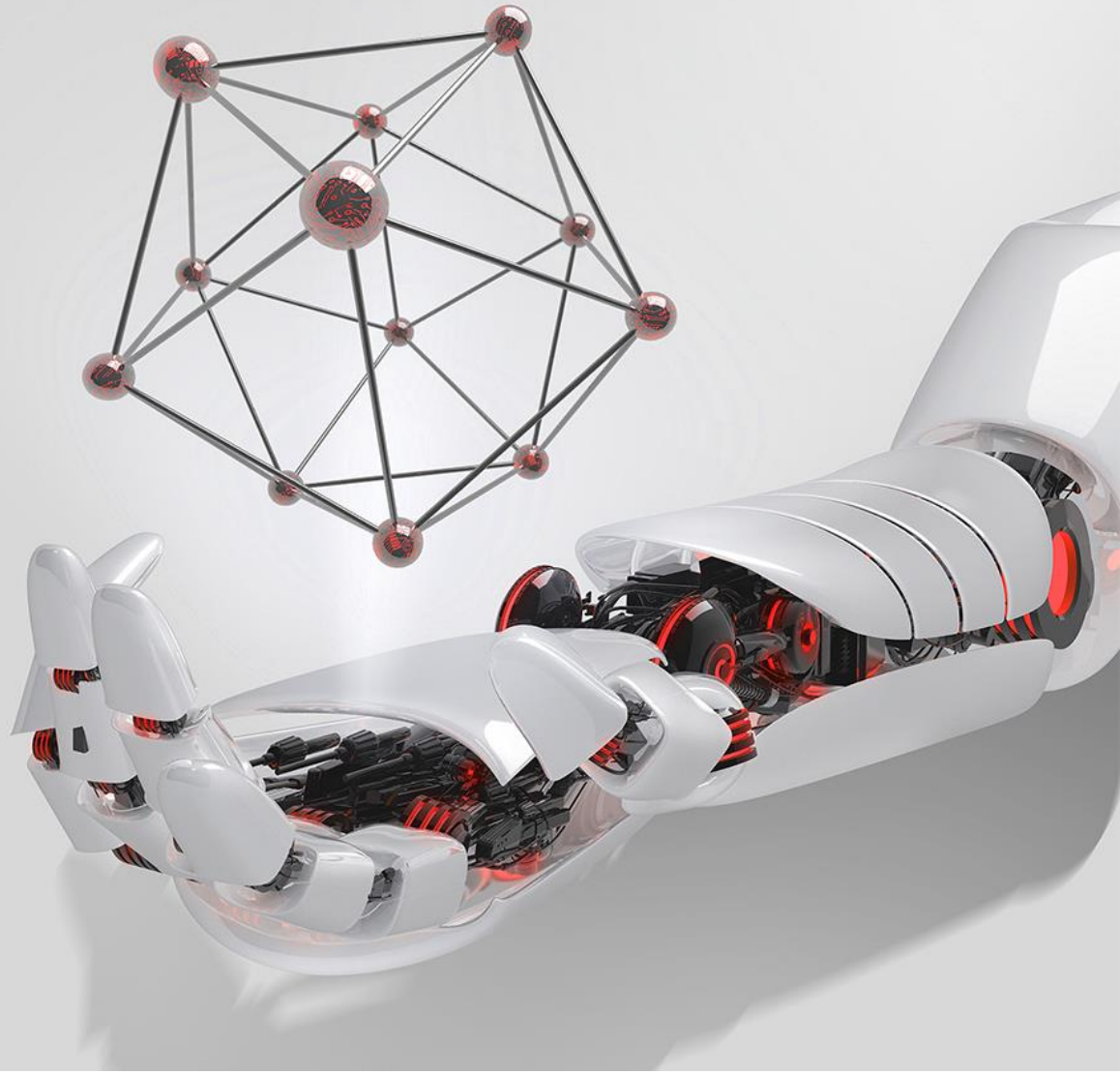
华为昇腾AI处理器及应用初识课程



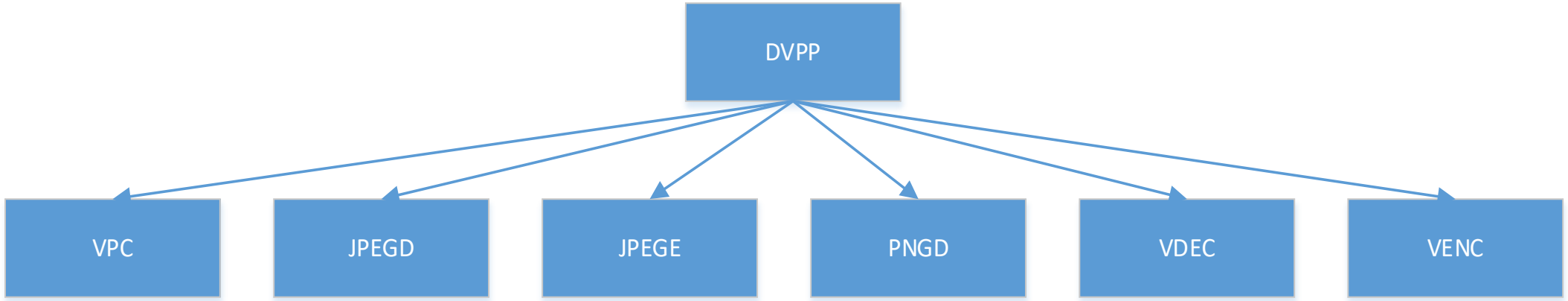
Chapter 4：编程指南

HUAWEI

目录

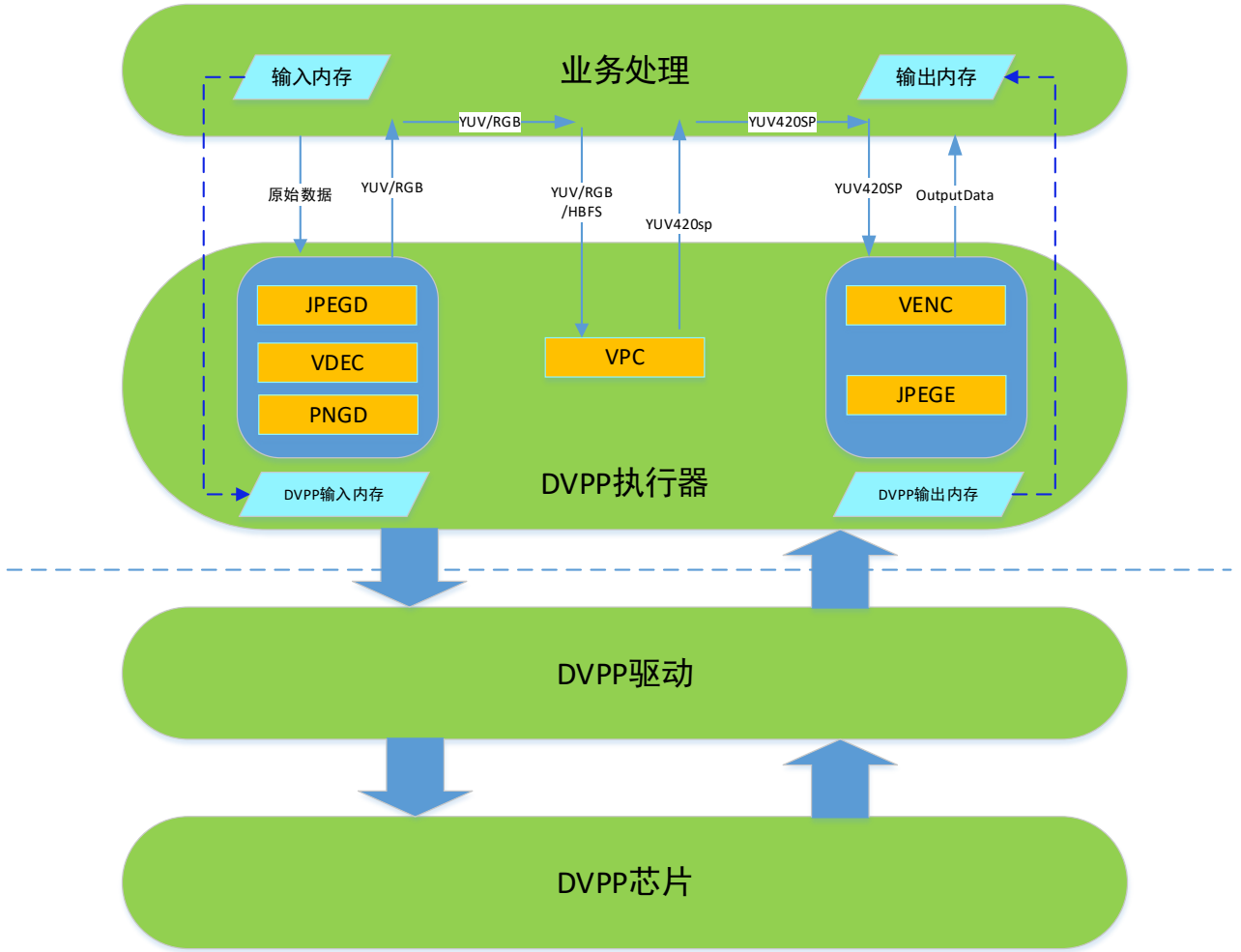
DVPP —— 架构概述

DVPP(Digital Video Pre-Processor)是数字视觉预处理，可以做图片、视频的编码、解码，图片的缩放、抠图、格式转换等功能。主要实现功能如下图：



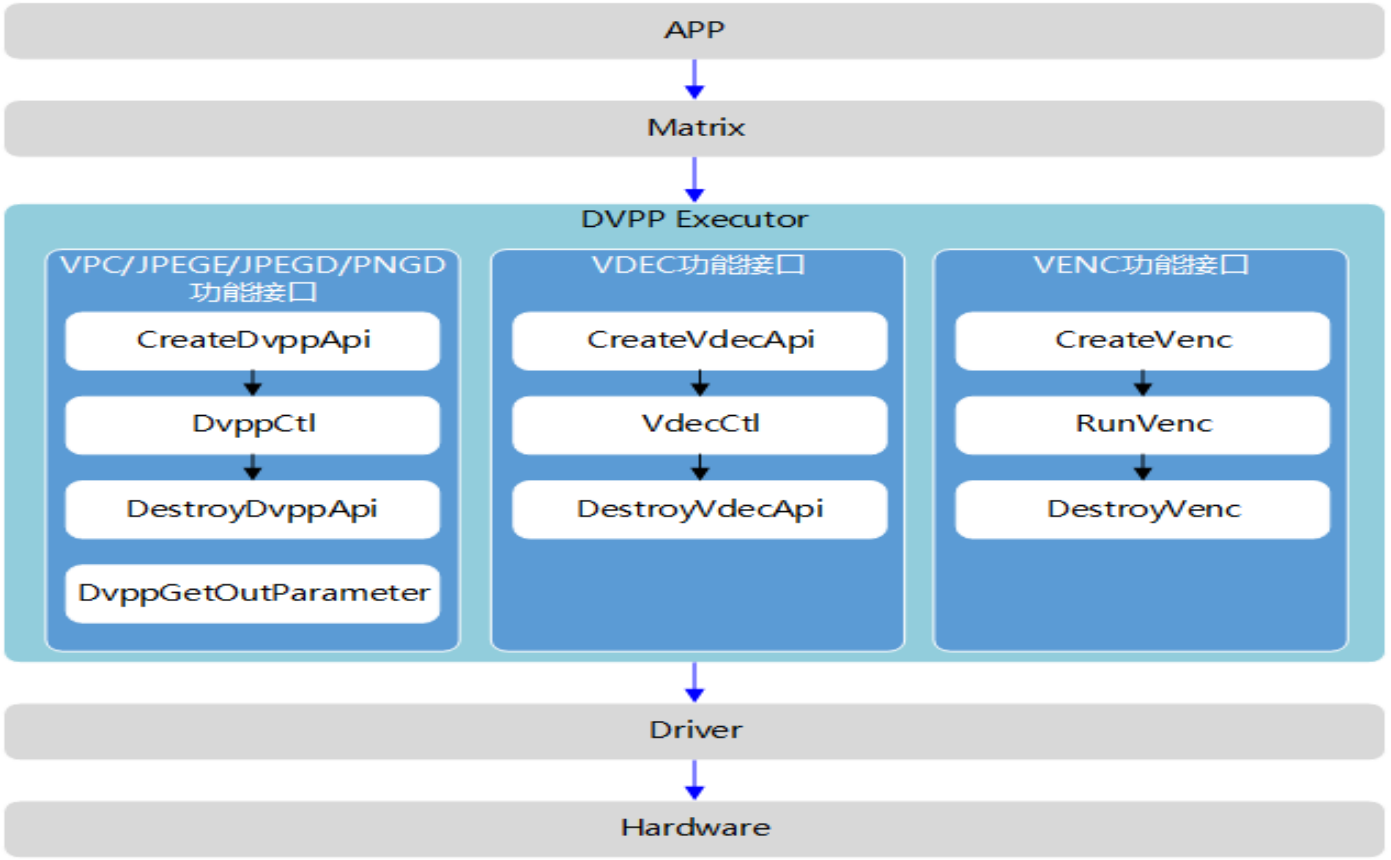
缩略语	英文全名	中文名称
DVPP	Digital Vision Pre-Process	数字视觉预处理
VPC	Vision Preprocess Core	视觉预处理模块
JPEGD	JPEG Decode	JPEG图片解码
JPEGE	JPEG Encoder	JPEG图片编码
PNGD	Portable Network Graphics Decoder	便携式网络图像格式解码
VDEC	Video Decoder	视频解码
VENC	Video Encoder	视频编码

DVPP —— 架构概述



DVPP 功能结构图

DVPP —— 接口总览



DVPP 接口总览图

DVPP —— 接口介绍

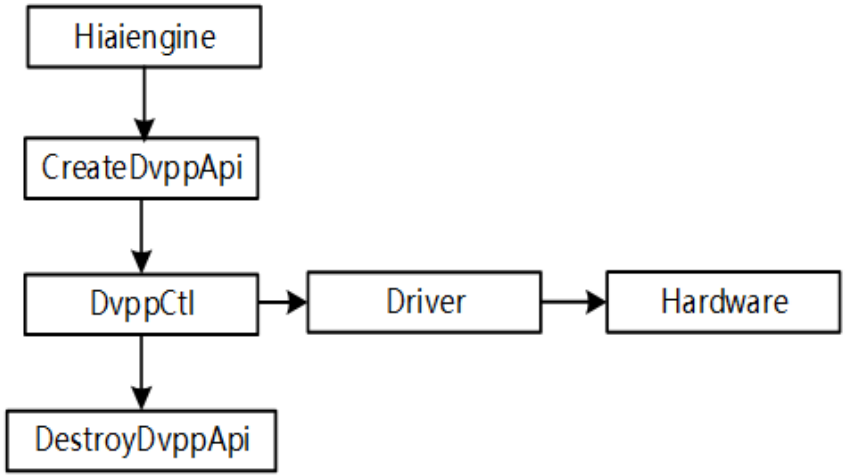
一、DVPP的VPC/JPEGD/JPEGE/PNGD模块对外提供三个接口

CreateDvppApi：负责创建DVPP API（Application Programming Interface）实例，DVPP API实例类似于打开驱动后返回的文件描述符。

DvppCtl：为控制DVPP各模块执行，主要调用各个模块暴露的Process接口函数，模块主要包括VPC（Vision Pre-processing Core）、JPEGD、JPEGE、PNGD等，

DestroyDvppApi：为销毁DVPP API

值得注意的是,一旦调用DestroyDvppApi函数,如果还想在继续调用DVPP,需要重新获取DvppApi实例。下图是hiaiengine（Matrix）调用dvpp的实现流程：



DVPP —— 接口介绍

二、用户管理内存新接口 DvppGetOutParameter

为JPEGD/JPEGE/PNGD三个模块获取输出内存大小的的接口

三、VDEC模块接口

CreateVdecApi：获取vdecapi实例，相当于vdec执行器句柄。
VdecCtl：控制DVPP执行器进行视频解码。
DestroyVdecApi：释放vdecapi，关闭VDEC执行器。

四、 VENC 模块

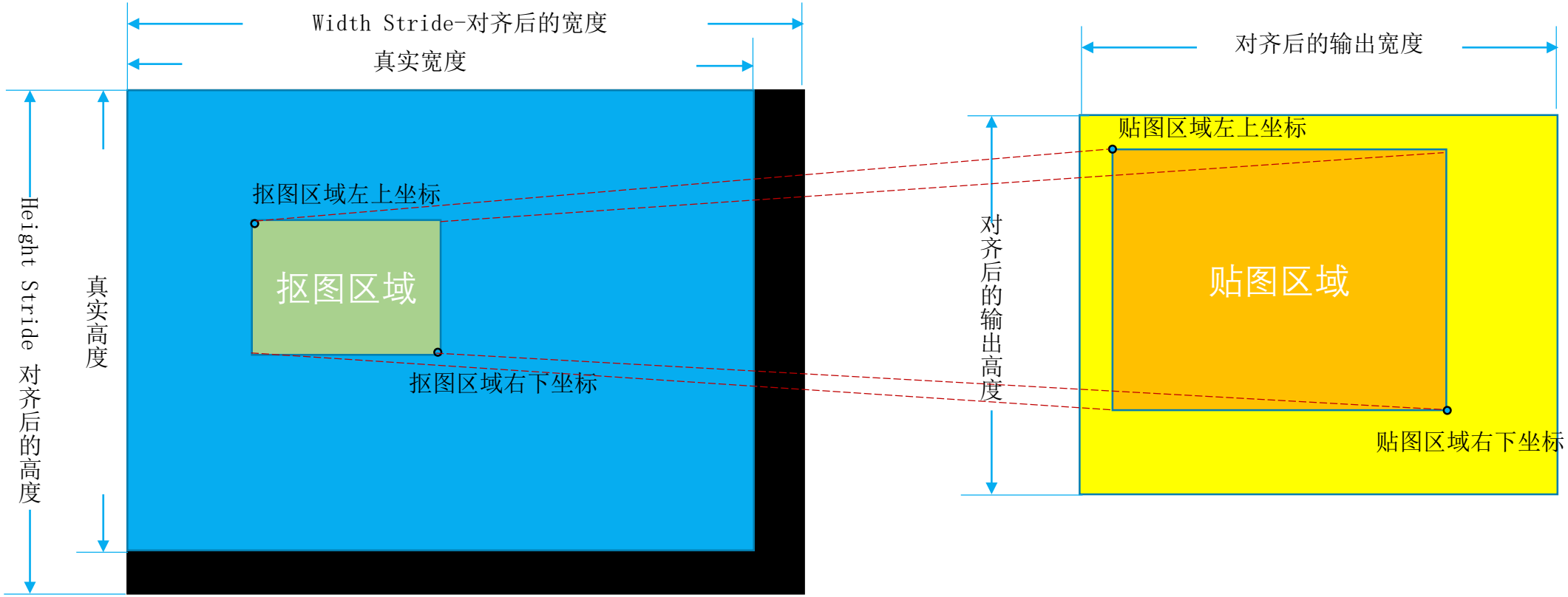
CreateVenc：获取VENC编码实例，相当于VENC执行器句柄。
RunVenc：控制DVPP执行器进行视频编码。
DestroyVenc：释放VENC编码实例，关闭VENC执行器。

DVPP ——功能模块详细描述

模块名字	支持功能
VPC	抠图、缩放、叠加、拼接 、格式转换，ＨＢＦＳ解压缩
JPEGD	JPG图片解码
JPEGE	JPG图片编码
PNGD	便携式网络图像格式解码
VDEC	h264/h265视频解码
VENC	h264/h265视频编码

DVPP —— 典型模块讲解（VPC）

VPC常用图像处理Crop+Resize:



DVPP ——典型模块讲解（VDEC+VPC）

VDEC主要业务流程和功能

VDEC

HFBC

VPC

推理

H264、H265码流

YUV图片

识别图片中的对象

VDEC支持两种输入格式：

H264 bp/mp/hp level5.1 yuv420sp编码的码流。

H265 8/10bit level5.1 yuv420sp编码的码流。

VDEC输出格式为：yuv420sp压缩后的HFBC数据。

注意：若码流中有坏帧、缺帧等情况，解码器VDEC可能会丢帧

DVPP ——业务代码实现

1. 针对六大类功能，一共有十几个sample样例供参考；

2. Sample源码可以通过下载和安装《Ascend_Sample-1.XX.TXX.BXXX.zip》包，从sample_dvpp.cpp获得；

3. 编译和执行参考 生态网站的《DVPP API 参考》；

目录

OMG —— 概述

Atlas200DK平台中提供了模型转换工具（OMG），可以将Caffe、Tensorflow等开源框架模型转换成Atlas200DK支持的模型，从而能够更方便快捷地把其他平台的模型放到Atlas200DK平台进行调试并拓展相关业务。

离线模型转换(OMG)

支持训练好的模型可以脱离训练框架直接在设备上执行

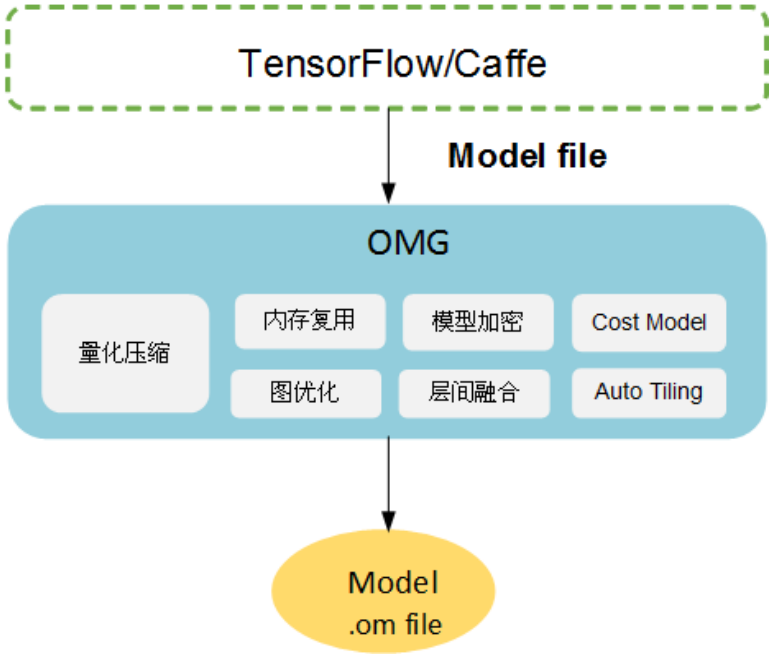
- **离线模型转换：**
  将开源的caffe&TF模型转换为Davinci模型，然后用D芯片进行推理
- **支持量化：**
  将把高精度数据进行低Bit（int8）量化，可以缩小模型的大小，提升网络推理性能。
- **支持自定义算子：**
  支持用户实现自己的算子，从而支撑整网在NPU上的完整运行。
- **AIPP：**
  预处理模块（AIPP），支持通道转换，色与转换，抠图等处理操作
- **高精度/高性能模型：**
  支持高精度模式/高性模式，根据场景需要进行使能
- **模型加密/解密：**
  支持对Davinci模型的加密与解密操作，确保私密性

离线模型执行(OME)

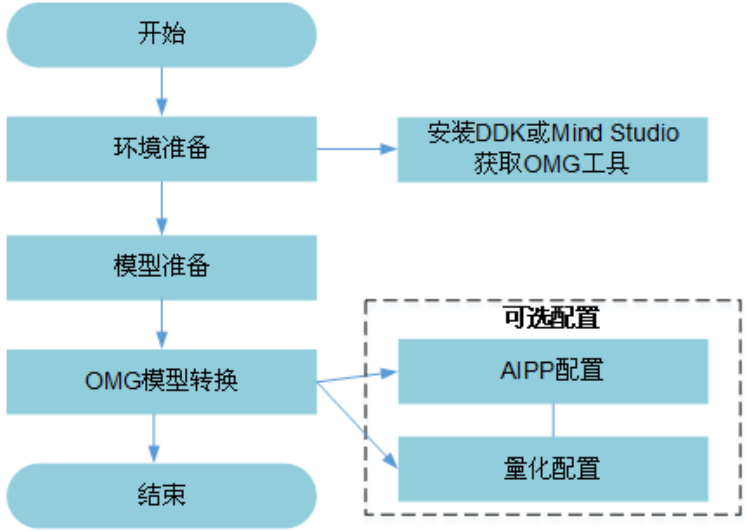
支持主流NN框架能够在DaVinci上进行推理

- **离线模型加载卸载：**
  便利的将模型进行动态加载，从而使得业务更具灵活性
- **离线模型推理**
  调用模型管家对离线模型进行推理，也可以对多模型进行管理

OMG —— 功能架构与运行流程



OMG工具功能架构

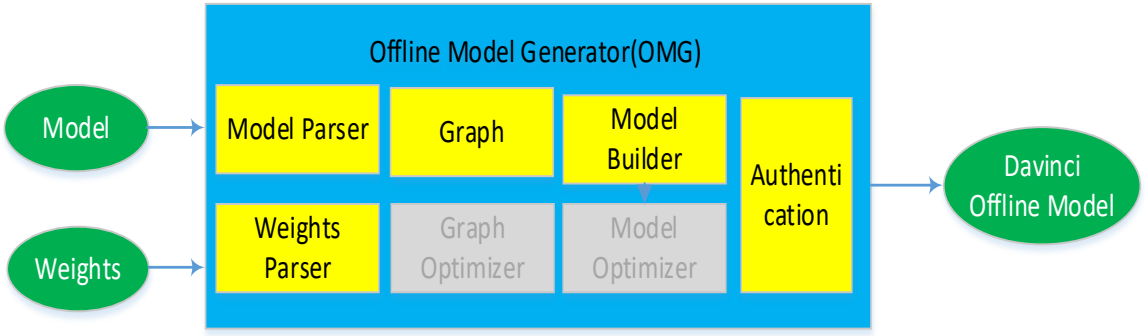


模型转换总体流程

1. 使用OMG工具之前，需先安装DDK或Mind Studio；
2. 准备要进行转换的模型，并上传到DDK所在服务器；
3. 使用OMG工具进行模型转换，在配置相关参数时，根据实际情况选择是否进行AIPP配置或量化配置；
   a. AIPP是昇腾AI处理器提供的硬件图像预处理模块，包括色域转换，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）等功能。DVPP模块输出的图片多为对齐后的YUV420SP类型，不支持输出RGB图片。因此，业务流需要使用AIPP模块转换对齐后YUV420SP类型图片的格式，并抠出模型需要的输入图片。
   b. 量化是指对高精度数据进行低Bit量化，在对模型大小和性能有更高要求的时候可以选择执行量化操作。模型转换过程中量化会将高精度数据向低比特数据进行量化，让最终生成的模型更加轻量化，从而达到节约网络存储空间、降低传输时延以及提高运算执行效率的目的。

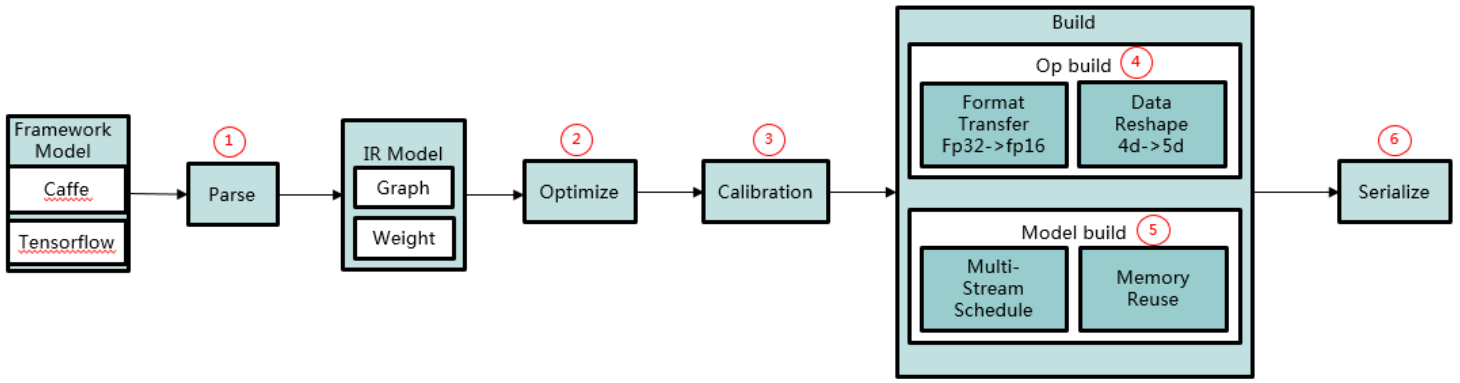
OMG —— OMG转换能力

开源框架	输入文件	存储格式
Caffe	*.prototxt，网络结构文件	protobuf格式
	*.caffemodel，权值文件	二进制文件
~~Caffe2~~	predict_net.pb网络定义文件	protobuf格式
	init_net.pb 网络的所有输入数据，包括所有权值数据和首个Op的输入数据描述。	protobuf格式
~~Mxnet~~	xx-symbol.json，网络结构文件	json格式
	xx.params，权值文件，主要包括权值节点的数据、数据类型、数据格式等信息。	二进制文件
Tensorsfolw	*.pb，网络模型和权值数据在一个文件中	protobuf格式



- **离线模型生成器，主要负责适配各种框架下的不同网络文件和权值文件，统一转化Davinci格式的离线文件，供OME使用。**
- **OMG主要在Host（Linux Ubuntu）上单独离线执行，以OMG可执行程序的方式提供给IDE，命令行方式调用。**

OMG —— OMG转换流程



Log:

Model parsing is complete. (1/5)

Weight parsing is complete. (2/5)

Graph optimization is complete. (3/5)

Model building is complete. (4/5)

Offline model saving is complete. (5/5)

OMG generate offline model success.

① **解析parse**：支持解析不同开源框架（Caffe,TensorFlow）生成的原始模型,并将网络、权值为统一的 Inner Graph定义,这一步出错的原因通常是算子不能支持，可以通过查看生成的jason文件知道哪些算子不支持解析。

② **优化Optimize**：针对得到的中间表示层的IR模型，继续做一些图形的优化，主要是计算图的融合，识别可以融合的算子，将多个计算层融合为一层。实现计算的加速。

③ **量化calibration**：优化后的结构，通过量化工具进行Int8量化，得到量化后的权值及scale参数、输入输出的scale/offset参数；量化为可选过程，对于模型大小和性能有更高要求的时候可以选择执行，量化会对模型的输出精度有影响。

④ **算子Build**：生成算子的离线结构表示，这个过程中，会对算子使用的权值进行数据格式、形式的转换，计算每个算子的输出维度、内存大小等信息。

⑤ **模型Build**：生成模型的离线结构表示，主要是对graph中的算子进行并发调度分析，生成算子的执行序列，并基于执行序列进行内存的复用优化，将相关信息写入模型和算子描述中。

⑥ **序列化**：将离线模型从内存输出到文件，序列化过程会对离线模型进行完整性保护，提供文件签名及加密功能。

OMG —— 模型转换的运行依赖库

```
root@kickseed:/home/mindtool/tools/che/ddk/ddk/uihost/bin# ldd omg
        linux-vdso.so.1 =>  (0x00007ffd71bdf000)
        libachk.so => /lib/libachk.so (0x00007f3ce9a5f000)
        libfmk_common.so => /usr/local/HiAI/runtime/lib64/libfmk_common.so (0x00007f3ce8f20000)
        libgflags.so.2.2 => /usr/local/HiAI/runtime/lib64/libgflags.so.2.2 (0x00007f3ce8d00000)
        libprotobuf.so.15 => /usr/local/HiAI/driver/lib64/libprotobuf.so.15 (0x00007f3ce881f000)
        libslog.so => /usr/local/HiAI/driver/lib64/libslog.so (0x00007f3ce8614000)
        libomg.so => /usr/local/HiAI/runtime/lib64/libomg.so (0x00007f3ce6b42000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f3ce693e000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f3ce65bc000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f3ce63a6000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3ce5fdc000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f3ce5dbf000)
        libcrypto.so.1.1 => /usr/local/HiAI/driver/lib64/libcrypto.so.1.1 (0x00007f3ce5940000)
        libc_sec.so => /usr/local/HiAI/driver/lib64/libc_sec.so (0x00007f3ce572e000)
        libmmpa.so => /usr/local/HiAI/driver/lib64/libmmpa.so (0x00007f3ce5522000)
        libcce.so => /usr/local/HiAI/runtime/lib64/libcce.so (0x00007f3ce4b9a000)
        libmodel_encrypt_tool.so => /usr/local/HiAI/driver/lib64/libmodel_encrypt_tool.so (0x00007f3ce494b000)
        libgraph.so => /usr/local/HiAI/runtime/lib64/libgraph.so (0x00007f3ce448d000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f3ce4285000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f3ce3f7c000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3ce9951000)
        libfusionengine.so => /usr/local/HiAI/runtime/lib64/libfusionengine.so (0x00007f3ce2dfc000)
        libcalibration.so => /usr/local/HiAI/runtime/lib64/libcalibration.so (0x00007f3ce24d9000)
        libruntime.so => /usr/local/HiAI/runtime/lib64/libruntime.so (0x00007f3ce224a000)
        libcce_aicpudev_online.so => /usr/local/HiAI/runtime/lib64/libcce_aicpudev_online.so (0x00007f3ce1efa000)
        libcce_aicore.so => /usr/local/HiAI/runtime/lib64/libcce_aicore.so (0x00007f3ce15d2000)
        libtiling.so => /usr/local/HiAI/runtime/lib64/libtiling.so (0x00007f3ce11f4000)
        libopencv_world.so.3.4 => /usr/local/HiAI/runtime/lib64/libopencv_world.so.3.4 (0x00007f3cdfbc7000)
        libcce_tools.so => /usr/local/HiAI/runtime/lib64/libcce_tools.so (0x00007f3cdf8b6000)
        libgomp.so.1 => /usr/lib/x86_64-linux-gnu/libgomp.so.1 (0x00007f3cdf694000)
        libdrvdevdrv.so => /usr/local/HiAI/driver/lib64/libdrvdevdrv.so (0x00007f3cdf46e000)
```

开源和内部实现库

- libfmk_common.so fmk通用库
- libprotobuf：开源的解析模型文件的库
- libslog.so 日志工具库
- libomg.so 模型转换依赖的so库
- libcce.so D芯片cce加速库
- libmodel_encrypt_tool.so 模型加密库
- libgraph.so 图处理so
- libfusionengine.so 融合相关so
- libcalibration.so 量化so
- libruntime.so runtime相关so
- libcce_aicpudevonline.so aicpu运行上的so
- libcce_aicore.so 运行在aicore上的so
- libopencv_world.so 第三方opencv so
- libdrvdevdrv.so 驱动相关so

OMG所需依赖库都会根据版本统一做升级，不需要用户手动升级，以确保so库一致性

OMG —— 命令使用及参数介绍

样例命令：

./omg --model=./alexnet.prototxt --weight=./alexnet.caffemodel --framework=0 --output=./domi

./omg --model=$HOME/ResNet18_deploy.prototxt --weight=
$HOME/ResNet18_model.caffemodel --framework=0 --output=$HOME/outdata/ResNet18 --
input_shape= “data:1 ,3, 224, 224” –-insert_op_conf= ./aipp_con.cfg

```
omg: usage: ./omg <args>
example:
./omg --model=./alexnet.prototxt --weight=./alexnet.caffemodel
--framework=0 --output=./domi
aguments explain:
  --model             Model file
  --weight            Weight file. Required when framework is Caffe
  --framework         Framework type(0:Caffe; 3:Tensorflow)
  --output            Output file path&name(needn't suffix, will add .om automatically)
  --encrypt_mode      Encrypt flag. 0: encrypt; -1(default): not encrypt
  --encrypt_key       Encrypt_key file
  --certificate       Certificate file
  --hardware_key      ISV file
  --private_key       Private key file
  --input_shape       Shape of input data. E.g.: "input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"
  --h/help            Show this help message
  --cal_conf          Calibration config file
  --insert_op_conf    Config file to insert new op
  --op_name_map       Custom op name mapping file
  --plugin_path       Custom op plugin path. Default value is: "./plugin". E.g.: "path1;path2;path3".
                      Note: A semicolon(;) cannot be included in each path, otherwise the resolved path will not match the expected one.
  --om                The model file to be converted to json
  --json              The output json file path&name which is converted from a model
  --mode              Run mode. 0(default): model => davinci; 1: framework/davinci model => json; 3: only pre-check
  --target            Target platform. (mini)
  --out_nodes         Output nodes designated by users. E.g.: "node_name1:0;node_name1:1;node_name2:0"
  --input_format      Format of input data. E.g.: "NCHW"
  --check_report      The pre-checking report file. Default value is: "check_result.json"
  --input_fp16_nodes  Input node datatype is fp16 and format is NCHW. E.g.: "node_name1;node_name2"
  --is_output_fp16    Net output node datatype is fp16 and format is NCHW, or not. E.g.: "false,true,false,true"
  --ddk_version       The ddk version. E.g.: "x.y.z.Patch.B350"
  --net_format        Set net prior format. ND: select op's ND format preferentially; 5D: select op's 5D format preferentially
  --output_type       Set net output type. Support FP32 and UINT8
  --fp16_high_prec    FP16 high precision. 0(default): not use fp16 high precision; 1: use fp16 high precision
```

常用使用参数：

--model	原始模型文件路径
--weight	权重文件路径
--framework	0：caffe 3：tensorflow
--output	存放转换后的离线模型文件
--plugin_path	自定义算子插件路径
--insert_op_conf	输入预处理数据的配置文件路径
--input_shape	模型输入数据的shape。
--fp16_high_prec	是否生成高精度FP16模型
--mode	0：生成davinci模型 1：模型转json 3：仅做预检

其余不常用的参数可根据场景具体查看Atlas200DK模型转换指导

OMG —— AIPP介绍

常用AIPP实例：

```
aipp_op {
aipp_mode : static
related_input_rank : 0
input_format YUV420SP_U8
src_image_size_w : 640
src_image_size_h : 608
crop : true
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 256
matrix_r0c1 : 0
matrix_r0c2 : 359
matrix_r1c0 : 256
matrix_r1c1 : -88
matrix_r1c2 : -183
matrix_r2c0 : 256
matrix_r2c1 : 454
matrix_r2c2 : 0
input_bias_0 : 0
input_bias_1 : 128
input_bias_2 : 128
}
```

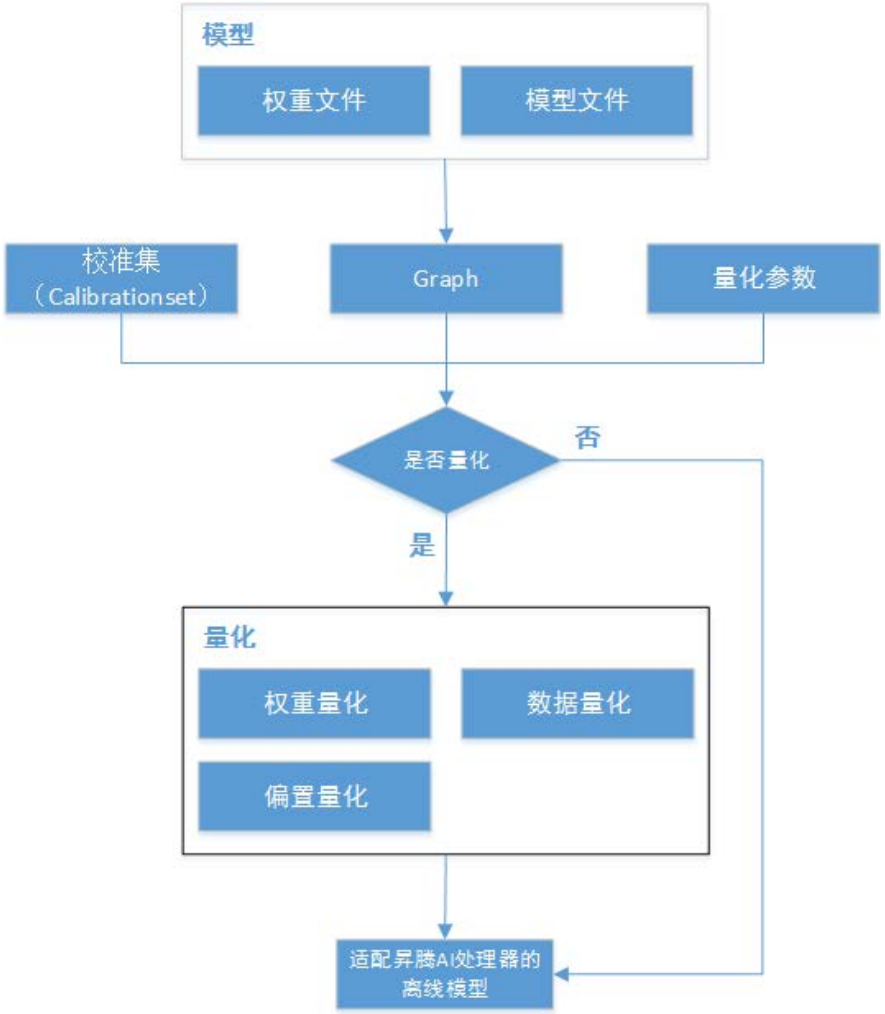
AIPP（AI Preprocessing）：用于在AI Core上完成图像预处理，包括色域转换（转换图像格式）、图像归一化（减均值/乘系数）和抠图（指定抠图起始点，抠出神经网络需要大小的图片）

AIPP支持的图像输入格式：YUV420SP_U8、XRGB8888_U8、RGB888_U8、YUV400_U8

参数名	参数解析
aipp_mode	指定了AIPP的模式，必须配置。 dynamic 表示动态AIPP，static 表示静态AIPP
related_input_rank	参数为可选，标识对模型的第几个输入做AIPP处理，从0开始，默认为0。 例如模型有两个输入，需要对第2个输入做AIPP，则配置related_input_rank为1。
input_format	输入图像类型：YUV420SP_U8/XRGB8888_U8/RGB888_U8/YUV400_U8
src_image_size_w/h	进入aipp实际图片的宽、高配置(需要考虑dvpp对齐)，不设置或同时设置为0，则会取网络输入定义的w和h
crop	AIPP处理图片时是否支持抠图
csc_switch	色域转换开关，静态AIPP配置
rbuv_swap_switch	色域转换前，R通道与B通道交换开关/U通道与V通道交换开关
matrix_rXcX	若色域转换开关为false，则本功能旁路 具体数据转换
input_bias_x	YUV转RGB时的输入偏移
output_bias_x	RGB转YUV时的输出偏移

其余不常用的参数可根据场景具体查看Atlas200DK模型转换指导

OMG —— 量化



量化处理流程

量化是指对高精度数据进行低Bit量化，从而达到节约网络存储空间、降低传输时延以及提高运算执行效率的目的。当前支持卷积算子（Convolution）、全连接算子（FullConnection）以及深度可分离卷积（ConvolutionDepthwise三种类型算子的量化，包括权重、偏置、数据量化。量化模式分为：无offset、数据offset。

常用使用参数：

参数	作用	取值范围
device	推理模式	USE_CPU
quantize_algo	量化模式	NON_OFFSET HALF_OFFSET
weight_type	权重量化模式	VECTOR_TYPE SCALAR_TYPE
preprocess_parameter	预处理及输入相关参数	无
input_type	输入数据类型	IMAGE BINARY
input_file_path	输入数据地址	无
image_format	图像输入数据三通道排序方式	BGR RGB
mean_value	图像预处理的均值	[0, 255.0]
standard_deviation	图像预处理的均方差	[0,FLOAT_MAX]

其余不常用的参数可根据场景具体查看Atlas200DK模型转换指导

目录

Matrix —— 基本功能介绍

简介：

Matrix为通用业务流程执行引擎，运行于操作系统之上，业务应用之下。Matrix可以屏蔽操作系统差异，为应用提供统一的标准化接口，包括流程编排接口（支持C/C++语言、Python语言）和模型管家接口（支持C++语言）

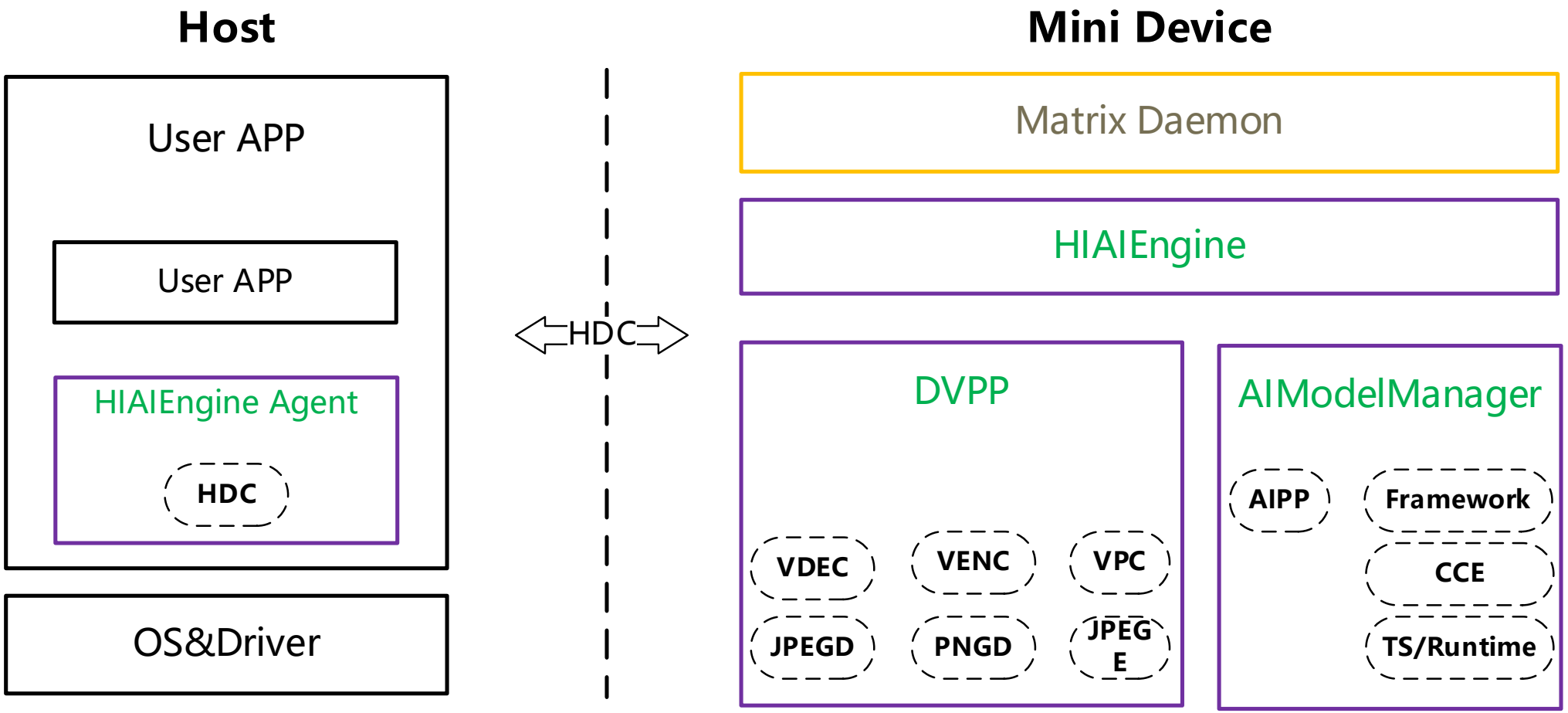
基本概念：

- Host指与Device相连接的X86服务器、ARM服务器或者Windows PC，会利用Device提供的NN（Neural-Network ）计算能力，完成业务。
- Device指安装了Ascend 310芯片的硬件设备，利用PCIe接口与Host侧连接，为Host提供NN计算能力。
- DVPP（Digital Vision Pre-Processing）：主要实现视频解码、视频编码、JPEG编解码、PNG解码、视觉预处理。
- Framework是深度学习框架，可以将caffe、tensorflow等开源框架模型转换成昇腾AI芯片支持的离线模型。

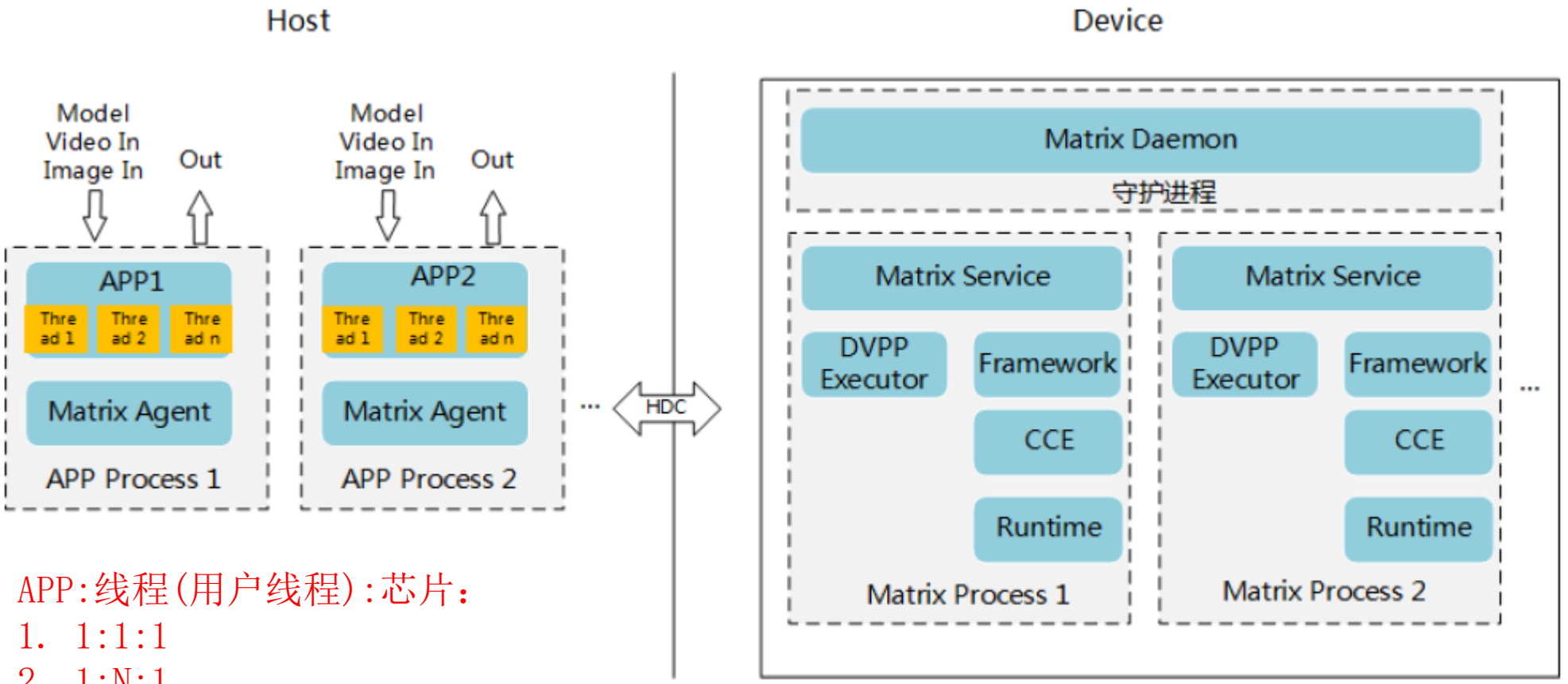
Matrix的组成：

- Matrix Agent：运行在Host侧，其功能如下。
  - ✓ 完成与Host APP进行控制命令和处理数据的交互。
  - ✓ 完成与Device间的IPC（InterProcess Communication）通信。
- Matrix Daemon：运行在Device侧，其功能如下。
  - ✓ 根据配置文件完成业务流程的建立。
  - ✓ 根据命令完成业务流程的销毁及资源回收。
  - ✓ 守护进程，负责拉起Matrix Service进程。
- Matrix Service：运行在Device侧，其功能如下。
  - ✓ 调用DVPP的API接口实现媒体预处理。
  - ✓ 调用模型管家（AIModelManager）的API接口实现模型推理。

Matrix —— 架构概述



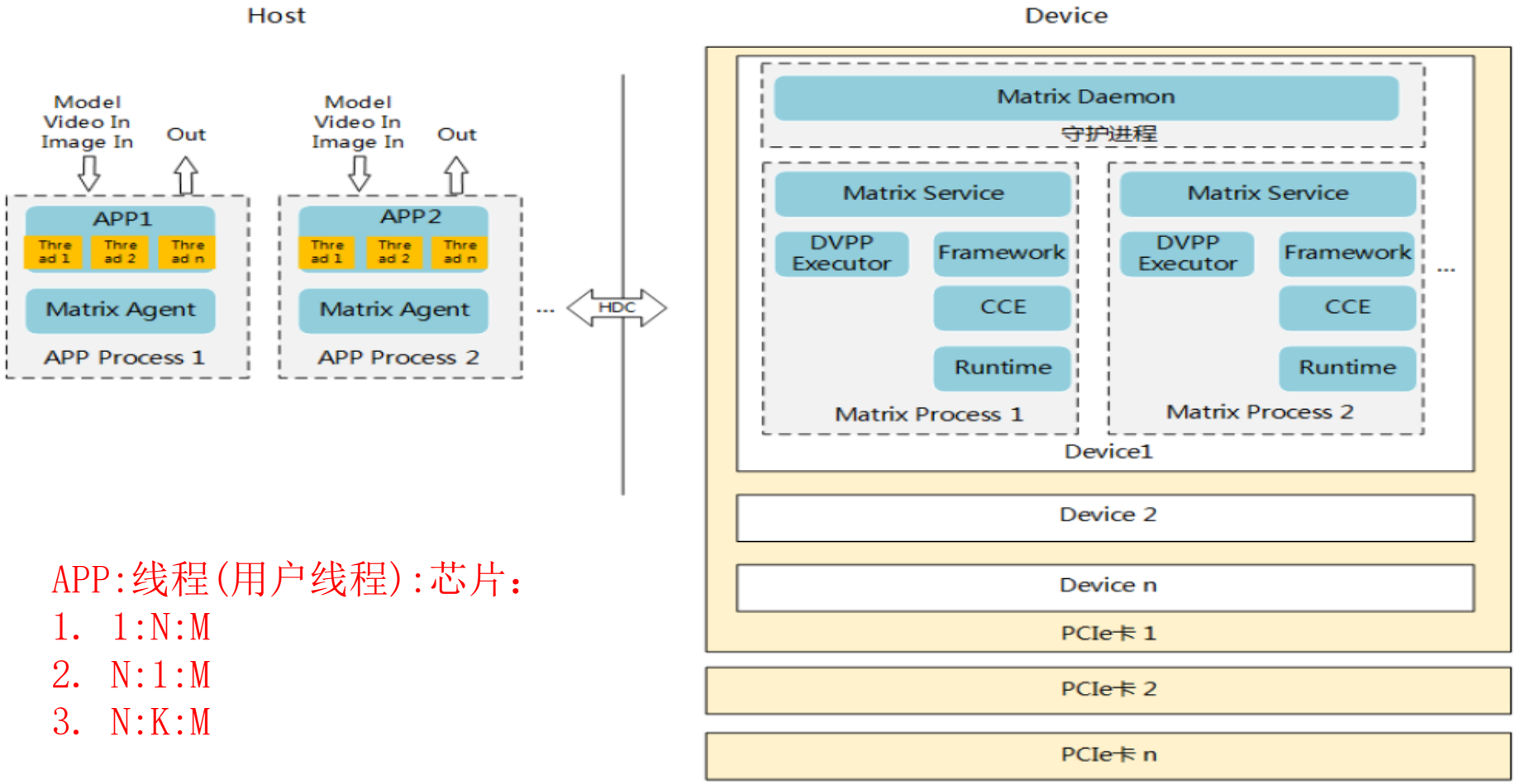
Matrix —— 架构（单Device场景）



APP:线程(用户线程):芯片：

1. 1:1:1
2. 1:N:1
3. N:1:1
4. N:N:1

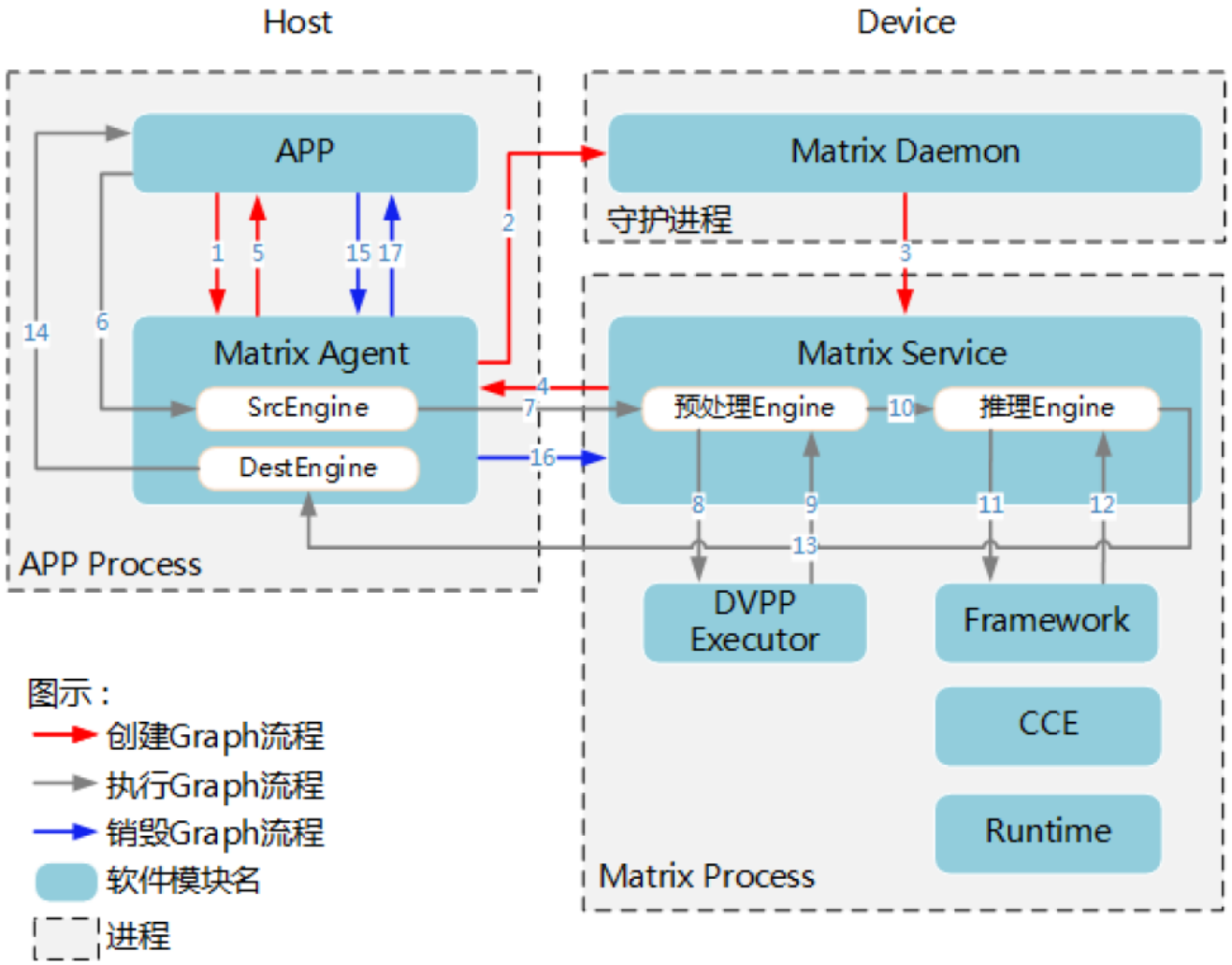
Matrix —— 架构（多Device场景）



APP:线程(用户线程):芯片：

1. 1:N:M
2. N:1:M
3. N:K:M

Matrix —— 典型流程



流程	序号	流程简介
创建Graph	1-5	● 根据Graph配置创建Graph对象。 ● 上传离线模型文件和配置文件到Device侧。 ● 初始化Engine，在此过程中推理Engine通过离线模型管家（AIModelManager）的Init接口加载模型。
Engine执行	6	输入数据。
	7-9	预处理Engine调用DVPP的API接口，进行数据预处理，例如对视频/图像进行编解码、抠图、缩放等。
	10-12	推理Engine调用离线模型管家（AIModelManager）的Process接口进行推理计算。
	13-14	推理Engine 调用Matrix提供的SendData接口将推理结果返回给DestEngine。DestEngie通过回调函数将推理结果返回给APP。
销毁Graph	15-17	结束程序，销毁Graph对象。

```
root@ubuntu:/usr/local/HiAI# cd source/
root@ubuntu:/usr/local/HiAI/source# ls
install.sh  readme.txt  source.tar.gz
root@ubuntu:/usr/local/HiAI/source#
```

注：Host侧的驱动代码和框架代码已经开源，在安装run包的300标卡环境中/usr/local/HiAI/source目录下并提供源码编译指导

Matrix —— 接口介绍

- **流程编排接口：**
  - Engine：可用于流程编排的独立功能单元
    ：Engine面向用户编程，对应一个线程
  - Graph：管理若干Engine 组成的流程
    ：Graph面向用户配置，对应一个进程
  - Graph配置文件
    1. Graph配置信息
    2. Engine配置信息
    3. Engine连接关系

- **单独API：**
  - 离线模型框架AIModelManger API
    - ✓ 离线模型转换：OMG
    - ✓ 离线模型执行：加载和运行
      - ✓ AIModelManger API
  - DVPP API
    - ✓ 视频和图片预处理功能

```
graphs {
  graph_id: 100
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    so_name: "/lib64/libhelloworld.so"
    side: DEVICE
    thread_num: 1
  }
  engines {
    id: 1002
    engine_name: "DestEngine"
    side: HOST
    thread_num: 1
  }
  connects {
    src_engine_id: 1000
    src_port_id: 0
    target_engine_id: 1001
    target_port_id: 0
  }
  connects {
    src_engine_id: 1001
    src_port_id: 0
    target_engine_id: 1002
    target_port_id: 0
  }
}
```

```
graphs {
  graph_id: 123456
  priority: 0
  device_id: "0"

  engines {
    id: 100
    engine_name: "InputFile"
    side: HOST
    thread_num: 1
    so_name: "./libInputFile.so"
  }

  engines {
    id: 123
    engine_name: "DeviceProcess"
    side: DEVICE
    thread_num: 1
    so_name: "./libDeviceProcess.so"
    ai_config {
      items {
        name: "model_path"
        value: "../resnet_jd_int8.om
      }

      items {
        name: "batch_size"
        value: "8"
      }

      items {
        name: "resize_width"
        value: "224"
      }

      items {
        name: "resize_height"
        value: "224"
      }

    }
  }
```

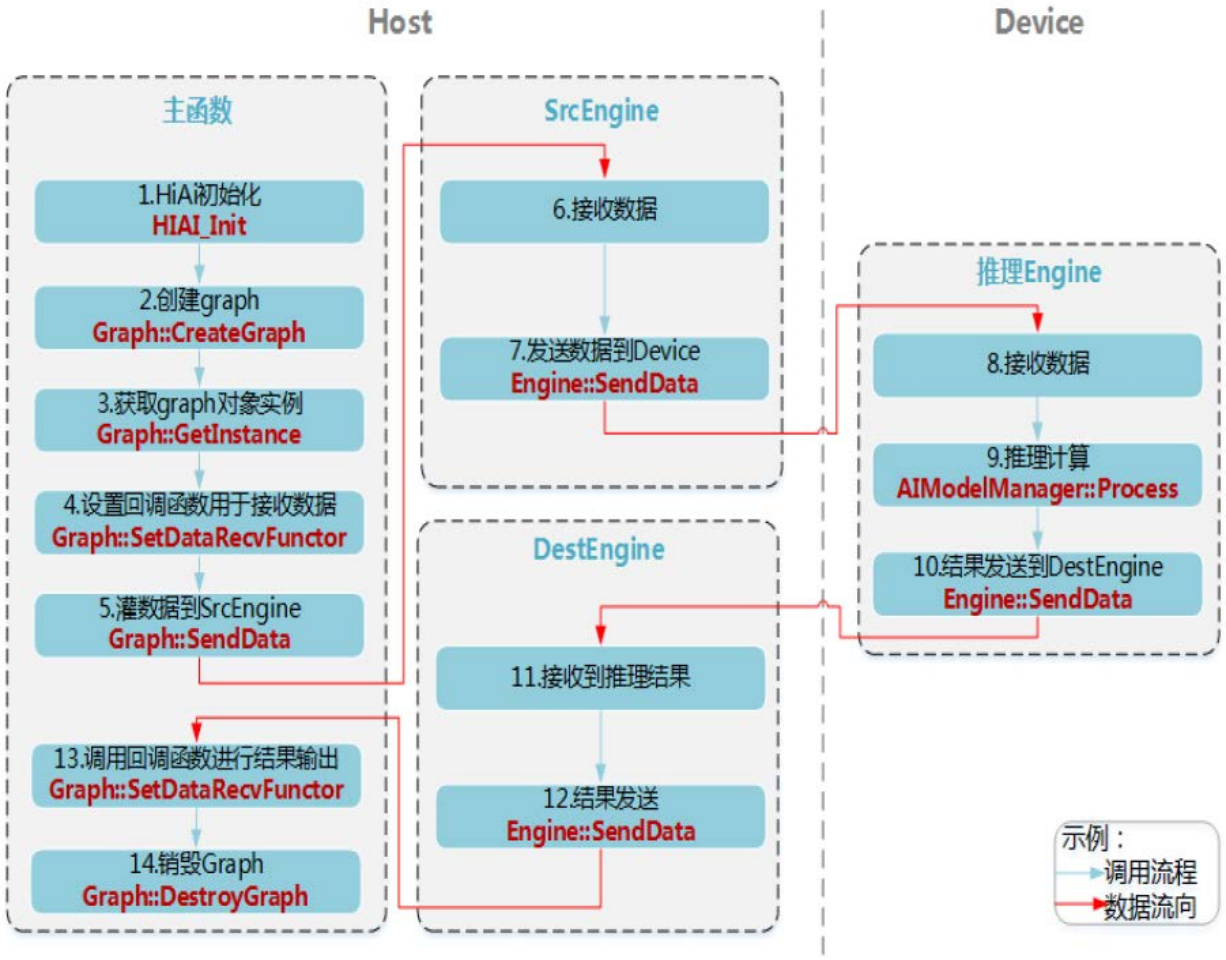
```
  engines {
    id: 789
    engine_name: "PostProcess"
    side: HOST
    thread_num: 1
    so_name: "./libPostProcess.so"
    ai_config {
    }
  }

  connects {
    src_engine_id: 100
    src_port_id: 0
    target_engine_id: 123
    target_port_id: 0
  }

  connects {
    src_engine_id: 123
    src_port_id: 0
    target_engine_id: 789
    target_port_id: 0
  }
}
```

指定Graph运行的Device

指定Engine的运行端侧

传递Device侧的文件路径名

Matrix —— Engine串联



序号	处理流程	说明
1	HiAi初始化。 `HIAI_Init();`	● 名称：主函数 ● 作用：创建Graph，并初始化Engine
2	通过CreateGraph接口创建Graph。 `hiai::Graph::CreateGraph(graph_config_proto_file);` hiai::Graph会调用hiai::Engine初始化Engine。	
3	获取Graph实例。 `std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(GRAPH_ID);`	
4	设置回调函数用于接收数据。 继承于DataRecvInterface的回调函数可以直接从Engine实现类的Port口获取数据，用于后处理engine. `graph->SetDataRecvFunctor(target_port_config, std::shared_ptr<DdkDataRecvInterface>( new DdkDataRecvInterface(test_dest_filename)));`	
5	调用Graph对象的发送函数Graph::SendData向Source Engine灌入数据。 `graph->SendData(engine_id, "string", std::static_pointer_cast<void>(src_data));`	

序号	处理流程	说明

6	通过参数arg0接收数据。
	`std::shared_ptr<Typename> input_arg = std::static_pointer_cast<Typename>(arg0);`
	`//可以通过此函数获取Graph::Send传进来的数据流。`
		● 名称：SrcEngine
		● 作用：读取数据

7	调用Engine的SendData发送函数，将数据发送到Device侧。
	`hiai::Engine::SendData(0, "Typename", std::static_pointer_cast<void>(input_arg));`
	`//可以通过此函数将数据流发送给需要传的Engine的port。`	

8	接收数据，通过参数arg0接收数据。 `std::shared_ptr<std::string> input_arg = std::static_pointer_cast<std::string>(arg0);`	● 名称：推理Engine ● 作用：模型推理
9	数据推理计算。 `ret = ai_model_manager_->Process(ai_context, input_data_vec, output_data_vec, 0);`	

10	将数据流发送给后处理engine.
	`hiai::Engine::SendData(0, "Typename", std::static_pointer_cast<void>(input_arg));`
	`//可以通过此函数将数据流发送给需要传的Engine的port。`	

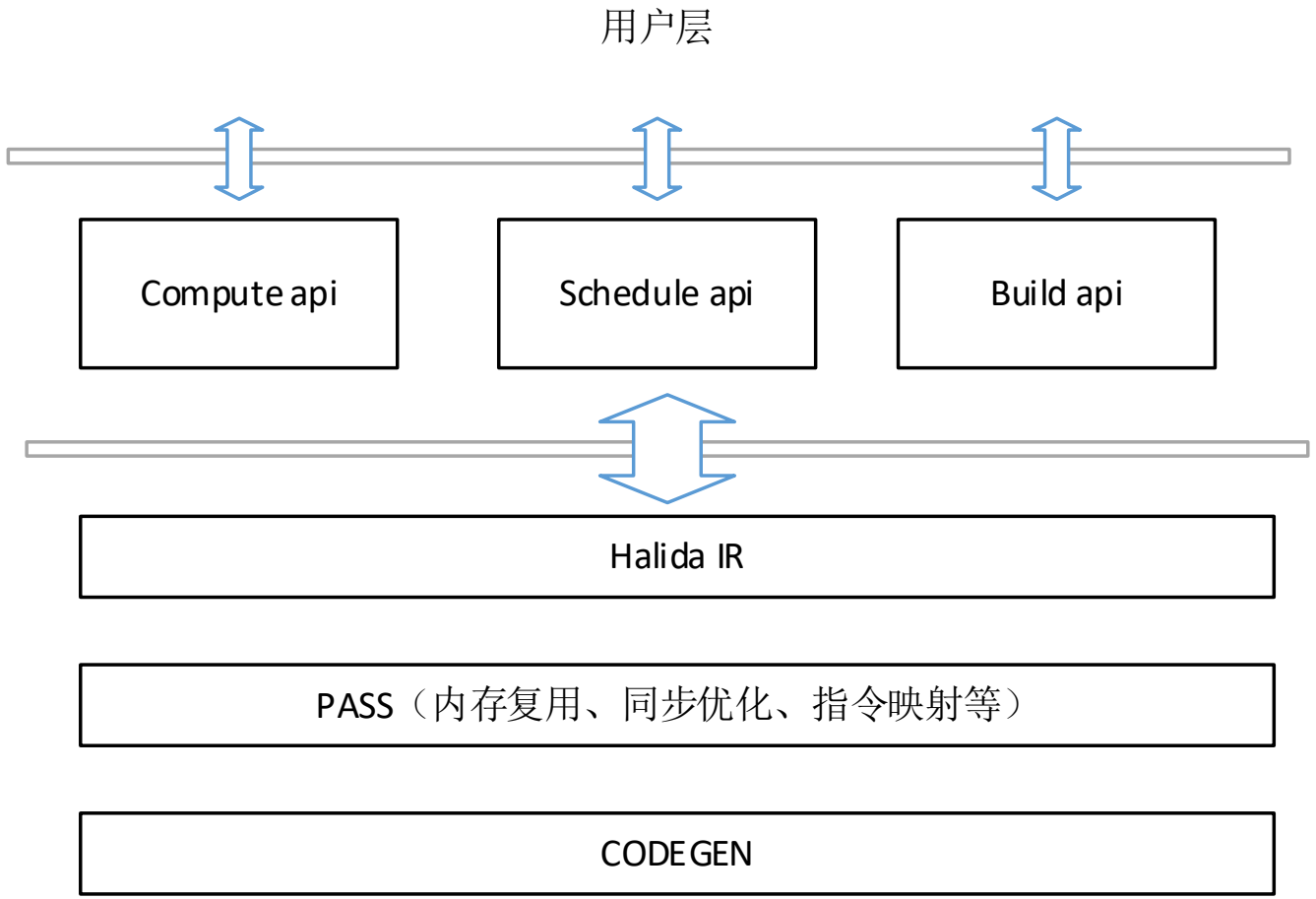
11	通过参数arg0接收数据。 `std::shared_ptr<std::string> input_arg = std::static_pointer_cast<std::string>(arg0);`	● 名称：DestEngine ● 作用：返回推理结果

12	本示例直接透传给输出端口0。
	`hiai::Engine::SendData(0, "Typename", std::static_pointer_cast<void>(input_arg));`
	`//可以通过此函数将数据流发送给需要传的Engine的port。`	

13	通过回调函数接收结果数据。 `graph->SetDataRecvFunctor(target_port_config, std::shared_ptr<DdkDataRecvInterface>( new DdkDataRecvInterface(test_dest_filename)));`	● 名称：主函数 ● 作用：结束程序，销毁graph对象。
14	结束程序，销毁graph对象。 `hiai::Graph::DestroyGraph(GRAPH_ID);`	

目录

TBE —— 架构概述

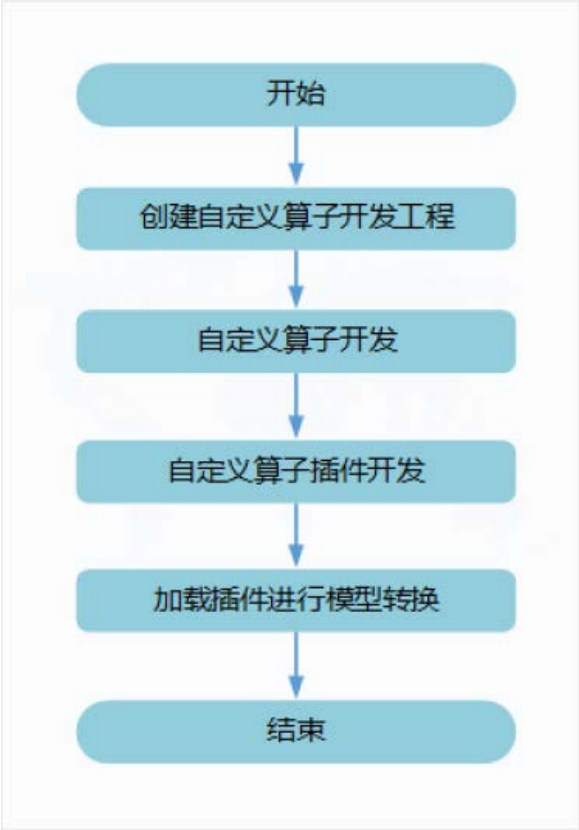


- ✓ **Compute api：** 用于描述Tensor等
- ✓ **Schedule api：** 用于描述算子实现的调度优化
- ✓ **Build api：** 用于描述算子编译动作
- ✓ **PASS：** 对原始HalidaIR或变形后的IR进行变形处理的编译器pass
- ✓ **CODEGEN：** 将变形后的HalidaIR转换成目标代码

TBE —— 算子开发流程

Atlas 200DK提供了TBE（Tensor Boost Engine）算子开发框架，可以开发自定义算子。TE是基于Python API接口，来描述和实现算子开发工具，其主要在TVM（ Tensor Virtual Machine ）框架的基础上，开放了一套描述算子计算的DSL接口。

TVM是一个用于CPU、GPU和专用加速器的端到端深度学习编译器栈，它提供了一套用易用的Python API接口，让用户快速构建特性平台上的描述程序。



- 创建自定义算子开发工程。
- 自定义未实现的算子，包括算子代码开发、单算子的编译、运行、验证。
- 自定义算子插件开发，将算子注册到Framework中，算子插件编译后生成.so插件文件
- 再次进行模型转换时，通过加载*.so插件文件，才能识别自定义的算子，进而完成模型转换。

TBE —— DSL方式开发TBE自定义算子

TE Compute API

TE算子都是调用框架提供的computeAPI来描述计算过程, 接口都是以te.lang.cce.name的形式;

compute API 现根据功能类型可分为以下几类:

·elewise_compute	对Tensor中每个原子值分别做相同操作； te.lang.cce.vabs 即对每个数值x求绝对值
·reduction_compute	对Tensor按轴进行操作；te.lang.cce.sum(data,axis)表示对data按axis进行累加
·segment_compute	对Tensor进行分段操作
·cast_compute	对Tensor中数据的数据类型进行转换；例如float16转换为float32
·broadcast_compute	对Tensor按照目标shape进行广播；shape为（3,1,2）的Tensor广播成(3,3,2)Tensor
·mmad_compute	矩阵乘法
·卷积相关compute	除以上几种compute，还有一些专门针对卷积的compute


文档 > Atlas 200 DK > API参考 > TE API参考

TE API参考

- TE简介
- 说明
- compute接口
- build接口
- 融合接口
- 编译依赖接口
- 使用方式
- 缩略语和术语一览

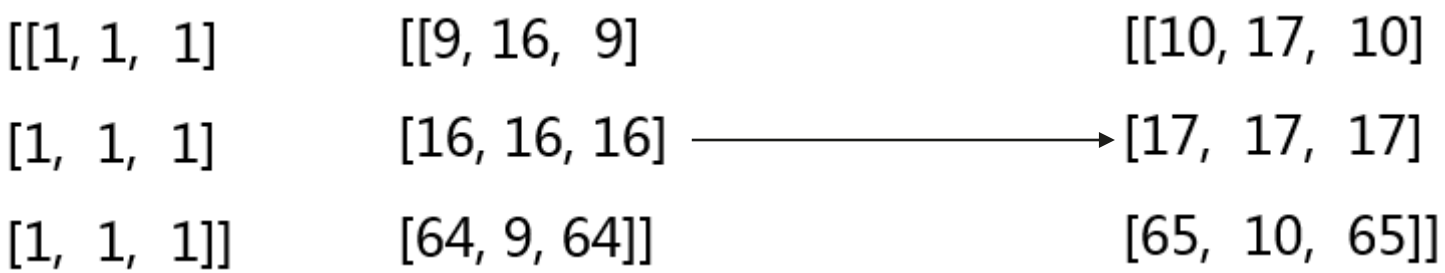
Atlas 200DK TensorEngine API参考

TBE —— DSL方式开发TBE自定义算子

- elewise_compute分为singleElewise 和 binaryElewise以及multipleElewise

singleElewise对Tensor中每个原子值分别做相同操作，如te.lang.cce.vabs 即对每个数值x求绝对值：

binaryElewise是输入两个shape相同的tensor,对应位置上的数值做操作，如te.lang.cce.vadd:



TBE —— 算子开发

TBE-DSL开发（Caffe_Reduction为例）

1. 算子定义：

Reduction算子是Caffe中的Redcution算子，对指定轴及其之后的轴做reduce操作，这个算子包含四种类型。

算子类型	说明
SUM	对被reduce的所有轴求和。
ASUM	对被reduce的所有轴求绝对值后求和。
SUMSQ	对被reduce的所有轴求平方后再求和。
MEAN	对被reduce的所有轴求均值。

TBE —— 算子开发

2. 导入tvm，te模块：

使用TBE定义算子时，需要导入如下模块：

```
import te.lang.cce              # te api
from te import tvm              # tvm api
from topi import generic        # auto_schedule api
from topi.cce import util       # util包含各种参数检查函数
```

3. 接口定义：

根据Caffe中Reduction的定义，我们定义接口如下：

```
def reduction(shape,                        # 输入数据shape
              dtype,                        # 数据类型
              axis,                         # 做Reduce操作的轴
              reduce_op,                    # Reduce操作类型
              coeff,                        # 缩放值
              kernel_name="Reduction",      # 生成kernel函数的名字
              need_build=False,             # 是否编译.o，默认为false
              need_print=False):            # 是否打印IR，默认为false
```

4. 参数校验：

调用util中的校验函数，如果没有，需要自己校验。若该校验函数常用，可加入util.py中方便以后调用：

```
check_list = ["float16", "float32"]
shape_len = len(shape)
util.check_shape_rule(shape)
util.check_shape_size(shape, SHAPE_SIZE_LIMIT)
util.check_dtype_rule(dtype, check_list)
util.check_kernel_name(kernel_name)
axis = util.axis_check(shape_len, axis)
```

util.py中路径：
工程：tensor_engine/topi/python/topi/cce/util.py
DDK：ddk/ddk/site-packages/topi-0.4.0.egg/topi/cce/util.py

TBE —— 算子开发

5. 声明输入：

```
shape = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])] #根据Reduction的定义，我们先对输入shape进行预处理。
data = tvm.placeholder(shape, name="data", dtype=inp_dtype)       #然后调用TVM的placeholder接口
```

在TBE中，使用placeholder接口定义输入Tensor。placeholder就是一个占位符，它返回的是一个Tensor对象，表示一组输入数据，知道它的大小和数据类型。数据的具体内容在运行时才能知道。

- shape是Tensor的尺寸。
- dtype定义数据类型。
- name定义输入的名字。

Tensor是TE中一个重要数据结构。它表示一个Tensor对象，包含：
Tensor的shape（尺寸），比如(10,)或者(1024,1024)或者(2,3,4)等。
Tensor的数据类型，如"float16"、"float32"、"int32"、"int8"。
一个op：op描述了计算信息，可以是一个输入（placeholder）或者一个计算（compute）。

input可以为1个或多个
注意：使用tvm.placeholder声明的输入必须参与计算过程，否则算子会报错。

TBE —— 算子开发流程

6. 描述计算过程：

根据算子定义，调用te.lang.cce的api实现计算过程：

```
if reduce_op == "ASUM":
        data_tmp_input = te.lang.cce.vabs(data)
        res_tmp = te.lang.cce.vmuls(data_tmp_input, cof)
elif reduce_op == "SUMSQ":
        data_tmp_input = te.lang.cce.vmul(data, data)
        res_tmp = te.lang.cce.vmuls(data_tmp_input, cof)
elif reduce_op == "MEAN":
        size = shape[-1]
        cof = float(cof) * (size ** (-0.5))
        res_tmp = te.lang.cce.vmuls(data, cof)
elif reduce_op == "SUM":
        data_tmp_input = te.lang.cce.vmuls(data, cof)
        res_tmp = data_tmp_input
res_tmp = te.lang.cce.sum(res_tmp, axis=axis)
res = te.lang.cce.cast_to(res_tmp, dtype, f1628IntegerFlag=True)
if reduce_op == "MEAN":
        size = shape[-1] ** (-0.5)
        res = te.lang.cce.vmuls(res_tmp, size)
```

TBE —— 算子开发流程

7. 编译算子参数:

经过上面的处理，已经描述了算子的计算过程，然后把它交给TBE，使用TBE的auto_schedule接口生成schedule对象。

- schedule对象中包含一个“中间表示”（IR），它用一种类似伪代码来描述计算过程，可以通过相关参数把它打印出来进行查看。关于schedule的详细描述，可参考TVM官方文档。
- TBE会根据schedule去构建（build）这个算子，最终生成可在硬件上执行的二进制文件。

样例中generic.auto_schedule()API，入参为compute的结果Tensor对象，返回Schedule对象。

```
with tvm.target.cce():                          # 指定硬件目标为CCE平台
    sch = generic.auto_schedule(res)            # 定义Schedule
config = {"print_ir": need_print,               # 定义build参数
          "need_build": need_build,
          "name": kernel_name,
          "tensor_list": [data, res]}
te.lang.cce.cce_build_code(sch, config)     # build算子，生成目标文件
```

TBE —— 算子插件开发

自定义算子插件开发参考实例代码及GE API文档进行开发，主要修改部分有以下几步：

1 解析算子参数（Caffe）

2 获取算子输出描述

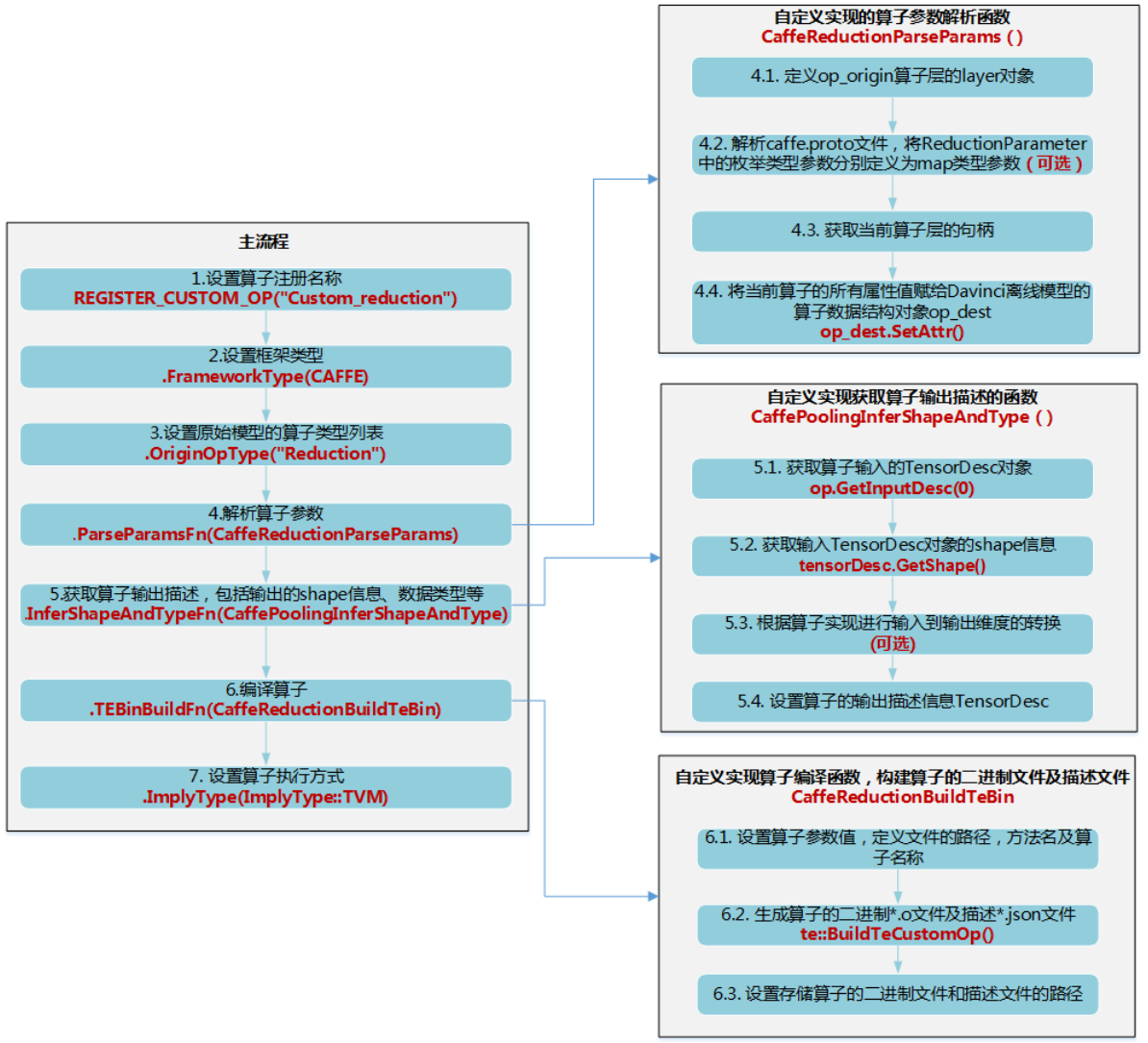
3 编译算子，获取算子的二进制文件及算子描述文件

4 注册算子

文档 > Atlas 200 DK > API参考 > GE API参考

GE API参考

- 简介
- 属性类接口
- 缓存类接口
- Graph类接口
- Model类接口
- Operator类接口
- Shape类接口
- Tensor类接口
- TensorDesc类接口
- Operator注册类
- 模型构建类接口



TBE —— 算子插件开发

算子参数解析函数ParseParamFunc

用户自定义并实现ParseParamFunc类函数，完成caffe模型参数和权值的转换，将结果填到Operator类中。

```
Status ParseParamFunc(const Message* op_origin, ge::Operator& op_dest);
```

参数	输入/输出	说明
op_origin	输入	protobuf格式的数据结构（来源于caffe模型的prototxt文件），包含算子参数信息。
op_dest	输出	适配昇腾处理器的离线模型的算子数据结构，保存算子信息。 关于Operator类，请参见 GE API参考 中的“Operator类接口”。

获取算子输出描述函数InferShapeFunc

用户自定义并实现InferShapeFunc类函数，用于获取算子的输出描述，包括输出shape信息、数据类型等张量描述信息。

```
Status InferShapeFunc(const ge::Operator& op, vector<ge::TensorDesc>&
v_output_desc);
```

参数	输入/输出	说明
op	输入	适配昇腾处理器的离线模型的算子数据结构。 关于Operator类，请参见 GE API参考 中的“Operator类接口”。
v_output_desc	输出	存储算子的输出描述。 关于TensorDesc类，请参见 GE API参考 中的“TensorDesc类接口”。

算子编译、构建算子二进制文件和描述文件的函BuildTeBinFunc

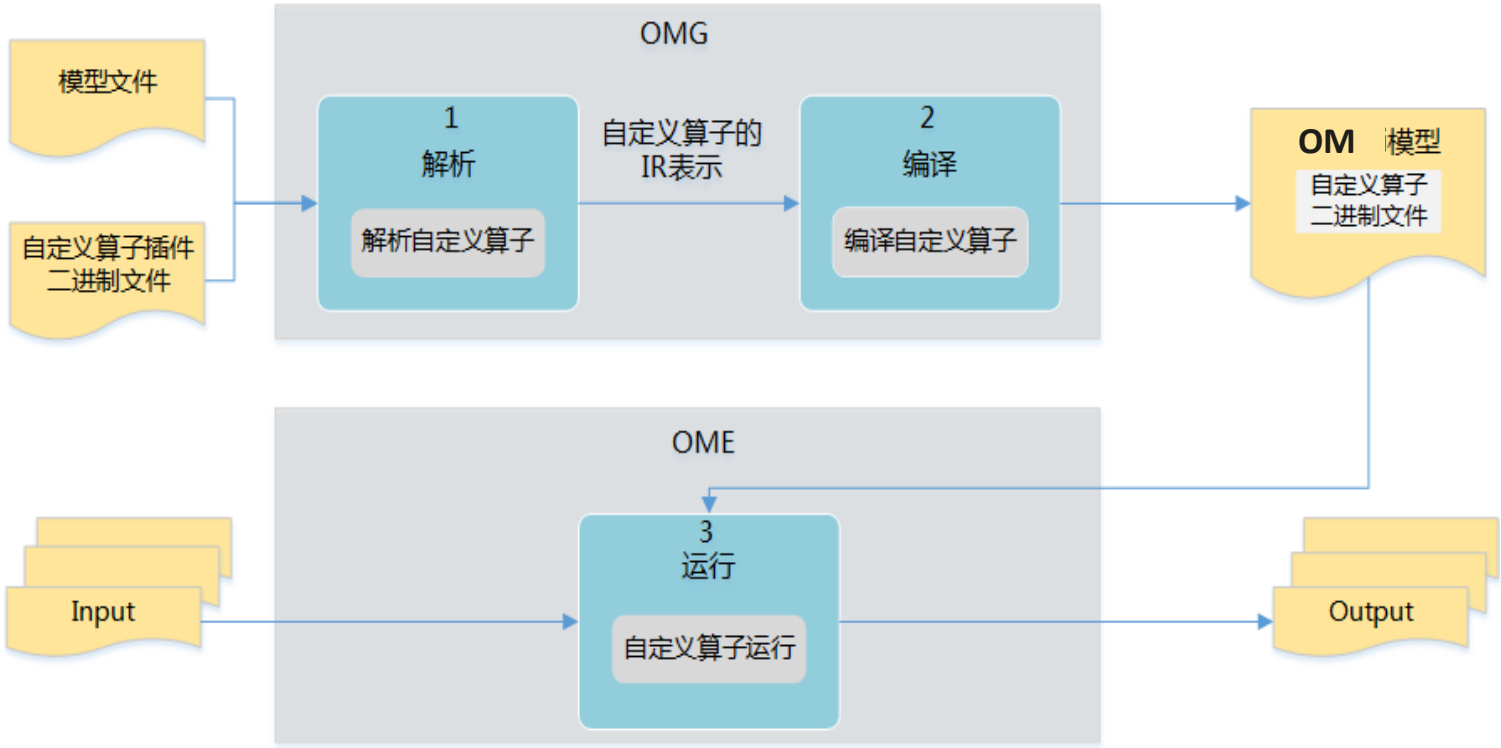
用户自定义并实现BuildTeBinFunc类函数，用于构建算子二进制文件。

```
virtual Status BuildTeBinFunc(const ge::Operator& op, TEBinInfo& teBinInfo);
```

参数	输入/输出	说明
op	输入	适配昇腾处理器的离线模型的算子数据结构，保存算子信息。 关于Operator类，请参见 GE API参考 中的“Operator类接口”。

teBinInfo	输出	存储自定义算子二进制文件路径和ddk描述的数据。
		struct TEBinInfo
		{
		std::string bin_file_path; //自动从json文件binFileName字段获取。为了兼容以前用户编写的用例，字段不删除。
		std::string json_file_path;
		std::string ddk_version;
		};

TBE —— 整网调试



- OMG（Offline Model Generate）加载模型文件、自定义算子插件，对模型文件中的算子进行解析，并将自定义算子转换为IR（Intermediate representation）表示。
- OMG根据运行环境对自定义算子进行数据转换、运行内存计算，同时编译生成自定义算子的二进制文件（*.o）文件，并生成Davinci离线模型（*.om）文件
- 应用程序运行时，OME会获取输入数据，加载离线模型文件进行算子的循环调用执行，输出结果数据。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI