
int rest_len = 10;
int32_t count = 0;
while (rest_len > 0) {
    // 7.1 创建输入视频码流描述信息，设置码流信息的属性
    streamInputDesc_ = acldvppCreateStreamDesc();
    // inBufferDev_表示Device存放输入视频数据的内存，inBufferSize_表示内存大小
    ret = acldvppSetStreamDescData(streamInputDesc_, inBufferDev_);
    ret = acldvppSetStreamDescSize(streamInputDesc_, inBufferSize_);

    // 7.2 申请Device内存picOutBufferDev_，用于存放VDEC解码后的输出数据
    ret = acldvppMalloc(&picOutBufferDev_, size);

    // 7.3 创建输出图片描述信息，设置图片描述信息的属性
    // picOutputDesc_是acldvppPicDesc类型
    picOutputDesc_ = acldvppCreatePicDesc();
    ret = acldvppSetPicDescData(picOutputDesc_, picOutBufferDev_);
    ret = acldvppSetPicDescSize(picOutputDesc_, size);
    ret = acldvppSetPicDescFormat(picOutputDesc_, static_cast<acldvppPixelFormat>(format_));
```



```
    // 7.4 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 = aclvdecSendFrame(vdecChannelDesc_, streamInputDesc_, picOutputDesc_, nullptr, nullptr);
    // ......
    ++count;
    rest_len = rest_len - 1;
    // ......
}

// 8.释放图片处理通道、图片描述信息
ret = aclvdecDestroyChannel(vdecChannelDesc_);
aclvdecDestroyChannelDesc(vdecChannelDesc_);

// 9. 释放运行管理资源（依次释放Stream、Context、Device）

// 10.AscendCL去初始化

// ......
```

返回码说明

表 9-1 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》。

返回码	含义	可能原因及解决方法
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》。
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**。

返回码	含义	可能原因及解决方法
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。
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。

返回码	含义	可能原因及解决方法
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。	-
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	函数参数中有空指针。	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》。

返回码	含义	可能原因及解决方法
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**。
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。	-

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

9.4.6 VENC 视频编码

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。

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。

须知

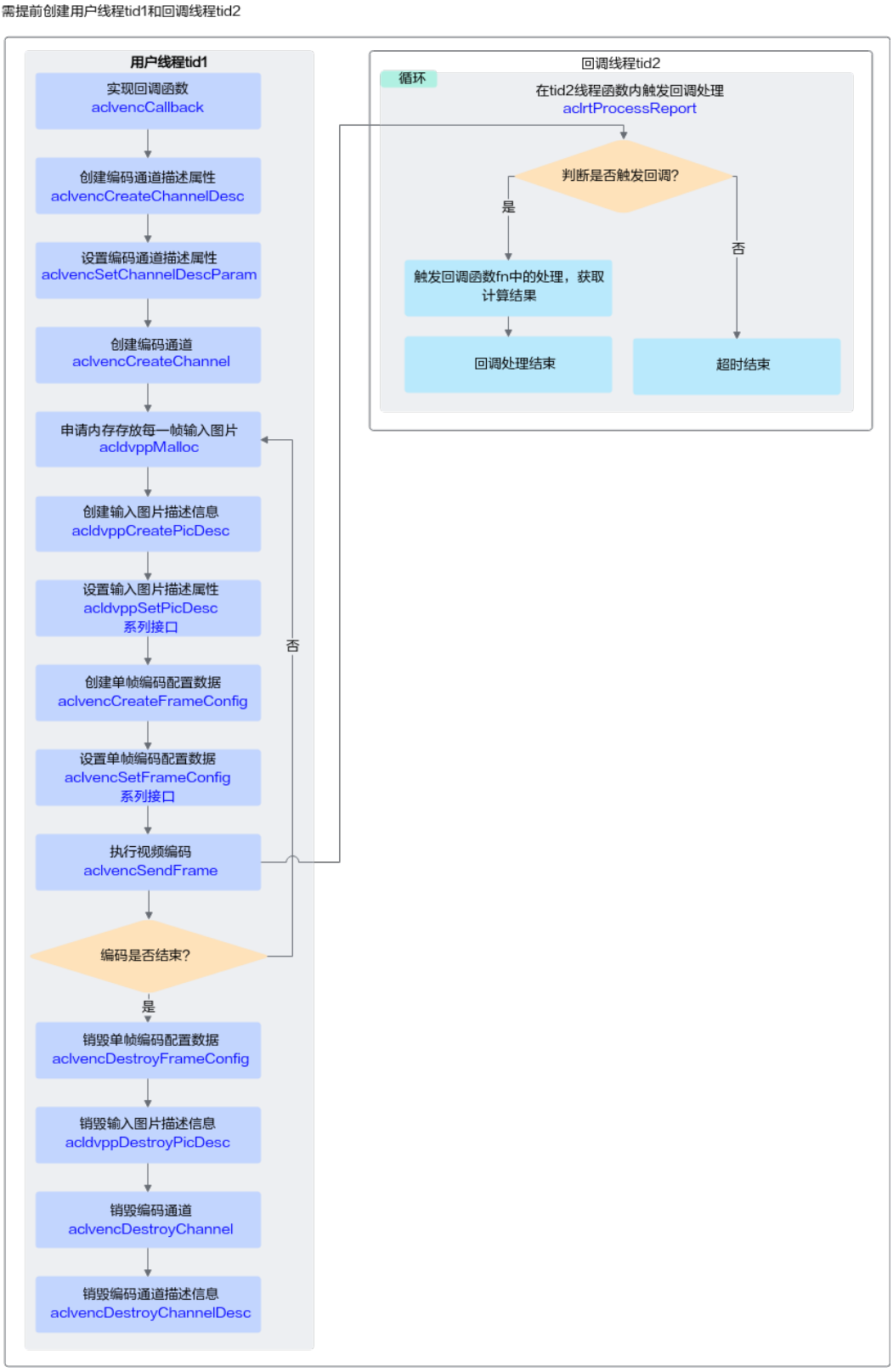
Atlas 训练系列产品上，不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-11 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aclvencCreateChannel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aclvencCreateChannelDesc接口**创建通道描述信息**。
     ii. 调用aclvencSetChannelDescParam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，回调函数的原型请参见aclvencCallback。
     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aclrtProcessReport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aclvencSetChannelDescParam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aclvencSetChannelDesc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。
   - aclvencCreateChannel接口内部封装了如下接口，无需用户单独调用：
     i. aclrtCreateStream接口：显式创建Stream，VENC内部使用。
     ii. aclrtSubscribeReport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aclvencSetChannelDescParam接口时指定的。
2. **调用aclvencSendFrame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acldvppCreatePicDesc接口创建输入图片描述信息，并调用acldvppSetPicDesc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aclvencCreateFrameConfig接口创建单帧编码配置数据，并调用aclvencSetFrameConfig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**，aclvencSendFrame接口内部封装了aclrtLaunchCallback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aclrtLaunchCallback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aclvencDestroyChannel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。
   - aclvencDestroyChannel接口内部封装了如下接口，无需用户单独调用：
     - aclrtUnSubscribeReport接口：取消线程注册（Stream上的回调函数不再由指定线程处理）。
     - aclrtDestroyStream接口：销毁Stream。
   - 销毁通道后，需调用aclvencDestroyChannelDesc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13.13.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍VENC视频编码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化

// 2.运行管理资源申请（依次申请Device、Context、Stream）

// 3.创建执行回调函数的线程及线程函数
static bool runFlag = true;
void *ThreadFunc(void *arg)
{
    // Notice: create context for this thread
    int deviceId = 0;
    aclrtContext context = nullptr;
    aclError ret = aclrtCreateContext(&context, deviceId);
    INFO_LOG("process callback thread start ");
    while (runFlag) {
        // Notice: timeout 1000ms
        (void)aclrtProcessReport(1000);
    }
    // ......
    ret = aclrtDestroyContext(context);
    return (void*)0;
}

int createThreadErr = pthread_create(&threadId_, nullptr, ThreadFunc, nullptr);

// 4.创建回调函数
void callback(acldvppPicDesc *input, acldvppStreamDesc *outputStreamDesc, void *userdata)
{
    // 获取视频编码结果数据，并写入文件
    void *outputDev = acldvppGetStreamDescData(outputStreamDesc);
    uint32_t streamDescSize = acldvppGetStreamDescSize(outputStreamDesc);
    if (!Utils::WriteToFile(g_outFileFp, outputDev, streamDescSize)) {
        ERROR_LOG("write file:%s failed.", g_outFile.c_str());
    }
    INFO_LOG("success to callback, stream size:%u", streamDescSize);
}

// 5.创建视频编码处理通道时的通道描述信息，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
vencChannelDesc_ = aclvencCreateChannelDesc();
auto ret = aclvencSetChannelDescThreadId(vencChannelDesc_, threadId_);
ret = aclvencSetChannelDescCallback(vencChannelDesc_, callback);
ret = aclvencSetChannelDescEnType(vencChannelDesc_, enType_);
ret = aclvencSetChannelDescPicFormat(vencChannelDesc_, format_);
ret = aclvencSetChannelDescPicWidth(vencChannelDesc_, 128);
ret = aclvencSetChannelDescPicHeight(vencChannelDesc_, 128);
ret = aclvencSetChannelDescKeyFrameInterval(vencChannelDesc_, 16);

// 6.创建视频码流处理的通道、单帧编码配置数据
ret = aclvencCreateChannel(vencChannelDesc_);
// vencFrameConfig_是aclvencFrameConfig类型
vencFrameConfig_ = aclvencCreateFrameConfig();

// 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
const char *fileName = "../dvpp_venc_128x128_nv12.yuv";
FILE *fp = fopen(fileName, "rb+");
fseek(fp, 0, SEEK_END);
long fileLenLong = ftell(fp);
fseek(fp, 0, SEEK_SET);
auto fileLen = static_cast<uint32_t>(fileLenLong);
uint32_t dataSize = fileLen;

// 调用aclrtGetRunMode接口获取软件栈的运行模式，如果调用aclrtGetRunMode接口获取软件栈的运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将输入图片数据传输到Device，数据传输完成后，需及时释放内存；否则直接申请并使用Device的内存
aclrtRunMode runMode;
ret = aclrtGetRunMode(&runMode);
if（runMode == ACL_HOST）{
    void *dataHost = malloc(fileLen);
    size_t readSize = fread(dataHost, 1, fileLen, fp)
    void *dataDev = nullptr;
    ret = acldvppMalloc(&dataDev, dataSize);
    ret = aclrtMemcpy(dataDev, dataSize, dataHost, fileLen, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放内存
    free(dataHost);
}
else {
    ret = acldvppMalloc(&dataDev, dataSize);
}

// 8.执行视频编码
size_t g_vencCnt = 16;
// 8.1 创建输入图片描述信息，设置输入图片数据的内存地址和内存大小
inputPicputDesc_ = acldvppCreatePicDesc();
ret = acldvppSetPicDescData(inputPicputDesc_, dataDev);
ret = acldvppSetPicDescSize(inputPicputDesc_, dataSize);
// 8.2 设置单帧编码配置数据，不是结束帧
ret = aclvencSetFrameConfigEos(vencFrameConfig_, 0);
ret = aclvencSetFrameConfigForceIFrame(vencFrameConfig_, 0)
// 8.3 创建输出码流描述信息
acldvppStreamDesc *outputStreamDesc = nullptr;
// 8.4 执行视频编码
while (g_vencCnt > 0) {
        ret = aclvencSendFrame(vencChannelDesc_, inputPicputDesc_,
            static_cast<void *>(outputStreamDesc), vencFrameConfig_, nullptr);
        g_vencCnt--;
    }
// 8.5 设置单帧编码配置数据，是结束帧
ret = aclvencSetFrameConfigEos(vencFrameConfig_, 1);
ret = aclvencSetFrameConfigForceIFrame(vencFrameConfig_, 0)
// 8.6 执行最后一帧的视频编码
ret = aclvencSendFrame(vencChannelDesc_, nullptr, nullptr, vencFrameConfig_, nullptr);

// 9.释放资源
(void)aclvencDestroyChannel(vencChannelDesc_);
(void)aclvencDestroyChannelDesc(vencChannelDesc_);
(void)acldvppDestroyPicDesc(inputPicputDesc_);
(void)aclvencDestroyFrameConfig(vencFrameConfig_);
// 释放内存和销毁线程
(void)acldvppFree(inBufferDev_);
void *res = nullptr;
int joinThreadErr = pthread_join(threadId_, &res);

// 10. 释放运行管理资源（依次释放Stream、Context、Device）

// 11.AscendCL去初始化

// ......
```

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


```
// 设置回调函数
void *func = (void *)callback;
aclvencSetChannelDescParam(vencChannelDesc_, ACL_VENC_CALLBACK_PTR, 8, &func);
// 获取回调函数
void *func1 = nullptr;
aclvencGetChannelDescParam(vencChannelDesc_, ACL_VENC_CALLBACK_PTR, 8, &len, &func1);

// 设置输入图片格式
acldvppPixelFormat format = PIXEL_FORMAT_YUV_SEMIPLANAR_420;
aclvencSetChannelDescParam(vencChannelDesc_, ACL_VENC_PIXEL_FORMAT_UINT32, 4, &format);
// 获取输入图片格式
acldvppPixelFormat format1 = PIXEL_FORMAT_YUV_SEMIPLANAR_420;
aclvencGetChannelDescParam(vencChannelDesc_, ACL_VENC_PIXEL_FORMAT_UINT32, 4, &len, &format1);

// 设置图片宽度
uint32_t width = 128;
aclvencSetChannelDescParam(vencChannelDesc_, ACL_VENC_PIC_WIDTH_UINT32, 4, &width);
// 获取图片高度
uint32_t width1 = 0;
aclvencGetChannelDescParam(vencChannelDesc_, ACL_VENC_PIC_WIDTH_UINT32, 4, &len, &width1);

// 设置图片高度
uint32_t height = 128;
aclvencSetChannelDescParam(vencChannelDesc_, ACL_VENC_PIC_HEIGHT_UINT32, 4, &height);
// 获取图片高度
uint32_t height1 = 0;
aclvencGetChannelDescParam(vencChannelDesc_, ACL_VENC_PIC_HEIGHT_UINT32, 4, &len, &height1);

// 设置编码输出缓存地址
ret = acldvppMalloc(&buf, bufSize);
aclvencSetChannelDescParam(vencChannelDesc_, ACL_VENC_BUF_ADDR_PTR, 8, &buf);
// 获取编码输出缓存地址
void *buf1 = nullptr;
aclvencGetChannelDescParam(vencChannelDesc_, ACL_VENC_BUF_ADDR_PTR, 8, &len, &buf1);
```

9.5 DVPP 图像/视频处理（媒体数据处理 V2）

9.5.1 VPC 图片处理典型功能

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。

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。

- 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。

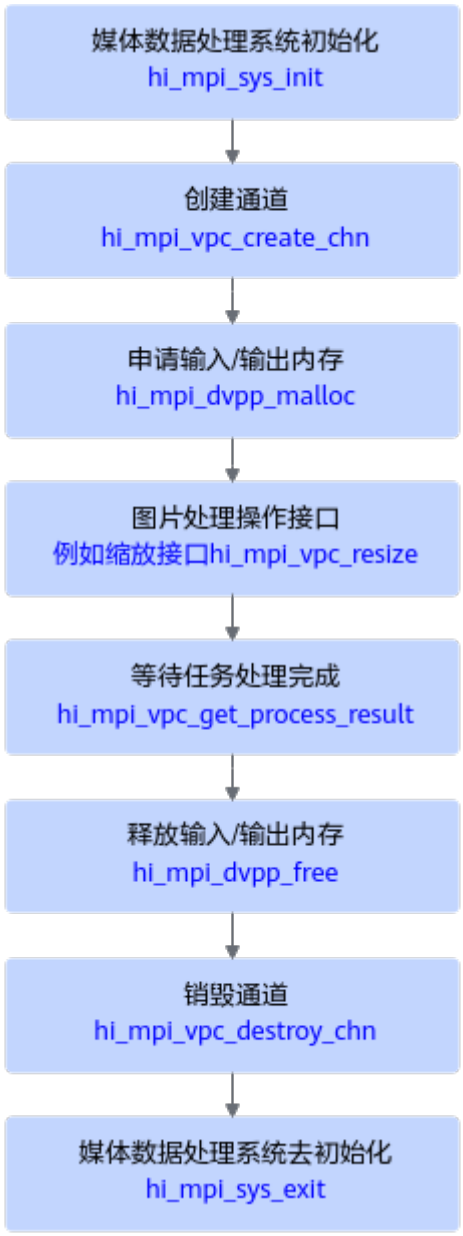
Atlas 训练系列产品上，当前版本不支持该功能。

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4.2 AscendCL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 9-12 主要接口调用流程



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hi_mpi_sys_init接口进行**媒体数据处理系统初始化**。
2. 调用hi_mpi_vpc_create_chn接口**创建通道**。

3. 调用hi_mpi_dvpp_malloc接口**申请Device上的内存**，存放输入或输出数据。

   Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。

   Atlas A2训练系列产品上，还支持使用aclrtMalloc接口申请内存。

   调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。

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hi_mpi_vpc_resize接口，对图片进行缩放。hi_mpi_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hi_mpi_vpc_get_process_result接口等待任务完成。

   - 可以跟hi_mpi_vpc_resize接口在同一个线程中调用hi_mpi_vpc_get_process_result接口，也可以新起一个线程调用hi_mpi_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aclrtGetRunMode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aclrtMemcpy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**8.3 单Batch&静态Shape输入推理**、**8.8 模型动态AIPP推理**。

5. 调用hi_mpi_dvpp_free接口**释放输入、输出内存**。

   Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

   Atlas A2训练系列产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

6. 调用hi_mpi_vpc_destroy_chn接口**销毁通道**。
7. 调用hi_mpi_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

您可以从13.16.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍图片缩放的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 3.初始化媒体数据处理系统
```



```
int32_t ret = hi_mpi_sys_init();

// 4.创建通道
hi_vpc_chn chnId;
hi_vpc_chn_attr stChnAttr;
ret = hi_mpi_vpc_sys_create_chn(&chnId, &stChnAttr);

// 5.执行缩放
// 5.1 构造存放输入图片信息的结构体
hi_vpc_pic_info inputPic;
inputPic.picture_width = 1920;
inputPic.picture_height = 1080;
inputPic.picture_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
inputPic.picture_width_stride = 1920;
inputPic.picture_height_stride = 1080;
inputPic.picture_buffer_size = inputPic.picture_width_stride * inputPic.picture_height_stride * 3 / 2;

// 5.2 准备输入图片数据
// 申请Device内存，用于媒体数据处理
ret = hi_mpi_dvpp_malloc(0, &inputPic.picture_address, inputPic.picture_buffer_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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* inputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&inputAddr, inputPic.picture_buffer_size);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputAddr, inputPic.picture_buffer_size);
    // 数据传输
    aclRet = aclrtMemcpy(inputPic.picture_address, inputPic.picture_buffer_size, inputAddr,
                         inputPic.picture_buffer_size, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放内存
    aclrtFreeHost(inputAddr);
    inputAddr = nullptr;
}
else {
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputPic.picture_address, inputPic.picture_buffer_size);
    }

// 5.3 构造存放输出图片信息的结构体
hi_vpc_pic_info outputPic;
outputPic.picture_width = 960;
outputPic.picture_height  = 540;
outputPic.picture_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
outputPic.picture_width_stride = 960;
outputPic.picture_height_stride = 540;
outputPic.picture_buffer_size = outputPic.picture_width_stride * outputPic.picture_height_stride * 3 / 2;
ret = hi_mpi_dvpp_malloc(0, &outputPic.picture_address, outputPic.picture_buffer_size);

// 初始化内存
if (runMode == ACL_HOST) {
    aclRet = aclrtMemset(outputPic.picture_address, outputPic.picture_buffer_size, 0,
outputPic.picture_buffer_size);
} else {
     memset(outputPic.picture_address, 0, outputPic.picture_buffer_size);
}

// 5.4 调用缩放接口
uint32_t taskID = 0;
ret = hi_mpi_vpc_resize(chnId, &inputPic, &outputPic, 0, 0, 0, &taskID, -1);

// 5.5 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
uint32_t taskIDResult = taskID;
ret = hi_mpi_vpc_get_process_result(chnId, taskIDResult, -1);

// 5.6 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
```



```
aclrtMemcpy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
if (g_runMode == ACL_HOST) {
    hi_vpc_pic_info outputPicHost = outputPic;
    aclRet = aclrtMallocHost(&outputPicHost.picture_address, outputPic.picture_buffer_size);
    aclRet = aclrtMemcpy(outputPicHost.picture_address, outputPic.picture_buffer_size,
outputPic.picture_address,outputPic.picture_buffer_size, ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // Host侧的数据使用完成后，释放Host内存
    aclrtFreeHost(outputPicHost.picture_address);
    outputPicHost.picture_address = nullptr;
} 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
    // TODO：推理相关的代码逻辑
}

// 5.7 释放输入、输出内存
ret = hi_mpi_dvpp_free(inputPic.picture_address);
ret = hi_mpi_dvpp_free(outputPic.picture_address);

// 6.销毁通道
ret = hi_mpi_vpc_destroy_chn(chnId);

// 7. 媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 8. 释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 9.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

抠图示例代码

您可以从13.16.1 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，不可以直接拷贝编译运行，仅供参考。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);
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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 3.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 4.创建通道
hi_vpc_chn chnId;
hi_vpc_chn_attr stChnAttr;
ret = hi_mpi_vpc_sys_create_chn(&chnId, &stChnAttr);

// 5.执行抠图
// 5.1 构造存放输入图片信息的结构体
hi_vpc_pic_info inputPic;
inputPic.picture_width = 1920;
inputPic.picture_height = 1080;
inputPic.picture_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
inputPic.picture_width_stride = 1920;
```



```
inputPic.picture_height_stride = 1080;
inputPic.picture_buffer_size = inputPic.picture_width_stride * inputPic.picture_height_stride * 3 / 2;

// 5.2 准备输入图片数据
// 申请Device内存，用于媒体数据处理
ret = hi_mpi_dvpp_malloc(0, &inputPic.picture_address, inputPic.picture_buffer_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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* inputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&inputAddr, inputPic.picture_buffer_size);
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputAddr, inputPic.picture_buffer_size);
    // 数据传输
    aclRet = aclrtMemcpy(inputPic.picture_address, inputPic.picture_buffer_size, inputAddr,
                         inputPic.picture_buffer_size, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放内存
    aclrtFreeHost(inputAddr);
    inputAddr = nullptr;
} else {
    // 将输入图片读入内存中，该自定义函数ReadPicFile由用户实现
    ReadPicFile(picName, inputPic.picture_address, inputPic.picture_buffer_size);
}

// 5.3 构造存放输出图片信息的结构体
// 该参数表示抠图数量
uint32_t multiCount = 1;
// cropRegionInfos数组的大小与抠图数量保持一致
hi_vpc_crop_region_info cropRegionInfos[1];
hi_vpc_pic_info outputPic;
for (uint32_t i = 0; i < multiCount; i++) {
    outputPic.picture_width = 960;
    outputPic.picture_height  = 540;
    outputPic.picture_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
    outputPic.picture_width_stride = 960;
    outputPic.picture_height_stride = 540;
    outputPic.picture_buffer_size = outputPic.picture_width_stride * outputPic.picture_height_stride * 3 / 2;
    ret = hi_mpi_dvpp_malloc(0, &outputPic.picture_address, outputPic.picture_buffer_size);

    // 初始化内存
    if (runMode == ACL_HOST) {
        aclRet = aclrtMemset(outputPic.picture_address, outputPic.picture_buffer_size, 0,
outputPic.picture_buffer_size);
    } else {
        memset(outputPic.picture_address, 0, outputPic.picture_buffer_size);
    }

    // 表示从输入图片中抠出以左上角为原点、分辨率960*540的子图
    cropRegionInfos[i].dest_pic_info = outputPic;
    cropRegionInfos[i].crop_region.left_offset = 0;
    cropRegionInfos[i].crop_region.top_offset = 0;
    cropRegionInfos[i].crop_region.crop_width = 960;
    cropRegionInfos[i].crop_region.crop_height = 540;
}

// 5.4 调用抠图接口
uint32_t taskID = 0;
ret = hi_mpi_vpc_crop(chnId, &inputPic, cropRegionInfos, 1, &taskID, -1);

// 5.5 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
uint32_t taskIDResult = taskID;
ret = hi_mpi_vpc_get_process_result(chnId, taskIDResult, -1);

// 5.6 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aclrtMemcpy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
```



```
if (g_runMode == ACL_HOST) {
    hi_vpc_pic_info outputPicHost = outputPic;
    aclRet = aclrtMallocHost(&outputPicHost.picture_address, outputPic.picture_buffer_size);
    aclRet = aclrtMemcpy(outputPicHost.picture_address, outputPic.picture_buffer_size,
outputPic.picture_address,outputPic.picture_buffer_size, ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // Host侧的数据使用完成后，释放Host内存
    aclrtFreeHost(outputPicHost.picture_address);
    outputPicHost.picture_address = nullptr;
} 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
    // TODO：推理相关的代码逻辑

}

// 5.7 释放输入、输出内存
ret = hi_mpi_dvpp_free(inputPic.picture_address);
inputPic.picture_address = nullptr;
for (uint32_t i = 0; i < multiCount; i++) {
     hi_mpi_dvpp_free(cropRegionInfos[i].dest_pic_info.picture_address);
     cropRegionInfos[i].dest_pic_info.picture_address = nullptr;
}

// 6.销毁通道
ret = hi_mpi_vpc_destroy_chn(chnId);

// 7. 媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 8. 释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 9.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

9.5.2 JPEGD 图片解码

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。

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。

须知

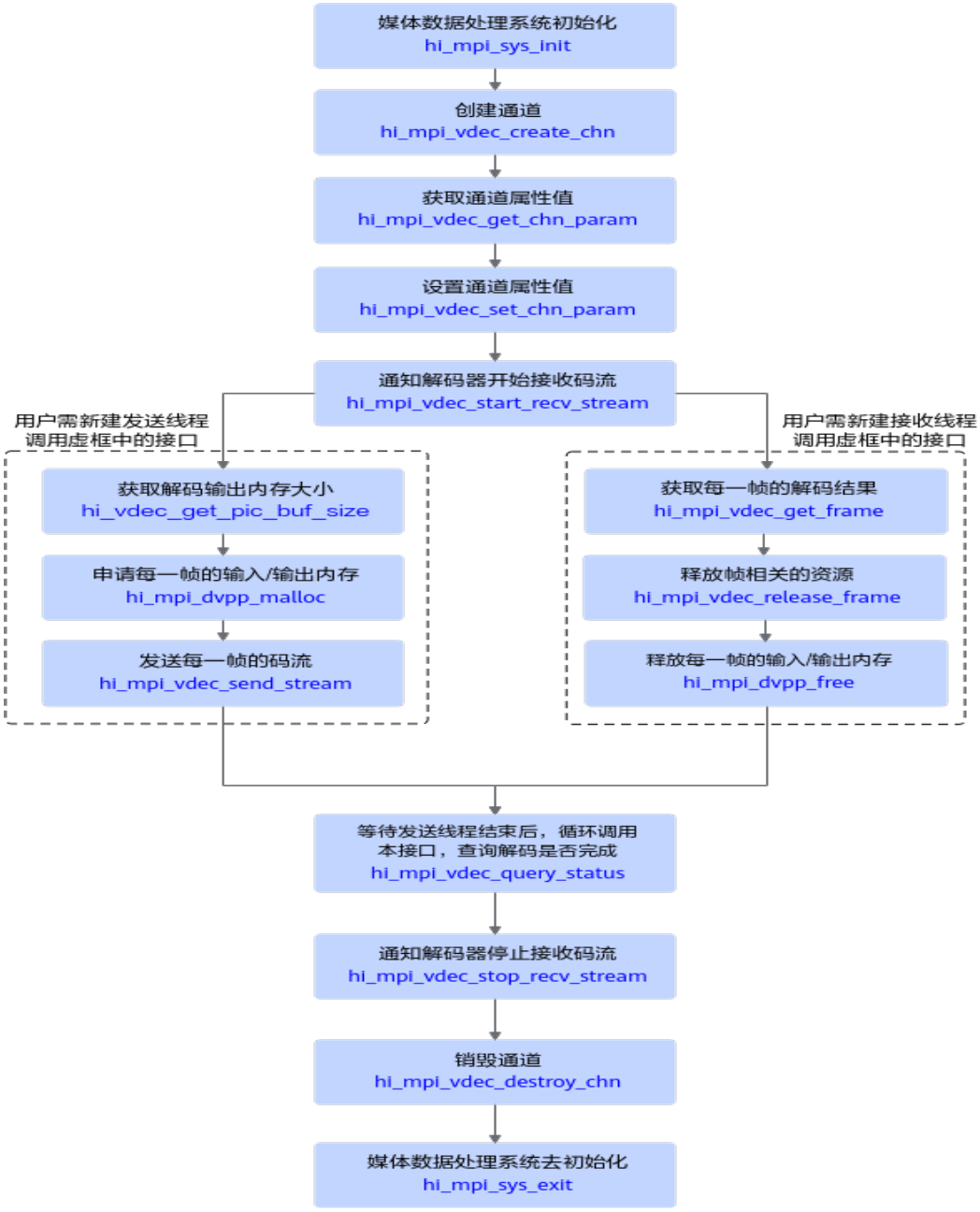
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-13 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hi_mpi_sys_init接口进行媒体数据处理系统初始化。
2. 调用hi_mpi_vdec_create_chn接口创建通道。
3. 调用hi_mpi_dvpp_malloc接口申请Device上的内存，存放输入或输出数据。

   Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。

   Atlas A2训练系列产品上，还支持使用aclrtMalloc接口申请内存。

   调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。

4. 解码前，需调用hi_mpi_vdec_start_recv_stream接口通知解码器启动接收码流，再调用hi_mpi_vdec_send_stream接口发送解码码流，hi_mpi_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hi_mpi_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hi_mpi_vdec_release_frame接口释放帧相关的资源。解码结束后，需调用hi_mpi_vdec_stop_recv_stream接口通知解码器停止接收码流。
5. 调用hi_mpi_dvpp_free接口释放输入、输出内存。

   Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

   Atlas A2训练系列产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。
6. 调用hi_mpi_vdec_destroy_chn接口销毁通道。
7. 调用hi_mpi_sys_exit接口进行媒体数据处理系统去初始化。

示例代码

您可以从13.17.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍JPEGD图片解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 3.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 4.创建通道
hi_vdec_chn chnId;
hi_vdec_chn_attr chnAttr;

chnAttr.type = HI_PT_JPEG;
chnAttr.mode = HI_VDEC_SEND_MODE_FRAME;
chnAttr.pic_width = 1920;
chnAttr.pic_height = 1080;
chnAttr.stream_buf_size = 1920 * 1080;

ret = hi_mpi_vdec_create_chn(chnId, &chnAttr);

// 5.设置通道属性
hi_vdec_chn_param chnParam;
ret = hi_mpi_vdec_get_chn_param(chnId, &chnParam);

chnParam.pic_param.pixel_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
chnParam.pic_param.alpha = 255;
chnParam.display_frame_num = 0;
ret = hi_mpi_vdec_set_chn_param(chnId, &chnParam);

// 6.解码器启动接收码流
```



```
ret = hi_mpi_vdec_start_recv_stream(chnId);

// 7.发送码流
// 7.1 申请输入内存
uint8_t* inputAddr = nullptr;
// inputsize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
int32_t inputSize = 1024;
ret = hi_mpi_dvpp_malloc(0, &inputAddr, inputSize);

// 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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* hostInputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&hostInputAddr, inputSize);
    // 将输入图片读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(fileName, hostInputAddr, inputSize);
    // 数据传输
    aclRet = aclrtMemcpy(inputAddr, inputSize, hostInputAddr, inputSize,
ACL_MEMCPY_HOST_TO_DEVICE);
} else {
    // 将输入图片读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(fileName, inputAddr, inputSize);
}

// 7.2 构造存放输入图片信息的结构体
hi_vdec_stream stStream{};
hi_img_info stImgInfo{};
stStream.pts = 0;
if (g_runMode == ACL_HOST) {
    stStream.addr  = (uint8_t *)hostInputAddr;
} else {
    stStream.addr  = (uint8_t *)inputAddr;
}
stStream.len = inputSize;
stStream.end_of_frame = HI_TRUE;
stStream.end_of_stream = HI_FALSE;
stStream.need_display  = HI_TRUE;

ret = hi_mpi_dvpp_get_image_info(HI_PT_JPEG, &stStream, &stImgInfo);
if (g_runMode == ACL_HOST) {
    // 如果不使用Host上的数据，需及时释放
   aclrtFreeHost(hostInputAddr);
    hostInputAddr = nullptr;
}
stStream.addr = (uint8_t *)inputAddr;

// 7.3 构造存放输出图片信息的结构体，并申请输出内存
hi_vdec_pic_info outPicInfo{};
void *outBuffer = nullptr;
outPicInfo.width  = stImgInfo.width;
outPicInfo.height = stImgInfo.height;
outPicInfo.width_stride  = stImgInfo.width_stride;
outPicInfo.height_stride = stImgInfo.height_stride;
outPicInfo.buffer_size   = stImgInfo.img_buf_size;
outPicInfo.pixel_format  = HI_PIXEL_FORMAT_UNKNOWN;

ret = hi_mpi_dvpp_malloc(0, &outBuffer, outPicInfo.buffer_size);

outPicInfo.vir_addr = (uint64_t)outBuffer;

// 7.4 发送需解码的输入图片
ret = hi_mpi_vdec_send_stream(chnId, &stStream, &outPicInfo, 0);

// 8.接收解码结果
// 8.1 获取解码结果
hi_video_frame_info frame;
hi_vdec_stream stream;
```



```
hi_vdec_supplement_info stSupplement;
ret = hi_mpi_vdec_get_frame(chnId, &frame, &stSupplement, &stream, 0);
if (ret == HI_SUCCESS) {
   decResult = frame.v_frame.frame_flag;
   if (decResult == 0) { // 0: Decode success
      printf("[%s][%d] Chn %u GetFrame Success, Decode Success \n",__FUNCTION__, __LINE__, chnId);
   } else { // Decode fail
      printf("[%s][%d] Chn %u GetFrame Success, Decode Fail \n",__FUNCTION__, __LINE__, chnId);
   }
}

// 8.2 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
if (g_runMode == ACL_HOST) {
    void* hostOutputAddr = nullptr;
    aclRet = aclrtMallocHost(&hostOutputAddr, outputSize);
    aclRet = aclrtMemcpy(hostOutputAddr, outputSize, frame.v_frame.virt_addr[0], outputSize,
ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // 数据使用完成后，及时释放不使用的内存
    aclrtFreeHost(hostOutputAddr);
    hostOutputAddr = nullptr;
} else {
    // 可以直接使用JPEGD的输出图片数据，在frame.v_frame.virt_addr[0]指向的内存中
    // ......
}


// 8.3 释放输入、输出内存
ret = hi_mpi_dvpp_free(frame.v_frame.virt_addr[0]);
ret = hi_mpi_dvpp_free(stream.addr);

// 8.4 释放资源
ret = hi_mpi_vdec_release_frame(chnId, &frame);

// 9.解码器停止接收码流
ret = hi_mpi_vdec_stop_recv_stream(chnId);

// 10.销毁通道
ret = hi_mpi_vdec_destroy_chn(chnId);

// 11. 媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 12. 释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 13.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

9.5.3 JPEGE 图片编码

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。

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JPEGE功能及约束说明。

须知

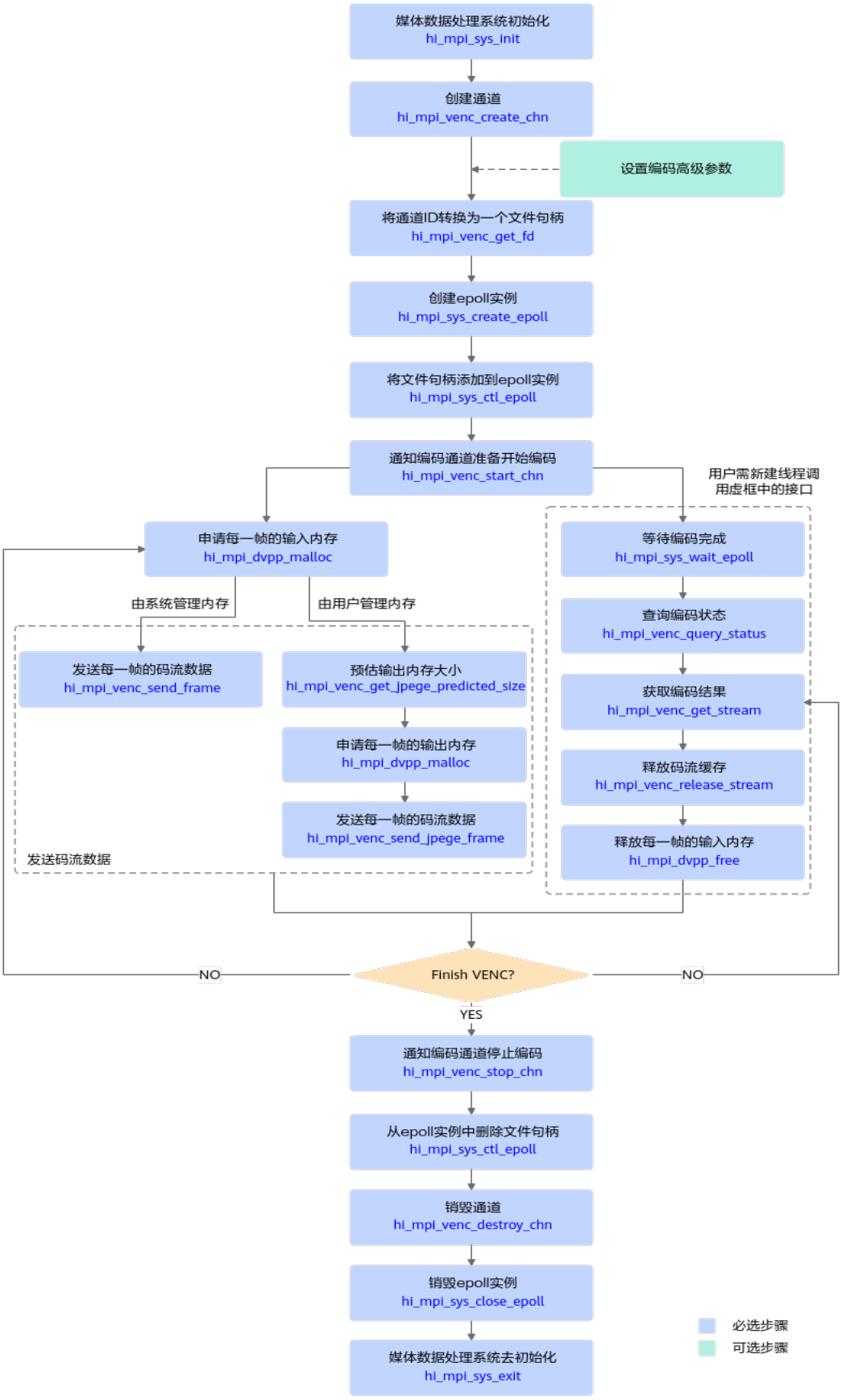
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-14 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. **资源初始化**：
   a. 调用hi_mpi_sys_init接口进行媒体数据处理系统初始化；
   b. 调用hi_mpi_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hi_mpi_venc_set_jpeg_param~hi_mpi_venc_compact_jpeg_tables章节中的接口说明。
   c. 调用hi_mpi_venc_get_fd将通道ID转换为一个文件句柄;
   d. 调用hi_mpi_sys_create_epoll函数创建DVPP epoll实例；
   e. 调用hi_mpi_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。
2. **图片编码**：
   a. 调用hi_mpi_venc_start_chn函数通知通道准备开始编码；
   b. 调用hi_mpi_dvpp_malloc接口申请存放Device上输入数据的内存。

      Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。

      Atlas A2训练系列产品上，还支持使用aclrtMalloc接口申请内存。

      调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。
   c. 启动一个用户态线程，调用hi_mpi_sys_wait_epoll函数等待编码完成；
   d. 之后用户就可以调用hi_mpi_venc_send_frame函数发送待编码的码流；
   e. 一旦编码完成，hi_mpi_sys_wait_epoll函数或select函数或poll函数就会返回，用户就可以调用hi_mpi_venc_query_status接口查询编码状态，再调用hi_mpi_venc_get_stream函数获取编码结果;
   f. 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hi_mpi_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；
   g. 调用hi_mpi_dvpp_free接口释放输入内存；

      Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

      Atlas A2训练系列产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。
   h. 当用户不需发送图像到目的通道继续编码时，需要调用hi_mpi_venc_stop_chn函数通知该通道不再接收新的输入图片；
3. **资源释放**：
   a. 调用hi_mpi_sys_ctl_epoll函数从epoll实例中删除编码通道的文件句柄；
   b. 当用户完成所有编码之后，需要调用hi_mpi_venc_destroy_chn释放编码通道以及内部内存资源；
   c. 调用hi_mpi_sys_close_epoll函数销毁DVPP epoll实例；
   d. 调用hi_mpi_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hi_mpi_venc_send_frame接口发送原始图像进行图像编码。

  在调用hi_mpi_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hi_mpi_venc_send_jpege_frame接口发送原始图像进行图像编码。

  在调用hi_mpi_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hi_mpi_venc_get_jpege_predicted_size接口预估输出内存大小，调用hi_mpi_dvpp_malloc/hi_mpi_dvpp_free接口申请/释放输出内存。

  该方式下，直接在调用hi_mpi_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

您可以从13.18.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍JPEGE图片编码的代码逻辑，AscendCL初始化和去初始化请参见5 AscendCL初始化，运行管理资源申请与释放请参见6.1 运行管理资源申请与释放。

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。

```
// 1.AscendCL初始化
aclRet = aclInit(nullptr);

// 2.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 3.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 4.创建通道
hi_venc_chn chn = 0;
hi_venc_chn_attr attr{};
attr.venc_attr.type = HI_PT_JPEG;
attr.venc_attr.profile = 0;
attr.venc_attr.max_pic_width = 128;
attr.venc_attr.max_pic_height = 128;
attr.venc_attr.pic_width = 128;
attr.venc_attr.pic_height = 128;
attr.venc_attr.buf_size = 2 * 1024 * 1024;
attr.venc_attr.is_by_frame = HI_TRUE;
attr.venc_attr.jpeg_attr.dcf_en = HI_FALSE;
attr.venc_attr.jpeg_attr.recv_mode = HI_VENC_PIC_RECV_SINGLE;
attr.venc_attr.jpeg_attr.mpf_cfg.large_thumbnail_num = 0;
ret = hi_mpi_venc_create_chn(chn, &attr);

// 5.通知编码器开始接收输入数据
```



```
hi_venc_start_param recv_param{};
recv_param.recv_pic_num = -1;
ret = hi_mpi_venc_start_chn(chn, &recv_param);

// 6.发送输入数据
// 6.1 申请输入内存
uint8_t* inputAddr = nullptr;
int32_t inputSize = 128 * 128 * 3 / 2;
ret = hi_mpi_dvpp_malloc(0, &inputAddr, inputSize);

// 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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* hostInputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&hostInputAddr, inputSize);
    // 将输入数据读入内存中，该自定义函数JpegeReadYuvFile由用户实现
    JpegeReadYuvFile(streamName, hostInputAddr, inputSize);
    // 数据传输
    aclRet = aclrtMemcpy(inputAddr, inputSize, hostInputAddr, inputSize, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放不使用的内存
    aclrtFreeHost(hostInputAddr );
    hostInputAddr = nullptr;
} else {
    // 将输入数据读入内存中，该自定义函数JpegeReadYuvFile由用户实现
    JpegeReadYuvFile(streamName, inputAddr, inputSize);
}

// 6.2 发送输入数据，开始编码
hi_video_frame_info frame{};
frame.mod_id = HI_ID_VENC;
frame.v_frame.width = 128;
frame.v_frame.height = 128;
frame.v_frame.field = HI_VIDEO_FIELD_FRAME;
frame.v_frame.pixel_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
frame.v_frame.video_format = HI_VIDEO_FORMAT_LINEAR;
frame.v_frame.compress_mode = HI_COMPRESS_MODE_NONE;
frame.v_frame.dynamic_range = HI_DYNAMIC_RANGE_SDR8;
frame.v_frame.color_gamut = HI_COLOR_GAMUT_BT709;
frame.v_frame.width_stride[0] = 128;
frame.v_frame.width_stride[1] = 128;
frame.v_frame.width_stride[2] = 128;
frame.v_frame.virt_addr[0] = inputAddr;
frame.v_frame.virt_addr[1] = (hi_void *)((uintptr_t)frame.v_frame.virt_addr[0] + 128 * 128);
frame.v_frame.frame_flag = 0;
frame.v_frame.time_ref = 0;
frame.v_frame.pts = 0;
ret = hi_mpi_venc_send_frame(chn, &frame, 0);

// 7.获取编码结果
// 7.1 创建EPOLL实例
int32_t epollFd = 0;
int32_t fd = hi_mpi_venc_get_fd(chn);
ret = hi_mpi_sys_create_epoll(10, &epollFd);

hi_dvpp_epoll_event event;
event.events = HI_DVPP_EPOLL_IN;
event.data = (void*)(unsigned long)(fd);
ret = hi_mpi_sys_ctl_epoll(epollFd, HI_DVPP_EPOLL_CTL_ADD, fd, &event);

int32_t eventCount = 0;
// 编码完成前，会超时阻塞在这里，一旦完成，才会往下执行
ret = hi_mpi_sys_wait_epoll(epollFd, event, 3, 1000, &eventCount);

// 7.2 获取编码结果
hi_venc_chn_status stat;
ret = hi_mpi_venc_query_status(chn, &stat);
hi_venc_stream stream;
```



```
stream.pack_cnt = stat.cur_packs;
stream.pack = new hi_venc_pack[stream.pack_cnt];
ret = hi_mpi_venc_get_stream(chn, &stream, 1000);

// 7.3 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
aclrtMemcpy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
if (g_runMode == ACL_HOST) {
    void* hostOutputAddr = nullptr;
    aclRet = aclrtMallocHost(&hostOutputAddr, outputSize);
    aclRet = aclrtMemcpy(hostOutputAddr, outputSize, stream.pack[0].addr, outputSize,
ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // 数据使用完成后，需及时释放不使用的内存
    aclrtFreeHost(hostOutputAddr);
    hostOutputAddr = nullptr;
} 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
    // ......
}

// 8.释放输入内存和输出码流
ret = hi_mpi_dvpp_free(inputAddr);
ret = hi_mpi_venc_release_stream(chn, &stream);
delete[] stream.pack;

// 9.通知编码器停止接收输入数据
ret = hi_mpi_venc_stop_chn(chn);
ret = hi_mpi_sys_ctl_epoll(epollFd, HI_DVPP_EPOLL_CTL_DEL, fd, NULL);
ret = hi_mpi_sys_close_epoll(epollFd);

// 10.销毁通道
ret = hi_mpi_venc_destroy_chn(chn);

// 11.媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 12.释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 13.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

9.5.4 PNGD 图片解码

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。

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PNGD图片解码功能。

须知

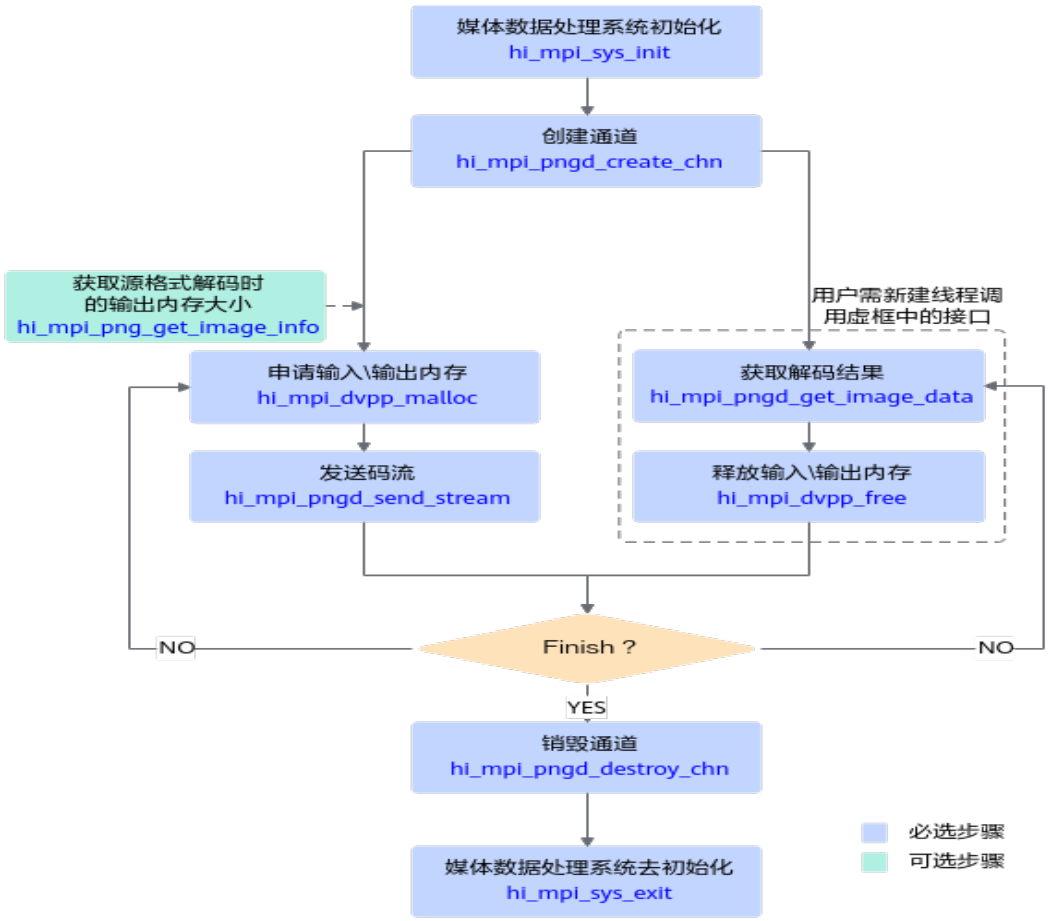
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明。**

图 9-15 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hi_mpi_sys_init接口进行媒体数据处理系统初始化。
2. 调用hi_mpi_pngd_create_chn接口创建通道。
3. 调用hi_mpi_dvpp_malloc接口申请Device上的内存，存放输入或输出数据。

   Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。

   Atlas A2训练系列产品上，还支持使用aclrtMalloc接口申请内存。

   调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。
4. 调用hi_mpi_pngd_send_stream接口发送解码码流，hi_mpi_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hi_mpi_pngd_get_image_data接口获取解码结果数据。
5. 调用hi_mpi_dvpp_free接口释放输入、输出内存。

Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

Atlas A2训练系列产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

6. 调用hi_mpi_pngd_destroy_chn接口销毁通道。
7. 调用hi_mpi_sys_exit接口进行媒体数据处理系统去初始化。

示例代码

您可以从13.21.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍PNGD图片解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 2.AscendCL初始化
aclRet = aclInit(nullptr);

// 3.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);

// 4.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 5.创建通道
hi_pngd_chn chnId;
hi_pngd_chn_attr chnAttr; // hi_pngd_chn_attr都是保留参数,无需设置
ret = hi_mpi_pngd_create_chn(chnId, &chnAttr);

// 6.发送码流
// 6.1 申请输入内存
uint8_t* inputAddr = nullptr;
// inputsize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
int32_t inputSize = 1024;
ret = hi_mpi_dvpp_malloc(0, &inputAddr, inputSize);

// 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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* hostInputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&hostInputAddr, inputSize);
    // 将输入图片读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(fileName, hostInputAddr, inputSize);
    // 数据传输
    aclRet = aclrtMemcpy(inputAddr, inputSize, hostInputAddr, inputSize,
ACL_MEMCPY_HOST_TO_DEVICE);
} else {
    // 将输入图片读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(fileName, inputAddr, inputSize);
}

// 6.2 构造存放输入图片信息的结构体
hi_img_stream stStream{};
```



```
hi_img_info stImgInfo{};
stStream.pts = 0;
if (g_runMode == ACL_HOST) {
    stStream.addr  = (uint8_t *)hostInputAddr;
} else {
    stStream.addr  = (uint8_t *)inputAddr;
}
stStream.len  = inputSize;
stStream.type = HI_PT_PNG;

ret = hi_mpi_png_get_image_info(&stStream, &stImgInfo);
if (g_runMode == ACL_HOST) {
    // 如果不使用Host上的数据，需及时释放
   aclrtFreeHost(hostInputAddr);
    hostInputAddr = nullptr;
}
stStream.addr = (uint8_t *)inputAddr;

// 6.3 构造存放输出图片信息的结构体，并申请输出内存
hi_pic_info outPicInfo{};
void *outBuffer = nullptr;
outPicInfo.picture_width  = stImgInfo.width;
outPicInfo.picture_height = stImgInfo.height;
outPicInfo.picture_width_stride  = stImgInfo.width_stride;
outPicInfo.picture_height_stride = stImgInfo.height_stride;
outPicInfo.picture_buffer_size   = stImgInfo.img_buf_size;
outPicInfo.picture_format = HI_PIXEL_FORMAT_UNKNOWN;

ret = hi_mpi_dvpp_malloc(0, &outBuffer, outPicInfo.buffer_size);

outPicInfo.picture_address = (uint64_t)outBuffer;

// 6.4 发送需解码的输入图片
ret = hi_mpi_pngd_send_stream(chnId, &stream, &outPicInfo, 0);

// 7.接收解码结果
// 7.1 获取解码结果
hi_pic_info picInfo;
hi_img_stream stream;
ret = hi_mpi_pngd_get_image_data(chnId, &picInfo, &stream, 0);
if (ret == HI_SUCCESS) { // Decode success
    printf("[%s][%d] Chn %u GetFrame Success, Decode Success \n",__FUNCTION__, __LINE__, chnId);
} else if (ret == HI_ERR_PNGD_BUF_EMPTY){ // Decoding
    printf("[%s][%d] Chn %u Decoding, try again \n",__FUNCTION__, __LINE__, chnId);
} else { // Decode fail
    printf("[%s][%d] Chn %u GetFrame Success, Decode Fail \n",__FUNCTION__, __LINE__, chnId);
}

// 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
aclrtMemcpy接口将Device的输出图片数据传输到Host
if (g_runMode == ACL_HOST) {
    void* hostOutputAddr = nullptr;
    aclRet = aclrtMallocHost(&hostOutputAddr, outputSize);
    aclRet = aclrtMemcpy(hostOutputAddr, outputSize, frame.v_frame.virt_addr[0], outputSize,
ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // 数据使用完成后，及时释放不使用的内存
    aclrtFreeHost(hostOutputAddr);
    hostOutputAddr = nullptr;
} 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、输出内存
ret = hi_mpi_dvpp_free(frame.v_frame.virt_addr[0]);
ret = hi_mpi_dvpp_free(stream.addr);

// 8.销毁通道
```



```
ret = hi_mpi_pngd_destroy_chn(chnId);

// 9. 媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 10. 释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 11.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

9.5.5 VDEC 视频解码

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。

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。

须知

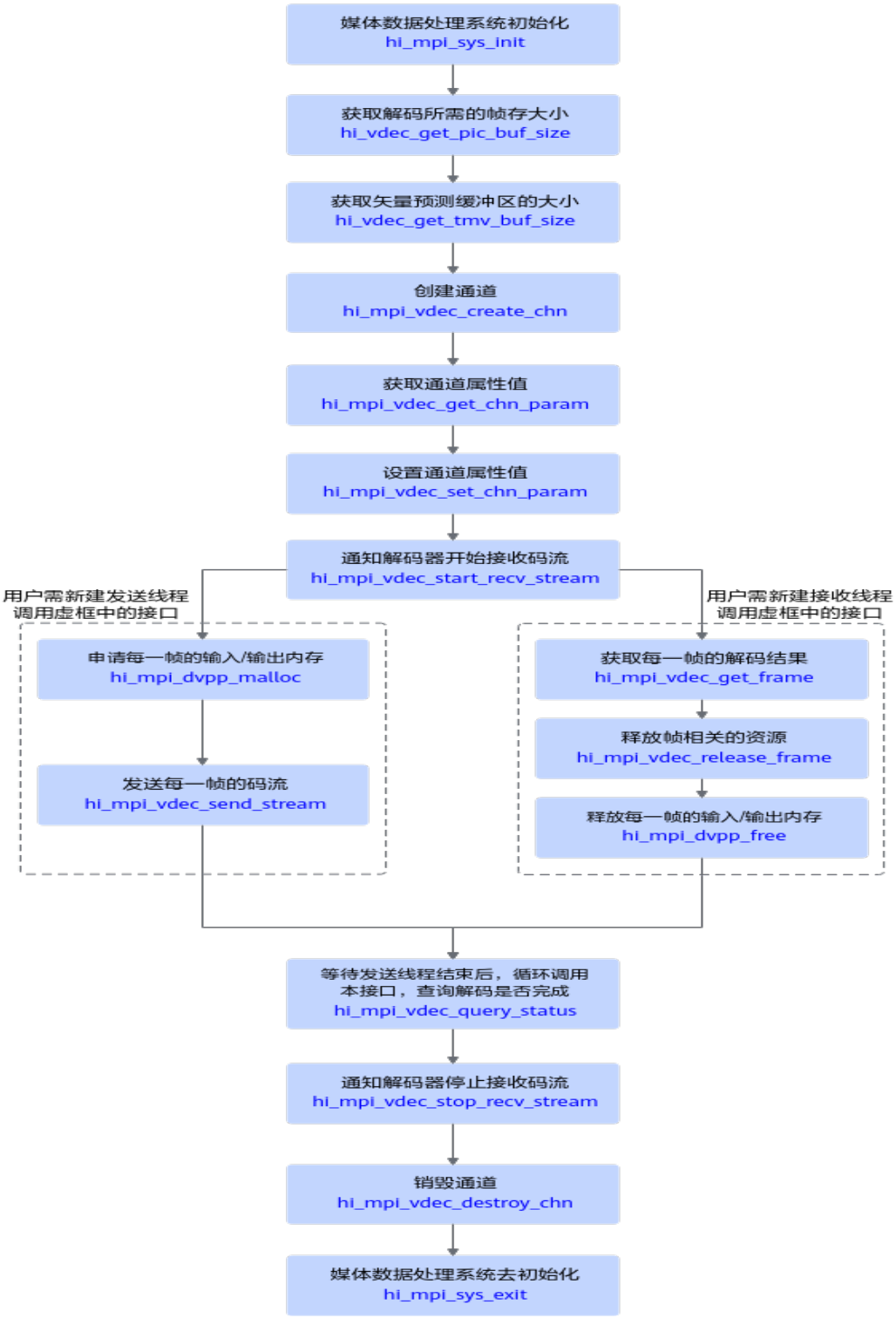
Atlas 200/300/500 推理产品上，当前版本不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-16 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hi_mpi_sys_init接口进行媒体数据处理系统初始化。
2. 调用hi_vdec_get_pic_buf_size接口获取解码所需的帧存大小、调用hi_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hi_mpi_vdec_create_chn接口创建通道。
4. 调用hi_mpi_dvpp_malloc接口申请Device上的内存，存放输入或输出数据。

   Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。

   Atlas A2训练系列产品上，还支持使用aclrtMalloc接口申请内存。

   调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。
5. 解码前，需调用hi_mpi_vdec_start_recv_stream接口通知解码器启动接收码流，再调用hi_mpi_vdec_send_stream接口发送解码码流，hi_mpi_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hi_mpi_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hi_mpi_vdec_release_frame接口释放帧相关的资源。解码结束后，需调用hi_mpi_vdec_stop_recv_stream接口通知解码器停止接收码流。
6. 调用hi_mpi_dvpp_free接口释放输入、输出内存。

   Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。

   Atlas A2训练系列产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。
7. 调用hi_mpi_vdec_destroy_chn接口销毁通道。
8. 调用hi_mpi_sys_exit接口进行媒体数据处理系统去初始化。

示例代码

您可以从13.19.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍VDEC视频解码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 2.AscendCL初始化
aclRet = aclInit(nullptr);

// 3.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);

// 4.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 5.创建通道
hi_vdec_chn chnId;
hi_vdec_chn_attr chnAttr;
hi_pic_buf_attr buf_attr{1920, 1080, 0, 8, HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420,
HI_COMPRESS_MODE_NONE};

chnAttr.type = HI_PT_H264;
chnAttr.mode = HI_VDEC_SEND_MODE_FRAME;
chnAttr.pic_width = 1920;
chnAttr.pic_height = 1080;
```



```
chnAttr.stream_buf_size = 1920 * 1080 * 3 / 2;
chnAttr.frame_buf_cnt = 16;
chnAttr.frame_buf_size = hi_vdec_get_pic_buf_size(HI_PT_H264, &buf_attr);
chnAttr.video_attr.ref_frame_num = 12;
chnAttr.video_attr.temporal_mvp_en = HI_TRUE;
chnAttr.video_attr.tmv_buf_size = hi_vdec_get_tmv_buf_size(HI_PT_H264, 1920, 1080);

ret = hi_mpi_vdec_create_chn(chnId, &chnAttr);

// 6.设置通道属性
hi_vdec_chn_param chnParam;
ret = hi_mpi_vdec_get_chn_param(chnId, &chnParam);

chnParam.video_param.dec_mode = HI_VIDEO_DEC_MODE_IPB;
chnParam.video_param.compress_mode = HI_COMPRESS_MODE_HFBC;
chnParam.video_param.video_format = HI_VIDEO_FORMAT_TILE_64x16;
chnParam.display_frame_num = 3;
chnParam.video_param.out_order = HI_VIDEO_OUT_ORDER_DISPLAY;

ret = hi_mpi_vdec_set_chn_param(chnId, &chnParam);

// 7.解码器启动接收码流
ret = hi_mpi_vdec_start_recv_stream(chnId);

// 8.发送码流
// 8.1 申请输入内存
uint8_t* inputAddr = nullptr;
// inputsize表示每一帧码流占用的内存大小，此处以1024 Byte为例，用户需根据实际情况计算内存大小
int32_t inputSize = 1024;
ret = hi_mpi_dvpp_malloc(0, &inputAddr, inputSize);

// 如果运行模式为ACL_HOST，则需要申请Host内存，将输入码流数据读入Host内存，再通过aclrtMemcpy接口将Host的码流数据传输到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
if (runMode == ACL_HOST) {
    void* hostInputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&hostInputAddr, inputSize);
    // 将输入码流读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(streamName, hostInputAddr, inputSize);
    // 数据传输
    aclRet = aclrtMemcpy(inputAddr, inputSize, hostInputAddr, inputSize, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放不使用的内存
    aclrtFreeHost(hostInputAddr );
    hostInputAddr = nullptr;
} else {
   // 将输入码流读入内存中，该自定义函数ReadStreamFile由用户实现
    ReadStreamFile(streamName, inputAddr, inputSize);
}

// 8.2 申请输出内存
uint8_t* outputAddr = nullptr;
// 输出图片数据占用的内存大小与输出图片格式有关
int32_t outputSize = 1920 * 1080 * 3 / 2;
ret = hi_mpi_dvpp_malloc(0, &outputAddr, outputSize);

// 8.3 构造存放一帧输入码流信息的结构体
hi_vdec_stream stream;
stream.addr = inputAddr;
stream.len = inputSize;
stream.end_of_frame = HI_TRUE;
stream.end_of_stream = HI_FALSE;
stream.need_display = HI_TRUE;

// 8.4 构造存放一帧输出结果信息的结构体
hi_vdec_pic_info outPicInfo;
outPicInfo.width = 1920;
outPicInfo.height = 1080;
outPicInfo.width_stride = 1920;
```



```
outPicInfo.height_stride = 1080;
outPicInfo.pixel_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
outPicInfo.vir_addr = outputAddr;
outPicInfo.buffer_size = outputSize;

// 8.5 发送一帧码流
ret = hi_mpi_vdec_send_stream(chnId, &stream, &outPicInfo, 0);

// 9.接收解码结果
// 9.1 获取解码结果
hi_video_frame_info frame;
hi_vdec_stream stream;
hi_vdec_supplement_info stSupplement;
ret = hi_mpi_vdec_get_frame(chnId, &frame, &stSupplement, &stream, 0);
if (ret == HI_SUCCESS) {
  decResult = frame.v_frame.frame_flag;
  if (decResult == 0) { // 0: Decode success
      printf("[%s][%d] Chn %u GetFrame Success, Decode Success \n",__FUNCTION__, __LINE__, chnId);
  } elseif (decResult == 1) { // 1: Decode fail
      printf("[%s][%d] Chn %u GetFrame Success, Decode Fail \n",__FUNCTION__, __LINE__, chnId);
  } elseif (decResult == 2) { // 2:This result is returned for the second field of
      printf("[%s][%d] Chn %u GetFrame Success, No Picture \n", __FUNCTION__, __LINE__, chnId);
  } elseif (decResult == 3) { // 3: Reference frame number set error
      printf("[%s][%d] Chn %u GetFrame Success, RefFrame Num Error \n",__FUNCTION__, __LINE__, chnId);
  } elseif (decResult == 4) { // 4: Reference frame size set error
      printf("[%s][%d] Chn %u GetFrame Success, RefFrame Size Error \n",__FUNCTION__, __LINE__, chnId);
 }
}

// 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
aclrtMemcpy接口将Device的输出图片数据传输到Host
if (g_runMode == ACL_HOST) {
    void* hostOutputAddr = nullptr;
    aclRet = aclrtMallocHost(&hostOutputAddr, outputSize);
    aclRet = aclrtMemcpy(hostOutputAddr, outputSize, frame.v_frame.virt_addr[0], outputSize,
ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // 数据使用完成后，需及时释放不需要使用的内存
    aclrtFreeHost(hostOutputAddr);
    hostOutputAddr = nullptr;
} else {
    // 可以直接使用VDEC的输出图片数据，在frame.v_frame.virt_addr[0]指向的内存中
    // TODO：推理相关的代码逻辑
}

// 9.3 释放输入、输出内存
ret = hi_mpi_dvpp_free(frame.v_frame.virt_addr[0]);
ret = hi_mpi_dvpp_free(stream.addr);

// 9.4 释放资源
ret = hi_mpi_vdec_release_frame(chnId, &frame);

// 10.解码器停止接收码流
ret = hi_mpi_vdec_stop_recv_stream(chnId);

// 11.销毁通道
ret = hi_mpi_vdec_destroy_chn(chnId);

// 12. 媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 13. 释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
aclRet = aclrtResetDevice(0);

// 14.AscendCL去初始化
aclRet = aclFinalize();

// ....
```


9.5.6 VENC 视频编码

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。

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VENC功能及约束说明。

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，请参见**优化视频编码质量**。

须知

Atlas 200/300/500 推理产品上，当前版本不支持该功能。

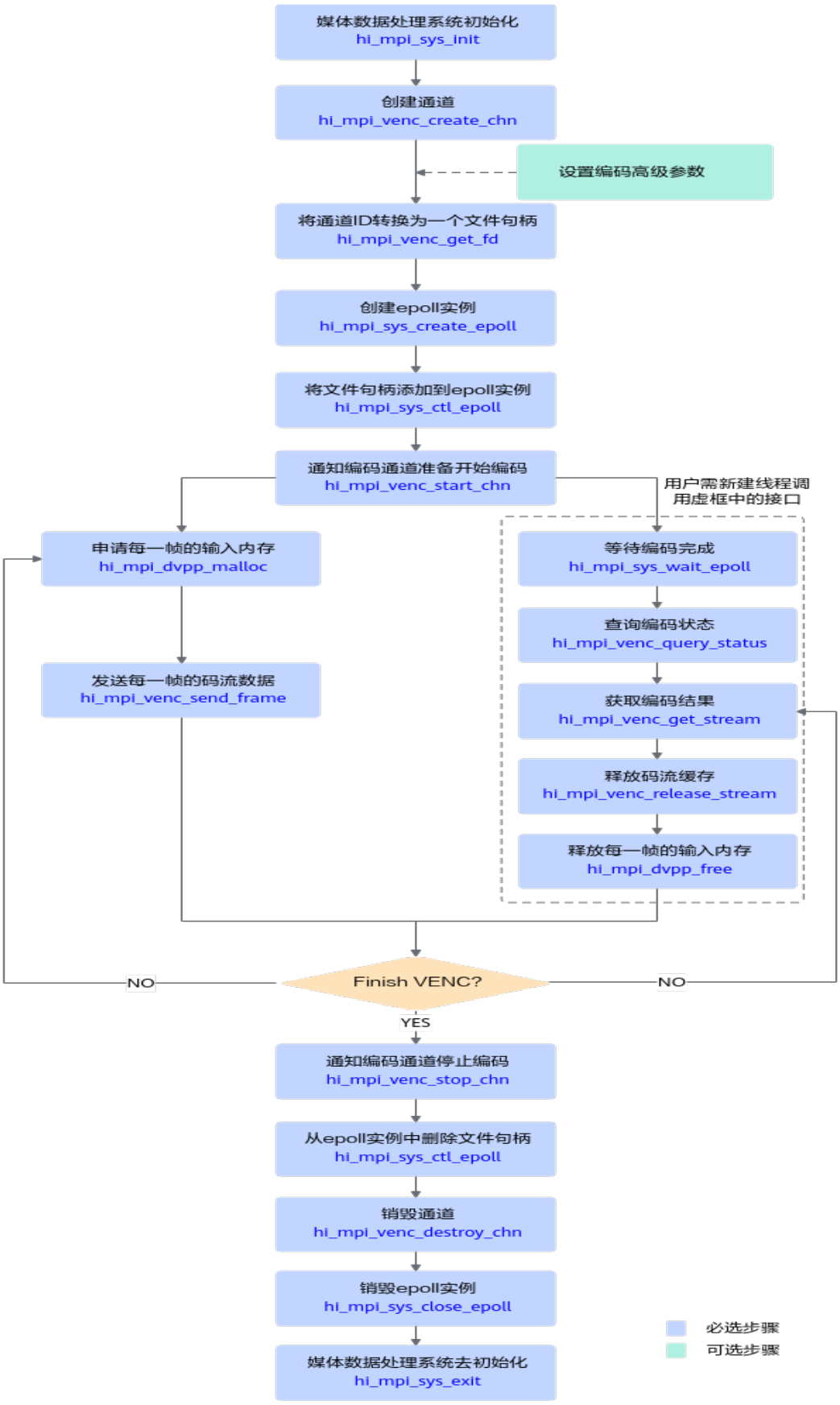
Atlas 训练系列产品上，当前版本不支持该功能。

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4.2 AscendCL接口调用流程了解整体流程，再查看本节中的流程说明**。

图 9-17 接口调用流程



当前系统支持H264/H265格式的视频码流，关键接口的说明如下：

1. **资源初始化**：
   a. 用hi_mpi_sys_init接口进行媒体数据处理系统初始化；
   b. 调用hi_mpi_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hi_mpi_venc_set_jpeg_param~hi_mpi_venc_compact_jpeg_tables章节中的接口说明。
   c. 调用hi_mpi_venc_get_fd将通道ID转换为一个文件句柄;
   d. 调用hi_mpi_sys_create_epoll函数创建DVPP epoll实例；
   e. 调用hi_mpi_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。
2. **视频编码**：
   a. 调用hi_mpi_venc_start_chn函数通知通道准备开始编码；
   b. 调用hi_mpi_dvpp_malloc接口申请存放Device上输入数据的内存。
      Atlas 200/500 A2推理产品上，还支持使用aclrtMalloc接口申请内存。
      调用hi_mpi_dvpp_malloc接口申请的内存为媒体数据处理的专用内存，但专用内存的地址空间有限，若关注内存规划或内存资源有限时，建议调用aclrtMalloc接口申请内存。
   c. 启动一个用户态线程，调用hi_mpi_sys_wait_epoll函数等待编码完成；
   d. 之后用户就可以调用hi_mpi_venc_send_frame函数发送待编码的码流；
   e. 一旦编码完成，hi_mpi_sys_wait_epoll函数或select函数或poll函数就会返回，用户就可以调用hi_mpi_venc_query_status接口查询编码状态，再调用hi_mpi_venc_get_stream函数获取编码结果;
   f. 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hi_mpi_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；
   g. 调用hi_mpi_dvpp_free接口释放输入内存；
      Atlas 200/500 A2推理产品上，若使用aclrtMalloc接口申请内存，则需使用aclrtFree接口释放内存。
   h. 当用户不需发送图像到目的通道继续编码时，需要调用hi_mpi_venc_stop_chn函数通知该通道不再接收新的输入图片；
3. **资源释放**：
   a. 调用hi_mpi_sys_ctl_epoll函数从epoll实例中删除编码通道的文件句柄；
   b. 当用户完成所有编码之后，需要调用hi_mpi_venc_destroy_chn释放编码通道以及内部内存资源；
   c. 调用hi_mpi_sys_close_epoll函数销毁DVPP epoll实例；
   d. 调用hi_mpi_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hi_mpi_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hi_venc_chn_attr.venc_attr结构内的profile参数来设置；
  - 帧率，通过hi_venc_chn_attr.rc_attr结构体内的src_frame_rate输入帧率参数、dst_frame_rate输出帧率参数来设置；
  - GOP，通过hi_venc_chn_attr.rc_attr结构体内的gop参数来设置；
  - 码率，通过hi_venc_chn_attr.rc_attr结构体内的bit_rate或max_bit_rate或target_bit_rate参数来设置。

  表 9-2 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	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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。

画质/分辨率	帧率	GOP	码率（Mbps）
1080P（蓝光） 1920*1080	25或30	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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。
720P（高清） 1280*720	25或30	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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。

画质/分辨率	帧率	GOP	码率（Mbps）
270P（流畅） 480*270	25或30	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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	建议GOP为帧率的整数倍，例如帧率为25时，GOP建议25或50。	• 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 9-3 高级配置项列表

配置项	接口	参数名	说明
码控模式	hi_mpi_venc_create_chn	hi_venc_chn_attr.rc_attr结构体内的rc_mode参数	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；

配置项	接口	参数名	说明
码率控制模型统计时间	hi_mpi_venc_create_chn	hi_venc_chn_attr.rc_attr内各模式属性值结构体内的stats_time参数	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	hi_mpi_venc_set_rc_param	hi_venc_rc_param结构内的threshold_i、threshold_p、threshold_b、direction、row_qp_delta参数	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。
第一帧的起始Qp值	hi_mpi_venc_create_chn	hi_venc_rc_param结构内的first_frame_start_qp参数	典型场景下，用户配置的码率小于**表9-2**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参数，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参数配置为35，同时将max_reencode_times参数配置为0，会获得较好的编码质量。
编码场景模式	hi_mpi_venc_set_scene_mode	hi_venc_scene_mode	安防场景配置为HI_VENC_SCENE_0；智能驾驶、直播、游戏、动画、电影配置为HI_VENC_SCENE_1。

示例代码

您可以从13.20.1 样例介绍中获取完整样例代码。

本节中的示例重点介绍VENC视频编码的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

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。

```
// 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）
aclrtRunMode runMode;
aclError aclRet = aclrtGetRunMode(&runMode);

// 2.AscendCL初始化
```



```
aclRet = aclInit(nullptr);

// 3.运行管理资源申请（依次申请Device、Context）
aclrtContext g_context;
aclRet = aclrtSetDevice(0);
aclRet = aclrtCreateContext(&g_context, 0);

// 4.初始化媒体数据处理系统
int32_t ret = hi_mpi_sys_init();

// 5.创建通道
hi_venc_chn chn = 0;
hi_venc_chn_attr attr{};
attr.venc_attr.type = HI_PT_H265;
attr.venc_attr.profile = 0;
attr.venc_attr.max_pic_width = 128;
attr.venc_attr.max_pic_height = 128;
attr.venc_attr.pic_width = 128;
attr.venc_attr.pic_height = 128;
attr.venc_attr.buf_size = 2 * 1024 * 1024;
attr.venc_attr.is_by_frame = HI_TRUE;
attr.rc_attr.rc_mode = HI_VENC_RC_MODE_H265_VBR;
attr.rc_attr.h265_vbr.gop = 30;
attr.rc_attr.h265_vbr.stats_time = 1;
attr.rc_attr.h265_vbr.src_frame_rate = 30;
attr.rc_attr.h265_vbr.dst_frame_rate = 30;
attr.rc_attr.h265_vbr.max_bit_rate = 4000;
attr.gop_attr.gop_mode = HI_VENC_GOP_MODE_NORMAL_P;
attr.gop_attr.normal_p.ip_qp_delta = 3;
ret = hi_mpi_venc_create_chn(chn, &attr);

// 6.通知编码器启动接收输入数据
hi_venc_start_param recv_param{};
recv_param.recv_pic_num = -1;
ret = hi_mpi_venc_start_chn(chn, &recv_param);

// 7.发送输入数据
// 7.1 申请输入内存
uint8_t* inputAddr = nullptr;
int32_t inputSize = 128 * 128 * 3 / 2;
ret = hi_mpi_dvpp_malloc(0, &inputAddr, inputSize);

// 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aclrtMemcpy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
if (runMode == ACL_HOST) {
    void* hostInputAddr = nullptr;
    // 申请Host内存
    aclRet = aclrtMallocHost(&hostInputAddr, inputSize);
    // 将输入数据读入内存中，该自定义函数VencReadYuvFile由用户实现
    VencReadYuvFile(streamName, hostInputAddr, inputSize);
    // 数据传输
    aclRet = aclrtMemcpy(inputAddr, inputSize, hostInputAddr, inputSize, ACL_MEMCPY_HOST_TO_DEVICE);
    // 完成数据传输后，需及时释放不使用的内存
    aclrtFreeHost(hostInputAddr );
    hostInputAddr = nullptr;
} else {
    // 将输入数据读入内存中，该自定义函数VencReadYuvFile由用户实现
    VencReadYuvFile(streamName, inputAddr, inputSize);
}

// 7.2 发送输入数据，开始编码
hi_video_frame_info frame{};
frame.mod_id = HI_ID_VENC;
frame.v_frame.width = 128;
frame.v_frame.height = 128;
frame.v_frame.field = HI_VIDEO_FIELD_FRAME;
frame.v_frame.pixel_format = HI_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
frame.v_frame.video_format = HI_VIDEO_FORMAT_LINEAR;
frame.v_frame.compress_mode = HI_COMPRESS_MODE_NONE;
```



```
frame.v_frame.dynamic_range = HI_DYNAMIC_RANGE_SDR8;
frame.v_frame.color_gamut = HI_COLOR_GAMUT_BT709;
frame.v_frame.width_stride[0] = 128;
frame.v_frame.width_stride[1] = 128;
frame.v_frame.width_stride[2] = 128;
frame.v_frame.virt_addr[0] = inputAddr;
frame.v_frame.virt_addr[1] = (hi_void *)((uintptr_t)frame.v_frame.virt_addr[0] + 128 * 128);
frame.v_frame.frame_flag = 0;
frame.v_frame.time_ref = 0;
frame.v_frame.pts = 0;
ret = hi_mpi_venc_send_frame(chn, &frame, 0);

// 8.获取编码结果
// 8.1 创建EPOLL实例
int32_t epollFd = 0;
int32_t fd = hi_mpi_venc_get_fd(chn);
ret = hi_mpi_sys_create_epoll(10, &epollFd);

hi_dvpp_epoll_event event;
event.events = HI_DVPP_EPOLL_IN;
event.data = (void*)(unsigned long)(fd);
ret = hi_mpi_sys_ctl_epoll(epollFd, HI_DVPP_EPOLL_CTL_ADD, fd, &event);

int32_t eventCount = 0;
// 编码完成前，会超时阻塞在这里，一旦完成，才会往下执行
ret = hi_mpi_sys_wait_epoll(epollFd, event, 3, 1000, &eventCount);

// 8.2 获取编码结果
hi_venc_chn_status stat;
ret = hi_mpi_venc_query_status(chn, &stat);
hi_venc_stream stream;
stream.pack_cnt = stat.cur_packs;
stream.pack = new hi_venc_pack[stream.pack_cnt];
ret = hi_mpi_venc_get_stream(chn, &stream, 1000);

// 8.3 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
aclrtMemcpy接口将Device的输出码流传输到Host
if (g_runMode == ACL_HOST) {
    void* hostOutputAddr = nullptr;
    aclRet = aclrtMallocHost(&hostOutputAddr, outputSize);
    aclRet = aclrtMemcpy(hostOutputAddr, outputSize, stream.pack[0].addr, outputSize,
ACL_MEMCPY_DEVICE_TO_HOST);
    // ......
    // 数据使用完成后，需及时释放不使用的内存
    aclrtFreeHost(hostOutputAddr);
    hostOutputAddr = nullptr;
} 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
    // TODO: 推理相关的代码逻辑
}

// 9.释放输入内存和输出码流
ret = hi_mpi_dvpp_free(inputAddr);
ret = hi_mpi_venc_release_stream(chn, &stream);
delete[] stream.pack;

// 10.通知编码器停止接收输入数据
ret = hi_mpi_venc_stop_chn(chn);
ret = hi_mpi_sys_ctl_epoll(epollFd, HI_DVPP_EPOLL_CTL_DEL, fd, NULL);
ret = hi_mpi_sys_close_epoll(epollFd);

// 11.销毁通道
ret = hi_mpi_venc_destroy_chn(chn);

// 12.媒体数据处理系统去初始化
ret = hi_mpi_sys_exit();

// 13.释放运行管理资源（依次释放Context、Device）
aclRet = aclrtDestroyContext(g_context);
```



```
aclRet = aclrtResetDevice(0);

// 14.AscendCL去初始化
aclRet = aclFinalize();

// ....
```

9.6 Camera 场景视频数据获取和处理

9.6.1 视频数据获取功能

视频获取功能需要ISP、MIPI Rx、VI等多个功能模块配合才能实现，本节介绍视频数据获取功能的总体接口调用流程、各功能模块的接口调用流程及注意事项。

当前需通过以下功能模块的配合实现视频数据获取功能：

- **ISP系统控制**

  系统控制部分用于注册3A算法、注册Sensor驱动、初始化ISP firmware、运行ISP firmware、退出ISP firmware、配置ISP属性等功能。

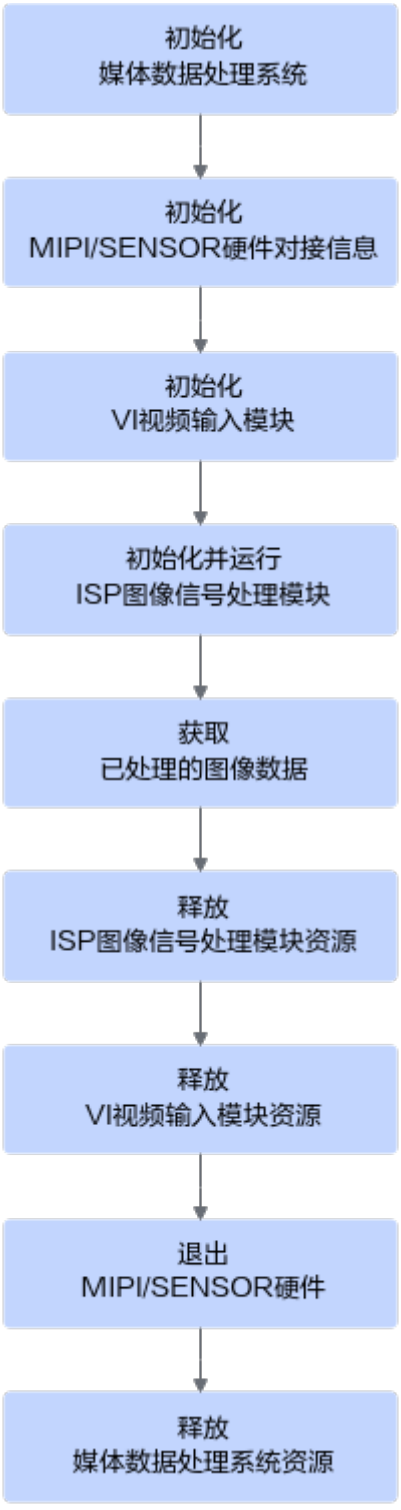
- **MIPI Rx ioctl命令字**

  MIPI Rx是一个支持多种差分视频输入接口的采集单元，通过combo-PHY接收MIPI/LVDS/sub-LVDS/HiSPi接口的数据，通过不同的功能模式配置，MIPI Rx可以支持多种速度和分辨率的数据传输需求，支持多种外部输入设备。

- **VI（Vedio Input）**

  VI模块捕获视频图像，可对其做裁剪、防抖、颜色优化、亮度优化、噪声去除等处理，并输出YUV或RAW格式的图像数据。

总体接口调用流程

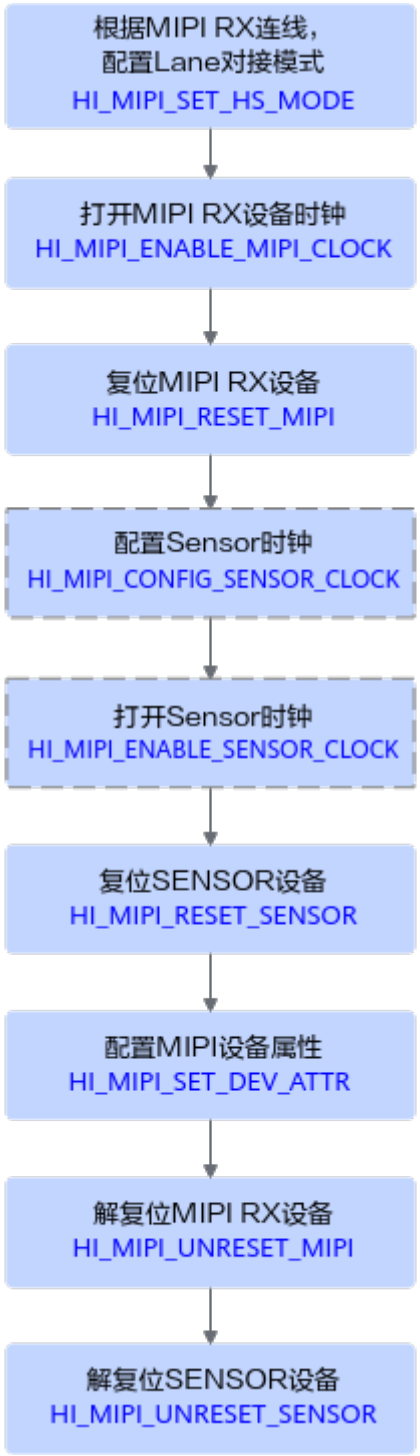


接口调用流程说明如下：

1. 调用hi_mpi_sys_init接口初始化媒体数据处理系统。
2. 使用MIPI Rx ioctl命令字初始化MIPI/Sensor硬件对接信息，接口调用流程请参见**初始化MIPI/SENSOR硬件对接信息**。

3. 使用VI（Vedio Input）功能接口初始化VI模块，接口调用流程请参见**初始化VI视频输入模块**。
4. 使用ISP（Image Signal Processing）系统控制接口初始化并运行ISP模块，接口调用流程请参见**初始化并运行ISP图像信号处理模块**。
5. 使用VI功能接口获取已处理的图像数据，接口调用流程请参见**获取已处理的图像数据**。
6. 使用ISP功能接口释放ISP模块资源，接口调用流程请参见**释放ISP图像信号处理模块资源**。
7. 使用VI功能接口释放VI模块资源，接口调用流程请参见**释放VI视频输入模块资源**。
8. 使用MIPI Rx ioctl命令字退出MIPI/Sensor硬件，接口调用流程请参见**退出MIPI/SENSOR硬件**。
9. 调用hi_mpi_sys_exit接口释放媒体数据处理系统资源。

初始化 MIPI/SENSOR 硬件对接信息

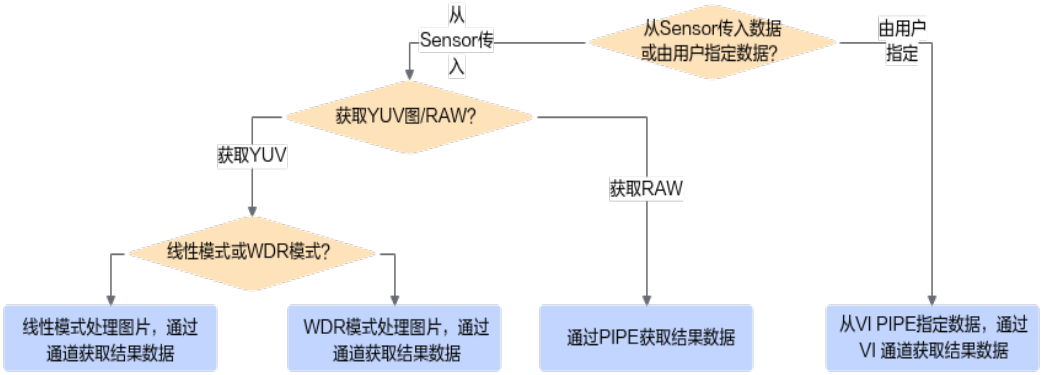


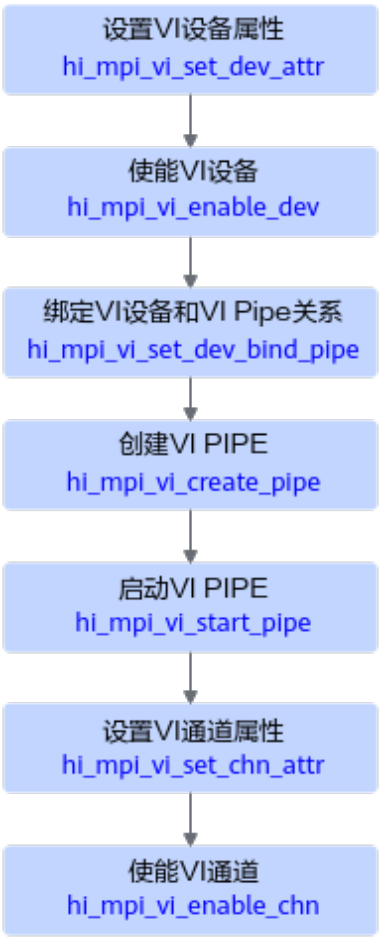
1. 使用HI_MIPI_SET_HS_MODE命令字设置模式。
2. 使用HI_MIPI_ENABLE_MIPI_CLOCK命令字打开MIPI时钟。
3. 使用HI_MIPI_RESET_MIPI命令字复位SENSOR所对接的MIPI。
4. （可选）使用HI_MIPI_CONFIG_SENSOR_CLOCK命令字配置SENSOR时钟。
5. （可选）使用HI_MIPI_ENABLE_SENSOR_CLOCK命令字打开SENSOR时钟。

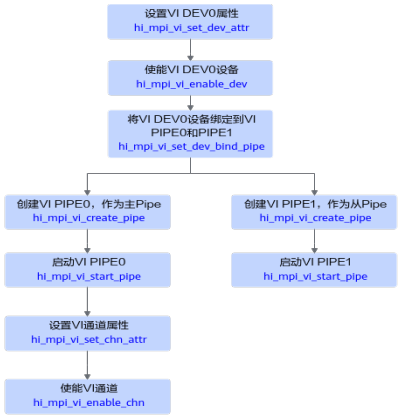
6. 使用HI_MIPI_RESET_SENSOR命令字复位SENSOR。
7. 使用HI_MIPI_SET_DEV_ATTR命令字配置MIPI Rx/设备属性。
8. 使用HI_MIPI_UNRESET_MIPI命令字撤销复位MIPI。
9. 使用HI_MIPI_UNRESET_SENSOR命令字撤销复位SENSOR。

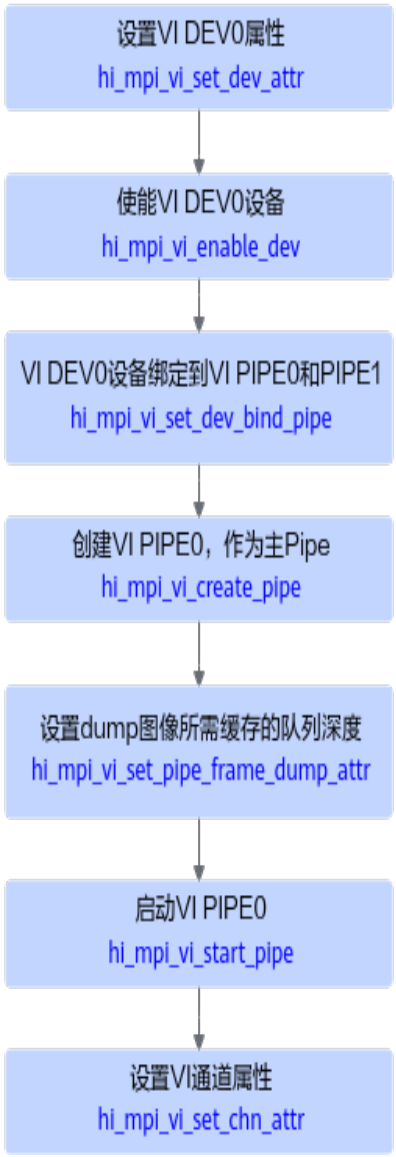
初始化 VI 视频输入模块

不同数据来源、不同数据格式、不同模式，初始化VI视频输入模块的流程不同。

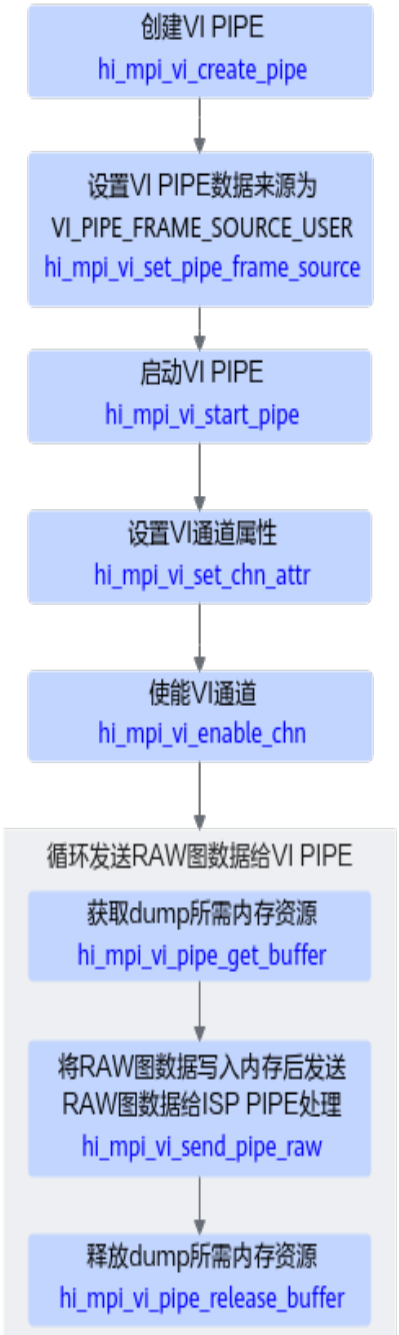


处理方式	接口调用流程	流程说明
从Sensor传入数据，若要获取YUV格式的数据，则通过VI通道处理，线性模式		1. 依次调用hi_mpi_vi_set_dev_attr、hi_mpi_vi_enable_dev接口，完成VI设备的属性配置并使能。 2. 调用hi_mpi_vi_set_dev_bind_pipe接口，完成设备和PIPE的绑定关系设置。 3. 依次调用hi_mpi_vi_create_pipe、hi_mpi_vi_start_pipe接口，创建并启动VI PIPE。需要合理设置hi_vi_pipe_attr.depth队列深度，队列深度越大，抗抖动性越好，建议设置为3或以上值。 4. 依次调用hi_mpi_vi_set_chn_attr、hi_mpi_vi_enable_chn接口，完成VI通道的属性配置并使能。需要合理设置hi_vi_chn_attr.depth队列深度，该队列深度除了要考虑VI内部处理预留内存外，还需结合用户自己的图像业务处理时长（从用户调用hi_mpi_vi_get_chn_frame接口取走图像资源，到用户调用hi_mpi_vi_release_chn_frame接口归还图像资源的时间间隔），合理设置队列深度大小。

处理方式	接口调用流程	流程说明
从Sensor传入数据，若要获取YUV格式的数据，则通过VI通道处理，WDR模式		相对于普通线性模式，WDR模式下，Sensor模组会通过长短曝光方式同时产生两帧图像数据，VI需要创建两个PIPE资源，并将两个PIPE绑定到同一个VI设备上，分别接收和处理对应的长短曝光帧图像，然后在主PIPE对应的通道中，输出长短曝光融合后的图像数据。所以，接口调用流程存在如下差异： 1. 需要调用hi_mpi_vi_set_dev_bind_pipe，将同一个Sensor设备的图像数据，绑定到两个Pipe上去，按示例图，将DEV0设备绑定到PIPE0和PIPE1上。 2. 需要通过接口hi_mpi_vi_create_pipe、hi_mpi_vi_start_pipe创建并启动两个PIPE，按示例图，创建了PIPE0和PIPE1，其中PIPE0作为主PIPE，接收并处理短曝光帧，PIPE1作为从PIPE，接收并处理长曝光帧。 **说明** WDR模式下，pipe0和pipe1为一组，pipe0为主pipe，pipe1和pipe2为一组，pipe1为主pipe。 3. 只需启动主PIPE上的通道，从PIPE上的通道可不使能，节省资源。

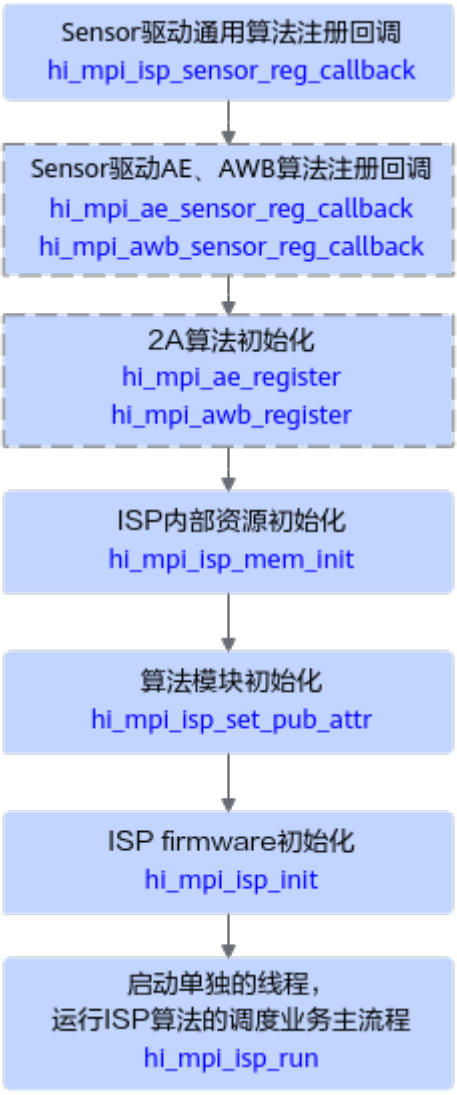
处理方式	接口调用流程	流程说明
若要获取RAW格式的数据，则通过VI PIPE处理		1. 调用hi_mpi_vi_set_dev_attr、hi_mpi_vi_enable_dev接口，完成VI设备的属性配置并使能。 2. 调用hi_mpi_vi_set_dev_bind_pipe接口，完成设备和Pipe的绑定关系设置。 3. 调用接口hi_mpi_vi_create_pipe创建PIPE。 a. 如果用户只需要获取RAW图，不需要图像经过VI处理和转换，则在hi_mpi_vi_create_pipe创建pipe时，可执行以下操作： a. 将pipe_bypass_mode设置为HI_VI_PIPE_BYPASS_BE，不经过ISP BE处理。 b. 设置hi_vi_pipe_attr.depth大小为hi_vi_dump_attr.depth，除了dump所需图像队列外，不额外申请多余的图像资源。 c. 需要调用hi_mpi_vi_set_chn_attr，不需要调用hi_mpi_vi_enable_chn接口使能VI通道。 b. 如果用户除了获取RAW图，还需要继续将图像送给VI处理和转换，则 a. 在hi_mpi_vi_create_pipe创建pipe时，需要合理设置hi_vi_pipe_attr.depth的值，该值需要考虑在hi_vi_dump_attr.depth大小的基础上，额外预留部分VI PIPE内部处理所需队列深度，一般为hi_vi_dump_attr.depth + 3。

处理方式	接口调用流程	流程说明
		b. 需要继续调用hi_mpi_vi_set_chn_attr、hi_mpi_vi_enable_chn接口，使能VI通道，并调用接口hi_mpi_vi_get_chn_frame获取VI处理后的图像结果数据并处理，处理完成后调用hi_mpi_vi_release_chn_frame接口释放对应图像的内存资源。 4. 要调用接口hi_mpi_vi_set_pipe_frame_dump_attr设置采图所需预留的图像队列深度。 5. 调用hi_mpi_vi_start_pipe接口启动PIPE。

处理方式	接口调用流程	流程说明
由用户指定RAW图数据，VI PIPE灌入并处理，获取YUV图		用户回灌图片场景，图片的数据来源不再是外部的摄像头设备，但因为当前版本还不支持虚拟pipe，只能通过物理pipe进行灌图，所以即使数据不从sensor输入，仍旧需要设置对应dev并调用hi_mpi_vi_set_dev_bind_pipe接口做dev和pipe的绑定。 1. 调用hi_mpi_vi_create_pipe接口创建PIPE。 2. 调用hi_mpi_vi_set_pipe_frame_source接口将PIPE的图像数据来源设置为VI_PIPE_FRAME_SOURCE_USER。 3. 调用hi_mpi_vi_start_pipe接口启动PIPE。 4. 调用hi_mpi_vi_set_chn_attr、hi_mpi_vi_enable_chn接口，完成VI通道的属性配置并使能。 5. 开始循环发送用户指定的图片数据。 a. 调用hi_mpi_vi_pipe_get_buffer获取空闲的图像数据，所能获取的最大可用内存数量由hi_mpi_vi_create_pipe接口下发的hi_vi_pipe_attr.depth属性决定。 b. 成功获取到可用的内存资源后，将需要灌入的图像数据写入返回的内存地址中，内存地址为hi_mpi_vi_pipe_get_buffer接口返回的frame_info.v_frame.virt_addr[0]，然后调用接口hi_mpi_vi_send_pipe_raw发送RAW图数据。 c. hi_mpi_vi_send_pipe_raw发送数据成功后，需要及时调用hi_mpi_vi_pipe_release_buffer接口，释放内存资源。

处理方式	接口调用流程	流程说明
		d. 图示为用户发送bayer格式图像的流程，如果用户需要发送YUV格式数据，则需要在调用hi_mpi_vi_create_pipe接口创建pipe时，指定像素格式pixel_format为YUV，并将isp_bypass设置为true，并将接口hi_mpi_vi_send_pipe_raw修改为hi_mpi_vi_send_pipe_yuv。当前版本支持发送的YUV图像格式为：HI_PIXEL_FORMAT_YVU_SEMIPLANAR_422、HI_PIXEL_FORMAT_YVU_SEMIPLANAR_420、HI_PIXEL_FORMAT_YUV_400。

初始化并运行 ISP 图像信号处理模块



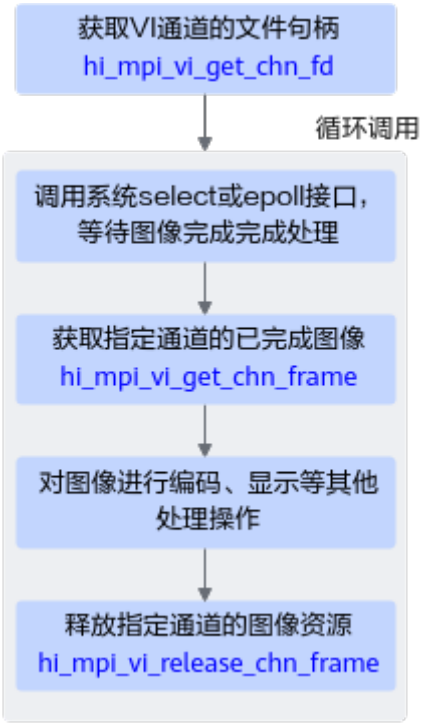
1. 调用hi_mpi_isp_sensor_reg_callback接口注册Sensor驱动通用算法。
2. （可选）调用hi_mpi_ae_sensor_reg_callback接口、hi_mpi_awb_sensor_reg_callback接口注册系统内置的Sensor驱动AE、AWB算法。

   此处用户可以根据需求注册自定义的算法。
3. （可选）调用hi_mpi_ae_register接口、hi_mpi_awb_register接口初始化系统内置的2A算法。

   此处用户可以根据需求注册自定义的算法。
4. 调用hi_mpi_isp_mem_init接口初始化ISP内部资源。
5. 调用hi_mpi_isp_set_pub_attr接口初始化算法模块。
6. 调用hi_mpi_isp_init接口初始化ISP firmware。
7. 启用单独线程，调用hi_mpi_isp_run接口运行ISP算法的调度业务主流程。

获取已处理的图像数据

- **获取YUV数据**



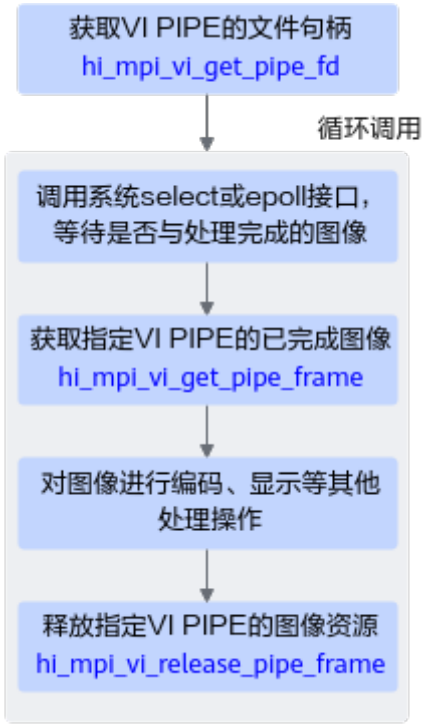
VI图像处理完成后，可在对应的VI通道上获取已完成图像并进行相关处理，典型接口调用流程如下：

a. （可选）通过系统文件句柄+select/epoll等待机制，等待图像处理完成事件，可通过hi_mpi_vi_get_chn_fd接口获取指定通道的系统文件句柄，然后获取并处理完一帧图像数据后，会唤醒系统的select/epoll读等待请求。

b. 调用hi_mpi_vi_get_chn_frame接口，获取已处理完成的图像数据。此时图像数据对应内存资源会自动被用户占用，用户必须在处理完图像数据后，调用hi_mpi_vi_release_chn_frame接口释放对应图像的内存资源。

c. 如果用户通过hi_mpi_vi_get_chn_frame接口获取到图像数据后，要再发布给其他进程使用，则可通过返回的hi_video_frame.user_data[0]得到acltdtBuf句柄，再结合acl的共享buf管理接口(如acltdtCopyBufRef)以及共享队列管理接口(如acltdtEnqueue)将对象发布给其他进程使用。

- **获取RAW数据**



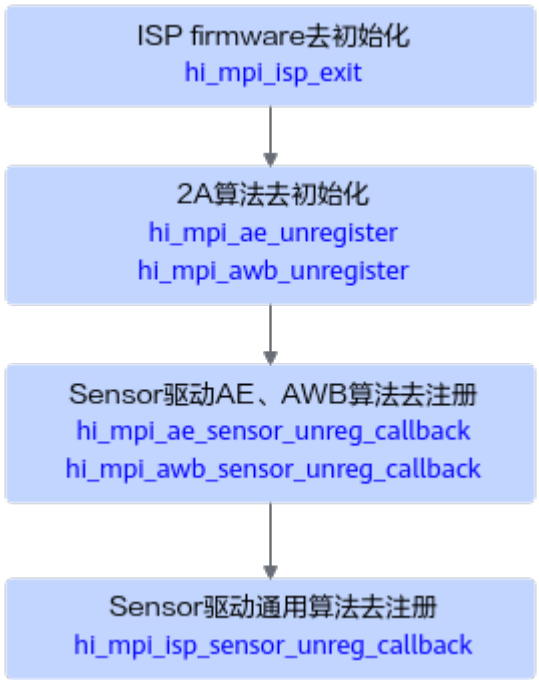
VI图像处理完成后，可在对应的VI PIPE上获取已完成图像并进行相关处理，典型接口调用流程如下：

a. （可选）通过系统文件句柄+select/epoll等待机制，等待图像处理完成事件，可通过hi_mpi_vi_get_pipe_fd接口获取指定通道的系统文件句柄，当后台获取并处理完一帧图像数据后，会唤醒系统的select/epoll读等待请求。

b. 调用hi_mpi_vi_get_pipe_frame接口，获取已处理完成的图像数据。此时图像数据对应内存资源会自动被用户占用，用户必须在处理完图像数据后，调用hi_mpi_vi_release_pipe_frame接口释放对应图像的内存资源。

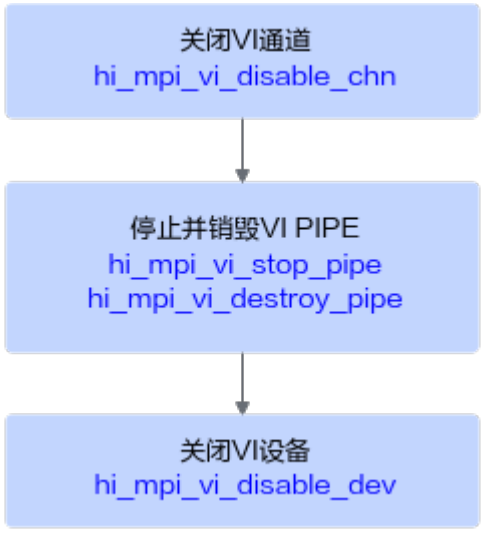
c. 如果用户通过hi_mpi_vi_get_pipe_frame接口获取到图像数据后，要再发布给其他进程使用，则可通过返回的hi_video_frame.user_data[0]得到acltdtBuf句柄，再结合acl的共享buf管理接口(如acltdtCopyBufRef)以及共享队列管理接口(如acltdtEnqueue)将对象发布给其他进程使用。

释放 ISP 图像信号处理模块资源



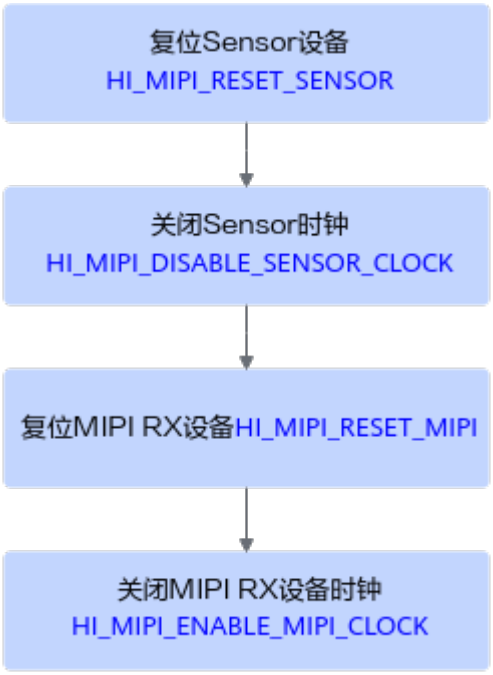
1. 调用hi_mpi_isp_exit接口去初始化ISP firmware。
2. 调用hi_mpi_ae_unregister接口、hi_mpi_awb_unregister接口去初始化2A算法。
3. 调用hi_mpi_ae_sensor_unreg_callback接口、hi_mpi_awb_sensor_unreg_callback接口取消注册Sensor驱动AE、AWB算法。
4. 调用hi_mpi_isp_sensor_unreg_callback接口取消注册Sensor驱动通用算法。

释放 VI 视频输入模块资源



1. 调用hi_mpi_vi_disable_chn接口关闭VI通道。
2. 依次调用hi_mpi_vi_stop_pipe、hi_mpi_vi_destroy_pipe接口停止并销毁VI PIPE。
3. 调用hi_mpi_vi_disable_dev接口关闭VI设备。

退出 MIPI/SENSOR 硬件

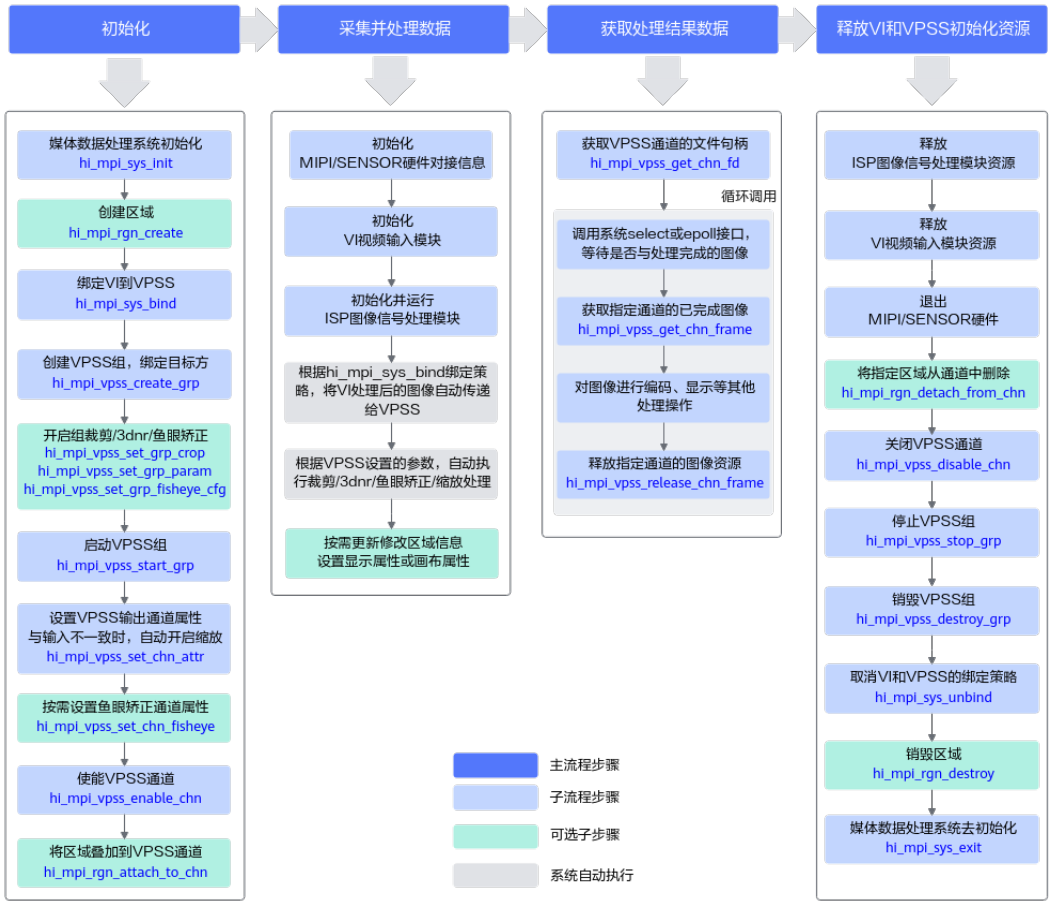


1. 使用HI_MIPI_RESET_SENSOR命令字复位SENSOR。
2. 使用HI_MIPI_DISABLE_SENSOR_CLOCK命令字关闭SENSOR所连接的时钟。
3. 使用HI_MIPI_RESET_MIPI命令字复位SENSOR所对接的MIPI。
4. 使用HI_MIPI_DISABLE_MIPI_CLOCK关闭MIPI。

9.6.2 视频数据获取+VPSS 视频处理功能

VPSS必须配合VI模块一起使用，本节介绍其接口调用流程及注意事项。

图 9-18 VPSS 调用流程



接口调用流程说明如下：

1. **初始化**：
   a. 调用hi_mpi_sys_init接口完成媒体系统初始化。
   b. 调用hi_mpi_rgn_create创建区域。
   c. VPSS模块只能作为被绑定方，被动接收视频流数据并处理，所以需要调用hi_mpi_sys_bind接口，将规划好的VI通道绑定到VPSS组上。
   d. 调用hi_mpi_vpss_create_grp接口创建VPSS组，作为hi_mpi_sys_bind接口的被绑定方。
   e. 按需开启VPSS上功能：
      - 如果需要做组裁剪，则调用hi_mpi_vpss_set_grp_crop接口，设置裁剪相关配置。
      - 如果需要开启3DNR，则除了在hi_mpi_vpss_create_grp接口时，打开nr开关并配置nr属性，还可通过hi_mpi_vpss_set_grp_param接口设置NR高级参数。
      - 如果需要开启鱼眼矫正，并通过hi_mpi_vi_set_chn_fisheye接口中hi_fisheye_attr属性开启了LMF(Lens Map Function)参数功能， 则必须调用hi_mpi_vpss_set_grp_fisheye_cfg接口设置LMF参数。
   f. 调用hi_mpi_vpss_start_grp接口启用VPSS组。

g. 调用hi_mpi_vpss_set_chn_attr接口设置通道属性，通道输出分辨率可以与输入源分辨率不一致，当不一致时，自动开启缩放，不同通道的缩放能力不一样。

h. 如果需要开启鱼眼畸变矫正，则还需要调用hi_mpi_vi_set_chn_fisheye接口，设置鱼眼矫正参数。

i. 调用hi_mpi_vpss_enable_chn接口启动VPSS通道。

j. 调用hi_mpi_rgn_attach_to_chn接口将区域叠加到VPSS通道上。

2. **采集并处理数据**：

a. 使用MIPI Rx ioctl命令字初始化MIPI/Sensor硬件对接信息，接口调用流程请参见**初始化MIPI/SENSOR硬件对接信息**。

b. 使用VI（Vedio Input）功能接口初始化VI模块，接口调用流程请参见**初始化VI视频输入模块**。

c. 使用ISP（Image Signal Processing）系统控制接口初始化并运行ISP模块，接口调用流程请参见**初始化并运行ISP图像信号处理模块**。

d. 根据hi_mpi_sys_bind接口设置的绑定策略，系统内部自动将VI处理后的图像自动传递给VPSS；

e. 根据VPSS设置的参数，系统内部自动执行裁剪/3dnr/鱼眼矫正/缩放处理。

f. 按需更新修改区域信息：

- 设置区域通道显示属性：
  1) 调用hi_mpi_rgn_get_display_attr接口获取区域当前的通道显示属性。
  2) 修改通道显示属性结构体重参数，调用hi_mpi_rgn_set_display_attr接口设置区域的通道显示属性。
- 设置区域的显示画布信息：
  1) 调用hi_mpi_rgn_get_canvas_info接口获取当前区域的显示画布信息。
  2) 调用hi_mpi_rgn_update_canvas接口更新显示画布信息。

3. **获取处理结果数据**：

此时，用户可调用hi_mpi_vpss_get_chn_fd接口获取VPSS指定通道的句柄，并通过select/epoll接口等待VPSS处理结果。VPSS处理完后，会自动唤醒select/epoll等待，此时可调用hi_mpi_vpss_get_chn_frame接口获取VPSS处理后的图像数据，做后续处理，最后调用hi_mpi_vpss_release_chn_frame释放一帧通道图像。

4. **释放VI和VPSS初始化资源**。

a. 使用ISP功能接口释放ISP模块资源，接口调用流程请参见**释放ISP图像信号处理模块资源**。

b. 使用VI功能接口释放VI模块资源，接口调用流程请参见**释放VI视频输入模块资源**。

c. 使用MIPI Rx ioctl命令字退出MIPI/Sensor硬件，接口调用流程请参见**退出MIPI/SENSOR硬件**。

d. 调用hi_mpi_rgn_detach_from_chn接口将指定区域从通道中删除。

e. 调用hi_mpi_vpss_disable_chn接口关闭VPSS通道。

f. 调用hi_mpi_vpss_stop_grp接口停止VPSS组。

g. 调用hi_mpi_vpss_destroy_grp接口销毁VPSS组。

h. 调用hi_mpi_sys_unbind接口取消VI和VPSS的绑定。
i. 调用hi_mpi_rgn_destroy接口销毁区域。
j. 最后调用hi_mpi_sys_exit接口完成后媒体系统的退出。

9.7 NVR 场景视频解码、处理和显示

本节介绍NVR视频业务处理的典型流程、关键接口及注意事项。

NVR，全称Network Video Recorder，即网络视频录像机，是网络视频系统的存储转发部分，NVR与网络摄像机协同工作，完成音频&视频的录像、存储及转发功能，同时，NVR具备本地人机交互界面、视频解码、视频显示及语音对讲功能。

NVR 视频场景说明

本章节描述NVR视频业务处理流程，包括下面两种：

1. **视频解码显示流程**

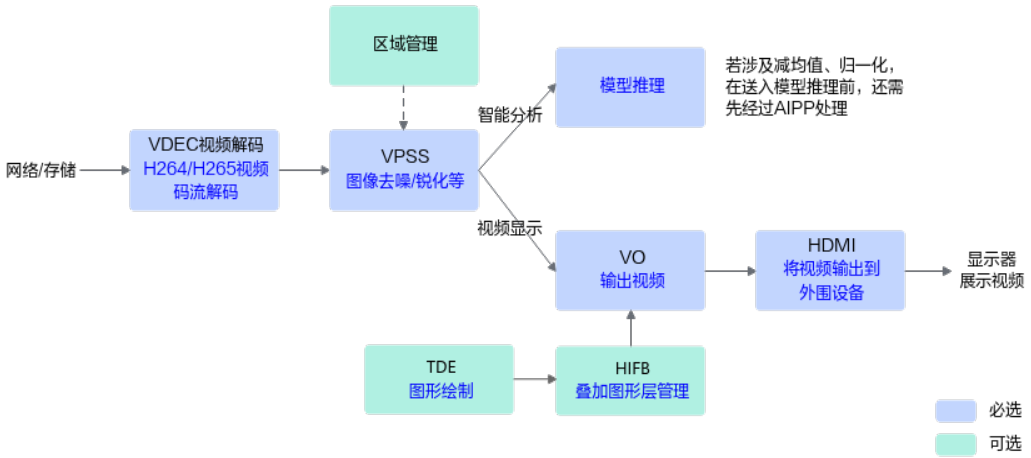
   视频解码显示流程涉及视频解码模块(VDEC)、视频处理模块(VPSS)、视频输出模块(VO)、HDMI接口模块, 以及增强的功能，例如**Region区域管理功能**、**TDE图形绘制功能**、**HIFB叠加图形层管理功能**。

   在该流程中，通过调用hi_mpi_sys_bind接口将VDEC通道与VPSS组绑定、将VPSS组与VO设备绑定，实现数据从VDEC到VPSS、从VPSS到VO的自动传输。通过底层寄存器，实现VO自动读取HIFB的输出数据。

   若TDE、HIFB与VO不在一个应用进程中，则需要确保先启动VO所在的进程，且TDE和HIFB所在的进程中也需要调用hi_mpi_sys_init接口初始化媒体数据处理系统、并在进程退出前调用hi_mpi_sys_exit接口实现媒体数据处理系统去初始化。

2. **视频解码智能分析流程**

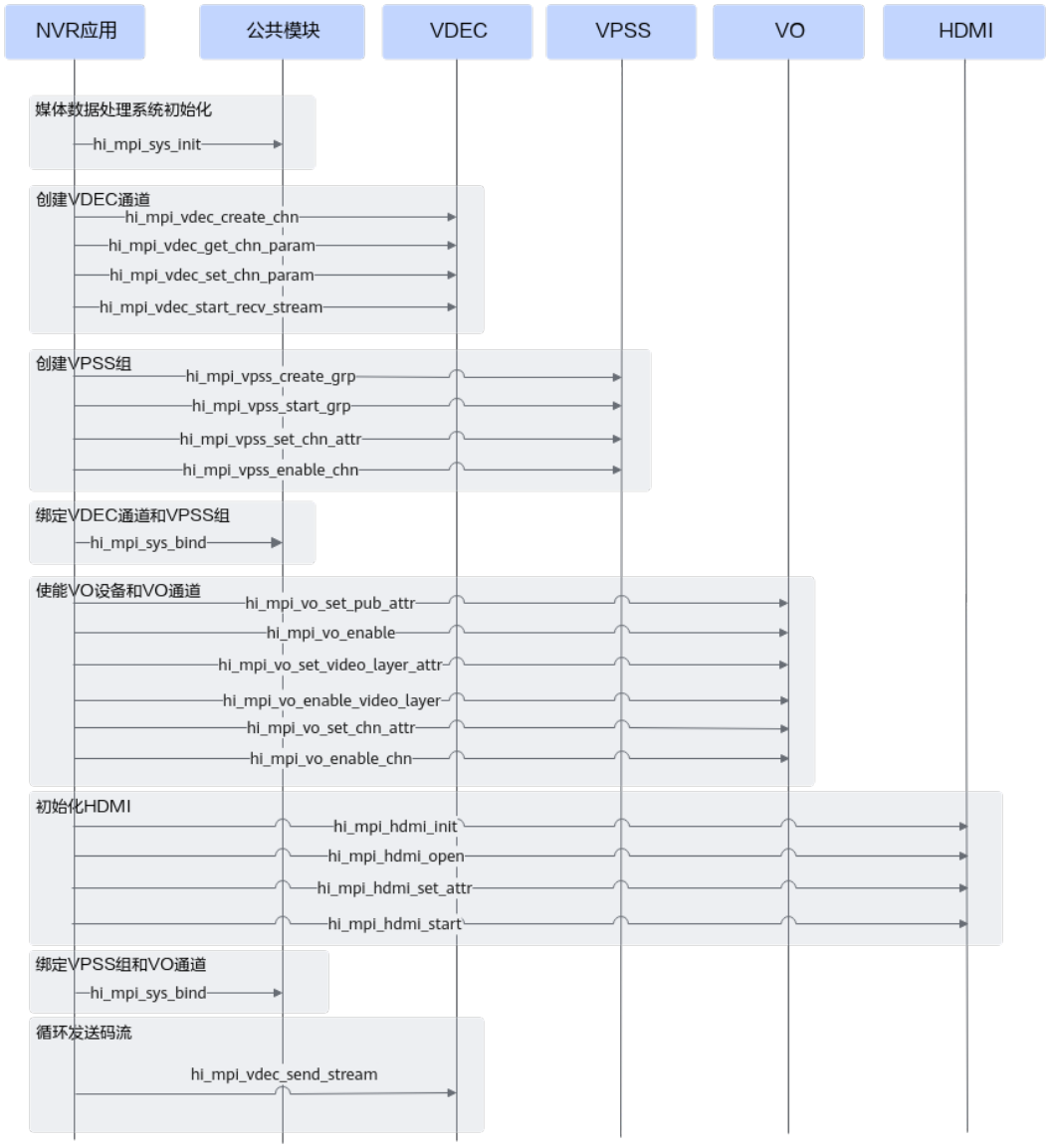
   视频解码智能分析流程涉及视频解码模块(VDEC)、视频处理模块(VPSS)、智能分析（模型推理）, 以及增强的功能，例如REGION区域管理模块。区域管理功能的接口调用流程请参见**Region区域管理功能**。



视频解码显示流程

视频解码显示流程涉及视频解码模块(VDEC)、视频处理模块(VPSS)、视频输出模块(VO)、HDMI接口模块, 以及增强的功能，例如**Region区域管理功能**、**TDE图形绘制功能**、**HIFB叠加图形层管理功能**。

图 9-19 业务启动、运行接口调用流程



NVR视频解码显示业务启动、运行接口调用流程说明如下：

1. 调用hi_mpi_sys_init接口完成**媒体系统初始化**。
2. **创建VDEC视频解码通道**（按需创建多个通道），并通知解码器启动接收码流。
   a. 调用hi_mpi_vdec_create_chn接口创建通道。
   b. 调用hi_mpi_vdec_get_chn_param接口获取通道属性、按需设置通道属性后，调用hi_mpi_vdec_set_chn_param接口设置通道参数。
   c. 解码前，需调用hi_mpi_vdec_start_recv_stream接口通知解码器启动接收码流。

3. **创建并启用VPSS组**（按需创建多个VPSS组）。
   a. 调用hi_mpi_vpss_create_grp接口创建VPSS组
   b. 调用hi_mpi_vpss_start_grp接口启用VPSS组。
   c. 调用hi_mpi_vpss_set_chn_attr接口设置通道属性，在和VO模块配合工作时，通道属性需要设置为Auto模式。
   d. 调用hi_mpi_vpss_enable_chn接口启动VPSS通道。
4. 调用hi_mpi_sys_bind接口**绑定视频解码通道和VPSS组**，经过VDEC视频解码的输出数据直接被送入对应的VPSS组继续处理。
   - 以单个显示器输出4路视频显示为例，绑定信息如下表所示：

VDEC通道	VPSS Group	VO通道	VO视频层	VO设备号	HDMI编号
1	1	1	视频层 VHD0	0	HDMI0
2	2	2			
3	3	3			
4	4	4			

   - 以两个显示器输出、每个显示器输出4路视频显示为例，绑定信息如下表所示:

VDEC通道	VPSS Group	VO通道	VO视频层	VO设备号	HDMI编号
1	1	1	视频层 VHD0	0	HDMI0
2	2	2			
3	3	3			
4	4	4			
5	5	5	视频层 VHD1	1	HDMI1
6	6	6			
7	7	7			
8	8	8			

5. **启用VO设备和VO通道**（按需创建多个VO通道）。
   a. 调用hi_mpi_vo_set_pub_attr配置显示设备属性，通过hi_mpi_vo_enable使能显示设备。
   b. 调用hi_mpi_vo_set_video_layer_attr配置显示视频层属性，通过hi_mpi_vo_enable_video_layer使能显示视频层。
   c. 调用hi_mpi_vo_set_chn_attr配置显示通道属性，通过hi_mpi_vo_enable_chn使能显示通道。
6. **初始化HDMI外设**。

a. 调用hi_mpi_hdmi_init初始化HDMI设备，调用hi_mpi_hdmi_open打开HDMI。

b. 调用hi_mpi_hdmi_set_attr配置HDMI属性。

c. 调用hi_mpi_hdmi_start启动HDMI外设，以便显示视频。

7. 调用hi_mpi_sys_bind接口**绑定VPSS组和VO通道**，经过VPSS处理后的输出数据直接被送入对应的VO通道继续处理。

   绑定信息请参见4。

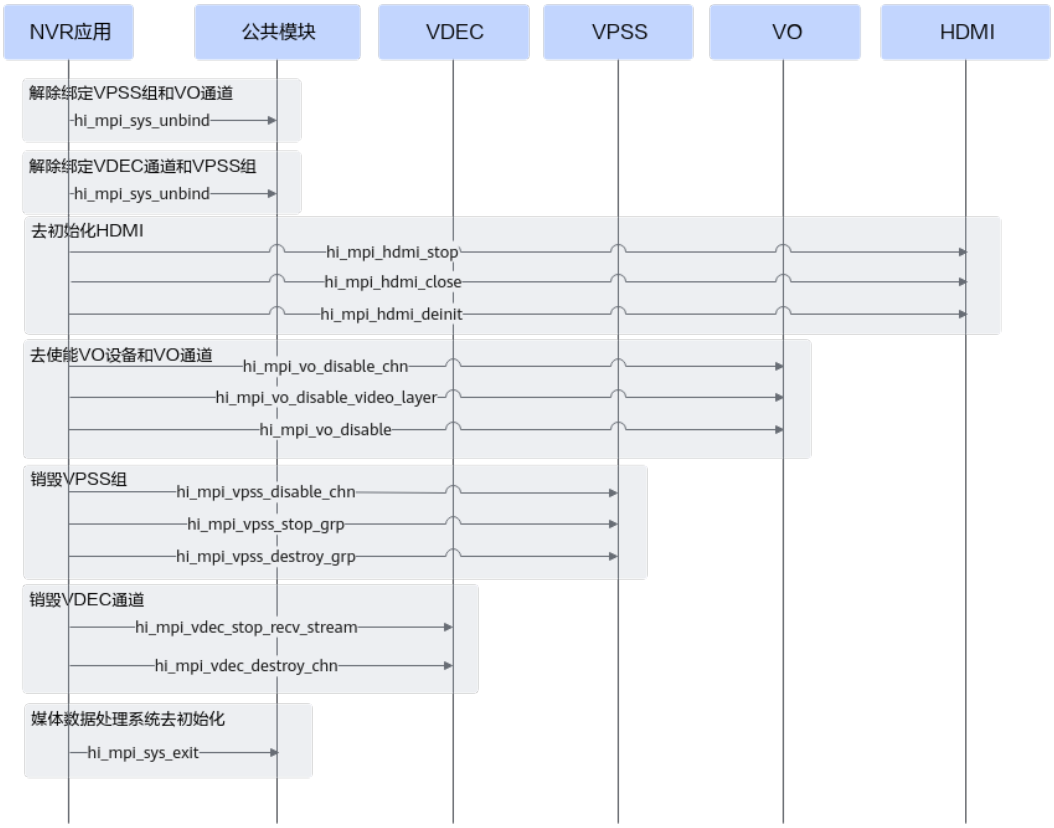
8. 循环调用hi_mpi_vdec_send_stream接口，**发送每一帧解码码流**。

   注意：

   a. 在视频解码通道和VPSS组绑定后，用户调用hi_mpi_vdec_send_stream发送码流时，接口参数中的vdec_pic_info可以设置为NULL, 此时的视频解码通道和VPSS模块绑定（参见4），解码结果数据直接被送入对应的VPSS组继续处理，不支持通过hi_mpi_vdec_get_frame接口获取解码结果数据。

   b. 视频解码支持两种模式，即回放模式和预览模式，可以通过hi_mpi_vdec_get_display_mode和hi_mpi_vdec_set_display_mode接口进行查询和设置，对于录像播放需要使用回放模式，此时支持播放控制。

图 9-20 资源释放接口调用流程



NVR视频解码显示业务资源释放接口调用流程说明如下：

1. 调用hi_mpi_sys_unbind接口**取消VO通道与VPSS组的绑定**。
2. 调用hi_mpi_sys_unbind接口**取消VPSS组与VDEC通道的绑定**。
3. **释放HDMI外设资源**。

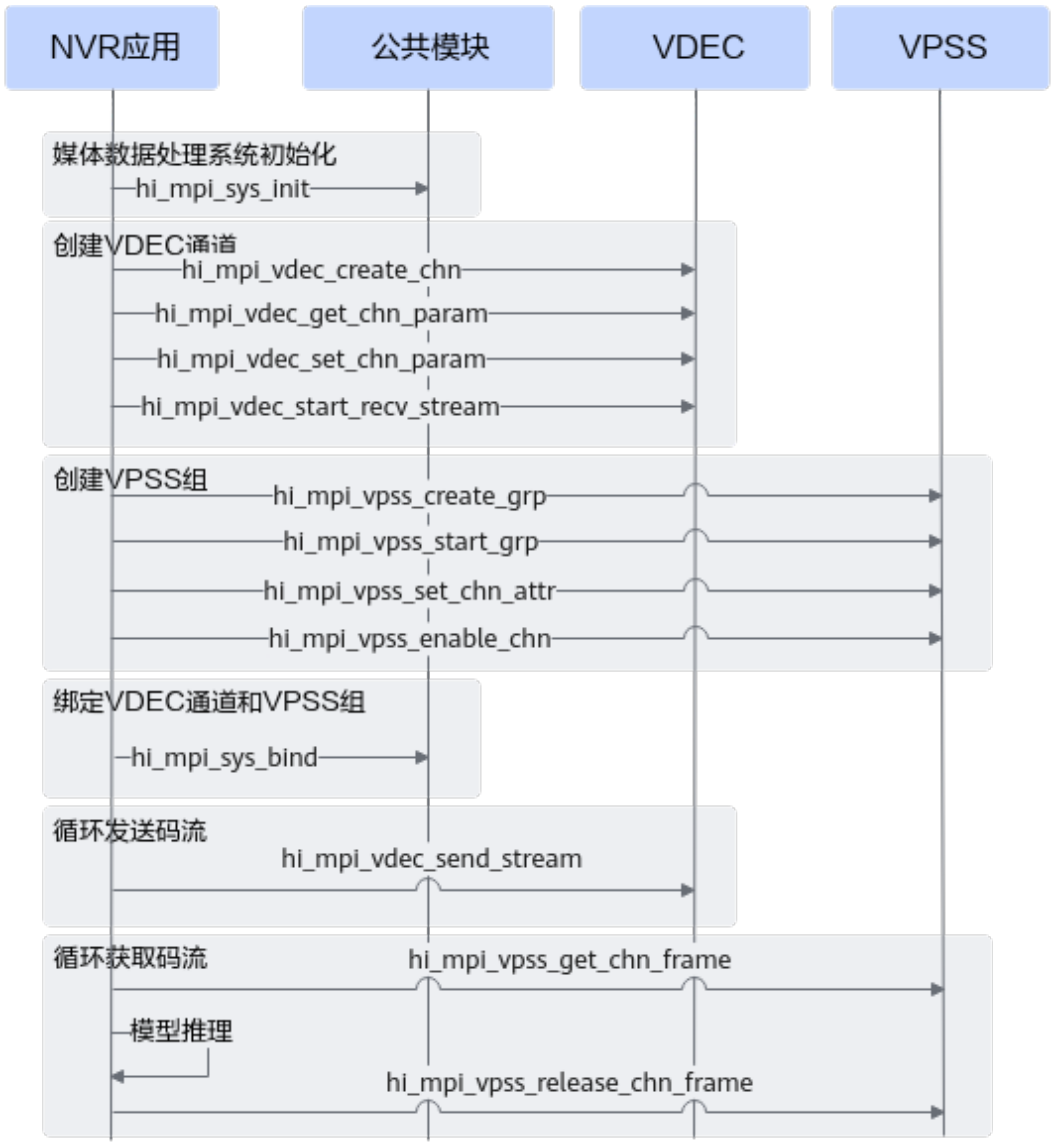
a. 首先调用hi_mpi_hdmi_stop停止HDMI。
b. 调用hi_mpi_hdmi_close关闭HDMI。
c. 调用hi_mpi_hdmi_deinit去初始化HDMI设备。

4. **释放VO设备、通道资源**。
   a. 调用hi_mpi_vo_disable_chn禁用通道。
   b. 调用hi_mpi_vo_disable_video_layer禁用视频层。
   c. 调用hi_mpi_vo_disable禁用显示设备。
5. **销毁VPSS组**。
   a. 调用hi_mpi_vpss_disable_chn接口关闭VPSS通道。
   b. 调用hi_mpi_vpss_stop_grp接口停止VPSS组。
   c. 调用hi_mpi_vpss_destroy_grp接口销毁VPSS组。
6. **销毁VDEC视频解码通道**。
   a. 调用hi_mpi_vdec_stop_recv_stream接口通知解码器停止接收码流。
   b. 调用hi_mpi_vdec_destroy_chn接口销毁通道。
7. 调用hi_mpi_sys_exit接口完成**媒体数据处理系统去初始化**。

视频解码智能分析流程

视频解码智能分析流程涉及视频解码模块(VDEC)、视频处理模块(VPSS)、智能分析（模型推理）, 以及增强的功能，例如REGION区域管理模块。区域管理功能的接口调用流程请参见**Region区域管理功能**。

图 9-21 业务启动、运行接口调用流程



视频解码智能分析业务启动、运行流程：

1. 调用hi_mpi_sys_init接口完成**媒体系统初始化**。
2. **创建VDEC视频解码通道**（按需创建多个通道），并通知解码器启动接收码流。
   a. 调用hi_mpi_vdec_create_chn接口创建通道。
   b. 调用hi_mpi_vdec_get_chn_param接口获取通道属性、按需设置通道属性后，调用hi_mpi_vdec_set_chn_param接口设置通道参数。
   c. 解码前，需调用hi_mpi_vdec_start_recv_stream接口通知解码器启动接收码流。
3. **创建并启用VPSS组**（按需创建多个VPSS组）。
   a. 调用hi_mpi_vpss_create_grp接口创建VPSS组
   b. 调用hi_mpi_vpss_start_grp接口启用VPSS组。
   c. 调用hi_mpi_vpss_set_chn_attr接口设置通道属性，通道属性需要设置为User模式。

d. 调用hi_mpi_vpss_enable_chn接口启动VPSS通道。

4. 调用hi_mpi_sys_bind接口**绑定视频解码通道和VPSS组**，经过VDEC视频解码的输出数据直接被送入对应的VPSS组继续处理。

   绑定信息请参见**4**。

5. 循环调用hi_mpi_vdec_send_stream接口，**发送每一帧解码码流**。

   注意：

   在视频解码通道和VPSS组绑定后，用户调用hi_mpi_vdec_send_stream发送码流时，接口参数中的vdec_pic_info可以设置为NULL, 此时的视频解码通道和VPSS模块绑定（参见**4**），解码结果数据直接被送入对应的VPSS组继续处理，不支持通过hi_mpi_vdec_get_frame接口获取解码结果数据。

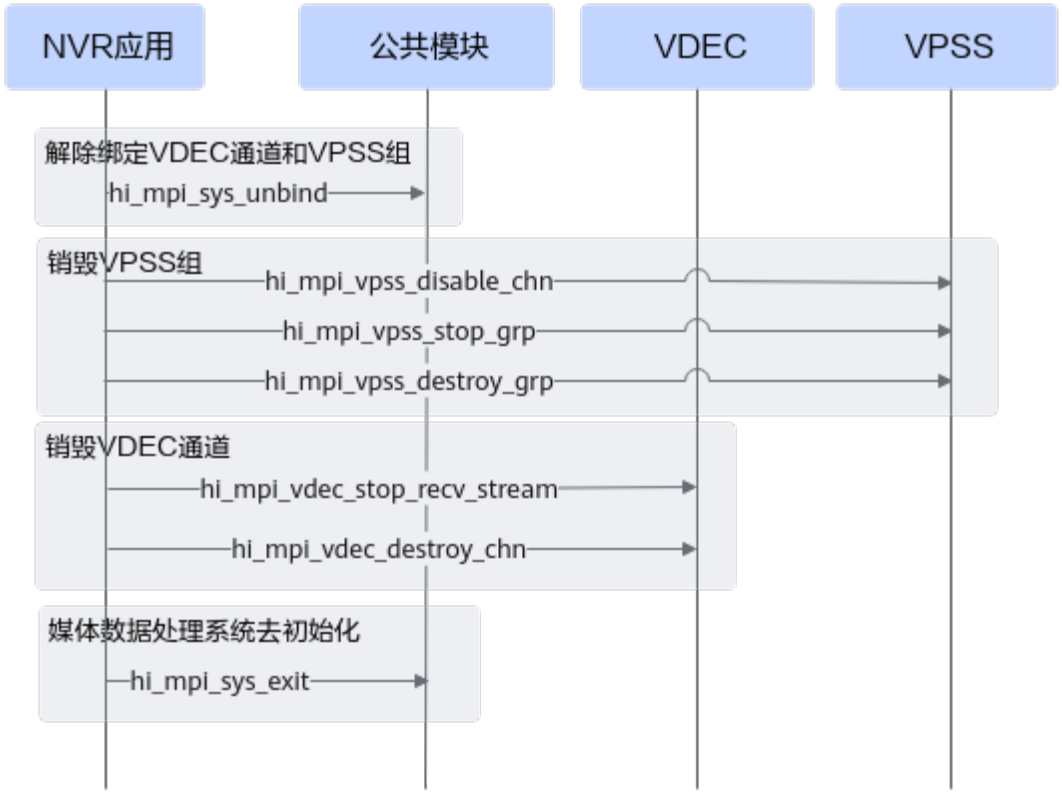
6. 调用hi_mpi_vpss_get_chn_frame接口**获取VPSS处理后的图像数据**，可送入模型推理（参见**8 模型推理**），推理结束后，最后调用hi_mpi_vpss_release_chn_frame**释放一帧通道图像**。

   注意：

   在多通道时，用户可调用hi_mpi_vpss_get_chn_fd接口获取VPSS指定通道的句柄，并通过调用epoll接口(参考hi_mpi_sys_create_epoll相关接口)等待VPSS处理结果。VPSS处理完后，会自动唤醒epoll等待，此时可调用hi_mpi_vpss_get_chn_frame接口获取VPSS处理后的图像数据。

   VPSS处理后的图像，在送入模型推理前，若图片尺寸、格式等不满足要求，需要经过DVPP的VPC功能模块、AIPP功能进一步处理，请参见**9.1 媒体数据处理基础知识**中关于VPC、AIPP的介绍。

图 9-22 资源释放接口调用流程



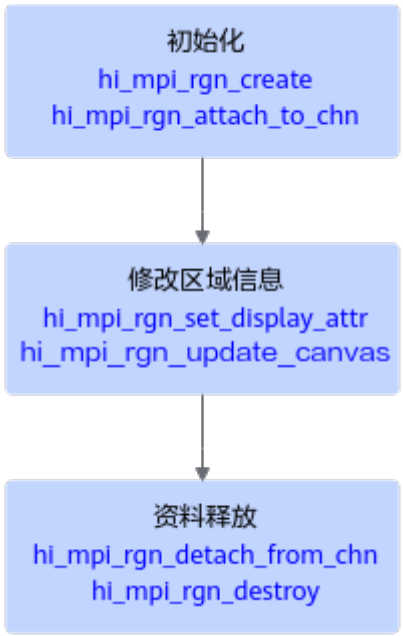
NVR视频解码智能分析业务资源释放接口调用流程说明如下：

1. 调用hi_mpi_sys_unbind接口**取消VPSS组与VDEC通道的绑定**。
2. **销毁VPSS组**。
   a. 调用hi_mpi_vpss_disable_chn接口关闭VPSS通道。
   b. 调用hi_mpi_vpss_stop_grp接口停止VPSS组。
   c. 调用hi_mpi_vpss_destroy_grp接口销毁VPSS组。
3. **销毁VDEC视频解码通**道。
   a. 调用hi_mpi_vdec_stop_recv_stream接口通知解码器停止接收码流。
   b. 调用hi_mpi_vdec_destroy_chn接口销毁通道。
4. 调用hi_mpi_sys_exit接口完成**媒体数据处理系统去初始化**。

Region 区域管理功能

叠加在视频上的OSD (On Screen Display)和遮挡在视频上的色块统称为区域。**区域管理模块**，用于统一管理这些区域资源，用于在视频上显示一些特定信息（如通道号、时间戳等）、或在视频中填充色块用于遮挡。

区域管理功能（REGION）必须配合VPSS模块一起使用，且区域管理功能需关联的VPSS组、VPSS通道已创建，接口调用流程如下所示。



1. **初始化**：
   a. 调用hi_mpi_rgn_create创建区域。
   b. 调用hi_mpi_rgn_attach_to_chn接口将区域叠加到VPSS通道上。
2. **按需修改区域信息**：
   - 设置区域通道显示属性：
     i. 调用hi_mpi_rgn_get_display_attr接口获取区域当前的通道显示属性。
     ii. 修改通道显示属性结构体重参数，调用hi_mpi_rgn_set_display_attr接口设置区域的通道显示属性。
   - 设置区域的显示画布信息：

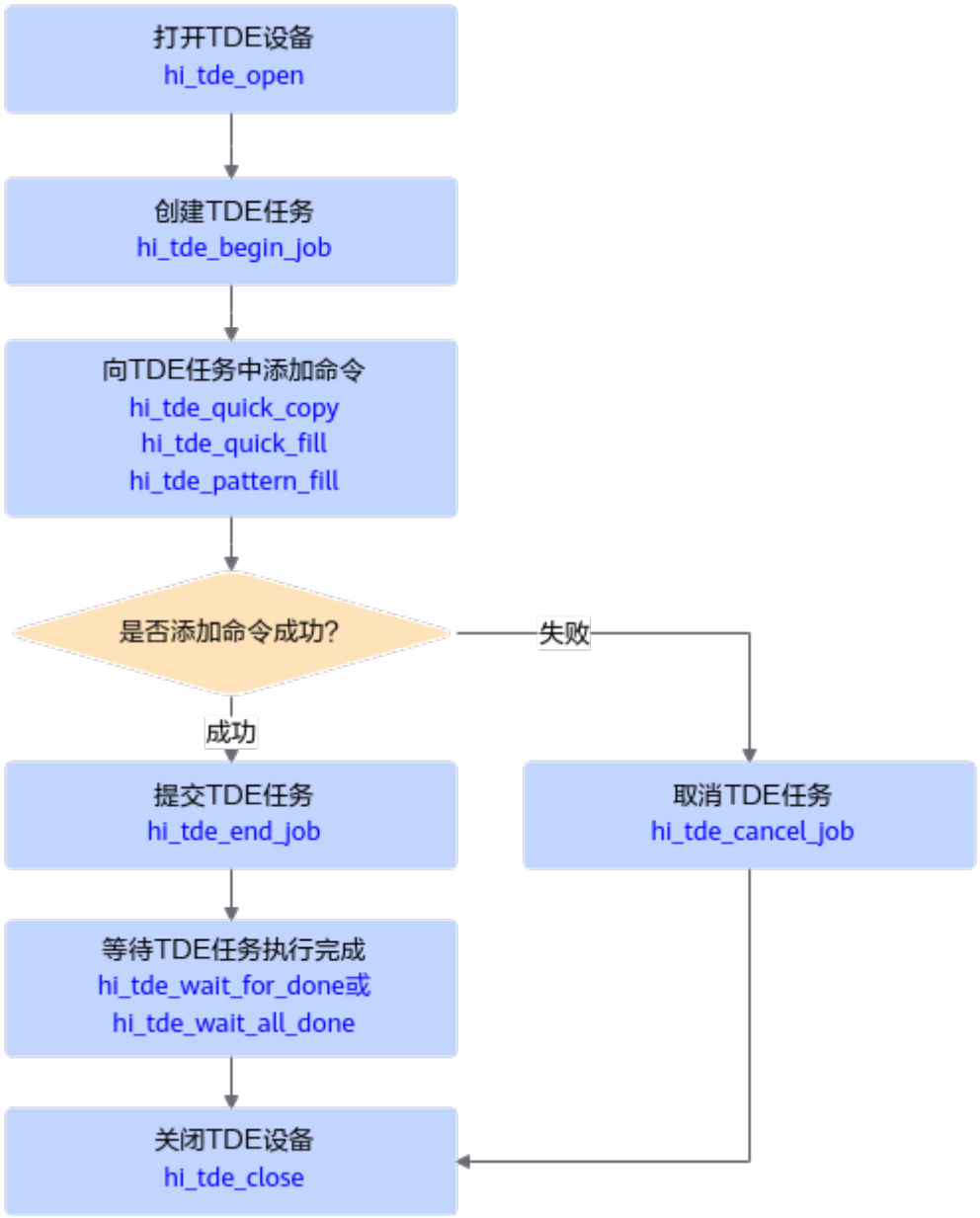
i. 调用hi_mpi_rgn_get_canvas_info接口获取当前区域的显示画布信息。
ii. 调用hi_mpi_rgn_update_canvas接口更新显示画布信息。

3. **资源释放**：
   a. 调用hi_mpi_rgn_detach_from_chn接口将指定区域从VPSS通道中删除。
   b. 调用hi_mpi_rgn_destroy接口销毁区域。

TDE 图形绘制功能

TDE是图形二维加速引擎，它利用硬件为 OSD（On Screen Display）和 GUI（Graphics User Interface）提供快速的图形绘制功能，主要有快速拷贝、快速色彩填充、模式填充（当前仅支持Alpha Blending操作）。

图 9-23 TDE 接口调用流程



1. 调用hi_tde_open接口**打开TDE设备**。
2. 调用hi_tde_begin_job接口**创建TDE任务**。
3. 调用各**命令执行**接口，例如hi_tde_quick_copy、hi_tde_quick_fill、hi_tde_pattern_fill。

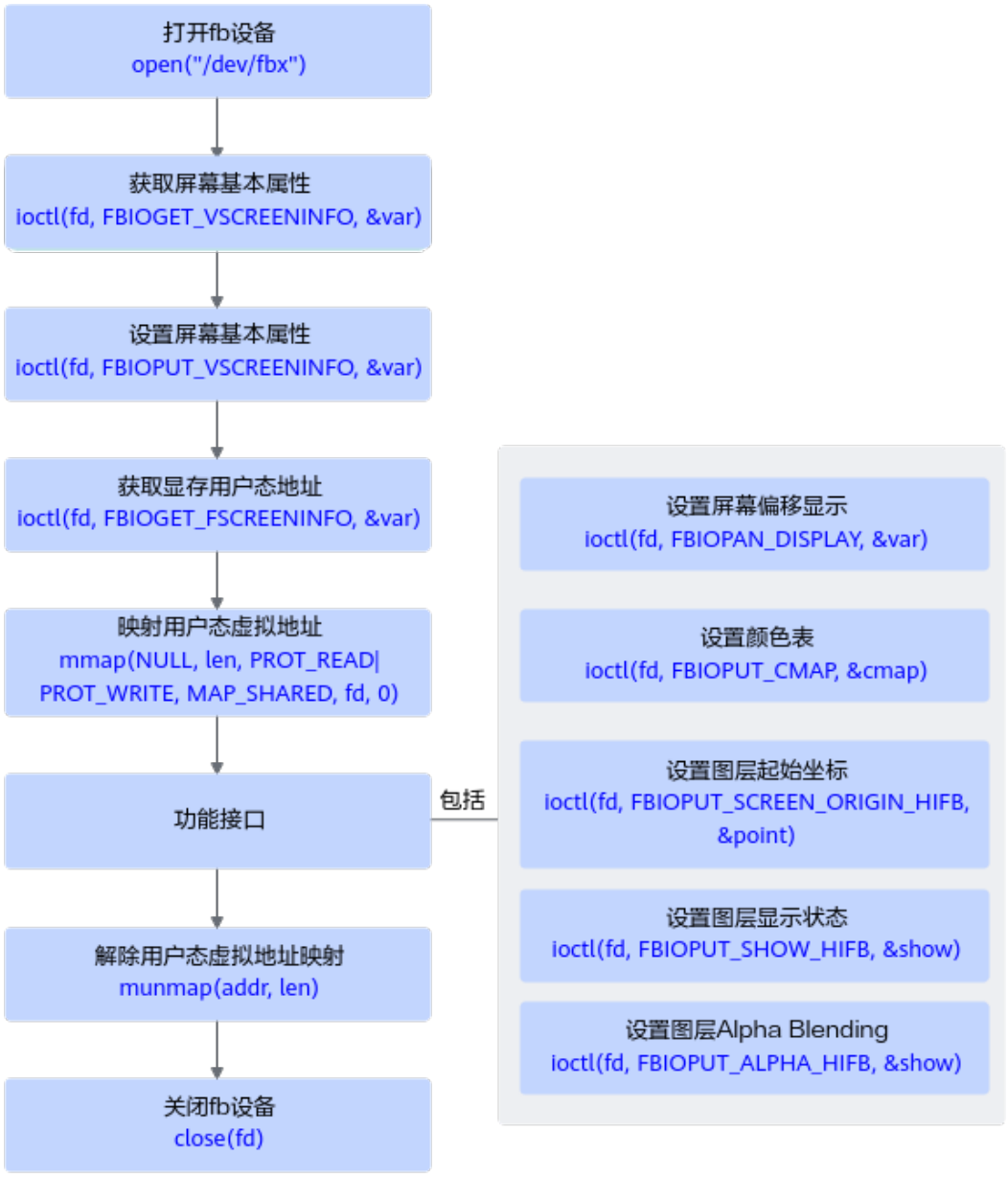
   调用命令执行接口前，需先调用HIFB提供的**int ioctl (int fd, FBIOGET_FSCREENINFO, fb_fix_screeninfo *fix)**接口获取显存用户态地址，作为目标位图的内存地址，TDE任务执行完成后，目标位图的数据会存放在该内存地址中，作为HIFB的输入数据。
4. 若添加命令失败，则调用hi_tde_cancel_job接口取消任务；若添加命令成功，则调用hi_tde_end_job接口提交任务。
5. **等待TDE任务完成**。

   目前有等待指定TDE任务执行完成（调用hi_tde_wait_for_done接口）、等待当前TDE设备上所有任务执行完成（调用hi_tde_wait_all_done接口）两种方式。

HIFB 叠加图形层管理功能

HIFB用于管理叠加图形层，它不仅提供Linux Framebuffer的基本功能，还在Linux Framebuffer的基础上增加图层显示起始位置修改、层间Alpha等扩展功能。

图 9-24 HIFB 接口调用流程



1. 通过系统调用open**打开fb设备**。设备文件fb0~fb4对应图层G0~G4。其中，G0和G1为高清图层、G2为鼠标图层、G3和G4为标清图层。

   各图层对应的fb设备、VO设备、支持的颜色格式、分辨率等说明，请参见表1。
2. 通过系统调用ioctl，传入命令码FBIOPUT_VSCREENINFO**设置屏幕基本属性**。
3. 通过系统调用ioctl，传入命令码FBIOGET_FSCREENINFO**获取显存用户态地址**。
4. 通过系统调用mmap**映射用户态虚拟地址**。
5. 通过系统调用ioctl，传入功能相关的命令码**设置功能属性**。
6. 通过系统调用munmap**解除用户态虚拟地址映射**。
7. 通过系统调用close**关闭fb设备**。

   如果HIFB与VO在同一个进程中配合使用，则需要在禁用VO设备（即调用hi_mpi_vo_disable）后关闭fb设备。

如果HIFB与VO不在同一个进程中，则需要先停VO所在的应用进程。

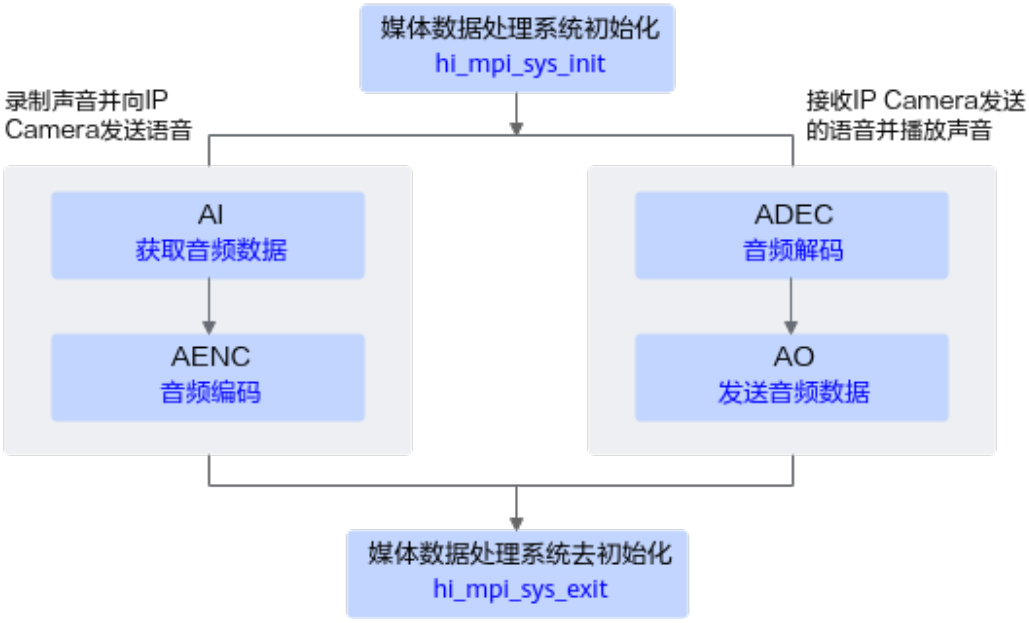
9.8 NVR 场景语音对讲

本节介绍NVR音频业务处理的典型流程、关键接口及注意事项。

NVR，全称Network Video Recorder，即网络视频录像机，是网络视频系统的存储转发部分，NVR与网络摄像机协同工作，完成音频&视频的录像、存储及转发功能，同时，NVR具备本地人机交互界面、视频解码、视频显示及语音对讲功能。

NVR 音频场景说明

本章节描述NVR音频业务（语音对讲功能）的接口调用流程。在语音对讲功能中，包括媒体数据处理系统初始化&去初始化、**接收IP Camera发送的语音并播放声音**、**录制声音并向IP Camera发送语音**，涉及的模块包括公共模块、音频输入模块（AI）,音频编码模块（AENC）,音频输出模块（AO）、音频解码模块（ADEC）。



接收 IP Camera 发送的语音并播放声音

图 9-25 接收 IP Camera 发送的语音并播放声音

接口调用流程说明如下：

1. 调用hi_mpi_adec_create_chn接口**创建音频解码通道**。
2. **启用AO音频输出设备和通道**：
   a. 调用hi_mpi_ao_set_pub_attr接口设置AO设备属性。
   b. 调用hi_mpi_ao_enable接口启动AO设备。
   c. 调用hi_mpi_ao_enable_chn接口启动AO通道
   d. 调用hi_mpi_ao_enable_resample接口启用AO重采样功能。
      由于AO的采样率固定为48kHz，G.711a、G.711u协议的采样率仅支持8kHz，因此需启用重采样功能；而48kHz在AAC协议采样率支持的范围内，因此使用AAC协议时，在AO时无需重采样。

3. 调用hi_mpi_sys_bind接口**绑定ADEC与AO**。

ADEC设备ID	ADEC通道号	AO设备ID	AO通道号
0	0	2	0
0	1	3	0

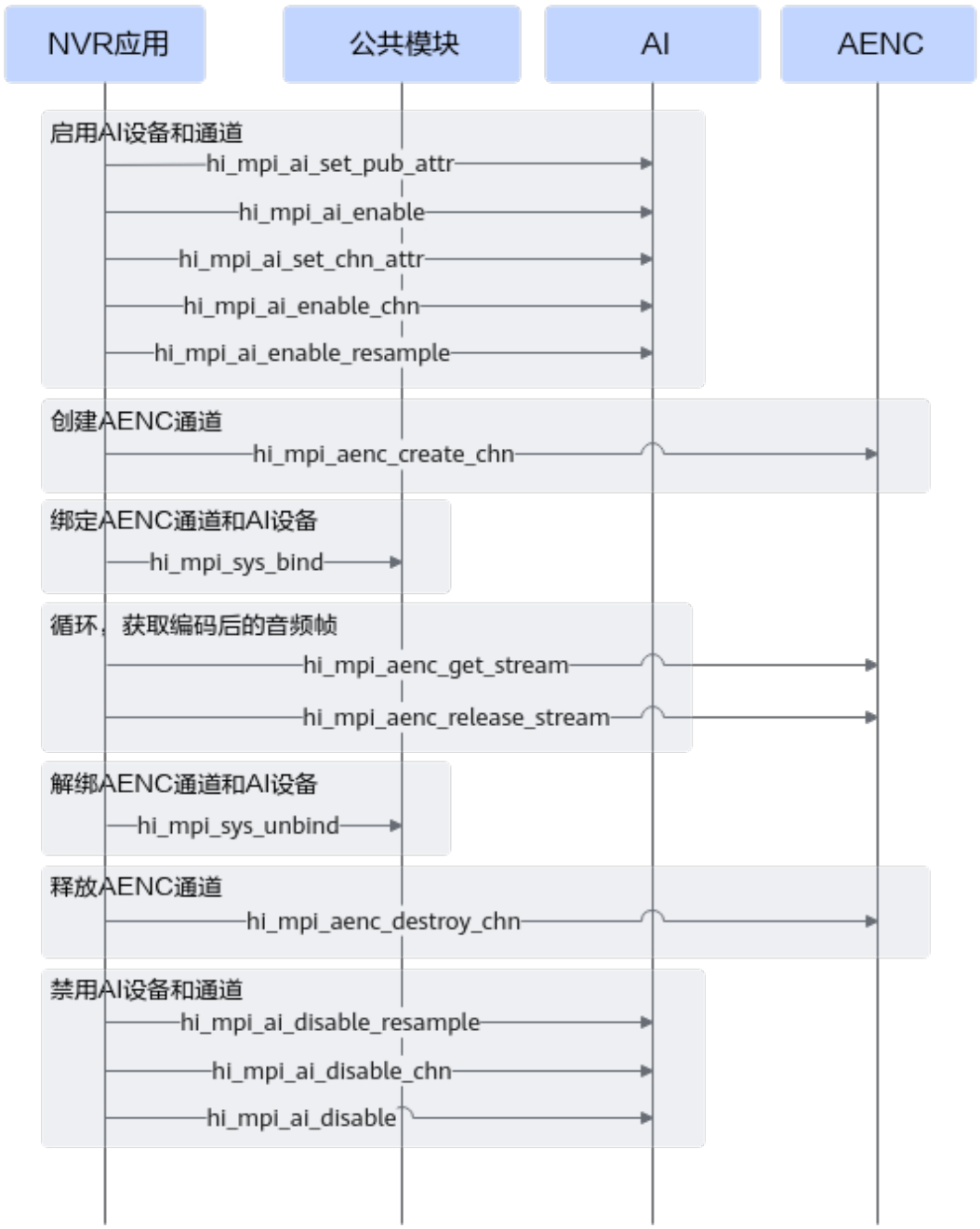
4. 循环调用hi_mpi_adec_send_stream接口将每一帧待解码音频数据发送给解码器进行**解码**。

   解码后的音频数据，根据**3**中的绑定关系，被自动发送到对应的AO设备，用于音频播放。

5. 音频播放完成后，在退出流程中，先调用hi_mpi_sys_unbind接口**解绑ADEC与AO**，再依次调用hi_mpi_ao_disable_resample接口禁用AO重采样功能、调用hi_mpi_ao_disable_chn接口禁用AO通道、调用hi_mpi_ao_disable接口禁用AO设备，最后调用hi_mpi_adec_destroy_chn接口进行销毁ADEC通道。

录制声音并向 IP Camera 发送语音

图 9-26 录制声音并向 IP Camera 发送语音



接口调用流程说明如下：

1. **启用AI音频输入设备和通道**：
   a. 调用hi_mpi_ai_set_pub_attr接口设置AI设备属性。
   b. 调用hi_mpi_ai_enable接口启动AI设备。
   c. 调用hi_mpi_ai_set_chn_attr接口设置AI通道属性。
   d. 调用hi_mpi_ai_enable_chn接口启动AI通道。

e. 调用hi_mpi_ai_enable_resample接口启用AI重采样功能。

由于AI的采样率固定为48kHz，G.711a、G.711u协议的采样率仅支持8kHz，因此需启用重采样功能；而48kHz在AAC协议采样率支持的范围内，因此使用AAC协议时，在AI时无需重采样。

2. 调用hi_mpi_aenc_create_chn接口**创建音频编码通道**。
3. 调用hi_mpi_sys_bind接口**绑定AI与AENC**。

AI设备ID	AI通道号	AENC设备ID	AENC通道号
2	0	0	0

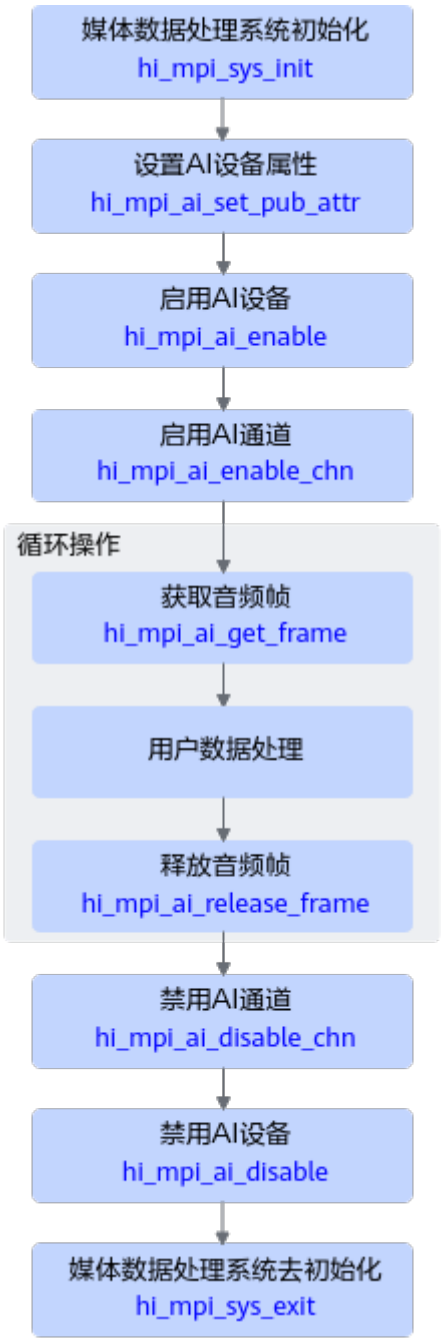
4. 循环调用hi_mpi_aenc_get_stream获取编码数据，编码数据使用完成后，及时调用hi_mpi_aenc_release_stream接口**释放编码数据**。

   经过AI设备获取到的音频数据，根据3中的绑定关系，被自动发送到对应的AENC通道进行编码，用于向IP Camera发送语音。
5. 发送语音完成后，在退出流程中，先调用hi_mpi_sys_unbind接口**解绑AI与AENC**，再调用hi_mpi_aenc_destroy_chn接口进行销毁AENC通道，最后依次调用hi_mpi_ai_disable_resample接口禁用AI重采样功能、调用hi_mpi_ai_disable_chn接口禁用AI通道、调用hi_mpi_ai_disable接口禁用AI设备。

9.9 音频获取&音频播放

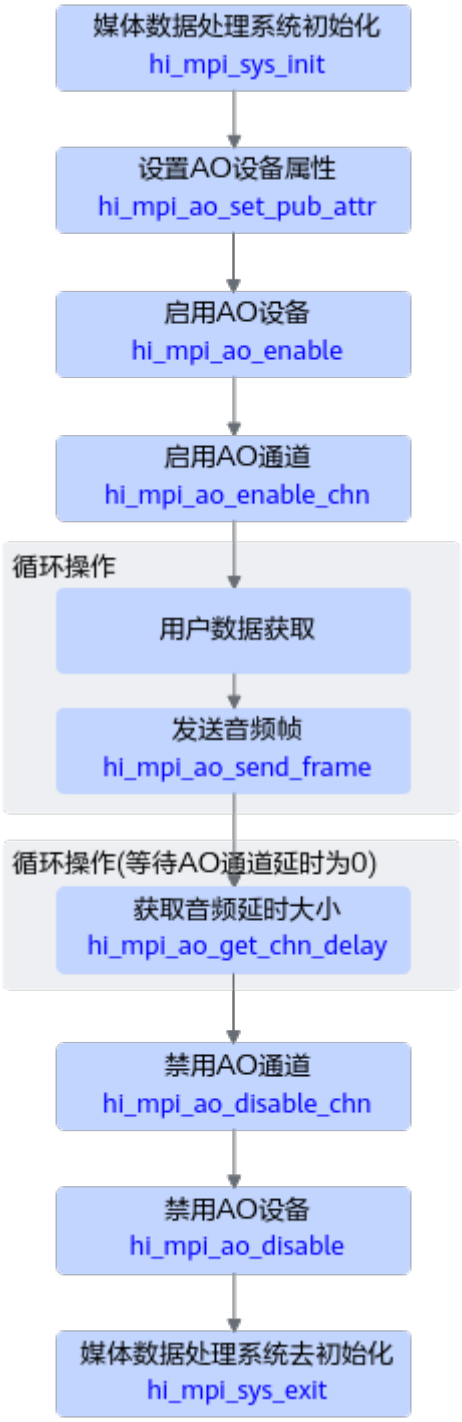
本节介绍音频获取、音频播放功能的接口调用流程及注意事项。

音频获取功能



1. 调用hi_mpi_sys_init接口初始化媒体公共模块。
2. 调用hi_mpi_ai_set_pub_attr接口配置属性。
3. 依次调用hi_mpi_ai_enable接口使能AI设备、调用hi_mpi_ai_enable_chn接口使能AI通道。
4. 调用hi_mpi_ai_get_frame获取录音数据进行处理，之后调用hi_mpi_ai_release_frame释放音频帧，循环往复。
5. AI采集音频结束时，先调用hi_mpi_ai_disable_chn接口禁用通道，然后调用hi_mpi_ai_disable接口禁用AI设备。
6. 调用hi_mpi_sys_exit接口释放媒体公共模块的初始化资源。

音频播放功能



1. 调用hi_mpi_sys_init接口初始化媒体公共模块。
2. 调用hi_mpi_ao_set_pub_attr接口配置属性。
3. 依次调用hi_mpi_ao_enable接口使能AO设备、调用hi_mpi_ao_enable_chn接口使能AO通道。
4. 周期性的获取数据，并调用hi_mpi_ao_send_frame接口进行播音。

5. AO播放音频结束时，先调用hi_mpi_ao_get_chn_delay接口获取AO通道中当前音频延时大小，延时为0后再调用hi_mpi_ao_disable_chn接口禁用通道，然后调用hi_mpi_ao_disable接口禁用AO设备。
6. 调用hi_mpi_sys_exit接口释放媒体公共模块的初始化资源。

9.10 精度提升建议

9.10.1 JPEGD+VPC+模型推理精度提升建议（Atlas 200/300/500 推理产品）

图片解码、图片抠图/缩放以及模型推理功能串联使用时，可能由于接口调用或配置导致推理精度存在偏差，本节基于该场景给出一些建议。

问题描述

JPEGD+VPC+模型推理串联使用，由于宽/高的对齐、输出图片格式等配置问题，可能会导致JPEGD与VPC之间、或VPC与模型推理之间的衔接存在偏差，进而影响整网的推理精度。

精度提升建议

关于JPEGD+VPC+模型推理多个功能串联使用时的一些问题及精度提升建议如下：

1. **JPEGD+VPC串联使用时**：

   由于JPEGD解码后的输出图片的宽stride*高stride有128*16对齐的约束，因此解码后的输出图片的宽、高有一些补边的无效数据，在调用VPC的缩放接口acldvppVpcResizeAsync时，需先调用acldvppSetPicDescWidth和acldvppSetPicDescHeight接口正确设置输入图片的原图宽高，VPC内部会根据原图宽高自行抠图，然后再执行缩放，实现去除无效数据对图像精度的影响。

   为保证模型推理的精度，建议参见**图9-27**、**图9-28**中的正例来编写代码逻辑。

   通过**图9-27**中的反例也可以看出，jpegd解码后，直接将对齐后的宽高作为原图宽高送入VPC，导致模型推理的输入图片存在无效数据，最终可能影响精度。

   JPEGD和VPC的接口调用流程及详细介绍请分别参见**JPEGD接口调用流程**、**VPC接口调用流程**。

2. **使用VPC时**：
   - VPC的抠图、缩放两个功能可以通过acldvppVpcCropAsync或acldvppVpcBatchCropAsync接口来完成，该接口的输出图片的宽stride、高stride必须满足16*2对齐，否则接口返回报错。

     AscendCL当前提供了acldvppVpcCropAsync接口进行抠图、acldvppVpcResizeAsync接口进行缩放，抠图和缩放串联使用时，可以直接用acldvppVpcCropAsync接口，性能更优。
   - VPC的抠图、缩放、贴图三个功能可以通过acldvppVpcCropAndPasteAsync或acldvppVpcBatchCropAndPasteAsync接口来完成，该接口的输出图片的宽stride、高stride必须满足16*2对齐，否则接口返回报错。
   - VPC缩放+贴图串联使用时，如果缩放后的图片宽不是16对齐，在贴图时，vpc会增加无效数据（见**图9-28**中的反例），使其16对齐，为防止无效数据对后续的推理精度有影响，此处建议用户将宽*高按照16*2对齐的要求进行缩放，见**图9-28**中的正例。vpc缩放时，如果完全按vpc等比例缩放，应该输出

238*416分辨率的图片，不满足16对齐，存在无效数据，为了使无效数据不影响精度，建议将图片缩放至240*416分辨率。

- VPC贴图时，贴图与输出图片的左边界距离必须满足16对齐，见**图9-28**中的正例。检测网络中、等比例缩放场景下，用户在实际使用时，如果贴图与输出图片的左边界距离d3满足16对齐后，可能导致贴图区域不在输出图片的中心位置，这时需注意，检测框与贴图区域左边界的距离d1=检测框与输出图片左边界的距离d2-d3。

  例如，**图9-28**中，在最后的vpc贴图操作中，输出图片的分辨率为416*416，贴图区域的宽是240*416，贴图相对输出图片的左偏移=（416-240）/2=88，但88不是16对齐的，如果想继续贴图，得确保贴图相对输出图片的左偏移为16对齐，例如96。这时，计算检测框与贴图区域左边界的距离d1=d2-96，而不是d2-88。

3. **模型推理时：**

   如果需要硬件AIPP进行色域转换，AIPP色域转换配置中的源图片格式要与VPC的输出图片格式保持一致，如果不一致，例如，VPC的输出格式是yuv420sp，色域转换的配置是yvu420sp（源图片格式）-->rgb888（目标图片格式），uv分量的顺序不同也会影响最后模型推理的精度。

   模型推理的接口调用流程及详细介绍请参见**8 模型推理**。

   色域转换的配置请参见《ATC工具使用指南》。

说明

用户训练模型后，如果在昇腾上验证精度是否达标，建议模型推理前的数据预处理建议与训练模型时的数据预处理过程保持一致。可参考**Modelzoo仓**中样例代码。

典型案例

图 9-27 典型分类网络示例

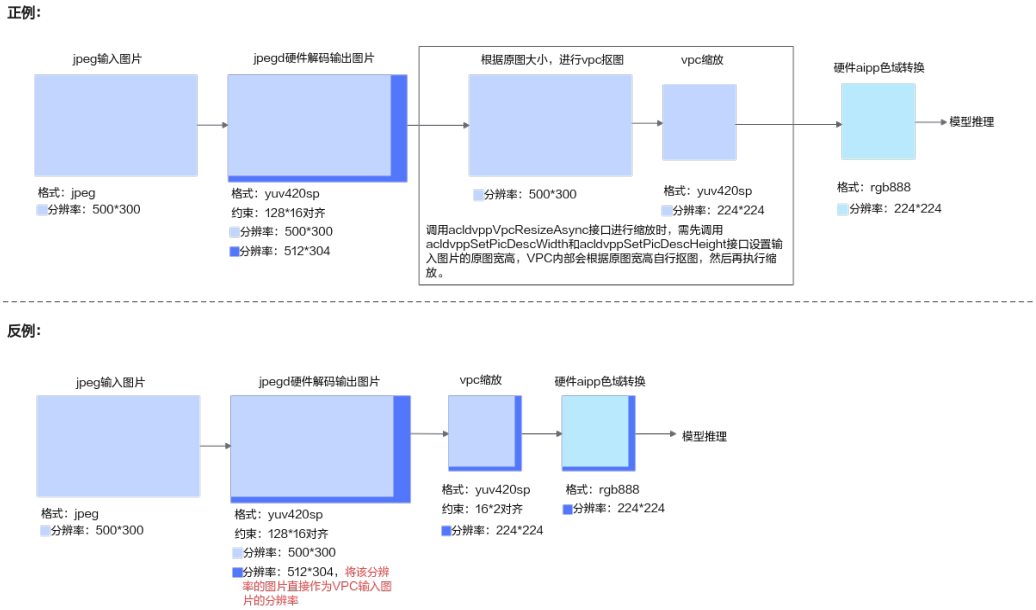
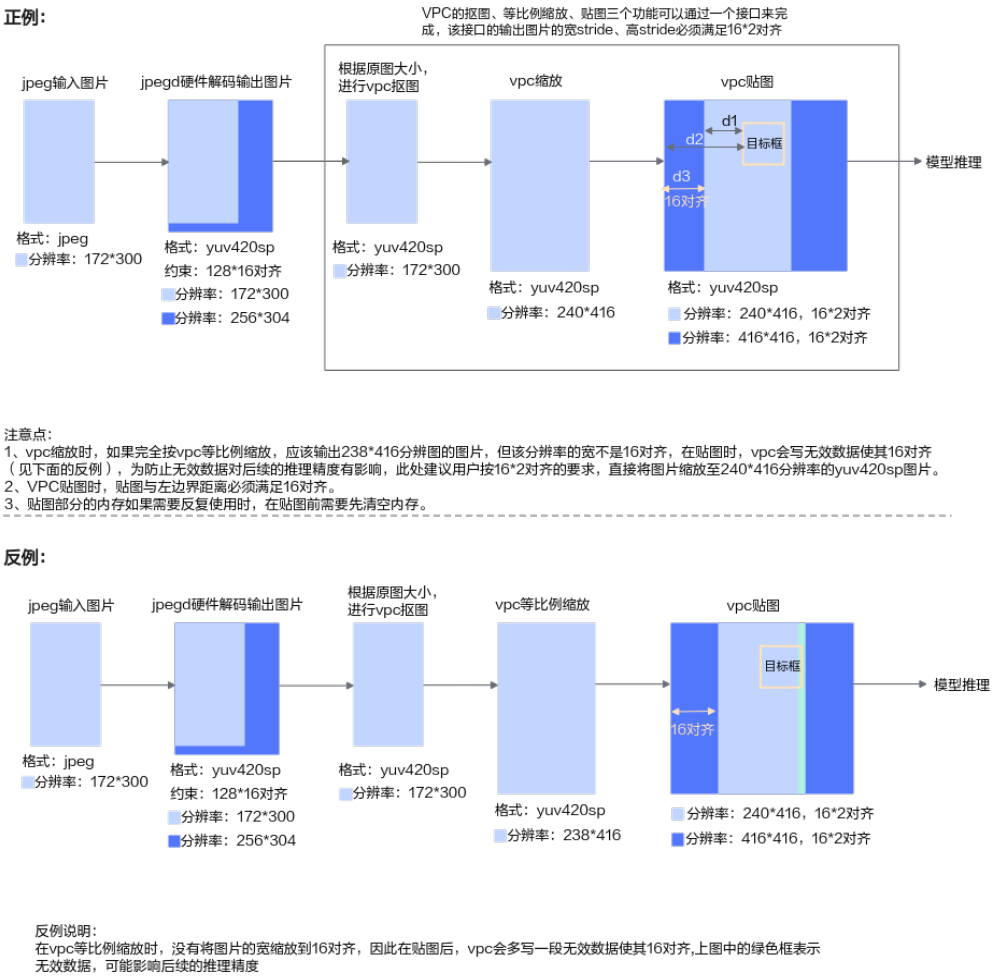


图 9-28 典型检测网络示例



示例代码

- 典型分类网络的示例代码请单击Link获取。
- 典型检测网络的示例代码请单击Link获取。

9.11 高性能编程建议

9.11.1 使用媒体数据处理 V1 版本接口

9.11.1.1 采用 VPC 多功能组合接口，减少系统调度压力，性能更优

背景说明

在对图像进行抠图、缩放、贴图、填充等处理时，AscendCL媒体数据处理部分提供了以下实现功能的接口：

- 一个接口只做一次操作（即单功能接口），例如acldvppVpcCropAsync、acldvppVpcResizeAsync、acldvppVpcMakeBorderAsync接口

该方式下，如果想实现多个功能，例如抠图+缩放+填充，您需要调用以上3个接口。

- 一个接口做多个操作（即多功能组合接口），例如：acldvppVpcBatchCropResizePasteAsync、acldvppVpcBatchCropResizeMakeBorderAsync接口

  该方式下，如果想实现多个功能，例如抠图+缩放+填充，您仅需要调用1个接口acldvppVpcBatchCropResizeMakeBorderAsync。

单功能接口与多功能组合接口的对应关系如下。

单功能接口	多功能组合接口
• acldvppVpcCropAsync（抠图） • acldvppVpcResizeAsync（缩放）	• acldvppVpcCropResizeAsync（抠图缩放）或acldvppVpcBatchCropResizeAsync（批量抠图缩放） • acldvppVpcCropAndPasteAsync（抠图贴图）或acldvppVpcBatchCropAndPasteAsync（批量抠图贴图） • acldvppVpcCropResizePasteAsync（抠图缩放贴图）、acldvppVpcBatchCropResizePasteAsync（批量抠图缩放贴图）
• acldvppVpcCropAsync（抠图） • acldvppVpcResizeAsync（缩放） • acldvppVpcMakeBorderAsync（填充）	acldvppVpcBatchCropResizeMakeBorderAsync（批量抠图缩放填充）

基本原理

一个接口内部会有多次Host和Device的任务交互，每次交互有时延，若对于抠图、缩放等多个功能，调用多次接口，Host和Device的任务交互次数就会增加，时延自然也会随之增加。

采用多个功能组合接口，调用一个接口完成多个功能，虽然是多个功能，但对于Device来说都是一次处理（一个多功能组合接口和一个单功能接口的硬件执行时间相同），相对调用多个单功能接口，能够减少Host和Device的调度次数，减少Device的处理次数，对调度和性能有较多的提升，在性能优化时可以考虑。

使用示例

此处以批量抠图、缩放为例说明如何调用多功能组合接口acldvppVpcBatchCropResizeAsync，完整代码请单击**Link**获取。

9.11.1.2 采用 VPC 批处理接口，降低时延，性能更优

背景说明

在对图像进行抠图、缩放等处理时，AscendCL媒体数据处理部分提供了以下两类接口：

- 一次处理一张图片，例如acldvppVpcCropAsync接口
  该方式下，如果存在多张输入图片，一般都采用for循环的方式，针对每张图片，都调用一次acldvppVpcCropAsync接口。
- 一次处理多张图片（即批处理接口），例如acldvppVpcBatchCropAsync接口
  该方式，如果存在多张输入图片，只需调用一次acldvppVpcBatchCropAsync接口。

以上两类接口的对应关系表如下。

单张图片处理接口	批量图片处理接口
acldvppVpcCropAsync（抠图）	acldvppVpcBatchCropAsync（批量抠图）
acldvppVpcCropResizeAsync（抠图缩放）	acldvppVpcBatchCropResizeAsync（批量抠图缩放）
acldvppVpcCropAndPasteAsync（抠图贴图）	acldvppVpcBatchCropAndPasteAsync（批量抠图贴图）
acldvppVpcCropResizePasteAsync（抠图缩放贴图）	acldvppVpcBatchCropResizePasteAsync（批量抠图缩放贴图）
-	acldvppVpcBatchCropResizeMakeBorderAsync（批量抠图缩放填充）

基本原理

昇腾AI处理器内置图像处理单元DVPP（Digital Video Pre-Processing），在DVPP中，有多个VPC（Vision Preprocessing Core）模块，处理图片的抠图、缩放、格式转换等任务。

在调用批处理接口时，批量任务会被均分到多个VPC模块、并行处理，批量接口的处理时延会降低，性能提升。

使用示例

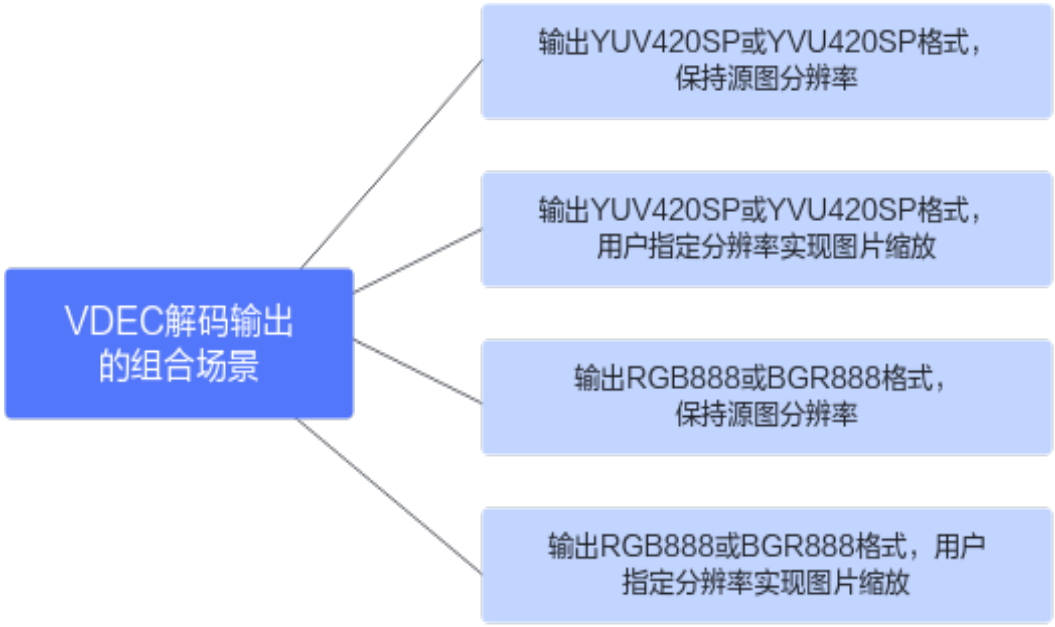
此处以批量抠图、缩放为例说明如何调用批处理接口acldvppVpcBatchCropResizeAsync，完整代码请单击**Link**获取。

9.11.1.3 合理选择 VDEC 视频解码输出格式和分辨率，性能更优

背景说明

如果用户需要借助DVPP进行视频解码，想要得到RGB格式图片，在部分昇腾AI处理器上，当前视频解码接口aclvdecSendFrame支持输出YUV420SP格式或RGB888格式，且支持在解码时对图片进行缩放，所以从提升性能的角度，可以优化代码逻辑，直接调用视频解码接口aclvdecSendFrame输出RGB888格式。

图 9-29 VDEC 解码输出的组合场景



基本原理

部分昇腾AI处理器上视频解码接口aclvdecSendFrame支持输出YUV420SP格式或RGB888格式（请参见功能及约束说明），可设置接口参数输出不同的格式，省去调用acldvppVpcConvertColorAsync进行格式转换的步骤，减少接口调用。

若视频码流分辨率与模型输入图片的分辨率不一致，需要对解码后的图片进行缩放处理，也可以在视频解码接口aclvdecSendFrame中设置输出图片的分辨率，在解码的同时完成图片的缩放，省去单独调用缩放接口的步骤，减少接口调用。

总结下来，可以在视频解码接口aclvdecSendFrame中完成解码+缩放+色域转换三个功能，减少调用接口的数量，提升性能。

使用示例

视频解码+推理的完整代码请单击**Link**获取。

如果视频解码需要输出RGB888格式，并且同时实现缩放功能，您需要设置aclvdecSendFrame接口的output参数（表示输出图片描述信息）中的图片格式、宽、高、宽Stride、高Stride，例如：

```
//创建输出图片描述信息
acldvppPicDesc picOutputDesc = acldvppCreatePicDesc();

//定义输出图片格式、宽、高、宽Stride、高Stride以及存放输出图片数据的内存
```



```
uint32_t width = 224;
uint32_t height = 224;
uint32_t widthStride = 224 * 3;
uint32_t heightStride = 224;
uint32_t size = widthStride * heightStride;
void *picOutBufferDev = nullptr;
acldvppMalloc(&picOutBufferDev, size);

//设置输出图片格式、宽、高、宽Stride、高Stride以及存放输出图片数据的内存
acldvppSetPicDescData(picOutputDesc, picOutBufferDev);
acldvppSetPicDescSize(picOutputDesc, size);
acldvppSetPicDescFormat(picOutputDesc, PIXEL_FORMAT_RGB_888);
acldvppSetPicDescWidth(picOutputDesc, width);
acldvppSetPicDescHeight(picOutputDesc, height);
acldvppSetPicDescWidthStride(picOutputDesc, widthStride);
acldvppSetPicDescHeightStride(picOutputDesc, heightStride);

//调用解码接口
aclvdecSendFrame(channelDesc, picInputDesc, picOutputDesc, nullptr, nullptr);
```

9.11.1.4 合理使用 VDEC 解码跳帧，减少内存申请，减轻 VPC 压力，性能更优

背景说明

在视频解码+模型推理的场景下，若视频的帧数比较多，且不是每一帧都需要进行推理，对于不需要推理的帧，推荐用户使用aclvdecSendSkippedFrame接口进行解码，不输出解码结果。

基本原理

视频解码是需要连续的数据，解码后的数据需要输出YUV格式，则每帧数据解码后，VDEC内部还需要通过VPC进行解压缩、缩放、格式转换的流程。

如果不想获取某一帧的解码结果， 可以调用aclvdecSendSkippedFrame接口，不需要申请输出内存，且VDEC内部也不通过VPC模块进行解压缩、缩放、格式转换的流程。减少内存申请，也减轻了VPC的处理压力，达到提升性能的目标。

使用说明

请参见aclvdecSendSkippedFrame处的说明。

9.11.1.5 VPC 处理时合理选择输出格式，降低内存申请，性能更优

背景说明

在Atlas 推理系列产品（Ascend 310P处理器）上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

在Atlas 200/500 A2推理产品上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

在Atlas A2训练系列产品上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

基本原理

在Atlas 推理系列产品（Ascend 310P处理器）上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

在Atlas 200/500 A2推理产品上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

在Atlas A2训练系列产品上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

使用说明

参见15.4.2.3 优化建议中的“YUV400格式图像处理”功能说明。

关于VPC功能的示例代码，请参见9.4.1 VPC图像处理典型功能。

9.11.2 使用媒体数据处理 V2 版本接口

9.11.2.1 采用 VPC 多功能组合接口，减少系统调度压力，性能更优

背景说明

在对图像进行抠图、缩放、贴图、填充等处理时，AscendCL媒体数据处理部分提供了以下实现功能的接口：

- 一个接口只做一次操作（即单功能接口），例如hi_mpi_vpc_crop、hi_mpi_vpc_resize、hi_mpi_vpc_copy_make_border接口

  该方式下，如果想实现多个功能，例如抠图+缩放+填充，您需要调用以上3个接口。
- 一个接口做多个操作（即多功能组合接口），例如：hi_mpi_vpc_batch_crop_resize_paste、hi_mpi_vpc_batch_crop_resize_make_border接口

  该方式下，如果想实现多个功能，例如抠图+缩放+填充，您仅需要调用1个接口hi_mpi_vpc_batch_crop_resize_make_border。

单功能接口与多功能组合接口的对应关系如下。

单功能接口	多功能组合接口
• hi_mpi_vpc_crop（抠图） • hi_mpi_vpc_resize（缩放）	• hi_mpi_vpc_crop_resize（抠图缩放） • hi_mpi_vpc_crop_resize_paste（抠图缩放贴图）、hi_mpi_vpc_batch_crop_resize_paste（批量抠图缩放贴图）
• hi_mpi_vpc_crop（抠图） • hi_mpi_vpc_resize（缩放） • hi_mpi_vpc_copy_make_border（填充）	hi_mpi_vpc_crop_resize_make_border（抠图缩放填充）或hi_mpi_vpc_batch_crop_resize_make_border（批量抠图缩放填充）

基本原理

一个接口内部会有多次Host和Device的任务交互，每次交互有时延，若对于抠图、缩放等多个功能，调用多次接口，Host和Device的任务交互次数就会增加，时延自然也会随之增加。

采用多个功能组合接口，调用一个接口完成多个功能，虽然是多个功能，但对于Device来说都是一次处理（一个多功能组合接口和一个单功能接口的硬件执行时间相同），相对调用多个单功能接口，能够减少Host和Device的调度次数，减少Device的处理次数，对调度和性能有较多的提升，在性能优化时可以考虑。

使用示例

此处以批量抠图、缩放、填充为例说明如何调用多功能组合接口hi_mpi_vpc_batch_crop_resize_make_border，完整代码请单击**Link**获取。

9.11.2.2 采用 VPC 批处理接口，降低时延，性能更优

背景说明

在对图像进行抠图、缩放等处理时，AscendCL媒体数据处理部分提供了以下两类接口：

- 一次处理一张图片，例如hi_mpi_vpc_crop_resize_make_border接口

  该方式下，如果存在多张输入图片，一般都采用for循环的方式，针对每张图片，都调用一次hi_mpi_vpc_crop_resize_make_border接口。
- 一次处理多张图片（即批处理接口），例如hi_mpi_vpc_batch_crop_resize_make_border接口

  该方式，如果存在多张输入图片，只需调用一次hi_mpi_vpc_batch_crop_resize_make_border接口。

以上两类接口的对应关系表如下。

单张图片处理接口	批量图片处理接口
hi_mpi_vpc_crop_resize_paste（抠图缩放贴图）	hi_mpi_vpc_batch_crop_resize_paste（批量抠图缩放贴图）
hi_mpi_vpc_crop_resize_make_border（抠图缩放填充）	hi_mpi_vpc_batch_crop_resize_make_border（批量抠图缩放填充）

基本原理

昇腾AI处理器内置图像处理单元DVPP（Digital Video Pre-Processing），在DVPP中，有多个VPC（Vision Preprocessing Core）模块，处理图片的抠图、缩放、格式转换等任务。

在调用批处理接口时，批量任务会被均分到多个VPC模块、并行处理，批量接口的处理时延会降低，性能提升。

使用示例

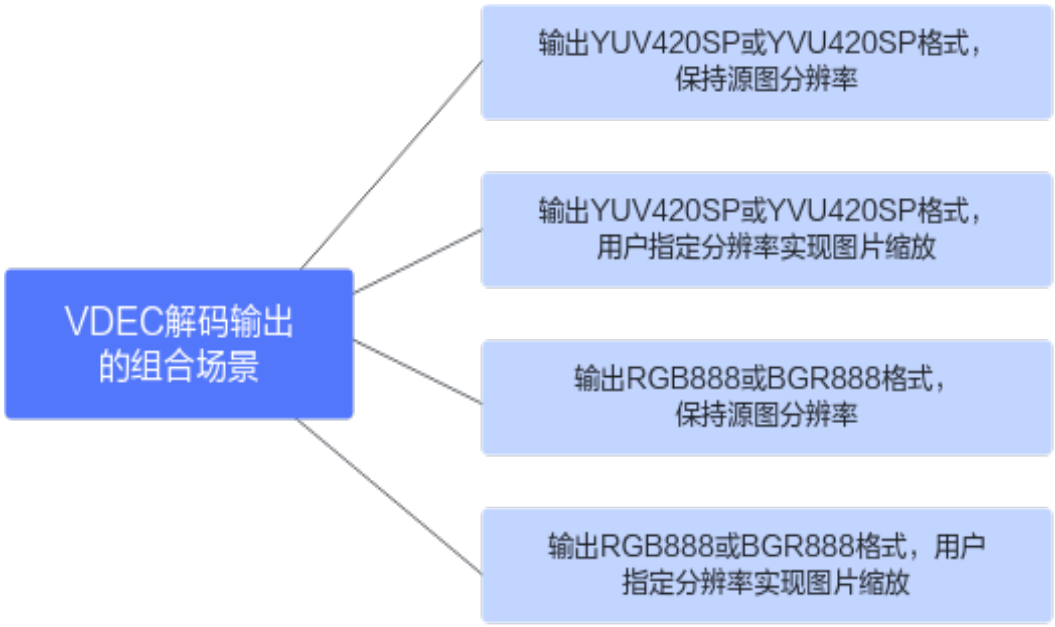
此处以批量抠图、缩放、填充为例说明如何调用多功能组合接口hi_mpi_vpc_batch_crop_resize_make_border，完整代码请单击Link获取。

9.11.2.3 合理选择 VDEC 视频解码输出格式和分辨率，性能更优

背景说明

如果用户需要借助DVPP进行视频解码，想要得到RGB格式图片，在部分昇腾AI处理器上当前视频解码接口hi_mpi_vdec_send_stream支持输出YUV420SP格式或RGB888格式，且支持在解码时对图片进行缩放，所以从提升性能的角度，可以优化代码逻辑，直接调用视频解码接口hi_mpi_vdec_send_stream输出RGB888格式。

图 9-30 VDEC 解码输出的组合场景



基本原理

部分昇腾AI处理器上视频解码接口hi_mpi_vdec_send_stream支持输出YUV420SP格式或RGB888格式（参见VDEC功能及约束说明），可设置接口参数输出不同的格式，省去调用hi_mpi_vpc_convert_color进行格式转换的步骤，减少接口调用。

若视频码流分辨率与模型输入图片的分辨率不一致，需要对解码后的图片进行缩放处理，也可以在视频解码接口hi_mpi_vdec_send_stream中设置输出图片的分辨率，在解码的同时完成图片的缩放，省去单独调用缩放接口的步骤，减少接口调用。

总结下来，可以在视频解码接口hi_mpi_vdec_send_stream中完成解码+缩放+色域转换三个功能，减少调用接口的数量，提升性能。

使用示例

视频解码的示例代码请单击**Link**获取。

9.11.2.4 合理使用 VDEC 解码跳帧，减少内存申请，减轻 VPC 压力，性能更优

背景说明

在视频解码+模型推理的场景下，若视频的帧数比较多，且不是每一帧都需要进行推理，对于不需要推理的帧，推荐用户在调用hi_mpi_vdec_send_stream接口解码时设置当前帧输出不显示，不输出解码结果。

基本原理

视频解码是需要连续的数据，解码后的数据需要输出YUV格式，则每帧数据解码后，VDEC内部还需要通过VPC进行解压缩、缩放、格式转换的流程。

如果不想获取某一帧的解码结果， 在调用hi_mpi_vdec_send_stream接口解码时设置当前帧输出不显示，这样，就不需要申请输出内存，且VDEC内部也不通过VPC模块进行解压缩、缩放、格式转换的流程。减少内存申请，也减轻了VPC的处理压力，达到提升性能的目标。

使用说明

请参见hi_mpi_vdec_send_stream接口的hi_vdec_stream结构体内的need_display参数说明。

9.11.2.5 VPC 处理时合理选择输出格式，降低内存申请，性能更优

在Atlas 推理系列产品（Ascend 310P处理器）上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

在Atlas 200/500 A2推理产品上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

在Atlas A2训练系列产品上，VPC图像处理功能支持输出YUV400格式（灰度图像），如果模型推理的输入图像是灰度图像，就直接使用VPC功能，无需再使用AIPP色域转换功能。

基本原理

在Atlas 推理系列产品（Ascend 310P处理器）上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

在Atlas 200/500 A2推理产品上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

在Atlas A2训练系列产品上，直接使用VPC功能输出YUV400格式（灰度图像），省去使用AIPP色域转换功能，降低AI Core单元的负载，从硬件上提升性能。

使用说明

关于VPC功能的示例代码，请参见**9.5.1 VPC图片处理典型功能**。

9.11.2.6 合理设置队列深度，减少硬件资源浪费，提升性能

背景说明

在Atlas 推理系列产品（Ascend 310P处理器）上，可使用媒体数据处理V2版本接口，配置VPC队列深度，队列深度范围[10，350]之间。用户业务下发任务时，若VPC队列满后，会反压阻塞任务下发，影响性能。

基本原理

适当加大VPC队列深度，可以缓解用户业务下发任务与VPC内部任务处理相互之间性能波动的影响。

- 如果用户短时间内下发任务多于VPC通道处理速度，则多出来的任务会缓存在队列中，不会阻塞用户继续下发任务；
- 如果短时间内用户下发任务较少，则VPC可以处理队列中缓存的任务，不会造成空闲，导致硬件资源浪费。

用户可基于业务下发任务的速度与VPC单通道处理的性能适当调整队列任务深度：

- 用户业务创建一个通道，起多个线程往同一通道下发任务，且下发任务数超过上面单通道参考性能，则建议用户将队列深度设大；
- 用户业务创建一个通道，只有一个线程往该通道下发任务，且为下发一次任务，获取一次结果，串行执行。该情况通道任务不会堆积，不必设大，保持默认即可。

说明

VPC单通道处理的性能，请参见性能指标说明。

使用说明

调用hi_mpi_vpc_create_chn接口或hi_mpi_vpc_sys_create_chn接口创建VPC通道时，通过hi_vpc_chn_attr结构体内的attr参数设置队列深度。

关于VPC队列深度设置示例代码，参见**sample_vpc.cpp**中queue_len参数。

10 更多特性

10.1 内存二次分配管理

用户通过AscendCL提供的内存管理接口申请内存后，若需二次分配管理，需关注各内存接口的约束，防止出现内存越界。

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，使用如下接口从内存池申请对应内存，由于接口对申请的内存地址、大小有约束，在内存池管理时，需要关注，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
aclrtMemcpyAsync	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。

接口	用途	输入内存/输出内存
aclrtMalloc	在Device上分配size大小的线性内存，并通过*devPtr返回已分配内存的指针。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。	• 若用户需申请大块内存并自行划分、管理内存时，建议使用aclrtMallocAlign32接口，该接口相比aclrtMalloc接口，只会对用户申请的size向上对齐成32字节整数倍，不会再多加32字节。 不管是aclrtMalloc接口，还是aclrtMallocAlign32接口，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+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。
acldvppMalloc	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字节对齐），同步接口。 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。

接口	用途	输入内存/输出内存
hi_mpi_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。
aclrtMallocHost	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+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。
aclrtMallocCached	在Device上申请size大小的线性内存，通过*devPtr返回已分配内存的指针，该接口在任何场景下申请的内存都是支持cache缓存。	其它约束与aclrtMalloc接口相同。

在计算机视觉领域，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的，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**9.4 DVPP图像/视频处理（媒体数据处理V1）**。

图 10-1 VDEC 场景

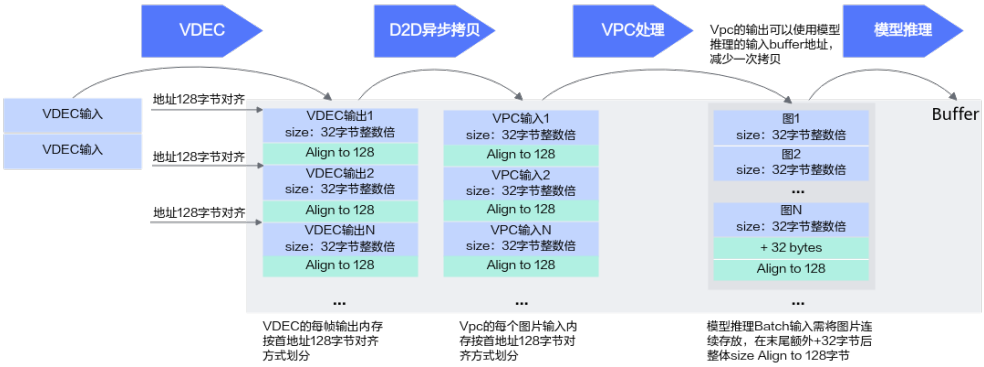
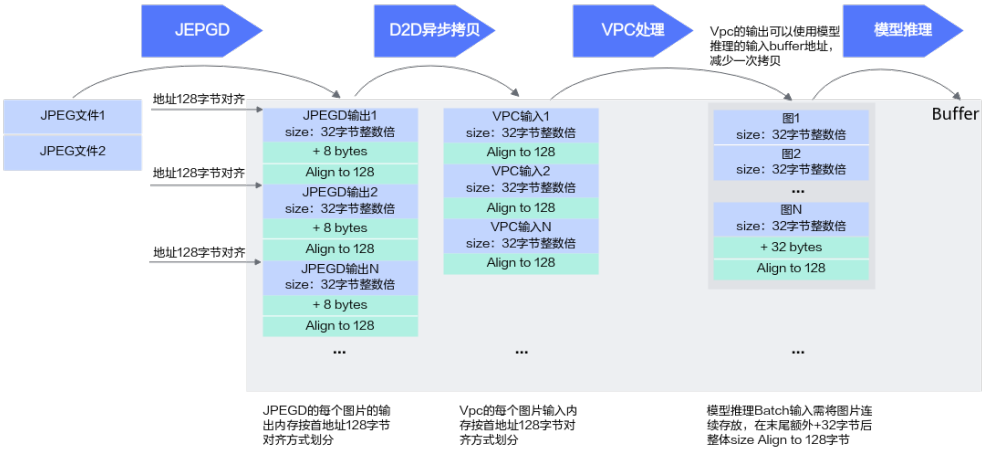


图 10-2 JPEGD 场景



10.2 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aclrtSetExceptionInfoCallback 。

   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aclrtGetDeviceIdFromExceptionInfo、aclrtGetStreamIdFromExceptionInfo、aclrtGetTaskIdFromExceptionInfo接口分别获取Device ID、Stream ID、Task ID。
b. 在异常回调函数fn内调用aclmdlCreateAndGetOpDesc接口获取算子的描述信息。
c. 在异常回调函数fn内调用aclGetTensorDescByIndex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内调用如下接口获取tensor描述中的数据，进行进一步分析。

   例如，调用aclGetTensorDescAddress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aclGetTensorDescType接口获取tensor描述中的数据类型、调用aclGetTensorDescFormat接口获取tensor描述中的Format、调用aclGetTensorDescNumDims接口获取tensor描述中的Shape维度个数、调用aclGetTensorDescDimV2接口获取Shape中指定维度的大小。

2. 调用aclrtSetExceptionInfoCallback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，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。

本节中的示例重点介绍AI Core异常信息获取的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**，模型加载、准备模型推理的输入/输出数据的接口调用流程、模型执行、模型卸载等请参见**8.3 单Batch&静态Shape输入推理**。

```
// 1.AscendCL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、创建Stream

// 3.模型加载，加载成功后，返回标识模型的modelId

// 4.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

// 5.实现异常回调函数
void callback(aclrtExceptionInfo *exceptionInfo)
{
    deviceId = aclrtGetDeviceIdFromExceptionInfo(exceptionInfo);
    streamId = aclrtGetStreamIdFromExceptionInfo(exceptionInfo);
    taskId = aclrtGetTaskIdFromExceptionInfo(exceptionInfo);

    char opName[256];
    aclTensorDesc *inputDesc = nullptr;
    aclTensorDesc *outputDesc = nullptr;
    size_t inputCnt = 0;
    size_t outputCnt = 0;
    // 用户可以将获取的算子信息写入到文件，或者另起线程，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在屏幕上显示
    aclmdlCreateAndGetOpDesc(deviceId, streamId, taskId, opName, 256,  &inputDesc, &inputCnt, &outputDesc, &outputCnt);
    // 可以调用AscendCL 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 (size_t i = 0; i < inputCnt; ++i) {
        const aclTensorDesc *desc = aclGetTensorDescByIndex(inputDesc, i);
        aclGetTensorDescAddress(desc);
        aclGetTensorDescFormat(desc);
    }
    for (size_t i = 0; i < outputCnt; ++i) {
```



```
        const aclTensorDesc *desc = aclGetTensorDescByIndex(outputDesc, i);
        aclGetTensorDescAddress(desc);
        aclGetTensorDescFormat(desc);
    }
    aclDestroyTensorDesc(inputDesc);
    aclDestroyTensorDesc(outputDesc);
}

// 6.设置异常回调
aclrtSetExceptionInfoCallback(callback);

// 7.执行模型
ret = aclmdlExecute(modelId, input, output);

// 8.处理模型推理结果

// 9.释放描述模型输入/输出信息、内存等资源，卸载模型

// 10.释放运行管理资源

// 11. AscendCL去初始化

// ......
```

10.3 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 AscendCL API（通过Profiling AscendCL 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**：

  - aclprofInit接口、aclprofStart接口、aclprofStop接口、aclprofFinalize接口配合使用，实现该方式的性能数据采集。该方式可获取AscendCL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aclInit接口，在AscendCL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AscendCL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aclInit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aclInit接口处的说明，不在本章节描述。

- **Profiling AscendCL API for Extension（Profiling AscendCL 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（aclprofStart接口与aclprofStop接口之间）调用Profiling AscendCL 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。

  一个进程内，可以根据需求多次调用这些接口，**接口调用方式如下**：在aclprofStart和aclprofStop接口之间调用aclprofCreateStamp、aclprofPush、aclprofPop、aclprofRangeStart、aclprofRangeStop、aclprofDestroyStamp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。

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”。

- **Profiling AscendCL API for Subscription（订阅算子信息的Profiling AscendCL 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AscendCL的接口获取性能数据。

  接口调用方式：aclprofModelSubscribe接口、aclprofGet*接口、aclprofModelUnSubscribe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 AscendCL 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，不可以直接拷贝编译运行，仅供参考。

本节中的示例重点介绍Profiling性能数据采集的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**，模型加载、准备模型推理的输入/输出数据的接口调用流程、模型执行、模型卸载等请参见**8.3 单Batch&静态Shape输入推理**。

```
// 1.AscendCL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、创建Stream

// 3.profiling初始化
// 设置数据落盘路径
const char *aclProfPath = "...";
aclprofInit(aclProfPath, strlen(aclProfPath));

// 4.进行profiling配置
uint32_t deviceIdList[1] = {0};
// 创建配置结构体
aclprofConfig *config = aclprofCreateConfig(deviceIdList, 1, ACL_AICORE_ARITHMETIC_UTILIZATION,
    nullptr,ACL_PROF_ACL_API | ACL_PROF_TASK_TIME | ACL_PROF_AICORE_METRICS | ACL_PROF_AICPU |
ACL_PROF_L2CACHE | ACL_PROF_HCCL_TRACE | ACL_PROF_MSPROFTX | ACL_PROF_RUNTIME_API);
const char *memFreq = "15";
ret = aclprofSetConfig(ACL_PROF_SYS_HARDWARE_MEM_FREQ, memFreq, strlen(memFreq));
aclprofStart(config);

// 5.模型加载，加载成功后，返回标识模型的modelId

// 6.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

// 7.执行模型
ret = aclmdlExecute(modelId, input, output);

// 8.处理模型推理结果

// 9.释放描述模型输入/输出信息、内存等资源，卸载模型

// 10.关闭profiling配置, 释放配置资源, 释放profiling组件资源
aclprofStop(config);
aclprofDestroyConfig(config);
aclprofFinalize();

// 11.释放运行管理资源

// 12. AscendCL去初始化
// ......
```


Profiling AscendCL 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，不可以直接拷贝编译运行，仅供参考。

Profiling msproftx接口，请参见如下示例中的加粗部分代码。

本节中的示例重点介绍Profiling性能数据采集的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**，模型加载、准备模型推理的输入/输出数据的接口调用流程、模型执行、模型卸载等请参见**8.3 单Batch&静态Shape输入推理**。

示例一（aclprofMark示例）：

```
//1.AscendCL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、创建Stream
stamp = aclprofCreateStamp();
aclprofSetStampTraceMessage(stamp, "aclrtCreateStream_mark", strlen("AscendCL_Init_Mark"));
aclprofMark(stamp);   //标记Create Stream事件
aclprofDestroyStamp(stamp);

//3..Profiling初始化
//设置数据落盘路径
const char *aclProfPath = "...";
aclprofInit(aclProfPath, strlen(aclProfPath));

//4.进行Profiling配置
uint32_t deviceIdList[1] = {0};
//创建配置结构体
aclprofConfig *config = aclprofCreateConfig(deviceIdList, 1, ACL_AICORE_ARITHMETIC_UTILIZATION,
    nullptr,ACL_PROF_ACL_API | ACL_PROF_TASK_TIME);
const char *memFreq = "15";
ret = aclprofSetConfig(ACL_PROF_SYS_HARDWARE_MEM_FREQ, memFreq, strlen(memFreq));
aclprofStart(config);

aclprofStepInfo *stepInfo = aclprofCreateStepInfo();
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_START, stream_);

//5.模型加载，加载成功后，返回标识模型的modelId
stamp = aclprofCreateStamp();
aclprofSetStampTraceMessage(stamp, "model_load_mark", strlen("model_load_mark"));
aclprofMark(stamp);     //标记模型加载事件
aclprofDestroyStamp(stamp);

//6.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

//7.执行模型
ret = aclmdlExecute(modelId, input, output);

//8.处理模型推理结果

//9.释放描述模型输入/输出信息、内存等资源，卸载模型
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_END, stream_);
aclprofDestroyStepInfo(stepInfo);

//10.关闭Profiling配置, 释放配置资源, 释放Profiling组件资源
aclprofStop(config);
aclprofDestroyConfig(config);
aclprofFinalize();

//11.释放运行管理资源

//12. AscendCL去初始化
//.....
```

示例二（aclprofPush/aclprofPop示例，适用于单线程）：


```
//1.AscendCL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、创建Stream

//3..Profiling初始化
//设置数据落盘路径
const char *aclProfPath = "...";
aclprofInit(aclProfPath, strlen(aclProfPath));

//4.进行Profiling配置
uint32_t deviceIdList[1] = {0};
//创建配置结构体
aclprofConfig *config = aclprofCreateConfig(deviceIdList, 1, ACL_AICORE_ARITHMETIC_UTILIZATION,
    nullptr,ACL_PROF_ACL_API | ACL_PROF_TASK_TIME);
const char *memFreq = "15";
ret = aclprofSetConfig(ACL_PROF_SYS_HARDWARE_MEM_FREQ, memFreq, strlen(memFreq));
aclprofStart(config);

aclprofStepInfo *stepInfo = aclprofCreateStepInfo();
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_START, stream_);

//5.模型加载，加载成功后，返回标识模型的modelId

//6.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

//7.执行模型（模型仅在单线程执行）
stamp = aclprofCreateStamp();
aclprofSetStampTraceMessage(stamp, "aclmdlExecute_duration", strlen("aclmdlExecute_duration"));
aclprofPush(stamp);
ret = aclmdlExecute(modelId, input, output);
aclprofPop(stamp);
aclprofDestroyStamp(stamp);

//8.处理模型推理结果

//9.释放描述模型输入/输出信息、内存等资源，卸载模型
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_END, stream_);
aclprofDestroyStepInfo(stepInfo);

//10.关闭Profiling配置, 释放配置资源, 释放Profiling组件资源
aclprofStop(config);
aclprofDestroyConfig(config);
aclprofFinalize();

//11.释放运行管理资源

//12. AscendCL去初始化
//.....
```

示例三（aclprofRangeStart/aclprofRangeStop示例，适用于单线程或跨线程）：

```
//1.AscendCL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、创建Stream

//3..Profiling初始化
//设置数据落盘路径
const char *aclProfPath = "...";
aclprofInit(aclProfPath, strlen(aclProfPath));

//4.进行Profiling配置
uint32_t deviceIdList[1] = {0};
//创建配置结构体
aclprofConfig *config = aclprofCreateConfig(deviceIdList, 1, ACL_AICORE_ARITHMETIC_UTILIZATION,
    nullptr,ACL_PROF_ACL_API | ACL_PROF_TASK_TIME);
const char *memFreq = "15";
ret = aclprofSetConfig(ACL_PROF_SYS_HARDWARE_MEM_FREQ, memFreq, strlen(memFreq));
aclprofStart(config);

aclprofStepInfo *stepInfo = aclprofCreateStepInfo();
```

```
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_START, stream_);

//5.模型加载，加载成功后，返回标识模型的modelId

//6.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

//7.执行模型（模型在跨线程执行）
stamp = aclprofCreateStamp();
aclprofSetStampTraceMessage(stamp, "aclmdlExecute_duration", strlen("aclmdlExecute_duration"));
aclprofRangeStart(stamp, &rangeId);
ret = aclmdlExecute(modelId, input, output);
aclprofRangeStop(rangeId);
aclprofDestroyStamp(stamp);

//8.处理模型推理结果

//9.释放描述模型输入/输出信息、内存等资源，卸载模型
int ret = aclprofGetStepTimestamp(stepInfo, ACL_STEP_END, stream_);
aclprofDestroyStepInfo(stepInfo);

//10.关闭Profiling配置, 释放配置资源, 释放Profiling组件资源
aclprofStop(config);
aclprofDestroyConfig(config);
aclprofFinalize();

//11.释放运行管理资源

//12. AscendCL去初始化
//.....
```

Profiling AscendCL API for Subscription 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。

本节中的示例重点介绍Profiling性能数据采集的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**，模型加载、准备模型推理的输入/输出数据的接口调用流程、模型执行、模型卸载等请参见**8.3 单Batch&静态Shape输入推理**。

```
// 1.AscendCL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、创建Stream

// 3.模型加载，加载成功后，返回标识模型的modelId

// 4.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

// 5.创建管道（UNIX操作系统下需要引用C++标准库头文件unistd.h），用于读取以及写入模型订阅的数据
int subFd[2];
// 读管道指针指向subFd[0]，写管道指针指向subFd[1]
pipe(subFd);

// 6.创建模型订阅的配置并且进行模型订阅
aclprofSubscribeConfig *config = aclprofCreateSubscribeConfig(1, ACL_AICORE_NONE, &subFd[1]);
// 模型订阅需要传入模型的modelId
aclprofModelSubscribe(modelId, config);

// 7.实现管道读取订阅数据的函数
// 7.1 自定义函数，实现从用户内存中读取订阅数据的函数
void getModelInfo(void *data, uint32_t len) {
    uint32_t opNumber = 0;
    uint32_t dataLen = 0;
    // 通过AscendCL接口读取算子信息个数
    aclprofGetOpNum(data, len, &opNumber);
    // 遍历用户内存的算子信息
    for (int32_t i = 0; i < opNumber; i++){
```



```
        // 获取算子的模型id
        uint32_t modelId = aclprofGetModelId(data,len, i);
        // 获取算子的类型名称长度
        size_t opTypeLen = 0;
        aclprofGetOpTypeLen(data, len, i, &opTypeLen);
        // 获取算子的类型名称
        char opType[opTypeLen];
        aclprofGetOpType(data, len, i, opType, opTypeLen);
        // 获取算子的详细名称长度
        size_t opNameLen = 0;
        aclprofGetOpNameLen(data, len, i, &opNameLen);
        // 获取算子的详细名称
        char opName[opNameLen];
        aclprofGetOpName(data, len, i, opName, opNameLen);
        // 获取算子的执行开始时间
        uint64_t opStart = aclprofGetOpStart(data, len, i);
        // 获取算子的执行结束时间
        uint64_t opEnd = aclprofGetOpEnd(data, len, i);
        uint64_t opDuration = aclprofGetOpDuration(data, len, i);
    }
}

// 7.2 自定义函数，实现从管道中读取数据到用户内存的函数
void *profDataRead(void *fd) {
    // 设置每次从管道中读取的算子信息个数
    uint64_t N = 10;
    // 获取单位算子信息的大小（Byte）
    uint64_t bufferSize = 0;
    aclprofGetOpDescSize(&bufferSize);
    // 计算存储算子信息的内存的大小，并且申请内存
    uint64_t readbufLen = bufferSize * N;
    char *readbuf = new char[readbufLen];
    // 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dataLen可能小于bufferSize * N，如果管道中没有数据，默认会阻塞直到读取到数据为止
    auto dataLen = read(*(int*)fd, readbuf, readbufLen);
    // 读取数据到readbuf成功
    while (dataLen > 0) {
        // 调用5.1实现的函数解析内存中的数据
        getModelInfo(readbuf, dataLen);
        memset(readbuf, 0, bufferSize);
        dataLen = read(*(int*)fd, readbuf, readbufLen);
    }
    delete []readbuf;
}

// 8. 启动线程读取管道数据并解析
pthread_t subTid = 0;
pthread_create(&subTid, NULL, profDataRead, &subFd[0]);

// 9.执行模型
ret = aclmdlExecute(modelId, input, output);

// 10.处理模型推理结果

// 11.释放描述模型输入/输出信息、内存等资源，卸载模型

// 12.取消订阅，释放订阅相关资源
aclprofModelUnSubscribe(modelId);
pthread_join(subTid, NULL);
// 关闭读管道指针
close(subFd[0]);
// 释放config指针
aclprofDestroySubscribeConfig(config);

// 13.释放运行管理资源

// 14. AscendCL去初始化
// ......
```


10.4 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aclInit接口初始化AscendCL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

10.5 特征向量检索

须知

Atlas 200/300/500 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。

Atlas 200/500 A2推理产品上，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aclInit接口进行AscendCL初始化、运行管理资源申请，调用aclfvCreateInitPara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aclfvCreateFeatureInfo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aclfvRepoAdd添加底库。
- **底库搜索**：调用aclfvSearch接口来实现检索。
- **精准修改或删除底库特征**：调用aclfvDel和aclfvModify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aclfvDestroyInitPara接口销毁aclfvInitPara类型的数据、调用aclfvRelease接口特征检索模块去初始化，释放内存空间。

示例代码

本节中的示例重点介绍特征向量检索的代码逻辑，AscendCL初始化和去初始化请参见**5 AscendCL初始化**，运行管理资源申请与释放请参见**6.1 运行管理资源申请与释放**。

示例代码如下，可以从**Link**中查看完整样例代码。

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，不可以直接拷贝编译运行，仅供参考。

```
// 1.AscendCL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
// 设置默认的运行模式为HOST
aclrtRunMode runMode = ACL_HOST;

// 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
size_t fsNum = 100000;
fvInitPara = aclfvCreateInitPara(fsNum);

// 3.2 指定特征向量检索的初始化参数
ret = aclfvInit(fvInitPara);

// 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
uint32_t offset = 0;
uint32_t featureCount = 1000;
uint32_t featureLen = 36;

// 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
void *featureData = BaseShortFeaAlloc(1000, static_cast<size_t>(featureCount), 0);
std::shared_ptr<void> feaBufPtr(featureData, [](void *p){(void)aclrtFreeHost(p);});
void *inputData = featureData;
std::shared_ptr<void> inputDataPtr = nullptr;

// 如果运行模式为ACL_HOST，则需要申请内存，再通过aclrtMemcpy接口将Host的随机特征数据传输到Device；否则直接将随机特征数据读入Device内存
if (aclrtGetRunMode(&runMode) == ACL_HOST) {
    // 为inputData申请内存
    ret = aclrtMalloc(&inputData, featureLen * featureCount,
                      ACL_MEM_MALLOC_NORMAL_ONLY);
    // 将随机特征数据读入到Device内存中
```



```
    inputDataPtr.reset(inputData, [](void *p) {(void)aclrtFree(p);});
    // 将featureData从Host侧拷贝到Device侧
    ret = aclrtMemcpy(inputData,
                      featureLen * featureCount,
                      featureData,
                      featureLen * featureCount,
                      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
auto featureInfo = aclfvCreateFeatureInfo(id0, id1, offset, featureLen, featureCount,
                      reinterpret_cast<uint8_t *>(inputData), featureLen * featureCount);

// 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
aclError ret = aclfvRepoAdd(SEARCH_1_N, featureInfo);

// 销毁aclfvFeatureInfo特征描述信息
aclfvDestroyFeatureInfo(featureInfo);

// 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
// ....

uint8_t featureData[36];
for (size_t i = 0; i < 36; i++) {
    featureData[i] = static_cast<uint8_t>(i);
}

// 创建内存并传输特征数据
void *inputData = nullptr;
aclrtMalloc(&inputData, 36, ACL_MEM_MALLOC_NORMAL_ONLY);
std::shared_ptr<void> inputDataPtr(inputData, [](void *p){(void)aclrtFree(p);});
aclrtMemcpyKind 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，否则无需拷贝，其中dataLen为featureData指针申请的内存长度
if (aclrtGetRunMode(&runMode) == ACL_HOST) {
    kind = ACL_MEMCPY_HOST_TO_DEVICE;
}
aclrtMemcpy(inputData, 36, featureData, dataLen, kind);

// 创建特征描述信息
uint32_t id0 = 0;
uint32_t id1 = 0;
auto featureInfo1 = aclfvCreateFeatureInfo(id0, id1, offset, 36, 1,
                                reinterpret_cast<uint8_t *>(inputData), 36);
std::shared_ptr<aclfvFeatureInfo> featureInfoPtr(featureInfo1,
                                [](aclfvFeatureInfo *p){(void)aclfvDestroyFeatureInfo(p);});

// 删除1个特征
aclfvDel(featureInfo1);

// 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
uint32_t queryCnt = 1;
uint32_t topK = 5;
uint32_t dataLen = queryCnt * topK * sizeof(uint32_t);
uint32_t resultNumDataLen = queryCnt * sizeof(uint32_t);
```



```
const uint32_t tableLen = 32 * 1024;
uint32_t tableDataLen = queryCnt * tableLen;

// 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AdcTabInit用于初始化特征检索输入Adc表，由用户自行实现
uint8_t *tableDataTmp = (uint8_t *)AdcTabInit(1000, queryCnt * 1024);
std::shared_ptr<void> tableDataTmpPtr(tableDataTmp,[](void *p){(void)aclrtFreeHost(p);});

// 为数据表分配内存，tableDataDev用于创建检索输入表信息
void *devPtr = nullptr;
aclrtMalloc(&devPtr, tableDataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
tableDataDev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});

// 拷贝表数据到Device侧
uint8_t *devPtrTmp = reinterpret_cast<uint8_t *>(devPtr);
for (uint32_t i = 0; i < queryCnt; ++i) {
    for (uint32_t j = 0; j < 32; ++j) {
        uint8_t *dst = devPtrTmp + i * 32 * 1024 + j * 1024;
        uint8_t *src = tableDataTmp + i * 1024;
        aclrtMemcpy(dst, 1024, src, 1024, ACL_MEMCPY_HOST_TO_DEVICE);
    }
}

// 为检索结果resultNumDev,id0Dev,id1Dev,resultOffsetDev,resultDistanceDev分配内存
aclrtMalloc(&devPtr, resultNumDataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
resultNumDev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});
aclrtMalloc(&devPtr, dataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
id0Dev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});
aclrtMalloc(&devPtr, dataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
id1Dev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});
aclrtMalloc(&devPtr, dataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
resultOffsetDev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});
aclrtMalloc(&devPtr, dataLen, ACL_MEM_MALLOC_NORMAL_ONLY);
resultDistanceDev.reset(devPtr, [](void *p) {(void)aclrtFree(p);});

// 创建检索输入表信息，结果用于创建检索任务输入信息
aclfvQueryTable *searchQueryTable = aclfvCreateQueryTable(queryCnt, tableLen, reinterpret_cast<uint8_t *>
                                    (tableDataDev.get()), tableDataLen);
searchQueryTable.reset(searchQueryTable, [](aclfvQueryTable *p){(void)aclfvDestroyQueryTable(p);});

// 创建特征库范围参数，结果用于创建检索任务输入信息
aclfvRepoRange *searchRange = aclfvCreateRepoRange(0, 1023, 0, 1023);
searchRange.reset(searchRange, [](aclfvRepoRange *p){(void)aclfvDestroyRepoRange(p);});

// 创建检索任务输入信息，结果用于特征1:N检索
aclfvSearchInput *searchInput = aclfvCreateSearchInput(searchQueryTable, searchRange, topK);
searchInput.reset(searchInput, [](aclfvSearchInput *p){(void)aclfvDestroySearchInput(p);});

// 创建检索结果信息，结果用于特征1:N检索
aclfvSearchResult *searchResult = aclfvCreateSearchResult(queryCnt,
                                  reinterpret_cast<uint32_t *>(resultNumDev.get()),
                                  resultNumDataLen,
                                  reinterpret_cast<uint32_t *>(id0Dev.get()),
                                  reinterpret_cast<uint32_t *>(id1Dev.get()),
                                  reinterpret_cast<uint32_t *>(resultOffsetDev.get()),
                                  reinterpret_cast<float *>(resultDistanceDev.get()),
                                  dataLen);
searchResult.reset(searchResult, [](aclfvSearchResult *p){(void)aclfvDestroySearchResult(p);});

// 5.2 特征1:N检索
aclfvSearch(SEARCH_1_N, searchInput.get(), searchResult.get());

// 5.3 特征检索结果处理
// 获取检索结果
uint32_t dataLen = queryCnt * topK * sizeof(uint32_t);
uint32_t *id0 = (uint32_t *)id0Dev.get();
uint32_t *id1 = (uint32_t *)id1Dev.get();
uint32_t *resultOffset= (uint32_t *)resultOffsetDev.get();
```

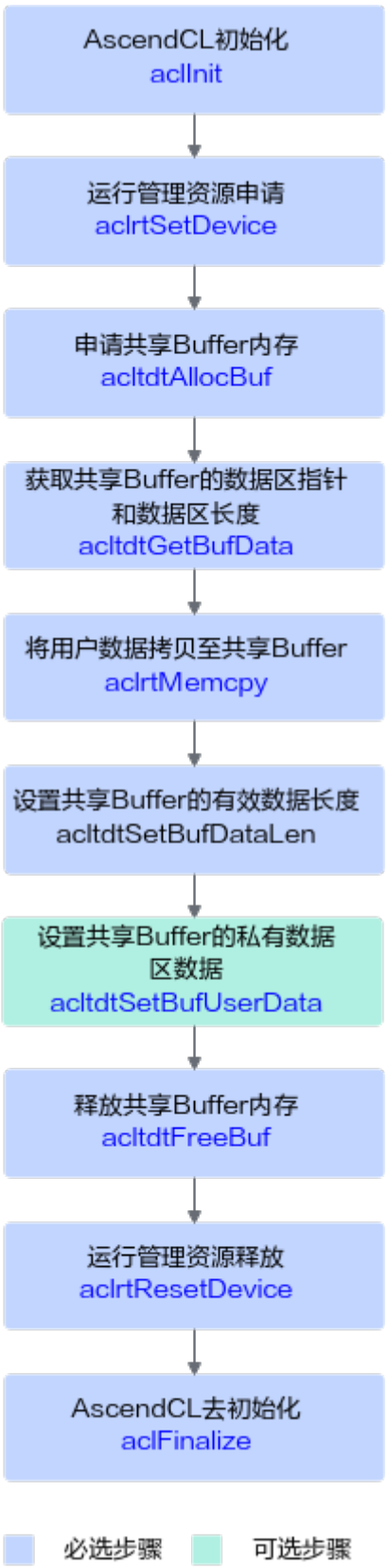


```
float *resultDistance = (float *)resultDistanceDev.get();

// 如果运行模式为ACL_HOST，则需要通过aclrtMemcpy接口将Device的检索结果回传到Host侧；否则无需回传
if (aclrtGetRunMode(&runMode) == ACL_HOST) {
        // 从Device侧拷贝数据到Host侧
        id0 = (uint32_t *)malloc(dataLen);
        id0Ptr.reset(id0);
        id1 = (uint32_t *)malloc(dataLen);
        id1Ptr.reset(id1);
        resultOffset = (uint32_t *)malloc(dataLen);
        resultOffsetPtr.reset(resultOffset);
        resultDistance = (float *)malloc(dataLen);
        resultDistancePtr.reset(resultDistance);
        aclrtMemcpy(id0, dataLen, id0Dev.get(), dataLen, ACL_MEMCPY_DEVICE_TO_HOST);
        aclrtMemcpy(id1, dataLen, id0Dev.get(), dataLen, ACL_MEMCPY_DEVICE_TO_HOST);
        aclrtMemcpy(resultOffset, dataLen, resultOffsetDev.get(), dataLen, ACL_MEMCPY_DEVICE_TO_HOST);
        aclrtMemcpy(resultDistance, dataLen, resultDistanceDev.get(), dataLen,
ACL_MEMCPY_DEVICE_TO_HOST);
}

// 展示底库中的数据
for (uint32_t i = 0; i < queryCnt; i++) {
     for (uint32_t j = 0; j < topK; ++j) {
          uint32_t i0 = id0[i * topK + j];
          uint32_t i1 = id1[i * topK + j];
          uint32_t offset = resultOffset[i * topK + j];
          float distance = resultDistance[i * topK + j];
    }
}

// 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
uint32_t id0Min = 0;
uint32_t id0Max = 1023;
uint32_t id1Min = 0;
uint32_t id1Max = 1023;
aclfvRepoRange *repoRange = aclfvCreateRepoRange(id0Min, id0Max, id1Min, id1Max);
aclfvRepoDel(SEARCH_1_N, repoRange)
// 销毁aclfvInitPara类型的数据
aclfvDestroyInitPara(fvInitPara);

// 7. 释放运行管理资源

// 8. AscendCL去初始化
// ......
```

10.6 共享 Buffer 管理

本特性提供了跨进程共享Buffer管理能力，可配合共享队列一起使用。共享Buffer的共享范围由共享队列的授权确定，共享范围包括“一主多从”的这些进程。

接口调用流程



关键接口的调用流程如上图所示，流程说明如下：

1. **AscendCL初始化**。

   调用aclInit接口实现初始化AscendCL。

2. **运行管理资源申请**。

   具体流程，请参见**6.1 运行管理资源申请与释放**。

3. **申请共享Buffer内存**。

   调用acltdtAllocBuf接口申请共享Buffer内存，此处需根据实际情况选择内存类型。

4. **向共享Buffer中填充有效数据**。

   需先调用acltdtGetBufData接口获取共享Buffer的数据区指针和数据区长度，接着调用aclrtMemcpy接口把用户数据复制到共享Buffer中，最后可调用acltdtSetBufDataLen接口设置有效数据长度，以便在多进程场景下调用acltdtGetBufDataLen接口获取有效数据长度，再根据有效数据长度获取共享Buffer中的数据。

5. **设置共享Buffer的私有数据区数据**。

   调用acltdtSetBufUserData接口设置共享Buffer的私有数据区数据，以便在多进程场景下调用acltdtGetBufUserData接口获取共享Buffer的私有数据区数据。

6. **释放共享Buffer内存**。

   调用acltdtFreeBuf接口来释放共享Buffer。

7. **运行管理资源释放**。

   接口调用流程，请参见**6.1 运行管理资源申请与释放**。

8. **AscendCL去初始化**。

   调用aclFinalize接口实现AscendCL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，不可以直接拷贝编译运行，仅供参考。

```
#include "acl/acl.h"

// 1. AscendCL初始化
// 此处的..表示相对路径，相对可执行文件所在的目录
// 例如，编译出来的可执行文件存放在out目录下，此处的..就表示out目录的上一级目录
const char *aclConfigPath = "../src/acl.json";
aclError ret = aclInit(aclConfigPath);

// 2. 运行管理资源申请，指定计算设备，此处以deviceId = 0为例
ret = aclrtSetDevice(0);

// 3. 申请mbuf内存并进行内存管理，此处以申请DVPP内存、内存大小1024U为例
size_t size = 1024U;
acltdtBuf buf;
ret = acltdtAllocBuf(size, ACL_TDT_DVPP_MEM, &buf);

// 4. 向共享Buffer中填充有效数据
// 4.1 获取共享Buffer的数据区指针和数据区长度
void *dataPtr = nullptr;
size_t dataSize = 0U;
ret = acltdtGetBufData(buf, &dataPtr, &dataSize);

// 4.2 把用户数据拷贝到共享Buffer中
size_t len = 512U; // 用户数据长度
void *ptr = new (std::nothrow) char_t[len];  // 用户申请自己的内存
// 用户对自己申请的数据进行处理
//……
```



```
ret = aclrtMemcpy(dataPtr, size, ptr, len, ACL_MEMCPY_HOST_TO_DEVICE);
// 5. 释放内存
delete[] ptr;
ptr = nullptr;
delete[] newPtr;
newPtr = nullptr;
ret = acltdtFreeBuf(buf);

// 6. 运行管理资源释放
ret = aclrtResetDevice(0);

// 7. AscendCL去初始化
ret = aclFinalize();
```


11 应用编译&运行

完成程序代码编写后，可按照本节中的指导编译程序、执行应用。

编译及运行应用

1. **编译代码**。

   可参考**13.10 基于Caffe ResNet-50网络实现图片分类（同步推理）**中的说明，主要步骤如下：

   a. 创建编译脚本。

      您可以从该**13.10 基于Caffe ResNet-50网络实现图片分类（同步推理）**样例中获取编译脚本CMakeLists.txt，在该编译脚本的基础上修改如下参数。

      - include_directories：添加头文件所在的目录。依赖的头文件请参见**调用接口依赖的头文件和库文件说明**。

        示例如下：

        ```
        include_directories(
            directoryPath1
            directoryPath2
        )
        ```

      - link_directories：添加库文件所在的目录。

        示例如下：

        ```
        link_directories(
            directoryPath3
            directoryPath4
        )
        ```

      - add_executable：修改可执行文件的名称（例如：*main*）、添加*.cpp文件所在的目录。

        示例如下：

        ```
        add_executable(main
            directoryPath5
            directoryPath6)
        ```

      - target_link_libraries：修改可执行文件的名称（与add_executable中设置的名称保持一致）、添加可执行文件依赖的库文件。依赖的库文件与接口所在的头文件有关，请参见**调用接口依赖的头文件和库文件说明**。

        示例如下：

        ```
        target_link_libraries(main
            ascendcl
        ```



```
    libName1
    libName2)
```

说明

编译基于AscendCL接口的代码逻辑时，请按照include的头文件依赖对应的库文件，如果引用多余的so文件（例如libascendcl.a），可能导致版本功能异常或后续版本升级时存在兼容性问题。

- 编译选项，修改可执行文件的名称（与与add_executable中设置的名称保持一致），以及可执行文件的安装目录。

  示例如下，表示*main*安装在${CMAKE_INSTALL_PREFIX}/out目录下，${CMAKE_INSTALL_PREFIX}变量定义的路径是相对路径，相对cmake命令执行的路径：

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY  "../../../out")

install(TARGETS main DESTINATION ${CMAKE_RUNTIME_OUTPUT_DIRECTORY})
```

b. **设置环境变量**。

如果用户是自行开发的编译脚本，未使用DDK_PATH、NPU_HOST_LIB环境变量，则无需设置该环境变量，用户根据编辑脚本逻辑自行增加相关配置即可。

如果用户是基于**13.10 基于Caffe ResNet-50网络实现图片分类（同步推理）**样例中的CMakeLists.txt编译脚本进行修改时，则需设置DDK_PATH、NPU_HOST_LIB环境变量。

DDK_PATH、NPU_HOST_LIB环境变量分别用于指向AscendCL头文件目录、库文件目录。如下为设置环境变量的示例，${INSTALL_DIR}表示CANN软件安装目录，例如，$HOME/Ascend/ascend-toolkit/latest，*arch*表示操作系统架构，*{os}*表示操作系统。

```
export DDK_PATH=${INSTALL_DIR}
export NPU_HOST_LIB=${INSTALL_DIR}/{arch-os}/devlib
```

须知

使用“${INSTALL_DIR}/*{arch-os}*/devlib”目录下的*.so库，是为了编译基于AscendCL接口的代码逻辑时，不依赖其它组件（例如Driver）的任何*.so库。因此在使用cmake编译命令时，请务必将DCMAKE_SKIP_RPATH设置为TRUE，代表不会将rpath路径（即“${INSTALL_DIR}/{arch-os}/devlib”）添加到编译生成的可执行文件中去。

编译通过后，运行应用时，通过配置环境变量，应用会链接到运行环境上“lib64”目录下的*.so库，运行时会自动链接到依赖其它组件的*.so库。

c. 执行cmake命令编译代码。

示例命令如下，其中，“../../../src”表示CMakeLists.txt文件所在的目录，请根据实际目录层级修改；通过-DCMAKE_CXX_COMPILER参数指定的编译器，请根据实际版本要求修改。

- 当开发环境与运行环境操作系统架构相同时，执行如下命令编译：

```
cmake ../../../src -DCMAKE_CXX_COMPILER=g++ -DCMAKE_SKIP_RPATH=TRUE
```

- 当开发环境与运行环境操作系统架构不同时，执行以下命令进行交叉编译：

  例如，当开发环境为X86架构，运行环境为AArch64架构时，执行以下命令进行交叉编译。


```
cmake ../../../src -DCMAKE_CXX_COMPILER=aarch64-linux-gnu-g++ -
DCMAKE_SKIP_RPATH=TRUE
```

说明

关于cmake参数的详细介绍，请参见**https://cmake.org/cmake/help/latest/guide/tutorial/index.html**，选择对应的版本后再查看参数。

d. 执行make命令生成可执行文件。

```
make
```

2. **运行可执行文件**。

将AscendCL初始化配置文件（acl.json）所在的目录、可执行文件所在的目录、测试图片所在的目录、*.om文件所在的目录都上传到运行环境的同一个目录下，再登录到运行环境，切换到可执行文件所在的目录，运行可执行文件，示例命令如下：

```
./main
```

如果在**AscendCL初始化**阶段，在aclInit接口中传入空指针，则无需将AscendCL初始化配置文件（acl.json）所在的目录上传到运行环境。

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**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，调用AscendCL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**10.2 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》。

12 精度/性能优化

12.1 调优简介

本章重点介绍推理应用的精度、性能调优，由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是推理精度错误、推理精度与标杆数据存在少量差距、模型推理性能不符合预期或待提升等问题。

- **应用的精度问题**可能由于推理功能与其它功能之间的串接问题、整网中算子本身的精度问题等，可参考本章中的建议排查功能串接时的接口参数配置问题、借助工具获取详细数据定位分析问题。
- **应用的性能问题**可能由于模型在昇腾AI处理器上的算子适配或数据读写问题、DVPP接口使用问题等，可参考本章中的建议排查接口使用问题、借助工具优化模型、借助工具获取详细数据定位分析问题。

12.2 模型推理精度提升建议

12.2.1 精度提升简介

本文介绍整网推理场景下的精度调优流程、相关配置及典型案例等。由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是推理精度错误，或推理精度与标杆数据存在少量差距。

在整网推理时，可能由于以下原因导致推理精度错误或者推理精度不达标：

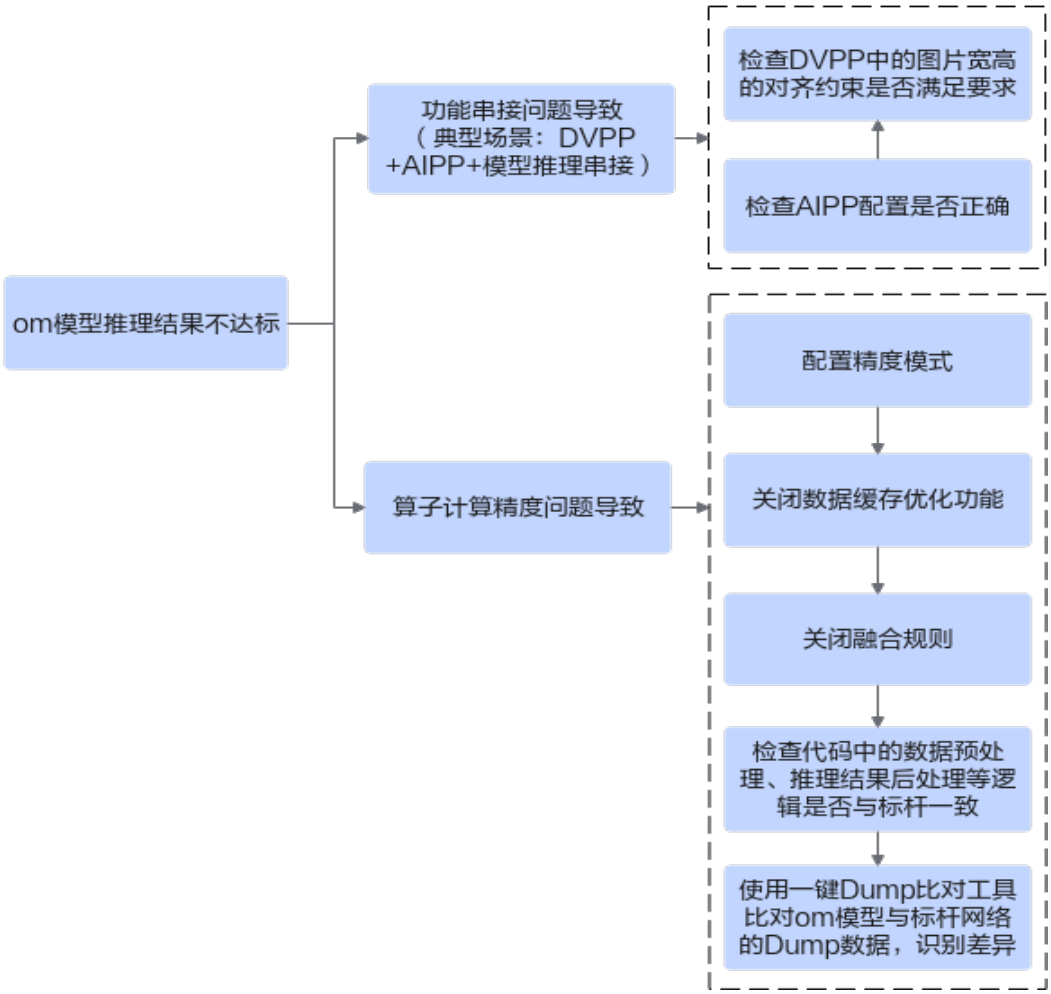
- 推理功能与其它功能之间的串接问题，当前比较典型的是DVPP+AIPP+模型推理串联使用的场景，该场景主要是因为接口中参数的配置问题、模型转换时AIPP的配置问题导致。

如果存在DVPP、AIPP功能、模型推理串联使用的场景，建议先排查这部分问题；否则，可以跳过该排查，直接排查算子本身的精度问题。

本文也结合具体的正、反例给出了配置建议，供参考。

- 整网中算子本身的精度问题，该类问题可以借助精度比对工具、一键Dump比对工具（点击Link获取），根据下文中具体的问题定位流程获取各类数据后，再进行比对、分析，确认是配置问题，还是算子实现问题，再逐一解决问题。本文中会结合具体的比对、分析的案例，介绍如何比对、分析。

图 12-1 推理精度问题



说明

- 本文中的“推理”，当前限定为使用om离线模型文件进行推理的场景。
- 本文中的算子精度问题，是指整网中算子的精度问题，对于自定义的TBE算子，运行验证时的发现精度问题，请参见《TBE&AI CPU自定义算子开发指南》。

12.2.2 DVPP+AIPP+模型推理精度提升建议（Atlas 200/300/500 推理产品）

DVPP+AIPP+模型推理串联使用，由于接口中参数的配置问题、模型转换时AIPP的配置问题可能会导致模块之间输入、输出的衔接存在问题，最后影响整网的推理精度。

请参考**9.10.1 JPEGD+VPC+模型推理精度提升建议（Atlas 200/300/500 推理产品）**中内容了解精度提升建议、典型案例以及示例代码。

12.2.3 算子精度导致推理结果不达标

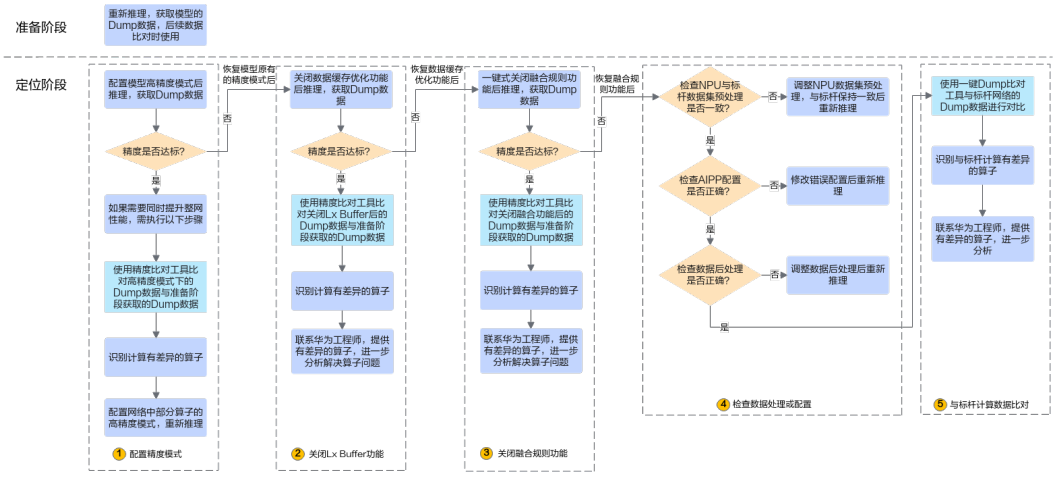
12.2.3.1 问题描述

推理结果不达标，包括以下两种情况：

- 算子精度导致推理结果错误，是指整网推理的功能已调通，但推理结果错误，例如目标检测网络MAP结果全0、昇腾om模型的推理结果与标杆网络的推理结果比对时余弦相似度为0。
- 算子精度导致推理精度不达标，是指整网推理的功能已调通，单次昇腾om模型的推理结果与标杆网络的推理结果比对时余弦相似度在95%以上，但数据集推理精度与标杆数据存在少量差距，例如：
  - 分类网络昇腾om模型，Top1/Top5分别为：0.90/0.70,；标杆网络Top1/Top5分别为：0.92/0.71。
  - 检测网络昇腾om模型MAP精度：0.54；标杆网络MAP精度：0.55。

12.2.3.2 问题定位流程

图 12-2 定位流程



步骤1 推理结果错误，为了后续定位问题，需要重新执行推理，用于获取模型的Dump数据。

获取模型的Dump数据，需要调用AscendCL接口打开Dump开关，详细描述请参见《精度比对工具使用指南》。

步骤2 配置精度模式。

1. 配置模型高精度模式后推理，获取模型的Dump数据。推理后，如果精度达标，则进行步骤**步骤2.2**；如果精度不达标，则进行步骤**步骤3**。

   配置模型高精度模式后推理，可能会影响推理性能，如果在精度达标的同时，需要保持性能，则执行**步骤2.2**~**步骤2.4**，配置部分算子保持原始网络中的数据类型。

   配置模型高精度模式，请参见**配置网络模型的高精度模式**。

2. 使用精度比对工具比对高精度模式下的Dump数据与**步骤1**获取的Dump数据。

   工具的使用请参见《精度比对工具使用指南》。

3. 根据**步骤2.2**中的比对结果识别计算有差异的算子。

   一般来说，每次识别一个差异算子（首个余弦相似度较低的算子，例如低于0.95），找到差异算子后，执行**步骤2.4**推理，推理的同时获取Dump数据，用来与高精度模式下的Dump数据比对，继续找到下一个差异算子。

   需要循环执行该步骤，直至没有差异算子。

4. 对于有差异的算子，配置该部分算子保持原始网络中的数据类型，再重新推理。

   配置部分算子的高精度模式，请参见**配置部分算子保持原始网络中的数据类型**。

步骤3 关闭数据缓存优化功能。

1. 恢复模型的原有精度模式后，关闭数据缓存优化功能后推理，如果精度达标，则进行步骤**步骤3.2**；如果精度不达标，则进行步骤**步骤4**。

   当前默认开启数据缓存优化，开启数据缓存优化可提高计算效率、提升性能，但由于部分算子在实现上可能存在未考虑的场景，导致影响精度，因此在出现精度问题时可以尝试关闭数据缓存优化。如果关闭数据缓存优化功能后，精度达标，则还是需要识别出问题算子，反馈给华为工程师进一步分析、解决算子问题，解决算子问题后，建议保持开启数据缓存优化。

   关闭数据缓存优化功能，请参见**12.2.3.4 关闭数据缓存优化**。

2. 使用精度比对工具比对关闭数据缓存优化功能后的Dump数据与**步骤1**获取的Dump数据。

   工具的使用请参见《精度比对工具使用指南》。

3. 根据**步骤3.2**中的比对结果识别计算有差异的算子。
4. 联系华为工程师（单击**Link**后新建Issue），提供有差异的算子，进一步分析。

步骤4 关闭融合规则功能。

1. 恢复启用数据缓存优化功能，关闭融合规则功能后推理，如果精度达标，则进行步骤**步骤4.2**；如果精度不达标，则进行步骤**步骤5**。

   当前默认开启融合规则，开启融合规则可提高计算效率、提升性能，但算子之间可能会融合，融合后的部分算子在实现上可能存在未考虑的场景，导致影响精度，因此在出现精度问题时可以尝试关闭融合规则。如果关闭融合规则功能后，精度达标，则还是需要识别出问题算子，反馈给华为工程师进一步分析、解决算子问题，解决算子问题后，建议保持开启融合规则功能。

   关闭融合规则功能，请参见**12.2.3.5 关闭融合规则**。关闭某些融合规则可能会导致功能问题，因此在配置关闭融合规则后，系统在不影响功能的前提下关闭部分融合规则，而不是全部融合规则。

2. 使用精度比对工具比对关闭融合规则后的Dump数据与**步骤1**获取的Dump数据。

   工具的使用请参见《精度比对工具使用指南》。

3. 根据**步骤4.2**中的比对结果识别计算有差异的算子。
4. 联系华为工程师（单击**Link**后新建Issue），提供有差异的算子，进一步分析。

步骤5 检查数据处理或配置。

推理精度不达标可能是由于数据集、AIPP、后处理方式的差异导致，需逐步进行排查，恢复启用融合规则功能后，请检查数据处理或配置，参见**12.2.3.6 检查数据处理或配置**。

如果数据处理逻辑或数据配置有问题，则需修改后重新推理；如果数据处理逻辑或数据配置没有问题，则进行**步骤6**。

步骤6 与标杆计算数据比对。

1. 使用一键Dump比对工具与标杆网络的Dump数据进行对比。
   一键Dump比对工具及其使用指导请单击**Link**获取。
2. 根据**步骤6.1**中的比对结果识别计算有差异的算子。
3. 联系华为工程师（单击**Link**后新建Issue），提供有差异的算子，进一步分析。

----结束

12.2.3.3 配置精度模式

如果在模式转换时不指定网络模型或算子的精度模式，默认采用fp16（float16）数据类型进行计算。

配置模型高精度模式后推理，可提升精度，但可能会影响推理性能，如果在精度达标的同时，需要保持性能，则可以配置部分算子保持原始网络中的数据类型。关于ATC参数的详细说明请参见《ATC工具使用指南》。

配置网络模型的高精度模式

步骤1 使用ATC工具转换模型时，增加高级参数--precision_mode，用于指定精度模式。

参数设置如下所示，表示如果网络模型中算子支持fp32（float32），则使用fp32；如果网络模型中算子不支持fp32，则使用fp16（float16）。

```
--precision_mode=allow_fp32_to_fp16
```

步骤2 使用转换后的om模型重新推理。

----结束

配置部分算子保持原始网络中的数据类型

步骤1 使用ATC工具转换模型时，增加高级参数--keep_dtype（指定部分算子计算时保持原始网络的数据类型）和--precision_mode（指定网络模型的精度模式）。

参数使用示例如下：

```
--keep_dtype=$HOME/execeptionlist.cfg --precision_mode=force_fp16
```

配置文件名举例为*execeptionlist.cfg*，配置文件样例如下，文件中每一行是一个算子的名称，将配置好的*execeptionlist.cfg*文件上传到ATC工具所在服务器任意目录：

```
Opname1
Opname2
…
```

步骤2 使用转换后的om模型重新推理。

----结束

12.2.3.4 关闭数据缓存优化

如果在模型转换时不指定关闭数据缓存优化功能，当前默认开启数据缓存优化，开启数据缓存优化可提高计算效率、提升性能，但由于部分算子在实现上可能存在未考虑的场景，导致影响精度，因此在出现精度问题时可以尝试关闭数据缓存优化。

如果关闭数据缓存优化功能后，精度达标，则还是需要识别出问题算子，反馈给华为工程师进一步分析、解决算子问题，解决算子问题后，建议保持开启数据缓存优化。

步骤1 使用ATC工具转换模型时，增加高级参数：--buffer_optimize，用于关闭数据缓存优化。

参数设置如下所示，：

```
--buffer_optimize=off_optimize
```

参数的详细说明请参见《ATC工具使用指南》。

步骤2 使用转换后的om模型重新推理。

----结束

说明

在联系华为工程师前（单击**Link**后新建Issue），设置DUMP_GE_GRAPH、DUMP_GRAPH_LEVEL环境变量，重新模型转换，打印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”。

12.2.3.5 关闭融合规则

如果在模型转换时不指定关闭融合规则，当前默认开启融合规则，开启融合规则可提高计算效率、提升性能，但算子之间可能会融合，融合后的部分算子在实现上可能存在未考虑的场景，导致影响精度，因此在出现精度问题时可以尝试关闭融合规则。

如果关闭融合规则功能后，精度达标，则还是需要识别出问题算子，反馈给华为工程师进一步分析、解决算子问题，解决算子问题后，建议保持开启融合规则功能。

步骤1 使用ATC工具转换模型时，增加高级参数：--fusion_switch_file

参数使用示例如下：

```
--fusion_switch_file=$HOME/module/fusion_switch.cfg
```

配置文件名举例为*fusion_switch.cfg*，配置文件样例如下，将配置好的*fusion_switch.cfg*文件上传到ATC工具所在服务器任意目录：

```
{
    "Switch":{
        "GraphFusion":{
            "ALL":"off"
        },
        "UBFusion":{
            "ALL":"off"
        }
    }
}
```

参数的详细说明请参见《ATC工具使用指南》。

步骤2 使用转换后的om模型重新推理。

----结束

说明

在联系华为工程师前（单击**Link**后新建Issue），设置DUMP_GE_GRAPH、DUMP_GRAPH_LEVEL环境变量，重新模型转换，打印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”。

12.2.3.6 检查数据处理或配置

步骤1 检查昇腾om模型与标杆网络推理的输入数据以及输入数据的处理是否一致，如果不一致，需调整成一致。

步骤2 检查AIPP配置。

AIPP（Artificial Intelligence Pre-Processing），用于在AI Core上完成图像预处理，包括改变图像尺寸、色域转换（转换图像格式）、减均值/乘系数（改变图像像素），数据处理之后再进行真正的模型推理。

如果AIPP配置错误可能导致模型推理的输入数据不准确，需要参见《ATC工具使用指南》中的“高级功能>AIPP使能”章节检查AIPP配置，如有不正确的AIPP配置，修改正确后，重新转换模型，再重新推理。

步骤3 检查昇腾om模型与标杆网络推理结果的后处理方式是否一致，如果不一致，需调整成一致。

----结束

12.2.3.7 案例介绍

案例描述

Fastrcnn网络，模型转换时，保持默认高性能模式、force_fp16精度模式，推理出来的精度错误，MAP结果为0。

然后，在模型转换时，设置模型的高精度模式（precision_mode=allow_fp32_to_fp16），推理出来的精度正确。

案例分析

1. 模型转换时保持默认高性能模式、force_fp16精度模式，进行推理，获取该模式下的Dump数据文件。
2. 再次模型转换，设置模型的高精度模式（precision_mode=allow_fp32_to_fp16），再次进行推理，获取该模式下的Dump数据文件。
3. 使用精度比对工具，比对**1**与**2**中的Dump数据。

   比对结果示例如下：

Index	LeftOp	RightOp	TensorIndex	CosineSim	MaxAbsolu	AccumulatRelativeEKullbackL	StandardE	CompareFailReason
844	multilevel_crop_an	multilevel	multilevel_crop_and_resize/AddN:input:1	NaN	NaN	NaN NaN NaN NaN	NaN	[multilevel_crop_and_resize/
844	multilevel_crop_an	multilevel	multilevel_crop_and_resize/AddN:input:2	NaN	NaN	NaN NaN NaN NaN	NaN	[multilevel_crop_and_resize/
844	multilevel_crop_an	multilevel	multilevel_crop_and_resize/AddN:input:3	NaN	NaN	NaN NaN NaN NaN	NaN	[multilevel_crop_and_resize/
844	multilevel_crop_an	multilevel	multilevel_crop_and_resize/AddN:output:0	0.722928	130050	460707.5 1.057259	inf	(84351.02Cannot compare by KL Divergen
845	multilevel_crop_an	multilevel	multilevel_crop_and_resize/Cast_2:input:0	0.722928	130050	460707.5 1.057259	inf	(84351.02Cannot compare by KL Divergen
845	multilevel_crop_an	multilevel	multilevel_crop_and_resize/Cast_2:output:0	0.722928	130050	460707.5 1.057259	inf	(84351.02Cannot compare by KL Divergen
846	multilevel_crop_an	multilevel	multilevel_crop_and_resize/GatherV2:input:0	0.999999	0.073951	343481.8 0.001661	0	(-0.025,0.937) (-0.025,0.937)
846	multilevel_crop_an	multilevel	multilevel_crop_and_resize/GatherV2:input:1	0.722928	130050	460707.5 1.057259	inf	(84351.02Cannot compare by KL Divergen
846	multilevel_crop_an	multilevel	multilevel_crop_and_resize/GatherV2:output:0	0.015732	16.76621	3.4E+08 1.304415	inf	(-0.060,1Cannot compare by KL Divergen

4. 从图中可以看CosineSimilarity这一列，余弦相似度算法比对出来的结果，范围是[-1,1]，比对的结果如果越接近1，表示两者的值越相近，越接近-1意味着两者的值越相反。对于大部分算子，值低于0.95就说明存在精度问题。

   上图中AddN算子第0个输出的余弦相似度只有0.72，说明这个算子可能存在精度问题，因此需要进一步分析该算子在高精度模式下的第0个输出的Dump数据文件（**2**中获取的Dump数据文件）。
5. 由于Dump数据文件无法通过文本工具直接查阅，因此在分析该Dump数据文件前，请参考《精度比对工具使用指南》的“附录>如何查看dump数据文件”章节，先将dump数据文件转换为numpy格式，再将numpy格式文件为txt格式文件。

在将numpy格式文件为txt格式文件的过程中，可以获取AddN算子第0个输出的最大值、最小值，命令示例如下（*******.npy***表示numpy格式文件的路径）：

```
$ python3
Python 3 (default, Mar  5 2020, 16:07:54)[GCC 5.4.0 20160609] on linuxType "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a = np.load("****.npy")
>>> a.max()
>>> 109508.0
>>> a.min()
>>> 70683.0
```

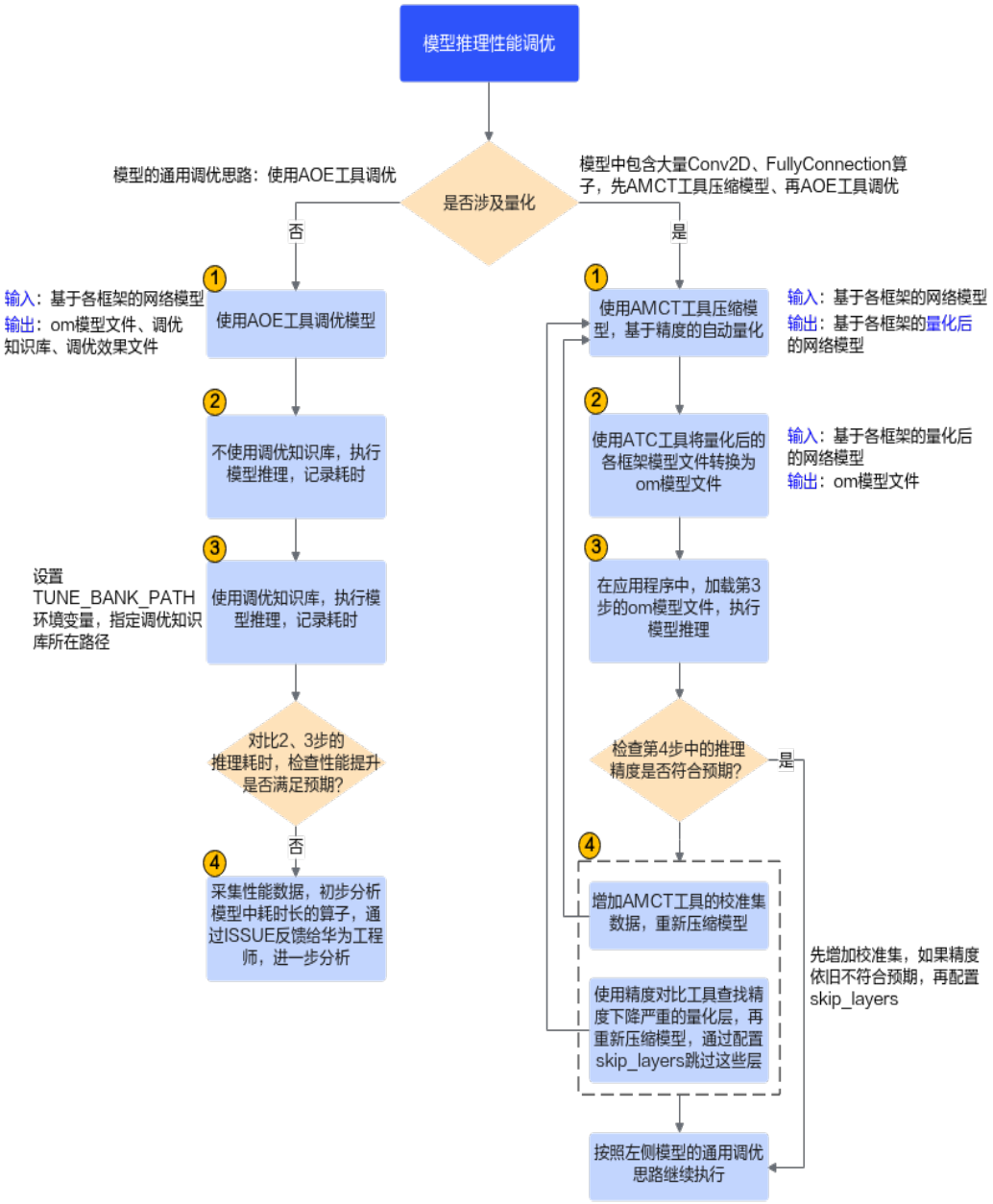
6. 从**5**获取到的AddN算子第0个输出的最大值、最小值，可以看出高精度模式下AddN算子输出tensor的最大值为109508.0，而高性能模式（fp16）下，输出tensor的最大值为65504.0（FP16能表达的最大值域范围为（-65505~65504），由此可以得出高精度模式下AddN算子的输出值大于fp16类型域表达范围，因此需要配置该算子走高精度模式，参见**配置部分算子保持原始网络中的数据类型**。

12.3 模型推理性能调优建议

12.3.1 模型推理性能调优思路

在整网推理时，可能由于模型在昇腾AI处理器上的算子适配、数据读写等问题，导致模型推理的性能不符合预期，您可以查阅本节介绍的内容，了解模型推理时的性能调优流程。由于是调优，因此在调优前，请确保已经完成了整网推理功能调测，功能不阻塞，只是模型推理性能不符合预期、待提升。

在下图的性能调优流程中，涉及调优的关键工具为：模型调优工具AOE（Ascend Optimization Engine）、模型压缩工具AMCT（Ascend Model Compression Toolkit）。在调优过程中，涉及转换模型、记录模型推理耗时、分析性能瓶颈点等操作时，还会辅助使用模型转换工具ATC、性能数据采集工具、精度比对工具。



12.3.2 使用 AOE 工具调优模型

使用aoe命令调优时，会生成：适配昇腾AI处理器的om模型文件、基于昇腾AI处理器的调优知识库、调优效果文件（通过效果文件可查阅模型中各算子的性能提升率）。由于生成om模型文件，所以aoe命令是包含模型转换的功能，因此aoe命令的参数，一般来说，是在atc命令参数的基础上增加了**--job_type**参数，增加**子图调优**、**算子调优**的功能。

先进行子图调优，再进行算子调优，原因是：先进行子图调优会生成图的切分方式，子图调优后算子已经被切分成最终的shape了，再进行算子调优，会基于这个最终shape去做算子调优。如果优先算子调优，这时调优的算子shape不是最终切分后的算子shape，可能会影响最终的调优效果。

本节介绍**推理**场景下使用**aoe命令**进行子图调优、算子调优的基本方法，关于AOE工具的其它参数、使用约束介绍请参见《AOE工具使用指南》。

操作步骤

1. **使用aoe命令依次执行子图调优、算子调优。**
   - **子图调优命令示例如下所示：**
     ```
     aoe --model=${HOME}/module/resnet50_pytorch_1.4.onnx --framework=5 --job_type=1
     ```
   - **算子调优命令示例如下所示：**
     ```
     aoe --model=${HOME}/module/resnet50_pytorch_1.4.onnx --framework=5 --job_type=2
     ```
2. **查看调优结果。**

   执行调优结果如下所示，代表调优完成并且性能有提升。调优后，同时生成自定义知识库、om模型文件以及调优效果文件。

   ```
   <xxxx> process finished. Performance improved by xx%   //xxxx：调优任务名称,xx%：性能提升的比例。
   ```

   调优结果文件如下：
   - **基于昇腾AI处理器的调优知识库**
     - 针对子图调优，默认存储到${HOME}/Ascend/latest/data/aoe/custom/graph/${soc_version}目录下。
     - 针对算子调优，默认存储到${HOME}/Ascend/latest/data/aoe/custom/op/${soc_version}路径下。
   - **适配昇腾AI处理器的om模型文件**

     调优后的om模型默认存放在执行aoe命令的当前目录下，具体路径如下：${model_name}_${timestamp}/tunespace/result/${model_name}_${timestamp}_tune.om（或者${model_name}_${timestamp}_tune_${os}_${arch}.om）。
   - **调优效果文件**

     执行aoe命令的当前目录下生成命名为“aoe_result_opat_{timestamp}_{pid*xxx*}.json”的文件，记录调优过程中被调优的算子信息。

     json文件中的内容片段示例如下：

     ```
     {
             "op_name": "Conv_125",
             "op_type": "Conv2D",
             "tune_performance": {
               "Schedule": {
                 "performance_after_tune(us)": 72.046,
                 "performance_before_tune(us)": 72.055,
                 "performance_improvement": "0.01%",
                 "update_mode": "add"
               }
             }
     }
     ```
3. **指定调优知识库，再执行模型推理。**
   a. 设置TUNE_BANK_PATH环境变量，指定为AOE调优后的自定义知识库存放路径，该路径下的graph目录下为子图调优知识库、op目录下为算子调优知识库。示例如下：
      ```
      export TUNE_BANK_PATH=/home/HwHiAiUser/custom
      ```
   b. AscendCL模型推理。

      使用**1**中执行算子调优命令生成的om模型文件，执行模型推理。

      或者可以使用**msame工具**快速推理，查看推理耗时数据。

> **说明**
>
> **支持“一次AOE调优+多次ATC模型转换”的场景**，使用AOE工具调优模型后，若由于其它业务需求需要重新转换模型，可通过环境变量指定AOE调优知识库的路径，再使用ATC工具重新转换模型，这样就可以基于知识库中的调优策略编译模型、转换出调优后的模型。

12.3.3 采集&解析性能数据

本节介绍**推理**场景下使用**msprof命令行**方式采集和解析性能数据、并通过生成的结果文件分析性能瓶颈的基本方法，关于msprof命令行的详细参数介绍、以及其它性能数据采集分析方式请参见《 性能分析工具使用指南 》。

采集、解析并导出性能数据

步骤1 **执行msprof命令一键式采集、解析并导出性能数据。**

```
msprof --application=/home/HwHiAiUser/HIAI_PROJECTS/MyAppname/out/main --output=/home/HwHiAiUser/profiling_output
```

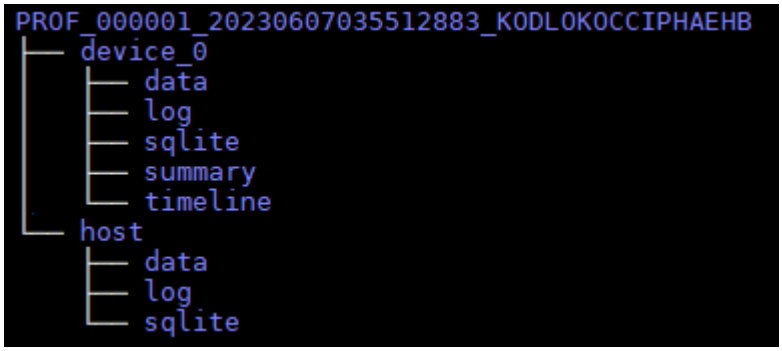
表 12-1 常用参数说明

参数	描述	可选/必选
--application	配置为运行环境上AI任务文件。 不建议配置其他用户目录下的AI任务，避免提权风险。 不建议使用此参数进行有安全风险的高危操作，如删除文件或目录、修改密码、提权命令等。	必选
--output	收集到的Profiling数据的存放路径，默认为AI任务文件所在目录。	可选

默认情况下，导出迭代数最多的模型ID（Model ID）对应的第一轮迭代的性能数据。

步骤2 命令执行完成后，在output指定的目录下生成**PROF_XXX目录，存放采集并解析后的性能数据**，目录结构如图12-3所示。

图 12-3 性能数据目录结构（仅为示例）



- data/sqlite文件夹为采集和解析的过程数据，一般无需关注。
- log文件夹为日志文件，一般无需关注。
- summary文件夹汇总了AI任务运行时的软硬件数据。
- timeline文件夹汇总了AI任务运行的时序信息。

步骤3 进入summary和timeline目录，**查看性能数据文件**。

默认情况下采集到的文件请参考**表12-2**。

表 12-2 msprof 默认配置采集的性能数据文件

文件夹	文件名	说明
timeline	msprof*.json	timeline数据总表。
	acl_*.json	AscendCL接口调用时序。训练场景不生成。
	ai_stack_time_*.json	昇腾AI软件栈各组件（AscendCL，GE，Runtime，Task Scheduler等）运行时序。
	ge_*.json	GE接口耗时数据。
	step_trace_*.json	迭代轨迹数据，每轮迭代的耗时。
	task_time_*.json	Task Scheduler任务调度时序。
	thread_group_*.json	AscendCL，GE，Runtime组件耗时数据。
	ge_op_execute_*.json	算子下发各阶段耗时数据。当模型为动态Shape时自动采集并生成该文件。
summary	acl_*.csv	AscendCL API调用过程。训练场景不生成。
	acl_statistic_*.csv	AscendCL API数据统计。训练场景不生成。
	op_summary_*.csv	AI Core和AI CPU算子数据。
	op_statistic _*.csv	AI Core和AI CPU算子调用次数及耗时统计。
	step_trace_*.csv	迭代轨迹数据。
	task_time_*.csv	Task Scheduler任务调度信息。
	ai_stack_time_*.csv	昇腾AI软件栈各组件（AscendCL，GE，Runtime，Task Scheduler等）信息。
	fusion_op_*.csv	模型中算子融合前后信息。
	ge_op_execute_*.csv	算子下发各阶段耗时数据。当模型为动态Shape时自动采集并生成该文件。
	prof_rule_0.json	调优建议。
注：“*”表示{device_id}_{model_id}_{iter_id}，其中{device_id}表示设备ID，{model_id}表示模型ID，{iter_id}表示某轮迭代的ID。		

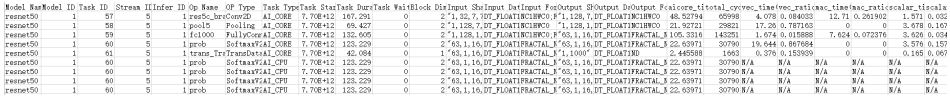
- timeline文件后缀为json，需要在Chrome浏览器中输入**chrome://tracing**，将文件拖到空白处进行打开，通过键盘上的快捷键（w：放大，s：缩小，a：左移，d：右移）。通过该文件可查看当前AI任务运行的时序信息，比如运行过程中接口调用时间线，如图12-4所示。

  图 12-4 查看 timeline 文件

- summary文件后缀为csv，可直接打开查看。通过该文件可以看到AI任务运行时的软硬件数据，比如各算子在AI处理器软硬件上的运行耗时，通过字段排序等可以快速找出需要的信息，如图12-5所示。

  图 12-5 查看 summary 文件

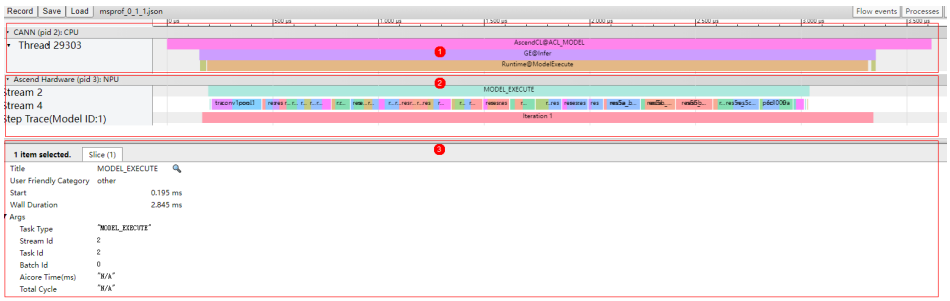
----结束

性能分析

从上文我们可以看到，性能数据文件较多，分析方法也较灵活，以下介绍几个重要文件及分析方法。

- 通过**msprof*.json**文件从整体角度查看AI任务运行的时序信息，进而分析出可能存在的瓶颈点。

  图 12-6 msprof*.json 文件示例

  - 区域1：CANN层数据，主要包含AscendCL、GE和Runtime组件的耗时数据。
  - 区域2：底层NPU数据，主要包含Task Schduler组件耗时数据、迭代轨迹数据。
  - 区域3：展示timeline中各算子、接口的详细信息，单击区域1和区域2中各timeline时展示。

  从上图可以大致分析出AI任务在哪个阶段耗时较多，比如发现区域1的AscendCL aclmdlExecute接口执行阶段耗时较多，可以继续查看区域2的Task Schduler任务调度信息，分析执行推理过程中具体耗时较长的任务，查看区域3的耗时较长的接口和算子，再结合summary文件进行量化分析，定位出具体的性能瓶颈。

- 通过**op_statistic_*.csv**文件分析各类算子的调用总时间、总次数等，排查是否某类算子总耗时较长，进而分析这类算子是否有优化空间。

  图 12-7 op_statistic_*.csv 文件示例

Model Nar	OP Type	Core Type	Count	Total Time(us)	Min Time(u	Avg Time(u	Max Time(	Ratio(%)
resnet50	Conv2D	AI_CORE	50	5618.09	39.11	112.361	330.37	84.1293
resnet50	Pooling	AI_CORE	2	263.65	80.84	131.825	182.81	3.9481
resnet50	ReadSelec	AI_CORE	3	209.36	68.23	69.786	71.4	3.1351
resnet50	FullyConne	AI_CORE	1	204.37	204.37	204.37	204.37	3.0604
resnet50	SoftmaxV2	AI_CORE	1	154.42	154.42	154.42	154.42	2.3124
resnet50	Cast	AI_CORE	2	115.9	37.61	57.95	78.29	1.7356
resnet50	TransData	AI_CORE	2	112.13	39.94	56.065	72.19	1.6791

  可以按照Total Time排序，找出哪类算子耗时较长。

- 通过**op_summary_*.csv**文件分析具体某个算子的信息和耗时情况，从而找出高耗时算子，进而分析该算子是否有优化空间。

  图 12-8 op_summary*.csv 文件示例

Model Nar	Model ID	Task ID	Stream ID	Infer ID	Op Name	OP Type	Task Type	Task Start	Task Duration(us)
resnet50	1	53	1	1	res5b_brar	Conv2D	AI_CORE	1.19E+15	330.37
resnet50	1	50	1	1	res5a_brar	Conv2D	AI_CORE	1.19E+15	325.1
resnet50	1	54	1	1	res5b_brar	Conv2D	AI_CORE	1.19E+15	281.66
resnet50	1	57	1	1	res5c_bran	Conv2D	AI_CORE	1.19E+15	281.1
resnet50	1	55	1	1	res5b_brar	Conv2D	AI_CORE	1.19E+15	273.23
resnet50	1	52	1	1	res5a_brar	Conv2D	AI_CORE	1.19E+15	238.59
resnet50	1	51	1	1	res5a_brar	Conv2D	AI_CORE	1.19E+15	213.7
resnet50	1	60	1	1	fc1000	FullyConne	AI_CORE	1.19E+15	204.37

  Task Duration字段为算子耗时信息，可以按照Task Duration排序，找出高耗时算子；也可以按照Task Type排序，查看不同核（AI Core和AI CPU）上运行的高耗时算子。

12.3.4 使用 AMCT 工具压缩模型

12.3.4.1 基于精度的自动量化

基于精度的自动量化是为了方便用户在对量化精度有一定要求时所使用的功能，是借助AMCT工具提供的Python接口实现。该方法能够在保证用户所需的模型精度前提下，自动搜索模型的量化配置并执行**训练后量化**的流程，最终生成满足精度要求的量化模型。

当前仅如下框架支持使用基于精度的自动量化：

- **PyTorch**框架，详细说明请参见《AMCT工具（PyTorch）》。
- **TensorFlow**框架，详细说明请参见《AMCT工具（TensorFlow）》。
- **Caffe**框架，详细说明请参见《AMCT工具（Caffe）》。

12.3.4.2 关于增加校准集的建议

在基于精度的自动量化时，手动修改量化配置中的batch_num，可根据batch大小以及量化需要使用的图片数量调整，以便调整校准使用的数据量。batch_num控制量化使用数据的batch数目，batch_num = total_nums / batch_size，其中，total_nums表示总的图片数量，batch_size表示每个batch使用的图片数量。

通常情况下：

batch_num越大，量化过程中使用的数据样本越多，量化后精度损失越小；但过多的数据并不会带来精度的提升，反而会占用较多的内存，降低量化的速度，并可能引起内存、显存、线程资源不足等情况；因此，建议batch_num*batch_size为16或32。

12.3.4.3 查找量化场景下的精度损失层

当前**仅支持Caffe**框架模型，使用精度比对工具查找量化场景下的精度损失层，主要包含以下两步的比对：

1. **定位量化阶段的精度问题**。
   执行非量化原始模型（GPU/CPU） vs 量化原始模型（GPU/CPU）。
2. **定位模型转换阶段产生的精度问题**，即量化离线模型在NPU上运行时的精度问题。
   执行量化原始模型（GPU/CPU） vs 量化离线模型（关闭融合规则）（NPU）。

详细操作请参见《精度比对工具使用指南》中的“GPU/CPU vs Ascend NPU精度比对（Caffe推理）”。

12.3.4.4 配置 skip_layers 跳过量化损失大的层

使用精度比对工具比对原始模型、量化后模型各层的精度，查找精度损失大的层，在基于精度的自动量化时通过配置skip_layers跳过量化损失大的层。

步骤1 修改基于精度的自动量化代码，增加skip_layers配置，跳过不支持量化的层。

代码示例如下，表示跳过某一层。如果涉及跳过多层，则每一层之间用逗号分隔，例如：skip_layers=['op1','op2','op3']

```
config_json_file = './config.json'
skip_layers = ['Default/network-DeepLabV3/resnet-Resnet/layer4-SequentialCell/0-Bottleneck/downsample-SequentialCell/0-Conv2d/Conv2D-op311']
batch_num = 1
amct.create_quant_config(config_json_file, model, input_data,
                         skip_layers, batch_num)

scale_offset_record_file = os.path.join(TMP, 'scale_offset_record.txt')
result_path = os.path.join(RESULT, 'model')
```

步骤2 执行精度的自动量化。

步骤3 重新执行推理。

如果跳过不支持量化的层影响模型推理的结果数据，则需要用户自行调整模型，再重新量化模型。

----结束

12.4 NN 类算子性能提升建议

12.4.1 使用静态 Kernel 提升模型执行性能

12.4.1.1 基本介绍

使用前须知

使用场景

对于纯静态shape网络或者shape变化较少的动态shape网络，如果您想快速提升网络模型执行性能，可以在网络部署时通过算子编译工具（op_compiler）编译生成静态Kernel包来提升算子调用性能。

调优原理

静态Kernel编译是指在编译时指定算子shape大小，运行时不需要指定shape大小。算子编译工具根据输入的算子信息统计文件，得到确定的shape信息，针对每一个shape都编译出一个算子二进制，从而实现提升算子执行效率和性能的目的。

静态Kernel编译的优势如下：

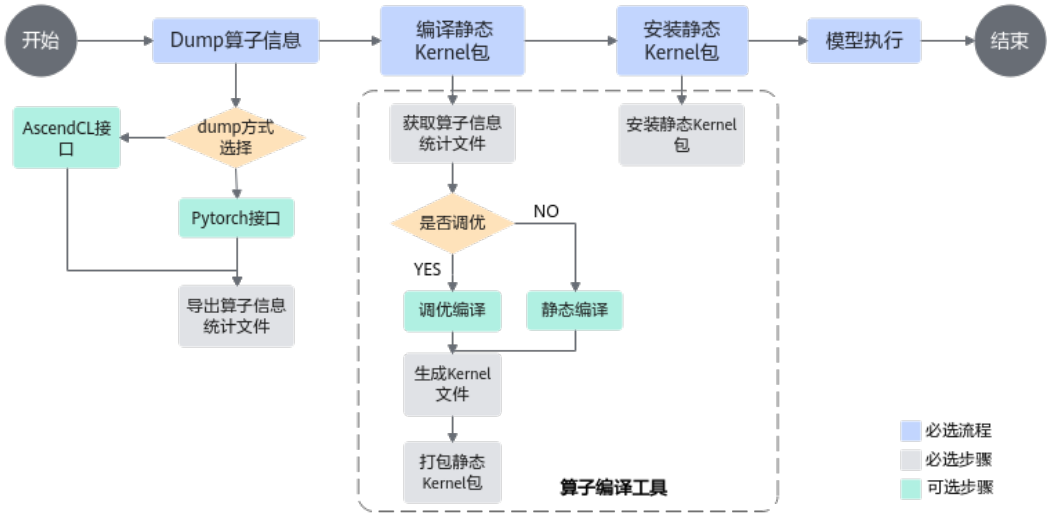
- 编译时已知所有tensor的大小，存储空间利用率高。
- 编译时可以针对实际的shape大小做针对性优化。
- AI处理器擅长并行指令运行，不擅长逻辑计算，如果有太多的Scalar操作可能会打断并行指令的运行，从而导致性能下降。静态编译可以在编译时完成标量的计算，一定程度上可以提升性能。
- 编译工具在编译时知道确切的操作数据大小，不会额外插入同步，不会导致并行执行多个指令变成串行执行，一定程度上可以提升性能。

使用约束

- 目前仅支持静态编译和调优编译模式。
- 静态编译模式支持的AI处理器型号：
  - **Atlas 训练系列产品**
  - **Atlas A2训练系列产品**
- 调优编译模式相关的约束：
  - 支持的AI处理器型号：**Atlas A2训练系列产品**
  - 不支持不同用户同时使用同一device进行调优。
  - 调优前，请确保关闭Profiling功能，避免影响调优结果。关闭Profiling功能具体操作请参见《性能分析工具使用指南》。

整体流程

图 12-9 使用静态 Kernel 提升性能的原理图



通过编译静态Kernel提升网络模型中算子执行性能的基本流程如图12-9所示，整个调优步骤如下：

1. Dump算子信息。

   对算子调优前，需要先获取网络模型中的算子信息。

   - 方式1：若采用PyTorch的Python接口编程，通过**Ascend PyTorch Profiler接口**Dump算子json文件。
   - 方式2：若采用AscendCL的C++接口编程，直接调用aclopStartDumpArgs和aclopStopDumpArgs接口Dump算子json文件。

2. 编译静态Kernel包。

   通过**算子编译工具**对Dump的算子信息统计文件（*.json)进行编译并生成Kernel包。

   说明

   算子编译工具是昇腾CANN提供的用于进行算子编译生成算子二进制文件的命令行工具。当算子shape固定或者变化较少时，可使用该工具编译静态kernel包并安装，提升算子调用的性能。

   a. 选择编译模式。

      算子编译工具默认是**静态编译**模式。如果想要进一步提升算子性能，可尝试进行算子调优，即开启“tune”**调优编译**模式。

   b. 打包静态Kernel包。

      算子编译工具将编译生成的Kernel文件打包成run包。

3. 安装静态Kernel包。

   将静态Kernel包上传到目标网络模型运行的服务器上，直接运行run包完成安装。

4. 模型执行效果验证。

   安装静态Kernel包后，重新运行目标网络模型，比较安装静态Kernel包和未安装静态Kernel包的整网运行性能和单算子运行耗时。

12.4.1.2 环境准备

软件包安装

步骤1 根据表12-3准备软件包。

表 12-3 软件包列表

组件	简介
Toolkit	开发、调测、调优工具包。主要包括算子工具、模型工具、应用工具。
Firmware	固件包。设备上的UEFI等基础固件，通常在device生产过程烧入该软件包，但也可以在后期通过安装该包实现固件版本升级。
Driver	驱动包。用于承载Host和Device之间的交互、调度、传输等，包含设备管理、查询驱动，图执行任务调度驱动，训练数据传输预处理驱动，AICPU算子加载执行驱动等。
PyTorch（可选）	当使用PyTorch的Python接口编程时，需安装PyTorch框架相关的软件。 **说明** 您可以单击Gitee，在“Ascend/pytorch”仓下载最新PyTorch软件。注意，版本建议是v5.0.rc2.1-pytorch1.11.0及以上版本，否则影响后续Dump算子json文件能力。

步骤2 安装软件包。

请参考《CANN软件安装指南》完成驱动、固件、开发套件包Ascend-cann-toolkit和AI框架PyTorch（可选）安装。

步骤3 配置环境变量。因为算子编译工具依赖AOE，所以要配置CANN软件基础环境变量和AOE工具所需的环境变量。

- CANN软件基础环境变量

  CANN组合包提供进程级环境变量设置脚本，供用户在进程中引用，以自动完成环境变量设置。执行命令参考如下，以下示例均为root或非root用户默认安装路径，请以实际安装路径为准。

  ```
  # 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
  ```

- AOE工具依赖Python，以Python3.7.5为例，请以运行用户执行如下命令设置Python3.7.5的相关环境变量。

  ```
  #用于设置python3.7.5库文件路径
  export LD_LIBRARY_PATH=/usr/local/python3.7.5/lib:$LD_LIBRARY_PATH
  #如果用户环境存在多个python3版本，则指定使用python3.7.5版本
  export PATH=/usr/local/python3.7.5/bin:$PATH
  ```

  Python3.7.5安装路径请根据实际情况进行替换，您也可以将以上命令写入~/.bashrc文件中，然后执行**source ~/.bashrc**命令使其立即生效。

- 调优前也可参考如下示例配置其他环境变量，但为可选配置，相关说明请参考表12-4。


```
export ASCEND_DEVICE_ID=0
export TUNE_BANK_PATH=/home/HwHiAiUser/custom_tune_bank
export TE_PARALLEL_COMPILER=8
export REPEAT_TUNE=False
```

说明

用户可将设置环境变量的命令写入自定义脚本，方便后续执行。

表 12-4 环境变量说明

环境变量	说明
ASCEND_DEVICE_ID	通过该环境变量指定昇腾AI处理器的逻辑ID。 取值范围[0,N-1]，默认为0。其中N为当前物理机/虚拟机/容器内的设备总数。
TUNE_BANK_PATH	可通过此环境变量指定调优后自定义知识库的存储路径。 设置的存储路径必须为绝对路径或相对于执行AOE调优引擎所在路径的相对路径，配置的路径需要为已存在的目录，且执行用户具有读、写、可执行权限。若配置的TUNE_BANK_PATH路径不存在或用户无权限，则调优进程会报错并退出。 自定义知识库存放路径的优先级为：TUNE_BANK_PATH>ASCEND_CACHE_PATH>默认，TUNE_BANK_PATH和ASCEND_CACHE_PATH详细信息请参考《环境变量参考》。 – 算子自定义知识库 ■ 若不配置此环境变量，请使用**env**命令查询ASCEND_CACHE_PATH是否存在，若存在，自定义知识库存储在：${ASCEND_CACHE_PATH}/aoe_data/${soc_version}；若不存在，自定义知识库默认存储在：${HOME}/Ascend/latest/data/aoe/custom/op/${soc_version}。 ■ 若配置此环境变量，则调优后的最优策略存储在配置路径的${soc_version}。 **说明** 在多用户共享知识库场景下，共享知识库的用户需要设置TUNE_BANK_PATH为同一路径，并且对配置的路径具有读、写权限。 若调优时自定义了知识库路径，后续进行模型转换时，若想直接使用自定义知识库，也需要配置此环境变量。
TE_PARALLEL_COMPILER	算子编译所需环境变量。 网络模型较大时，可通过配置此环境变量，开启算子的并行编译功能。 TE_PARALLEL_COMPILER的值代表算子编译进程数（配置为整数），当取值大于1时开启算子的并行编译功能。开启AOE调优场景下：配置不能超过CPU核数*80%/昇腾AI处理器的个数，取值范围：1~32，默认值为8。 由于该环境变量能够加速算子编译，所以可以加快涉及算子编译的相关流程调优。

环境变量	说明
REPEAT_TUNE	是否重新发起调优，此环境变量在开启子图调优或算子调优的场景下生效。 如果知识库（内置或者自定义）中已经存在网络模型中的调优case（针对某shape的调优策略），则会跳过此case的调优流程，若想重新发起调优，可设置此环境变量为True。例如某些算子进行了逻辑的变更，如GEMM算子新增了支持ND的输入，该情况下需要设置此环境变量后，重新发起调优。 取值范围：True或者False，默认值为False。

----结束

工具获取

工具所在目录：“${INSTALL_DIR}/compiler/bin/op_compiler”。

${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。例如，若安装的Ascend-cann-toolkit软件包，则安装后文件存储路径为：$HOME/Ascend/ascend-toolkit/latest。。

12.4.1.3 算子调优

步骤1 Dump算子信息。

算子调优前，需要先获取模型中算子信息统计文件（*.json），包括算子的shape、dtype、format等信息。目前支持两种方式Dump算子json文件，请根据实际情况选择合适的方式。

- **使用PyTorch的Python接口编程时**，可通过Ascend PyTorch Profiler接口dump算子json文件，具体请参考《PyTorch模型迁移和训练指南》中“性能调优 > Profiling数据采集及分析 >（推荐）Ascend PyTorch Porfiler数据采集与分析”章节。
  a. 使用Ascend PyTorch Profiler接口开启PyTorch训练时的性能数据采集。
     在训练前，开启扩展参数experimental_config中“算子信息统计功能”，即参数**record_op_args**置为True。
  b. 查看采集到的PyTorch训练性能数据结果文件。
     训练结束后，Dump的算子信息统计文件默认在{worker_name}_{时间戳}_ascend_pt_op_args/{pid}目录下。
- **使用AscendCL的C++接口编程时**，可通过aclopStartDumpArgs和aclopStopDumpArgs接口将算子信息统计文件Dump到指定目录下。

步骤2 编译静态Kernel包。

在任意目录下，以运行用户（如HwHiAiUser）身份执行如下命令，进行算子编译：

- **（默认）静态编译**命令样例：
```
op_compiler --op_params_dir=<dump_dir>  --soc_version=<soc_version> --log=info --job=128 --output=<output_dir>
op_compiler -p <dump_dir>  -v <soc_version> -l info -j 128 -o <output_dir>
```
- **调优编译**命令样例：

```
op_compiler --op_params_dir=<dump_dir>  --soc_version=<soc_version> --log=info --job=128 --compile_mode=tune --output=<output_dir>
op_compiler -p <dump_dir>  -v <soc_version> -l info -j 128 -m tune -o <output_dir>
```

关键参数释义如下，请根据实际情况设置。

- --op_params_dir：必选参数，简写-p，Dump工具导出的统计数据所在的文件夹路径，支持绝对和相对路径。
- --soc_version：执行算子编译功能时为必选参数，简写-v，指定算子编译时昇腾AI处理器的版本。

  说明

  如果无法确定当前设备的soc_version，则在安装昇腾AI处理器的服务器执行**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*。
- --log：可选参数，简写-l，设置算子编译过程中日志的级别。默认为null，即不输出日志。如需输出日志，可设置debug、info、warning、error级别。

  说明

  - debug：输出debug、info、warning、error、event级别的运行信息。
  - info：输出info、warning、error、event级别的运行信息。
  - warning：输出warning、error、event级别的运行信息。
  - error：输出error、event级别的运行信息。
- --job：可选参数，简写-j，设置编译时工作进程数。默认为2*cpu最大物理核数-1，最小取值1。
- --complie_mode：可选参数，简写-m，指定编译的模式。默认为compile纯编译模式，如需调优编译，参数取值tune。
- --output：可选参数，简写-o，编译输出的目录+安装包名称，如xxx/xxx/*.run，支持相对路径和绝对路径。

  不输入路径的情况下，默认在当前编译命令执行路径下生成；不输入安装包名称的情况下，安装包默认命名为static_kernel_${datatime}.run。

说明

算子编译工具提供了**--count**参数和-p参数配合使用，用于统计-p参数指定的目录下算子信息统计json文件的数目。

样例如下：

```
op_compiler -p <dump_dir> --count
```

只有动态shape才能dump出算子统计信息，安装静态kernel包后，静态kernel包对应算子的统计信息就不会dump出来。所以在安装静态kernel包后，如果网络有调整，可以通过调整前后dump的json文件的数量来判断静态kernel包和当前网络是否匹配。

通过调整网络前后，各执行一次dump操作，并通过--count命令来统计dump生成的json文件的数目，如果调整后的数目比调整前大，则说明静态kernel包中有部分算子不再匹配当前网络，此时开发者可以：

- 卸载静态kernel包，重新走dump流程，编译安装新的静态kernel包。
- 仍使用当前静态kernel包，此时需要注意不匹配的算子会走动态流程，得不到性能收益。

步骤3 安装静态Kernel包。

进入static_kernel_${datatime}.run包所在目录，以运行用户（如HwHiAiUser）身份运行run包：

```
./static_kernel_${datatime}.run
```

当出现如下回显信息代表安装成功。


```
Verifying archive integrity...  100%   SHA256 checksums are OK. All good.
Uncompressing STATIC KERNEL RUN PACKAGE  100%
```

目前暂不支持指定目录安装，run包默认安装到${install_path}/opp/static_kernel路径下，其中${install_path}为CANN软件安装后文件存储路径，请根据实际情况替换该路径。

run包安装后的目录结构样例如下，

```
|-- ${install_path}/opp/static_kernel
   |-- ai_core
      |-- config
        |-- ascendxxxx
            |-- binary_info_config.json           # 全量静态Kernel包的总索引
      |-- config.ini                              # 记录安装顺序的配置文件。
      |-- static_kernel_230808110316              # 时间戳为“230808110316”的静态Kernel文件
        |-- ascendxxxx
        |   |-- Add                               # 算子二进制目录
        |       |-- Add_float16_NCL_xxxx_d0.json
        |       |-- Add_float16_NCL_xxxx_d1.json
        |       |-- Add_float16_NCL_xxxx_d0.o
        |       |-- Add_float16_NCL_xxxx_d1.o
        |   |-- xxxx
        |       |-- xxx.json
        |       |-- xxx.o
        |   |-- ......
        |-- config                                # 单个静态Kernel包索引
        |   |-- ascendxxxx
        |       |-- binary_info_config.json
        |-- scripts                               # 工具涉及的通用脚本
        |   |-- ......
        |-- uninstall.sh                          # 单包卸载脚本
      |-- static_kernel_xxxx                      # 不同时间戳的静态Kernel文件
      |-- uninstall.sh                            # 全量卸载脚本
      |-- version.info                            # 版本信息
```

说明

支持多个Kernel包安装，如果多个包中存在相同的算子Kernel，以后安装的Kernel包为准。

步骤4 （可选）当不再需要静态Kernel包时，可以单包卸载或全量卸载。

- 单包卸载

  进入static_kernel_${datatime}.run包的安装目录，以运行用户（如HwHiAiUser）身份运行uninstall.sh。

  ```
  cd ${install_path}/opp/static_kernel/ai_core/static_kernel_${datatime}
  ./uninstall.sh
  ```

  此时，ai_core目录下static_kernel_${datatime}文件夹被删除。

- 全量卸载

  进入${install_path}/opp/static_kernel/ai_core目录下，以运行用户（如HwHiAiUser）身份运行uninstall.sh。

  ```
  cd ${install_path}/opp/static_kernel/ai_core/
  ./uninstall.sh
  ```

  此时，ai_core目录下所有内容被删除，所有已安装的kernel包均被卸载。

----结束

12.5 DVPP 数据处理高性能编程建议

目前在处理图像、视频数据时，可以使用VPC多功能组合接口、VPC批处理接口、合理选择VDEC的输出格式等编码方式，来提升应用的性能，具体编程建议请参见**9.11 高性能编程建议**。

13 应用样例参考

13.1 样例列表(Atlas 200/300/500 推理产品)

本文中提及的样例如下表所示。单击Gitee或Github获取更多样例。

表 13-1 Sample 列表

Sample名称	Sample获取	基本功能	编译运行指导（Ascend EP标准形态）（Ascend RC形态）
gemm	单击gemm获取样例	实现矩阵-矩阵乘运算	请参见样例工程中的README
vpc_resnet50_imagenet_classification	单击vpc_resnet50_imagenet_classification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）	请参见样例工程中的README
vpc_jpeg_resnet50_imagenet_classification	单击vpc_jpeg_resnet50_imagenet_classification获取样例	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）	请参见样例工程中的README
vdec_resnet50_classification	单击vdec_resnet50_classification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）	请参见样例工程中的README
resnet50_imagenet_classification	单击resnet50_imagenet_classification获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	请参见样例工程中的README
resnet50_async_imagenet_classification	单击resnet50_async_imagenet_classification获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	请参见样例工程中的README
batchcrop	单击batchcrop获取样例	媒体数据处理V1（抠图，一图多框）	请参见样例工程中的README
venc_image	单击venc_image获取样例	媒体数据处理V1（视频编码）	请参见样例工程中的README
smallResolution_cropandpaste	单击smallResolution_cropandpaste获取样例	媒体数据处理V1（抠图贴图）	请参见样例工程中的README

Sample名称	Sample获取	基本功能	编译运行指导（Ascend EP标准形态）（Ascend RC形态）
YOLOV3_dynamic_batch_detection_picture	单击YOLOV3_dynamic_batch_detection_picture获取样例	基于Caffe YOLOv3网络实现目标检测（动态Batch/动态分辨率）	请参见样例工程中的README
API Samples	API Samples	基于AscendCL架构开发的一系列样例代码，包含简单的编译演示样例	下载样例后查看readme
图像目标检测	图像目标检测	基于AscendCL架构的目标检测开发Demo	下载样例后查看readme

13.2 样例列表(Atlas 推理系列产品（Ascend 310P 处理器）)

本文中提及的样例如下表所示。单击Gitee或Github获取更多样例。

表 13-2 Sample 列表

Sample名称	Sample获取	基本功能	编译运行指导（Ascend EP标准形态）（Ascend RC形态）
gemm	单击gemm获取样例	实现矩阵-矩阵乘运算	请参见样例工程中的README
vpc_resnet50_imagenet_classification	单击vpc_resnet50_imagenet_classification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）	请参见样例工程中的README
vpc_jpeg_resnet50_imagenet_classification	单击vpc_jpeg_resnet50_imagenet_classification获取样例	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）	请参见样例工程中的README
vdec_resnet50_classification	单击vdec_resnet50_classification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）	请参见样例工程中的README

Sample名称	Sample获取	基本功能	编译运行指导（Ascend EP标准形态）（Ascend RC形态）
resnet50_imagenet_classification	单击**resnet50_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	请参见样例工程中的README
resnet50_async_imagenet_classification	单击**resnet50_async_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	请参见样例工程中的README
batchcrop	单击**batchcrop**获取样例	媒体数据处理V1（抠图，一图多框）	请参见样例工程中的README
venc_image	单击**venc_image**获取样例	媒体数据处理V1（视频编码）	请参见样例工程中的README
smallResolution_cropandpaste	单击**smallResolution_cropandpaste**获取样例	媒体数据处理V1（抠图贴图）	请参见样例工程中的README
YOLOV3_dynamic_batch_detection_picture	单击**YOLOV3_dynamic_batch_detection_picture**获取样例	基于Caffe YOLOv3网络实现目标检测（动态Batch/动态分辨率）	请参见样例工程中的README
vpc_sample	单击**vpc_sample**获取样例	媒体数据处理V2（VPC抠图\贴图\缩放等）	请参见样例工程中的README
jpegd_sample	单击**jpegd_sample**获取样例	媒体数据处理V2（JPEG图片解码）	请参见样例工程中的README
jpege_sample	单击**jpege_sample**获取样例	媒体数据处理V2（JPEG图片编码）	请参见样例工程中的README
vdec_sample	单击**vdec_sample**获取样例	媒体数据处理V2（VDEC视频解码）	请参见样例工程中的README
venc_sample	单击**venc_sample**获取样例	媒体数据处理V2（VENC视频编码）	请参见样例工程中的README
pngd_sample	单击**pngd_sample**获取样例	媒体数据处理V2（PNGD图片解码）	请参见样例工程中的README

13.3 样例列表(Atlas 200/500 A2 推理产品)

本文中提及的样例如下表所示。单击**Gitee**或**Github**获取更多样例。

表 13-3 Sample 列表

Sample名称	Sample获取	基本功能	编译运行指导
gemm	单击**gemm**获取样例	实现矩阵-矩阵乘运算	请参见样例工程中的README
vpc_jpeg_resnet50_imagenet_classification	单击**vpc_jpeg_resnet50_imagenet_classification**获取样例	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）	请参见样例工程中的README
vdec_resnet50_classification	单击**vdec_resnet50_classification**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）	请参见样例工程中的README
resnet50_imagenet_classification	单击**resnet50_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	请参见样例工程中的README
resnet50_async_imagenet_classification	单击**resnet50_async_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	请参见样例工程中的README
batchcrop	单击**batchcrop**获取样例	媒体数据处理V1（抠图，一图多框）	请参见样例工程中的README
venc_image	单击**venc_image**获取样例	媒体数据处理V1（视频编码）	请参见样例工程中的README
smallResolution_cropandpaste	单击**smallResolution_cropandpaste**获取样例	媒体数据处理V1（抠图贴图）	请参见样例工程中的README
YOLOV3_dynamic_batch_detection_picture	单击**YOLOV3_dynamic_batch_detection_picture**获取样例	基于Caffe YOLOv3网络实现目标检测（动态Batch/动态分辨率）	请参见样例工程中的README
vpc_sample	单击**vpc_sample**获取样例	媒体数据处理V2（VPC抠图\贴图\缩放等）	请参见样例工程中的README
jpegd_sample	单击**jpegd_sample**获取样例	媒体数据处理V2（JPEG图片解码）	请参见样例工程中的README
jpege_sample	单击**jpege_sample**获取样例	媒体数据处理V2（JPEG图片编码）	请参见样例工程中的README
vdec_sample	单击**vdec_sample**获取样例	媒体数据处理V2（VDEC视频解码）	请参见样例工程中的README

Sample名称	Sample获取	基本功能	编译运行指导
venc_sample	单击**venc_sample**获取样例	媒体数据处理V2（VENC视频编码）	请参见样例工程中的README
pngd_sample	单击**pngd_sample**获取样例	媒体数据处理V2（PNGD图片解码）	请参见样例工程中的README

13.4 样例列表(Atlas 训练系列产品)

本文中提及的样例如下表所示。单击**Gitee**或**Github**获取更多样例。

表 13-4 Sample 列表

Sample名称	Sample获取	基本功能	编译运行
gemm	单击**gemm**获取样例	实现矩阵-矩阵乘运算	请参见样例工程中的README
vpc_resnet50_imagenet_classification	单击**vpc_resnet50_imagenet_classification**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）	请参见样例工程中的README
vpc_jpeg_resnet50_imagenet_classification	单击**vpc_jpeg_resnet50_imagenet_classification**获取样例	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）	请参见样例工程中的README
vdec_resnet50_classification	单击**vdec_resnet50_classification**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）	请参见样例工程中的README
resnet50_imagenet_classification	单击**resnet50_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（同步推理）	请参见样例工程中的README
resnet50_async_imagenet_classification	单击**resnet50_async_imagenet_classification**获取样例	基于Caffe ResNet-50网络实现图片分类（异步推理）	请参见样例工程中的README
batchcrop	单击**batchcrop**获取样例	媒体数据处理V1（抠图，一图多框）	请参见样例工程中的README

Sample名称	Sample获取	基本功能	编译运行
smallResolution_cropandpaste	单击smallResolution_cropandpaste获取样例	媒体数据处理V1（抠图贴图）	请参见样例工程中的README
YOLOV3_dynamic_batch_detection_picture	单击YOLOV3_dynamic_batch_detection_picture获取样例	基于Caffe YOLOv3网络实现目标检测（动态Batch/动态分辨率）	请参见样例工程中的README

13.5 样例列表(Atlas A2 训练系列产品)

本文中提及的样例如下表所示。单击Gitee或Github获取更多样例。

表 13-5 Sample 列表

Sample名称	Sample获取	基本功能	编译运行指导
gemm	单击gemm获取样例	实现矩阵-矩阵乘运算	请参见样例工程中的README
batchcrop	单击batchcrop获取样例	媒体数据处理V1（抠图，一图多框）	请参见样例工程中的README
smallResolution_cropandpaste	单击smallResolution_cropandpaste获取样例	媒体数据处理V1（抠图贴图）	请参见样例工程中的README
vpc_sample	单击vpc_sample获取样例	媒体数据处理V2（VPC抠图\贴图\缩放等）	请参见样例工程中的README
jpegd_sample	单击jpegd_sample获取样例	媒体数据处理V2（JPEG图片解码）	请参见样例工程中的README
jpege_sample	单击jpege_sample获取样例	媒体数据处理V2（JPEG图片编码）	请参见样例工程中的README
vdec_sample	单击vdec_sample获取样例	媒体数据处理V2（VDEC视频解码）	请参见样例工程中的README
venc_sample	单击venc_sample获取样例	媒体数据处理V2（VENC视频编码）	请参见样例工程中的README

Sample名称	Sample获取	基本功能	编译运行指导
pngd_sample	单击**pngd_sample**获取样例	媒体数据处理V2（PNGD图片解码）	请参见样例工程中的README

13.6 实现矩阵-矩阵乘运算

13.6.1 样例介绍

获取样例

单击**gemm**获取样例

功能描述

该样例主要实现矩阵-矩阵相乘的运算：C = αAB + βC，A、B、C都是16*16的矩阵，即：m=16，n=16，k=16。矩阵乘的结果是一个16*16的矩阵。

图 13-1 Sample 示例

$$\text{矩阵 A=}\begin{pmatrix} a11 & a12 & a13 & \cdots & a1k \\ a21 & a22 & a23 & \cdots & a2k \\ a31 & a32 & a33 & \cdots & a3k \\ \cdots & \cdots & \cdots & & \cdots \\ am1 & am2 & am3 & \cdots & amk \end{pmatrix}, B=\begin{pmatrix} b11 & b12 & b13 & \cdots & b1n \\ b21 & b22 & b23 & \cdots & b2n \\ b31 & b32 & b33 & \cdots & b3n \\ \cdots & \cdots & \cdots & & \cdots \\ bk1 & bk2 & bk3 & \cdots & bkn \end{pmatrix}, C=\begin{pmatrix} C11 & C12 & C13 & \cdots & C1n \\ C21 & C22 & C23 & \cdots & C2n \\ C31 & C32 & C33 & \cdots & C3n \\ \cdots & \cdots & \cdots & & \cdots \\ Cm1 & Cm2 & Cm3 & \cdots & Cmn \end{pmatrix}$$

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

表 13-6 关键功能点

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。

Device管理	● 调用aclrtSetDevice接口指定用于运算的Device。 ● 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 ● 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Stream管理	● 调用aclrtCreateStream接口创建Stream。 ● 调用aclrtDestroyStream接口销毁Stream。 ● 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	● 调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
单算子调用	● 调用aclblasGemmEx接口实现矩阵-矩阵相乘的运算，由用户指定矩阵中元素的数据类型。在aclblasGemmEx接口内部封装了系统内置的矩阵乘算子GEMM。 ● 使用ATC（Ascend Tensor Compiler）工具将内置的矩阵乘算子GEMM的算子描述信息（包括输入输出Tensor描述、算子属性等）编译成适配昇腾AI处理器的离线模型（*.om文件），用于验证矩阵乘算子GEMM的运行结果。

目录结构

样例代码结构如下所示。

```
├── inc
│   ├── common.h                //定义公共函数（例如：文件读取函数）的头文件
│   ├── gemm_runner.h           //定义矩阵乘运算相关函数的头文件

├── run
│   ├── out
│   │   ├──test_data
│   │   │   ├── config
│   │   │   │   ├── acl.json          //系统初始化的配置文件
│   │   │   │   ├── gemm.json         //矩阵乘算子的算子描述信息
│   │   │   ├── data
│   │   │   │   ├── generate_data.py  //用于生成矩阵A、矩阵B的数据

├── src
│   ├── CMakeLists.txt          //编译脚本
│   ├── common.cpp              //公共函数（例如：文件读取函数）的实现文件
│   ├── gemm_main.cpp           //主函数的实现文件
│   ├── gemm_runner.cpp         //执行矩阵乘运算相关函数的实现文件
```

13.6.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**gemm**获取样例，查看该样例下的README。

13.6.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**gemm**获取样例，查看该样例下的README。

13.7 基于 Caffe ResNet-50 网络实现图片分类（图片解码+缩放+同步推理）

13.7.1 样例介绍

获取样例

单击**vpc_resnet50_imagenet_classification**获取样例

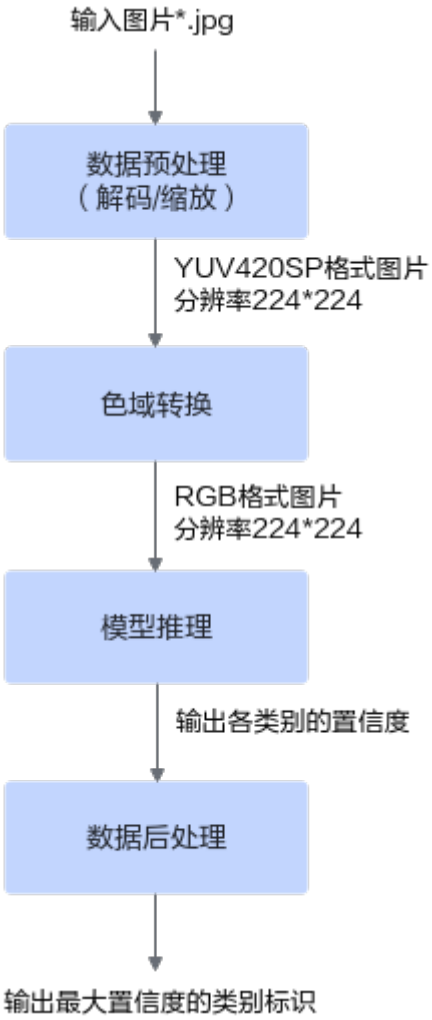
功能描述

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。

图 13-2 Sample 示例



原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。

Stream管理	● 调用aclrtCreateStream接口创建Stream。 ● 调用aclrtDestroyStream接口销毁Stream。 ● 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	● 调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用acldvppMalloc申请内存、调用acldvppFree接口释放内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V1	● 调用acldvppJpegDecodeAsync接口将*.jpg图片解码成YUV420SP格式图片。 ● 调用acldvppVpcResizeAsync接口将YUV420SP格式图片缩小成分辨率为224*224的图片。
模型推理	● 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 ● 调用aclmdlExecute接口执行模型推理。 推理前，通过*.om文件中的色域转换参数将YUV420SP格式的图片转换为RGB格式的图片。 ● 调用aclmdlUnload接口卸载模型。
数据后处理（单算子调用）	处理模型推理的结果，通过调用算子Cast将推理结果的数据类型从float32转成float16，再调用ArgMaxD算子从推理结果中查找最大置信度的类别标识。 通过aclopSetModelDir接口加载单算子模型文件，通过aclopCast接口执行Cast算子、通过aclopExecuteV2接口执行ArgMaxD算子。

目录结构

样例代码结构如下所示。

```
├── caffe_model
│   ├── aipp.cfg        //带色域转换参数的配置文件，模型转换时使用

├── data
│   ├── dog1_1024_683.jpg          //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── dog2_1024_683.jpg          //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── dvpp_process.h             //声明媒体数据处理相关函数的头文件
│   ├── model_process.h            //声明模型处理相关函数的头文件
│   ├── sample_process.h           //声明资源初始化/销毁相关函数的头文件
│   ├── singleOp_process.h         //声明单算子执行相关函数的头文件
```



```
│   ├── utils.h                    //声明公共函数（例如：文件读取函数）的头文件

├── out
│   ├── op_models
│   ├──├──op_list.json             //Cast算子和ArgMaxD算子的算子描述信息

├── src
│   ├── acl.json         //系统初始化的配置文件
│   ├── CMakeLists.txt         //编译脚本
│   ├── dvpp_process.cpp       //媒体数据处理相关函数的实现文件
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── model_process.cpp      //模型处理相关函数的实现文件
│   ├── sample_process.cpp     //资源初始化/销毁相关函数的实现文件
│   ├── singleOp_process.cpp   //单算子执行相关函数的实现文件
│   ├── utils.cpp              //公共函数（例如：文件读取函数）的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.7.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**vpc_resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.7.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**vpc_resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.8 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）

13.8.1 样例介绍

获取样例

单击**vpc_jpeg_resnet50_imagenet_classification**获取样例

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：

- 将一张YUV420SP格式的图片编码为*.jpg格式的图片。
- 将两张*.jpg格式的解码成两张YUV420SP NV12格式的图片，缩放，再进行模型推理，分别得到两张图片的推理结果后，处理推理结果，输出最大置信度的类别标识以及top5置信度的总和。
- 将两张*.jpg格式的解码成两张YUV420SP NV12格式的图片，抠图，再进行模型推理，分别得到两张图片的推理结果后，处理推理结果，输出最大置信度的类别标识以及top5置信度的总和。

- 将两张*.jpg格式的解码成两张YUV420SP NV12格式的图片，抠图贴图，再进行模型推理，分别得到两张图片的推理结果后，处理推理结果，输出最大置信度的类别标识以及top5置信度的总和。
- 将YUV420SP NV12格式的图片（分辨率8192*8192）缩放，得到4000*4000。

该样例中用于推理的模型文件是*.om文件（适配昇腾AI处理器的离线模型），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。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	● 调用aclInit接口初始化AscendCL配置。 ● 调用aclFinalize接口实现AscendCL去初始化。
Device管理	● 调用aclrtSetDevice接口指定用于运算的Device。 ● 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 ● 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	● 调用aclrtCreateContext接口创建Context。 ● 调用aclrtDestroyContext接口销毁Context。
Stream管理	● 调用aclrtCreateStream接口创建Stream。 ● 调用aclrtDestroyStream接口销毁Stream。 ● 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	● 调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用acldvppMalloc申请内存、调用acldvppFree接口释放内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。

媒体数据处理V1	• 图片编码 调用acldvppJpegEncodeAsync接口将YUV420SP格式的图片编码为*.jpg格式的图片。 • 图片解码 调用acldvppJpegDecodeAsync接口将*.jpg图片解码成YUV420SP格式图片。 • 缩放 调用acldvppVpcResizeAsync接口对YUV420SP格式的输入图片进行缩放。 • 抠图 调用acldvppVpcCropResizeAsync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。 • 抠图贴图 调用acldvppVpcCropResizePasteAsync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。
模型推理	• 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 • 调用aclmdlExecute接口执行模型推理。 推理前，通过*.om文件中的色域转换参数将YUV420SP格式的图片转换为RGB格式的图片。 • 调用aclmdlUnload接口卸载模型。

目录结构

样例代码结构如下所示。

```
├── caffe_model
│   ├── aipp.cfg        //带色域转换参数的配置文件，模型转换时使用

├── data
│   ├── persian_cat_1024_1536_283.jpg          //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── wood_rabbit_1024_1061_330.jpg          //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── wood_rabbit_1024_1068_nv12.yuv         //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── dvpp_vpc_8192x8192_nv12.yuv            //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── dvpp_process.h              //声明媒体数据处理相关函数的头文件
│   ├── model_process.h             //声明模型处理相关函数的头文件
│   ├── sample_process.h            //声明资源初始化/销毁相关函数的头文件
│   ├── utils.h                     //声明公共函数（例如：文件读取函数）的头文件

├── src
│   ├── acl.json        //系统初始化的配置文件
│   ├── CMakeLists.txt        //编译脚本
│   ├── dvpp_process.cpp      //媒体数据处理相关函数的实现文件
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── model_process.cpp     //模型处理相关函数的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```


13.8.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**vpc_jpeg_resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.8.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**vpc_jpeg_resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.9 基于 Caffe ResNet-50 网络实现图片分类（视频解码+同步推理）

13.9.1 样例介绍

获取样例

单击**vdec_resnet50_classification**获取样例

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，将1个*.h265格式的视频码流（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，再对10张YUV420SP NV12格式的图片进行推理，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。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。

Context管理	● 调用aclrtCreateContext接口创建Context。 ● 调用aclrtDestroyContext接口销毁Context。
Stream管理	● 调用aclrtCreateStream接口创建Stream。 ● 调用aclrtDestroyStream接口销毁Stream。 ● 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用acldvppMalloc申请内存、调用acldvppFree接口释放内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V1	● 视频解码 调用aclvdecSendFrame接口将视频码流解码成YUV420SP格式图片。 ● 缩放 调用acldvppVpcResizeAsync接口将YUV420SP NV12格式图片缩小成分辨率为224*224的图片。
模型推理	● 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 ● 调用aclmdlExecute接口执行模型推理。 推理前，通过*.om文件中的色域转换参数将YUV420SP格式的图片转换为RGB格式的图片。 ● 调用aclmdlUnload接口卸载模型。

目录结构

样例代码结构如下所示。

```
├── caffe_model
│   ├── aipp.cfg       //带色域转换参数的配置文件，模型转换时使用

├── data
│   ├── vdec_h265_1frame_rabbit_1280x720.h265          //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── dvpp_process.h            //声明媒体数据处理相关函数的头文件
│   ├── model_process.h           //声明模型处理相关函数的头文件
│   ├── sample_process.h            //声明资源初始化/销毁相关函数的头文件
│   ├── utils.h                   //声明公共函数（例如：文件读取函数）的头文件
│   ├── vdec_process.h            //声明视频处理函数的头文件
```



```
├── src
│   ├── acl.json        //系统初始化的配置文件
│   ├── CMakeLists.txt        //编译脚本
│   ├── dvpp_process.cpp      //媒体数据处理相关函数的实现文件
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── model_process.cpp     //模型处理相关函数的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件
│   ├── vdec_process.cpp              //声明视频处理函数的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.9.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**vdec_resnet50_classification**获取样例，查看该样例下的README。

13.9.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**vdec_resnet50_classification**获取样例，查看该样例下的README。

13.10 基于 Caffe ResNet-50 网络实现图片分类（同步推理）

13.10.1 样例介绍

获取样例

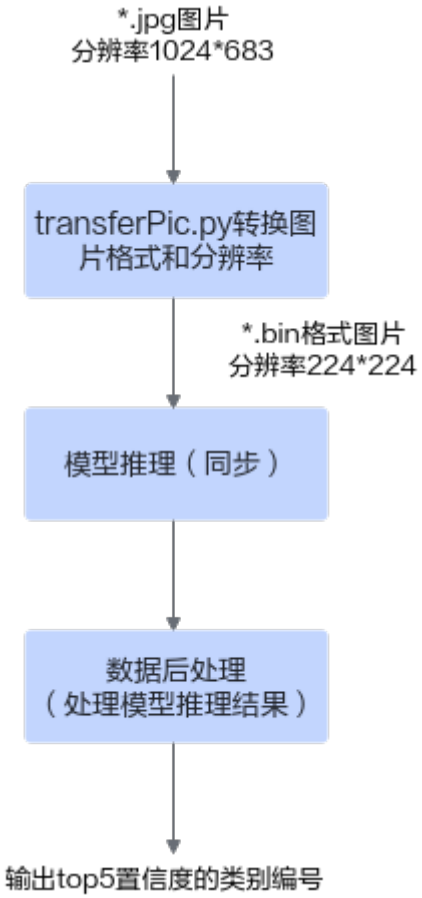
单击**resnet50_imagenet_classification**获取样例

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。

图 13-3 Sample 示例



原理介绍

在该Sample中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
Stream管理	• 调用aclrtCreateStream接口创建Stream。 • 调用aclrtDestroyStream接口销毁Stream。
内存管理	• 调用aclrtMalloc接口申请Device上的内存。 • 调用aclrtFree接口释放Device上的内存。

数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
模型推理	● 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 ● 调用aclmdlExecute接口执行模型推理，同步接口。 ● 调用aclmdlUnload接口卸载模型。

数据后处理	提供样例代码，处理模型推理的结果，直接在终端上显示top5置信度的类别编号。
	另外，样例中提供了自定义接口DumpModelOutputResult，用于将模型推理的结果写入文件（运行可执行文件后，推理结果文件在运行环境上的应用可执行文件的同级目录下），默认未调用该接口，用户可在sample_process.cpp中，在调用OutputModelResult接口前，增加如下代码调用DumpModelOutputResult接口：
	// print the top 5 confidence values with indexes.use function DumpModelOutputResult
	// if want to dump output result to file in the current directory
	modelProcess.DumpModelOutputResult();
	modelProcess.OutputModelResult();

目录结构

样例代码结构如下所示。

```
├── data
│   ├── dog1_1024_683.jpg           //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── dog2_1024_683.jpg           //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── model_process.h             //声明模型处理相关函数的头文件
│   ├── sample_process.h            //声明资源初始化/销毁相关函数的头文件
│   ├── utils.h                   //声明公共函数（例如：文件读取函数）的头文件

├── script
│   ├── transferPic.py              //将*.jpg转换为*.bin，同时将图片从1024*683的分辨率缩放为224*224

├── src
│   ├── acl.json        //系统初始化的配置文件
│   ├── CMakeLists.txt        //编译脚本
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── model_process.cpp     //模型处理相关函数的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.10.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.10.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**resnet50_imagenet_classification**获取样例，查看该样例下的README。

13.11 基于 Caffe ResNet-50 网络实现图片分类（异步推理）

13.11.1 样例介绍

获取样例

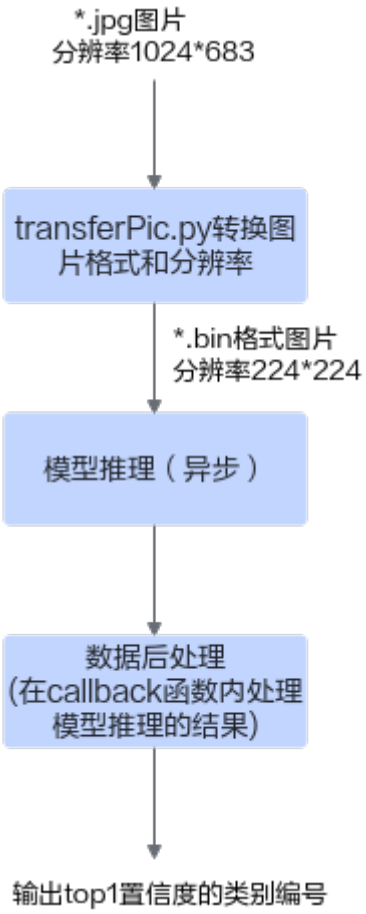
单击**resnet50_async_imagenet_classification**获取样例

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n次异步推理（n作为运行应用的参数，由用户配置），分别得到n次推理结果后，再对推理结果进行处理，输出top1置信度的类别标识。

图 13-4 Sample 示例



原理介绍

在该Sample中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtSetCurrentContext接口设置线程的Context。 • 调用aclrtDestroyContext接口销毁Context。
Stream管理	• 调用aclrtCreateStream接口创建Stream。 • 调用aclrtDestroyStream接口销毁Stream。

内存管理	● 调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
模型推理	● 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 ● 创建新线程（例如t1），在线程函数内调用aclrtProcessReport接口，等待指定时间后，触发回调函数（例如CallBackFunc，用于处理模型推理结果）。 ● 调用aclrtSubscribeReport接口，指定处理Stream上回调函数（CallBackFunc）的线程（t1）。 ● 调用aclmdlExecuteAsync接口执行模型推理，异步接口。 ● 调用aclrtLaunchCallback接口，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（CallBackFunc）。 ● 调用aclrtSynchronizeStream接口，阻塞应用程序运行，直到指定Stream中的所有任务都完成。 ● 调用aclrtUnSubscribeReport接口，取消线程注册，Stream上的回调函数（CallBackFunc）不再由指定线程（t1）处理。 ● 模型推理结束后，调用aclmdlUnload接口卸载模型。

数据后处理	提供样例代码，处理模型推理的结果，直接在终端上显示top1置信度的类别编号。
	另外，样例中提供了自定义接口DumpModelOutputResult，用于将模型推理的结果写入文件（运行可执行文件后，推理结果文件在运行环境上的应用可执行文件的同级目录下），默认未调用该接口，用户可在model_process.cpp中，在调用OutputModelResult接口前，增加如下代码调用DumpModelOutputResult接口：
	// OutputModelResult prints the top 1 confidence value with index.
	// If want to dump output result to file in the current directory,
	// use function DumpModelOutputResult.
	ModelProcess::DumpModelOutputResult(data.second);
	ModelProcess::OutputModelResult(data.second);

目录结构

样例代码结构如下所示。

```
├── data
│   ├── dog1_1024_683.jpg          //测试数据,需要按指导获取测试图片，放到data目录下
│   ├── dog2_1024_683.jpg          //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── memory_pool.h              //声明内存池处理相关函数的头文件
│   ├── model_process.h            //声明模型处理相关函数的头文件
│   ├── sample_process.h           //声明资源初始化/销毁相关函数的头文件
```



```
│   ├── utils.h                   //声明公共函数（例如：文件读取函数）的头文件

├── script
│   ├── transferPic.py             //将*.jpg转换为*.bin，同时将图片从1024*683的分辨率缩放为224*224

├── src
│   ├── acl.json        //系统初始化的配置文件
│   ├── CMakeLists.txt        //编译脚本
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── memory_pool.cpp       //内存池处理相关函数的实现文件
│   ├── model_process.cpp     //模型处理相关函数的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.11.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**resnet50_async_imagenet_classification**获取样例，查看该样例下的README。

13.11.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**resnet50_async_imagenet_classification**获取样例，查看该样例下的README。

13.12 媒体数据处理 V1（抠图，一图多框）

13.12.1 样例介绍

获取样例

单击**batchcrop**获取样例

功能描述

该样例从一张YUV420SP NV12格式的输入图片中按指定区域分别抠出八张224*224子图（YUV420SP NV12）。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。

Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
Stream管理	• 调用aclrtCreateStream接口创建Stream。 • 调用aclrtDestroyStream接口销毁Stream。 • 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	• 调用aclrtMallocHost接口申请Host上内存。 • 调用aclrtFreeHost释放Host上的内存。 • 调用aclrtMalloc接口申请Device上的内存。 • 调用aclrtFree接口释放Device上的内存。 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用acldvppMalloc申请内存、调用acldvppFree接口释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理	• 抠图 调用acldvppVpcBatchCropResizeAsync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。

目录结构

样例代码结构如下所示。

```
├── data
│   ├── dvpp_vpc_1920x1080_nv12.yuv          //测试数据,需要按指导获取测试图片，放到data目录下
├── inc
│   ├── dvpp_process.h             //声明媒体数据处理相关函数的头文件
│   ├── sample_process.h             //声明模型处理相关函数的头文件
│   ├── utils.h                    //声明公共函数（例如：文件读取函数）的头文件
├── src
│   ├── acl.json             //系统初始化的配置文件
│   ├── CMakeLists.txt         //编译脚本
│   ├── dvpp_process.cpp       //媒体数据处理相关函数的实现文件
│   ├── main.cpp             //主函数，一图多框抠图功能的实现文件
│   ├── sample_process.cpp     //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp            //公共函数（例如：文件读取函数）的实现文件
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.12.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**batchcrop**获取样例，查看该样例下的README。

13.12.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**batchcrop**获取样例，查看该样例下的README。

13.13 媒体数据处理 V1（视频编码）

13.13.1 样例介绍

获取样例

单击**venc_image**获取样例

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，默认为16次）。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用acldvppMalloc接口申请device上内存。 • 调用acldvppFree释放device上的内存。

数据传输	调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为编码的输入数据。 ● 编码结束后，将编码结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理	视频编码，调用aclvencSendFrame接口将将待编码的图片传到编码器进行编码。

目录结构

样例代码结构如下所示。

```
├── data
│   ├── dvpp_venc_128x128_nv12.yuv          //测试数据,需要按指导获取测试图片，放到data目录下
├── inc
│   ├── venc_process.h              //声明媒体数据处理相关函数的头文件
│   ├── sample_process.h              //声明模型处理相关函数的头文件
│   ├── utils.h                     //声明公共函数（例如：文件读取函数）的头文件
├── src
│   ├── acl.json              //系统初始化的配置文件
│   ├── CMakeLists.txt        //编译脚本
│   ├── main.cpp              //主函数，多图多框抠图功能的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件
│   ├── venc_process.cpp      //媒体数据处理相关函数的实现文件
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.13.2 Ascend EP，编译及在 Host 上运行应用（Atlas 200/300/500 推理产品）（Atlas 推理系列产品（Ascend 310P 处理器））

单击**venc_image**获取样例，查看该样例下的README。

13.13.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**venc_image**获取样例，查看该样例下的README。

13.14 媒体数据处理 V1（抠图贴图）

13.14.1 样例介绍

获取样例

单击**smallResolution_cropandpaste**获取样例

功能描述

该样例主要实现以下三种场景的抠图贴图：

1、抠图区域小于10*6。

2、贴图区域与抠图区域的缩放比例在[1/32, 16]范围内。

3、满足VPC分辨率要求的输入图片的抠图贴图，分辨率的要求，请参见约束说明。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	● 调用aclInit接口初始化AscendCL配置。 ● 调用aclFinalize接口实现AscendCL去初始化。
Device管理	● 调用aclrtSetDevice接口指定用于运算的Device。 ● 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 ● 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	● 调用aclrtCreateContext接口创建Context。 ● 调用aclrtDestroyContext接口销毁Context。
Stream管理	● 调用aclrtCreateStream接口创建Stream。 ● 调用aclrtDestroyStream接口销毁Stream。 ● 调用aclrtSynchronizeStream接口阻塞程序运行，直到指定Stream中的所有任务都完成。
内存管理	● 调用aclrtMallocHost接口申请Host上内存。 ● 调用aclrtFreeHost释放Host上的内存。 ● 调用aclrtMalloc接口申请Device上的内存。 ● 调用aclrtFree接口释放Device上的内存。 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用acldvppMalloc申请内存、调用acldvppFree接口释放内存。
数据传输	调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理	抠图贴图： 调用acldvppVpcCropAndPasteAsync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。

目录结构

样例代码结构如下所示。

```
├── data
│   ├── dvpp_vpc_1920x1080_nv12.yuv          //测试数据,需要按指导获取测试图片，放到data目录下

├── inc
│   ├── dvpp_process.h             //声明媒体数据处理相关函数的头文件
│   ├── sample_process.h             //声明模型处理相关函数的头文件
│   ├── utils.h                    //声明公共函数（例如：文件读取函数）的头文件

├── src
│   ├── acl.json             //系统初始化的配置文件
│   ├── CMakeLists.txt         //编译脚本
│   ├── dvpp_process.cpp       //媒体数据处理相关函数的实现文件
│   ├── main.cpp             //主函数，抠图贴图功能的实现文件
│   ├── sample_process.cpp     //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp            //公共函数（例如：文件读取函数）的实现文件

├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.14.2 Ascend EP，编译及在 Host 上运行应用（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）

单击**smallResolution_cropandpaste**获取样例，查看该样例下的README。

13.14.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击**smallResolution_cropandpaste**获取样例，查看该样例下的README。

13.15 基于 Caffe YOLOv3 网络实现目标检测（动态 Batch/动态分辨率）

13.15.1 样例介绍

获取样例

单击**YOLOV3_dynamic_batch_detection_picture**获取样例

功能描述

该样例主要是基于Caffe YOLOv3网络、在动态Batch或动态多分辨率场景下实现目标检测的功能。

将Caffe YOLOv3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），转换命令中需要设置不同档位的batch size（样例中batch档位分为1，2，4，8）或不同档位的分辨率（样例中分辨率档位分为416, 416；832，832；1248，1248），在应用中加载该om文件，通过传参设置选择不同档位的batch size或者分辨率进行推理，并将推理结果保存到文件中。

原理介绍

在该Sample中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
Stream管理	• 调用aclrtCreateStream接口创建Stream。 • 调用aclrtDestroyStream接口销毁Stream。
内存管理	• 调用aclrtMalloc接口申请Device上的内存。 • 调用aclrtFree接口释放Device上的内存。
数据传输	如果在Host上运行应用，则需调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
模型推理	• 调用aclmdlLoadFromFileWithMem接口从*.om文件加载模型。 • 调用aclmdlSetDynamicBatchSize设置batch size或者调用aclmdlSetDynamicHWSize设置分辨率。 • 调用aclmdlExecute接口执行模型推理，同步接口。 • 调用aclmdlUnload接口卸载模型。
数据后处理	样例中提供了自定义接口DumpModelOutputResult，用于将模型推理的结果写入文件（运行可执行文件后，推理结果文件在运行环境上的应用可执行文件的同级目录下）： **processModel.DumpModelOutputResult();**

目录结构

样例代码结构如下所示。

```
├── data
│   ├── tools_generate_data.py            //测试数据生成脚本
├── inc
│   ├── model_process.h                   //声明模型处理相关函数的头文件
│   ├── sample_process.h                  //声明资源初始化/销毁相关函数的头文件
│   ├── utils.h                           //声明公共函数（例如：文件读取函数）的头文件
├── src
│   ├── acl.json          //系统初始化的配置文件
```



```
│   ├── CMakeLists.txt        //编译脚本
│   ├── main.cpp              //主函数，图片分类功能的实现文件
│   ├── model_process.cpp     //模型处理相关函数的实现文件
│   ├── sample_process.cpp    //资源初始化/销毁相关函数的实现文件
│   ├── utils.cpp             //公共函数（例如：文件读取函数）的实现文件

├── .project    //工程信息文件，包含工程类型、工程描述、运行目标设备类型等
├── CMakeLists.txt    //编译脚本，调用src目录下的CMakeLists文件
```

13.15.2 Ascend EP，编译及在 Host 上运行应用（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）

单击YOLOV3_dynamic_batch_detection_picture获取样例，查看该样例下的README。

13.15.3 Ascend RC，编译及运行应用（Atlas 200/300/500 推理产品）

单击YOLOV3_dynamic_batch_detection_picture获取样例，查看该样例下的README。

13.16 媒体数据处理 V2（VPC 抠图/贴图/缩放等）

13.16.1 样例介绍

获取样例

单击vpc_sample获取样例

功能描述

该样例实现图片的抠图、缩放、边界填充、色域转换、金字塔、LUT重映射、直方图统计等功能。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。

Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见VPC图像处理功能中的接口。

目录结构

样例代码结构如下所示。

```
├── common                    //示例代码文件所在的目录
│   ├── sample_comm.cpp         //公共函数的实现文件
│   ├── sample_comm.h           //声明公共函数的头文件
│   ├── sample_comm_vpc_*.cpp   //抠图、缩放等函数的实现文件

├── smoke_vpc                 //示例代码文件所在的目录
│   ├── sample_vpc.cpp          //main函数的实现文件

├── CMakeLists.txt            //编译脚本
```

13.16.2 编译及运行应用

单击**vpc_sample**获取样例，查看该样例下的README。

13.17 媒体数据处理 V2（JPEGD 图片解码）

13.17.1 样例介绍

获取样例

单击**jpegd_sample**获取样例

功能描述

该样例实现.jpg、.jpeg、.JPG、.JPEG图片的解码。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见VDEC视频解码功能/JPEGD图片解码功能中的接口。

目录结构

样例代码结构如下所示。

```
├── common                    //示例代码文件所在的目录
│   ├── sample_comm.h           //声明解码函数的头文件
│   ├── sample_comm_jpegd.cpp   //解码函数的实现文件

├── smoke_vpc                 //示例代码文件所在的目录
│   ├── sample_jpegd.cpp        //main函数的实现文件

├── CMakeLists.txt            //编译脚本
```

13.17.2 编译及运行应用

单击**jpegd_sample**获取样例，查看该样例下的README。

13.18 媒体数据处理 V2（JPEGE 图片编码）

13.18.1 样例介绍

获取样例

单击**jpege_sample**获取样例

功能描述

该样例实现将YUV格式图片编码成.jpg图片。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见VENC视频编码功能/JPEGE图片编码功能中的接口。

目录结构

样例代码结构如下所示。

```
├── common                    //示例代码文件所在的目录
│   ├── sample_comm.h           //声明编码函数的头文件
│   ├── sample_comm_venc.cpp    //编码函数的实现文件

├── smoke_vpc                 //示例代码文件所在的目录
│   ├── sample_jpege.cpp        //main函数的实现文件

├── CMakeLists.txt            //编译脚本
```

13.18.2 编译及运行应用

单击**jpege_sample**获取样例，查看该样例下的README。

13.19 媒体数据处理 V2（VDEC 视频解码）

13.19.1 样例介绍

获取样例

单击**vdec_sample**获取样例

功能描述

该样例实现H264码流格式的视频解码。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见VDEC视频解码功能/JPEGD图片解码功能中的接口。

目录结构

样例代码结构如下所示。

```
├── Vdec.h          //声明解码函数的头文件
├── Vdec.cpp        //解码函数的实现文件
```

```
├── VdecDemo.cpp          //main函数的实现文件
├── CMakeLists.txt        //编译脚本
```

13.19.2 编译及运行应用

单击**vdec_sample**获取样例，查看该样例下的README。

13.20 媒体数据处理 V2（VENC 视频编码）

13.20.1 样例介绍

获取样例

单击**venc_sample**获取样例

功能描述

该样例实现将YUV420SP、YVU420SP格式的视频编码成H264、H265格式的码流。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。
数据传输	调用aclrtMemcpy接口： • 将数据从Host传输到Device上，作为解码的输入数据。 • 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见VENC视频编码功能/JPEGE图片编码功能中的接口。

目录结构

样例代码结构如下所示。

```
├── libHisiSdkCodec             //示例代码文件所在的目录
│   ├── hmev_hisi_sdk_*.h       //声明编码函数的头文件
│   ├── hmev_hisi_sdk_*.cpp     //编码函数的实现文件

├── CMakeLists.txt              //编译脚本
```

13.20.2 编译及运行应用

单击**venc_sample**获取样例，查看该样例下的README。

13.21 媒体数据处理 V2（PNGD 图片解码）

13.21.1 样例介绍

获取样例

单击**pngd_sample**获取样例

功能描述

该样例实现.png、.PNG图片的解码。

原理介绍

在该样例中，涉及的关键功能点，如下表所示。

初始化	• 调用aclInit接口初始化AscendCL配置。 • 调用aclFinalize接口实现AscendCL去初始化。
Device管理	• 调用aclrtSetDevice接口指定用于运算的Device。 • 调用aclrtGetRunMode接口获取软件栈的运行模式，根据运行模式的不同，内部处理流程不同。 • 调用aclrtResetDevice接口复位当前运算的Device，回收Device上的资源。
Context管理	• 调用aclrtCreateContext接口创建Context。 • 调用aclrtDestroyContext接口销毁Context。
内存管理	• 调用aclrtMallocHost接口/aclrtFreeHost接口申请/释放Host上内存。 • 执行媒体数据处理时，若需要申请Device上的内存存放输入或输出数据，需调用hi_mpi_dvpp_malloc接口/hi_mpi_dvpp_free接口申请/释放内存。

数据传输	调用aclrtMemcpy接口： ● 将数据从Host传输到Device上，作为解码的输入数据。 ● 模型推理结束后，将推理结果从Device传输到Host。 如果在板端环境上运行应用，则无需进行数据传输。
媒体数据处理V2	参见PNGD图片解码功能中的接口。

目录结构

样例代码结构如下所示。

```
├── src                          //示例代码文件所在的目录
│   ├── sample_comm.h            //声明解码函数的头文件
│   ├── sample_comm_pngd.cpp     //解码函数的实现文件
│   ├── sample_pngd.cpp          //解码函数的实现文件

├── CMakeLists.txt               //编译脚本
```

13.21.2 编译及运行应用

单击**pngd_sample**获取样例，查看该样例下的README。

14 FAQ

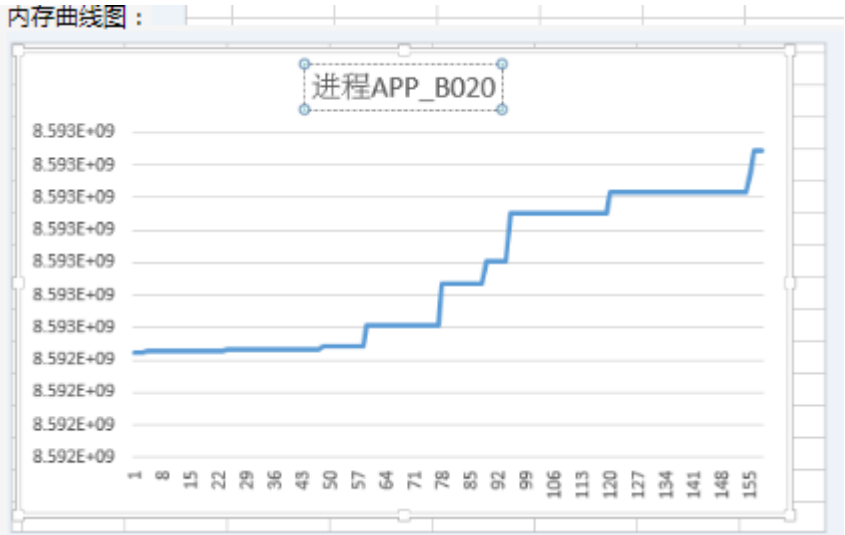
14.1 资源异常问题

14.1.1 内存未释放导致内存泄露

现象描述

测试用例长稳运行时，出现内存泄漏的现象，内存占用持续上升。如图14-1所示。

图 14-1 内存占用持续上升



可能原因

分析上述日志信息，可能存在以下故障原因：

系统存在只申请内存不释放内存的问题，正常情况下，内存申请与释放必须成对出现。

处理步骤

针对分析的故障可能原因，可以参考下面步骤处理：

排查所有的内存申请和释放的地方，保证申请与释放一一对应。例如aclrtMalloc与aclrtFree，aclrtMallocHost与aclrtFreeHost、aclrtCreateStream与aclrtDestroyStream等。

14.1.2 Event 数量超过上限导致 aclrtRecordEvent 接口返回失败

现象描述

调用aclrtRecordEvent接口在Stream中记录一个Event时，日志中的报错如下，红框中是关键日志信息，提示Event ID申请失败：

```
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.381 18408 StreamSynchronize:stream Synchronize failed
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.404 18408 StreamSynchronize:stream synchronize failed, error = 14
[ERROR] ASCENDCL(18401,main):2020-05-29-17:53:11.569.427 18408 acldvppDestroyChannel:acl/single_op/dvpp/channel.cpp:193: "synchronize stream failed, result =
[INFO] RUNTIME(18401,main):2020-05-29-17:53:11.569.453 18407 ReceivingRun:report[0].task_id=32772
[INFO] RUNTIME(18401,main):2020-05-29-17:53:11.569.473 18407 TryDelRecordedTask:del public task from stream, stream_id=835, tailTaskId=32772, delTaskId=32772,
[INFO] RUNTIME(18401,main):2020-05-29-17:53:11.569.491 18407 TaskFinished:device_id=0, stream_id=835, sq_id=835, task_id=32772, task_type=7,task_finish_num=4
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.517 18408 StreamDestroy:stream is not in current ctx
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.536 18408 StreamDestroy:destroy stream failed, error = 6
[ERROR] ASCENDCL(18401,main):2020-05-29-17:53:11.569.553 18408 DestroyNotifyAndStream:acl/single_op/dvpp/channel.cpp:41: "fail to destroy stream, ret = 6"
[INFO] ASCENDCL(18401,main):2020-05-29-17:53:11.569.586 18408 acldvppDestroyChannelDesc:acl/types/dvpp.cpp:401: "destroy DvppChannelDesc info: channelIndex =
mdListLen = 2097152."
[ERROR] DRV(18401,main):2020-05-29-17:53:11.569.787 [devdrv] [drvEventIdAlloc 647] error.
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.804 18409 EventIdAlloc:drvEventIdAlloc:errorcode = 7
[INFO] RUNTIME(18401,main):2020-05-29-17:53:11.569.819 18409 EventIdAlloc:id = -1
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.834 18409 Record:event id alloc error, error = 14
[WARNING] RUNTIME(18401,main):2020-05-29-17:53:11.569.849 18409 Record:fail to init record task
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.865 18409 Synchronize:fail to record
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.880 18409 StreamSynchronize:stream Synchronize failed
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.895 18409 StreamSynchronize:stream synchronize failed, error = 14
[ERROR] ASCENDCL(18401,main):2020-05-29-17:53:11.569.911 18409 acldvppDestroyChannel:acl/single_op/dvpp/channel.cpp:193: "synchronize stream failed, result =
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.927 18409 StreamDestroy:stream is not in current ctx
[ERROR] RUNTIME(18401,main):2020-05-29-17:53:11.569.942 18409 StreamDestroy:destroy stream failed, error = 6
[ERROR] ASCENDCL(18401,main):2020-05-29-17:53:11.569.958 18409 DestroyNotifyAndStream:acl/single_op/dvpp/channel.cpp:41: "fail to destroy stream, ret = 6"
[INFO] ASCENDCL(18401,main):2020-05-29-17:53:11.569.982 18409 acldvppDestroyChannelDesc:acl/types/dvpp.cpp:401: "destroy DvppChannelDesc info: channelIndex =
mdListLen = 2097152."
```

可能原因

分析上述日志信息，可能存在以下故障原因：Event ID的数量超过上限。

处理步骤

多Stream之间同步等待的场景下，Event ID的资源时可以复用的，复用Event ID的流程是：在调用aclrtRecordEvent接口+aclrtStreamWaitEvent接口后，若指定的Event已完成，则需要及时调用aclrtResetEvent接口释放Event资源。

需要用户按照复用Event ID的流程优化代码逻辑。

14.1.3 进程异常退出后重新执行任务失败

现象描述

进程异常退出时，包括强行终止任务（如ctrl + c或者kill命令终止进程）的场景，然后重新启动任务失败。

可能原因

进程异常退出时，只能依赖系统检测到程序退出后才进行资源释放，释放资源最长需要一分钟的执行时间。如果在未执行完资源释放前执行新的任务，可能导致新执行的任务失败。

处理步骤

进程异常退出后需要等待一分钟，才能保证下一次重新执行任务成功。

14.1.4 进程异常，下一次执行任务报错“unbind model stream failed”

现象描述

用户捕获异常退出信号，并在信号处理函数中释放已申请资源，下一次执行时会报执行失败。此时查看日志，会发现unbind model stream failed报错。

可能原因

进程异常时，Host侧内核态驱动会自动检测并发起对应进程Device侧资源释放的流程，不需要用户捕获进程异常的信号并主动完成清理。若用户主动释放，会影响到系统的资源释放流程。

处理步骤

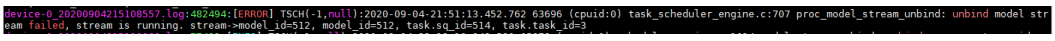
用户无需关注进程异常退出信号，不要对异常退出信号做处理。

14.1.5 进程异常时资源清理的处理建议

现象描述

用户捕获异常退出信号，并在信号处理函数中释放已申请资源，下一次执行时会报执行失败。此时查看日志，会发现如图14-2所示报错。

图 14-2 unbind model stream failed



可能原因

进程异常时，host侧内核态驱动会自动检测并发起对应进程device侧资源释放的流程，不需要用户捕获进程异常的信号并主动完成清理。若用户主动释放，会影响到系统的资源释放流程。

处理步骤

用户无需关注进程异常退出信号。

14.1.6 用户进程异常退出后重启进程失败

现象描述

用户进程卡住或者用户强制退出进程后，再次重启，重启后发现进程无法正常启动。类似的日志信息如下：

AscendCL日志信息：aclrtProcessReport failed

```
aclrtProcessReport failed, ret = 107012
aclrtProcessReport failed, ret = 107012
```

Runtime日志信息：halResourceIdAlloc xxx failed

```
[ERROR] DRV(2086,rtstest_host):2021-06-09-02:14:46.034.368 [ascend][curpid: 2086, 2086][drv][tsdrv]
[halResourceIdAlloc 477]id is exhausted, type(0 stream), range[0, 1024), dev_id(0), tsid(0).
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.380 [npu_driver.cc:285]2086 StreamIdAlloc:
[driver interface] halResourceIdAlloc streamid failed: device_id=0, tsId=0, drvRetCode=48!
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.401 [stream.cc:448]2086 Setup:Failed to
alloc stream id, retCode=0x702001a.
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.416 [context.cc:1251]2086
StreamCreate:Setup stream failed, retCode=0x702001a.
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.440 [logger.cc:211]2086
StreamCreate:Create stream failed, priority=7 ,flags=0.
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.458 [api_c.cc:461]2086
rtStreamCreateWithFlags:ErrCode=207008, desc=[driver error:no stream resource], InnerCode=0x702001a
[ERROR] RUNTIME(2086,rtstest_host):2021-06-09-02:14:46.034.469 [error_message_manage.cc:26]2086
ReportFuncErrorReason:rtStreamCreateWithFlags execute failed, reason=[driver error:no stream resource]
```

可能原因

通过日志分析无法正常重启的原因可能是public taskid、stream id、eventid等资源申请不到引起的：

- 资源已经被其他进程占用完。
- 上一个进程退出时还未完全释放完资源。

处理步骤

针对上述可能原因，可以按以下方式处理：

- 等待一分钟后再重新启动进程，保证上一个进程资源释放完成。
- 停止其他进程或者等其他进程执行完成后再启动进程。
- 如果通过上述方式处理后仍然申请失败，建议检查是否超过了可用的资源上限，如果未超上限，则需要重启环境强行释放资源、恢复环境。

14.1.7 AI 应用进程未退出，导致休眠唤醒失败

问题现象

休眠失败。

查看应用类日志，系统内部的任务分发模块hwts正处于busy状态，检查发现不满足休眠条件，日志片段示例如下：

```
[ERROR] TSCH(-1,null):2023-01-01-02:53:45.850.781 1 (dieid:0,cpuid:0) device_management_plat.c:563
suspend_ack: suspend pre check fail, hwts is busy
[EVENT] TSCH(-1,null):2023-01-01-02:53:45.850.803 2 (dieid:0,cpuid:0) device_management.c:411
process_low_power_cmd: ts suspend ack ret=1.
```


原因分析

根据休眠唤醒的流程，休眠前AI应用进程必须先退出，相关硬件资源处于idle态，才允许休眠。不满足休眠条件，会有相关报错，本案例中因为AI应用进程未退出，在休眠唤醒时检测到hwts处于busy状态，因此休眠失败。

解决办法

用户需要确保AI应用进程已经运行结束或者优雅退出，推荐使用**kill -2 *PID***退出相关进程，***PID***需按照替换为实际进程ID。

14.1.8 算子插件未注册报错

现象描述

查看日志， 存在报错某个算子类型不支持：

```
Check op[%s]'s type[%s] failed, it is not supported.
```

或者

进行模型转换的时候，某个算子类型转换不符合预期，被转换成了frameworkop类型。

可能原因

根据日志分析，可能存在以下可能原因：

- 算子插件so未加载成功。
- 算子未注册映射关系，或者未编译到算子的插件so中。

解决措施

针对分析可能的故障原因，可以参考下面步骤处理：

步骤1 确认算子插件so是否加载成功。

1. 算子插件so加载成功打印类似信息：

```
plugin load /usr/local/Ascend/opp/built-in/framework/onnx/libops_all_onnx_plugin.so success.
```

2. 加载失败的告警关键信息：

```
dlopen failed, plugin name:%s. Message(%s).
```

步骤2 如果算子插件so加载成功，则需要继续确认算子注册的映射关系是否编译进加载的插件so中了。

使用nm命令查看so符号表， 如果没有注册， 则需要注册该算子插件， 可以参考《TBE&AI CPU自定义算子开发指南》的“算子适配”章节内容实现。

说明

nm -D命令可查看so文件符号表。

步骤3 如果算子插件so未加载成功，参考失败告警中Message提示内容处理。

----结束

14.1.9 算子原型未注册报错

现象描述

查看日志， 存在报错某个算子没有原型定义：

```
op[%s] type[%s] have no ir factory.
```

或者

```
IR for op[%s] optype[%s] is not registered.
```

说明

op[%s] type[%s]中的%s分别表示具体的算子名称和算子类型。

可能原因

根据日志分析，可能存在以下可能原因：

- 算子原型so未加载成功。
- 算子未定义注册该类型算子， 并编译到算子的原型so中。

解决措施

针对分析可能的故障原因，可以参考下面步骤处理：

步骤1 确认算子原型so是否加载成功。

1. 算子原型so加载成功打印类似信息：

OpsProtoManager plugin load /usr/local/Ascend/opp/built-in/op_proto/libopsproto.so success.

2. 加载失败的告警关键信息：

OpsProtoManager dlopen failed, plugin name:%s. Message(%s).

步骤2 如果算子原型so加载成功， 需要确认算子原型定义是否编译进加载的so中了。

使用nm查看so符号表， 如果没有注册， 则需要注册该算子原型， 可以参考《 TBE&AI CPU自定义算子开发指南 》的“算子原型定义”章节内容实现。

说明

nm -D命令可查看so文件符号表。

步骤3 如果算子原型so未加载成功，参考失败告警中Message提示内容处理。

----结束

14.1.10 动态 shape 模型用户输入和模型推导结果不匹配

现象描述

场景1：模型执行报错，日志信息中包含以下关键信息：“[Check][Size] %s(%s) index[%d] mem size out of range! Expected size: %ld, but given input size: %ld.”


```
(INFO) GE(2284607, ir_build):2021-11-08-19:31:33.920.888 [task_context.cc:257]2284607 AllocateOutput: To allocate output for node: add2. index = 0, tensor desc = [TensorDescl DataType = 6 ， Format = 2 · Shape = [50, ]
[ERROR] GE(2284607, ir_build):2021-11-08-19:31:33.920.900 [task_context.cc:257]2284607 AllocateOutput: ErrorNo: 50331649() [LOAD][LOAD]【Check】【Size】 add2(Add) index[0] mem size out of range! Expected size: 128, but given input size: 2.
```

场景2：单算子场景执行报错，日志信息中包含以下关键信息：“[Check][Param:outputs]Output size mismatch. index = %zu, model expect %zu, but given %zu(after align)”

```
(INFO) GE(130191,python):2021-09-27-17:23:17.922.047 [single op.cc:161]131511 ValidateArgs:lnput [1], aligned_size:96, inputs.length:40, input_sizes_:96
(INFO) GE(130191,python):2021-09-27-17:23:17.922.054 [single op.cc:186]131511 ValidateArgs:Output [0], aligned_size:3145760, outputs.length:3145728, input_sizes_:18446744073709551615
[ERROR] GE(130191,python):2021-09-27-17:23:17,922.095 [single op.cc:161]131511 ValidateArgs: ErrorNo: 145000(Parameter invalid.) [FINALI][FlNALl]【CheCkl】【Param.Outputs】Output Size mismatch. index = 0, model expect 18446744073709551615, but given ....
[ERROR] ASCENDCL(130191,python):2021-09-27-17:23:17,922.213 [op_executor.cpp:68]131511 DoExecteAsync: [FINALI][FlNALl][Exec][Op]Execte op failed. ge result = 145000
```

可能原因

场景1：

此处为模型执行时，校验用户分配的output_buffer大小和模型经过一层层infershape后得到的模型输出的output_buffer大小不匹配。

场景2：

此处为单算子场景执行时，校验用户分配的output_buffer大小和单算子编译时infershape推出的output_size大小不匹配。

解决措施

针对分析的故障可能原因，可以参考下面步骤处理：

针对场景1：

若发生该错误，用户需要检查分配的output_buffer大小是否正确，应该与模型推导的大小保持一致。建议用户分配的output_buffer按照上报的ERROR中提示的大小进行分配。

针对场景2：

若发生该错误，用户需检查分配的output_buffer大小和算子infershape提示的size大小是否一致，建议按照报错提示的大小进行分配。

14.1.11 动态 shape 模型输入大小校验失败

现象描述

动态shape模型输入大小校验失败，日志信息中包含以下关键信息：tensor size mismatches. expected: ..., but given ...

```
plog-187626_20211011143937175.log:17480:[ERROR] GE(187626,python3):2021-10-11-14:39:37:270.942 [execution_engine.cc:450]191154 ValidInputTensor:ErrorNo: 1343225860(Internal errors) [EXEC][EXEC] [Check][Size] for [PartitionedCall_14(PartitionedCall)] Input[40]:tensor size mismatches.expected: 768032, but given 32.
```

可能原因

动态shape场景每个node都会在执行时对shape进行推导，该故障现象可能为算子的shape校验不合法。

解决措施

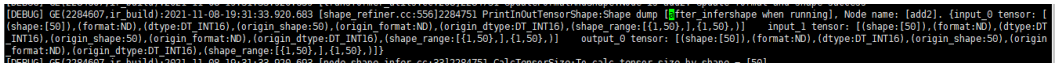
针对分析的故障可能原因，应根据图上的连接关系，对shape的来源进行排查，找出不合理的shape推导：

可搜索执行plog日志查看关键词：

"before_infershape when running"：显示算子shape推导前的输入、输出shape等信息。

“after_infershape when running ”：显示算子shape推导后的输入、输出shape等信息。

从报错节点的输入、输出shape开始进行排查， 检查当前节点shape推导结果是否正确（即判定根据输入shape推出的输出shape是否符合预期）， 如果是输入shape存在问题，则按照相同方法继续排查输入节点的shape推导。



14.1.12 AI Core 算子执行报错

现象描述

Runtime执行报错，在plog日志中Runtime打印了类似fault kernel_name和func_name的关键信息。

plog日志在{install_path}/ascend/log/debug/plog路径下，日志格式为plog-pid_yyymmddhhmmss.log。

```
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.403.262 [engine.cc:1103]4150867 ReportExceptProc:[EXEC][DEFAULT]Task exception! device_id=0, stream_id=20, task_id=1, type=13, retCode=0x91, [the model stream execute failed].
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.423 [device_error_proc.cc:495]4150867 PrintCoreErrorInfo:[EXEC][DEFAULT]The error from device(0), serial number is 193, there is an aicore error, core id is 8, error code = 0x800000, dump info: pc start: 0x800120080047000, current: 0x1200800471cc, vec error info: 0x7cafc4e, mte error info: 0x3000052, ifu error info: 0xc33f87bd7a80, ccu error info: 0xffd2bbd5005fe9d7, cube error info: 0x84, biu error info: 0, aic error mask: 0x65000200d000288, para base: 0x120080016300, errorStr: The DDR address of the MTE instruction is out of range.
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.443 [device_error_proc.cc:526]4150867 PrintCoreErrorInfo:[EXEC][DEFAULT]report error module_type=5, module_name=EZ9999
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.449 [device_error_proc.cc:526]4150867 PrintCoreErrorInfo:[EXEC][DEFAULT]The extend info from device(0), serial number is 193, there is aicore error, core id is 8, aicore int: 0x10, aicore error2: 0, axi clamp ctrl: 0, axi clamp state: 0x1717, biu status0: 0x101d14000000000, biu status1: 0x80000201020000, clk gate mask: 0, dbg address: 0, ecc en: 0, mte ccu ecc 1bit error: 0x2e80000000000000, vector cube ecc 1bit error: 0, run stall: 0x1, dbg data0: 0, dbg data1: 0, dbg data2: 0, dbg data3: 0, dfx data: 0x8b
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.607 [task.cc:1021]4150867 PrintErrorInfo:[EXEC][DEFAULT]Aicore kernel execute failed, device_id=0, stream_id=23, report_stream_id=20, task_id=24, flip_num=0, fault kernel_name=168057361183146196491_0_1_Add_35, func_name=te_add_729e2a87c649f49de98ac1a6fd491b3262ee7db9c1c2d6f4add7d7439aa3d22e_1__kernel0, program id=22, hash=3338199064661472585.
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.618 [task.cc:3275]4150867 ReportErrorInfo:[EXEC][DEFAULT]model execute error, retCode=0x91, [the model stream execute failed].
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.624 [task.cc:3247]4150867 PrintErrorInfo:[EXEC][DEFAULT]model execute task failed, device_id=0, model stream_id=20, model task_id=1, flip_num=0, model_id=3, first_task_id=65535
```



```
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.714 [stream.cc:929]4150867 GetError:[EXEC]
[DEFAULT]Stream Synchronize failed, stream_id=20, retCode=0x91, [the model stream execute failed].
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.742 [model.cc:581]4150867
SynchronizeExecute:[EXEC][DEFAULT]Fail to synchronize forbbiden stream_id=20, retCode=0x7150050!
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.748 [model.cc:605]4150867
GetStreamToSyncExecute:[EXEC][DEFAULT]report error module_type=0, module_name=EE9999
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.753 [model.cc:605]4150867
GetStreamToSyncExecute:[EXEC][DEFAULT]Model synchronize execute failed, model_id=3!
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.774 [logger.cc:856]4150867 ModelExecute:
[EXEC][DEFAULT]Execute model failed.
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.787 [api_c.cc:2063]4150867 rtModelExecute:
[EXEC][DEFAULT]ErrCode=507011, desc=[the model stream execute failed], InnerCode=0x7150050
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.793 [error_message_manage.cc:49]4150867
FuncErrorReason:[EXEC][DEFAULT]report error module_type=3, module_name=EE8888
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.801 [error_message_manage.cc:49]4150867
FuncErrorReason:[EXEC][DEFAULT]rtModelExecute execute failed, reason=[the model stream execute failed]
```

可能原因

从日志报错可知，AI Core 算子执行失败，可能算子本身代码问题：数据输入不匹配、访问越界、计算溢出等异常。

查阅plog日志，根据fault kernel_name和func_name可获取报错算子名称和报错函数名称。

plog日志在/root/ascend/log/debug/plog路径下，日志格式为plog-pid_yyymmddhhmmss.log。

```
[ERROR] RUNTIME(4150867,msame):2022-09-22-09:27:46.404.607 [task.cc:1021]4150867 PrintErrorInfo:
[EXEC][DEFAULT]Aicore kernel execute failed, device_id=0, stream_id=23, report_stream_id=20, task_id=24,
flip_num=0, fault kernel_name=16805736118314619649-1_0_1_Add_35,
func_name=te_add_729e2a87c649f49de98ac1a6fd491b3262ee7db9c1c2d6f4add7d7439aa3d22e_1__kernel0,
 program id=22, hash=3338199064661472585.
```

处理步骤

该类型错误，需要联系华为算子开发工程师定位排查。 您可以通过https://gitee.com/ascend网站提交issue获取帮助。

可能导致的故障

模型下沉场景下，该问题可能导致AscendCL报错Execute model failed，并打印在plog日志中。

```
[ERROR] ASCENDCL(4150867,msame):2022-09-22-09:27:46.404.834 [model.cpp:699]4150867
ModelExecute: [EXEC][DEFAULT][Exec][Model]Execute model failed, ge result[507011], modelId[1]
[ERROR] ASCENDCL(4150867,msame):2022-09-22-09:27:46.404.857 [model.cpp:1547]4150867
aclmdlExecute: [EXEC][DEFAULT][Exec][Model]modelId[1] execute failed, result[507011]
```

非模型下沉场景下，该问题可能导致算子执行失败AscendCL报错get op desc faild，Runtime报错Aicore kernel execute failed，并打印在plog日志中。

```
[ERROR] RUNTIME(2856615,xaclfk):2022-09-15-11:36:47.817.465 [task.cc:1058]2856939 PreCheckTaskErr:
[EXEC][DEFAULT]Kernel task happen error, retCode=0x26, [aicore exception].
[ERROR] RUNTIME(2856615,xaclfk):2022-09-15-11:36:47.817.538 [task.cc:1029]2856939 PrintErrorInfo:
[EXEC][DEFAULT]Aicore kernel execute failed, device_id=0, stream_id=0, report_stream_id=0, task_id=615,
flip_num=0, fault kernel_name=12646006_1663210912148832_-1_0_while/transformer_0/decoder/
layer_0/rnn/rnn/while/Select,
func_name=te_select_7b314df6791292127cb82df985d04ddaf6d069cb31aaccec00e0b8ee2e997f20_1__kernel
0, program id=131, hash=14736095126365135477.
[ERROR] GE(2856615,xaclfk):2022-09-15-11:36:47.818.283 [graph_execute.cc:557]2856939 GetOpDescInfo:
ErrorNo: 4294967295(failed) [EXEC][DEFAULT][Get][OpDescInfo] failed, device_id:0, stream_id:0,
task_id:615.
[ERROR] GE(2856615,xaclfk):2022-09-15-11:36:47.818.308 [ge_executor.cc:1332]2856939 GetOpDescInfo:
```



```
ErrorNo: 4294967295(failed) [EXEC][DEFAULT][Get][OpDescInfo] failed, device_id:0, stream_id:0,
task_id:615.
[ERROR] ASCENDCL(2856615,xaclfk):2022-09-15-11:36:47.818.315 [model.cpp:2216]2856939
aclmdlCreateAndGetOpDesc: [EXEC][DEFAULT][Get][OpDescInfo]get op desc faild, ge result[-1],
deviceId[0], streamId[0], taskId[615]
```

14.1.13 AI CPU 算子执行报错

现象描述

Runtime执行报错，在plog日志中Runtime打印了PrintAicpuErrorInfo的错误信息。

plog日志在{install_path}/ascend/log/debug/plog路径下，日志格式为plog-pid_yyymmddhhmmss.log。

```
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.791.865 [engine.cc:1103]16282
ReportExceptProc:Task exception! device_id=0, stream_id=7, task_id=2, type=1, retCode=0x2a, [aicpu
exception].
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.489 [device_error_proc.cc:669]16282
ProcessAicpuErrorInfo:report error module_type=0, module_name=E39999
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.498 [device_error_proc.cc:669]16282
ProcessAicpuErrorInfo:An exception occurred during AICPU execution, stream_id:7, task_id:2, errcode:5,
msg:aicpu execute failed.
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.932 [task.cc:1050]16282
PreCheckTaskErr:report error module_type=5, module_name=EZ9999
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.941 [task.cc:1050]16282
PreCheckTaskErr:Kernel task happen error, retCode=0x2a, [aicpu exception].
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.981 [task.cc:759]16282
PrintAicpuErrorInfo:report error module_type=0, module_name=E39999
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.793.990 [task.cc:759]16282
PrintAicpuErrorInfo:Aicpu kernel execute failed, device_id=0, stream_id=7, task_id=2.
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.116 [task.cc:777]16282
PrintAicpuErrorInfo:Aicpu kernel execute failed, device_id=0, stream_id=7, task_id=2, flip_num=0, fault
so_name=, fault kernel_name=, fault op_name=Unique, extend_info=(info_type:4, info_len:6,
msg_info:Unique).
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.384 [stream.cc:929]16243 GetError:[EXEC]
[DEFAULT]Stream Synchronize failed, stream_id=7, retCode=0x2a, [aicpu exception].
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.407 [stream.cc:932]16243 GetError:[EXEC]
[DEFAULT]report error module_type=0, module_name=E39999
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.419 [stream.cc:932]16243 GetError:[EXEC]
[DEFAULT]Aicpu kernel execute failed, device_id=0, stream_id=7, task_id=2, flip_num=0, fault so_name=,
fault kernel_name=, fault op_name=Unique, extend_info=(info_type:4, info_len:6, msg_info:Unique)
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.482 [logger.cc:305]16243 StreamSynchronize:
[EXEC][DEFAULT]Stream synchronize failed, stream = 0x5643fe3e28d0
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.510 [api_c.cc:661]16243 rtStreamSynchronize:
[EXEC][DEFAULT]ErrCode=507018, desc=[aicpu exception], InnerCode=0x715002a
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.519 [error_message_manage.cc:49]16243
FuncErrorReason:[EXEC][DEFAULT]report error module_type=3, module_name=EE8888
[ERROR] RUNTIME(16243,msame):2022-09-22-11:27:01.794.532 [error_message_manage.cc:49]16243
FuncErrorReason:[EXEC][DEFAULT]rtStreamSynchronize execute failed, reason=[aicpu exception]
```

可能原因

从日志报错可知，AI CPU算子执行失败，可能算子本身代码问题：数据输入不匹配、访问越界、AI CPU线程挂死等问题。

比如通过查阅AI CPU的device日志，是数据输入不匹配导致。

device日志在/root/ascend/log/debug/device-x/路径下，日志格式为device-pid_yyymmddhhmmss.log。

```
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.218 [aicpu_tf_kernel.cc:348][tid:2317]
[TFAdapter] AICPUKernelAndDevice::Run failure, kernel_id=0, op_name=Unique, op_type=UniqueExt,
error=Invalid argument: unique expects a 1D vector.
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.242 [tf_adpt_session_mgr.cc:74][tid:2317]
```

```
[TFAdapter] [sessionID:0] Failed to Run kernel, kernel_id=0.
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.261 [tf_adpt_session_mgr.cc:434]
[tid:2317][TFAdapter] [sessionID:0] Run kernel on session failed.
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.277 [tf_adpt_api.cc:85][tid:2317]
[TFAdapter] [sessionID:0] Invoke TFOperateAPI failed.
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.296 [ae_kernel_lib_fwk.cc:229]
[TransformKernelErrorCode][tid:2317][AICPU_PROCESSER] Call tf api return failed:5, input param to tf
api:0x124040017004
[ERROR] CCECPU(2309,aicpu_scheduler):2022-09-22-11:27:00.733.366 [aicpusd_event_process.cpp:1325]
[ExecuteTsKernelTask][tid:2317] Aicpu engine process failed, result[5].
```

处理步骤

该类型错误，需要联系华为算子开发工程师定位排查。 您可以通过https://gitee.com/ascend网站提交issue获取帮助。

14.1.14 调用 Device 失败

现象描述

调用Device失败（容器内），日志中报以下类似信息：

日志信息（1）：日志中有打印ERROR信息“drvDeviceOpen”失败字样，如下所示。

```
[ascend][ERROR][2021-06-26-17:53:15:689064][curpid: 32707, 32707][devdrv][drvDeviceOpen 79]get phys
failed, devId(0), phy_devId(0)
```

日志信息（2）：查看内核日志(dmesg)中有打印“device is in used”字样。如下所示。

```
[52783.010878] [ascend] [ERROR] [devdrv] [devdrv_manager_container_table_devlist_add_ns 832]
<drv_hlt_dsmi_te:5615> device is in used
[52783.010881] [ascend] [ERROR] [tsdrv] [devdrv_open 194] <drv_hlt_dsmi_te:5615> add to list failed.
dev_id(0)
[52783.013546] [ascend] [devdrv] [devdrv_manager_container_get_devnum 1306] <drv_hlt_dsmi_te:5615>
weird device number, dev_num = 0
[52783.013547] [ascend] [ERROR] [devdrv] [devdrv_manager_container_get_devlist_ns 1210]
<drv_hlt_dsmi_te:5615> some devices are used by other docker, mnt_ns = 0xffff8d33a5963720
[52783.013548] [ascend] [ERROR] [devdrv] [devdrv_manager_container_get_davinci_devlist 1248]
<drv_hlt_dsmi_te:5615> get davinci devlist failed, ret(-22).
```

可能原因

针对调用Device失败，可能原因如下：

- 日志信息（1）：Docker容器启动时未映射设备。
- 日志信息（2）：Device被其他容器占用。

处理步骤

- 针对日志信息（1）可能原因，可以参考以下方法处理：

  在容器内使用 **ls /dev** 指令查看是否映射Device。

  ```
  [root@Euler /]# ls /dev
  console  core  fd  full  mqueue  null  ptmx  pts  random  shm  stderr  stdin  stdout  tty  urandom  zero
  [root@Euler /]#
  ```

  如果查看到Docker容器内未映射设备，请参考《CANN软件安装指南》资料启动容器，起Docker的命令中必须加挂载设备的参数，例如：

  --device=/dev/davinciX --device=/dev/davinci_manager --device=/dev/devmm_svm --device=/dev/hisi_hdc

- 针对日志信息（2）可能原因，需要等使用该Device的容器被释放后再使用。
  可以在容器内执行**ls /dev**命令查看当前容器内分配到的DavinciX，使用同样的方法查看其他容器分配到的DavinciX，然后可以通过手工停止Docker容器（例如：docker stop 容器id）的方式提前释放Device。

14.1.15 内存申请失败，出现 OOM 情况

现象描述

内存申请失败，Host侧日志提示EL9999返回码，有如下打印信息：

```
[ERROR] DRV(2936187,python3):2022-04-21-14:19:39.429.481 [ascend][curpid: 2936187, 2969960][drv][devmm][devmm_alloc_managed 182]<errno:12, 6> Heap_alloc_managed out of memory. (temp_ptr=0x1; bytesize=8592031776)
[ERROR] RUNTIME(2936187,python3):2022-04-21-14:19:39.429.491 [npu_driver.cc:780]2969960 DevMemAllocHugePageManaged:report error module_type=1, module_name=EL9999
[ERROR] RUNTIME(2936187,python3):2022-04-21-14:19:39.429.495 [npu_driver.cc:780]2969960 DevMemAllocHugePageManaged:[driver interface] halMemAlloc failed: device_id=1, size=8592031776, type=0, env_type=3, drvRetCode=6!
```

可能原因

根据日志信息分析，判断为内存申请失败。可能原因：

1. 网络并行运行，导致内存不足。
2. 网络运行需要内存过大，导致内存申请失败。

处理步骤

步骤1 查看运行网络时是否存在并行情况。

步骤2 查询网络运行需要内存大小或者减少batchsize，查看网络是否可以正常运行。

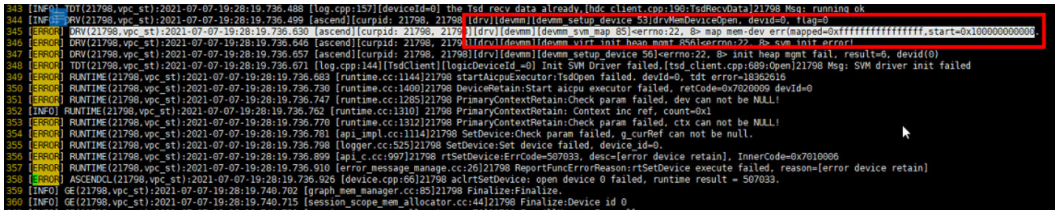
----结束

14.1.16 虚拟地址抢占导致 mmap 失败

现象描述

mmap申请失败，Host侧日志显示drvMemDeviceOpen失败，出现如图14-3日志信息。

图 14-3 drvMemDeviceOpen 失败



可能原因

根据日志信息，判断为mmap地址失败。可能原因：

（1）用户程序编译选项中启动了地址消毒（-lasan）：asan地址消毒会预留所有的虚拟地址空间，而svm在mmap时也需要预留8T的地址空间，二者冲突，所以mmap时会失败。

（2）用户预留地址与SVM模块预留地址相同。

处理步骤

步骤1 关闭编译选项即可。

步骤2 查看用户预留地址是否与SVM模块预留地址重叠，需要修改用户预留地址空间。

----结束

14.2 推理问题

14.2.1 使用 dump 功能未获取 dump 结果

现象描述

日志显示正确执行了Dump功能，但在Dump结果路径下没有Dump的结果。日志信息包含了以下关键字：

```
[INFO] ASCENDCL ****** "HandleDumpConfig end in HandleDumpConfig."
[INFO] ASCENDCL ****** "set HandleDumpConfig success in aclInit"
```

可能原因

分析上述日志信息，可能存在以下故障原因：Dump配置的模型名与实际的模型名不匹配。

处理步骤

针对分析的故障可能原因，可以参考下面步骤处理：

检查Dump配置文件acl.json，确保Dump配置文件合法，例如model_name是否配置正确。示例如下：

```
{
"dump":{
"dump_list":[
{
"model_name":"ResNet-50",
"layer":[
"convlconvl_relu"
]
},
{
```


"model_name":"mxnet-model"

}

],

"dump_mode":"output",

"dump_path":"/home/test/output/dump"

}

}

通过ATC命令生成模型的json文件，在json文件中查找“"name"”字段对应值，查找模型名称和算子名称，模型名称在"graph"字段外、算子名称在"graph"字段内。

14.2.2 动态 shape 推理申请内存失败

现象描述

模型推理过程中，申请了大小为0的内存，日志报错信息中包含以下关键信息：

```
[INFO] ASCENDCL ****** start to execute aclrtMalloc, size = 0
[ERROR] ASCENDCL ****** malloc size must be greater than zero
```

可能原因

模型为动态shape模型，模型的输出shape中含有-1，所以直接调用aclmdlGetOutputSizeByIndex接口取到的size为0。

然后申请了大小为0的内存，导致失败。

处理步骤

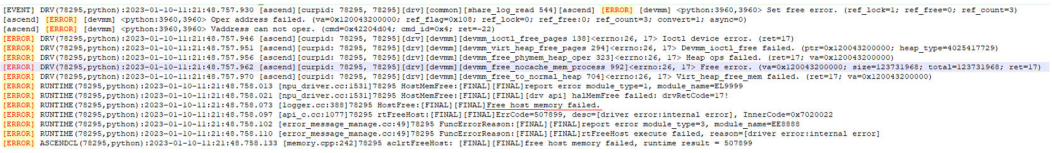
请参见**8.9 模型动态Shape输入推理**章节内容处理。

在aclmdlGetOutputSizeByIndex取到size为0时，用户需要预估一块较大的内存。

14.2.3 异步拷贝调用查询接口报错

现象描述

通过event实现H2D或D2H异步拷贝任务的同步等待时，在调用aclrtQueryEventStatus确认任务完成后，先调用aclrtFreeHost释放Host内存再调用aclrtDestroyEvent接口，可能会有如下报错信息打印：



可能原因

报错是因为使用了异步拷贝任务之后下发了一个event record任务，期望使用aclrtQueryEventStatus查询到event record任务是否完成，从而判断异步拷贝任务是否完成，而后释放内存调用aclrtFreeHost。

实际上aclrtQueryEventStatus查询到的是Device执行完任务，并未透传到Host侧，所以此时释放内存，未先销毁Event会有时序问题导致报错。

处理步骤

处理该问题可以参考以下方案：

方案一：使用aclrtSynchronizeStream接口判断任务是否执行完成。

方案二：使用aclrtQueryEventStatus接口时，先调用aclrtDestroyEvent接口，再调用aclrtFreeHost接口，保证无时序问题。

14.2.4 注册算子数超过最大规格

现象描述

推理过程中，用户load model出现报错。在plog日志中Runtime打印了类似ProgramRegister:Program register failed, program out of xxx和Register binary failed的关键信息。

plog日志在{install_path}/ascend/log/debug/plog路径下，日志格式为plog-pid_yyymmddhhmmss.log。

```
[ERROR] RUNTIME(3093,rtstest_host):2021-06-09-02:30:34.400.124 [runtime.cc:967]3093
ProgramRegister:Program register failed, program out of 40000000
[ERROR] RUNTIME(3093,rtstest_host):2021-06-09-02:30:34.400.155 [logger.cc:23]3093
DevBinaryRegister:Register binary failed.
[ERROR] RUNTIME(3093,rtstest_host):2021-06-09-02:30:34.400.182 [api_c.cc:127]3093
rtDevBinaryRegister:ErrCode=507032, desc=[program register num out of use], InnerCode=0x7090007
[ERROR] RUNTIME(3093,rtstest_host):2021-06-09-02:30:34.400.185 [error_message_manage.cc:26]3093
ReportFuncErrorReason:rtDevBinaryRegister execute failed, reason=[program register num out of use]
```

可能原因

通过日志分析报错的原因可能是一个进程内算子等资源注册超过最大规格：

- 模型太大，一个进程内的注册总算子数超过最大规格：Online模式规格为4000万，Offline模式规格为200万。

处理步骤

针对上述可能原因，可以按以下方式处理：

- 分析model，简化模型或者降低动态batch档次。
- 算子数是进程资源，model太大的情况下建议一个进程open一个device。
- 避免同一算子在不同模型中反复注册。
- 注册算子数不超过最大规格。

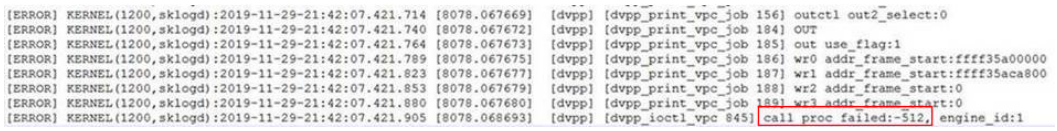
14.3 DVPP 处理数据问题

14.3.1 DVPP 驱动引擎异常返回码

现象描述

视频解码失败，Device侧日志提示-512返回码：[dvpp_ioct1_vpc 845] call proc failed:-512, engine_id:1，如图14-4所示。

图 14-4 DVPP 引擎返回-512 异常码



可能原因

根据日志信息，判断为VPC引擎返回的异常码，再根据-512返回码可以判断为等待视频解码异常中断。可能原因：用户态进程退出，导致等待解码时被异常中断。

解决方法

查看Host侧的日志文件，检查用户态进程异常退出原因。

其他异常码参考

除样例VPC引擎返回的-512异常码外，DVPP驱动引擎还有其他引擎及各自的返回码，如表14-1所示。

您可以参考上述样例的方法，根据自身业务类型及Device侧日志文件中记录的异常返回码信息，先判断返回异常码的引擎类型，再根据返回码数值获取对应的返回码说明，初步判断故障原因。

表 14-1 DVPP 驱动引擎异常返回码

引擎类型	驱动IOCTL返回码	返回码说明	常见错误日志参考
JPEGE	-22	参数错误或者不支持	常见错误打印： 1.返回值-35：日志“JPEGE time wait, engine_id:xx”编码超时(执行器输入编码参数不对等导致) 2.返回值-512：日志“Signal interrupted, engine_id:xx”编码错误(编码被异常中断)
	-11	引擎处理异常，需要重试	
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-35	编码超时	
	-5	编码失败且复位成功	

引擎类型	驱动IOCTL返回码	返回码说明	常见错误日志参考
	5	编码失败且复位失败	
	-512	等待编码时被异常中断	
PNGD	-22	参数错误或者不支持	常见错误打印： 1.返回值5/-5：日志1“Time out: wait failed xx”解码超时(图片格式不正确等码流异常等导致) 1.返回值5/-5：日志2“Signal interrupted:xx, engine_id:xx”解码时被异常中断
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-5	解码失败且复位成功	
	5	解码失败且复位失败	
JPEGD	-22	参数错误或者不支持或者解码成功但是重置寄存器失败	常见错误打印： 1.返回值-1/-23：日志“Decode abnormal, abnormal status:xx decode error! engine id:xx”硬件上报异常中断(图片格式不正确等码流异常导致) 2.返回值-2/-24：日志“Decode abnormal, abnormal status:xx decode over time!, engine id:xx”解码超时(总线繁忙，系统缺页等导致)
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-1	解码失败且复位成功	
	-2/-3	解码超时且复位成功	
	-23	解码失败并且复位失败	
	-24/-25	解码超时并且复位失败	

引擎类型	驱动IOCTL返回码	返回码说明	常见错误日志参考
VPC	-22	参数错误或者不支持	常见错误打印： 1. 返回值-35：日志“dvpp_vpc_engine_proc pipe wait time out: engine id:xx”解码超时(总线繁忙、异常缺页等导致)。 2. 返回值-32：日志“Hardware process failed, engine id:xx”解码错误(码流异常等导致)。
	-11	引擎处理异常，需要重试	
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-35	解码超时	
	-512	等待解码时被异常中断	
	-32	硬件解码失败驱动捕捉到异常	
VPC_cmdlist	-22	参数错误或者不支持	常见错误打印： 1. 返回值-35：日志“dvpp_vpc_engine_proc pipe wait time out: engine id:xx”解码超时(总线繁忙、异常缺页等导致)。 2. 返回值-32：日志“Hardware process failed, engine id:xx”解码错误(码流异常等导致)。
	-11	引擎处理异常，需要重试	
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-35	解码超时	
	-512	等待解码时被异常中断	
	-32	硬件解码失败	
VENC	-22	参数错误或者不支持	常见错误打印： 1. 返回值-35：日志“wait timeout, Ret value is xx”未等到完成中断，编码失败(执行器配置参数错误或者总线繁忙等导致)。 2. 返回值-61：日志“venc err interrupt state = xx, wait_time_remain = xx”编码失败(执行器配置参数错误)。
	-11	引擎处理异常，需要重试	
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-35	未等到完成中断，编码失败	

引擎类型	驱动IOCTL返回码	返回码说明	常见错误日志参考
	-61	上报编码超时中断	
VDEC	-22	参数错误或者不支持	常见错误打印： 1. 返回值-35：日志“VDM wait time out,wait_time_remain = xx ,engine_id:xx”解码超时(码流异常、总线繁忙等导致)。 2. 返回值=-61：日志“wait_time_remain = xx,have vdm_error,p_vdm_backup_state->VdmState=xx ,engine_id:xx”上报解码错误中断(码流异常等导致)。
	-11	引擎处理异常，需要重试	
	-14	拷贝用户态数据时地址异常或者释放引擎失败	
	-16	申请引擎失败	
	-35	解码超时	
	-61	上报解码错误中断	
	-4	等待解码被异常中断	

14.3.2 两个版本的 DVPP 接口混用导致应用程序报错退出

问题现象

使用媒体数据处理V1接口（接口名以acldvpp开头）创建了通道，同时又使用媒体数据处理V2接口（接口名以hi_mpi开头）接口创建通道，后者创建通道时失败。

日志示例如下：

```
acl and himpi mode is incompatible, please check! used channel-num: 10
```

原因分析

媒体数据处理V1接口（接口名以acldvpp开头）与媒体数据处理V2接口（接口名以hi_mpi开头）不兼容，多进程或者多线程使用的时候，只能选择其中一个，否则DVPP会进行拦截、报错退出应用程序。

解决方法

排查业务代码流程，检查V1、V2两套接口混用的情形，目前无论是多线程还是多进程场景，均不支持两套接口混用。

14.3.3 VPC 图片处理

14.3.3.1 调用错误的内存申请接口，导致内存地址校验出错

现象描述

调用VPC接口，返回HI_ERR_VPC_BADADDR （ 0xA0078011 ）错误码，同时日志中有错误提示，不同版本的报错日志可能存在差别：

- 日志示例1
  device 0, vpc address is illegal, please make sure it has been allocated with hi_mpi_dvpp_malloc or acldvppMalloc.
- 日志示例2
  dvpp_check_mem_usable [Line]:85 mem:0x****f000****4020 is not usable, please check:1. mem not alloced or has been freed;2. make sure mem actual size should be:8017920

可能原因

根据日志提示，是由于没有使用指定的接口申请内存。

解决方法

检查代码，是否使用媒体数据处理V1版本中的acldvppMalloc接口/媒体数据处理V2版本中的hi_mpi_dvpp_malloc接口申请存放VPC输入或输出数据的内存。

14.3.3.2 使用正确的内存申请接口，但内存大小传值错误

现象描述

不同版本的报错日志可能存在差别：

- 日志示例1
  buffer size(3110400) is smaller than need buffer size(4147200) when format is 3.
- 日志示例2
  device 0, vpc end address is illegal, check allocated buffer size: configured buffer size: 3110400, current pic: format 3 width_stride 1920 height_stride 1080.
- 日志示例3
  dvpp_check_mem_usable [Line]:85 mem:0x****f000****4020 is not usable, please check:1. mem not alloced or has been freed;2. make sure mem actual size should be:8017920

可能原因

1. 代码中申请的内存大小小于该格式所需的输入或输出内存大小;

2. VPC任务接口传入的buffer size正常，与输入格式匹配，但是超出了实际申请的内存长度，所以校验出来结束地址非法。

解决方法

1. 检查代码，根据图片格式、宽高对齐、内存约束中的说明，检查对应格式的内存大小要求;

2. 在代码中增加打印内存长度的日志，检查VPC任务接口传入的buffer size是否与实际申请的内存长度一致。

14.3.3.3 读/写内存地址无效，导致异常中断

现象描述

Device侧内核态日志报错VPC异常（不同版本的报错日志可能存在差别）：

- 日志1：
  ```
  vpc get err int: vpc_cvdr_axi_rd_resp_err
  ```
- 日志2：
  ```
  vpc get err int: vpc_cvdr_axi_wr_resp_err
  ```

可能原因

- **cvdr_axi_rd_resp_err**表示读地址越界，可能申请的输入内存太小或内存地址无效，昇腾AI处理器执行读操作时访问到了无效地址。
- **cvdr_axi_wr_resp_err**表示写地址越界，可能申请的输出内存太小或内存地址无效，昇腾AI处理器执行写操作时访问到了无效地址。

解决方法

1. 在申请DVPP内存的接口处、以及在VPC异常任务接口处增加日志打印，检查申请的输入\输出内存大小与实际使用的输入\输出内存大小是否一致；

2. 在释放DVPP内存的接口处增加打印日志，检查VPC任务完成之前是否存在内存被提前释放的情况。

14.3.3.4 VPC 调用失败

现象描述

VPC模块调用失败（不同版本的报错日志可能存在差别），查看日志有类似如下报错信息：

日志信息（1）：

```
RoiNum(0), inputArea rightOffset is 1918, it should be odd!
```

日志信息（2）：

```
Output bufferSize(65536) should not be smaller than widthStride(256) * heightStride(256) * 3 / 2 = 98304
```

日志信息（3）：

```
Input widthStride(300) is not right, it should be 16 algined!
Input widthStride(16) is not right, it should not be smaller than 32!
```

日志信息（4）：

```
bareDataAddr(0xaaaadeccdcd0), bareDataBufferSize(3133440) should be allocated by acldvppMalloc!
```

日志信息（5）：

```
Both RoiNum(1) outputAddr(0xaaaadeccdcd0) and first roi outputAddr(0xffff00002000) should be allocated by acldvppMalloc!
```

日志信息（6）：

```
RoiNum(0): inputConfigure cropArea, leftOffset(26) should be smaller than rightOffset(25)!
RoiNum(0): inputConfigure cropArea, upOffset(80) should be smaller than downOffset(79)!
RoiNum(0): inputConfigure cropArea, cropWidth(1931) should not be bigger than width(1920)!
```



```
RoiNum(0): inputConfigure cropArea, cropHeight(1270) should not be bigger than height(1088)!
RoiNum(0): inputConfigure cropArea, cropWidth(9) should be between [10, 8192]!
RoiNum(0): inputConfigure cropArea, cropHeight(4) should be between [6, 8192]!
RoiNum(0): inputConfigure cropArea, rightOffset(1921) should be smaller than width(1920)!
RoiNum(0): inputConfigure cropArea, downOffset(1089) should be smaller than height(1088)!
```

日志信息（7）：

```
RoiNum(0): scale must be in [1/32, 16], cropWidth(1920), pasteWidth(10)!
RoiNum(0): scale must be in [1/32, 16], cropHeight(6), pasteHeight(100)!
```

可能原因

针对上面日志信息分析，可能存在以下对应原因：

- 日志信息（1）：VPC抠图区域右偏移坐标需是奇数，日志信息里1918是偶数，不符合要求。
- 日志信息（2）：当输出内存大小应该大于等于宽stride*高stride*3/2，日志信息显示不满足这个条件。
- 日志信息（3）：输入图片的宽stride（即每行图像占用的内存大小）必须是16倍数、且最小值32。日志信息里宽stride是300，不满足16倍数的要求，需要将图像做对齐后，并将宽stride设置成对齐后的值。
- 日志信息（4）：VPC的输入内存需要使用acldvppMalloc接口申请。
- 日志信息（5）：VPC的输出内存需要使用acldvppMalloc接口申请。
- 日志信息（6）：VPC的抠图区域不符合约束要求，因此报错。
- 日志信息（7）：VPC的缩放范围为[1/32, 16]。日志信息提示了缩放范围，并且显示了抠图宽为1920，输出区域的宽为10，通过计算可以得到：10/1920 < 1/32，因此报错。

定位思路

1. 根据日志描述的错误信息，找到VPC对应的配置参数，根据提示进行修改。
2. 根据日志描述的错误信息，参考**9 媒体数据处理（含图像/视频等）**中VPC功能参数的约束修改。

解决方法

根据提示的错误信息进行修改：

步骤1 如果为日志信息（1），说明输入图片抠图区域的右偏移错误，应该设置为奇数。

步骤2 如果为日志信息（2），说明输入内存的大小不正确，应该检查申请输入内存的代码，申请内存大小应该为1920*1088*3/2，并且bareDataBufferSize这个值也要填写为1920*1088*3/2。

步骤3 如果为日志信息（3），说明输入图片的stride值不符合要求，需设置为16的倍数。

步骤4 如果为日志信息（4）和日志信息（5），代码中申请内存时，需要使用acldvppMalloc接口申请。

步骤5 如果为日志信息（6），需要修改抠图宽度。

步骤6 如果为日志信息（7），需要修改缩放范围。

----结束

14.3.3.5 VPC 参数校验失败

现象描述

调用VPC功能接口返回0xA0078003，即HI_ERR_VPC_ILLEGAL_PARAM，参数超出合法范围。查看日志有类似如下报错信息，不同版本的报错日志可能存在差别：

- VPC缩放宽大于输出宽日志信息（1）：
  ```
  resize width (224) is greater than output width (120)
  ```
  或
  ```
  width should be in [10, 32768], width is 654321
  ```
- VPC抠图宽大于输入宽日志信息（2）：
  ```
  crop width 120 cannot be greater than input width 100
  ```
- 输入的缩放算法不正确日志信息（3）：
  ```
  resize interpolation [8] is not supported
  ```
  或
  ```
  flip mode[5] not supported, support mode [0, 1, 2]
  ```
- 图片内存大小不够日志信息（4）：
  ```
  buffer size(50176) is smaller than need buffer size(95264) when format is 1
  ```
  或
  ```
  buffer size(50176) is smaller than need buffer size(95264) when format is 1
  ```
- 图片地址校验失败日志信息（5）：
  ```
  on device 0, num 0 input addr (start 0x100020003000 end 0x100020004000) is illegal
  ```
  或
  ```
  buffer address is null
  ```
- VPC抠图的输出宽大于输入宽或者输出高大于输入高日志信息（6）：
  ```
  crop width[300] or height[300] is greater than input width[224] or height[224]
  ```

可能原因

针对上面日志信息分析，可能存在以下对应原因：

- 日志信息（1）：VPC缩放宽大于输出宽，
- 日志信息（2）：VPC抠图宽大于输入宽
- 日志信息（3）：输入的缩放算法不正确
- 日志信息（4）：图片内存大小不够
- 日志信息（5）：图片地址校验失败
- 日志信息（6）：VPC抠图的输出宽大于输入宽或者输出高大于输入高

定位思路

1. 根据日志描述的错误信息，找到VPC对应的配置参数，根据提示进行修改。
2. 根据日志描述的错误信息，参考**9 媒体数据处理（含图像/视频等）**中VPC功能参数的约束修改。

解决方法

根据提示的错误信息进行修改：

步骤1 如果为日志信息（1），修改resize宽，使其小于等于输出宽。

步骤2 如果为日志信息（2）和（6），修改抠图宽，使其小于输入宽。

步骤3 如果为日志信息（3），修改缩放算法为当前版本支持的类型。

步骤4 如果为日志信息（4），需要按照格式申请足够的内存，并正确配置buffer_size

步骤5 如果为日志信息（5），使用hi_mpi_dvpp_malloc或acldvppMalloc 申请图片地址

----结束

14.3.3.6 VPC 创建通道失败

现象描述

调用hi_mpi_vpc_create_chn创建通道返回错误码HI_ERR_VPC_EXIST，查看日志有如下类似错误信息，不同版本的报错日志可能存在差别：

```
device 0, chn 0 has already been created!
```

或

```
dev 0 chnl 0 is busy
```

可能原因

对于VPC模块，调用hi_mpi_vpc_create_chn指定通道号，创建通道，任何情况下通道号不能重复。原因是该通道已存在了。

定位思路

检查代码，查看通道号的使用。

解决方法

1、规划通道号，hi_mpi_vpc_create_chn创建时传入未被使用的通道号。

2、或者使用hi_mpi_vpc_sys_create_chn，调用完成后系统会分配一个未被使用过的通道号。

14.3.3.7 VPC 获取结果失败

现象描述

调用hi_mpi_vpc_get_process_result，返回HI_ERR_VPC_ILLEGAL_PARAM，查看日志信息，不同版本的报错日志可能存在差别：

```
this channel doesn't have taskID 8845!, Channel id 0
```

或

```
taskId:8845 does not exist
```

可能原因

调用hi_mpi_vpc_get_process_result传入一个非法的task id。

解决方法

调用hi_mpi_vpc_get_process_result传入的task id，必须是功能接口中传出的。

将功能接口中传出的task id传入hi_mpi_vpc_get_process_result。

14.3.3.8 VPC 输出图片存在花屏/绿边等

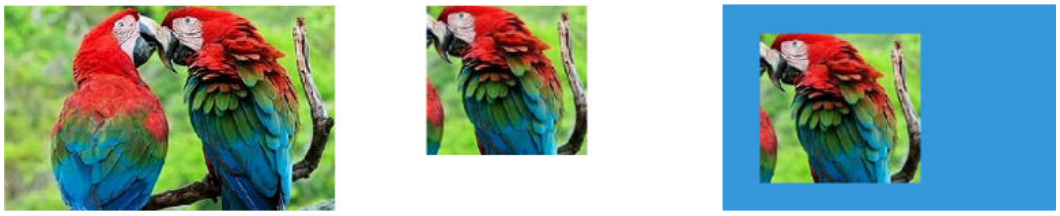
问题现象

输出图片存在花屏、绿边等。

示例图片如下所示，图片右侧存在绿边：

可能原因

1. 由于硬件约束，vpc对输出图片的宽、高有对齐要求，且不同版本的硬件其对齐要求不同，这个绿边可能是无效的对齐数据。例如，若VPC要求输出图片宽16对齐，但当前输出图片宽为100，不满足16对齐，因此用户需配置对齐后的宽，比如128，这时多出来的28就是无效的对齐数据，会导致输出图片存在绿边。
2. 如果使用抠图缩放贴图功能，当抠图缩放大小与输出图片不一致时，如下图所示，在贴图时最终蓝色的底图部分可能为输出内存里的随机值，这部分随机值会导致花屏。



解决方法

1. 检查代码中配置的输出图片宽、输出图片对齐后的宽分别是多少，如果绿边为对齐的数据，则需要用户按输出图片宽逐行写文件，剔除对齐数据。

   可参见**vpc_sample**中的样例代码。

2. 如果使用抠图缩放贴图功能，当抠图缩放大小与输出图片不一致时，需检查输出内存是否提前做过memset_s，通过初始化内存去除随机值的影响。

14.3.4 JPEGD 图片解码/VDEC 视频解码

14.3.4.1 调用错误的内存申请接口，导致内存地址校验出错

现象描述

日志报错如下：

```
address check failed ret 0x3, please check: 1. use hi_mpi_dvpp_malloc or aclDvppMalloc to alloc dvpp memory; 2. current buffer size 3110400 may be larger than actually allocated.
```

可能原因

根据日志提示，可能由于：

1. 没有使用指定的DVPP内存申请接口;

2. 接口传入的buffer size小于实际申请的内存大小。

处理步骤

检查代码：

1. 是否使用媒体数据处理V1版本中的acldvppMalloc接口/媒体数据处理V2版本中的hi_mpi_dvpp_malloc接口申请存放JPEGD图片解码/VDEC视频解码输入或输出数据的内存;

2. 对于DVPP内存申请接口，增加日志打印内存大小及地址，检查接口hi_mpi_vdec_get_frame/aclvdecSendFrame/acldvppJpegDecodeAsync送入的buffer size是否超出了实际申请的内存区域。

14.3.4.2 内存被提前释放，导致解码数据花屏

现象描述

原始H264/H265每一帧视频流正常，解码过程无异常（无异常日志），仅仅输出图片有异常。

可能原因

解码过程无异常，说明送入的码流非异常码流，仅仅输出被破坏，可能由于：

1. 输出内存被别人复用，被踩或者被提前释放;

2. 解码需要的输出内存比实际申请的内存大。

处理步骤

1. 对于DVPP内存申请接口，增加日志打印内存大小及地址，检查VDEC输出内存，检查申请的内存大小是否与实际使用的一致，比如典型的错误场景，VDEC解码输出格式预期是RGB，实际仍按照YUV420sp申请内存。
2. 在DVPP内存释放接口处、以及hi_mpi_vdec_get_frame/aclvdecCallback/acldvppJpegDecodeAsync接口处，增加内存大小及地址的打印日志，确认内存释放时序，是否存在内存地址解码完成前被提前释放的情况。

14.3.4.3 JPEGD 图像解码进程超时退出

现象描述

用户进程退出。

查看应用类日志，发现类似ERROR日志“task timeout, userData= ..., timeout=30, duration=…”和WARNING日志“frames statistic: ACL receive(*n*), send(*n*-1)”，*n*表示处理任务数量，需根据实际情况确定。

日志片段举例如下：

```
[ERROR] AICPU(pid,pName):DateTimeMS [dvpp_timeout_manager.cc:33][OnPulse][tid:2581]
[DVPP_KERNELS] WaitId[10] task timeout, userData=0xe7ffe0001280, timeout=30, duration=30.930062.
[INFO] AICPU(pid,pName):DateTimeMS [dvpp_kernel_base.cc:222][SendTaskCompleteToTs][tid:2581]
[DVPP_KERNELS] Send task complete to ts success, taskId=2, streamId=44, errorCode=1.
[WARNING] DVPP(pid,pName):DateTimeMS [JpegdAsyncManager.cpp:405][API] [PrintFrameCount:405]
[T208] DFX[JPEGD]: frames statistic: ACL receive(16), send(15)
```

可能原因

多路并发解码JPEG图片场景下，如果每一路解码的JPEG图片中，都包含带旋转信息的大分辨率图片，例如3840*2160分辨率及以上的图片，则可能导致图片解码时间过长，从而导致用户进程超时退出。

处理步骤

步骤1 确定大分辨率的图片是否包含旋转信息。

使用JPEG码流分析工具（例如JPEGsnoop）解析大分辨率的图片，查看其是否包含旋转信息，若Orientation信息为1，则表示不旋转；否则，都带有一定角度的旋转，例如下图解析出来的Orientation信息为8，表示顺时针旋转270°。


```
EXIF IFD0 @ Absolute 0x00000026
  Dir Length = 0x0005
  [Orientation                        ] = 8 = Row 0: left, Col 0: bottom
  [Software                           ] = "ACD Systems ........"
  [DateTime                           ] = "2018:04:10 10:43:17"
  [YCbCrPositioning                   ] = Centered
  [ExifOffset                         ] = @ 0x0074
  Offset to Next IFD = 0x00000000
```

步骤2 如果确定这些图片是带旋转的大分辨率图片，则建议用户先调用第三方库（例如OpenCV）进行解码。

----结束

14.3.4.4 JPEGD 图片解码失败

现象描述

JPEGD模块解码失败，查看日志有类似如下报错信息：

日志信息（1）：

```
just support jpeg with YUV 444 422 420 400
do not support progressive mode
do not support arithmetic code, support huffman code only
```

日志信息（2）：

```
EOI segment of the stream is invalid
```

可能原因

分析上面日志信息，可能存在以下可能原因：

1. 数据格式不支持

   JPEGD只支持huffman编码(colorspace: yuv, subsample: 444/422/420/400)，不支持算术编码，不支持渐进编码，不支持jpeg2000格式。

2. 图像数据不完整

处理步骤

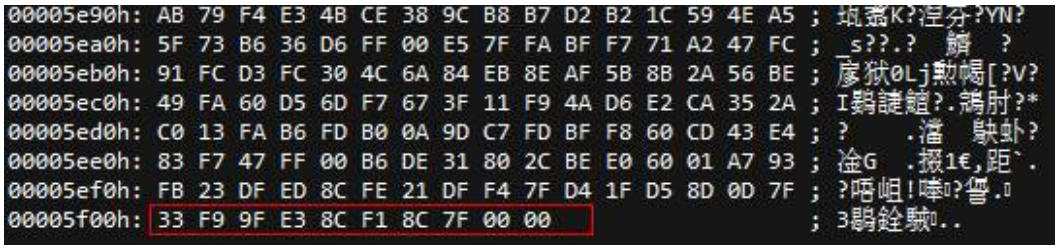
针对上述可能原因，请按以下方式处理：

步骤1 针对目前不支持的超规格图像格式，建议用户自行使用第三方软件解码。

步骤2 针对图像数据不完整，根据报错提示，通过第三方软件查看原图像二进制进行确认。

例如“EOI segment of the stream is invalid”或“EOI segment of the stream is invalid, it should be FFD9. Try software decoding.”报错，表示图像缺失最后的EOI结束符，对应图像二进制类似下图所示。正常JPEG图片最后应该由标记码FF D9结束，该数据最后缺失FF D9标记码。

如果确认原图数据不完整，报错属于正常现象，需更换数据。



步骤3 如果原图像数据完整，可能数据在传输过程中存在损坏，需要在调用图片编码之前，通过fwrite函数将传输给JPEGD的码流保存下来，与原图进行二进制比较。如果不一致，传输过程出现数据缺失，需自行进一步定位传输过程数据缺失问题。

----结束

14.3.4.5 VDEC 视频解码丢帧/丢包

现象描述

视频解码丢帧，出现重影或不连续现象。查看Device侧日志，发现日志中存在以下几个报错的内容信息中的一个或多个，不同版本日志信息可能有所不同。

- H264码流缺少IDR帧日志报错信息（1）

  ```
  [HiDvpp][A618] [Vfmw]:ppssps_check [Line]:6803 pps with this pic_parameter_set_id = %d haven't decode
  [HiDvpp][A618] [Vfmw]:process_slice_header_first_part [Line]:7401 PPS or SPS of this slice not valid
  [HiDvpp][A618] [Vfmw]:h264_dec_slice [Line]:7915 sliceheader dec err
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-23:51:51.339.518 [VDEC][PPSSPSCheckTmpId:7065][T3] PPSSPSCheckTmpId: pps with this pic_parameter_set_id = 0 haven't decode
  [ERROR] DVPP:2020-12-31-23:51:51.339.616 [VDEC][ProcessSliceHeaderFirstPart:7627][T3] PPS or SPS of this slice not valid
  [ERROR] DVPP:2020-12-31-23:51:51.339.678 [VDEC][InquiresSlceProperty:10582][T3] sliceheader dec err
  ```

- H264码流缺少I帧日志报错信息（2）

  ```
  [HiDvpp][A618] [Vfmw]:h264_dec_slice [Line]:7983 init pic err, find next recover point or next valid sps, pps, or exit
  [HiDvpp][A618] [Vfmw]:h264_dec_slice [Line]:3716 dec list error, ret=-1
  [HiDvpp][A618] [Vfmw]:receive_packet [Line]:10676 nal_release_err
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-20:51:51.318.218 [VDEC][InitPic:6039][T3] line 6039: frame gap(=48) > dpb size(=2)
  [ERROR] DVPP:2020-12-31-20:51:51.318.266 [VDEC][H264_DecSlice:8238][T3] init pic err, find next recover point or next valid sps, pps, or exit
  [ERROR] DVPP:2020-12-31-20:51:51.318.336 [VDEC][H264_DecOneNal:10077][T3] DecList error, ret=-1
  [ERROR] DVPP:2020-12-31-20:51:51.318.392 [VDEC][ReceivePacket:10400][T3] nal_release-err
  ```

- H264码流缺少P帧日志报错信息（3）

  ```
  [HiDvpp][A618] [Vfmw]:init_list_x [Line]:4829 for P slice size of list equal 0.ctx->dpb.ref_frames_in_buffer:0.
  [HiDvpp][A618] [Vfmw]:dec_list [Line]:5068 init list error.
  [HiDvpp][A618] [Vfmw]:h264_dec_list [Line]:4829 dec_list error, ret=-1
  [HiDvpp][A618] [Vfmw]:h264_dec_one_nal [Line]:10298 slice_check failed, clear current slice.
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-20:30:22.188.008 [VDEC][InitListX:4513][T3] for P slice size of list equal 0.
  [ERROR] DVPP:2020-12-31-20:30:22.188.056 [VDEC][DecList:4832][T3] line: 4832 InitListX failed
  [ERROR] DVPP:2020-12-31-20:30:22.188.128 [VDEC][H264_DecSlice:8260][T3] DecList error, ret=-1
  [ERROR] DVPP:2020-12-31-20:30:22.188.199 [VDEC][H264_DecOneNal:10077][T3] Decoder Slice failed
  ```


- H264码流缺少B帧日志报错信息（4）

  ```
  [HiDvpp][A618] [Vfmw]:init_list_x [Line]:4865 for B slice size of two list all equal 0.
  [HiDvpp][A618] [Vfmw]:dec_list [Line]:5068 init list error.
  [HiDvpp][A618] [Vfmw]:h264_dec_list [Line]:4829 dec_list error, ret=-1
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-10:20:28.528.090 [VDEC][InitListX:4653][T3] for B slice size of two list all equal 0.
  [ERROR] DVPP:2020-12-31-10:20:28.528.168 [VDEC][DecList:4830][T3] line: 4832 InitListX failed
  [ERROR] DVPP:2020-12-31-10:20:28.528.266 [VDEC][H264_DecSlice:8257][T3] DecList error, ret=-1
  ```

- H265码流缺少IDR帧日志报错信息（5）

  ```
  [HiDvpp][A618] [Vfmw]:hevc_vps_sps_pps_check [Line]:7300 pps with this pic_parameter_set_id = 0 haven't be decoded
  [HiDvpp][A618] [Vfmw]:hevc_dec_slice_segment_header [Line]:3857 hevc_vps_sps_pps_check != HEVC_DEC_NORMAL
  [HiDvpp][A618] [Vfmw]:hevc_inquire_slice_property [Line]:9004 hevc_dec_slice_segment_header dec err
  [HiDvpp][A618] [Vfmw]:hevc_dec_decode_packet[Line]:9004 hevc_inquire_slice_property error.
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-10:30:22.130.500 [VDEC][HEVC_VpsSpsPpsCheck:8084][T10] pps with this pic_parameter_set_id = 0 haven't be decoded
  [ERROR] DVPP:2020-12-31-10:30:22.130.598 [VDEC][HEVC_DecSliceSegmentHeader:2793][T10] HEVC_VpsSpsPpsCheck != HEVC_DEC_NORMAL
  [ERROR] DVPP:2020-12-31-10:30:22.130.686 [VDEC][HEVC_InquireSliceProperty:10169][T10] HEVC_DecSliceSegmentHeader dec err
  [ERROR] DVPP:2020-12-31-10:30:22.130.789 [VDEC][HEVCDEC_DecodePacket:753][T10] HEVC_InquireSliceProperty error.
  ```

- H265码流缺少I帧或者P帧日志报错现象（6）

  ```
  [HiDvpp][A618] [Vfmw]:hevc_ref_pic_process [Line]:3474 ref frame(poc 15) lost.
  [HiDvpp][A618] [Vfmw]:hevc_create_lost_picture [Line]:5839 DPB no suited fs for lost pic.
  [HiDvpp][A618] [Vfmw]:hevc_create_lost_picture [Line]:5847 take poc(17) to create lost poc(15).
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-11:22:28.800.158 [VDEC][HEVC_RefPicProcess:2480][T10] Ref frame(poc 15) lost.
  [ERROR] DVPP:2020-12-31-11:22:28.800.236 [VDEC][HEVC_CreateLostPicture:6392][T10] DPB no suited fs for lost pic.
  [ERROR] DVPP:2020-12-31-11:22:28.800.352 [VDEC][HEVC_RefPicProcess:2480][T10] Ref frame(poc 18) lost.
  [ERROR] DVPP:2020-12-31-11:22:28.800.426 [VDEC][HEVC_CreateLostPicture:6392] [T10] DPB no suited fs for lost pic.
  [ERROR] DVPP:2020-12-31-11:22:28.800.522 [VDEC][HEVC_RefPicProcess:2480] [T10] Ref frame(poc 18) lost.
  ```

- H265码流缺少I帧或者B帧日志报错信息（7）

  ```
  [ERROR] DVPP:2020-12-31-11:56:35.038.109 [VDEC][HEVC_RefPicProcess:2480] [T56] Ref frame(poc 15) lost.
  [ERROR] DVPP:2020-12-31-11:56:35.038.283 [VDEC][HEVC_CreateLostPicture:6392] [T56] Take poc(17) to create lost poc(15).
  [ERROR] DVPP:2020-12-31-11:56:35.038.502 [VDEC][FSP_SetRef:934] [T56] check condition: pstLogicFs->IsDummyFs == 0 fail
  [ERROR] DVPP:2020-12-31-11:56:35.038.801 [VDEC][FSP_SetRef:934] [T56] check condition: pstLogicFs->IsDummyFs == 0 fail
  [ERROR] DVPP:2020-12-31-11:56:35.039.128 [VDEC][FSP_SetRef:934] [T56] check condition: pstLogicFs->IsDummyFs == 0 fail
  ```

可能原因

分析上述日志报错信息现象，分别可能存在以下可能原因：

- 日志报错信息（1）可能原因：H264码流缺少IDR帧
- 日志报错信息（2）可能原因：H264码流缺少I帧
- 日志报错信息（3）可能原因：H264码流缺少P帧

- 日志报错信息（4）可能原因：H264码流缺少B帧
- 日志报错信息（5）可能原因：H265码流缺少IDR帧
- 日志报错信息（6）可能原因：H265码流缺少I帧或者P帧
- 日志报错信息（7）可能原因：H265码流缺少I帧或者B帧

处理步骤

针对可能原因分析，参考以下步骤处理：

步骤1 检查输入的源码流是否有问题。

使用第三方工具（如：eseye u）对输入码流进行检查，查看码流是否异常。

步骤2 若查看的源码流结果为正常，则可能码流在传输给设备侧VDEC的过程中遭到破坏，需要在调用发送码流接口之前，通过fwrite函数将输送给VDEC的码流保存下来。

- 使用第三方工具对保存的码流进行检查，如果码流异常，用户需自行排查将码流送进去之前是否有送流问题。
- 通过对应版本的sample，解码这段保留下来的码流，验证码流是否正常或VDEC是否支持该格式。

  如果sample 解码正常，那就是开发代码有问题，可以参考VDEC示例代码，找到对应的视频解码的代码参考优化。

----结束

14.3.4.6 VDEC 视频解码花屏

现象描述

输入码流给VDEC进行解码，得到的解码数据不正确，产生花屏现象，如图14-5所示。并且日志中存在类似“decode error”、“input stream error, can't decode, report to user”、“num_ref_idx_l0_active(30) out of range(0,16)”、“dvpp_vdec_vdm_process failed”、“Chan 0 ErrRatio = 44”信息。

图 14-5 视频花屏

可能原因

输入的码流中某些帧数据不完整、存在坏帧，导致硬件解码产生花屏。

处理步骤

针对可能原因分析，参考以下步骤处理：

步骤1 检查输入的源码流是否有问题。

使用第三方工具（如：eseye u等）对输入码流进行解码播放，查看是否存在花屏，若不花屏则进行**步骤2**；若花屏则替换输入码流。

步骤2 若查看的源码流结果为正常，则可能码流在传输给设备侧VDEC的过程中遭到破坏，需要在调用发送码流接口之前，通过fwrite函数将输送给VDEC的码流保存下来。

- 使用第三方工具对保存的码流进行检查，如果码流异常，用户需自行排查将码流送进去之前是否有送流问题。
- 通过对应版本的sample，解码这段保留下来的码流，验证码流是否正常或VDEC是否支持该格式。

  如果sample 解码正常，那就是开发代码有问题，可以参考**9 媒体数据处理（含图像/视频等）**中的VDEC示例代码，找到对应的视频解码的代码参考优化。

----结束

14.3.4.7 VDEC 视频解码性能问题

现象描述

VDEC解码性能下降，小于VDEC对外发布的性能规格，导致发生卡顿等现象。

可能原因

分析VDEC性能下降导致的卡顿故障，可能原因有：

- 视频解码回调函数中耗时过长，影响解码性能。
- 输入码流中I帧比例过大，解码I帧耗时比解码P帧耗时长，影响解码性能。
- 输入码流中存在异常帧，影响解码性能。

处理步骤

针对上述可能的故障原因，可以参考以下方式进行处理：

步骤1 在回调函数中打点测试耗时，查看耗时是否过长(回调函数允许的最大耗时和帧率相关，计算公式为：最大耗时=1/帧率，例如帧率=30fps，则最大耗时=1/（30fps）=0.033s)。

步骤2 使用第三方工具打开输入码流，查看I帧比例是否过大。一般GOP值为30(即I帧间隔为30)，如果I帧比例过大，则需要替换成正常码流进行性能测试。

步骤3 使用第三方工具打开输入码流，查看是否存在异常帧（如用第三方工具打开显示花屏或解码报错），如果存在异常帧，会造成不满足规格的现象。

----结束

14.3.4.8 输入码流不符合要求导致 VDEC 视频解码失败

现象描述

视频解码失败，日志中打印如下所示的异常信息，不同版本日志信息可能有所不同：

- 日志信息：Invalid width or height

  ```
  [HiDvpp][A618] [Vfmw] vdec_drv_check_video_wh [Line]:127 device 0 chn 0 type(255) pic size(32x32) is out of range[128,128]x[4096,4096].
  ```

  或

  ```
  [ERROR] DVPP:2020-12-31-20:50:51.318.218 [VDEC][IsValidResolution:639][T56] Invalid width or height, valid range (w:128~4096) (h:128~4096), current width = 32, height = 32, realWidth = 18, realHeight = 18
  [ERROR] DVPP:2020-12-31-20:50:51.318.365 [VDEC][upgrade_picture_info_in_detail:679][T56] check condition: ret == OMX_ErrorNone fail
  [ERROR] DVPP:2020-12-31-20:50:51.318.429 [VDEC][EventHandLer:386][T56] Dynamic Resouces Unavailable now
  [ERROR] DVPP:2020-12-31-20:50:51.368.106 [VDEC][hanlde_release_instance:1352][T56] wait Componment Exit Message Thread
  ```

- 日志信息：bit_depth_luma(*) not equal *

  ```
  [HiDvpp][A618] [Vfmw] process_sps [Line]:8994 bit_depth_luma(%d) not equal 8.
  [HiDvpp][A618] [Vfmw] hevc_process_sps [Line]:1462 chn 0, bit_depth_luma(9) is not supported, hevc only support 8 or 10 bit depth.
  ```

  或

  ```
  [ERROR] DVPP(13757,dvpp_performance):2020-12-31-21:18:26.998.657 [VDEC][ProcessSPS:9055] [T26] bit_depth_luma(10) not epual 8.
  [ERROR] DVPP(13757,dvpp_performance):2020-12-31-21:18:26.998.763 [VDEC][ProcessSPS:9070] [T26] bit_depth_chroma(10) not epual 8.
  ```

可能原因

针对上述日志分析，可能存在输入码流规格不符合软硬件约束。

处理步骤

根据**9 媒体数据处理（含图像/视频等）**中的VDEC功能及约束，使用第三方工具（如：eseye u等）对输入码流进行检查，查看码流类型、码流宽高等信息是否符合要求。如果码流不满足要求，请替换符合要求的码流。

14.3.4.9 VDEC 视频帧解码失败不触发回调函数

现象描述

用户输入码流给VDEC解码，某些帧或所有帧都没有触发回调函数，用户收不到解码结果。

可能原因

码流中某些帧是坏帧，导致语法解析不出这些帧的含义，或者解码这些帧失败，从而不调用回调函数。

处理步骤

针对分析的可能原因，请参考以下步骤进行处理：

步骤1 查看日志中是否有**14.3.4.5 VDEC视频解码丢帧/丢包**中的日志报错信息，若有，则是因为异常帧解码失败导致没有回调。

步骤2 若没有**14.3.4.5 VDEC视频解码丢帧/丢包**中的日志报错信息，则调整DVPP模块或全局日志级别为info，查看下述三条日志打印的总次数是否和输入的帧数相等。

- （1）"The queue is empty, so call the non-intelligent pointer callback interface."
- （2）"The queue is not empty, so call the smart pointer callback interface."
- （3）"The queue is not empty, but hiai_data_sp is nullptr."

生成上述3种日志信息的场景如下：

- 未使用hiai_data_sp，成功解码返回，打印（1）日志，然后调用用户注册的回调函数。
- 每一帧对应设置一个hiai_data_sp，成功解码返回，打印（2）日志，然后调用用户注册的回调函数。
- N帧对应设置一个hiai_data_sp，第1帧成功解码返回，打印（2）日志，然后调用用户注册的回调函数；其他N-1帧，成功解码后打印（3）日志信息，并调用用户注册的回调函数。

不论上述场景中哪一种，均会调用用户注册的回调函数，即只要调用一次用户回调函数就说明解码返回一帧。所以上述三条日志出现的次数总和与用户输入总帧数相等，则说明无解码丢帧，用户需排查自身接收解码结果的统计是否有误。

----结束

14.3.4.10 VDEC 视频解码异常导致进程卡死，无法退出

现象描述

用户进程卡死，无法退出。

查看应用类日志，一直重复提示信息“**fault kernel_name=DvppSendVdecFrame**”、“**Kernel task happen error, retCode=0x28, [aicpu timeout]**”，表示AI CPU异常，无法处理VDEC解码任务，导致任务超时。

日志片段举例如下：

```
[ERROR] RUNTIME(pid,pName):DateTimeMS [task.cc:878]1827 PreCheckTaskErr:[DVPP][DEFAULT]Kernel task happen error, retCode=0x28, [aicpu timeout].
[ERROR] RUNTIME(pid,pName):DateTimeMS [task.cc:676]1827 PrintAicpuErrorInfo:[DVPP][DEFAULT]Aicpu kernel execute failed, device_id=0, stream_id=177, task_id=4, fault so_name=libdvpp_kernels.so, fault kernel_name=DvppSendVdecFrame, fault op_name=, extend_info=.
[ERROR] RUNTIME(pid,pName):DateTimeMS [task.cc:878]1831 PreCheckTaskErr:[DVPP][DEFAULT]Kernel task happen error, retCode=0x28, [aicpu timeout].
[ERROR] RUNTIME(pid,pName):DateTimeMS [task.cc:676]1831 PrintAicpuErrorInfo:[DVPP][DEFAULT]Aicpu kernel execute failed, device_id=0, stream_id=170, task_id=8, fault so_name=libdvpp_kernels.so, fault kernel_name=DvppSendVdecFrame, fault op_name=, extend_info=.
[ERROR] RUNTIME(pid,pName):DateTimeMS [engine.cc:960]1766 ReportExceptProc:[DVPP][DEFAULT]Task exception! device_id=0, stream_id=107, task_id=8, type=1, retCode=0x28.
[ERROR] RUNTIME(pid,pName):DateTimeMS [engine.cc:960]1773 ReportExceptProc:[DVPP][DEFAULT]Task exception! device_id=0, stream_id=130, task_id=4, type=1, retCode=0x28.
```

可能原因

Device内存不足，AI CPU无法处理VDEC解码任务，导致任务超时。

处理步骤

步骤1 在使用媒体数据处理V1版本的VDEC视频解码功能前，可参考功能及约束说明中“每路VDEC解码的内存消耗计算公式”，预估需使用的Device内存，并合理规划Device上的内存。

步骤2 优化应用程序的代码逻辑，增加异常处理机制，获取VDEC解码异常信息，强制退出进程。

在调用aclinit接口之后，定义异常回调函数，并调用aclrtSetExceptionInfoCallback接口设置异常回调函数，用于获取任务异常信息，以便在异常分支中根据任务异常信息来判断是否退出应用进程。

基本接口调用逻辑如下：

1. 定义并实现异常回调函数fn(aclrtExceptionInfoCallback类型)，回调函数原型为：typedef void (*aclrtExceptionInfoCallback)(aclrtExceptionInfo *exceptionInfo)

   在异常回调函数fn内调用aclrtGetDeviceIdFromExceptionInfo、aclrtGetStreamIdFromExceptionInfo、aclrtGetTaskIdFromExceptionInfo接口分别获取Device ID、Stream ID、Task ID。

   根据Stream ID、Task ID判断Device是否异常，若异常，则强制退出进程。

   异常回调函数实现示例如下：

   ```
   void dvpp_callback(aclrtExceptionInfo * exception_info)
   {
       uint32_t taskId = aclrtGetTaskIdFromExceptionInfo(exception_info);
       uint32_t streamId = aclrtGetStreamIdFromExceptionInfo(exception_info);
       uint32_t deviceId = aclrtGetDeviceIdFromExceptionInfo(exception_info);

       if(taskId == 0xffffffff) || (streamId == 0xffffffff) {
           //Device异常，强制退出进程
       } else {
           //任务异常，如果频繁出现（例如，统计1秒内触发异常回调函数的次数），进程退出
       }
       return;
   }
   ```

2. 调用aclrtSetExceptionInfoCallback接口设置异常回调函数。
3. 执行VDEC解码，接口调用流程请参见**9.4.5 VDEC视频解码**。

----结束

14.3.4.11 retCode 返回值设置错误，导致 VDEC 视频解码异常

现象描述

调用aclvdecSendFrame接口发送一帧码流后，继续复用output输出图片描述信息，进行后续帧码流的解码操作，结果反复出现解码不成功、解码异常的情况。

日志片段举例如下：

```
Channel[0]: success to aclvdecSendFrame, loop=1, count=7
get frame success, totalCount=7
packet.size is 26084.
Channel[0]: begin to send frame, loop=1, count=8
acldvppGetPicDescRetCode, retCode: 2.
Vdec ERROR!!!!!!!!!!!!!!!!
errCount is 3. total count is 3.
!!!!!!!!!!!!!!!!!!acldvppGetPicDescRetCode, retCode: 2.right_count:0,fail_count:3,total_count:3
Channel[0]: success to aclvdecSendFrame, loop=1, count=8
get frame success, totalCount=8
```



```
packet.size is 27927.
Channel[0]: begin to send frame, loop=1, count=9
acldvppGetPicDescRetCode, retCode: 2.
Vdec ERROR!!!!!!!!!!!!!!!!!
errCount is 4. total count is 4.
!!!!!!!!!!!!!!!!!!acldvppGetPicDescRetCode, retCode: 2.right_count:0,fail_count:4,total_count:4
```

可能原因

根据日志中的提示，通过acldvppGetPicDescRetCode接口获取到的retCode为2，retCode为非0值时，表示解码异常。

再查看代码逻辑时，发现由于前一帧码流解码失败，retCode被置为2，在复用output输出图片描述信息时，retCode也继承了前一帧解码失败的状态值2，导致AscendCL在解码后续帧时，获取到retCode值为2，就一直判断解码是失败。

处理步骤

如果存在复用output输出图片描述信息的场景，需先调用acldvppSetPicDescRetCode设置为0，防止前一帧解码异常的状态影响后续解码。

14.3.4.12 VDEC 视频解码无报错，但无解码结果数据、CPU 占用率高

现象描述

查看应用类日志，无ERROR报错、无解码结果数据输出，另外，在运行应用程序的Linux服务器上执行top命令，该应用进程的cpu占用率持续升高。

可能原因

1. 无ERROR、无解码结果数据输出，判断可能是因为解码发帧接口aclvdecSendFrame调用正常，但未触发回调函数，无法获取解码结果数据。
2. 检查触发回调函数的代码逻辑。

   按照VDEC视频解码的接口调用逻辑：由用户提前创建一个单独的线程，并自定义线程函数，在线程函数内调用aclrtProcessReport接口，通过该接口配置超时时间，等待指定的超时时间后，触发回调函数，获取解码结果数据。

   ```
   void *ThreadFunc(aclrtContext sharedContext)
   {
       if (sharedContext == nullptr) {
           ERROR_LOG("sharedContext can not be nullptr");
           return ((void*)(-1));
       }
       INFO_LOG("use shared context for this thread");
       aclError ret = aclrtSetCurrentContext(sharedContext);
       if (ret != ACL_SUCCESS) {
           ERROR_LOG("aclrtSetCurrentContext failed, errorCode = %d", static_cast<int32_t>(ret));
           return ((void*)(-1));
       }

       INFO_LOG("thread start ");
       while (runFlag) {
           // Notice: timeout 1000ms
           (void)aclrtProcessReport(1000);
       }
       return (void*)0;
   }
   ```

3. 如果触发回调函数的接口调用逻辑正确，则在aclrtProcessReport接口处增加日志打印，判断应用运行过程中线程是否成功调用了aclrtProcessReport接口，只有成功调用aclrtProcessReport接口，才会触发回调函数。

示例代码如下：

```
while (runFlag) {
        // Notice: timeout 1000ms
        aclError ret = aclrtProcessReport(1000);
       printf("aclrtProcessReport failed, ret=%d.\n", ret);
}
```

4. 修改代码后，重新编译运行应用。

   在终端屏幕重复出现以下打印信息，表示调用aclrtProcessReport接口失败：

   ```
   aclrtProcessReport failed, ret = 107012
   ```

   查阅该接口的返回值说明，107012表示线程未订阅或重复订阅。

5. 检查代码逻辑，检查是否调用aclvdecSetChannelDescThreadId接口绑定用户新建的线程，按照VDEC视频解码的接口调用逻辑，只有调用该接口绑定用户线程，才可以触发调用aclrtProcessReport接口，进而触发回调函数。
6. 修改代码后，重新编译运行应用，VDEC视频解码正常，正常输出解码结果数据，同时cpu占用率下降。

解决办法

参见VDEC的**9.4.5 VDEC视频解码**或者参考**Link**上的VDEC功能样例开发VDEC功能。

其中，需关注以下注意点：

- 创建新线程，并自定义线程函数，在线程函数内调用aclrtProcessReport接口，等待指定时间后，触发回调函数中的回调函数。
- 需调用aclvdecSetChannelDescThreadId接口绑定用户创建的新线程。
- 释放资源时，依次销毁通道、销毁通道描述信息后，才可以销毁中用户创建的新线程。

14.3.4.13 复用输出图片描述类型，VDEC 视频解码报错，提示有不支持的图片格式

问题现象

循环调用aclvdecSendFrame接口解码视频中的每一帧码流时，在遇到异常帧之后，解码下一帧就会报错，退出应用进程。

分别查看Host侧日志、Device侧日志，发现Device日志中提示**the out format 0 is not supported**，日志片段如下：

- Device侧日志：

```
[ERROR] KERNEL(2234,sklogd):2023-06-13-19:21:22.987.969 [klogd.c:246][652145.056916] [HiDvpp] [A618] [Vdec]:vdec_check_resize_param [Line]:6768 pid 23973 usr chn 0 device 0 chn 0 the out format 0 is not supported.
```

- Host侧日志：

```
[ERROR] RUNTIME(17174,AIMCDemo):2023-06-13-19:21:23.664.211 [api_c.cc:721]17184 rtStreamSynchronize:[DVPP][DEFAULT]ErrCode=507018, desc=[aicpu exception], InnerCode=0x715002a
[ERROR] RUNTIME(17174,AIMCDemo):2023-06-13-19:21:23.664.215 [error_message_manage.cc:49]17184 FuncErrorReason:[DVPP][DEFAULT]report error module_type=3, module_name=EE8888
[ERROR] RUNTIME(17174,AIMCDemo):2023-06-13-19:21:23.664.221 [error_message_manage.cc:49]17184 FuncErrorReason:[DVPP][DEFAULT]rtStreamSynchronize execute failed, reason=[aicpu exception]
[INFO] GE(17174,AIMCDemo):2023-06-13-19:21:23.664.227 [error_manager.cc:252]17184 ReportInterErrMessage:report error_message, error_code:EE8888, work_stream_id:1717417184
[ERROR] ASCENDCL(17174,AIMCDemo):2023-06-13-19:21:23.664.234 [video_processor_v200.cpp:1089]17184 aclvdecSendFrame: [DVPP][DEFAULT][Sync][Stream]vdec fail to synchronize sendFrameStream, runtime errorCode = 507018, channelId = 0.
```

原因分析

检查应用代码，发现循环解码视频中的每一帧码流时，复用aclvdecSendFrame接口的输出图片描述类型acldvppPicDesc，但在下一次解码前没有重新设置输出图片format、width、height、widthStride、heightStride，这时，如果前一帧解码失败，acldvppPicDesc的参数format、width、height、widthStride、heightStride变成默认值0，width、height、widthStride、heightStride为0时，vdec会以实际图片宽高解码输出，但format为0，表示YUV400格式，vdec不支持解码输出该格式，会导致下一帧参数不合法解码失败。

解决方法

优化应用代码逻辑，复用输出图片描述类型acldvppPicDesc时，在下一次解码前需重新设置输出图片format、width、height、widthStride、heightStride。

正例代码片段：

```
aclError ret;
int restLen = 10;
uint32_t inBufferSize = 0;
void *g_picOutBufferDev;
void *inBufferDev = nullptr;
acldvppPicDesc *picOutputDesc;
size_t dataSize = (INPUT_WIDTH * INPUT_HEIGHT * 3) / 2;

// 申请一个picOutputDesc,每帧复用
picOutputDesc = acldvppCreatePicDesc();
// read file to device memory
 ReadFileToDeviceMem(filePath.c_str(), inBufferDev, inBufferSize);
 while (restLen > 0) {
        // 等待前一个使用picOutputDesc解码帧结束，重新复用picOutputDesc,并针对这一帧重新设置Format、width、height、widthStride、heightStride参数值
        ret = acldvppSetPicDescFormat(picOutputDesc, static_cast<acldvppPixelFormat>(1)); // 1：YUV420 semi-planner（nv12）
        ret = acldvppSetPicDescWidth(picOutputDesc, 1920);
        ret = acldvppSetPicDescHeight(picOutputDesc, 1080);
        ret = acldvppSetPicDescWidthStride(picOutputDesc, 1920);
        ret = acldvppSetPicDescHeightStride(picOutputDesc, 1080);
        ret = acldvppMalloc(&g_picOutBufferDev, dataSize);
        ret = acldvppSetPicDescData(picOutputDesc, g_picOutBufferDev);
        ret = acldvppSetPicDescSize(picOutputDesc, dataSize);
        ret = aclvdecSendFrame(vdecChannelDesc, streamInputDesc, picOutputDesc, nullptr, nullptr);
        restLen = restLen - 1;
}
```

反例代码片段：

```
aclError ret;
int restLen = 10;
uint32_t inBufferSize = 0;
void *g_picOutBufferDev;
void *inBufferDev = nullptr;
acldvppPicDesc *picOutputDesc;
size_t dataSize = (INPUT_WIDTH * INPUT_HEIGHT * 3) / 2;

// 申请一个picOutputDesc，每帧复用，且对Format、width、height、widthStride、heightStride参数值只设置了一次
picOutputDesc = acldvppCreatePicDesc();
ret = acldvppSetPicDescFormat(picOutputDesc, static_cast<acldvppPixelFormat>(1)); // 1：YUV420 semi-planner（nv12）
ret = acldvppSetPicDescWidth(picOutputDesc, 1920);
ret = acldvppSetPicDescHeight(picOutputDesc, 1080);
ret = acldvppSetPicDescWidthStride(picOutputDesc, 1920);
ret = acldvppSetPicDescHeightStride(picOutputDesc, 1080);
// read file to device memory
ReadFileToDeviceMem(filePath.c_str(), inBufferDev, inBufferSize);
while (restLen > 0) {
```



```
        // 等待前一个使用picOutputDesc解码帧结束，重新复用picOutputDesc,但没有重新设置Format、width、height、widthStride、heightStride参数值
    // 如果前一帧解码失败，picOutputDesc_的参数Format、width、height、widthStride、heightStride变成默认值0，
    // width、height、widthStride、heightStride为0时，vdec会以实际图片宽高解码输出，但Format为0，表示YUV400格式，vdec不支持解码输出该格式，会导致本帧参数不合法解码失败
        ret = acldvppMalloc(&g_picOutBufferDev, dataSize);
        ret = acldvppSetPicDescData(picOutputDesc, g_picOutBufferDev);
        ret = acldvppSetPicDescSize(picOutputDesc, dataSize);
        ret = aclvdecSendFrame(vdecChannelDesc, streamInputDesc, picOutputDesc, nullptr, nullptr);
        restLen = restLen - 1;
    }
```

14.3.4.14 异常码流导致 VDEC 视频解码失败

问题现象

调用hi_mpi_vdec_get_frame接口成功，返回的解码结果为失败(frame_info->v_frame.frame_flag = 1)，表示可能由于输入码流异常、设置的码流格式与实际码流不一致等问题导致解码失败。

常见语法解析日志报错示例如下：

- 日志示例1
  ```
  pid 0 usr chn 0 device 0 chn 0, input stream error, can't decode, report to user
  ```
- 日志示例2
  ```
  ppssps_check_tmp_id: pps is null with this pic_parameter_set_id = 1 haven't decode
  ```
- 日志示例3
  ```
  PPS or SPS of this slice not valid
  ```
- 日志示例4
  ```
  sliceheader dec err
  ```
- 日志示例5
  ```
  H264DEC inquire_slice_property error
  ```
- 日志示例6
  ```
  hevc_inquire_slice_property error
  ```
- 日志示例7
  ```
  ref frame(poc 15) lost
  ```
- 日志示例8
  ```
  SH hevc_dec_short_term_ref_pic_set error
  ```
- 日志示例9
  ```
  p_temp_r_pset->num_negative_pics(66) out of range(0,15)
  ```

原因分析

frame_flag = 1解码失败，一般是由于送给VDEC视频解码模板的输入码流存在异常，或者网络不稳定导致播包丢帧导致码流异常，所以解码会报错失败，针对这种情况，可以通过保存输入码流来进行判断输入码流是否异常。

解决方法

1. 在调用hi_mpi_vdec_send_stream接口成功之后，将输入码流进行写文件保存，可参考如下代码。
   - 适用于Ascend RC形态：
     ```
     ret = hi_mpi_vdec_send_stream(chn, stream, vdec_pic_info, milli_sec);
     if (ret == HI_SUCCESS) {
         FILE *fd = NULL;
     ```



```
    fd = fopen("input_stream", "a+");
    fwrite(stream->addr, stream->len, 1, fd);
    fclose(fd);
}
```

- 适用于Ascend EP形态：

  需要先将码流数据从Host传输至Device，再发送解码码流。

```
 aclrtMemcpy(stream->addr, stream->len, buf, size, ACL_MEMCPY_HOST_TO_DEVICE);
ret = hi_mpi_vdec_send_stream(chn, stream, vdec_pic_info, milli_sec);
if (ret == HI_SUCCESS) {
    FILE *fd = NULL;
    fd = fopen("input_stream", "a+");
    fwrite(buf, size, 1, fd);
    fclose(fd);
}
```

2. 使用第三方工具查看保存下来的input_stream码流，检查是否存在异常，例如花屏、报错提示等。

14.3.4.15 非 annex-B 格式的码流导致 VDEC 视频解码失败

问题现象

解码失败。

日志示例如下：

```
pid 0 usr chn 0 device 0 video format unsupport at event chn 0
```

或

```
pid 0 usr chn 0 device 0 chn 0, input stream error, can't decode, report to user
```

原因分析

VDEC视频解码模块目前只支持annex-B格式的裸码流，不支持其它格式（例如AVCC格式）的裸码流。

解决方法

annex-B格式裸码流，码流样式如下，固定以0x00000001开始（以H264为例），建议用户使用第三方工具打开码流，以16进制方式查看码流并排查格式。

```
<pre><code>0x0000 | 00 00 00 01 67 64 00 0A AC 72 84 44 26 84 00 00
```

14.3.4.16 不支持的协议字段 0x31 导致 JPEGD 图片解码结果异常

问题现象

JPEGD图片解码的结果数据都是0，查看日志，有报错“**Unsupported marker type 0x31**”，日志示例如下：

```
[ERROR] KERNEL(1234,sklogd):2023-10-27-00:39:44.324.726 [701889.574406]  [dvpp] [dvpp_check_decode_status 954] decode unfinish, image height:1440, the decoded line num:1424
[ERROR] KERNEL(1234,sklogd):2023-10-27-00:39:44.324.752 [701889.574946]  [dvpp] [dvpp_jpegd_engine_proc 412] jpegd_done_config failed! errcode:-1, engine id:1
[ERROR] KERNEL(1234,sklogd):2023-10-27-00:39:44.324.777 [701889.575016]  [dvpp] [jpegd_res_off 653] engine 1 decode error, no need to turn off decoder clock
[ERROR] KERNEL(1234,sklogd):2023-10-27-00:39:44.324.804 [701889.575020]  [dvpp] [dvpp_ioctl_jpegd
```



```
794] call proc failed:-1, engine_id:1
[ERROR] DVPP(14577,graph_1):2023-10-27-00:39:44.411.853 [JPEGD] [SoftwareProcess:267] [T87]
tjDecompressToYUV2 fail: Unsupported marker type 0x31
[ERROR] DVPP(14577,graph_1):2023-10-27-00:39:44.411.977 [JPEGD] [Process:312] [T87] Jpeg hardware
and software decode are both failed!
```

原因分析

存在JPEGD图片解码模块不支持的协议字段0x31，导致硬解和软解都失败，因此没有输出解码数据，即内存中数据都是默认值0。

以下是常用标记的标记代码和表示的意义：

标记	标记代码	意义
SOI	0xFFD8	图像开始
APP0	0xFFE0	应用程序保留标记0
DQT	0xFFDB	定义量化表
SOF0	0xFFC0	帧图像开始
DHT	0xFFC4	定义哈夫曼表
SOS	0xFFDA	扫描开始
EOI	0xFFD9	图像结束

解决方法

1. 建议用户使用第三方工具打开图片码流，检查码流中的标记。
2. 如果存在0x31的标记，则更换图片，重新解码。

14.3.4.17 码流大小校验失败导致 VDEC 视频解码失败

问题现象

调用hi_mpi_vdec_send_stream接口返回错误码HI_ERR_VDEC_ILLEGAL_PARAM，查看应用类日志，有报错“**stream len can't be zero!**”，日志示例如下：

```
[Vdec]:vdec_check_send_stream [Line]:6563 pid 578 usr chn 64 device 0 chn 64 stream len can't be zero!
```

原因分析

查看日志中报错stream len can't be zero!，被VDEC代码参数校验拦截，建议排查传入的参数stream->len或stream->addr是否合法。

解决方法

检查代码中的stream->len参数值，发现len为0，修改参数赋值代码逻辑。

14.3.4.18 实际码流类型与接口中设置的解码器类型不一致导致 VDEC 视频解码失败

问题现象

每一帧解码都失败。

Device侧日志示例如下：

```
temporal_id(-1) is not supported.
sps id 2 is larger than 2.
pid 2487 usr chn 4 device 0 video format unsupport at event chn 4
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

创建VDEC视频解码通道时，需设置解码协议类型，VDEC视频解码涉及HI_PT_H264、HI_PT_H265两种类型，若设置的解码协议类型与实际解码码流的类型不一致时，会出现以上解码失败问题。

解决方法

需排查hi_mpi_vdec_create_chn接口传入的入参，attr->type是HI_PT_H264还是HI_PT_H265，并和码流实际类型做对比。

14.3.4.19 多帧码流当一帧送入解码导致 VDEC 视频解码异常

问题现象

解码发生丢帧，有ERROR日志。

Device侧日志示例如下：

```
User send more than one stream date, but only send one outbuf
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

VDEC视频解码当前仅支持按帧发送模式，因此调用hi_mpi_vdec_send_stream接口时，要求用户每次送入独立的一帧码流数据以及对应的输出buffer，当用户一次送入多帧数据（即输入码流内存中有多帧码流数据），这时VDEC视频解码时除了第一帧解码成功外，其余帧都会被丢弃，同时打印ERROR日志。

解决方法

需排查代码逻辑，检查hi_mpi_vdec_send_stream接口的输入码流内存（stream->addr参数）中是否一次读入多帧数据，若是则需调整代码逻辑。

14.3.4.20 送帧太快，VDEC 视频解码发生 OOM

问题现象

进行多路解码时，送帧间隔设置太短，导致发帧过快，出现OOM现象，解码卡住。

Device日志示例如下：

```
OOM_NOTIFIER: oom type 2
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

每送一帧数据都需要申请一个输出buffer，当送帧间隔设置太短时，远远超过解码帧率时，则需申请大量的输出buffer，占用大量的内存，导致产生OOM。

解决方法

根据性能规格数据（参见VDEC性能指标数据），自行调整送帧间隔。

14.3.4.21 解码器帧存大小以及参考帧个数设置过小，VDEC 视频解码卡住

问题现象

解码卡住，日志中提示无法申请帧存的相关错误。

Device侧日志示例如下：

```
pid 0 usr chn 0 device 0 chn 0, user set frame buffer size(1000 Byte) and ref frame num(5) is not enough for actual frame buffer size(2000 Byte) and actual ref frame num(7), pic width = 1280, height = 720, bit_width =8
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

解码器内部需要申请一定个数的帧存，在进行帧存自适应时，如果用户创建通道时设置的**帧存大小*帧存个数**小于实际所需要的**帧存大小*帧存个数**，则解码时会失败。

解决方法

在调用hi_mpi_vdec_create_chn接口创建解码通道时，调整传入的帧存大小和帧存个数的值，即attr->frame_buf_size、attr->frame_buf_cnt参数，或者直接这两个参数设置为0，由解码器内部自适应。

14.3.4.22 昇腾虚拟化实例场景使用不含 DVPP 的算力模板时，创建 VDEC 通道失败

问题现象

无法创建VDEC视频解码通道。

Device侧日志示例如下：

```
pid 0 usr chn 0 device 0 chn 0 vf_id 1 has created chn num(0) is full, max chn num = 0
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

从报错日志提示，可见昇腾虚拟化实例场景下最大通道数为0，怀疑算力模板中不含DVPP，因为若算力模板中不含DVPP，则表示该算力下无DVPP能力，才会导致创建VDEC通道失败，这种情况下的报错属于正常情况。

解决方法

1. 查询算力模板资源信息。

   EP模式下，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令导出Device的日志信息，按导出时间，进入对应时间戳的目录下，打开“module_info\dev-os-0\dvpp\dvpp_proc.log”日志查看对应vf 1下有几个vdec core。

   RC模式下，登录板端环境，执行”cat /proc/umap/sys”，查看对应vf 1下有几个vdec core。

   如下图所示，vf 1下没有分配vdec core，才会导致创建VDEC通道失败，这种情况下的报错属于正常情况。

   ```
   -----Device 0, VF 0 Dvpp Utilization Ratio-------------
   Average Vdec Core Ratio          0
   Vdec Core   0          0
   Vdec Core   1          0
   Vdec Core   2          0
   Vdec Core   3          0
   Vdec Core   4          0
   Vdec Core   5          0
   ```

   ```
   -----Device 0, VF 1 Dvpp Utilization Ratio-------------
   Average Vdec Core Ratio          0
   Average Venc Core Ratio          0
   Average Jpegd Core Ratio         0
   Average Jpege Core Ratio          0
   Average Vpc Core Ratio          0
   ```

2. 若需要使用DVPP能力，可使用含DVPP能力的算力模板，算力模板信息如下所示。

表 14-2 昇腾虚拟化实例配置表（Atlas 推理系列产品（Ascend 310P 处理器））

name（切分名称）	切分规格	AIC核数（算力资源中的aicore数量配置）	内存容量(GB)	AICPU（算力资源中的device_aicpu数量配置）	VPC	JPEGD	JPEGE	VENC	VDEC
总资源	1	8	等于通过dsmi_get_memory_info接口获取的memory_size取值。	7	12	16	8	3	12
vir04	1/2	4	memory_size/2	4	6	8	4	2	6
vir04_3c	1/2	4	memory_size/2	3	6	8	4	1	6
vir04_4c_dvpp	1/2	4	memory_size/2	4	12	16	8	3	12
vir04_3c_ndvpp	1/2	4	memory_size/2	3	0	0	0	0	0
vir02	1/4	2	memory_size/4	2	3	4	2	1	3

name（切分名称）	切分规格	AIC核数（算力资源中的aicore数量配置）	内存容量(GB)	AICPU（算力资源中的device_aicpu数量配置）	VPC	JPEGD	JPEGE	VENC	VDEC
vir02_1c	1/4	2	memory_size/4	1	3	4	2	0	3
vir01	1/8	1	memory_size/8	1	1	2	1	0	1

说明

Atlas 推理系列产品（Ascend 310P处理器）的2P场景下，p0对应的算力模板名称如上表所示，p1对应的算力模板名称均多了p1的标记，比如：p1_vir04。

14.3.4.23 昇腾虚拟化实例场景下 VDEC 通道数量达到上限，无法创建通道

问题现象

创建一定数量的通道后，无法继续创建vdec通道。

Device侧日志示例如下：

```
pid 0 usr chn 0 device 0 chn 0 vf_id 1 has created chn num(27) is full, max chn num = 27
```

说明

EP模式下，运行解码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，可导出Device的日志信息。

RC模式下，登录板端环境，执行**cat /proc/umap/vdec**命令，可导出解码相关信息。

原因分析

不同的算力模板有不同的总通道数上限，超过通道上限就无法继续创建通道。

解决方法

1. 根据所使用的算力模板查询该模版中的VDEC资源信息。

表 14-3 昇腾虚拟化实例配置表（Atlas 推理系列产品（Ascend 310P 处理器））

name（切分名称）	切分规格	AIC核数（算力资源中的aicore数量配置）	内存容量(GB)	AICPU（算力资源中的device_aicpu数量配置）	VPC	JPEGD	JPEGE	VENC	VDEC
总资源	1	8	等于通过dsmi_get_memory_info接口获取的memory_size取值。	7	12	16	8	3	12
vir04	1/2	4	memory_size/2	4	6	8	4	2	6
vir04_3c	1/2	4	memory_size/2	3	6	8	4	1	6
vir04_4c_dvpp	1/2	4	memory_size/2	4	12	16	8	3	12
vir04_3c_ndvpp	1/2	4	memory_size/2	3	0	0	0	0	0

name（切分名称）	切分规格	AIC核数（算力资源中的aicore数量配置）	内存容量(GB)	AICPU（算力资源中的device_aicpu数量配置）	VPC	JPEGD	JPEGE	VENC	VDEC
vir02	1/4	2	memory_size/4	2	3	4	2	1	3
vir02_1c	1/4	2	memory_size/4	1	3	4	2	0	3
vir01	1/8	1	memory_size/8	1	1	2	1	0	1

2. 根据VDEC资源信息计算通道总数上限。
   解码通道总数上限=(JPEGD + VDEC) / (16 + 12) * 256.
   例如vir01模板，解码通道总数上限=(2 + 1) / (16 + 12) * 256 = 27

14.3.5 JPEGE 图片编码/VENC 视频编码

14.3.5.1 调用错误的内存申请接口，导致内存地址校验出错

现象描述

日志报错如下：

- 日志1
  ```
  device:0 chan 0, venc or jpege input buffer is invalid, make sure it has been allocated with hi_mpi_dvpp_malloc or aclDvppMalloc.
  ```
- 日志2
  ```
  device:0 chan 0, venc or jpege output buffer is invalid, make sure it has been allocated with hi_mpi_dvpp_malloc or aclDvppMalloc.
  ```
- 日志3
  ```
  device:0 chan 0, jpege input buffer is invalid, make sure it has been allocated with hi_mpi_dvpp_malloc or aclDvppMalloc.
  ```


- 日志4
  device:0 chan 0, jpege output buffer is invalid, make sure it has been allocated with hi_mpi_dvpp_malloc or aclDvppMalloc.

可能原因

根据日志提示，是由于没有使用指定的接口申请内存。

处理步骤

使用媒体数据处理V1版本中的acldvppMalloc接口/媒体数据处理V2版本中的hi_mpi_dvpp_malloc接口申请DVPP内存，存放JPEGE图片编码/VENC视频编码输入或输出数据。

14.3.5.2 使用正确的内存申请接口，但内存大小传值错误

现象描述

日志报错如下：

device:0 chan 0, output terminal address is invalid, maybe outBufSize:3110400 is invalid.

可能原因

传入的buffer size太大，超出了实际申请的buffer范围，导致内部结束地址校验出错。

处理步骤

如果内存申请接口使用正常，业务流程中dvpp内存申请接口增加地址及长度日志，检查接口hi_mpi_venc_send_frame/hi_mpi_venc_send_jpege_frame/aclvencSendFrame/acldvppJpegEncodeAsync传入buffer长度是否一致。

14.3.5.3 内存被提前释放，导致 VENC 编码数据花屏

现象描述

编码出来的数据花屏，其他无异常日志信息。

可能原因

VENC输入内存是YUV图片数据，输入内存被踩或者被提前释放。

处理步骤

在DVPP内存释放接口处、以及hi_mpi_venc_get_stream/aclvencCallback/acldvppJpegEncodeAsync接口处，增加内存大小及地址的打印日志，确认内存释放时序，是否存在输入内存在编码完成前被提前释放的情况。

14.3.5.4 VENC 创建通道失败

现象描述

调用VENC创建通道的接口hi_mpi_venc_create_chn返回值非0，通道创建失败。

可能原因

导致创建通道失败可能原因有以下：

- 用户传入的通道ID超出了规定的合法范围，VENC规定通道ID在[0,255]总共256个通道。
- 用户传入的通道属性参数不在规定的合理范围内或设置了暂不支持的参数。
- 用户试图创建已经存在的通道，比如开始创建了通道号为0的通道，在这个通道还没销毁的情况下，又去创建通道号为0的通道。

处理步骤

针对分析的可能原因，请参考以下方法处理：

- 首先确定hi_mpi_venc_create_chn接口失败时返回的错误码是多少。
- 如果是0xa0088002，则说明用户传入的通道ID超出了规定的合法范围，需要用户修改通道ID在[0,255]内。
- 如果是0xa0088003或0xa0088008，则说明用户传入的通道属性参数不在规定的合理范围内或设置了暂不支持的参数，

  具体是哪个参数传入有问题可以进一步查看内核日志打印，如下图所示，是传入的分辨率不正确。

  ```
  [Venc]:venc_drv_check_resolution [Line]:342 max picture width (0) err! should in [128,4096]!
  ```

  常见原因：1.入参的结构体没有进行memset初始化，导致有些参数如果没有主动设置就会是一些随机值；2.头文件不匹配，导致枚举类型传入和预期不符合; 3.参数支持范围不了解，各个参数的支持范围可以详细查看DVPP对外接口文档。
- 如果是0xa0088004，则说明用户试图创建已经存在的通道，比如开始创建了通道号为0的通道，在这个通道还没销毁的情况下，又去创建通道号为0的通道

  ```
  [Venc]:venc_create_chn [Line]:2449 device:0 chnl:0 had been created!
  ```

  这种情况建议用户排查代码逻辑：（1）创建通道后是否销毁；（2）是否使用同一个通道ID创建通道。

14.3.5.5 VENC 送入待编码图像帧失败

现象描述

调用VENC发送帧的接口hi_mpi_venc_send_frame返回值非0，发送失败。

可能原因

导致发送帧失败可能原因有以下：

- 用户传入的图像帧参数不在规定的合理范围内或设置了暂不支持的参数。
- 用户送帧的频率太快，大于了性能的规格。

处理步骤

针对分析的可能原因，请参考以下方法处理：

- 如果是0xa0088003或0xa0088008，则说明用户传入的图像帧参数不在规定的合理范围内或设置了暂不支持的参数，

具体是哪个参数传入有问题可以进一步查看内核日志打印，如下图所示，是传入的YUV格式不正确。

```
[Venc]:hevc_check_pixel_format [Line]:1110 H.265 don't support format 5,should be NV12(1) or NV21(2)
```

常见原因：1.入参的结构体没有进行memset初始化，导致有些参数如果没有主动设置就会是一些随机值；2.头文件不匹配，导致枚举类型传入和预期不符合; 3.参数支持范围不了解，各个参数的支持范围可以详细查看DVPP对外接口文档。

- 如果是0xa008800d，则说明VENC的输入空闲队列已满，此时无法再继续往内部送入数据帧，这种问题一般是由于用户送帧的频率太快，大于了芯片处理的速度，导致输入队列堆积，VENC的输入空闲队列长度为6帧，只要堆积到了6帧再继续往里送就会出现这个报错。这种情况建议用户控制调用hi_mpi_venc_send_frame的时间间隔，比如编码帧率30fps，调用的间隔可以控制在33ms一帧。

14.3.5.6 buf_size 参数设置不合理导致视频编码异常

视频编码场景下，需通过hi_venc_attr结构体中buf_size参数值来设置编码缓冲区的内存大小，buf_size参数值设置地不合理，可能会导致视频编码耗时长或编码失败。

现象及可能原因（Ascend RC）

视频编码耗时长或编码失败的场景下，使用proc命令排查问题，proc查询结果中关键信息含义如下：EncStart表示启动编码的帧数，EndSuccessed表示成功编码的帧数，Lost和Disc（Disacrd）表示编码失败的帧数，Recode表示重编的次数。

1. 编码进程运行过程中，登录Device。
2. 执行命令**cat /proc/umap/h265e**或者**cat /proc/umap/h264e**。
   - 出现类似下面红框的现象：Lost和Disc的数量为0或很低，但是Recode的数量比较大，表示大部分帧能够编码成功，但是重编次数太多。

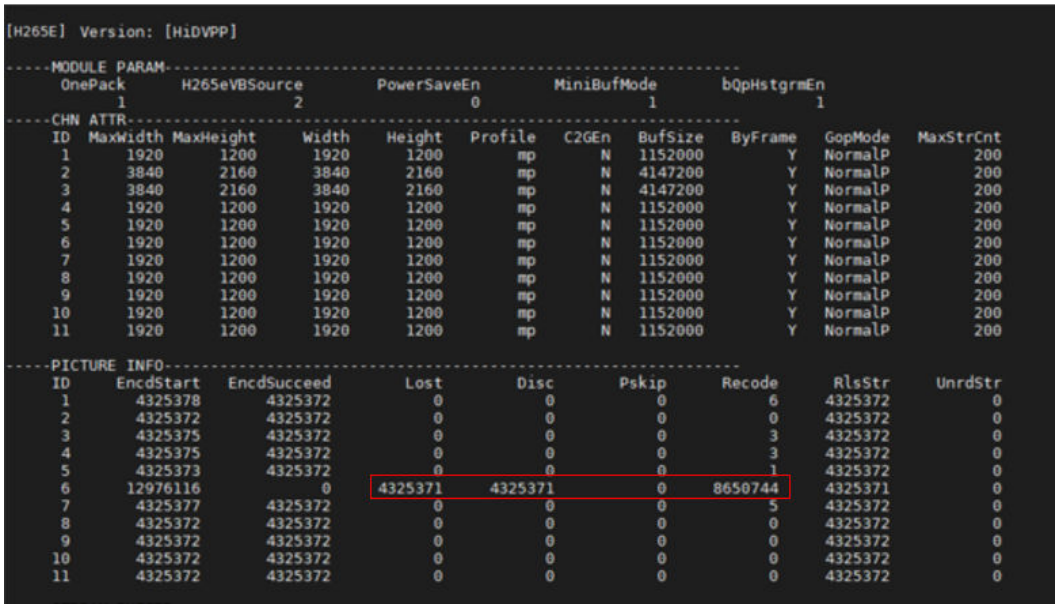
     当实际的编码结果大小大于编码缓冲区中的可用内存大小时，编码模块会自动调整参数重编，减小编码结果数据大小。因此buf_size设置的太小，缓冲帧数少，导致出现重编的概率高，进而导致编码时延增加，帧率变低，性能下降。

     ```
     [H265E] Version: [HiDVPP]

     -----MODULE PARAM----------------------------------------------------------
          OnePack      H265eVBSource       PowerSaveEn       MiniBufMode       bQpHstgrmEn
                1                  2                 0                 1                 1
     -----CHN ATTR-------------------------------------------------------------
        ID  MaxWidth MaxHeight     Width    Height   Profile   C2GEn   BufSize   ByFrame   GopMode   MaxStrCnt
         1      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         2      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
         3      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
         4      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         5      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         6      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         7      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         8      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
         9      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
        10      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
        11      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200

     -----PICTURE INFO---------------------------------------------------------
        ID     EncdStart   EncdSucceed        Lost        Disc       Pskip      Recode      RlsStr     UnrdStr
         1          6597          6594           0           0           0           3        6594           0
         2          6594          6593           0           0           0           0        6593           0
         3          6816          6593           0           0           0         222        6593           0
         4          6597          6594           0           0           0           3        6594           0
         5          6596          6594           0           0           0           2        6594           0
     ```

   - 出现类似下面红框的现象：Lost和Disc的数量比较大，同时Recode的数量也比较大，表示有比较多的帧编码失败了。

     当实际的编码结果大小大于编码缓冲区中的可用内存大小时，编码模块会自动调整参数重编，减小编码结果数据大小。如果重编次数全部用完，但是编

码结果大小依然大于编码缓冲区中的可用内存大小，此时编码模块会将该帧丢弃。因此buf_size设置的太小，缓冲帧数少，导致出现重编的概率高，丢帧概率高。



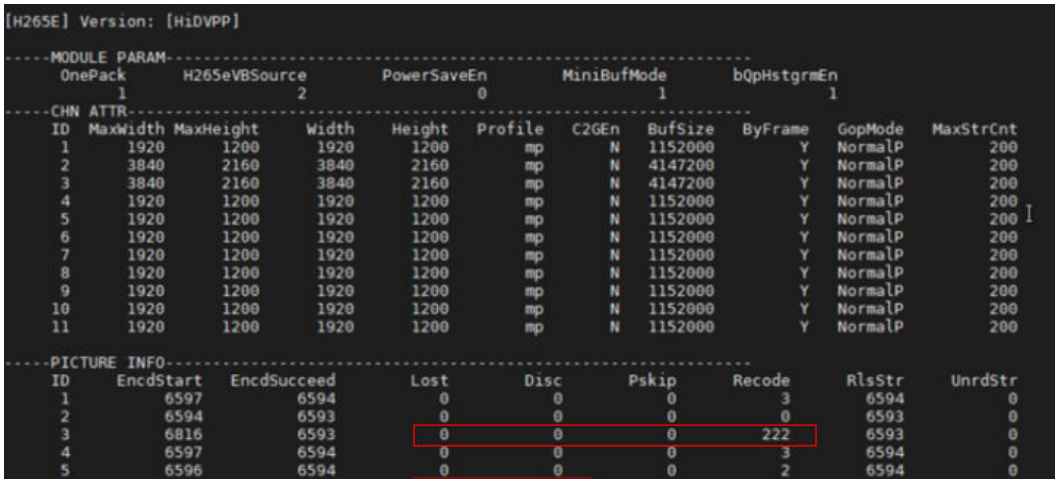
```
[H265E] Version: [HiDVPP]

-----MODULE PARAM----------------------------------------------------------
     OnePack      H265eVBSource        PowerSaveEn        MiniBufMode       bQpHstgrmEn
           1                  2                  0                  1                 1
-----CHN ATTR-------------------------------------------------------------------
    ID  MaxWidth MaxHeight     Width     Height   Profile   C2GEn   BufSize   ByFrame   GopMode   MaxStrCnt
     1      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     2      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     3      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     4      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     5      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     6      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     7      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     8      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     9      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    10      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    11      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200

-----PICTURE INFO---------------------------------------------------------
    ID     EncdStart   EncdSucceed        Lost        Disc       Pskip      Recode      RlsStr     UnrdStr
     1       4325378       4325372           0           0           0           6     4325372           0
     2       4325372       4325372           0           0           0           0     4325372           0
     3       4325375       4325372           0           0           0           3     4325372           0
     4       4325375       4325372           0           0           0           3     4325372           0
     5       4325373       4325372           0           0           0           1     4325372           0
     6      12976116             0     4325371     4325371           0     8650744     4325371           0
     7       4325377       4325372           0           0           0           5     4325372           0
     8       4325372       4325372           0           0           0           0     4325372           0
     9       4325372       4325372           0           0           0           0     4325372           0
    10       4325372       4325372           0           0           0           0     4325372           0
    11       4325372       4325372           0           0           0           0     4325372           0
```

现象及可能原因（Ascend EP）

视频编码耗时长或编码失败的场景下，使用dvpp的proc信息排查问题，proc信息中关键信息含义如下：EncStart表示启动编码的帧数，EndSuccessed表示成功编码的帧数，Lost和Disc（Disacrd）表示编码失败的帧数，Recode表示重编的次数。

1. 运行编码进程后，登录Host，在有读、写、执行权限的目录下执行**msnpureport -a**命令，导出Device的日志信息。
2. 按导出时间，进入对应时间戳的目录下，打开“module_info\dev-os-0\dvpp\dvpp_proc.log”日志：
   - 出现类似下面红框的现象：Lost和Disc的数量为0或很低，但是Recode的数量比较大，表示大部分帧能够编码成功，但是重编次数太多。

     当实际的编码结果大小大于编码缓冲区中的可用内存大小时，编码模块会自动调整参数重编，减小编码结果数据大小。因此buf_size设置的太小，缓冲帧数少，导致出现重编的概率高，进而导致编码时延增加，帧率变低，性能下降。



```
[H265E] Version: [HiDVPP]

-----MODULE PARAM----------------------------------------------------------
     OnePack      H265eVBSource        PowerSaveEn        MiniBufMode       bQpHstgrmEn
           1                  2                  0                  1                 1
-----CHN ATTR-------------------------------------------------------------------
    ID  MaxWidth MaxHeight     Width     Height   Profile   C2GEn   BufSize   ByFrame   GopMode   MaxStrCnt
     1      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     2      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     3      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     4      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     5      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     6      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     7      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     8      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     9      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    10      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    11      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200

-----PICTURE INFO---------------------------------------------------------
    ID     EncdStart   EncdSucceed        Lost        Disc       Pskip      Recode      RlsStr     UnrdStr
     1          6597          6594           0           0           0           3       6594           0
     2          6594          6593           0           0           0           0       6593           0
     3          6816          6593           0           0           0         222       6593           0
     4          6597          6594           0           0           0           3       6594           0
     5          6596          6594           0           0           0           2       6594           0
```


- 出现类似下面红框的现象：Lost和Disc的数量比较大，同时Recode的数量也比较大，表示有比较多的帧编码失败了。

  当实际的编码结果大小大于编码缓冲区中的可用内存大小时，编码模块会自动调整参数重编，减小编码结果数据大小。如果重编次数全部用完，但是编码结果大小依然大于编码缓冲区中的可用内存大小，此时编码模块会将该帧丢弃。因此buf_size设置的太小，缓冲帧数少，导致出现重编的概率高，丢帧概率高。

```
[H265E] Version: [HiDVPP]

-----MODULE PARAM----------------------------------------------------------
     OnePack      H265eVBSource        PowerSaveEn        MiniBufMode       bQpHstgrmEn
           1                  2                  0                  1                 1
-----CHN ATTR-------------------------------------------------------------
    ID  MaxWidth MaxHeight     Width     Height   Profile   C2GEn   BufSize   ByFrame   GopMode   MaxStrCnt
     1      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     2      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     3      3840      2160      3840      2160        mp       N   4147200         Y   NormalP         200
     4      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     5      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     6      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     7      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     8      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
     9      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    10      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200
    11      1920      1200      1920      1200        mp       N   1152000         Y   NormalP         200

-----PICTURE INFO---------------------------------------------------------
    ID     EncdStart   EncdSucceed        Lost        Disc        Pskip       Recode      RlsStr     UnrdStr
     1       4325378       4325372           0           0           0            6     4325372           0
     2       4325372       4325372           0           0           0            0     4325372           0
     3       4325375       4325372           0           0           0            3     4325372           0
     4       4325375       4325372           0           0           0            3     4325372           0
     5       4325373       4325372           0           0           0            1     4325372           0
     6      12976116             0     4325371     4325371           0      8650744     4325371           0
     7       4325377       4325372           0           0           0            5     4325372           0
     8       4325372       4325372           0           0           0            0     4325372           0
     9       4325372       4325372           0           0           0            0     4325372           0
    10       4325372       4325372           0           0           0            0     4325372           0
    11       4325372       4325372           0           0           0            0     4325372           0
```

处理步骤

创建编码通道时，合理设置buf_size参数，具体参见hi_venc_attr结构体中buf_size成员的说明。

14.3.5.7 VENC 编码无输出

现象描述

VENC通道创建，发送帧都成功，但调用VENC获取码流的接口hi_mpi_venc_get_stream返回0xa008800e，获取不到编码后码流数据。

可能原因

可能原因有以下：

- 用户传入的图像帧内存不是使用dvpp内存申请接口hi_mpi_dvpp_malloc申请的。
- 用户传入的图像帧内存和分辨率不匹配。
- OS内存管理出现问题。

处理步骤

针对分析的可能原因，请参考以下方法处理：

- 查看日志是否有出现看门狗相关的打印，如下图所示，如果出现看门狗，基本可以确认是内存使用存在问题


```
[Chnl]:chnl_watch_dog_timer_isr [Line]:1141 find VEDU_0  down,now reset it
[Chnl]:chnl_watch_dog_blackbox [Line]:1097 vpu_id is 0, venc watchdog fail enter blackbx
```

- 排查用户代码中输入图像帧的内存申请方式，如果不是使用dvpp内存申请接口hi_mpi_dvpp_malloc申请的，VENC芯片将无法正常访问该内存，导致编码无输出，需要修改为使用hi_mpi_dvpp_malloc申请输入内存。
- 排查输入图像帧的内存和分辨率是否不匹配，比如NV12或NV21格式的YUV，一帧图像大小为宽x高x1.5，如果实际送给VENC的内存大小比设置的分辨率宽x高x1.5小的话，会产生访问越界等不可预期的行为，也会导致编码无输出。需要保证申请的内存大小和实际设置的分辨率参数匹配。
- 如果以上排查都不存在问题，可能是OS的内存管理出现问题，您可以通过https://gitee.com/ascend网站提交issue获取帮助。

14.3.5.8 播放 VENC 编码的码流，亮暗与原始 YUV 不一致

现象描述

用户使用第三方播放器播放经过VENC编码的码流，发现编码后的码流亮暗与原始YUV不一致。

可能原因

由于播放器的渲染效果不可控，播放器在解码、显示码流内容时，播放器的目标像素值域范围与VENC编码时设置的video_full_range_flag标志位不一致，导致发生像素值域映射，进而出现码流亮暗与原始YUV不一致。

关于full_range的原理介绍请参见**参考信息**。

处理步骤

不建议用播放器验收亮暗效果，因为播放器的渲染效果不可控，建议将VENC码流解码为YUV文件后再与原始YUV对比，同时需确保编码和解码时使用一致的video_full_range_flag标志位。

1. 在VENC编码时，指定video_full_range_flag参数值。

   在Atlas 推理系列产品（Ascend 310P处理器）上，对于H.264、H.265码流，当前VENC编码时video_full_range_flag默认值为0（表示limited_range）。

   在Atlas 200/500 A2推理产品上，对于H.265码流，当前VENC编码时video_full_range_flag默认值为0（表示limited_range）；对于H.264码流，当前VENC编码时video_full_range_flag默认值为1（表示full_range）。

   若默认值不满足要求，用户可以调用hi_mpi_venc_set_h264_vui或hi_mpi_venc_set_h265_vui接口修改video_full_range_flag参数值。
2. 确认VENC编码后码流的video_full_range_flag标志位。

   通过码流分析工具查看SPS字段的video_full_range_flag，flag=1表示原始YUV是full_range的，flag=0表示是limited_range的。

vui parameters()	
aspect ratio info present flag	0
overscan info present flag	0
video signal type present flag	1
video format	5
video full range flag	1
colour description present flag	1
colour primaries	1
transfer characteristics	1
matrix coefficients	1
chroma loc info present flag	0
neutral chroma indication flag	0
field seq flag	0
frame field info present flag	0
default display window flag	0
vui timing info present flag	0
bitstream restriction flag	0

3. 解码时指定video_full_range_flag，然后再将解码后YUV文件后与原始YUV对比亮暗。

   不同解码器的video_full_range_flag使用方式不同，此处仅以FFmpeg为例，可以通过-vf参数指定目标输出video_full_range_flag，默认为输出limited_range。

   此处以ffmpeg 为例，示例指令如下，供参考：

   ```
   ffmpeg -i ${instream} -pix_fmt nv12 -vf scale=out_range=full/limited -y ${outyuv}
   ```

   此处举例说明亮暗对比情况。现在的YUV一般都是full_range的，暂时排除源YUV本身为limited_range的情况，考虑Venc和FFmpeg参数组合4种情况，vui表示VENC编码时设置的video_full_range_flag，ffmpeg表示FFmpeg解码时设置的video_full_range_flag：

   - **vui0_ffmpeg0**和**vui1_ffmpeg1**结果一致，并且对比过源YUV也是一致的，这是因为vui和ffmpeg参数一致，直接解压输出YUV；
   - **vui0_ffmpeg1**图像"更暗"，这是因为ffmpeg认为需要从limited_range转换为full_range，所以对像素值分布进行了往两侧拉伸，而该场景本身偏暗，像素值更多地往0值靠拢，表现为变暗；
   - **vui1_ffmpeg0**图像"更亮"，因为ffmpeg做了标准的值域压缩，从0~255压到16~235，图中大量低像素值往中间区间抬升，图像表现为变亮。

参考信息

电视机一般支持240个色阶，从16~255，也就是**limited_range**，对应YUV值域：Y[16, 235]，UV [16, 240]。

现代电脑显示支持255个色阶，从0~255，也就是**full_range**，对应YUV值域：YUV[0, 255]。

以下标记等价，是在不同软件或者模块中各自的表达方式：

- **“full range” = “jpeg” = “pc” = “cg” = “high rgb”**
- **“limited range” = “mpeg” = “tv” = “broadcast” = “low rgb”**

H.265码流中VUI字段的video_full_range_flag = 0表示源YUV是limited_range，video_full_range_flag = 1表示源YUV是full_range。注意**这个标记位不影响VENC编码过程**，编码生成的码流数据只由输入YUV的实际像素值决定，编码阶段不会发生像素值映射。

例如，ffmpeg解码时会判断源YUV格式和输出YUV格式是否匹配，**仅在两者不匹配时触发像素值重映射**。当原始YUV是full_range的，此时VENC编码设置了video_full_range_flag = 1，若ffmpeg解码输出YUV格式是limited_range的，它发现YUV格式从full_range降为limited_range，于是在解码后对像素值进行了映射，从 [0, 255] 缩小到 [16, 235]，此时就会发现解出来的YUV和原始YUV存在亮暗差异。

14.4 单算子调用问题

14.4.1 执行单算子产生 coredump 的定位处理

现象描述

单算子执行结束，出现重复释放内存，导致coredump，屏幕显示关键日志信息：

```
double free or corruption（！prev）
```

可能原因

分析屏显日志信息，可能存在以下故障原因：代码中出现重复释放内存的操作。

处理步骤

通过gdb挂载可执行文件，通过查看栈信息做排查：

- 重复释放内存代码是否是用户自身代码bug，如果是则需修复代码bug。
- 提供栈信息，通过https://gitee.com/ascend网站提交issue获取帮助。

具体步骤如下：

步骤1 gdb挂载可执行文件。

步骤2 执行gdb调试。

步骤3 查看调用栈。

如果该问题非用户代码问题，需要联系华为算子开发工程师定位排查。 您可以通过https://gitee.com/ascend网站提交issue获取帮助。

----结束

14.4.2 单算子匹配失败

现象描述

单算子执行过程中，出现匹配失败，日志显示如下类似信息。

离线加载执行场景：

```
EH9999 [Match][OpModel]failed to match model, opName = xxx Has not been compiled or loaded, Please make sure the op executed and the op compiled is matched, you can check the op type, op inputs number, outputs number, input format, origin format, datatype, memtype, attr, dim range, and so on.
```

在线加载执行场景：

```
EH9999 [Match][OpModel]MatchOpModel fail from static map or dynamic map. Please make sure the op executed and the op compiled is matched, you can check the op type, op inputs number, outputs number, input format, origin format, datatype, memtype, attr, dim range, and so on.
```

可能原因

- 离线场景：算子om文件未出现在aclSetModelDir中指定的路径下。
- 在线场景：一些特殊的匹配规则未适配，导致算子匹配失败。

处理步骤

- 离线场景：重新编译缺少的算子，并复制到aclSetModelDir中指定的路径下，余下步骤按离线单算子推理步骤进行。
- 在线场景：根据报错信息检查前后两次编译的算子的opType、dataType、format等信息是否一致。

14.4.3 opp 安装版本问题导致加载单算子失败

现象描述

加载单算子报错失败，日志显示如下类似信息：

```
E19999: Inner Error
E19999 The opp version of the model does not match the current opp run package, Model is [6.4.T11.0.B300], opp run package is [7.0.T3.0.B107], try to convert the om again!
```

可能原因

动态Shape算子场景下或者昇腾虚拟化实例场景下，单算子模型数据加载环境中的算子库包安装版本（包名为CANN-opp-*-linux.*.run，命名中的*为版本号或架构类型）与om模型文件编译环境的**版本不一致**，导致加载算子时会报错。

处理步骤

动态Shape算子场景下或者昇腾虚拟化实例场景下，单算子模型数据加载环境中的算子库包安装版本（包名为CANN-opp-*-linux.*.run，命名中的*为版本号或架构类型）需与om模型文件编译环境的版本**需保持一致**，出现该报错后，需排查安装版本，选择更换算子加载环境的opp包版本或更换编译算子om文件环境的opp包版本，若选择更换后者，则需要重新转换模型。

14.5 Camera 处理图片问题

14.5.1 Camera 找不到设备

适用场景

- 业务场景：Camera出图
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

显示open ISP device error! 返回错误码0xa01c8040。

可能原因

没有插入isp ko。

处理步骤

步骤1 lsmod检查ko的插入情况。

步骤2 执行以下命令重新插入ao相关的ko。

```
insmod /var/davinci/driver/drv_isp.ko
```

步骤3 再次执行Camera业务。

----结束

14.5.2 Camera 不出图

适用场景

- 业务场景：Camera出图
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

取Sensor数据失败，创建ISP失败，日志中有类似的打印信息：

```
[imx477_sensor_ctl.c:121][sensor]viPipe:0, addr:100, data:0
 [imx477_sensor_ctl.c:149][sensor]I2C WRITER DATA error!
 [imx477_sensor_ctl.c:209][sensor]imx477 vipipe:0, i2caddr:256,i2cval
```

可能原因

sensor没有安装好，或者sensor型号与预期设置不匹配。

处理步骤

步骤1 插拔sensor，或者检查相应配置。

步骤2 如果步骤**步骤1**无法解决，替换新sensor设备。

步骤3 再次启动Camera。

----结束

14.5.3 Camera 一直丢帧，无图片 dump 出来

适用场景

- 业务场景：Camera出图，同时dump raw和yuv图片
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

反复刷屏打印：pipe *x* chn *x* get buffer fail,hi_size *xxxx*!

可能原因

申请的pipe depth小于或者等于dump raw depth，导致无多余buffer送给后端chn。

处理步骤

步骤1 查看相关depth数，修改相关属性，确保pipe depth + chn attr depth - dump_attr depth大于零即可。

步骤2 再次启动Camera。

----结束

14.5.4 Camera 出图效果不对

适用场景

- 业务场景：Camera出图，dump图片
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

dump图片后发现图片为花图。

可能原因

检查pixel format和看图工具的pixel format是否一致，分辨率大小是否一致。

处理步骤

步骤1 检查pixel format和看图工具的pixel format，例如：YUV420，则选择对应420格式和分辨率。

步骤2 设置完成后，再次启动Camera。

----结束

14.6 Audio 处理声音问题

14.6.1 无法正常播音

适用场景

- 业务场景：播音
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在执行播音业务时，不能正常播音，并产生报错，返回错误码0xa0168009。


```
sample_comm_audio_start_ao: hi_mpi_ao_set_pub_attr(2) failed with 0xa0168009!
```

可能原因

没有插入播音相关的ko。

处理步骤

步骤1 lsmod检查ko的插入情况。

步骤2 执行以下命令重新插入ao相关的ko。

```
insmod /var/davinci/driver/drv_ao.ko
```

步骤3 再次执行播音命令。

----结束

14.6.2 播音属性设置无效

适用场景

- 业务场景：播音
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在初始化播音相关参数时失败，并产生报错，返回错误码0xa0168003或者0xa016800c。

```
sample_comm_audio_start_ao: hi_mpi_ao_set_pub_attr(2) failed with 0xa0168003!
```

可能原因

播音涉及到的参数设置有误。

处理步骤

步骤1 检查Audio接口中各个输入参数（例如位宽、采样率、声道数等）是否在范围之内，若这些参数值不在取值范围内，需修改至有效范围内。

步骤2 再次执行播音命令。

----结束

14.6.3 播音音调、语速不对

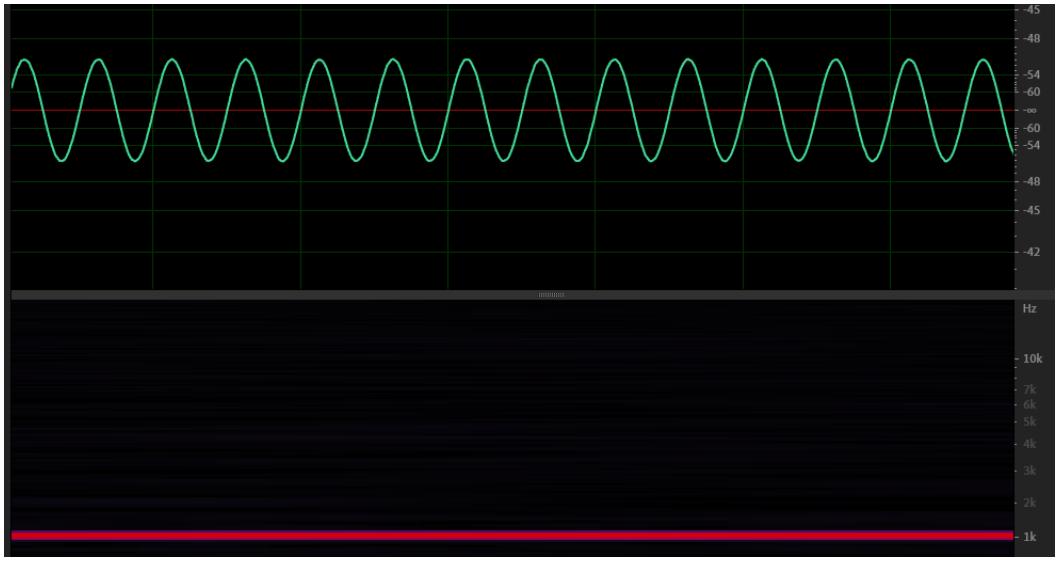
适用场景

- 业务场景：播音
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

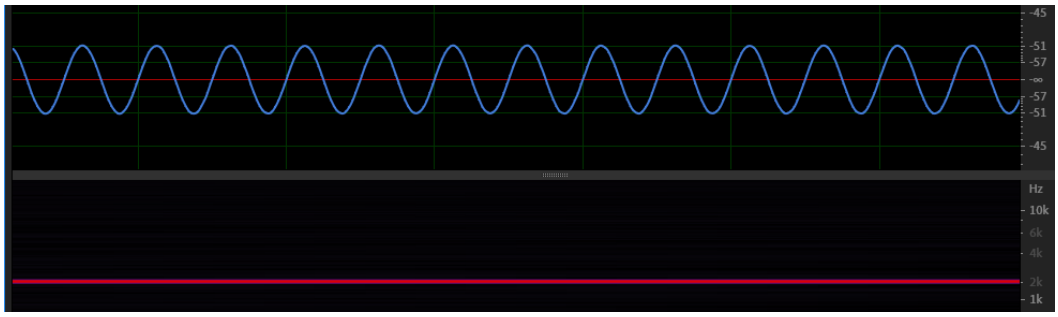
现象描述

执行播音命令之后，pcm音频文件被成功读取，音乐播放有声音，但速度变慢或者变快，对应的音调变低或变高。

例如：原始pcm为单音正弦波，频率为1kHz。



经过播放之后频率变为2kHz，播放速度变快，音调变高。



可能原因

播音的参数设置与pcm音频文件的格式不一致。由于pcm文件是raw数据，不带文件头，所以无法单从数据中获取声道、位宽、采样率等信息。例如pcm文件是单声道文件，但是以双声道的方式读取文件，那么播放速度就会是正常的两倍，对应的音调也会变高。

处理步骤

步骤1 检查Audio接口中的声道、位宽等参数是否与pcm文件的参数一致。

步骤2 修改诸位宽、采样率、声道数并设置属性。

步骤3 再次执行播音命令。

----结束

14.6.4 音量调节失效

适用场景

- 业务场景：播音/录音时调节音量
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在执行播音或者录音的音量调节时，发现音量并无变化，音量调节失败。

可能原因

1. 音量调节超出范围，目前支持对播音、录音的左右声道分别进行音量调节，控制音量的参数范围为0~127，如果超过范围则会报错，音量调节失败。
   ```
   HI_ACODEC_SET_ADCL_VOLUME failed!
   ```
2. 播音时误用录音的音量调节接口，或者在录音时误用播音的音量调节接口。

处理步骤

步骤1 检查Audio接口中的音量调节范围是否配置正确。

步骤2 确认播音时调用的是DAC相关的接口调节音量，录音时调用ADC相关的接口调节音量。

步骤3 再次执行播音/录音命令。

----结束

14.7 HDMI 显示数据问题

14.7.1 HDMI OPEN 失败

适用场景

- 业务场景：通过HDMI接口送显
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在执行HDMI接口送显业务时，在日志中产生如下HDMI OPEN失败的用户态错误：

```
[ERROR] DSS(4808,vo_test_nvr_hdmi_hi_test):1970-01-01-08:31:51.173.346 [mpi_hdmi_com.c:338][lib_hdmi][mpi_hdmi_com_open]:HDMI device not init
```

可能原因

调用hi_mpi_hdmi_open接口前没有调用hi_mpi_hdmi_init接口。

处理步骤

步骤1 检查用例中hi_mpi_hdmi_open接口、hi_mpi_hdmi_init接口的调用顺序，修正用例。

步骤2 再次执行hdmi接口送显用例。

----结束

14.7.2 HDMI 与 VO 模块多进程时报错

适用场景

- 业务场景：通过vo模块及HDMI接口送显
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在执行通过VO模块及HDMI接口送显业务时，在日志中产生如下的不支持多进程错误。

```
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.632.632 [klogd.c:246][3146.786343] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.632.666 [klogd.c:246][3146.786375] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.647.470 [klogd.c:246][3146.793796] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.662.320 [klogd.c:246][3146.801217] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.662.343 [klogd.c:246][3146.808650] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.662.353 [klogd.c:246][3146.816065] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] DSS(5303,vo_test_nvr_hdmi_hi_test):1970-01-01-08:52:26.669.726 [mpi_hdmi_com.c:264][lib_hdmi][mpi_hdmi_com_init]:open HDMI err.
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:26.678.229 [klogd.c:246][3146.823479] [drv_hdmi][ERR][hdmi_file_open:2750]:HDMI functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:38.192.079 [klogd.c:246][3158.345748] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:38.192.199 [klogd.c:246][3158.345800] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
[ERROR] KERNEL(3720,sklogd):1970-01-01-08:52:38.207.148 [klogd.c:246][3158.353257] [drv_vo][ERR][vo_open:433]:VO functions must work in the same pid!!!
```

可能原因

有其他调用了VO、HDMI接口的用例在后台执行。

处理步骤

步骤1 检查是否通过mobax等软件开启了多个端口并正在执行调用了VO、HDMI接口的用例。

步骤2 检查后台是否有其他调用了VO、HDMI接口的用例，命令: ps -elf。

步骤3 关闭其他调用了VO、HDMI接口的用例。

步骤4 再次执行通过VO模块及HDMI接口送显业务用例。

----结束

14.7.3 HDMI 接口无显示

适用场景

- 业务场景：通过HDMI接口送显
- 适用处理器：Atlas 200/500 A2推理产品
- 处理器形态：EP、RC

现象描述

在执行通过VO模块及HDMI接口送显业务时，HDMI接口不显示，且无报错日志。

可能原因

可能存在以下原因：

- 用例最后没有调用hi_mpi_hdmi_start接口来开启具体的hdmi接口，或者开启错误。
- 用例中设置hdmi属性、信息帧相关参数错误。
- 显示器不支持当前分辨率。

处理步骤

步骤1 检查是否调用了hi_mpi_hdmi_start接口来开启具体的hdmi接口，或者开启错误。

步骤2 排查用例参数，信息帧（avi infoframe）里面的时序信息（video_format），属性hi_hdmi_attr里面的时序参数（timing_mode）、像素时钟参数（pix_clk）是否与前级（vo模块）输出不符合。

步骤3 读取当前显示器的edid信息，确认显示器是否支持当前分辨率，如果不支持，需要更换支持该分辨率的显示器。

步骤4 再次执行通过HDMI接口送显业务用例。

----结束

14.8 编译运行问题

14.8.1 编译运行应用样例报错，提示找不到头文件或库文件

问题现象

获取**13 应用样例参考**章节中应用源码：

- 编译源码时，提示找不到头文件acl.h，报错片段举例如下：

```
fatal error: acl/acl.h: No such file or directory
#include "acl/acl.h"
         ^~~~~~~~~~
compilation terminated.
CMakeFiles/main.dir/build.make:62: recipe for target 'CMakeFiles/main.dir/main.cpp.o' failed
make[2]: *** [CMakeFiles/main.dir/main.cpp.o] Error 1
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/main.dir/all' failed
make[1]: *** [CMakeFiles/main.dir/all] Error 2
Makefile:129: recipe for target 'all' failed
```

- 编译源码时，提示找不到库文件libascendcl.so（报错中的-lascendcl，-l表示查找库文件，ascendcl前后分别加上lib和.so组成库文件的名称libascendcl.so），报错片段举例如下：

```
/usr/bin/ld: cannot find -lascendcl
collect2: error: ld returned 1 exit status
CMakeFiles/main.dir/build.make:94: recipe for target '/home/HwHiAiUser/sample/
resnet50_firstapp/out/main' failed
make[2]: *** [/home/HwHiAiUser/sample/resnet50_firstapp/out/main] Error 1
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/main.dir/all' failed
make[1]: *** [CMakeFiles/main.dir/all] Error 2
Makefile:129: recipe for target 'all' failed
```


原因分析

编译应用源码依赖定义AscendCL接口的头文件和库文件，**13 应用样例参考**章节中的样例是通过分别配置{DDK_PATH}、{NPU_HOST_LIB}环境变量来查找头文件、库文件，当前报错提示找不到头文件、库文件，则可能是{DDK_PATH}、{NPU_HOST_LIB}环境变量配置地不正确。

解决方法

步骤1 登录编译源码的环境，分别使用如下命令查看{DDK_PATH}、{NPU_HOST_LIB}环境变量的值。

- 查看{DDK_PATH}环境变量的值：
  ```
  echo $DDK_PATH
  ```
  回显信息示例如下（若无回显信息，则环境变量未配置，跳转到**步骤4**配置该环境变量）：
  ```
  /home/HwHiAiUser/Ascend/ascend-toolkit/latest
  ```
- 查看{NPU_HOST_LIB}环境变量的值：
  ```
  echo $NPU_HOST_LIB
  ```
  回显信息示例如下（若无回显信息，则环境变量未配置，跳转到**步骤4**配置该环境变量）：
  ```
  /home/HwHiAiUser/Ascend/ascend-toolkit/latest/runtime/lib64/stub
  ```

步骤2 根据**步骤1**中获取到的{DDK_PATH}环境变量值，检查对应路径下是否存在头文件。

样例中的编译脚本会根据“{DDK_PATH}环境变量值/runtime/include/acl”目录查找编译依赖的头文件，因此可先检查“{DDK_PATH}环境变量值/runtime/include/acl”路径是否存在，若存在，则检查该路径下的acl.h头文件是否存在；若路径或头文件有一个不存在，则需要重新配置{DDK_PATH}环境变量，跳转到**步骤4**。

- 检查路径是否存在，命令示例如下：
  ```
  cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/runtime/include/acl
  ```
- 检查acl.h是否存在，命令示例如下：
  ```
  ll acl.h
  ```

步骤3 根据**步骤1**中获取到的{NPU_HOST_LIB}环境变量值，检查对应路径下是否存在库文件。

样例中的编译脚本会根据{NPU_HOST_LIB}环境变量指向的路径查找编译依赖的库文件，因此可先检查{NPU_HOST_LIB}环境变量指向的路径是否存在，若存在，则检查该路径下的libascendcl.so库文件是否存在；若路径或库文件有一个不存在，则需要重新配置{NPU_HOST_LIB}环境变量，跳转到**步骤4**。

- 检查路径是否存在，命令示例如下：
  ```
  cd /home/HwHiAiUser/Ascend/ascend-toolkit/latest/runtime/lib64/stub
  ```
- 检查libascendcl.so是否存在，命令示例如下：
  ```
  ll libascendcl.so
  ```

步骤4 配置环境变量。

编译脚本会根据“{DDK_PATH}环境变量值/runtime/include/acl”目录查找编译依赖的头文件，根据{NPU_HOST_LIB}环境变量指向的目录查找编译依赖的库文件。“$HOME/Ascend”请替换“Ascend-cann-toolkit”包的实际安装路径。

- 当开发环境与运行环境的操作系统架构相同时，配置示例如下所示：
  ```
  export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest
  export NPU_HOST_LIB=$DDK_PATH/runtime/lib64/stub
  ```


- 当开发环境与运行环境的操作系统架构不同时，配置示例如下所示：

  例如，当开发环境为X86架构、运行环境为AArch64架构时，则涉及交叉编译，需在开发环境上安装AArch64架构的软件包，将{DDK_PATH}环境变量的路径指向AArch64架构的软件包安装目录（如下所示），便于使用与运行环境架构相同的软件包中的头文件和库文件来编译代码。

  ```
  export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest/arm64-linux
  export NPU_HOST_LIB=$DDK_PATH/runtime/lib64/stub
  ```

  说明

  - 您可以登录对应的环境，执行**uname -a**命令查询其操作系统的架构。
  - 如果不清楚“Ascend-cann-toolkit”包的安装路径，也可以使用**find -name "*filename*"**命令，查找acl.h、libascendcl.so文件所在的路径，再配置环境变量。

    当环境上安装多个软件版本时，请根据实际情况选择版本，其中，latest目录默认指向最后安装的软件版本。

    当同一个版本下有多个acl.h、libascendcl.so文件时，是由于兼容旧版本的原因，其中部分文件是软链接，方便用户在旧版本下编译应用源码。

----结束

14.8.2 执行应用程序的权限不足导致 AscendCL 初始化报错

问题现象

用户进程报错并退出。

查看应用类日志，提示获取Device信息失败，最终导致AscendCL初始化失败，日志片段示例如下：

```
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.635 [runtime.cc:1065]89696 CheckHaveDevice:
[INIT][DEFAULT]Call halGetDeviceInfo failed: drvRet=4, module type=0, info type=1.
[ERROR] ASCENDCL(89696,main):2023-03-07-17:13:27.994.723 [acl.cpp:164]89696 aclInit: [INIT][DEFAULT]
[Init][Version]init soc version failed, ret = 507008
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.774 [api_impl.cc:3490]89696
GetDevErrMsg:report error module_type=3, module_name=EE8888
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.798 [api_impl.cc:3490]89696 GetDevErrMsg:ctx is
NULL!
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.827 [api_impl.cc:3546]89696 GetDevMsg:Failed
to GetDeviceErrMsg, retCode=0x7070001.
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.849 [logger.cc:1348]89696
GetDevMsg:GetDeviceMsg failed, getMsgType=0.
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.888 [api_c.cc:3595]89696
rtGetDevMsg:ErrCode=107002, desc=[context pointer null], InnerCode=0x7070001
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.910 [error_message_manage.cc:49]89696
FuncErrorReason:report error module_type=3, module_name=EE8888
[ERROR] RUNTIME(89696,main):2023-03-07-17:13:27.994.932 [error_message_manage.cc:49]89696
FuncErrorReason:rtGetDevMsg execute failed, reason=[context pointer null]
EL0003: The argument is invalid.
        Solution: Try again with a valid argument.
        TraceBack (most recent call last):
        [Init][Version]init soc version failed, ret = 507008[FUNC:ReportInnerError][FILE:log_inner.cpp]
[LINE:145]
        ctx is NULL![FUNC:GetDevErrMsg][FILE:api_impl.cc][LINE:3490]
        rtGetDevMsg execute failed, reason=[context pointer null][FUNC:FuncErrorReason]
[FILE:error_message_manage.cc][LINE:49]

[ERROR] acl init failed
[ERROR] Sample init resource failed
```

原因分析

可能存在以下原因：

- Device状态异常，未正常启动。
- 执行应用程序的用户权限不足，无法查询Device信息。

解决办法

1. 首先，确认Device是否正常启动。
   a. 以root用户登录安装Driver包的环境，执行以下命令查询其安装路径。
   ```
   cat /etc/ascend_install.info
   ```
   在该文件中，Driver_Install_Path_Param表示Driver包的安装路径。
   b. 进入Driver安装路径，使用upgrade-tool工具查看下Device侧运行文件系统版本，如果能正常查询，则说明Device侧已经正常启动。
   ```
   ./upgrade-tool --device_index -1 --system_version
   ```
   正常查询返回信息类似如下：
   ```
   [root@localhost tools]# /usr/local/Ascend/driver/tools/upgrade-tool --device_index -1 --system_version
   {
   Get system version [redacted] succeed, deviceId(0)
           {"device_id":0, "version"[redacted]
   Get system version [redacted] succeed, deviceId(1)
           {"device_id":1, "version":[redacted]
   }
   ```
2. 其次，检查运行应用程序的用户权限是否正确。
   要求运行应用程序的用户，需与Driver运行用户在一个属组内。在“cat /etc/passwd”文件中，可查看用户属组，Driver的默认运行用户为HwHiAiUser。
   修改用户属组的命令示例如下：
   ```
   usermod -g 组名 用户名
   ```
3. 如果以上方法解决不了问题，则需要参考如下步骤将获取日志，并在**modelzoo仓**通过提Issue反馈给华为工程师。
   a. 登录到运行应用程序的环境，执行如下命令将日志级别设置为Debug。
   ```
   export ASCEND_GLOBAL_LOG_LEVEL=0
   ```
   b. 重新运行应用程序。
   c. 从日志存放路径下获取应用类日志。
   存放日志的默认路径为“$HOME/ascend/log”。
   d. 使用msnpureport工具，获取指定Device上的Debug日志。
   命令示例如下，其中deviceID需要设置为指定Device的ID：
   ```
   msnpureport -g debug -d deviceID
   ```

14.8.3 AscendCL 接口执行无输出无报错

现象描述

AscendCL执行过程中，无接口报错，但是没得到预期结果。

可能原因

- AscendCL执行过程中链接到了stub中的so。
- 异步场景下，在拷贝输出数据时没有做流同步。

处理步骤

针对第一种情况：使用ldd命令查看执行文件链接的so是否正确，保证链接了有效的so。

针对第二种情况：请参考《**AscendCL应用软件开发指南（C&C++）**》手册中“AscendCL API参考”章节的“aclmdlExecuteAsync”接口介绍，排查代码。

14.8.4 APP 使用 dvpp 接口编译失败

问题现象

编译提示DVPP的相关接口未定义，编译报错，日志关键字包括：undefined reference to ***

原因分析

分析上述日志信息，可能存在以下故障原因：

DVPP与AscendCL已经分别打包到libacl_dvpp.so与libascendcl.so，测试用例使用了DVPP的相关接口，但没有链接libacl_dvpp.so。

解决方法

针对分析的可能原因，可以参考下面步骤处理：

排查测试用例是否使用了预处理的接口，但未链接libacl_dvpp.so。如果未链接，需要在编译文件中链接libacl_dvpp.so。

需要排查CmakeLists中的target_link_libraries()选项是否连接了acl_dvpp这个target。

示例：

```
add_executable(main
utils.cpp
main.cpp)
target_link_libraries(main
ascendcl acl_dvpp stdc++)
```

14.8.5 环境变量访问冲突，导致应用程序异常终止

问题现象

多卡训练场景，出现多次core dump，应用程序异常终止。

原因分析

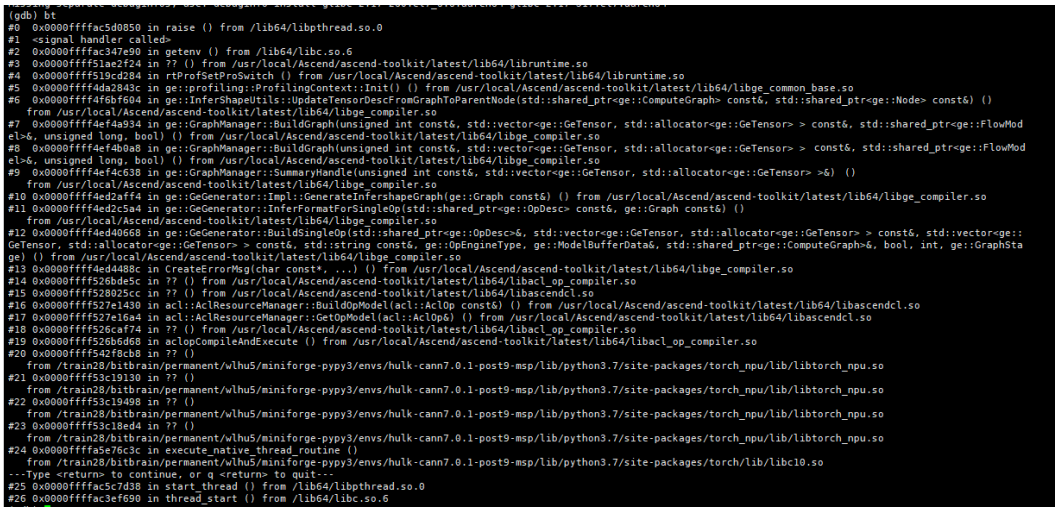
1. 生成coredump文件。
   - 物理机场景，执行**ulimit -c unlimited**命令，表示在程序崩溃时生成coredump文件：

     完成问题定位后，如果不需要生成coredump文件，可执行**ulimit -c 0**命令。
   - Docker场景，在Docker启动命令中增加**--ulimit core=-1**设置。
2. 运行训练脚本，若进程崩溃，即可在当前目录下生成coredump文件。
3. 使用gdb工具调试core文件、打印堆栈信息。

   进入gdb模式，调试coredump文件，命令示例如下。其中，python3表示产生coredump文件的可执行程序名称，可根据实际情况修改；coredump文件名需根据实际文件名称修改。

   ```
   gdb python3 core*.*
   ```


执行命令后，gdb工具会将发生异常的代码、其所在的函数、文件名和所在文件的行数打印到屏幕，堆栈信息的最上面是最底层的调用栈信息，方便定位问题。堆栈信息举例如下：



注意，调试coredump文件、打印堆栈信息要在出现问题的运行环境中，如果换一套环境，可能导致调试的堆栈信息不准确。

若环境中未安装gdb，则需要安装gdb，可通过包管理（如apt-get install gdb、yum install gdb）进行安装，详细安装步骤及使用方法请参见**GDB官方文档**。

4. 分析堆栈信息。

   生成coredump文件、检查打印的堆栈信息后，发现应用程序集中在getenv()函数时异常退出，因此初步判断可能是getenv()函数使用问题，该函数用于读取环境变量。

   在程序中，对于环境变量的操作，如果同时存在读、写操作，例如getenv、putenv，则可能导致环境变量访问冲突，进而导致程序异常。

5. 排查训练脚本中是否存在写环境变量的操作，导致与读环境变量的操作getenv冲突。

   环境变量支持通过命令、接口、配置等方式实现，包括export命令、putenv/getenv/setenv/unsetenv/clearenv函数、os.environ、os.getenv等。您可以在训练脚本中排查这些方式，若存在，则可能引起环境变量访问冲突，导致程序异常。

   本例中，训练脚本存在如下设置环境变量的代码，该方式实际调用的是C语言的putenv函数，putenv函数与算子编译时的getenv函数存在环境变量访问冲突。

   ```
   os.environ["xxxxxxxxx"] = "xxxxxxxxx"
   ```

解决方法

修改用户程序代码逻辑，删除代码中动态设置环境变量的逻辑，可在执行程序前设置环境变量。

15 附录

15.1 使用约束

表 15-1 总体约束列表

分类	约束项
关于低功耗	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。

分类	约束项
关于进程	● 不支持使用fork函数以及封装了fork的函数（如system、posix_spawnp等）创建多个子进程，且在进程中调用AscendCL接口的场景，否则进程运行时会报错或者卡死 。 ● 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。
关于创建类和销毁类接口	● 对于创建类接口（例如：aclrtCreateStream、aclrtCreateEvent、aclCreateDataBuffer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：aclrtDestroyStream、aclrtDestroyEvent、aclDestroyDataBuffer等），否则，程序可能会异常。 ● 对于销毁类接口（例如：aclrtDestroyStream、aclrtDestroyEvent、aclrtFree、aclDestroyDataBuffer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，置为NULL）。

分类	约束项
关于内存	• 不支持在aclrtMemcpyAsync、aclrtMemsetAsync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。 • 使用AscendCL提供的内存申请接口（例如aclrtMalloc）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aclrtMemset或aclrtMemsetAsync接口先清空内存，例如aclrtMemset(devBufferPtr, devBufferSize, 0, devBufferSize)。 • 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`
旧版本昇腾AI处理器->新版本昇腾AI处理器的应用迁移	需在迁移后的昇腾AI处理版本上重新转换模型、编译应用程序，否则可能存在应用执行异常的情况。

15.2 表达约定

接口命名规则

接口命名同时满足如下规则：

1. 规则1：acl+*接口类别缩写+操作动词+对象*
2. 规则2：操作动词和对象均采用首字母大写

媒体数据处理V2版本下的接口命名规则例外，这一类接口命名以“hi_mpi”开头。

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	dvpp或vdec或venc	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类接口

接口类别	缩写	描述
model	mdl	表示模型推理类的接口
graph	grph	表示graph类的接口
driver	drv	表示驱动类的接口
OP	op	表示算子执行类的接口
fv	fv	表示特征向量检索接口
Profiling	prof	表示Profiling配置类接口
tdt	tdt	表示Tensor数据传输接口

注：

1. 缩写原则上不超过4个字母。

2. 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：aclmdlLoadFromFileWithMem，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。

15.3 昇腾 AI 处理器各工作模式的差异

昇腾AI 处理器有EP、RC两种工作模式。在进行应用开发时，不同的工作模式有以下方面的差异。

环境准备

表 15-2 环境准备

工作模式	参考手册
EP	开发环境与运行环境的准备： 请参见《 CANN软件安装指南 》。
RC	● Atlas 200 AI加速模块（型号 3000） 开发环境的准备请参见《 CANN软件安装指南 》。 运行环境的准备请参见《 Atlas 200 AI加速模块 软件安装与维护指南（RC场景）》

应用编译

应用编译时，在EP模式与RC模式下有两点差异：环境变量的配置与编译命令。

- 环境变量配置：

  表 15-3 环境变量配置

工作模式	环境变量
EP	● 如果开发环境与运行环境CPU架构相同，以开发环境为x86架构为例： export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest/x86_64-linux export NPU_HOST_LIB=$HOME/Ascend/ascend-toolkit/latest/x86_64-linux/compiler/lib64/stub ● 如果开发环境与运行环境CPU架构不同，以开发环境为x86架构、运行环境为ARM架构为例： export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest/arm64-linux export NPU_HOST_LIB=$HOME/Ascend/ascend-toolkit/latest/arm64-linux/runtime/lib64/stub
RC	由于RC模式下，开发环境的操作系统为Ubuntu X86而运行环境的操作系统为Ubuntu ARM， 开发环境与运行环境的CPU架构不相同，所以只有一种环境变量配置方式。 export DDK_PATH=$HOME/Ascend/ascend-toolkit/latest/arm64-linux export NPU_HOST_LIB=$HOME/Ascend/ascend-toolkit/latest/arm64-linux/runtime/lib64/stub

  说明

  表格中配置的环境变量仅为示例，请将$HOME/Ascend/ascend-toolkit/latest按场景替换：

  - EP场景：替换为标准形态compiler或runtime软件包的实际安装路径。
  - RC场景：替换为minirc形态runtime软件包的实际安装路径。

- 编译命令：

  表 15-4 编译命令

工作模式	环境变量
EP	● 如果开发环境与运行环境CPU架构相同： cmake ../../../src -DCMAKE_CXX_COMPILER=g++ -DCMAKE_SKIP_RPATH=TRUE ● 如果开发环境与运行环境CPU架构不同： cmake ../../../src -DCMAKE_CXX_COMPILER=aarch64-linux-gnu-g++ -DCMAKE_SKIP_RPATH=TRUE
RC	由于RC模式下，开发环境的操作系统为Ubuntu X86而运行环境的操作系统为Ubuntu ARM， 开发环境与运行环境的CPU架构不相同，所以只有一种编译命令。 cmake ../../../src -DCMAKE_CXX_COMPILER=aarch64-linux-gnu-g++ -DCMAKE_SKIP_RPATH=TRUE

15.4 Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P 处理器）的媒体数据处理接口迁移指引

15.4.1 功能支持度对比列表

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，例如：

- 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。

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。

模块	功能或约束	Atlas 200/300/500 推理产品媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V2接口（接口名以hi_mpi开头）
JPEGE	图片编码成jpeg	支持	支持	支持
	huffman表可配置	不支持	不支持	支持
PNGD	PNG解码	支持	支持	支持
JPEGD	jpeg图片解码成YUV	支持	支持	支持
VENC	h264编码	支持	支持	支持
	h265编码	支持	支持	支持
	码率可调	支持	支持	支持
	帧率可调	支持	支持	支持
	定码率（CBR）	支持	支持	支持
	变码率（VBR）	支持	支持	支持
	强制I帧	支持	支持	支持
	长时间稳定的可变比特率（CVBR）	不支持	不支持	支持
	自适应可变比特率（AVBR）	不支持	不支持	支持

模块	功能或约束	Atlas 200/300/500 推理产品媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V2接口（接口名以hi_mpi开头）
	基于主观图像质量的可变比特率（QVBR）	不支持	不支持	支持
	I帧间隔（GOP）	支持	支持	支持
	场景模式配置	不支持	不支持	支持
VDEC	H264解码	支持	支持	支持
	H265解码	支持	支持	支持
	隔行扫描	支持	支持	支持
	实时出帧	支持	支持	支持
	抽帧	支持	支持	支持
	解码缩放	支持	支持	支持
	输出RGB888	不支持	支持	支持
VPC	抠图	支持	支持	支持
	缩放	支持	支持	支持
	填充功能	不支持	支持	支持
	抠图缩放	支持	支持	支持
	抠图缩放贴图	支持	支持	支持
	抠图缩放填充	不支持	支持	支持
	一图多框贴图	支持	支持	支持
	一图多框填充	不支持	支持	支持
	多图多框贴图	支持	支持	支持
	多图多框填充	不支持	支持	支持
	色域转换（CSC）	支持	支持	支持
	色域转换（CSC）系数配置	不支持	支持	支持
	金字塔	不支持	支持	支持

模块	功能或约束	Atlas 200/300/500 推理产品媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V1接口（接口名以acldvpp开头）	Atlas 推理系列产品（Ascend 310P处理器）媒体数据处理V2接口（接口名以hi_mpi开头）
	直方图均衡（LUT）	不支持	支持	支持
	直方图统计	不支持	支持	支持

15.4.2 Atlas 200/300/500 推理产品媒体数据处理 V1->Atlas 推理系列产品（Ascend 310P 处理器）媒体数据处理 V1 迁移指引

15.4.2.1 使用场景说明

媒体数据处理V1版本接口（接口名以acldvpp开头）从Atlas 200/300/500 推理产品迁移到Atlas 推理系列产品（Ascend 310P处理器）上，由于是同一套接口在不同昇腾处理器上执行，接口的调用逻辑、调用示例基本一致，头文件、库文件不涉及修改，但迁移时存在一些兼容性的功能，本节仅介绍**涉及兼容性的功能列表**，更多功能及接口说明请参见如下章节。

- 接口调用流程及示例在“图像/视频数据处理>**媒体数据处理V1**”章节。
- 接口说明在“AscendCL API参考>媒体数据处理V1”章节。

15.4.2.2 兼容性说明

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置VENC输出buffer。	aclvencSetChannelDescBufAddr接口 aclvencSetChannelDescParam接口	用户不需要设置输出buffer，回调时进行一次内存拷贝，拷贝至用户输出buffer。	需要用户设置输出buffer，编码输出结果**无需多一次拷贝操作**。	需要调用aclvencSetChannelDescBufAddr接口或aclvencSetChannelDescParam接口显式设置输出buffer。

15.4.2.3 优化建议

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
VPC功能中，设置缩放算法。	acldvppSetResizeConfigInterpolation接口	支持5种算法： ● 0：默认值。表示华为自研的高阶滤波算法。 ● 1：业界通用的Bilinear算法（与OpenCV算法的计算过程类似） ● 2：业界通用的Nearest neighbor算法（与OpenCV算法的计算过程类似） ● 3：业界通用的Bilinear算法（与Tensorflow算法的计算精过程类似）。 ● 4：业界通用的Nearest neighbor算法（与Tensorflow算法的计算精过程类似）。	支持2种算法： ● 0：默认值。设置为0时，系统内部也会自动采用1。 ● 1：业界通用的Bilinear算法（与OpenCV算法的计算过程类似） ● 2：业界通用的Nearest neighbor 算法（与OpenCV算法的计算过程类似）	如果用户没有显式调用该接口，迁移到Atlas 推理系列产品（Ascend 310P处理器）后，无需修改代码，直接使用Atlas 推理系列产品（Ascend 310P处理器）上默认的缩放算法，**精度更高**。 如果用户显式调用该接口将缩放算法设置为1或2，则无需修改代码。 如果用户显式调用该接口将缩放算法设置为3或4，则用户修改代码，设置Atlas 推理系列产品（Ascend 310P处理器）所支持的缩放算法。修改代码后，需要重新编译。

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
VPC功能中，YUV400格式图像处理	acldvppSetPicDescSize接口	Atlas 200/300/500 推理产品不支持YUV400格式图像处理，使用YUV420格式进行处理(取Y分量数据)，因此需要将输入图片的格式设置为YUV420，内存大小设置为不小于widthStride*heightStride*3/2的值，VPC会根据YUV420图片格式校验内存大小。	Atlas 推理系列产品（Ascend 310P处理器）支持YUV400格式图像处理，因此直接将输入图片的格式设置为YUV400，内存大小设置为不小于widthStride*heightStride的值，VPC会根据YUV400图片格式校验内存大小。	迁移到Atlas 推理系列产品（Ascend 310P处理器）后，用户可以继续使用“从YUV420格式中取Y分量”来实现YUV400的方式，这样，就不涉及修改代码，也无需重新编译。 另外，由于在Atlas 推理系列产品（Ascend 310P处理器）上扩展支持了YUV400格式，所以如果用户选择直接使用YUV400格式，就需要修改代码，将输出格式设置为YUV400，将内存大小设置为不小于“widthStride*heightStride”的值，**节省内存**。修改代码后，用户需要重新编译。

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
VDEC功能中，设置解码码流最大宽度和高度。	aclvdecSetChannelDescOutPicWidth接口 aclvdecSetChannelDescOutPicHeight接口	• 解码图像帧存大小：创建通道时不会使用解码码流最大宽度和高度申请内部使用的帧存，会在解码过程中根据码流信息获取码流的真实宽高，申请帧存。 • 输入码流缓存大小：1080P分辨率及以下的输入码流，输入码流缓存大小默认为6M；1080P分辨率以上的输入码流，输入码流缓存大小默认为9M。	• 解码图像帧存大小：创建通道时不会使用解码码流最大宽度和高度申请内部使用的帧存，会在解码过程中根据码流信息获取码流的真实宽高，申请帧存。 • 输入码流缓存大小：解码码流最大宽度*解码码流最大高度*2。	迁移到Atlas 推理系列产品（Ascend 310P处理器）后，用户无需修改代码，系统内部会自行根据解码码流的信息申请帧存。 另外，在Atlas 推理系列产品（Ascend 310P处理器）上，系统内部申请的帧存大小需在一个最大值的范围内，该最大值与用户设置的解码码流最大宽度和高度、参考帧数量、码流位宽参数有关，帧存最大值 ≈ (*最大宽度* * *最大高度* * 3/2) * *码流位宽* / 8 * (*参考帧数量* + 3) 建议用户迁移时合理规划VDEC内存，详细描述请参见性能指标说明（Atlas 推理系列产品（Ascend 310P处理器））。

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
VDEC功能中，设置参考帧数量。	aclvdecSetChannelDescRefFrameNum接口	不支持设置参考帧。	支持设置参考帧，如果不设置，默认参考帧数量是8。 在创建通道时，会根据默认值申请帧存，但在解码过程中，会根据码流信息获取码流的真实参考帧信息，进行帧存自适应。	
VENC功能中，指定输出码率。	aclvencSetChannelDescMaxBitRate接口 aclvencSetChannelDescParam接口	输出码率默认值为300。	输出码率默认值为2000。	如果用户没有显式调用接口设置输出码率，迁移到Atlas 推理系列产品（Ascend 310P处理器）后，也无需修改代码，直接使用Atlas 推理系列产品（Ascend 310P处理器）上默认的输出码率值2000，**画质更优**。 如果用户需要调整画质，则需根据实际情况设置输出码率。修改代码后，需要重新编译。 要注意，输出码率不一样，编码输出码流大小会不一样。

功能或约束	涉及的AscendCL接口	Atlas 200/300/500 推理产品的实现	Atlas 推理系列产品（Ascend 310P处理器）的实现	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置通道模式。 （提醒：明确图片数据处理通道用于实现哪种功能（目前支持VPC、JPEGD、JPEGE、PNGD功能），若不调用该接口，则系统默认先创建VPC模式的通道，可能会占用VPC通道数，推荐用户根据实际功能指定通道模式。）	acldvppSetChannelDescMode接口	不支持设置通道模式。	支持设置通道模式，如果不设置通道模式，为了减少通道和内存资源的浪费，会默认先创建VPC模式的通道（会占用VPC通道数）。	迁移到Atlas 推理系列产品（Ascend 310P处理器）后，用户可以不调用该接口设置通道模式，就不涉及修改代码，也无需重新编译。此时，系统内部为了减少通道和内存资源的浪费，默认先创建VPC模式的通道（会占用VPC通道数），在用户调用JPEGD或JPEGE或PNGD功能的接口时，会触发系统内部自动创建JPEGD或JPEGE或PNGD模式的通道。 建议用户根据实际业务，调用该接口设置对应的通道模式，减少VPC通道的占用和内存资源的浪费。

15.4.3 Atlas 200/300/500 推理产品媒体数据处理 V1->Atlas 推理系列产品（Ascend 310P 处理器）媒体数据处理 V2 迁移指引

15.4.3.1 使用场景说明

从Atlas 200/300/500 推理产品的AscendCL媒体数据处理V1版本接口（接口名以acldvpp或aclvdec或aclvenc开头）迁移到Atlas 推理系列产品（Ascend 310P处理器）的AscendCL媒体数据处理V2版本接口（接口名以hi_mpi开头）的场景下，由于接口的调用逻辑、调用示例，差别较大，为方便用户迁移，本节仅根据**常用的关键功能点**列举了V1、V2版本之间接口的对应关系，供迁移的开发人员参考，更多功能及接口说明请参见下表中的相关章节。

表 15-5 应用开发指南中的相关章节说明

媒体数据处理V1版本接口相关章节	媒体数据处理V2版本接口相关章节
接口调用流程及示例在“图像/视频数据处理>**媒体数据处理V1**”章节。	接口调用流程在“图像/视频数据处理>**媒体数据处理V2**”章节。

媒体数据处理V1版本接口相关章节	媒体数据处理V2版本接口相关章节
接口说明在“AscendCL API参考>媒体数据处理V1”章节。	接口说明在“AscendCL API参考>媒体数据处理V2”章节

15.4.3.2 头文件迁移

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
CANN软件安装目录/include/acl/ops/acl_dvpp.h	CANN软件安装目录/include/acl/dvpp下的头文件	用户需要修改代码，include“CANN软件安装目录/include/acl/dvpp”目录下的hi_dvpp.h文件。

15.4.3.3 库文件迁移

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
CANN软件安装目录/lib64/libacl_dvpp.so	CANN软件安装目录/lib64/libacl_dvpp_mpi.so	用户需要修改编译应用程序的脚本，修改依赖的库文件，改成libacl_dvpp_mpi.so。

15.4.3.4 VPC 功能迁移

VPC功能，命名以acldvpp开头的接口与命名以hi_mpi开头的接口，最大的区别在于：用acldvpp接口实现VPC功能时，设置输入、输出图片信息，都是通过set接口，图片处理结束后，再通过get接口获取结果数据所在的内存地址；用hi_mpi命名的接口实现VPC功能时，设置输入、输出图片信息都是通过给结构体中的成员变量赋值，图片处理结束后，从对应的内存地址中获取结果数据。

15.4.3.4.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.4.2 创建通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： acldvppCreateChannelDesc	无对应接口，直接使用hi_vpc_chn_attr结构体，且该结构体参数内部保留，用户配置任意值（除空指针）即可。	用户需要修改代码，直接声明hi_vpc_chn_attr结构体的变量。
创建通道： acldvppCreateChannel	创建通道： hi_mpi_vpc_create_chn	用户需要修改代码，改用hi_mpi_vpc_create_chn接口创建通道。

15.4.3.4.3 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
申请内存： acldvppMalloc	申请内存： hi_mpi_dvpp_malloc	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.4.4 VPC 抠图缩放贴图功能（一图一框）

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyPicDesc	无对应接口，配置输入图片参数时，直接对hi_vpc_pic_info结构体的成员赋值： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，对结构体hi_vpc_pic_info成员赋值来配置输入图片信息的参数。
配置输出图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyPicDesc	无对应接口，配置输出图片参数时，直接对hi_vpc_crop_resize_paste_region结构体内的dest_pic_info成员赋值，dest_pic_info成员是hi_vpc_pic_info结构体，该结构体内的成员如下： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的dest_pic_info成员赋值。

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置从输入图片中的抠图位置信息： acldvppCreateRoiConfig acldvppSetRoiConfig acldvppSetRoiConfigLeft acldvppSetRoiConfigRight acldvppSetRoiConfigTop acldvppSetRoiConfigBottom acldvppDestroyRoiConfig	无对应接口，配置抠图位置信息时，直接对hi_vpc_crop_resize_paste_region结构体内的crop_region成员赋值，crop_region成员是hi_vpc_crop_region结构体，该结构体内的成员如下： top_offset; left_offset; crop_width; crop_height;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的crop_region成员赋值。
配置缩放算法信息： acldvppCreateResizeConfig acldvppSetResizeConfigInterpolation acldvppDestroyResizeConfig	无对应接口，配置缩放信息时，直接对hi_vpc_crop_resize_paste_region结构体内的resize_info成员赋值，resize_info成员是hi_vpc_resize_info结构体，该结构体内的成员如下： resize_width; resize_height; interpolation;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的resize_info成员赋值。

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输出图片中的贴图位置信息： acldvppCreateRoiConfig acldvppSetRoiConfig acldvppSetRoiConfigLeft acldvppSetRoiConfigRight acldvppSetRoiConfigTop acldvppSetRoiConfigBottom acldvppDestroyRoiConfig	无对应接口，配置贴图位置信息时，直接对hi_vpc_crop_resize_paste_region结构体内的dest_top_offset成员、dest_left_offset成员赋值。	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的dest_top_offset成员、dest_left_offset成员赋值。
抠图缩放贴图功能： acldvppVpcCropResizePasteAsync	抠图缩放贴图功能： hi_mpi_vpc_crop_resize_paste	用户需要修改代码，改成调用hi_mpi_vpc_crop_resize_paste接口。 注意： hi_mpi_vpc_crop_resize_paste接口支持一图一框和一图多框，如果想使用一图多框功能，则需要配置每个框的抠图位置信息、缩放算法信息、贴图位置信息。

15.4.3.4.5 VPC 批量抠图缩放贴图功能

Atlas 200/300/500 推理产品 acldvpp 接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入batch的图片信息： acldvppCreateBatchPicDesc acldvppGetPicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyBatchPicDesc	无对应接口，配置输入图片参数时，直接对hi_vpc_pic_info结构体数据的成员赋值，其中数组的每一个元素代表每一个输入图片，pic_num代表batch大小： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，对结构体hi_vpc_pic_info成员赋值来配置输入图片信息的参数。
配置每个输入图片的抠图数量： roiNums size	数据结构对应如下，数组长度跟输入pic_num长度一致： count[]	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的dest_pic_info成员赋值。

Atlas 200/300/500 推理产品 acldvpp 接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输出batch的图片信息： acldvppCreateBatchPicDesc acldvppGetPicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyBatchPicDesc	无对应接口，配置输出图片参数时，直接对hi_vpc_crop_resize_paste_region结构体内的dest_pic_info成员赋值，dest_pic_info成员是hi_vpc_pic_info结构体，该结构体内的成员如下： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的dest_pic_info成员赋值。
配置从输入图片中的抠图位置信息： acldvppCreateRoiConfig acldvppSetRoiConfig acldvppSetRoiConfigLeft acldvppSetRoiConfigRight acldvppSetRoiConfigTop acldvppSetRoiConfigBottom acldvppDestroyRoiConfig	无对应接口，配置抠图位置信息时，直接对hi_vpc_crop_resize_paste_region结构体内的crop_region成员赋值，crop_region成员是hi_vpc_crop_region结构体，该结构体内的成员如下： top_offset; left_offset; crop_width; crop_height;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的crop_region成员赋值。

Atlas 200/300/500 推理产品 acldvpp 接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置缩放算法信息： acldvppCreateResizeConfig acldvppSetResizeConfigInterpolation acldvppDestroyResizeConfig	无对应接口，配置缩放信息时，直接对hi_vpc_crop_resize_paste_region结构体内的resize_info成员赋值，resize_info成员是hi_vpc_resize_info结构体，该结构体内的成员如下： resize_width; resize_height; interpolation;	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的resize_info成员赋值。
配置输出图片中的贴图位置信息： acldvppCreateRoiConfig acldvppSetRoiConfig acldvppSetRoiConfigLeft acldvppSetRoiConfigRight acldvppSetRoiConfigTop acldvppSetRoiConfigBottom acldvppDestroyRoiConfig	无对应接口，配置贴图位置信息时，直接对hi_vpc_crop_resize_paste_region结构体内的dest_top_offset成员、dest_left_offset成员赋值。	用户需要修改代码，改为对hi_vpc_crop_resize_paste_region结构体内的dest_top_offset成员、dest_left_offset成员赋值。
批量抠图缩放贴图功能： acldvppVpcBatchCropResizePasteAsync	批量抠图缩放贴图功能： hi_mpi_vpc_batch_crop_resize_paste	用户需要修改代码，改成调用hi_mpi_vpc_batch_crop_resize_paste接口。 需注意：原始图片数量pic_num，与source_pic数组长度、count数组长度保持一致。

15.4.3.4.6 VPC 缩放功能

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyPicDesc	无对应接口，配置输入图片参数时，直接对source_pic赋值，source_pic是hi_vpc_pic_info结构体，该结构体内的成员如下： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，改为对source_pic赋值。
配置输出图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride acldvppSetPicDescFormat acldvppDestroyPicDesc	无对应接口，配置输出图片参数时，直接对dest_pic赋值，dest_pic是hi_vpc_pic_info结构体，该结构体内的成员如下： picture_address; picture_buffer_size; picture_width; picture_height; picture_width_stride; picture_height_stride; picture_format;	用户需要修改代码，改为对dest_pic赋值。

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置缩放算法信息： acldvppCreateResizeConfig acldvppSetResizeConfigInterpolation acldvppDestroyResizeConfig	无对应接口，直接对interpolation进行赋值	用户需要修改代码，改为对interpolation赋值。
缩放功能： acldvppVpcResizeAsync	缩放功能： hi_mpi_vpc_resize	用户需要修改代码，改成调用hi_mpi_vpc_resize接口。

15.4.3.4.7 获取处理结果

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
等待任务处理完成后，从接口acldvppVpcCropResizePasteAsync的输出参数outputDesc中调用对应的get接口获取输出图片数据。 ● 等待任务完成接口如下：aclrtSynchronizeStream ● 对应的get接口如下：acldvppGetPicDescData acldvppGetPicDescWidth acldvppGetPicDescHeight acldvppGetPicDescWidthStride acldvppGetPicDescHeightStride acldvppGetPicDescFormat	等待任务处理完成后，获取输出图片数据。 ● 等待任务完成的接口如下：hi_mpi_vpc_get_process_result ● 针对不同功能，获取输出图片数据的参数如下： 使用hi_mpi_vpc_crop_resize_paste接口实现抠图缩放贴图功能时，从hi_vpc_crop_resize_paste_region.dest_pic_info.picture_address参数获取输出图片数据。 使用hi_mpi_vpc_batch_crop_resize_paste接口实现批量抠图缩放贴图功能时，从hi_vpc_crop_resize_paste_region.dest_pic_info.picture_address参数获取输出图片数据。 使用hi_mpi_vpc_resize接口实现缩放功能时，从dest_pic.picture_address参数获取输出图片数据。	用户需要修改代码，将aclrtSynchronizeStream接口修改成hi_mpi_vpc_get_process_result接口，并改为直接从对应的成员参数中获取输出图片数据。

15.4.3.4.8 销毁通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： acldvppDestroyChannel	销毁通道： hi_mpi_vpc_destroy_chn	用户需要修改代码，改为调用hi_mpi_vpc_destroy_chn接口。

15.4.3.4.9 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

15.4.3.5 VDEC 功能迁移

VDEC功能，命名以acldvpp或aclvdec开头的接口与命名以hi_mpi开头的接口，最大的区别在于：用acldvpp接口实现VDEC功能时，需设置回调函数，每解码一帧数据会自动调用回调函数，在回调函数内调用对应acldvpp的Get接口获取解码数据的内存地址、内存大小等；用hi_mpi命名的接口实现VDEC功能时，需要用户另起线程调用hi_mpi_vdec_get_frame接口来获取解码结果。

15.4.3.5.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.5.2 创建通道

Atlas 200/300/500 推理产品 aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： aclvdecCreateChannelDesc	无对应接口，直接使用hi_vdec_chn_attr结构体：	用户需要修改代码，直接声明hi_vdec_chn_attr结构体的变量。

Atlas 200/300/500 推理产品 aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置通道描述的参数： aclvdecSetChannelDescEnType aclvdecSetChannelDescOutPicWidth aclvdecSetChannelDescOutPicHeight aclvdecSetChannelDescRefFrameNum aclvdecSetChannelDescChannelId（仅acl有） aclvdecSetChannelDescThreadId（仅acl有） aclvdecSetChannelDescCallback（仅acl有） aclvdecSetChannelDescOutPicFormat（仅acl有） aclvdecSetChannelDescOutMode（仅acl有）	无对应接口，直接对hi_vdec_chn_attr结构体的成员赋值： hi_vdec_chn_attr.type hi_vdec_chn_attr.pic_width hi_vdec_chn_attr.pic_height hi_vdec_chn_attr.video_attr.ref_frame_num hi_vdec_chn_attr.mode（仅hi_mpi有） hi_vdec_chn_attr.stream_buf_size（仅hi_mpi有） hi_vdec_chn_attr.frame_buf_cnt（仅hi_mpi有） hi_vdec_chn_attr.frame_buf_size（仅hi_mpi有） hi_vdec_chn_attr.video_attr.temporal_mvp_en（仅hi_mpi有） hi_vdec_chn_attr.video_attr.tmv_buf_size（仅hi_mpi有）	用户需要修改代码，对结构体hi_vdec_chn_attr成员赋值来配置通道描述的参数。 需注意，对于仅acl有的这几个参数设置，在hi_mpi上也有对应的接口： ● 通道ID，不在hi_vdec_chn_attr结构体中，而是在用hi_mpi_vdec_create_chn接口创建通道时传入。 ● 线程ID、Callback（这两个跟回调函数相关），不在hi_vdec_chn_attr结构体中，因为使用hi mpi接口时不需要设置回调函数，需要自行另起线程，调用hi_mpi_vdec_get_frame接口来获取解码结果。 ● 输出图片格式outPicFormat参数不在hi_vdec_chn_attr结构体内，是在调用hi_mpi_vdec_send_stream接口发送码流数据时通过hi_vdec_pic_info.pixel_format参数指定。 ● 设置是否实时出帧outMode参数（即发送一帧解码一帧，无需依赖后续帧的传入）不在hi_vdec_chn_attr结构体内，是在调用hi_mpi_vdec_set_chn_param接口设置通道属性时通过hi_vdec_chn_param.h

Atlas 200/300/500 推理产品 aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
		i_vdec_video_param.hi_video_out_order参数指定的，其中，解码序表示快速出帧。
无对应接口	设置通道属性： hi_mpi_vdec_set_chn_param	设置解码出帧模式时，需要调用hi_mpi_vdec_set_chn_param接口进行设置。
创建通道 aclvdecCreateChannel	创建通道 hi_mpi_vdec_create_chn	用户需要修改代码，改用hi_mpi_vdec_create_chn接口创建通道。

15.4.3.5.3 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
申请内存： acldvppMalloc	申请内存： hi_mpi_dvpp_malloc	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.5.4 发送码流

Atlas 200/300/500 推理产品 acldvpp/aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入码流信息： acldvppSetStreamDescData acldvppSetStreamDescSize acldvppSetStreamDescEos	无对应接口，配置输入码流参数时，直接对hi_vdec_stream结构体的成员赋值： hi_vdec_stream.addr hi_vdec_stream.len hi_vdec_stream.end_of_stream	用户需要修改代码，对结构体hi_vdec_chn_attr成员赋值来配置输入码流信息的参数。 hi_mpi接口需要在所有码流数据发送结束后主动发送eos帧，将hi_vdec_stream.end_of_stream设置为HI_TRUE，其他码流发送时将hi_vdec_stream.end_of_stream设置为HI_FALSE。
配置输出图片信息： acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescFormat acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride	无对应接口，配置输出图片参数时，直接对hi_vdec_pic_info结构体的成员赋值： hi_vdec_pic_info.vir_addr hi_vdec_pic_info.buffer_size hi_vdec_pic_info.pixel_format hi_vdec_pic_info.width hi_vdec_pic_info.height hi_vdec_pic_info.width_stride hi_vdec_pic_info.height_stride hi_vdec_stream.need_display（仅hi_mpi有）	用户需要修改代码，对结构体hi_vdec_pic_info成员赋值来配置输出图片信息的参数。 需注意，acl接口中，如果不想获取某一帧的解码结果，通过调用aclvdecSendSkippedFrame接口实现，但在hi_mpi接口中，是通过设置hi_vdec_stream.need_display来完成。
无对应接口，无需通知解码器开始接收码流数据	通知解码器开始接收码流数据（一个通道只需要调用一次该接口）： hi_mpi_vdec_start_recv_stream	用户需要修改代码，增加调用hi_mpi_vdec_start_recv_stream接口。

Atlas 200/300/500 推理产品 acldvpp/aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
aclvdecSendSkippedFrame	hi_vdec_stream.need_display（仅hi_mpi有）	acl接口中，如果不想获取某一帧的解码结果，通过调用aclvdecSendSkippedFrame接口实现，但在hi_mpi接口中，是通过设置hi_vdec_stream.need_display来完成。
每一帧都需调用以下接口发送解码码流： aclvdecSendFrame	每一帧都需调用以下接口发送解码码流： hi_mpi_vdec_send_stream	用户需要修改代码，改为调用hi_mpi_vdec_send_stream接口发送码流，同时需要传入通道号。

15.4.3.5.5 接收码流

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置通道描述信息时已注册回调函数，每解码一帧数据会自动调用回调函数，在回调函数内调用对应的Get接口获取解码数据的内存地址、内存大小等。 acldvppGetPicDescData acldvppGetPicDescWidth acldvppGetPicDescHeight acldvppGetPicDescWidthStride acldvppGetPicDescHeightStride acldvppGetPicDescFormat acldvppGetPicDescRetCode（仅acl有）	调用hi_mpi_vdec_get_frame接口获取一帧解码结果，再从hi_video_frame_info结构体的以下成员中获取解码数据： hi_video_frame_info.v_frame.virt_addr[0] hi_video_frame_info.v_frame.width hi_video_frame_info.v_frame.height hi_video_frame_info.v_frame.width_stride[0] hi_video_frame_info.v_frame.height_stride[0] hi_video_frame_info.v_frame.pixel_format hi_video_frame_info.v_frame.frame_flag（与retCode对应）	用户需要修改代码，改为另起线程，调用hi_mpi_vdec_get_frame接口来主动获取解码结果。
无对应接口，无需通知解码器停止接收码流数据	通知解码器停止接收码流数据（一个通道只需要调用一次该接口）： hi_mpi_vdec_stop_recv_stream	用户需要修改代码，增加调用hi_mpi_vdec_stop_recv_stream接口。

15.4.3.5.6 销毁通道

Atlas 200/300/500 推理产品 aclvdec接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： aclvdecDestroyChannel	销毁通道： hi_mpi_vdec_destroy_chn	用户需要修改代码，改为调用hi_mpi_vdec_destroy_chn接口。 需注意，在调用hi_mpi接口销毁通道前，可以先调用hi_mpi_vdec_query_status接口获取解码器状态，判断是否所有码流数据都已完成解码，然后进行通道的销毁。

15.4.3.5.7 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

15.4.3.6 JPEGD 功能迁移

JPEGD功能，命名以acldvpp开头的接口与命名以hi_mpi开头的接口，最大的区别在于：用acldvpp接口实现JPEGD功能时，无需通知解码器开始或结束接收码流数据；用hi_mpi命名的接口实现JPEGD功能时，需通知解码器开始或结束接收码流数据，另外，用户可以另起线程调用hi_mpi_vdec_get_frame接口来获取解码结果。

15.4.3.6.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.6.2 创建通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： acldvppCreateChannelDesc	无对应接口，直接使用hi_vdec_chn_attr结构体。	用户需要修改代码，直接声明hi_vdec_chn_attr结构体的变量。
无对应接口	直接对hi_vdec_chn_attr结构体的成员赋值： hi_vdec_chn_attr.type（仅hi_mpi有） hi_vdec_chn_attr.pic_width（仅hi_mpi有） hi_vdec_chn_attr.pic_height（仅hi_mpi有） hi_vdec_chn_attr.mode（仅hi_mpi有） hi_vdec_chn_attr.stream_buf_size（仅hi_mpi有） hi_vdec_chn_attr.frame_buf_cnt（仅hi_mpi有） hi_vdec_chn_attr.frame_buf_size（仅hi_mpi有）	用户需要修改代码，对结构体hi_vdec_chn_attr成员赋值来配置通道描述的参数。 需注意，对于仅acl有的这几个参数设置，在hi_mpi上也有对应的接口：
创建通道 acldvppCreateChannel	创建通道 hi_mpi_vdec_create_chn	用户需要修改代码，改用hi_mpi_vdec_create_chn接口创建通道。

15.4.3.6.3 预估输出内存大小

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
acldvppJpegPredictDecSize	hi_mpi_dvpp_get_image_info 源格式解码场景下，可调用hi_mpi_dvpp_get_image_info接口获取解码输出内存的大小； 输出图片格式为YUV420SP的场景下，需使用计算公式计算输出内存大小：hi_img_info.width_stride * hi_img_info.height_stride * 3 /2	用户需要修改代码，改用hi_mpi_dvpp_get_image_info接口或使用计算公式获取图片解码后的输出内存大小。 注意：width_stride 、height_stride的取值要求请参见图片格式、宽高对齐、内存约束。

15.4.3.6.4 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
申请内存： acldvppMalloc	申请内存： hi_mpi_dvpp_malloc	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.6.5 发送图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入码流信息： 无对应接口，在调用acldvppJpegDecodeAsync时直接传入输入图片信息	配置输入码流参数时，直接对hi_vdec_stream结构体的成员赋值： hi_vdec_stream.addr hi_vdec_stream.len hi_vdec_stream.end_of_stream hi_vdec_stream.end_of_frame hi_vdec_stream.display	用户需要修改代码，对结构体hi_vdec_chn_attr成员赋值来配置输入码流信息的参数。
配置输出图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescFormat acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride	无对应接口，配置输出图片参数时，直接对hi_vdec_pic_info结构体的成员赋值： hi_vdec_pic_info.vir_addr hi_vdec_pic_info.buffer_size hi_vdec_pic_info.pixel_format hi_vdec_pic_info.width hi_vdec_pic_info.height hi_vdec_pic_info.width_stride hi_vdec_pic_info.height_stride	用户需要修改代码，对结构体hi_vdec_pic_info成员赋值来配置输出图片信息的参数。
无对应接口，无需通知解码器开始接收码流数据	通知解码器开始接收码流数据（一个通道只需要调用一次该接口）： hi_mpi_vdec_start_recv_stream	用户需要修改代码，增加调用hi_mpi_vdec_start_recv_stream接口。
每一帧都需调用以下接口发送图片： acldvppJpegDecodeAsync	每一帧都需调用以下接口发送解码码流： hi_mpi_vdec_send_stream	用户需要修改代码，改为调用hi_mpi_vdec_send_stream接口发送码流，同时需要传入通道号。

15.4.3.6.6 接收图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
需要调用aclrtSynchronizeStream流同步接口，等待图片解码结束，然后获取解码结果： acldvppGetPicDescData acldvppGetPicDescWidth acldvppGetPicDescHeight acldvppGetPicDescWidthStride acldvppGetPicDescHeightStride acldvppGetPicDescFormat acldvppGetPicDescRetCode（仅acl有）	调用hi_mpi_vdec_get_frame接口获取一帧解码结果，再从hi_video_frame_info结构体的以下成员中获取解码结果： hi_video_frame_info.v_frame.virt_addr[0] hi_video_frame_info.v_frame.width hi_video_frame_info.v_frame.height hi_video_frame_info.v_frame.width_stride hi_video_frame_info.v_frame.height_stride hi_video_frame_info.v_frame.pixel_format hi_video_frame_info.v_frame.frame_flag（仅hi_mpi有）	用户需要修改代码，改为另起线程，调用hi_mpi_vdec_get_frame接口来主动获取解码结果。
无对应接口，无需通知解码器停止接收码流数据	通知解码器停止接收码流数据（一个通道只需要调用一次该接口）： hi_mpi_vdec_stop_recv_stream	用户需要修改代码，增加调用hi_mpi_vdec_stop_recv_stream接口。

15.4.3.6.7 销毁通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： acldvppDestroyChannel	销毁通道： hi_mpi_vdec_destroy_chn	用户需要修改代码，改为调用hi_mpi_vdec_destroy_chn接口。 需注意，在调用hi_mpi接口销毁通道前，可以先调用hi_mpi_vdec_query_status接口获取解码器状态，判断是否所有图片数据都已完成解码，然后进行通道的销毁。

15.4.3.6.8 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

15.4.3.7 JPEGE 功能迁移

JPEGE功能，命名以acldvpp开头的接口与命名以hi_mpi开头的接口，最大的区别在于：用acldvpp接口实现JPEGE功能时，每编码一帧图片，需要传入输出内存地址，并且通过流同步等待一帧编码完成，然后在传入的输出内存中获取编码结果；用hi_mpi命名的接口实现JPEGE功能时，通过用户另起线程调用hi_mpi_venc_get_stream接口来异步获取编码结果。

15.4.3.7.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.7.2 创建通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： acldvppCreateChannelDesc	无对应接口，直接使用hi_venc_chn_attr结构体。	用户需要修改代码，直接声明hi_venc_chn_attr结构体的变量。
无相关接口	直接对hi_venc_chn_attr结构体的成员赋值： hi_venc_chn_attr.venc_attr.type hi_venc_chn_attr.venc_attr.pic_width hi_venc_chn_attr.venc_attr.pic_height	用户需要修改代码，对结构体hi_venc_chn_attr成员赋值来配置通道描述的参数。
创建通道 acldvppCreateChannel	创建通道 hi_mpi_venc_create_chn	用户需要修改代码，改用hi_mpi_venc_create_chn接口创建通道。

15.4.3.7.3 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
调用acldvppJpegPredictEncSize来预估JPEG图片编码后所需的输出内存的大小，再调用acldvppMalloc接口申请输出内存。	输出内存可以由系统管理，这时用户无需申请输出内存； 输出内存也可以由用户管理，这时用户可调用hi_mpi_venc_get_jpege_predicted_size接口预估JPEG图片编码后所需的输出内存的大小，再调用hi_mpi_dvpp_malloc接口申请内存。	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.7.4 发送编码图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescFormat acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride	无对应接口，配置输入图片时，直接对hi_video_frame_info结构体的成员赋值： hi_video_frame_info.v_frame.virt_addr hi_video_frame_info.v_frame.pixel_format hi_video_frame_info.v_frame.width hi_video_frame_info.v_frame.height hi_video_frame_info.v_frame.width_stride hi_video_frame_info.v_frame.height_stride	用户需要修改代码，对结构体hi_video_frame_info成员赋值来配置输入图片的参数。 hi_mpi接口无需设置输入数据的大小，系统内部会根据图片分辨率、格式等信息自行计算。
配置图片编码质量 acldvppCreateJpegeConfig acldvppSetJpegeConfigLevel	配置图片编码质量，直接修改hi_venc_jpeg_param.qfactor的值 hi_mpi_venc_set_jpeg_param	用户需要修改代码，改为调用hi_mpi_venc_set_jpeg_param来配置JPEGE的图片编码质量。
无对应接口，无需通知编码器开始接收编码图片	通知编码器开始接收输入数据（一个通道只需要调用一次该接口）： hi_mpi_venc_start_chn	用户需要修改代码，增加调用hi_mpi_venc_start_chn接口。
每一帧都需调用以下接口发送编码图片，同时传入用户申请好的输出内存地址： acldvppJpegEncodeAsync	由系统管理输出内存时，需调用hi_mpi_venc_send_frame接口，每一帧都需调用该接口发送编码图片； 由用户管理输出内存时，需调用hi_mpi_venc_send_jpege_frame接口，每一帧都需调用该接口发送编码图片。	用户需要修改代码，改为调用hi_mpi_venc_send_frame接口或hi_mpi_venc_send_jpege_frame接口发送编码图片，同时需要传入通道号。

15.4.3.7.5 获取编码图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
需要调用aclrtSynchronizeStream流同步接口，等待图片编码结束，然后获取编码结果： 编码后图片存放地址为调用acldvppJpegEncodeAsync时传入的输出内存地址，图片大小从acldvppJpegEncodeAsync输出参数size获取	调用hi_mpi_venc_get_stream接口获取一帧编码结果，再从hi_venc_stream结构体的以下成员中获取编码数据： hi_venc_stream.pack[i].addr hi_venc_stream.pack[i].len hi_venc_stream.pack[i].data_type（从该成员结构体中获取输出图片格式） hi_venc_stream.pack[i].pts	用户需要修改代码，改为另起线程，调用hi_mpi_venc_get_stream接口来主动获取编码结果。 需注意，用户需要通过调用hi_mpi_venc_query_status得到每次编码出的包数n，然后再调用hi_mpi_venc_get_stream前，提前给参数hi_venc_stream.pack分配不小于n * sizeof（hi_venc_pack）大小的内存。 获取编码结果成功后，需调用hi_mpi_venc_release_stream接口释放码流缓存。
无对应接口，无需通知编码器停止接收编码数据帧	通知编码器停止接收输入数据（一个通道只需要调用一次该接口）： hi_mpi_venc_stop_chn	用户需要修改代码，增加调用hi_mpi_venc_stop_chn接口。

15.4.3.7.6 销毁通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： acldvppDestroyChannel	销毁通道： hi_mpi_venc_destroy_chn	用户需要修改代码，改为调用hi_mpi_venc_destroy_chn接口。 需注意，在调用hi_mpi接口销毁通道前，可以先调用hi_mpi_venc_query_status接口获取编码器状态，判断是否所有输入数据都已完成编码，然后进行通道的销毁。

15.4.3.7.7 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

15.4.3.8 VENC 功能迁移

VENC功能，命名以acldvpp或aclvenc开头的接口与命名以hi_mpi开头的接口，最大的区别在于：用acldvpp接口实现VENC功能时，需设置回调函数，每编码一帧数据会自动调用回调函数，在回调函数内调用对应的acl的Get接口获取编码数据的内存地址、内存大小等；用hi_mpi命名的接口实现VENC功能时，需要用户另起线程调用hi_mpi_venc_get_stream接口来获取编码结果。

15.4.3.8.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.8.2 创建通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： aclvencCreateChannelDesc	无对应接口，直接使用hi_venc_chn_attr结构体。	用户需要修改代码，直接声明hi_venc_chn_attr结构体的变量。

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置通道描述的参数： aclvencSetChannelDescEnType aclvencSetChannelDescPicWidth aclvencSetChannelDescPicHeight aclvencSetChannelDescKeyFrameInterval aclvencSetChannelDescThreadId（仅acl有） aclvencSetChannelDescCallback（仅acl有） aclvencSetChannelDescRcMode aclvencSetChannelDescSrcRate aclvencSetChannelDescMaxBitRate aclvencSetChannelDescPicFormat（仅acl有）	无对应接口，直接对hi_venc_chn_attr结构体的成员赋值： hi_venc_chn_attr.venc_attr.type hi_venc_chn_attr.venc_attr.pic_width hi_venc_chn_attr.venc_attr.pic_height hi_venc_chn_attr.rc_attr.h26x_xbr.gop hi_venc_chn_attr.rc_attr.rc_mode hi_venc_chn_attr.rc_attr.h26x_xbr.src_frame_rate hi_venc_chn_attr.rc_attr.h26x_xbr.bit_rate	用户需要修改代码，对结构体hi_venc_chn_attr成员赋值来配置通道描述的参数。 需注意，对于仅acl有的这几个参数设置，在hi_mpi上也有对应的接口： ● 线程ID、Callback（这两个跟回调函数相关），不在hi_venc_chn_attr结构体中，因为使用hi mpi接口时不需要设置回调函数，需要自行另起线程，调用hi_mpi_venc_get_stream接口来获取编码结果。 ● 输入图片格式不在hi_venc_chn_attr结构体中，是在调用hi_mpi_venc_send_frame接口发送编码数据帧时通过hi_video_frame_info.v_frame.pixel_format参数指定。
创建通道： aclvencCreateChannel	创建通道： hi_mpi_venc_create_chn	用户需要修改代码，改用hi_mpi_venc_create_chn接口创建通道。

15.4.3.8.3 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
申请内存： acldvppMalloc	申请内存： hi_mpi_dvpp_malloc	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.8.4 发送编码视频帧

Atlas 200/300/500 推理产品 acldvpp/aclvenc接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入视频帧信息： acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescFormat acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride	无对应接口，配置输入视频帧时，直接对hi_video_frame_info结构体的成员赋值： hi_video_frame_info.v_frame.virt_addr hi_video_frame_info.v_frame.pixel_format hi_video_frame_info.v_frame.width hi_video_frame_info.v_frame.height hi_video_frame_info.v_frame.width_stride hi_video_frame_info.v_frame.height_stride	用户需要修改代码，对结构体hi_video_frame_info成员赋值来配置输入视频帧的参数。 hi_mpi接口无需设置输入数据的大小，系统内部会根据图片分辨率、格式等信息自行计算。
无对应接口，无需通知编码器开始接收编码数据帧	通知编码器开始接收编码数据帧（一个通道只需要调用一次该接口）： hi_mpi_venc_start_chn	用户需要修改代码，增加调用hi_mpi_venc_start_chn接口。

Atlas 200/300/500 推理产品 acldvpp/aclvenc接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
每一帧都需调用以下接口发送编码视频帧： aclvencSendFrame	每一帧都需调用以下接口发送编码视频帧： hi_mpi_venc_send_frame	用户需要修改代码，改为调用hi_mpi_venc_send_frame接口发送编码视频帧，同时需要传入通道号。

15.4.3.8.5 获取码流

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
设置通道描述信息时已注册回调函数，每编码一帧数据会自动调用回调函数，在回调函数内调用对应的Get接口获取编码数据的内存地址、内存大小等。 acldvppGetStreamDescData acldvppGetStreamDescSize acldvppGetStreamDescFormat acldvppGetStreamDescTimestamp	调用hi_mpi_venc_get_stream接口获取一帧编码结果，再从hi_venc_stream结构体的以下成员中获取编码数据： hi_venc_stream.pack[i].addr hi_venc_stream.pack[i].len hi_venc_stream.pack[i].data_type（从该成员结构体中获取输出图片格式） hi_venc_stream.pack[i].pts	用户需要修改代码，改为另起线程，调用hi_mpi_venc_get_stream接口来主动获取编码结果。 需注意，用户需要通过调用hi_mpi_venc_query_status得到每次编码出的包数n，然后再调用hi_mpi_venc_get_stream前，提前给参数hi_venc_stream.pack分配不小于n * sizeof（hi_venc_pack）大小的内存。 获取编码结果成功后，需调用hi_mpi_venc_release_stream接口释放码流缓存。
无对应接口，无需通知编码器停止接收编码数据帧	通知编码器停止接收编码数据帧（一个通道只需要调用一次该接口）： hi_mpi_venc_stop_chn	用户需要修改代码，增加调用hi_mpi_venc_stop_chn接口。

15.4.3.8.6 销毁通道

Atlas 200/300/500 推理产品 aclvenc接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： aclvencDestroyChannel	销毁通道： hi_mpi_venc_destroy_chn	用户需要修改代码，改为调用hi_mpi_venc_destroy_chn接口。 需注意，在调用hi_mpi接口销毁通道前，可以先调用hi_mpi_venc_query_status接口获取编码器状态，判断是否所有输入数据都已完成编码，然后进行通道的销毁。

15.4.3.8.7 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

15.4.3.9 PNGD 功能迁移

15.4.3.9.1 媒体数据处理模块初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块初始化	hi_mpi_sys_init	用户需要修改代码，增加调用媒体数据处理模块初始化的接口hi_mpi_sys_init。

15.4.3.9.2 创建通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
创建通道描述信息的结构体： acldvppCreateChannelDesc	无对应接口，直接使用hi_pngd_chn_attr结构体。	用户需要修改代码，直接声明hi_pngd_chn_attr结构体的变量。
创建通道： acldvppCreateChannel	创建通道： hi_mpi_pngd_create_chn	用户需要修改代码，改用hi_mpi_pngd_create_chn接口创建通道。

15.4.3.9.3 预估输出内存

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
预估输出内存： acldvppPngPredictDecSize	源格式解码场景下，可调用hi_mpi_png_get_image_info接口获取解码输出内存的大小； 其它场景下，需使用计算公式计算输出内存大小：hi_img_info.width_stride * hi_img_info.height_stride	用户需要修改代码，改用hi_mpi_png_get_image_info接口或使用计算公式获取图片解码后的输出内存大小。 注意：width_stride 、height_stride的取值要求请参见图片格式、宽高对齐、内存约束。

15.4.3.9.4 内存申请

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
申请内存： acldvppMalloc	申请内存： hi_mpi_dvpp_malloc	用户需要修改代码，改用hi_mpi_dvpp_malloc接口申请媒体数据处理功能需要的内存。
释放内存 acldvppFree	释放内存： hi_mpi_dvpp_free	用户需要修改代码，改用hi_mpi_dvpp_free接口释放内存。

15.4.3.9.5 发送图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器）hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
配置输入码流信息： 无对应接口，在调用acldvppPngDecodeAsync时直接传入输入图片信息	配置输入码流参数时，直接对hi_img_stream结构体的成员赋值： hi_img_stream.type hi_img_stream.addr hi_img_stream.len hi_img_stream.pts	用户需要修改代码，对结构体hi_vdec_chn_attr成员赋值来配置输入码流信息的参数。
配置输出图片信息： acldvppCreatePicDesc acldvppSetPicDescData acldvppSetPicDescSize acldvppSetPicDescFormat acldvppSetPicDescWidth acldvppSetPicDescHeight acldvppSetPicDescWidthStride acldvppSetPicDescHeightStride	无对应接口，配置输出图片参数时，直接对hi_pic_info结构体的成员赋值： hi_pic_info.picture_address hi_pic_info.picture_buffer_size hi_pic_info.picture_format hi_pic_info.picture_width hi_pic_info.picture_height hi_pic_info.picture_width_stride hi_pic_info.picture_height_stride	用户需要修改代码，对结构体hi_pic_info成员赋值来配置输出图片信息的参数。

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
每一帧都需调用以下接口发送图片： acldvppPngDecodeAsync	每一帧都需调用以下接口发送解码码流： hi_mpi_pngd_send_stream	用户需要修改代码，改为调用hi_mpi_pngd_send_stream接口发送码流，同时需要传入通道号。

15.4.3.9.6 接收图片

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
需要调用aclrtSynchronizeStream流同步接口，等待图片解码结束，然后获取解码结果： acldvppGetPicDescData acldvppGetPicDescWidth acldvppGetPicDescHeight acldvppGetPicDescWidthStride acldvppGetPicDescHeightStride acldvppGetPicDescFormat acldvppGetPicDescRetCode（仅acl有，表示解码状态，即解码是否成功）	调用hi_mpi_pngd_get_image_data接口获取一帧解码结果，再从hi_pic_info结构体的以下成员中获取解码结果： hi_pic_info.picture_address hi_pic_info.picture_width hi_pic_info.picture_height hi_pic_info.picture_width_stride hi_pic_info.picture_height_stride hi_pic_info.picture_format hi_pic_info.picture_buffer_size（仅himpi有）	用户需要修改代码，改为另起线程，调用hi_mpi_pngd_get_image_data接口来主动获取解码结果。 注意，hi_mpi接口中，如果能获取到解码数据，则表示解码成功；如果获取不到解码数据，则解码失败。

15.4.3.9.7 销毁通道

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
销毁通道： acldvppDestroyChannel	销毁通道： hi_mpi_pngd_destroy_chn	用户需要修改代码，改为调用hi_mpi_pngd_destroy_chn接口。 需注意，在调用hi_mpi接口销毁通道前，需要确保所有图片数据都已完成解码，然后进行通道的销毁。

15.4.3.9.8 媒体数据处理模块去初始化

Atlas 200/300/500 推理产品 acldvpp接口	Atlas 推理系列产品（Ascend 310P处理器） hi_mpi接口	Atlas 200/300/500 推理产品->Atlas 推理系列产品（Ascend 310P处理器）迁移时，对用户的影响
无需进行媒体数据处理模块去初始化	hi_mpi_sys_exit	用户需要修改代码，增加调用媒体数据处理模块去初始化的接口hi_mpi_sys_exit。

