Atlas 200 AI 加速模块

应用软件开发指南（型号 3000）

文档版本	01
发布日期	2019-09-30

华为技术有限公司

版权所有 © 华为技术有限公司 2020。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

HUAWEI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默示的声明或保证。

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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

1 前言.....1
2 开发流程.....2
3 搭建 DDK 开发环境.....4
3.1 DDK 安装指南 （CentOS, x86）.....4
3.1.1 简介.....4
3.1.2 软件包获取.....4
3.1.3 环境准备.....5
3.1.4 DDK 安装.....9
3.1.4.1 安装前准备.....9
3.1.4.2 安装.....10
3.1.4.3 常用操作.....11
3.1.4.3.1 卸载 DDK.....11
3.1.4.3.2 查询 DDK 版本.....11
3.1.4.3.3 配置 OpenPGP 公钥.....12
3.1.5 FAQ.....14
3.1.5.1 pip3（或 pip3.5）升级后运行 **pip3 -V**（或 pip3.5 -V）提示 Import Error:cannot import name main.....14
3.1.5.2 安装 epel-release 失败.....15
3.1.5.3 安装 DDK 过程中提示 Warning: DDK functions except TE have been successfully installed.....15
3.1.5.4 软件包完整性校验返回 WARNING 或 FAIL.....16
3.1.6 附录.....17
3.1.6.1 软件包说明.....17
3.1.6.2 开源第三方库说明.....18
3.2 DDK 安装指南（Ubuntu, x86）.....19
3.2.1 简介.....19
3.2.2 软件包获取.....19
3.2.3 环境准备.....19
3.2.4 DDK 安装.....22
3.2.4.1 安装前准备.....22
3.2.4.2 安装.....23
3.2.4.3 常用操作.....23
3.2.4.3.1 卸载 DDK.....23
3.2.4.3.2 查询 DDK 版本.....24

3.2.4.3.3 配置 OpenPGP 公钥 24
3.2.5 FAQ 26
3.2.5.1 Mind Studio 或 DDK 安装过程中提示 pip2 或 pip 不可用 26
3.2.5.2 软件包完整性校验返回 WARNING 或 FAIL 27
3.2.6 附录 29
3.2.6.1 软件包说明 29
3.2.6.2 开源第三方库说明 30
4 模型转换 32
4.1 简介 32
4.2 转换 Caffe/TensorFlow 网络模型 34
4.2.1 约束及参数说明 34
4.2.2 使用 OMG 工具转换模型 40
4.3 AIPP 配置 41
4.3.1 基本介绍 41
4.3.2 配置文件模板 42
4.3.3 色域转换配置说明 45
4.3.4 Crop/Padding 配置说明 49
4.3.5 根据模型输入指定 AIPP 配置说明 50
4.3.6 AIPP 对模型输入大小的校验说明 50
4.3.7 动态 AIPP 的参数输入结构 51
4.4 量化配置 52
4.4.1 基本介绍 52
4.4.2 配置文件模板 53
4.4.3 参数配置说明 55
4.4.4 参数调优说明 63
4.5 FAQ 66
4.5.1 DDK 安装用户为 HwHiAiUser 时，日志未输出到屏幕 66
4.5.2 DDK 所在服务器操作系统以及架构为 Arm（aarch64），模型转换耗时较长 66
4.5.3 模型转换时提示 Unrecognized layer:xxx, layer type xxx 错误 66
4.5.4 模型转换时提示 It is recommended to convert layers-structure to layer-structure by caffe tool 错误 67
5 算子开发 69
5.1 开发前必读 69
5.1.1 概述 69
5.1.2 目标读者 71
5.2 算子开发注意事项 71
5.3 环境准备 72
5.4 开发流程总览 72
5.5 配置环境变量 73
5.6 创建自定义算子开发工程 73
5.7 自定义算子开发 75
5.7.1 算子特性 75

5.7.2 算子代码实现 79
5.7.2.1 实现流程 79
5.7.2.2 导入 Python 模块 80
5.7.2.3 算子实现 80
5.7.2.4 算子调度与编译 82
5.7.2.5 执行算子 83
5.7.3 代码样例 83
5.7.4 算子运行验证 84
5.7.4.1 算子编译 84
5.7.4.2 构造输入数据文件 86
5.7.4.3 运行单算子 86
5.8 caffe.proto 文件算子定义（可选） 88
5.9 自定义算子插件开发 89
5.9.1 插件代码实现 89
5.9.1.1 插件实现流程 89
5.9.1.2 包含头文件 90
5.9.1.3 解析算子 91
5.9.1.4 推理算子输出张量描述 95
5.9.1.5 编译算子 97
5.9.1.6 算子注册 99
5.9.2 插件代码样例 100
5.9.3 编译算子插件 103
5.10 加载插件转换模型 104
5.11 算子插件开发函数参考 105
5.11.1 ParseParamsFn 函数 105
5.11.2 InferShapeAndTypeFn 函数 106
5.11.3 TEBinBuildFn 函数 106
5.11.4 WeightFormats 函数 107
5.11.5 算子构建函数注册宏 108
5.12 算子清单 109
5.12.1 参数说明 109
5.12.2 算子边界 109
5.12.2.1 整体约束 110
5.12.2.2 Caffe 算子边界 110
5.12.2.3 Tensorflow 算子边界 153
6 应用软件开发 200
6.1 开发前必读 200
6.1.1 适用场景 200
6.1.2 关键概念 201
6.1.3 业务开发约束 202
6.2 软件开发说明（重要） 203
6.3 开发前准备 204

6.3.1 获取样例程序.....204
6.4 开发环境介绍.....205
6.4.1 DDK 目录分布.....205
6.4.2 头文件与库介绍.....205
6.4.3 编译工具介绍.....206
6.4.4 开发工具介绍.....207
6.5 通用推理业务流程介绍.....207
6.5.1 软件业务流程介绍.....207
6.5.2 软件模块与硬件模块对应关系示例.....208
6.6 代码运行样例（HelloDavinci 程序）.....208
6.6.1 获取 HelloDavinci 代码.....208
6.6.2 HelloDavinci 流程框架介绍.....208
6.6.3 HelloDavinci 编译运行.....209
6.7 软件代码开发.....210
6.7.1 配置 Matrix 框架.....210
6.7.1.1 配置、创建与销毁 Graph.....211
6.7.1.2 配置 Engine.....213
6.7.1.3 配置数据传输.....214
6.7.2 调用 DVPP 接口.....215
6.7.2.1 使用 DVPP 接口.....215
6.7.2.2 申请 DVPP 的内存.....216
6.7.3 离线模型推理.....217
6.7.3.1 配置 AIPP.....217
6.7.3.2 转换离线模型.....218
6.7.3.3 执行模型推理.....219
6.7.4 优化软件性能.....221
6.7.5 软件日志调测.....221
6.7.5.1 日志注册.....221
6.7.5.2 日志打印.....222
6.7.5.3 日志查看.....222
6.7.5.4 日志重启.....223
6.7.6 业务软件编译.....224
6.8 Graph 关键字.....224
7 高性能应用编程.....229
7.1 简介.....229
7.2 要点介绍.....229
7.2.1 内存管理.....230
7.2.1.1 原生语言的内存管理接口.....230
7.2.1.2 Matrix 框架提供的内存管理接口.....230
7.2.2 Host-Device 数据传输.....235
7.2.3 DVPP 使用.....238
7.2.3.1 图像/视频编解码.....238

7.2.3.2 图像 Crop/Resize.....238
7.2.4 模型转换预处理配置.....240
7.2.5 Batch 和超时.....240
7.2.6 算法推理输入输出数据处理.....240
7.2.7 回传数据优化处理.....240
7.3 算子使用建议.....241
7.4 示例说明.....242
7.4.1 数据流.....242
7.4.2 ”0“拷贝.....243
7.5 FAQ.....243
7.5.1 申请自动释放内存时使用智能指针，但析构器不为空，导致异常.....243
7.5.2 申请手动释放内存时使用智能指针， 但未将析构器指定为 HIAI_DFree，导致异常.....244
8 API 参考.....246
8.1 Framework API 参考.....246
8.1.1 简介.....246
8.1.2 非类成员函数.....246
8.1.2.1 custom_op_run 函数.....246
8.1.2.2 custom_op_compare 函数.....248
8.1.2.3 GetContext 函数.....248
8.1.2.4 AutoMappingFn 函数.....249
8.1.2.5 aicpu_run_func 函数.....249
8.1.3 类成员函数.....250
8.1.3.1 StatusFactory 类.....250
8.1.3.1.1 Instance 函数.....250
8.1.3.1.2 RegisterErrorNo 函数.....251
8.1.3.1.3 GetErrDesc 函数.....251
8.1.3.1.4 StatusFactory 函数.....251
8.1.3.1.5 ~StatusFactory 函数.....252
8.1.3.2 ErrorNoRegisterar 类.....252
8.1.3.2.1 ErrorNoRegisterar 函数.....252
8.1.3.2.2 ~ErrorNoRegisterar 函数.....252
8.1.3.3 OpTypeContainer 类.....253
8.1.3.3.1 Instance 函数.....253
8.1.3.3.2 Register 函数.....253
8.1.3.3.3 IsExisting 函数.....253
8.1.3.3.4 OpTypeContainer 函数.....254
8.1.3.4 OpTypeRegistrar 类.....254
8.1.3.4.1 OpTypeRegistrar 函数.....254
8.1.3.4.2 ~OpTypeRegistrar 函数.....254
8.1.3.5 OpRegistrationData 类.....255
8.1.3.5.1 OpRegistrationData 函数.....255
8.1.3.5.2 ~OpRegistrationData.....255

8.1.3.5.3 FrameworkType 函数 255
8.1.3.5.4 OriginOpType 函数 256
8.1.3.5.5 ParseParamsFn 函数 256
8.1.3.5.6 InferShapeAndTypeFn 函数 257
8.1.3.5.7 UpdateOpDescFn 函数 258
8.1.3.5.8 GetWorkspaceSizeFn 函数 259
8.1.3.5.9 TEBinBuildFn 函数 260
8.1.3.5.10 ImplyType 函数 261
8.1.3.5.11 Formats 函数 261
8.1.3.5.12 WeightFormats 函数 262
8.1.3.5.13 Finalize 函数 263
8.1.3.6 OpRegistry 类 263
8.1.3.6.1 Instance 函数 264
8.1.3.6.2 Register 函数 264
8.1.3.6.3 GetImplyType 函数 264
8.1.3.6.4 GetOpTypeByImplyType 函数 265
8.1.3.6.5 GetFormats 函数 265
8.1.3.6.6 GetWeightFormats 函数 266
8.1.3.6.7 GetParseParamFunc 函数 266
8.1.3.6.8 GetInferShapeFunc 函数 266
8.1.3.6.9 GetGetWorkspaceSizeFunc 函数 267
8.1.3.6.10 GetUpdateOpDescFunc 函数 267
8.1.3.6.11 GetBuildTeBinFunc 函数 267
8.1.3.6.12 GetTransWeightFunc 函数 268
8.1.3.7 OpReceiver 类 268
8.1.3.7.1 OpReceiver 函数 268
8.1.3.7.2 ~OpReceiver 函数 269
8.1.4 宏定义 269
8.1.4.1 错误码生成宏 269
8.1.4.2 COMMON 模块错误码生成宏 269
8.1.4.3 OMG 模块错误码生成宏 270
8.1.4.4 OME 模块错误码生成宏 270
8.1.4.5 CALIBRATION 模块错误码生成宏 271
8.1.4.6 获取错误码描述宏 271
8.1.4.7 算子类型注册宏 271
8.1.4.8 算子类型是否存在宏 272
8.1.4.9 算子注册宏 272
8.1.4.9.1 REGISTER_CUSTOM_OP 宏 272
8.1.4.10 对外接口标记宏 272
8.1.4.10.1 FMK_FUNC_HOST_VISIBILITY 273
8.1.4.10.2 FMK_FUNC_DEV_VISIBILITY 273
8.2 Matrix API 参考 273

8.2.1 简介 273
8.2.2 流程编排接口 274
8.2.2.1 流程串接接口（C 语言） 274
8.2.2.1.1 HIAI_Init（与 C++共用） 274
8.2.2.1.2 HIAI_CreateGraph 275
8.2.2.1.3 HIAI_DestroyGraph 276
8.2.2.2 流程串接接口（C++语言） 276
8.2.2.2.1 Graph::GetInstance 276
8.2.2.2.2 Graph::CreateGraph（根据配置文件创建 Graph） 277
8.2.2.2.3 Graph::CreateGraph（根据配置文件创建 Graph，并将生成的 Graph 回写到 list 中） 278
8.2.2.2.4 Graph::CreateGraph（根据 Protobuf 数据格式创建 Graph） 279
8.2.2.2.5 Graph::DestroyGraph 280
8.2.2.2.6 Graph::UpdateEngineConfig 280
8.2.2.3 数据传输接口（C 语言） 281
8.2.2.3.1 HIAI_SetDataRecvFunctor 281
8.2.2.3.2 HIAI_C_SendData 282
8.2.2.4 数据传输接口（C++语言） 283
8.2.2.4.1 Graph::SetDataRecvFunctor 283
8.2.2.4.2 Graph::SendData 284
8.2.2.4.3 HIAI_SendData 285
8.2.2.5 接口使用示例 286
8.2.2.5.1 C 语言接口使用示例 286
8.2.2.5.2 C++接口使用示例（根据配置文件创建 Graph） 288
8.2.2.5.3 C++接口使用示例（根据配置文件创建 Graph，并将生成的 Graph 回写到 list 中） 290
8.2.3 Engine 实现接口（C++语言） 292
8.2.3.1 Engine 的构造函数与析构函数 292
8.2.3.2 Engine::Init 293
8.2.3.3 宏:HIAI_DEFINE_PROCESS 294
8.2.3.4 宏:HIAI_IMPL_ENGINE_PROCESS 294
8.2.3.5 Engine::SetDataRecvFunctor 296
8.2.3.6 Engine::SendData 297
8.2.3.7 调用示例 298
8.2.4 模型管家接口（C++语言） 300
8.2.4.1 离线模型管家 300
8.2.4.1.1 AIModelManager::Init 300
8.2.4.1.2 AIModelManager::SetListener 302
8.2.4.1.3 AIModelManager::Process 302
8.2.4.1.4 AIModelManager::CreateOutputTensor 303
8.2.4.1.5 AIModelManager::CreateInputTensor 304
8.2.4.1.6 AIModelManager::IsPreAllocateOutputMem 304
8.2.4.1.7 AIModelManager::GetModelIOTensorDim 305
8.2.4.1.8 AIModelManager::GetMaxUsedMemory 306

8.2.4.1.9 AISimpleTensor::SetBuffer 306
8.2.4.1.10 AITensorFactory::CreateTensor 307
8.2.4.1.11 调用示例 308
8.2.4.2 AIPP 配置接口 310
8.2.4.2.1 简介 310
8.2.4.2.2 SetDynamicInputIndex 310
8.2.4.2.3 SetDynamicInputEdgeIndex 311
8.2.4.2.4 SetInputFormat 311
8.2.4.2.5 SetCscParams（设置默认参数值） 312
8.2.4.2.6 SetCscParams（根据需求设置参数值） 314
8.2.4.2.7 SetRbuvSwapSwitch 316
8.2.4.2.8 SetAxSwapSwitch 317
8.2.4.2.9 SetSrcImageSize 317
8.2.4.2.10 SetCropParams 318
8.2.4.2.11 SetPaddingParams 319
8.2.4.2.12 SetDtcPixelMean 320
8.2.4.2.13 SetDtcPixelMin 321
8.2.4.2.14 SetPixelVarReci 322
8.2.4.2.15 SetInputDynamicAIPP 323
8.2.4.2.16 GetDynamicInputIndex 324
8.2.4.2.17 GetDynamicInputEdgeIndex 325
8.2.4.2.18 调用示例 326
8.2.4.3 数据类型 328
8.2.4.3.1 AIConfigItem 328
8.2.4.3.2 AIConfig 328
8.2.4.3.3 AITensorParaDescription 328
8.2.4.3.4 AITensorDescription 328
8.2.4.3.5 AITensorDescriptionList 329
8.2.4.3.6 AIModelDescription 329
8.2.4.3.7 AINNNodeDescription 330
8.2.4.3.8 AINNNodeDescriptionList 330
8.2.4.3.9 AIAPIDescription 330
8.2.4.3.10 AIAPIDescriptionList 330
8.2.4.3.11 AIOPDescription 330
8.2.4.3.12 AIOPDescriptionList 331
8.2.4.3.13 NodeDesc 331
8.2.4.3.14 EngineDesc 331
8.2.4.3.15 GraphInitDesc 331
8.2.4.3.16 GeneralFileBuffer 332
8.2.4.3.17 AIContext 332
8.2.4.3.18 TensorDimension 332
8.2.4.3.19 IAIListener 333

8.2.4.3.20 IAITensor 334
8.2.4.4 其他用于编译依赖的接口 335
8.2.4.4.1 AIAlgAPIFactory 335
8.2.4.4.2 AIAlgAPIRegisterar 335
8.2.4.4.3 REGISTER_ALG_API_UNIQUE 336
8.2.4.4.4 REGISTER_ALG_API 336
8.2.4.4.5 AIModelManager 336
8.2.4.4.6 getModelInfo 337
8.2.4.4.7 IAINNNode 337
8.2.4.4.8 AINNNodeFactory 339
8.2.4.4.9 AINNNodeRegisterar 339
8.2.4.4.10 REGISTER_NN_NODE 340
8.2.4.4.11 AITensorGetBytes 340
8.2.4.4.12 AITensorFactory 340
8.2.4.4.13 REGISTER_TENSOR_CREATER_UNIQUE 341
8.2.4.4.14 REGISTER_TENSOR_CREATER 341
8.2.4.4.15 AISimpleTensor 342
8.2.4.4.16 AINeuralNetworkBuffer 343
8.2.4.4.17 AIImageBuffer 345
8.2.4.4.18 HIAILog 346
8.2.4.4.19 HIAI_ENGINE_LOG 347
8.2.4.5 异常处理 348
8.2.5 辅助接口 348
8.2.5.1 数据获取接口（C++语言） 348
8.2.5.1.1 获取 Device 数目 348
8.2.5.1.2 获取第一个 DeviceID 349
8.2.5.1.3 获取第一个 GraphID 349
8.2.5.1.4 获取下一个 DeviceID 350
8.2.5.1.5 获取下一个 GraphID 351
8.2.5.1.6 获取 Engine 指针 352
8.2.5.1.7 获取 Graph 的 GraphId 352
8.2.5.1.8 获取 Graph 的 DeviceID 352
8.2.5.1.9 获取 Engine 的 GraphId 353
8.2.5.1.10 获取 Engine 队列最大大小 353
8.2.5.1.11 获取 Engine 指定端口当前队列大小 353
8.2.5.1.12 解析 Matrix 配置文件 354
8.2.5.1.13 获取 PCIe 的 Info 354
8.2.5.1.14 获取版本号 355
8.2.5.1.15 获取 OamConfig 智能指针 355
8.2.5.2 数据类型序列化和反序列化（C++语言） 356
8.2.5.2.1 宏:HIAI_REGISTER_DATA_TYPE 356
8.2.5.2.2 宏:HIAI_REGISTER_TEMPLATE_DATA_TYPE 357

8.2.5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC 358
8.2.5.2.4 Graph::ReleaseDataBuffer 359
8.2.5.2.5 接口使用示例 360
8.2.5.3 内存管理（C 语言） 362
8.2.5.3.1 HIAI_DMalloc 363
8.2.5.3.2 HIAI_DFree 365
8.2.5.3.3 HIAI_DVPP_DMalloc 365
8.2.5.3.4 HIAI_DVPP_DFree 366
8.2.5.4 内存管理（C++语言） 367
8.2.5.4.1 HIAIMemory::HIAI_DMalloc 367
8.2.5.4.2 HIAIMemory::HIAI_DFree 370
8.2.5.4.3 HIAIMemory:: IsDMalloc 370
8.2.5.4.4 HIAIMemory::HIAI_DVPP_DMalloc 371
8.2.5.4.5 HIAIMemory::HIAI_DVPP_DFree 372
8.2.5.5 日志（C++语言） 373
8.2.5.5.1 错误码注册 373
1. 宏 HIAI_DEF_ERROR_CODE 373
2. 接口使用示例 373
8.2.5.5.2 日志打印 374
1. 日志打印格式 1 374
2. 日志打印格式 2 375
3. 日志打印格式 3 375
4. 日志打印格式 4 376
5. 日志打印格式 5 376
6. 日志打印格式 6 377
7. 日志打印格式 7 378
8. 日志打印格式 8 378
8.2.5.6 队列管理 MultiTypeQueue 接口（C++语言） 379
8.2.5.6.1 MultiTypeQueue 构造函数 379
8.2.5.6.2 PushData 379
8.2.5.6.3 FrontData 380
8.2.5.6.4 PopData 380
8.2.5.6.5 PopAllData 381
8.2.5.6.6 接口使用示例 381
8.2.5.7 事件注册接口（C++语言） 381
8.2.5.7.1 Graph::RegisterEventHandle 382
8.2.5.8 其他（C++语言） 382
8.2.5.8.1 DataRecvInterface::RecvData 382
8.2.5.8.2 Graph::SetPublicKeyForSignatureEncryption 383
8.2.6 Python 接口 384
8.2.7 附录 386
8.2.7.1 后续版本会删除的接口 386

8.2.7.1.1 低功耗接口（C++语言）..... 386
1. PowerState::SubscribePowerStateEvent..... 386
2. PowerState::UnsubscribePowerStateEvent..... 387
8.2.7.1.2 数据类型..... 388
1. 枚举：PowerState::DEVICE_POWER_STATE..... 388
2. PowerState::PowerStateNotifyCallbackT..... 388
8.2.7.2 Matrix 数据类型..... 388
8.2.7.2.1 Protobuffer 数据类型..... 388
8.2.7.2.2 Matrix 自定义数据类型..... 392
8.2.7.3 Matrix 已经注册的数据结构..... 393
8.2.7.4 示例..... 408
8.2.7.4.1 编排配置示例..... 408
8.2.7.4.2 性能优化传输示例..... 410
8.3 DVPP API 参考..... 411
8.3.1 概述..... 411
8.3.1.1 接口简介..... 411
8.3.1.2 接口列表..... 412
8.3.1.3 VPC 功能..... 413
8.3.1.4 JPEGE 功能..... 418
8.3.1.5 JPEGD 功能..... 419
8.3.1.6 PNGD 功能..... 420
8.3.1.7 VDEC 功能..... 421
8.3.1.8 VENC 功能..... 422
8.3.1.9 关于输入输出内存的说明..... 423
8.3.2 VPC/JPEGE/JPEGD/PNGD 功能接口..... 424
8.3.2.1 CreateDvppApi..... 424
8.3.2.2 DvppCtl..... 425
8.3.2.2.1 接口说明..... 425
8.3.2.2.2 VPC 参数说明..... 427
8.3.2.2.3 JPEGE 参数说明..... 432
8.3.2.2.4 JPEGD 参数说明..... 433
8.3.2.2.5 PNGD 参数说明..... 437
8.3.2.2.6 查询 DVPP 引擎参数说明..... 439
8.3.2.3 DestroyDvppApi..... 441
8.3.2.4 DvppGetOutParameter..... 442
8.3.3 VDEC 功能接口..... 443
8.3.3.1 总体说明..... 443
8.3.3.2 CreateVdecApi..... 443
8.3.3.3 VdecCtl..... 444
8.3.3.4 DestroyVdecApi..... 447
8.3.4 VENC 功能接口..... 447
8.3.4.1 总体说明..... 447

8.3.4.2 CreateVenc.....448
8.3.4.3 RunVenc.....449
8.3.4.4 DestroyVenc.....450
8.3.5 兼容旧版本的功能.....451
8.3.5.1 兼容性说明.....451
8.3.5.2 VPC 功能及参数说明.....451
8.3.5.3 CMDLIST 功能及参数说明.....465
8.3.5.4 VENC 功能及参数说明.....470
8.3.6 调用示例.....473
8.3.6.1 实现 VPC 功能.....473
8.3.6.2 实现 JPEGE 功能.....481
8.3.6.3 实现 JPEGD 功能.....484
8.3.6.4 实现 PNGD 功能.....486
8.3.6.5 实现 VDEC 功能.....488
8.3.6.6 实现 VENC 功能.....492
8.3.7 数据类型.....493
8.3.7.1 VpcUserImageConfigure 中的结构体.....493
8.3.7.2 vdec_in_msg 中的结构体和类.....496
8.3.7.3 IMAGE_CONFIG 中的结构体.....499
8.3.7.4 dvpp_engine_capability_stru 中的结构体.....501
8.3.7.5 vpc_in_msg 中的结构体.....509
8.3.7.5.1 Rdma 通道结构体 RDMACHANNEL.....509
8.3.7.5.2 Vpc 内部优化结构体 VpcTurningReverse.....510
8.3.8 附录.....510
8.3.8.1 辅助功能接口.....510
8.3.8.2 不推荐使用接口列表.....511
8.3.8.3 DVPP 执行器样例使用说明.....512
8.3.8.3.1 环境准备.....512
8.3.8.3.2 DVPP 执行器样例入参说明.....512
8.3.8.3.3 VPC 使用说明.....515
1. VPC 基础功能 1.....515
2. VPC 基础功能 2.....515
8.3.8.3.4 VDEC 使用说明.....516
8.3.8.3.5 VENC 使用说明.....516
8.3.8.3.6 JPEGE 使用说明.....517
8.3.8.3.7 JPEGD 使用说明.....517
8.3.8.3.8 PNGD 使用说明.....518
8.3.8.3.9 JPEGD+VPC+JPEGE 串联使用说明.....518
8.3.8.4 异常处理.....518
8.3.8.5 缩略语和术语一览.....519
8.4 GE API 参考.....519
8.4.1 简介.....519

8.4.2 属性类接口 520
8.4.2.1 SetAttr 520
8.4.2.2 GetAttr 521
8.4.2.3 HasAttr 521
8.4.2.4 DelAttr 522
8.4.2.5 SetName 523
8.4.2.6 GetName 523
8.4.2.7 GetItem 524
8.4.2.8 GetValueType 525
8.4.2.9 IsEmpty 525
8.4.2.10 Copy 526
8.4.2.11 MutableTensor 526
8.4.2.12 MutableListTensor 527
8.4.2.13 InitDefault 528
8.4.3 缓存类接口 528
8.4.3.1 ClearBuffer 528
8.4.3.2 GetData 529
8.4.3.3 GetSize 530
8.4.3.4 CopyFrom 530
8.4.4 Graph 类接口 531
8.4.4.1 SetInputs 531
8.4.4.2 SetOutputs 532
8.4.4.3 IsValid 533
8.4.4.4 AddOp 533
8.4.4.5 FindOpByName 534
8.4.4.6 CheckOpByName 535
8.4.4.7 GetAllOpName 535
8.4.5 Model 类接口 536
8.4.5.1 SetName 536
8.4.5.2 GetName 537
8.4.5.3 SetVersion 537
8.4.5.4 GetVersion 538
8.4.5.5 SetPlatformVersion 539
8.4.5.6 GetPlatformVersion 539
8.4.5.7 GetGraph 540
8.4.5.8 SetGraph 540
8.4.5.9 Save 541
8.4.5.10 Load 542
8.4.5.11 IsValid 542
8.4.6 Operator 类接口 543
8.4.6.1 Operator 构造函数 543
8.4.6.2 GetName 544

8.4.6.3 SetInput（srcOprt 只存在一个输出）.....544
8.4.6.4 SetInput（srcOprt 存在多个输出）.....545
8.4.6.5 GetInputDesc.....546
8.4.6.6 TryGetInputDesc.....547
8.4.6.7 UpdateInputDesc.....548
8.4.6.8 GetOutputDesc.....548
8.4.6.9 UpdateOutputDesc.....549
8.4.6.10 GetDynamicInputDesc.....550
8.4.6.11 UpdateDynamicInputDesc.....551
8.4.6.12 GetDynamicOutputDesc.....552
8.4.6.13 UpdateDynamicOutputDesc.....553
8.4.6.14 SetAttr.....554
8.4.6.15 GetAttr.....555
8.4.7 Shape 类接口.....556
8.4.7.1 GetDimNum.....556
8.4.7.2 GetDim.....557
8.4.7.3 GetDims.....557
8.4.7.4 SetDim.....558
8.4.7.5 GetShapeSize.....559
8.4.8 Tensor 类接口.....559
8.4.8.1 GetTensorDesc.....559
8.4.8.2 MutableTensorDesc.....560
8.4.8.3 SetTensorDesc.....560
8.4.8.4 GetData.....561
8.4.8.5 MutableData.....562
8.4.8.6 SetData.....562
8.4.8.7 Clone.....563
8.4.9 TensorDesc 类接口.....564
8.4.9.1 Update.....564
8.4.9.2 GetShape.....566
8.4.9.3 MutableShape.....566
8.4.9.4 SetShape.....567
8.4.9.5 GetFormat.....567
8.4.9.6 SetFormat.....568
8.4.9.7 GetDataType.....569
8.4.9.8 SetDataType.....569
8.4.9.9 Clone.....570
8.4.9.10 IsValid.....570
8.4.10 Operator 注册类.....571
8.4.10.1 REG_OP.....571
8.4.10.2 ATTR.....572
8.4.10.3 REQUIRED_ATTR.....574

8.4.10.4 INPUT 575
8.4.10.5 OPTIONAL_INPUT 577
8.4.10.6 DYNAMIC_INPUT 578
8.4.10.7 OUTPUT 579
8.4.10.8 DYNAMIC_OUTPUT 579
8.4.10.9 INFER_SHAPE_AND_TYPE 580
8.4.10.10 ATTR_ALL_VERIFY 582
8.4.10.11 OP_END 583
8.4.10.12 内置的算子类型列表 583
8.4.11 模型构建类接口 584
8.4.11.1 CreateModelBuff 584
8.4.11.2 BuildIRModel 585
8.4.11.3 ReleaseModelBuff 586
8.5 TE API 参考 587
8.5.1 TE 简介 587
8.5.2 说明 587
8.5.3 compute 接口 592
8.5.3.1 te.lang.cce.vadd(lhs, rhs) 592
8.5.3.2 te.lang.cce.vsub(lhs, rhs) 592
8.5.3.3 te.lang.cce.vmul(lhs, rhs) 593
8.5.3.4 te.lang.cce.vmin(lhs, rhs) 593
8.5.3.5 te.lang.cce.vmax(lhs, rhs) 594
8.5.3.6 te.lang.cce.vor(lhs, rhs) 594
8.5.3.7 te.lang.cce.vand(lhs, rhs) 594
8.5.3.8 te.lang.cce.vadds(raw_tensor, scalar) 595
8.5.3.9 te.lang.cce.vmuls(raw_tensor, scalar) 595
8.5.3.10 te.lang.cce.vlog(raw_tensor) 596
8.5.3.11 te.lang.cce.vexp(raw_tensor) 596
8.5.3.12 te.lang.cce.vabs(raw_tensor) 597
8.5.3.13 te.lang.cce.vrec(raw_tensor) 597
8.5.3.14 te.lang.cce.cast_to(data, dtype, f1628IntegerFlag=False) 597
8.5.3.15 te.lang.cce.vrelu(raw_tensor) 598
8.5.3.16 te.lang.cce.vnot(raw_tensor) 599
8.5.3.17 te.lang.cce.vaxpy(lhs, rhs, scalar) 599
8.5.3.18 te.lang.cce.vmla(x, y, z) 600
8.5.3.19 te.lang.cce.vmadd(x, y, z) 600
8.5.3.20 te.lang.cce.vmaddrelu(x, y, z) 601
8.5.3.21 te.lang.cce.ceil(raw_tensor) 601
8.5.3.22 te.lang.cce.floor(raw_tensor) 601
8.5.3.23 te.lang.cce.round(raw_tensor) 602
8.5.3.24 te.lang.cce.sum(raw_tensor, axis, keepdims=False) 602
8.5.3.25 te.lang.cce.reduce_min(raw_tensor, axis, keepdims=False) 603

8.5.3.26 te.lang.cce.reduce_max(raw_tensor, axis, keepdims=False) 603
8.5.3.27 te.lang.cce.reduce_prod(raw_tensor, axis, keepdims=False) 604
8.5.3.28 te.lang.cce.broadcast(var, shape, output_dtype=None) 604
8.5.3.29 te.lang.cce.unsorted_segment_sum(tensor, segment_ids, num_segments, init_value=0) 605
8.5.3.30 te.lang.cce.unsorted_segment_mean(tensor, segment_ids, num_segments, init_value=0) 606
8.5.3.31 te.lang.cce.unsorted_segment_prod(tensor, segment_ids, num_segments, init_value=0) 607
8.5.3.32 te.lang.cce.unsorted_segment_min(tensor, segment_ids, num_segments, init_value=0) 609
8.5.3.33 te.lang.cce.unsorted_segment_max(tensor, segment_ids, num_segments, init_value=0) 610
8.5.3.34 te.lang.cce.concat(raw_tensors, axis) 611
8.5.3.35 te.lang.cce.conv(*args) 612
8.5.3.36 te.lang.cce.compute_four2five(input, raw_shape_4D) 613
8.5.3.37 te.lang.cce.compute_five2four(input, raw_shape_4D) 614
8.5.3.38 te.lang.cce.matmul(tensor_a, tensor_b, trans_a=False, trans_b=False, alpha_num=1.0, beta_num=0.0, tensor_c=None) 614
8.5.4 build 接口 615
8.5.4.1 topi.generic.auto_schedule(outs) 615
8.5.4.2 te.lang.cce.cce_build_code(sch, config_map = {}) 615
8.5.5 融合接口 616
8.5.5.1 bool BuildTeCustomOp(std::string ddkVer, std::string opName, std::string opPath, std::string opFuncName, const char *format, ...) 616
8.5.6 编译依赖接口 617
8.5.7 使用方式 618
8.5.7.1 使用示例 618
8.5.7.2 异常处理 618

9 命令参考 620

9.1 IDE-daemon-client 命令参考 620
9.1.1 简介 620
9.1.2 IDE Daemon 发送命令给 Device 侧执行 621
9.1.3 IDE Daemon 发送命令给 Host 侧执行 628
9.1.4 IDE Daemon 传输文件给 Device 侧 635
9.1.5 IDE Daemon 传输文件给 Host 侧 637
9.1.6 IDE Daemon 从 Host 侧获取文件 639
9.1.7 检测 IDE Daemon 与 Host 之间的心跳 640
9.1.8 设置日志级别 641
9.1.9 IDE Daemon 将 Profiling 采集项的信息发送给 Host 644
9.1.10 从 Host 侧采集黑匣子的 dump 异常信息 645
9.1.11 获取与 Host 连接的 Device 信息 647
9.1.12 生成字符串加密文件 648
9.1.13 同步 IDE Daemon 与 Host 侧的时间 649
9.1.14 查看帮助信息 650

10 附录 652

10.1 如何获取帮助 652

10.1.1 联系华为前的准备.....652
10.1.2 联系华为技术支持.....652
10.1.3 做好必要的调试准备.....652
10.1.4 如何使用文档.....653
10.1.5 如何从网站获取帮助.....653
10.1.6 联系华为的方法.....654
10.2 免责声明.....654

1 前言

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
危险	表示如不避免则将会导致死亡或严重伤害的具有高等级风险的危害。
警告	表示如不避免则可能导致死亡或严重伤害的具有中等级风险的危害。
注意	表示如不避免则可能导致轻微或中度伤害的具有低等级风险的危害。
须知	用于传递设备或环境安全警示信息。如不避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修订记录

文档版本	发布日期	修改说明
01	2019-09-30	第一次正式发布。

2 开发流程

本文档详细讲述了开发应用软件的相关工作和流程，主要包括搭建DDK开发环境、模型转换、算子开发和开发过程。

搭建 DDK 开发环境

DDK为用户提供基于昇腾AI处理器的相关算法开发工具包，可以用于构建相关工程的编译环境。具体搭建指导请参见**3 搭建DDK开发环境**。

模型转换

将开源框架的网络模型，例如Caffe、TensorFlow等框架训练好的模型，通过离线模型生成器OMG（Offline Model Generator）将其转换成昇腾AI处理器支持的离线模型。模型转换过程中可以实现算子调度的优化、权值数据重排、量化压缩、内存使用优化等，可以脱离设备完成模型的预处理。具体转换方法请参见**4 模型转换**。

算子开发

当开发者想自己开发算子以此提高计算性能，或者昇腾AI软件栈不支持模型中的算子时，开发者可以选择自己开发算子。具体开发流程请参见**5 算子开发**，TE自定义算子开发框架的接口说明请参见**8.5 TE API参考**，自定义算子插件开发的接口说明请参见**8.4 GE API参考**。

应用软件开发

上述准备工作完成后，就可以基于昇腾AI软件栈进行应用软件的开发，具体开发流程和方法请参见**6 应用软件开发**。

在开发过程中，开发者会调用**表2-1**所示的昇腾AI软件栈组件实现应用开发和模型推理。

表 2-1 昇腾 AI 软件栈组件

组件	说明	参考文档
框架管理器（Framework）	将原始神经网络模型转换成昇腾AI处理器支持的形态，并将转换的模型与昇腾AI处理器相融合，引导神经网络运行并高效发挥其性能。	**8.1 Framework API参考**
流程编排器（Matrix）	负责完成神经网络在昇腾AI处理器上的落地与实现，统筹整个神经网络生效的过程。	**8.2 Matrix API参考**
数字视觉处理模块（DVPP）	Davinci架构对输入数据有固定的格式要求，如果数据未满足架构规定的输入格式、分辨率等要求，需要调用DVPP进行格式转换，从而保证后续的推理业务。	**8.3 DVPP API参考**
数据回传工具（IDE-daemon-hiai）	提供接口给Matrix和Framework调用，用于DVPP预处理数据和神经网络数据回传。	**9.1 IDE-daemon-client命令参考**

为了保证在开发应用过程中数据传输、DVPP处理以及模型推理环节进行较高性能的处理，开发者可以参考**7 高性能应用编程**章节提供的建议进行应用开发，以便快速构建起自己的高性能应用。

此外，为了提升代码的开发效率，针对昇腾AI处理器的软件组件和推理业务开源了一些代码样例和工具，请登录**https://www.huaweicloud.com/ascend/resources/CodeSamples**获取。

3 搭建 DDK 开发环境

3.1 DDK安装指南（CentOS, x86）
3.2 DDK安装指南（Ubuntu, x86）

3.1 DDK 安装指南（CentOS, x86）

3.1.1 简介

本文介绍独立部署DDK（Device Development Kit）的安装过程以及常见问题处理方法，适用于基于后台命令行的方式进行开发的场景。

DDK为用户提供基于NPU的数字开发者套件。DDK可以用于构建相关工程的编译环境。不同的发布包里集成了不同NPU形态的DDK。当前版本的DDK集成了TE、DVPP、流程编排等组件的依赖库和头文件，用户可以通过makefile编译相应的工程文件。

3.1.2 软件包获取

在工具安装前，请准备如下软件包。

DDK安装包：请访问**https://support.huawei.com/enterprise**，在搜索栏中搜索对应产品名进入产品节点下，单击“软件”页签，获取DDK安装包。

软件包说明如**表3-1**所示。

表 3-1 软件包说明

安装包	包说明	软件包完整性校验文件
MSpore_DDK-{version}-x86_64.centOS7.4-x86_64.centOS7.4-aarch64.miniOS.tar.gz	DDK安装包。	MSpore_DDK-{version}-x86_64.centOS7.4-x86_64.centOS7.4-aarch64.miniOS.tar.gz.asc

说明

{version}为DDK版本号。

3.1.3 环境准备

环境要求

安装DDK的环境，所要求的硬件及操作系统要满足以下条件。

表 3-2 CentOS 系统版本信息

类别	版本限制	获取方式	注意事项
操作系统	7.4	请从**http://vault.centos.org/7.4.1708/isos/x86_64/**网站下载对应版本软件进行安装，可以下载如下推荐的版本：**CentOS-7-x86_64-DVD-1708.iso**	DDK的安装及离线安装包都是基于Centos的基础版本服务器系统**Infrastructure Server**进行适配及测试的，所以用户在进行系统安装时务必选择安装**Infrastructure Server**系统软件，不能选择其它系统软件，例如Minimal Install、Development and Creative Workstation等。
Python	Python2：2.7+ Python3：3.5+	请参见**安装依赖**。	-

准备 DDK 安装用户（可选）

您可以使用任意用户（root或非root用户）安装DDK。

- 若使用root用户安装，则不需要操作该章节。
- 若使用已存在的非root用户安装，须保证该用户对$HOME目录具有读写以及可执行权限。
- 若使用新的非root用户安装，请参考如下步骤进行创建，如下操作请在root用户下执行。本手册以该种场景为例执行DDK的安装。
  a. 执行以下命令创建DDK安装用户并设置该用户的$HOME目录，并确保用户对$HOME目录具有读写以及可执行权限。
  ```
  useradd -d /home/username -m username
  ```

b. 执行以下命令设置密码。
```
passwd username
```

说明

*username*为安装DDK的用户名，该用户的umask值不能大于0027：

- 若要查看umask的值，则执行命令：**umask**
- 若要修改umask的值，则执行命令：**umask *新的取值***

若创建的非root用户为**HwHiAiUser**，且DDK部署在Host侧服务器，使用该用户进行离线模型转换时，日志默认输入到host侧/var/dlog目录中，您也可以通过设置如下环境变量，使日志默认输出到屏幕上。
```
export  SLOG_PRINT_TO_STDOUT=1
```

- 更改文件描述符限制大小。

a. 使用root用户打开系统配置文件：“/etc/security/limits.conf”，命令为：
```
vi /etc/security/limits.conf
```

b. 在limits.conf文件末尾添加如下内容：
```
* soft nproc 65535
* hard nproc 65535
* soft nofile 65535
* hard nofile 65535
```

说明

“*”代表所有用户，建议替换为DDK安装用户名。

c. 保存文件并退出：**:wq!**。

配置 DDK 安装用户权限（可选）

当使用非root用户安装时，需要操作该章节，否则请忽略。

DDK安装前需要下载相关依赖软件，下载依赖软件需要使用**sudo yum**权限，请以root用户执行如下操作。

1. 执行以下命令打开“/etc/sudoers”文件：
```
chmod u+w /etc/sudoers
vi /etc/sudoers
```
2. 在该文件“root ALL=(ALL) ALL”下面增加如下内容：
```
username ALL=(ALL:ALL)   NOPASSWD:SETENV: /usr/bin/yum
```
“username”为执行安装脚本的普通用户名。
3. 添加完成后，执行**:wq!**保存文件。
4. 执行以下命令取消“/etc/sudoers”文件的写权限。
```
chmod u-w /etc/sudoers
```

说明

请确保“/etc/sudoers”文件的最后一行为“#includedir /etc/sudoers.d”，如果没有该信息，请手动添加。

配置网络

如果CentOS系统启动之后，无法通过远程终端连接网络，需要进行网卡配置，配置方法如下：

在root用户下打开网卡配置文件，命令为：

```
vi /etc/sysconfig/network-scripts/ifcfg-enp2s0f0
```

说明

ifcfg-enp2s0f0为网卡名，此处作为示例，使用时请以用户的网卡名为准。

更改“ONBOOT”参数为“yes”，设置自动启动网络连接，如下图所示。

图 3-1 设置网络参数

```
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=enp2s0f0
UUID=71b3c94b-e746-44f8-a2e8-21b452746dba
DEVICE=enp2s0f0
ONBOOT=yes
IPADDR=10.174.28.173
PREFIX=22
GATEWAY=10.174.28.1
DNS1=10.129.2.34
IPV6_PRIVACY=no
```

说明

BOOTPROTO表示设备的IP类型，如果是静态IP可设置成static或none；如果是动态IP，请配置成dhcp，自动获取IP。

设置完毕后保存文件并退出，执行如下命令重启网络服务：

```
service network restart
```

配置源

DDK安装过程需要下载相关依赖，请确保安装DDK的服务器能够连接网络。请以root用户执行如下操作。

步骤1 执行如下命令检查源是否可用。

```
yum makecache
```

如果命令执行报错，则检查网络是否连接或者把“/etc/yum.repos.d/CentOS-Base.repo”文件中的源更换为可用的源。

步骤2 配置特殊源。

- 安装epel-release，用于支持python下载安装，安装在ius-release之前，命令为：

– 如果为首次安装，命令为：
```
yum install -y epel-release
```
– 如果非首次安装，命令为：
```
yum reinstall -y epel-release
```

说明

如果用yum源安装epel-release失败，请参见FAQ 中的*安装epel-release失败*解决。

- 安装ius-release，执行如下命令：
```
wget https://centos7.iuscommunity.org/ius-release.rpm --no-check-certificate
rpm -vih ius-release.rpm
```
如果安装依赖时无法安装python35u，则参见如下方式替换ius.repo：

配置ius源，用于支持python35u下载安装，ius源在“/etc/yum.repos.d/ius.repo”文件中，请将如下内容替换ius.repo文件中的内容：
```
[ius]
name=IUS Community Packages for Enterprise Linux 7 - $basearch
#baseurl=https://dl.iuscommunity.org/pub/ius/stable/CentOS/7/$basearch
mirrorlist=https://mirrors.iuscommunity.org/mirrorlist?repo=ius-centos7&arch=
$basearch&protocol=http
failovermethod=priority
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-IUS-7

[ius-debuginfo]
name=IUS Community Packages for Enterprise Linux 7 - $basearch - Debug
#baseurl=https://dl.iuscommunity.org/pub/ius/stable/CentOS/7/$basearch/debuginfo
mirrorlist=https://mirrors.iuscommunity.org/mirrorlist?repo=ius-centos7-debuginfo&arch=
$basearch&protocol=http
failovermethod=priority
enabled=0
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-IUS-7

[ius-source]
name=IUS Community Packages for Enterprise Linux 7 - $basearch - Source
#baseurl=https://dl.iuscommunity.org/pub/ius/stable/CentOS/7/SRPMS
mirrorlist=https://mirrors.iuscommunity.org/mirrorlist?repo=ius-centos7-
source&arch=source&protocol=http
failovermethod=priority
enabled=0
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-IUS-7
```

步骤3 更新软件信息。
```
yum clean all
yum makecache
```

----结束

安装依赖

请使用**su - *username***命令切换到DDK安装用户执行如下操作，安装DDK工具依赖的gcc、python等组件。

步骤1 执行如下命令安装相关依赖。
```
sudo -E yum install -y gcc-c++ cmake make python python-devel python2-pip python35u-devel python35u-
pip
```
依赖安装完毕，执行命令**python3**检查是否存在python3，如果提示没有python3，执行如下命令创建软链接，将python3链接到python3.5：

1. **which python3.5**：检查python3.5的安装路径。
2. **sudo ln -s /*xxx*/python3.5 /*xxx*/python3**： /*xxx*表示python3.5以及python3所在的绝对路径。

说明

如果执行该命令时提示没有权限，请切换到root用户设置软链接，设置完毕后，再切换到普通用户。

步骤2 安装python相关依赖。

- 针对Python2.7的依赖安装，首先确保上面的过程执行成功，输入命令**pip -V**检查pip版本信息，若系统的回显信息如下所示：

```
pip 8.1.2 from /usr/lib/python2.7/site-packages (python 2.7)
```

表示pip安装正确，然后执行如下命令，安装python2.7的相关依赖软件（numpy版本仅支持1.11.0~1.16.0，如下以1.16.0为例）：

```
pip install decorator --user
pip install numpy==1.16.0 --user
pip install setuptools --upgrade --user
```

- 针对Python3.5的依赖安装，输入命令**pip3.5 -V**检查pip版本信息，若系统的回显信息如下所示。

```
pip 9.0.1 from /usr/lib/python3.5/site-packages (python 3.5)
```

表示pip3.5安装正确，然后执行如下命令，安装python3.5的相关依赖软件：

```
pip3.5 install numpy --user
pip3.5 install decorator --user
pip3.5 install setuptools --upgrade --user
```

说明

如果执行**pip3.5 -V**时提示“Import Error:cannot import name main”，请参见**3.1.5.1 pip3（或pip3.5）升级后运行pip3 -V（或pip3.5 -V）提示Import Error:cannot import name main**解决。

----结束

3.1.4 DDK 安装

本节介绍DDK的安装方法以及常用操作。

3.1.4.1 安装前准备

上传安装包

使用DDK的安装用户将如下文件上传到**$HOME**目录任意路径下。

- “MSpore_DDK-{version}-x86_64.centOS7.4-x86_64.centOS7.4-aarch64.miniOS.tar.gz”DDK安装包
- “MSpore_DDK-{version}-x86_64.centOS7.4-x86_64.centOS7.4-aarch64.miniOS.tar.gz.asc”DDK安装包校验文件。

说明

以上两个文件请确保在同一目录。

软件包完整性校验

为了防止软件包在传输过程中由于网络原因或存储设备原因出现下载不完整或文件破坏的问题，在执行安装前，建议您对软件包的完整性进行校验。

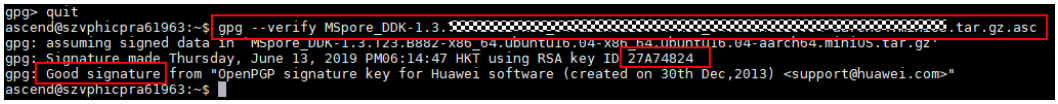
在DDK安装包所在目录，执行如下操作：

步骤1 配置OpenPGP公钥信息，请参考**3.1.4.3.3 配置OpenPGP公钥**。

步骤2 使用DDK安装用户执行如下命令，检测软件包是否合法完整，如**图3-2**所示。

```
gpg --verify "MSpore_DDK****.tar.gz.asc"
```

图 3-2 软件包完整性检测



- 返回信息中“27A74824”为公钥ID。
- 提示信息返回“Good signature”且信息中无 WARNING 或 FAIL，表明此签名为有效签名，软件包完整性校验通过。
- 若提示信息存在 WARNING 或 FAIL，则表明验证不通过，请参见**3.1.5.4 软件包完整性校验返回WARNING或 FAIL**处理建议解决。

说明

- 操作时，请将*MSpore_DDK****.tar.gz.asc*替换为实际安装包对应的校验文件。
- 软件包和软件包.asc文件必须放在同一个路径，才能进行完整性校验。

----结束

3.1.4.2 安装

前提条件

已完成**3.1.3 环境准备**和**3.1.4.1 安装前准备**。

操作步骤

请切换到DDK安装用户执行如下操作。

步骤1 执行如下命令解压安装包。

```
tar -zxvf MSpore_DDK****tar.gz
```

操作时请将*MSpore_DDK****.tar.gz*替换为实际安装包，解压后文件说明请参见**表3-4**。

步骤2 执行如下命令安装DDK。

```
bash install.sh DDK安装目录
```

DDK安装目录在执行安装时会自动创建，例如安装目录设置为“$HOME/tools/che/ddk”，则在安装过程中会自动创建“tools/che/ddk”目录。您也可以自己指定安装路径。

若出现“Successfully installed the DDK!”信息，则表明DDK安装成功。

说明

- 若安装时提示“Warning: DDK functions except TE have been successfully installed.”则参见**3.1.5.3 安装DDK过程中提示Warning: DDK functions except TE have been successfully installed.**解决。
- DDK安装完毕后，安装路径下“$HOME/tools/che/ddk/ddk/toolchains/profiler”的Profiling功能不可用，如果用户需要使用Profiling功能，则请参见《Ascend 310 Mind Studio工具安装指南》手册（请根据Mind Studio安装服务器选择对应形态的安装手册）安装Mind Studio与DDK。

----结束

3.1.4.3 常用操作

介绍DDK安装以及使用过程中的常用操作。

3.1.4.3.1 卸载 DDK

如果您想更新DDK版本，则请卸载当前版本然后再按照5.1-安装步骤安装新的版本，本章节给出DDK的卸载方法。

进入DDK安装路径，本示例为"*$HOME/tools/che/ddk*/ddk/scripts"，使用DDK安装用户执行如下命令卸载DDK：

```
./uninstall.sh
```

如果出现如下所示信息，则说明DDK卸载成功：

```
Info: ide_daemon pem uninstall succ
Starting to remove ddk dir.
```

3.1.4.3.2 查询 DDK 版本

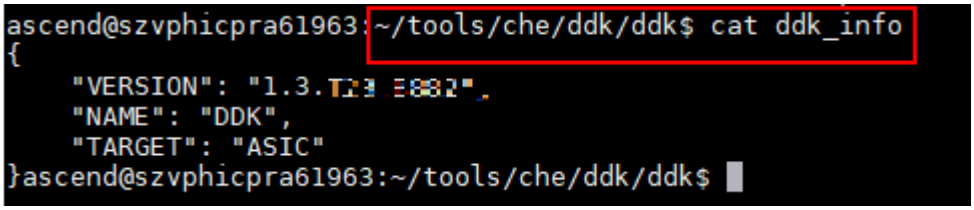
DDK安装成功后，您可以在DDK安装目录下查看DDK版本号。

例如本示例DDK安装目录为"*$HOME/tools/che/ddk*"，则您可以在"*$HOME/tools/che/ddk*/ddk"目录下执行如下命令查看DDK版本号：

```
cat ddk_info
```

返回结果如**图3-3**所示。

图 3-3 查询 DDK 版本号



其中：

- VERSION：表示DDK版本号，查询结果请以实际环境中的为准。
- TARGET：表示DDK的运行环境，目前包括ASIC和Atlas DK。

3.1.4.3.3 配置 OpenPGP 公钥

前提条件

- 请使用DDK的安装用户配置公钥。
- Linux系统已经安装GnuPG工具。

  检查方法：

  – 若已经安装GnuPG工具，在 Shell中输入 **gpg --version**命令，可看到如图3-4所示的回显信息。

    图 3-4 信息回显

```
ascend@ascend-HP-ProDesk-600-G4-PCI-MT:~$ gpg --version
gpg (GnuPG) 1.4.20
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

  – 若没有安装GnuPG工具，则在GnuPG的官方网站**https://www.gnupg.org/**，按照网站的指引，完成工具安装。

配置公钥

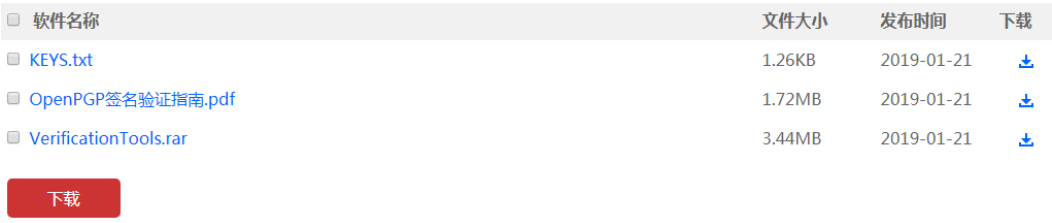
步骤1 获取公钥文件。

进入**OpenPGP下载页面**，单击下载链接，如图3-5所示，界面跳转到文件下载页面。

图 3-5 单击下载文件

文件名为“KEYS”的文件为公钥文件，如图3-6所示。

图 3-6 选择 KEYS 文件



说明

单击链接进入界面显示为中文，若想切换为英文，请单击右上角 选择区域/语言 进行切换。

版权所有 © 华为技术有限公司

步骤2 将下载的**KEYS.txt**文件上传到DDK所在linux系统中。

例如传到"/home/*username*/openpgp/keys"新建目录中。

步骤3 导入公钥文件。

执行如下命令进入 KEYS 公钥文件所在的目录。

```
gpg --import "/home/username/openpgp/keys/KEYS.txt"
```

图 3-7 导入公钥文件

```
ascend@szvphicpra61963:~$ gpg --import "/home/ascend/openpgp/keys/KEYS.txt"
gpg: keyring `/home/ascend/.gnupg/secring.gpg' created
gpg: /home/ascend/.gnupg/trustdb.gpg: trustdb created
gpg: key 27A74824: public key "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

说明

其中“/home/*username*/openpgp/keys”是公钥文件“KEYS”所在的绝对路径，username为DDK的安装用户名，请修改为实际路径。

步骤4 执行如下命令查看公钥导入结果。

```
gpg --fingerprint
```

图 3-8 查看结果

```
ascend@szvphicpra61963:~$ gpg --fingerprint
/home/ascend/.gnupg/pubring.gpg
-------------------------------
pub   2048R/27A74824 2013-12-30
      Key fingerprint = B100 0AC3 8C41 525A 19BD  C087 99AD 81DF 27A7 4824
uid                  OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>
```

步骤5 验证公钥。

- OpenPGP 公钥的合法性需要根据公钥的 ID、指纹、uid 等信息与发布公钥的主体进行合法性验证。当前对外发布的OpenPGP公钥信息如下：
  - 公钥 ID：27A74824
  - 公钥指纹(Key fingerprint)：B100 0AC3 8C41 525A 19BD C087 99AD 81DF 27A7 4824
  - 用户 ID(uid)：OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

  完成信息核实后，可以对该公钥设置信任级别。
- 执行如下命令设置公钥的信任级别。

  ```
  gpg --edit-key "OpenPGP signature key for Huawei software" trust
  ```

  屏幕显示类似如下信息，其中红框部分需要手工输入，“Your decision?”后输入“5”，表示“I trust ultimately”；“Do you really want to set this key to ultimate trust? (y/N)”后输入“y”。

图 3-9 设置公钥信任级别

```
ascend@szvphicpra61963:~$ gpg --edit-key "OpenPGP signature key for Huawei" trust
gpg (GnuPG) 1.4.20; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: unknown       validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: unknown       validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y

pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: ultimate      validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

gpg> quit
```

步骤6 执行**quit**命令退出。

----结束

3.1.5 FAQ

3.1.5.1 pip3（或 pip3.5）升级后运行 pip3 -V（或 pip3.5 -V）提示 Import Error:cannot import name main

问题描述

pip3（或pip3.5）升级后，由于高版本的pip库函数有所变动，所以在重新调用pip3（pip3.5）的运行命令时，会出现**Import Error:cannot import name main**的错误。

解决方法 1

执行如下升级命令。

- pip3对应命令：
  python3 -m pip install --upgrade pip --user
- pip3.5对应命令：
  python3.5 -m pip install --upgrade pip --user

如果该方法无法解决问题，请参见**解决方法2**。

解决方法 2

pip3修改方式：

打开对应目录：**vim /usr/bin/pip3**

将代码修改为**图3-10**所示形式：

图 3-10 修改 pip3 文件代码

```
1 from pip import __main__
2 if __name__ == '__main__':
3     sys.exit(__main__._main())
```

pip3.5修改方法同pip3，目录修改为pip3.5对应目录即可。

3.1.5.2 安装 epel-release 失败

如果使用**sudo -E yum install -y epel-release**安装epel-release失败，请参见如下方法手动安装epel-release：

1. 查看系统版本，确保系统版本和epel版本匹配，例如执行如下命令：
   ```
   cat /etc/redhat-release
   ```
   若得到的结果为CentOS Linux release 7.***，则需要选择的epel版本为7。
2. 进入epel官网**https://dl.fedoraproject.org/pub/epel**，下载对应版本epel安装包，使用命令为：
   ```
   wget https://dl.fedoraproject.org/pub/epel/7Server/x86_64/Packages/e/epel-release-7-11.noarch.rpm --no-check-certificate
   ```
3. 执行如下命令安装相关epel（本例使用的epel版本为7）：
   ```
   rpm -vih epel-release-7-11.noarch.rpm
   ```
4. 执行如下命令更新元数据缓存：
   ```
   yum clean all && yum makecache
   ```

3.1.5.3 安装 DDK 过程中提示 Warning: DDK functions except TE have been successfully installed.

问题描述

执行DDK安装脚本后，安装过程中提示“error : install te-0.4.0.egg failed”、“error : install topi-0.4.0.egg failed”等信息，安装最后提示“Warning: DDK functions except TE have been successfully installed.”，DDK安装成功，但是TE相关功能无法使用。

可能原因

该问题可能是由于安装DDK的服务器，root用户下存在太老的setuptools版本（0.9.8）,切换到DDK安装用户（非root用户）安装DDK时，仍旧读取的是root用户下的setuptools路径，DDK安装用户下的setuptools未生效。

解决方法

若出现上述问题，则使用DDK安装用户先执行如下命令设置环境变量，使DDK安装用户下的setuptools生效。然后重新安装DDK。

```
export PATH=/usr/local/python3/bin:$HOME/bin:$HOME/.local/bin:$PATH
```

3.1.5.4 软件包完整性校验返回 WARNING 或 FAIL

软件包完整性校验如果返回WARNING或 FAIL，则表示验证未通过，请参见表3-3处理建议解决。

表 3-3 场景举例

验证结果场景	输出信息举例	验证结果	处理建议
签名验证通过，没有异常	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: Good signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>"	PASS	NA
签名验证失败	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: BAD signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>"	FAIL	重新下载目标文件。
找不到公钥	gpg: Signature made Thu Jan 9 15:20:01 2014 CST using RSA key ID 27A74824 gpg: Can't check signature: public key not found	FAIL	重新下载公钥，请参见“配置OpenPGP公钥-配置公钥”章节。
签名验证通过，但是公钥没有被设置为完全信任	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: Good signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>" gpg: WARNING: This key is not certified with a trusted signature! gpg: There is no indication that the signature belongs to the owner. Primary key fingerprint: B100 0AC3 8C41 525A 19BD C087 99AD 81DF 27A7 4824	WARNING	确认KeyID为27A74824后，将华为公钥设置为可信，请参见“配置OpenPGP公钥-配置公钥-验证公钥”章节。
找不到对应的源文件	gpg: no signed data gpg: can't hash datafile: No data	FAIL	重新下载目标文件。
签名已到期	gpg: Signature made 04/24/13 10:50:29 CST using RSA key ID 133B64E5 gpg: Expired signature from " OpenPGP signature test key <support@huawei.com>" gpg: Signature expired 04/25/13 10:50:29 CST	FAIL	下载更新过签名的目标文件。

版权所有 © 华为技术有限公司

验证结果场景	输出信息举例	验证结果	处理建议
签名验证通过，但是公钥已被撤销	gpg: Signature made 06/13/13 11:14:49 CST using RSA key ID 133B64E5 gpg: Good signature from " OpenPGP signature test key <support@huawei.com>" gpg: WARNING: This key has been revoked by its owner! gpg: This could mean that the signature is forged. gpg: reason for revocation: Key is no longer used gpg: revocation comment:	WARNING	下载最新公钥和更新了签名的目标文件。
源文件找不到对应的签名文件	无	WARNING	下载目标文件对应的签名文件。

3.1.6 附录

3.1.6.1 软件包说明

DDK安装包解压后包的详细内容请参见表3-4。

表 3-4 软件包说明

安装包	包内容	使用场景
MSpore_DDK-{version}-<uihost arch.os>-<host arch.os>-<device arch.os>.tar.gz 各参数解释请参见表3-5。	ddk.tar.gz	DDK套件安装包。
	install.sh	安装脚本。
	check_sha.sh	校验以上两个文件的完整性，install.sh执行过程中会自动调用该脚本进行完整性校验。

表 3-5 DDK 安装包命名规范

参数	说明
{version}	版本号
<uihost arch.os>	UIhost侧 CPU架构和操作系统及版本，如: x86_64.ubuntu16.04、x86_64.centOS7.4等

版权所有 © 华为技术有限公司

参数	说明
<host arch.os>	Host侧 CPU架构和操作系统及版本，如: x86_64.ubuntu16.04、x86_64.centOS7.4等
<device arch.os>	Device侧 CPU架构和操作系统及版本，如: aarch64.ubuntu16.04、aarch64.miniOS等 ● 开发者板形态（Atlas 200 DK）device侧架构：aarch64.ubuntu16.04 ● 非开发者板形态（Atlas 300等）device侧架构：aarch64.miniOS

3.1.6.2 开源第三方库说明

cereal

cereal是一个开源的（BSD License）、轻量级的、支持C++11特性的、仅仅包含头文件实现的、跨平台 的C++序列化库。它可以将任意的数据类型序列化成不同的表现形式，比如二进制、XML格式或JSON。当前使用版本：1.2.2

详细介绍可参考cereal官网：http://uscilab.github.io/cereal/

gflags

gflags (the Global Flags Editor)是google的一个开源的处理命令行参数的库，比getopt 更方便，更功能强大，使用c++开发，具备python接口。当前使用版本：2.2.1

详细介绍可参考gflags官网：https://github.com/gflags/gflags

glog

glog是Google的一个开源库，用于实现应用级别的logging。 它提供了一系列类似于C++流风格的logging API，以及一些预定义的宏。它有点类似于C里面的assert，但是比它具备更丰富的输出信息以及使用灵活性。

详细介绍可参考glog官网：https://github.com/google/glog

opencv

OpenCV的全称是Open Source Computer Vision Library，是一个跨平台的计算机视觉库。OpenCV可用于开发实时的图像处理、计算机视觉以及模式识别程序。当前使用版本：3.4.2

详细介绍可参考opencv官方网站：https://opencv.org/

Protobuf

Protobuf(Google Protocol Buffers)是Google提供一个具有高效的协议数据交换格式工具库；它是一种平台无关、语言无关、可扩展且轻便高效的序列化数据结构的协议，可以用于网络通信和数据存储。当前使用版本：3.5.1

详细介绍可参考google官方网站：https://developers.google.com/protocol-buffers/

3.2 DDK 安装指南（Ubuntu, x86）

3.2.1 简介

本文介绍独立部署DDK（Device Development Kit）的安装过程以及常见问题处理方法，适用于基于后台命令行的方式进行开发的场景。

DDK为用户提供基于NPU的数字开发者套件。DDK可以用于构建相关工程的编译环境。不同的发布包里集成了不同NPU形态的DDK。当前版本的DDK集成了TE、DVPP、流程编排等组件的依赖库和头文件，用户可以通过makefile编译相应的工程文件。

3.2.2 软件包获取

在工具安装前，请准备如下软件包。

DDK安装包：请访问**https://support.huawei.com/enterprise**，在搜索栏中搜索对应产品名进入产品节点下，单击“软件”页签，获取DDK安装包。

软件包说明如下所示。

表 3-6 软件包说明

安装包	包说明	软件包完整性校验文件
MSpore_DDK-{version}-x86_64.ubuntu16.04-x86_64.ubuntu16.04-aarch64.miniOS.tar.gz	非开发者板形态（Atlas 300等）场景对应的DDK安装包。	MSpore_DDK-{version}-x86_64.ubuntu16.04-x86_64.ubuntu16.04-aarch64.miniOS.tar.gz.asc
MSpore_DDK-{version}-x86_64.ubuntu16.04-aarch64.ubuntu16.04-aarch64.ubuntu16.04.tar.gz	开发者板形态（Atlas 200 DK）场景对应的DDK安装包。	MSpore_DDK-{version}-x86_64.ubuntu16.04-aarch64.ubuntu16.04-aarch64.ubuntu16.04.tar.gz.asc

说明

{version}为DDK版本号。

3.2.3 环境准备

环境要求

安装DDK的环境，所要求的硬件及操作系统要满足以下条件。

表 3-7 Ubuntu 系统版本信息

类别	版本限制	获取方式	注意事项
操作系统	16.04.3	请从**http://old-releases.ubuntu.com/releases/16.04.3/**网站下载对应版本软件进行安装，可以下载桌面版：“ubuntu-16.04.3-desktop-amd64.iso”，或Server版：“ubuntu-16.04.3-server-amd64.iso”。	-
Python	Python2：2.7+ Python3：3.5+	请参见**安装依赖**。	-

准备 DDK 安装用户（可选）

您可以使用任意用户（root或非root用户）安装DDK。

- 若使用root用户安装，则不需要操作该章节。
- 若使用已存在的非root用户安装，须保证该用户对$HOME目录具有读写以及可执行权限。
- 若使用新的非root用户安装，请参考如下步骤进行创建，如下操作请在root用户下执行。本手册以该种场景为例执行DDK的安装。
  a. 执行以下命令创建DDK安装用户并设置该用户的$HOME目录，并确保用户对$HOME目录具有读写以及可执行权限。
  ```
  useradd -d /home/username -m username
  ```
  b. 执行以下命令设置密码。
  ```
  passwd username
  ```

  说明

  *username*为安装DDK的用户名，该用户的umask值不能大于0027：
  - 若要查看umask的值，则执行命令：**umask**
  - 若要修改umask的值，则执行命令：**umask *新的取值***

  若创建的非root用户为**HwHiAiUser**，且DDK部署在Host侧服务器，使用该用户进行离线模型转换时，日志默认输入到host侧/var/dlog目录中，您也可以通过设置如下环境变量，使日志默认输出到屏幕上。
  ```
  export SLOG_PRINT_TO_STDOUT=1
  ```

配置 DDK 安装用户权限（可选）

当使用非root用户安装时，需要操作该章节，否则请忽略。

DDK安装前需要下载相关依赖软件，下载依赖软件需要使用**sudo apt-get**权限，请以root用户执行如下操作。

1. 打开“/etc/sudoers”文件：
```
chmod u+w /etc/sudoers
vi /etc/sudoers
```

2. 在该文件“# User privilege specification”下面增加如下内容：
```
username ALL=(ALL:ALL)    NOPASSWD:SETENV:/usr/bin/apt-get
```
“username”为执行安装脚本的普通用户名。

说明

请确保“/etc/sudoers”文件的最后一行为“#includedir /etc/sudoers.d”，如果没有该信息，请手动添加。

3. 添加完成后，执行**:wq!**保存文件。
4. 执行以下命令取消“/etc/sudoers”文件的写权限：
```
chmod u-w /etc/sudoers
```

检查源

DDK安装过程需要下载相关依赖，请确保安装DDK的服务器能够连接网络。

请在root用户下执行如下命令检查源是否可用。

```
apt-get update
```

说明

如果命令执行报错，则检查网络是否连接或者把“/etc/apt/sources.list”文件中的源更换为可用的源。

安装依赖

请使用**su - *username***命令切换到DDK安装用户执行如下操作，安装DDK工具依赖的gcc、python等组件。

步骤1 执行以下命令安装DDK相关依赖。
```
sudo -E apt-get install gcc g++ cmake make python-pip python3-pip python3 python
```

步骤2 安装 Python开发环境

1. Python2环境安装
   - 环境部署依赖pip，安装TE软件包依赖：numpy（numpy版本仅支持1.11.0~1.16.0，如下以1.16.0为例），decorator，执行如下命令分别安装：
     ```
     pip2 install numpy==1.16.0 --user
     pip2 install decorator --user
     ```
   - Python2环境检查

     使用如下命令分别检查python、pip、setuptools：
     ```
     python2 --version
     pip2 --version
     easy_install --version
     ```
     若分别返回如下所示信息，则说明安装成功：
     ```
     Python 2.7.12
     pip 19.0.1 from /usr/local/lib/python2.7/dist-packages/pip (python 2.7)
     setuptools 20.7.0 from /usr/lib/python2.7/dist-packages (Python 2.7)
     ```
2. Python3 (3.5+)环境安装
   - 环境部署依赖pip3，安装TE软件包依赖：numpy，decorator，执行如下命令分别安装：
     ```
     pip3 install numpy --user
     pip3 install decorator --user
     ```
   - Python3环境检查

     使用如下命令分别检查python3、pip3、setuptools：

```
python3 --version
pip3 --version
easy_install3 --version
```

若分别返回如下所示信息，则说明安装成功：

```
Python 3.5.2
pip 8.1.1 from /usr/lib/python3/dist-packages (python 3.5)
setuptools 20.7.0 from /usr/lib/python3/dist-packages (Python 3.5)
```

----结束

说明

- 如果依赖安装过程中pip安装异常，请参见**3.2.5.1 Mind Studio或DDK安装过程中提示pip2或pip不可用**解决。
- 上述安装成功后返回信息中的版本号只是样例，请以实际环境中的返回信息为准。

3.2.4 DDK 安装

本节介绍DDK的安装方法以及常用操作。

3.2.4.1 安装前准备

上传安装包

使用DDK的安装用户将如下文件上传到**$HOME**目录任意路径下。

- “MSpore_DDK****.tar.gz”：DDK安装包
- “MSpore_DDK****.tar.gz.asc”：DDK安装包校验文件。

说明

DDK具体使用包名请参见**表3-6**，"*"代表DDK具体安装包。以上两个文件请确保在同一目录。

软件包完整性校验

为了防止软件包在传输过程中由于网络原因或存储设备原因出现下载不完整或文件破坏的问题，在执行安装前，建议您对软件包的完整性进行校验。

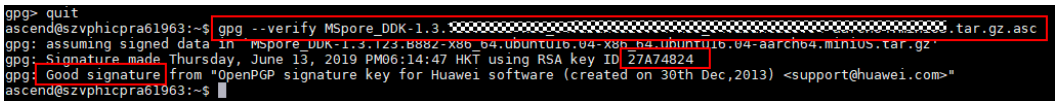
在DDK安装包所在目录，执行如下操作：

步骤1 配置OpenPGP公钥信息，请参考**3.2.4.3.3 配置OpenPGP公钥**。

步骤2 使用DDK安装用户执行如下命令，检测软件包是否合法完整，如**图3-11**所示。

```
gpg --verify "MSpore_DDK****.tar.gz.asc"
```

图 3-11 软件包完整性检测



- 返回信息中“27A74824”为公钥ID。
- 提示信息返回“Good signature”且信息中无 WARNING 或 FAIL，表明此签名为有效签名，软件包完整性校验通过。

- 若提示信息存在 WARNING 或 FAIL，则表明验证不通过，请参见**3.2.5.2 软件包完整性校验返回WARNING或 FAIL**处理建议解决。

说明

- 操作时，请将*MSpore_DDK****.tar.gz.asc*替换为实际安装包对应的校验文件。
- 软件包和软件包.asc文件必须放在同一个路径，才能进行完整性校验。

----结束

3.2.4.2 安装

前提条件

已完成**3.2.3 环境准备**和**3.2.4.1 安装前准备**。

操作步骤

请切换到DDK安装用户执行如下操作。

步骤1 执行如下命令解压安装包。

```
tar -zxvf MSpore_DDK****tar.gz
```

操作时请将*MSpore_DDK****.tar.gz*替换为实际安装包，解压后文件说明请参见**表3-9**。

步骤2 执行如下命令安装DDK。

```
bash install.sh DDK安装目录
```

DDK安装目录在执行安装时会自动创建，例如安装目录设置为“$HOME/tools/che/ddk”，则在安装过程中会自动创建“tools/che/ddk”目录。您也可以自己指定安装路径。

若出现“Successfully installed the DDK!”信息，则表明DDK安装成功。

说明

DDK安装完毕后，安装路径下“$HOME/tools/che/ddk/ddk/toolchains/profiler”的Profiling功能不可用，如果用户需要使用Profiling功能，则请参见《Ascend 310 Mind Studio工具安装指南》手册（请根据Mind Studio安装服务器选择对应形态的安装手册）安装Mind Studio与DDK。

----结束

3.2.4.3 常用操作

介绍DDK安装以及使用过程中的常用操作。

3.2.4.3.1 卸载 DDK

如果您想更新DDK版本，则请卸载当前版本然后再按照5.1-安装步骤安装新的版本，本章节给出DDK的卸载方法。

进入DDK安装路径，本示例为"*$HOME/tools/che/ddk*/ddk/scripts"，使用DDK安装用户执行如下命令卸载DDK：

```
./uninstall.sh
```

如果出现如下所示信息，则说明DDK卸载成功：

版权所有 © 华为技术有限公司

```
Info: ide_daemon pem uninstall succ
Starting to remove ddk dir.
```

3.2.4.3.2 查询 DDK 版本

DDK安装成功后，您可以在DDK安装目录下查看DDK版本号。

例如本示例DDK安装目录为"*$HOME/tools/che/ddk*"，则您可以在"*$HOME/tools/che/ddk*/ddk"目录下执行如下命令查看DDK版本号：

```
cat ddk_info
```

返回结果如图3-12所示。

图 3-12 查询 DDK 版本号

```
ascend@szvphicpra61963:~/tools/che/ddk/ddk$ cat ddk_info
{
    "VERSION": "1.3.T23.B882",
    "NAME": "DDK",
    "TARGET": "ASIC"
}ascend@szvphicpra61963:~/tools/che/ddk/ddk$
```

其中：

- VERSION：表示DDK版本号，查询结果请以实际环境中的为准。
- TARGET：表示DDK的运行环境，目前包括ASIC和Atlas DK。

3.2.4.3.3 配置 OpenPGP 公钥

前提条件

- 请使用DDK的安装用户配置公钥。
- Linux系统已经安装GnuPG工具。

  检查方法：

  – 若已经安装GnuPG工具，在 Shell中输入 **gpg --version**命令，可看到如图3-13所示的回显信息。

    图 3-13 信息回显

```
ascend@ascend-HP-ProDesk-600-G4-PCI-MT:~$ gpg --version
gpg (GnuPG) 1.4.20
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

  – 若没有安装GnuPG工具，则在GnuPG的官方网站**https://www.gnupg.org/**，按照网站的指引，完成工具安装。

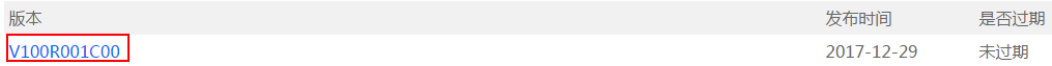
版权所有 © 华为技术有限公司

配置公钥

步骤1 获取公钥文件。

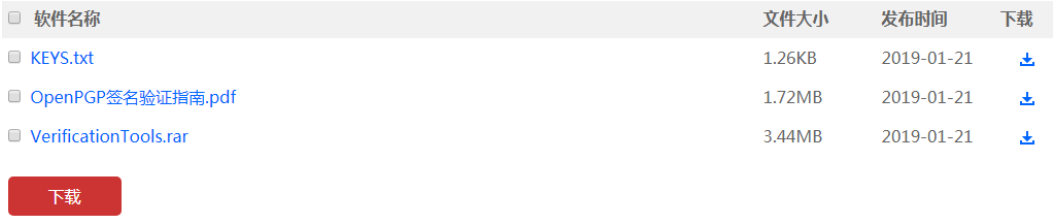
进入**OpenPGP下载页面**，单击下载链接，如**图3-14**所示，界面跳转到文件下载页面。

图 3-14 单击下载文件



文件名为“KEYS”的文件为公钥文件，如**图3-15**所示。

图 3-15 选择 KEYS 文件



说明

单击链接进入界面显示为中文，若想切换为英文，请单击右上角

进行切换。

步骤2 将下载的**KEYS.txt**文件上传到DDK所在linux系统中。

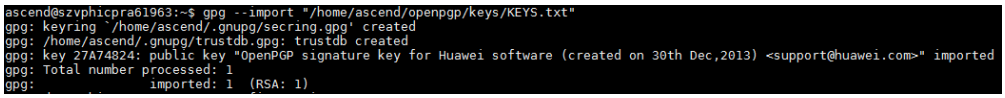
例如传到"/home/*username*/openpgp/keys"新建目录中。

步骤3 导入公钥文件。

执行如下命令进入 KEYS 公钥文件所在的目录。

```
gpg --import "/home/username/openpgp/keys/KEYS.txt"
```

图 3-16 导入公钥文件



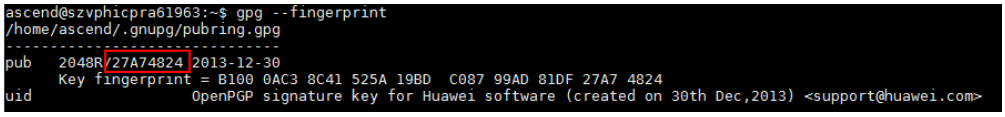
说明

其中“/home/*username*/openpgp/keys”是公钥文件“KEYS”所在的绝对路径，username为DDK的安装用户名，请修改为实际路径。

步骤4 执行如下命令查看公钥导入结果。

```
gpg --fingerprint
```

图 3-17 查看结果



版权所有 © 华为技术有限公司

步骤5 验证公钥。

- OpenPGP 公钥的合法性需要根据公钥的 ID、指纹、uid 等信息与发布公钥的主体进行合法性验证。当前对外发布的OpenPGP公钥信息如下：
  - 公钥 ID：27A74824
  - 公钥指纹(Key fingerprint)：B100 0AC3 8C41 525A 19BD C087 99AD 81DF 27A7 4824
  - 用户 ID(uid)：OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

  完成信息核实后，可以对该公钥设置信任级别。
- 执行如下命令设置公钥的信任级别。

```
gpg --edit-key "OpenPGP signature key for Huawei software" trust
```

  屏幕显示类似如下信息，其中红框部分需要手工输入，“Your decision?”后输入“5”，表示“I trust ultimately”；“Do you really want to set this key to ultimate trust? (y/N)”后输入“y”。

图 3-18 设置公钥信任级别

```
ascend@szvphicpra61963:~$ gpg --edit-key "OpenPGP signature key for Huawei" trust
gpg (GnuPG) 1.4.20; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: unknown       validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: unknown       validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y

pub  2048R/27A74824  created: 2013-12-30  expires: never       usage: SC
                     trust: ultimate      validity: unknown
[ unknown] (1). OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

gpg> quit
```

步骤6 执行**quit**命令退出。

----结束

3.2.5 FAQ

3.2.5.1 Mind Studio 或 DDK 安装过程中提示 pip2 或 pip 不可用

问题描述

Mind Studio或DDK安装过程中提示pip2或pip不可用，并退出安装，提示信息如图3-19、图3-20所示。

图 3-19 pip2 不可用提示信息

```
Python 2.7.12
Traceback (most recent call last):
  File "/usr/bin/pip2", line 9, in <module>
    load_entry_point('pip==8.1.1', 'console_scripts', 'pip2')()
  File "/usr/local/lib/python2.7/dist-packages/setuptools-0.6c11-py2.7.egg/pkg_resources.py", line 318, in load_entry_point
  File "/usr/local/lib/python2.7/dist-packages/setuptools-0.6c11-py2.7.egg/pkg_resources.py", line 2221, in load_entry_point
  File "/usr/local/lib/python2.7/dist-packages/setuptools-0.6c11-py2.7.egg/pkg_resources.py", line 1954, in load
  File "/usr/lib/python2.7/dist-packages/pip/__init__.py", line 18, in <module>
    from pip.commands import get_summaries, get_similar_commands
  File "/usr/lib/python2.7/dist-packages/pip/commands/__init__.py", line 8, in <module>
    from pip.commands.freeze import FreezeCommand
  File "/usr/lib/python2.7/dist-packages/pip/commands/freeze.py", line 8, in <module>
    from pip.operations.freeze import freeze
  File "/usr/lib/python2.7/dist-packages/pip/operations/freeze.py", line 11, in <module>
    from pip._vendor.pkg_resources import RequirementParseError
ImportError: cannot import name RequirementParseError
#### Your pip2 has problem, please solve it before install MindStudio, Exit 1
~
```

图 3-20 pip 不可用提示信息

```
Traceback (most recent call last):
  File "/usr/bin/pip", line 9, in <module>
    from pip import main
ImportError: cannot import name main
```

可能原因

该问题可能是由于pip重新安装过程中，pip2没有正确更新。

解决方法 1

步骤1 **su root**切换到root用户，执行**pip list**，如果返回信息没有错误提示，说明pip可用；执行**pip2 list**提示错误，说明pip2不可用。

步骤2 root用户下删除pip2，删除命令为：**rm /usr/bin/pip2**。

步骤3 将pip2软连接到pip，命令为：**ln -s pip pip2**。

步骤4 再次执行**pip2 list**，若返回信息没有错误提示，则说明问题解决。

若pip、pip2仍不可用，则参见**解决方法2**。

----结束

解决方法 2

如果依赖安装过程中pip安装异常，请依次执行如下命令。

```
sudo apt-get remove python-pip python3-pip
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py --user
python3 get-pip.py  --user
```

3.2.5.2 软件包完整性校验返回 WARNING 或 FAIL

软件包完整性校验如果返回WARNING或 FAIL，则表示验证未通过，请参见**表3-8**处理建议解决。

版权所有 © 华为技术有限公司

表 3-8 场景举例

验证结果场景	输出信息举例	验证结果	处理建议
签名验证通过，没有异常	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: Good signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>"	PASS	NA
签名验证失败	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: BAD signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>"	FAIL	重新下载目标文件。
找不到公钥	gpg: Signature made Thu Jan 9 15:20:01 2014 CST using RSA key ID 27A74824 gpg: Can't check signature: public key not found	FAIL	重新下载公钥，请参见“配置OpenPGP公钥-配置公钥”章节。
签名验证通过，但是公钥没有被设置为完全信任	gpg: Signature made Thu Jan 9 15:29:06 2014 CST using RSA key ID 27A74824 gpg: Good signature from "OpenPGP signature key for Huawei software (created on 30th Dec,2013) <support@huawei.com>" gpg: WARNING: This key is not certified with a trusted signature! gpg: There is no indication that the signature belongs to the owner. Primary key fingerprint: B100 0AC3 8C41 525A 19BD C087 99AD 81DF 27A7 4824	WARNING	确认KeyID为27A74824后，将华为公钥设置为可信，请参见“配置OpenPGP公钥-配置公钥-验证公钥”章节。
找不到对应的源文件	gpg: no signed data gpg: can't hash datafile: No data	FAIL	重新下载目标文件。
签名已到期	gpg: Signature made 04/24/13 10:50:29 CST using RSA key ID 133B64E5 gpg: Expired signature from " OpenPGP signature test key <support@huawei.com>" gpg: Signature expired 04/25/13 10:50:29 CST	FAIL	下载更新过签名的目标文件。

版权所有 © 华为技术有限公司

验证结果场景	输出信息举例	验证结果	处理建议
签名验证通过，但是公钥已被撤销	gpg: Signature made 06/13/13 11:14:49 CST using RSA key ID 133B64E5 gpg: Good signature from " OpenPGP signature test key <support@huawei.com>" gpg: WARNING: This key has been revoked by its owner! gpg: This could mean that the signature is forged. gpg: reason for revocation: Key is no longer used gpg: revocation comment:	WARNING	下载最新公钥和更新了签名的目标文件。
源文件找不到对应的签名文件	无	WARNING	下载目标文件对应的签名文件。

3.2.6 附录

3.2.6.1 软件包说明

DDK安装包解压后包的详细内容请参见表3-9。

表 3-9 软件包说明

安装包	包内容	使用场景
MSpore_DDK-{version}-<uihost arch.os>-<host arch.os>-<device arch.os>.tar.gz 各参数解释请参见表3-10。	ddk.tar.gz	DDK套件安装包。
	install.sh	安装脚本。
	check_sha.sh	校验以上两个文件的完整性，install.sh执行过程中会自动调用该脚本进行完整性校验。

表 3-10 DDK 安装包命名规范

参数	说明
{version}	版本号
<uihost arch.os>	UIhost侧 CPU架构和操作系统及版本，如: x86_64.ubuntu16.04、x86_64.centOS7.4等

版权所有 © 华为技术有限公司

参数	说明
<host arch.os>	Host侧 CPU架构和操作系统及版本，如: x86_64.ubuntu16.04、x86_64.centOS7.4等
<device arch.os>	Device侧 CPU架构和操作系统及版本，如: aarch64.ubuntu16.04、aarch64.miniOS等 ● 开发者板形态（Atlas 200 DK）device侧架构：aarch64.ubuntu16.04 ● 非开发者板形态（Atlas 300等）device侧架构：aarch64.miniOS

3.2.6.2 开源第三方库说明

cereal

cereal是一个开源的（BSD License）、轻量级的、支持C++11特性的、仅仅包含头文件实现的、跨平台 的C++序列化库。它可以将任意的数据类型序列化成不同的表现形式，比如二进制、XML格式或JSON。当前使用版本：1.2.2

详细介绍可参考cereal官网：http://uscilab.github.io/cereal/

gflags

gflags (the Global Flags Editor)是google的一个开源的处理命令行参数的库，比getopt 更方便，更功能强大，使用c++开发，具备python接口。当前使用版本：2.2.1

详细介绍可参考gflags官网：https://github.com/gflags/gflags

glog

glog是Google的一个开源库，用于实现应用级别的logging。 它提供了一系列类似于C++流风格的logging API，以及一些预定义的宏。它有点类似于C里面的assert，但是比它具备更丰富的输出信息以及使用灵活性。

详细介绍可参考glog官网：https://github.com/google/glog

opencv

OpenCV的全称是Open Source Computer Vision Library，是一个跨平台的计算机视觉库。OpenCV可用于开发实时的图像处理、计算机视觉以及模式识别程序。当前使用版本：3.4.2

详细介绍可参考opencv官方网站：https://opencv.org/

Protobuf

Protobuf(Google Protocol Buffers)是Google提供一个具有高效的协议数据交换格式工具库；它是一种平台无关、语言无关、可扩展且轻便高效的序列化数据结构的协议，可以用于网络通信和数据存储。当前使用版本：3.5.1

详细介绍可参考google官方网站：https://developers.google.com/protocol-buffers/

4 模型转换

4.1 简介
4.2 转换Caffe/TensorFlow网络模型
4.3 AIPP配置
4.4 量化配置
4.5 FAQ

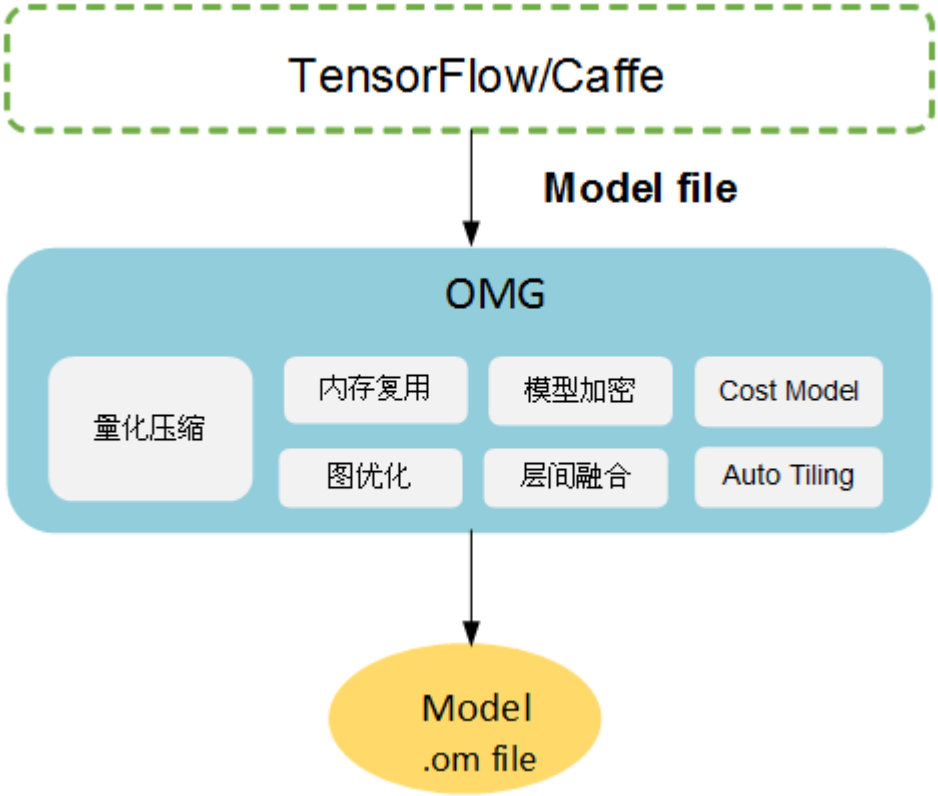
4.1 简介

本文介绍如何将开源框架的网络模型，例如Caffe、TensorFlow等框架训练好的模型，通过OMG（Offline Model Generator：离线模型生成器）将其转换成昇腾AI处理器支持的离线模型，模型转换过程中可以实现算子调度的优化、权值数据重排、量化压缩、内存使用优化等，可以脱离设备完成模型的预处理。

工具功能架构

OMG工具功能架构如**图4-1**所示。

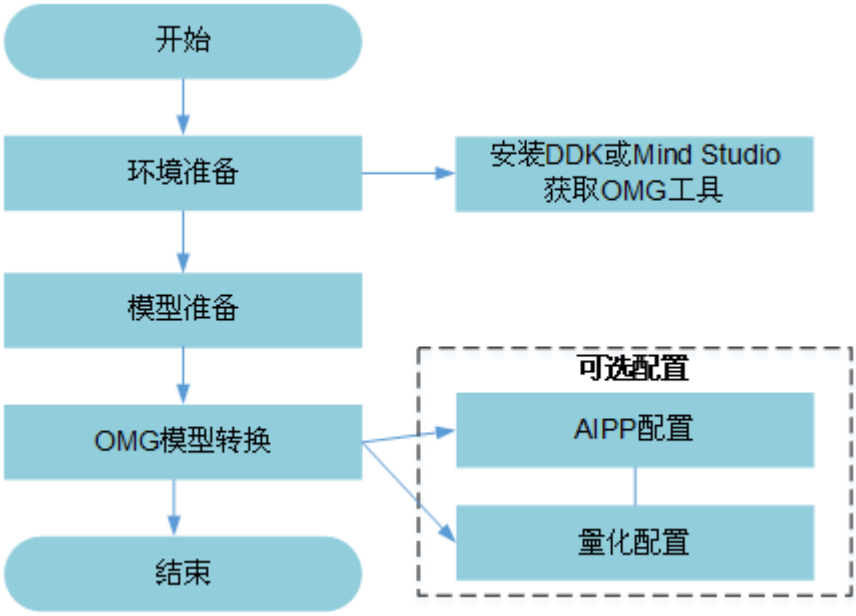
图 4-1 OMG 工具功能架构



工具运行流程

使用OMG工具进行模型转换的总体流程如图4-2所示。

图 4-2 运行流程



详细流程说明如下：

1. 使用OMG工具之前，请先在服务器安装DDK或Mind Studio，获取相关路径下的OMG工具，详细说明请参见**准备动作**中的环境准备。
2. 准备要进行转换的模型，并上传到DDK所在服务器，详细说明请参见**使用示例**。
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**或者**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。

4.2 转换 Caffe/TensorFlow 网络模型

本节介绍用户使用Caffe/Tensorflow等模型，如何通过OMG工具将其转换为昇腾AI处理器支持的离线模型。

4.2.1 约束及参数说明

约束说明

在进行模型转换前，请务必查看如下约束要求：

- 只支持原始框架类型为caffe和tensorflow的模型转换，当原始框架类型为caffe时，输入数据类型为FLOAT；当原始框架类型为tensorflow时，输入数据类型为INT32、BOOL、UINT8、FLOAT。
- 当原始框架类型为caffe时，模型文件（.prototxt）和权重文件（.caffemodel）的op name、op type必须保持名称一致（包括大小写）。
- 当原始框架类型为Caffe时，除了top与bottom相同的layer以外（例如BatchNorm，Scale，ReLU等），其他layer的top名称需要与其name名称保持一致。
- 当原始框架类型为tensorflow时，只支持FrozenGraphDef格式。
- 不支持动态shape的输入，例如：NHWC输入为[?,?,?,3]多个维度可任意指定数值。模型转换时需指定固定数值。
- 输入数据最大支持四维，转维算子（reshape、expanddim等）不能输出五维。
- 模型中的所有层算子除const算子外，输入和输出需要满足dim!=0。
- 模型转换不支持含有训练算子的模型。
- 量化（uint8）后的模型不支持模型转换。
- 模型中的算子只支持2D卷积，暂不支持3D卷积。
- 只支持《算子清单》中的算子，并需满足算子限制条件。

参数说明

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--mode	运行模式 ● 0：生成适配昇腾AI处理器的离线模型 ● 1：离线模型或模型文件转json ● 3：仅做预检，检查模型文件的内容是否合法。	否	0
--model	原始模型文件路径。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	是	不涉及
--weight	权重文件路径。 当原始模型是caffe时需要指定。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--framework	原始框架类型 ● 0：caffe ● 3：tensorflow **说明** ● 当mode为1时，该参数可选，可以指定caffe或tensorflow，不指定时默认为离线模型转json，如果指定时需要保证--om模型和--framework类型对应一致，例如：--framework=0 --om=/home/*username*/*test*/resnet18.prototxt ● 当mode为0或3时，该参数必选，可以指定caffe或tensorflow。	是	不涉及
--output	存放转换后的离线模型的路径（包含文件名），例如“out/caffe_resnet18”。 转换后的离线模型，会自动以“.om”的后缀结尾。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	是	不涉及
--encrypt_mode（预留，目前不支持）	加密模式 ● 0：加密 ● -1：不加密	否	-1

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--encrypt_key（预留，目前不支持）	用于加密的随机数文件所在的路径。 加密模式下必填。 **说明** ● 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)。 ● 测试时，您可以使用**openssl rand 32 -out ek_key**命令生成一个随机数文件。实际商用时，用户可根据实际需求选择其它工具生成随机数文件。	否	不涉及
--hardware_key（预留，目前不支持）	加密使用的ISV硬件密钥文件（这个文件是经过加密的）路径。 加密模式下必填。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--certificate（预留，目前不支持）	加密使用的ISV证书文件路径。 加密模式下必填。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--private_key（预留，目前不支持）	加密使用的ISV私钥文件路径。 加密模式下必填。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--cal_conf	量化配置文件路径。 **说明** ● 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.) ● 量化配置文件的内容示例如下： device：USE_CPU bin：150 type：JSD quantize_algo：NON_OFFSET inference_with_data_quantized：true inference_with_weight_quantized：true ● 量化配置文件的配置说明，请查看**4.4 量化配置**。	否	不涉及

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--check_report	预检结果保存文件路径。若不指定该路径，在模型转换失败或mode为3（仅做预检）时，将预检结果保存在当前路径下。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	check_result.json
--h或--help	显示帮助信息。	否	不涉及
--input_format	输入数据格式：NCHW和NHWC ● 当原始框架是tensorflow时，默认是NHWC。如果实际是NCHW的话，需要通过此参数指定NCHW。 ● 原始框架为Caffe时，只支持NCHW格式。	否	不涉及
--input_fp16_nodes	该参数与--is_output_fp16配合使用。 多网络串联时，指定下一个网络输入数据类型为fp16的输入节点名称。 例如：“node_name1; node_name2”。 举例说明：两个网络net1和net2串联，net1的输出作为net2的输入，则通过该参数指定net2网络接收net1输出数据类型为fp16的节点名称。	否	不涉及
--input_shape	模型输入数据的shape。 例如：“input_name1: n1, c1, h1, w1; input_name2: n2, c2, h2, w2”。 input_name必须是转换前的网络模型中的节点名称。 若原始模型为动态shape，例如input_name1:？,h,w,c，该参数必填。 其中“？”为batch数，表示一次处理的图片数量，需要用户根据实际情况填写，用于将动态shape的原始模型转换为固定shape的离线模型。	否	不涉及
--is_output_fp16	多网络串联时，指定前一个网络输出的数据类型是否为fp16。 例如：false,true,false,true。 举例说明：两个网络net1和net2串联，net1的输出作为net2的输入，则通过该参数指定net1的输出数据类型为fp16。	否	false

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--json	离线模型或模型文件转换为json格式文件的路径。 **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--om	当mode为1时必填。 需要转换为json格式的离线模型或模型文件的路径。例如/home/*username*/test/out/caffe_resnet18.om或/home/*username*/test/resnet18.prototxt **说明** 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)	否	不涉及
--op_name_map	算子映射配置文件路径，网络中包含DetectionOutput算子时需要指定。 例如：不同的网络中DetectionOutput算子的功能不同，可能指定DetectionOutput（Davinci模型中的算子）到如下算子的映射： ● FSRDetectionOutput：fasterrcnn网络中的算子 ● SSDDetectionOutput：SSD网络中的算子。 ● RefinedetDetectionOutput：Refinedet网络中的算子。 **说明** ● 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.) ● 算子映射配置文件的内容示例如下：DetectionOutput：SSDDetectionOutput。	否	不涉及
--out_nodes	指定输出节点。 如果不指定输出节点（算子名称），则模型的输出默认为最后一层的算子信息，某些情况下，用户想要查看某层算子参数是否合适，则需要将该层算子的参数输出，既可以在模型转换时通过该参数指定输出某层算子，模型转换后，在相应.om模型文件最后一层即可以看到指定输出算子的参数信息，如果通过.om模型文件无法查看，则可以将.om模型文件转换成json格式后查看。 例如：“node_name1:0;node_name1:1;node_name2:0”。 node_name必须是模型转换前的网络模型中的节点名称，冒号后的数字表示第几个输出，例如node_name1:0，表示节点名称为node_name1的第0个输出。	否	不涉及

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--plugin_path	自定义算子插件路径。 例如："/home/a1/b1;/home/a2/b2;/home/a3/b3" **说明** 自定义算子插件路径中可以包含多个路径，两个路径之间以分号分割，每个路径内不能包含分号，否则将导致解析得到的路径与预期不符。	否	./plugin
--target	当前仅支持设置为“mini”。 mini：量化中eltwise算子支持双输出；量化中roipooling算子支持int8输出；量化中conv算子支持混合精度。	否	mini
--ddk_version	指定自定义算子运行需要匹配的ddk环境的版本号。	否	不涉及
--net_format	指定网络算子优先选用的数据格式，ND（N<=4）和5D。仅在网络中算子的输入数据同时支持ND和5D两种格式时，指定该参数才生效。 ● ND：模型中算子按NCHW转换成通用格式。 ● 5D：模型中算子按华为自研的5维转换成华为格式。	否	不涉及

--insert_op_conf	输入预处理算子的配置文件路径，例如aipp算子。
	说明
	● 路径部分：支持大小写字母、数字，下划线；文件名部分：支持大小写字母、数字，下划线和点(.)
	● 配置文件的内容示例如下：
	aipp_op {
	aipp_mode: static
	input_format：YUV420SP_U8
	csc_switch：true
	var_reci_chn_0：0.00392157
	var_reci_chn_1：0.00392157
	var_reci_chn_2：0.00392157
	}
	● aipp配置文件的配置说明，请查看**4.3 AIPP配置**。
		否	不涉及

版权所有 © 华为技术有限公司

参数名称	参数描述	是否必选(以mode为0和3为准)	默认值
--fp16_high_prec	指定是否生成高精度FP16 Davinci模型。 ● 0：默认值，生成普通FP16 Davinci模型，推理性能更好。 ● 1：生成高精度FP16 Davinci模型，推理精度更好。 高精度当前仅支持如下算子Caffe：Convolution，Pooling，FullConnection；TensorFlow：tf.nn.conv2d，tf.nn.max_poo算子。	否	0
--output_type	网络输出数据类型： ● FP32：默认值，推荐分类网络、检测网络使用。 ● UINT8：图像超分辨率网络，推荐使用，推理性能更好。	否	FP32
--enable_l2dynamic	L2动态优化开关，该参数可能会影响网络模型的推理性能，如果性能不满足要求，可以尝试关闭该开关，验证性能影响。 ● true：是，打开L2动态优化开关。 ● false：否，关闭L2动态优化开关。	否	true

4.2.2 使用 OMG 工具转换模型

准备动作

- **环境准备**

  本手册介绍命令行方式的模型转换，如果用户想通过Mind Studio界面进行模型转换，则请参见《Ascend 310 Mind Studio基本操作》。

  本手册以DDK独立安装为例进行说明。

  如果DDK安装用户为HwHiAiUser，日志未输出到屏幕或者DDK所在服务器操作系统以及架构为Arm（aarch64），模型转换耗时较长，则可以分别参见**4.5.1 DDK安装用户为HwHiAiUser时，日志未输出到屏幕**和**4.5.2 DDK所在服务器操作系统以及架构为Arm（aarch64），模型转换耗时较长**解决。

- 以DDK安装用户将模型转换过程中使用到的模型文件（*.prototxt）、权重文件（*.caffemodel）等上传到DDK所在服务器/home/*username*/*test*/目录下。

使用示例

步骤1 以DDK安装用户登录DDK所在服务器。

步骤2 设置环境变量：

```
export LD_LIBRARY_PATH=DDK安装目录/ddk/uihost/lib
```


版权所有 © 华为技术有限公司

步骤3 进入DDK安装目录“ddk/uihost/bin”下获取omg工具。执行以下命令生成模型文件。（如下命令中使用的目录以及文件均为样例，请以实际为准）

```
./omg --model=/home/username/test/resnet18.prototxt --weight=/home/username/test/
resnet18.caffemodel --framework=0 --output=/home/username/test/out/caffe_resnet18
```

成功执行命令后，在output参数指定的路径下，可查看模型文件（如：caffe_resnet18.om）。

说明

如果您直接复制示例中的命令，由于PDF文档格式的限制，超过单行的命令，会自动换行，因此您需要手动将多行命令合并成一行，参数之间以空格分割。

模型转换过程中如果有如下错误提示信息“It is recommended to convert layers-structure to layer-structure by caffe tool”或“Type XXX unsupported”，则请参见**4.5 FAQ**解决。

步骤4 （可选）如果模型转换过程中指定了输出节点（即使用了--out_nodes参数），转换成.om模型后无法查看最后一层算子的输出信息，则可以将.om模型转换成json格式后查看，使用命令为：

```
omg --mode=1 --om=/home/username/test/caffe_resnet18.om  --json=/home/username/test/out/resnet.json
```

----结束

4.3 AIPP 配置

4.3.1 基本介绍

AIPP（AI Preprocessing）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）。

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Davinci模型，每次模型推理阶段都使用相同的AIPP参数。

  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，每次模型推理前需要在推理Engine的代码中设置动态AIPP参数值，然后在模型推理时可使用不同的AIPP参数。在推理Engine的代码中设置动态AIPP参数值涉及的API及示例请参见《Matrix API参考》中的“模型管家接口”>“AIPP配置接口”。

  如果使用动态AIPP方式，多batch使用不同的参数，体现在动态参数结构体中，每个batch可以配置不同的crop、resize等参数。关于动态参数结构体，请参见**4.3.7 动态AIPP的参数输入结构**。

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

- 对于YUV420SP_U8，根据UV交织顺序不同，YUV420SP_U8又分为YUV420SP_UV(NV12)和YUV420SP_VU(NV21)，默认为YUV420SP_UV(NV12)：
  - 若**4.3.2 配置文件模板**中rbuv_swap_switch（R通道与B通道交换开关/U通道与V通道交换开关）设置为false，则AIPP输出为YUV420SP_UV(NV12)
  - 若**4.3.2 配置文件模板**中rbuv_swap_switch设置为true，则AIPP输出为YUV420SP_UV(NV12)。

- 对于RGB888_U8，根据rbuv_swap_switch参数的取值不同，AIPP输出不同：
  - 若4.3.2 配置文件模板中rbuv_swap_switch设置为false，则AIPP输出为RGB888_U8。
  - 若4.3.2 配置文件模板中rbuv_swap_switch设置为true，则AIPP输出为BGR888_U8。

须知

- AIPP的输入格式为“YUV420SP_U8”（默认为“YUV420SP_UV”格式），若格式为“YUV420SP_VU”，请修改参数“rbuv_swap_switch”为false，否则会影响输出结果。
- 开启AIPP时，模型输入为“RGB888_U8”或“BGR888_U8”，对应不同的色域转换矩阵。
- 模型转换是否开启AIPP功能，执行推理业务时，对输入图片数据的要求：
  - 模型转换时开启AIPP：图像选择XRGB8888_U8或RGB888_U8，使用该种配置转换后的模型，在进行推理业务时，输入图片数据要求为NHWC排布。
  - 模型转换时没有开启AIPP，模型转换完毕，在进行推理业务时，输入图片数据要求为NCHW排布，因此需要用户自行把NHWC排布的原始图片数据转换为NCHW排布。

4.3.2 配置文件模板

配置文件说明如下：

```
# AIPP的配置以aipp_op开始，标识这是一个AIPP算子的配置，aipp_op支持配置多个
aipp_op {
#
# AIPP当前支持色域转换、抠图、减均值、乘系数、通道数据交换、单行模式的能力。
# 输入图片的类型仅支持UINT8格式。
# 使用此配置文件时，请将需要配置的参数去注释，并改为合适的值。
# 模板中参数值为默认值，其中input_format属性为必选属性，其余属性均为可选配置。
#========================= 全局设置（start）
==========================================================================================
================================
# aipp_mode指定了AIPP的模式，必须配置
# 类型：enum
# 取值范围：dynamic/static，dynamic 表示动态AIPP，static 表示静态AIPP
# aipp_mode:

# 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。
# 类型: 整型
# 配置范围 >= 0
# related_input_rank: 0

# input_edge_idx参数为可选，如果一个模型输入为多个算子共有，即Data算子后面跟着多个算子，配置该参数，对Data算子的不同的输出边做不同的AIPP处理。
# 类型: 整型
# 配置范围 >= 0
# input_edge_idx: 0
#========================= 全局设置（end）
==========================================================================================
================================

#========================= 动态AIPP需设置，静态AIPP无需设置（start）
==========================================================================================
=======
# 输入图像最大的size，必须大于等于原始图像的大小
```

```
# 类型：int
# max_src_image_size: 0

# 是否支持旋转，保留字段，暂不支持该功能
# 类型：bool
# 取值范围：true/false，true表示支持旋转，false表示不支持旋转
# support_rotation: false
#======================== 动态AIPP需设置，静态AIPP无需设置（end）
==========================================================================================
========

#======================== 静态AIPP需设置，动态AIPP无需设置 （start）
==========================================================================================
======
# 输入图像类型
# 类型: enum
# 取值范围：YUV420SP_U8/XRGB8888_U8/RGB888_U8/YUV400_U8
# input_format :

# 图像的宽度、高度
# 类型：uint13
# 取值范围 & 约束： [0,4096]、对于YUV420SP_U8类型的图像，要求取值是偶数
# 说明：请根据实际图片的宽、高配置src_image_size_w、src_image_size_h，若src_image_size_w、
src_image_size_h同时不设置或同时设置为0，则会取网络输入定义的w和h
# src_image_size_w :0
# src_image_size_h :0

# c_padding_value :0.0
#========= crop参数设置（配置样例请参见AIPP配置 > Crop/Padding配置说明） =========
# AIPP处理图片时是否支持抠图
# 类型：bool
# 取值范围：true/false，true表示支持，false表示不支持
# crop :false

# 抠图起始位置水平、垂直方向坐标，抠图大小为网络输入定义的w和h
# 类型：uint13
# 取值范围 & 约束： [0,4096]、对于YUV420SP_U8类型的图像，要求取值是偶数
#说明：load_start_pos_w加上网络输入定义的w需要小于等于src_image_size_w，load_start_pos_h加上网络输
入定义的h需要小于等于src_image_size_h
# load_start_pos_w :0
# load_start_pos_h :0

# 抠图后的图像size
# 类型：uint13
# 取值范围 & 约束： [0,4096]、偶数、load_start_pos_w + crop_size_w <= src_image_size_w、
load_start_pos_h + crop_size_h <= src_image_size_h
# crop_size_w :0
# crop_size_h :0


#================================= resize参数设置 ================================
# AIPP处理图片时是否支持缩放，保留字段，暂不支持该功能
# 类型：bool
# 取值范围：true/false，true表示支持，false表示不支持
resize :false

# 缩放后图像的宽度和高度，保留字段，暂不支持该功能
# 类型：uint13
# 取值范围 & 约束： [0,4096]、偶数、小于src_image_size
resize_output_w :0
resize_output_h :0


#======== padding参数设置（配置样例请参见AIPP配置 > Crop/Padding配置说明） =========
# AIPP处理图片时padding使能开关
# 类型：bool
# 取值范围：true/false，true表示支持，false表示不支持
# padding :false
```

```
# C方向的填充值，静态AIPP配置
# 类型： float16
# left_padding_size :0
# right_padding_size :0
# top_padding_size :0
# bottom_padding_size :0


#================================ rotation参数设置 ==================================
# AIPP处理图片时的旋转角度，保留字段，暂不支持该功能
# 类型：uint8
# 范围：{0, 1, 2, 3} 0不旋转，1顺时针90°，2顺时针180°，3顺时针270°
# rotation_angle :0


#========= 色域转换参数设置（配置样例请参见AIPP配置 > 色域转换配置说明） =============
# 色域转换开关，静态AIPP配置
# 类型：bool
# 取值范围：true/false，true表示开启色域转换，false表示关闭
# csc_switch :false

# 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
# 类型：bool
# 取值范围：true/false，true表示开启通道交换，false表示关闭
# rbuv_swap_switch :false

# 单行处理模式（只处理抠图后的第一行）开关
# 类型：bool
# 取值范围：true/false，true表示开启单行处理模式，false表示关闭
# single_line_mode :false

# 若色域转换开关为false，则本功能旁路。
# 若输入图片通道数为4，则忽略第一通道。
# YUV转BGR：
# | B |   | matrix_r0c0 matrix_r0c1 matrix_r0c2 | | Y - input_bias_0 |
# | G | = | matrix_r1c0 matrix_r1c1 matrix_r1c2 | | U - input_bias_1 | >> 8
# | R |   | matrix_r2c0 matrix_r2c1 matrix_r2c2 | | V - input_bias_2 |
# BGR转YUV：
# | Y |   | matrix_r0c0 matrix_r0c1 matrix_r0c2 | | B |        | output_bias_0 |
# | U | = | matrix_r1c0 matrix_r1c1 matrix_r1c2 | | G | >> 8 + | output_bias_1 |
# | V |   | matrix_r2c0 matrix_r2c1 matrix_r2c2 | | R |        | output_bias_2 |

# 3*3 CSC矩阵元素
# 类型：int16
# 取值范围：[-32768 ,32767]
# matrix_r0c0 :298
# matrix_r0c1 :516
# matrix_r0c2 :0
# matrix_r1c0 :298
# matrix_r1c1 :-100
# matrix_r1c2 :-208
# matrix_r2c0 :298
# matrix_r2c1 :0
# matrix_r2c2 :409

# RGB转YUV时的输出偏移
# 类型：uint8
# 取值范围：[0, 255]
# output_bias_0 :16
# output_bias_1 :128
# output_bias_2 :128

# YUV转RGB时的输入偏移
# 类型：uint8
# 取值范围：[0, 255]
# input_bias_0 :16
# input_bias_1 :128
# input_bias_2 :128
```

```
#============================= 减均值、乘系数设置 ================================
# 计算规则如下：
# 当uint8->uint8时，本功能旁路
# 当uint8->int8时，pixel_out_chx(i) = pixel_in_chx(i) – mean_chn_i
# 当uint8->fp16时，pixel_out_chx(i) = [pixel_in_chx(i) – mean_chn_i – min_chn_i] * var_reci_chn

# 通道n均值
# 类型：uint8
# 取值范围：[0, 255]
# mean_chn_0 :0
# mean_chn_1 :0
# mean_chn_2 :0

# 通道n最小值
# 类型：float16
# 取值范围：[-65504, 65504]
# min_chn_0 :0.0
# min_chn_1 :0.0
# min_chn_2 :0.0

# 通道n方差或(max-min)的倒数
# 类型：float16
# 取值范围：[-65504, 65504]
# var_reci_chn_0 :1.0
# var_reci_chn_1 :1.0
# var_reci_chn_2 :1.0
}
#========================= 静态AIPP需设置，动态AIPP无需设置 （end）
==================================================================================
======
```

4.3.3 色域转换配置说明

一旦确认了AIPP处理前与AIPP处理后的图片格式（即模型输入的图片格式与模型训练阶段的图片格式），即可确定色域转换相关的参数值（matrix_r*c*配置项的值是固定的，不需要调整），配置可参考如下内容。

在AIPP处理前，针对模型输入的图片或视频（各颜色编码方式，如YUV420SP_U8、XRGB8888_U8、RGB888_U8、YUV400_U8），当前给出以下两种典型场景下的色域转换配置：

- 对于JPEG图像文件格式（如后缀为jpg、jpeg、JPG、JPEG的图像文件），可以选择以下表格中“JPEG”列的色域转换配置。
- 对于视频解码后的数据，可以根据不同的彩色视频数字化标准（如BT-601、BT-709等）配置色域转换参数。

YUV420SP_U8 转 YVU444SP_U8

```
aipp_op {
    aipp_mode: static
    input_format : YUV420SP_U8
    csc_switch : false
    rbuv_swap_switch : true
}
```

YUV420SP_U8 转 RGB888_U8

表 4-1 YUV420SP_U8 转 RGB888_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : YUV420SP_U8
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_op {
	aipp_mode: static
	input_format : YUV420SP_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 298
	matrix_r0c1 : 0
	matrix_r0c2 : 409
	matrix_r1c0 : 298
	matrix_r1c1 : -100
	matrix_r1c2 : -208
	matrix_r2c0 : 298
	matrix_r2c1 : 516
	matrix_r2c2 : 0
	input_bias_0 : 16
	input_bias_1 : 128
	input_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : YUV420SP_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 298
		matrix_r0c1 : 0
		matrix_r0c2 : 459
		matrix_r1c0 : 298
		matrix_r1c1 : -55
		matrix_r1c2 : -136
		matrix_r2c0 : 298
		matrix_r2c1 : 541
		matrix_r2c2 : 0
		input_bias_0 : 16
		input_bias_1 : 128
		input_bias_2 : 128
		}

YUV420SP_U8 转 BGR888_U8

表 4-2 YUV420SP_U8 转 BGR888_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : YUV420SP_U8
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 256
matrix_r0c1 : 454
matrix_r0c2 : 0
matrix_r1c0 : 256
matrix_r1c1 : -88
matrix_r1c2 : -183
matrix_r2c0 : 256
matrix_r2c1 : 0
matrix_r2c2 : 359
input_bias_0 : 0
input_bias_1 : 128
input_bias_2 : 128
}
	aipp_op {
	aipp_mode: static
	input_format : YUV420SP_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 298
	matrix_r0c1 : 516
	matrix_r0c2 : 0
	matrix_r1c0 : 298
	matrix_r1c1 : -100
	matrix_r1c2 : -208
	matrix_r2c0 : 298
	matrix_r2c1 : 0
	matrix_r2c2 : 409
	input_bias_0 : 16
	input_bias_1 : 128
	input_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : YUV420SP_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 298
		matrix_r0c1 : 541
		matrix_r0c2 : 0
		matrix_r1c0 : 298
		matrix_r1c1 : -55
		matrix_r1c2 : -136
		matrix_r2c0 : 298
		matrix_r2c1 : 0
		matrix_r2c2 : 459
		input_bias_0 : 16
		input_bias_1 : 128
		input_bias_2 : 128
		}

YUV420SP_U8 转 GRAY

```
aipp_op {
    aipp_mode: static
    input_format : YUV420SP_U8
    csc_switch : true
    rbuv_swap_switch : false
    matrix_r0c0 : 256
    matrix_r0c1 : 0
    matrix_r0c2 : 0
```

```
    matrix_r1c0 : 0
    matrix_r1c1 : 0
    matrix_r1c2 : 0
    matrix_r2c0 : 0
    matrix_r2c1 : 0
    matrix_r2c2 : 0
    input_bias_0 : 0
    input_bias_1 : 0
    input_bias_2 : 0
}
```

XRGB8888_U8 转 YUV444SP_U8

表 4-3 XRGB8888_U8 转 YUV444SP_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : XRGB8888_U8
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 77
matrix_r0c1 : 150
matrix_r0c2 : 29
matrix_r1c0 : -43
matrix_r1c1 : -85
matrix_r1c2 : 128
matrix_r2c0 : 128
matrix_r2c1 : -107
matrix_r2c2 : -21
output_bias_0 : 0
output_bias_1 : 128
output_bias_2 : 128
}
	aipp_op {
	aipp_mode: static
	input_format : XRGB8888_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 66
	matrix_r0c1 : 129
	matrix_r0c2 : 25
	matrix_r1c0 : -38
	matrix_r1c1 : -74
	matrix_r1c2 : 112
	matrix_r2c0 : 112
	matrix_r2c1 : -94
	matrix_r2c2 : -18
	output_bias_0 : 16
	output_bias_1 : 128
	output_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : XRGB8888_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 47
		matrix_r0c1 : 157
		matrix_r0c2 : 16
		matrix_r1c0 : -26
		matrix_r1c1 : -87
		matrix_r1c2 : 112
		matrix_r2c0 : 112
		matrix_r2c1 : -102
		matrix_r2c2 : -10
		output_bias_0 : 16
		output_bias_1 : 128
		output_bias_2 : 128
		}

XRGB8888_U8 转 YVU444SP_U8

表 4-4 XRGB8888_U8 转 YVU444SP_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : XRGB8888_U8
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 77
matrix_r0c1 : 150
matrix_r0c2 : 29
matrix_r1c0 : 128
matrix_r1c1 : -107
matrix_r1c2 : -21
matrix_r2c0 : -43
matrix_r2c1 : -85
matrix_r2c2 : 128
output_bias_0 : 0
output_bias_1 : 128
output_bias_2 : 128
}
	aipp_op {
	aipp_mode: static
	input_format : XRGB8888_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 66
	matrix_r0c1 : 129
	matrix_r0c2 : 25
	matrix_r1c0 : 112
	matrix_r1c1 : -94
	matrix_r1c2 : -18
	matrix_r2c0 : -38
	matrix_r2c1 : -74
	matrix_r2c2 : 112
	output_bias_0 : 16
	output_bias_1 : 128
	output_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : XRGB8888_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 47
		matrix_r0c1 : 157
		matrix_r0c2 : 16
		matrix_r1c0 : 112
		matrix_r1c1 : -102
		matrix_r1c2 : -10
		matrix_r2c0 : -26
		matrix_r2c1 : -87
		matrix_r2c2 : 112
		output_bias_0 : 16
		output_bias_1 : 128
		output_bias_2 : 128
		}

XRGB8888_U8 转 GRAY

```
aipp_op {
    aipp_mode: static
    input_format : XRGB8888_U8
    csc_switch : true
    rbuv_swap_switch : false
    matrix_r0c0 : 76
    matrix_r0c1 : 150
    matrix_r0c2 : 30
    matrix_r1c0 : 0
    matrix_r1c1 : 0
    matrix_r1c2 : 0
    matrix_r2c0 : 0
    matrix_r2c1 : 0
    matrix_r2c2 : 0
    output_bias_0 : 0
    output_bias_1 : 0
    output_bias_2 : 0
}
```

RGB888_U8 转 BGR888_U8

```
aipp_op {
    aipp_mode: static
    input_format : RGB888_U8
    csc_switch : false
    rbuv_swap_switch : true
}
```

RGB888_U8 转 YUV444SP_U8

表 4-5 RGB888_U8 转 YUV444SP_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : RGB888_U8
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 77
matrix_r0c1 : 150
matrix_r0c2 : 29
matrix_r1c0 : -43
matrix_r1c1 : -85
matrix_r1c2 : 128
matrix_r2c0 : 128
matrix_r2c1 : -107
matrix_r2c2 : -21
output_bias_0 : 0
output_bias_1 : 128
output_bias_2 : 128
}
	aipp_op {
	aipp_mode: static
	input_format : RGB888_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 66
	matrix_r0c1 : 129
	matrix_r0c2 : 25
	matrix_r1c0 : -38
	matrix_r1c1 : -74
	matrix_r1c2 : 112
	matrix_r2c0 : 112
	matrix_r2c1 : -94
	matrix_r2c2 : -18
	output_bias_0 : 16
	output_bias_1 : 128
	output_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : RGB888_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 47
		matrix_r0c1 : 157
		matrix_r0c2 : 16
		matrix_r1c0 : -26
		matrix_r1c1 : -87
		matrix_r1c2 : 112
		matrix_r2c0 : 112
		matrix_r2c1 : -102
		matrix_r2c2 : -10
		output_bias_0 : 16
		output_bias_1 : 128
		output_bias_2 : 128
		}

RGB888_U8 转 YVU444SP_U8

表 4-6 RGB888_U8 转 YVU444SP_U8

JPEG	BT-601NARROW	BT-709NARROW

aipp_op {
aipp_mode: static
input_format : RGB888_U8
csc_switch : true
rbuv_swap_switch : false
matrix_r0c0 : 77
matrix_r0c1 : 150
matrix_r0c2 : 29
matrix_r1c0 : 128
matrix_r1c1 : -107
matrix_r1c2 : -21
matrix_r2c0 : -43
matrix_r2c1 : -85
matrix_r2c2 : 128
output_bias_0 : 0
output_bias_1 : 128
output_bias_2 : 128
}
	aipp_op {
	aipp_mode: static
	input_format : RGB888_U8
	csc_switch : true
	rbuv_swap_switch : false
	matrix_r0c0 : 66
	matrix_r0c1 : 129
	matrix_r0c2 : 25
	matrix_r1c0 : 112
	matrix_r1c1 : -94
	matrix_r1c2 : -18
	matrix_r2c0 : -38
	matrix_r2c1 : -74
	matrix_r2c2 : 112
	output_bias_0 : 16
	output_bias_1 : 128
	output_bias_2 : 128
	}
		aipp_op {
		aipp_mode: static
		input_format : RGB888_U8
		csc_switch : true
		rbuv_swap_switch : false
		matrix_r0c0 : 47
		matrix_r0c1 : 157
		matrix_r0c2 : 16
		matrix_r1c0 : 112
		matrix_r1c1 : -102
		matrix_r1c2 : -10
		matrix_r2c0 : -26
		matrix_r2c1 : -87
		matrix_r2c2 : 112
		output_bias_0 : 16
		output_bias_1 : 128
		output_bias_2 : 128
		}

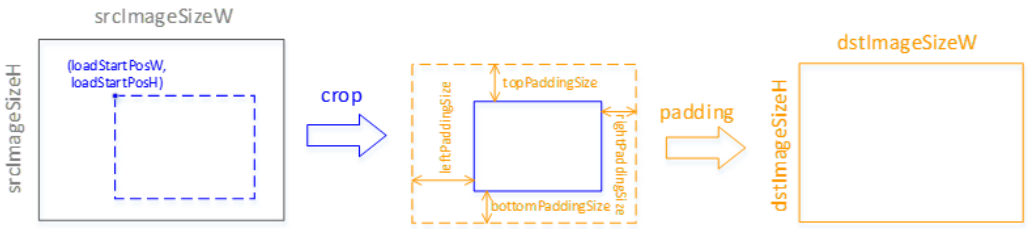
RGB888_U8 转 GRAY

```
aipp_op {
    aipp_mode: static
    input_format : RGB888_U8
    csc_switch : true
    rbuv_swap_switch : false
    matrix_r0c0 : 76
    matrix_r0c1 : 150
    matrix_r0c2 : 30
    matrix_r1c0 : 0
    matrix_r1c1 : 0
    matrix_r1c2 : 0
    matrix_r2c0 : 0
    matrix_r2c1 : 0
    matrix_r2c2 : 0
    output_bias_0 : 0
    output_bias_1 : 0
    output_bias_2 : 0
}
```

4.3.4 Crop/Padding 配置说明

目前AIPP支持Crop（抠图）、Padding两种改变图像尺寸的操作。

图 4-3 改变图像尺寸



对于YUV420SP_U8图片类型，load_start_pos_w、load_start_pos_h、crop_size_w与crop_size_h四个参数必须配置为偶数。抠图后的图像的宽、高需和网络输入定义的w和h相等。

配置样例如下：

```
aipp_op {
    aipp_mode: static
    input_format : YUV400_U8

    src_image_size_w :320
    src_image_size_h :240

    crop :true
    load_start_pos_w :10
    load_start_pos_h :20
    crop_size_w :50
    crop_size_h :60

    padding : true
    left_padding_size :10
    right_padding_size :20
    top_padding_size :10
    bottom_padding_size :20
}
```

4.3.5 根据模型输入指定 AIPP 配置说明

AIPP配置文件支持定义多组AIPP配置，对不同的模型输入进行不同的AIPP处理，配置多组AIPP参数时，将一组AIPP配置放到一个aipp_op配置项里，举例如下：

```
aipp_op {
        related_input_rank: 0
        aipp_mode : dynamic
        max_src_image_size: 60000
}
aipp_op {
        related_input_rank : 1
        aipp_mode : dynamic
        max_src_image_size: 80000
}
```

上述配置定义了两组AIPP参数，分别对模型第一个和第二个输入进行AIPP处理，AIPP的模式为动态AIPP，两个输入可以接受的最大的图片大小分别是60000字节和80000字节。

4.3.6 AIPP 对模型输入大小的校验说明

如果有配置AIPP，无论静态AIPP还是动态AIPP，最终生成的Davinci模型的接收的图片的大小（即input_size）均会被Crop、Padding等操作影响。在模型推理阶段，OME会对传入的图片的大小进行校验，如果是动态AIPP，要求模型推理时传入的图片的大小小于等于动态AIPP配置的max_src_image_size大小；如果是静态AIPP，要求模型推理时传入的图片的大小与**表4-7**中的计算的input_size相等。

对于静态AIPP，假设模型的Batch数量为N，图片的宽为src_image_size_w，高为src_image_size_h，最后模型输入的Size的计算公式如**表4-7**所示。

表 4-7 input_size 校验公式

input_format	input_size
YUV400_U8	N * src_image_size_w * src_image_size_h
YUV420SP_U8	N * src_image_size_w * src_image_size_h * 1.5
XRGB8888_U8	N * src_image_size_w * src_image_size_h * 4
RGB888_U8	N * src_image_size_w * src_image_size_h * 3

生成带动态AIPP的模型时，OMG会为动态AIPP添加模型输入。新增动态AIPP输入的input_size计算公式为：

sizeof(kAippDynamicPara) + (batch_count - 1) * sizeof(kAippDynamicBatchPara)

4.3.7 动态 AIPP 的参数输入结构

```
typedef struct tagAippDynamicBatchPara
{
    int8_t cropSwitch;              //crop switch
    int8_t scfSwitch;               //resize switch
    int8_t paddingSwitch;   // 0: unable padding,
                            // 1: padding config value,sfr_filling_hblank_ch0 ~    sfr_filling_hblank_ch2
                            // 2: padding source picture data, single row/collumn copy
                            // 3: padding source picture data, block copy
                            // 4: padding source picture data, mirror copy
    int8_t rotateSwitch;  //rotate switch，0: non-ratate，1: ratate 90°clockwise，2: ratate 180°clockwise，3:
ratate 270° clockwise
    int8_t reserve[4];
    int32_t cropStartPosW;          //the start horizontal position of cropping
    int32_t cropStartPosH;          //the start vertical position of cropping
    int32_t cropSizeW;              //crop width
    int32_t cropSizeH;              //crop height
    int32_t scfInputSizeW;          //input width of scf
    int32_t scfInputSizeH;          //input height of scf
    int32_t scfOutputSizeW;         //output width of scf
    int32_t scfOutputSizeH;         //output height of scf
    int32_t paddingSizeTop;         //top padding size
    int32_t paddingSizeBottom;      //bottom padding size
    int32_t paddingSizeLeft;        //left padding size
    int32_t paddingSizeRight;       //right padding size
    int16_t dtcPixelMeanChn0;       //mean value of channel 0
    int16_t dtcPixelMeanChn1;       //mean value of channel 1
    int16_t dtcPixelMeanChn2;       //mean value of channel 2
    int16_t dtcPixelMeanChn3;       //mean value of channel 3
#ifndef DAVINCI_TINY
    uint16_t dtcPixelMinChn0;       //min value of channel 0
    uint16_t dtcPixelMinChn1;       //min value of channel 1
    uint16_t dtcPixelMinChn2;       //min value of channel 2
    uint16_t dtcPixelMinChn3;       //min value of channel 3
    uint16_t dtcPixelVarReciChn0;   //sfr_dtc_pixel_variance_reci_ch0
    uint16_t dtcPixelVarReciChn1;   //sfr_dtc_pixel_variance_reci_ch1
    uint16_t dtcPixelVarReciChn2;   //sfr_dtc_pixel_variance_reci_ch2
    uint16_t dtcPixelVarReciChn3;   //sfr_dtc_pixel_variance_reci_ch3
    int8_t reserve1[16];            //32B assign, for ub copy
#endif
}kAippDynamicBatchPara;
typedef struct tagAippDynamicPara
{
    uint8_t inputFormat;        //input format：YUV420SP_U8/XRGB8888_U8/RGB888_U8
    //uint8_t outDataType; //output data type: CC_DATA_HALF,CC_DATA_INT8, CC_DATA_UINT8
```

```
    int8_t cscSwitch;              //csc switch
    int8_t rbuvSwapSwitch;         //rb/ub swap switch
    int8_t axSwapSwitch;           //RGBA->ARGB, YUVA->AYUV swap switch
    int8_t batchNum;               //batch parameter number
    int8_t reserve1[3];
    int32_t srcImageSizeW;         //source image width
    int32_t srcImageSizeH;         //source image height
    int16_t cscMatrixR0C0;         //csc_matrix_r0_c0
    int16_t cscMatrixR0C1;         //csc_matrix_r0_c1
    int16_t cscMatrixR0C2;         //csc_matrix_r0_c2
    int16_t cscMatrixR1C0;         //csc_matrix_r1_c0
    int16_t cscMatrixR1C1;         //csc_matrix_r1_c1
    int16_t cscMatrixR1C2;         //csc_matrix_r1_c2
    int16_t cscMatrixR2C0;         //csc_matrix_r2_c0
    int16_t cscMatrixR2C1;         //csc_matrix_r2_c1
    int16_t cscMatrixR2C2;         //csc_matrix_r2_c2
    int16_t reserve2[3];
    uint8_t cscOutputBiasR0;   //output bias for RGB to YUV, element of row 0, unsigned number
    uint8_t cscOutputBiasR1;   //output bias for RGB to YUV, element of row 1, unsigned number
    uint8_t cscOutputBiasR2;   //output bias for RGB to YUV, element of row 2, unsigned number
    uint8_t cscInputBiasR0;    //input bias for YUV to RGB, element of row 0, unsigned number
    uint8_t cscInputBiasR1;    //input bias for YUV to RGB, element of row 1, unsigned number
    uint8_t cscInputBiasR2;    //input bias for YUV to RGB, element of row 2, unsigned number
    uint8_t reserve3[2];
    int8_t reserve4[16];           //32B assign, for ub copy
    kAippDynamicBatchPara aippBatchPara;  //allow transfer several batch para.
} kAippDynamicPara;
```

4.4 量化配置

4.4.1 基本介绍

此处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。

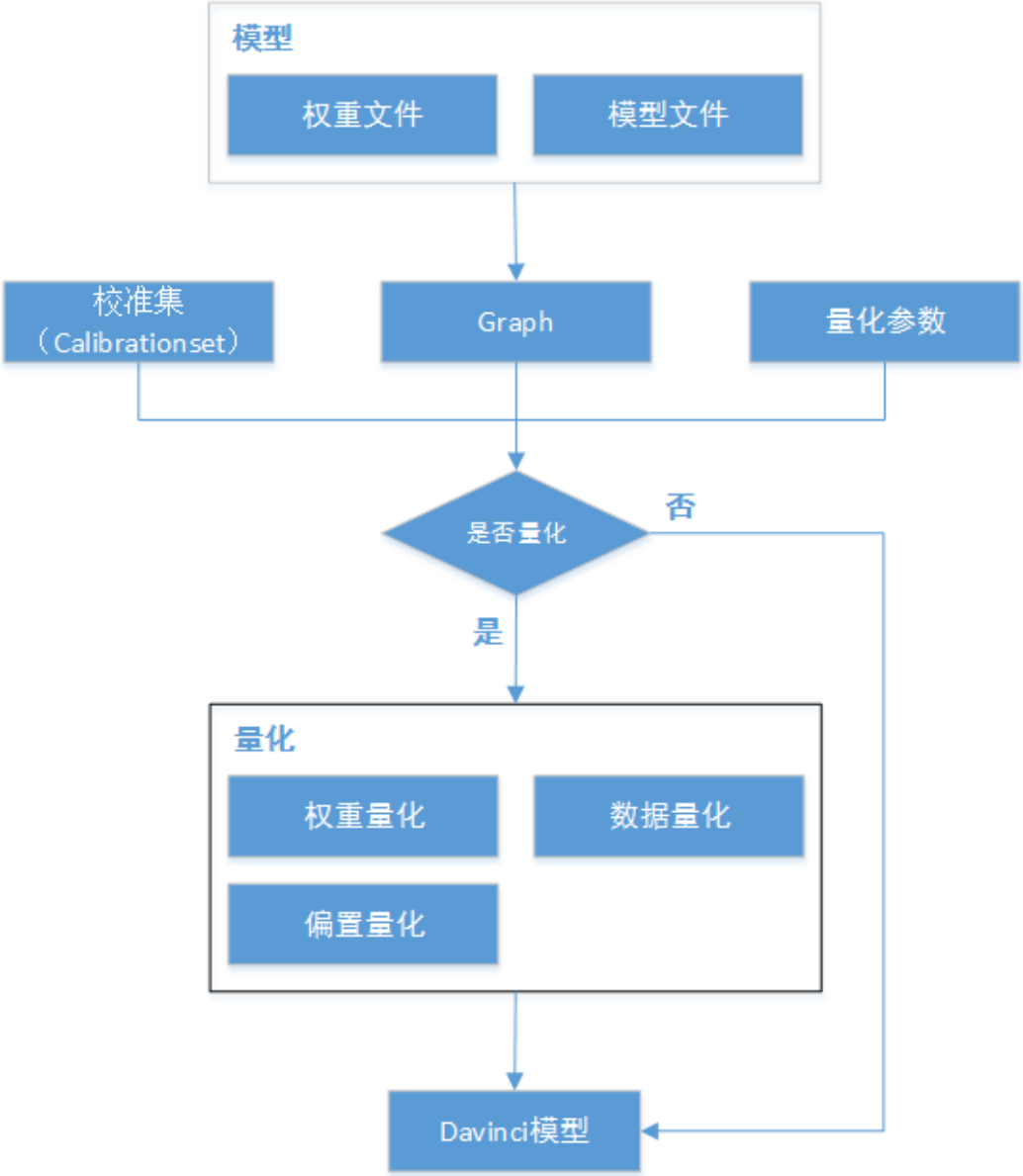
当前支持Convolution、Full Connection、ConvolutionDepthwise三种类型算子的量化，包括权重、偏置、数据量化。量化模式分为：无offset、数据offset。

量化后的权重、偏置在Davinci模型生成阶段即被保存在Davinci模型中，在模型推理阶段使用量化后权重、偏置对数据进行计算。

各概念的说明如下：

- 无offset：权重和数据都采用无偏移模式。
- 数据offset：数据采用有偏移模式，权重采用无偏移模式。
- 权重量化：根据量化算法对权重文件进行Int8量化。若是有offset量化模式，则输出Int8权重、scale和offset；若是无offset量化模式，则输出Int8权重和scale。当前权重量化仅支持无offset量化模式。
- 数据量化：经过有限输入（校准集，用于训练量化参数、保证精度）进行推理执行，根据频率统计和散度计算算法等，若是数据offset量化模式，则计算出量化数据的scale和offset；如果无offset量化模式，则计算出量化数据的scale。
- 偏置量化：根据权重量化的scale、数据量化的scale，将FP32偏置数据量化为Int32输出。

图 4-4 量化处理流程



4.4.2 配置文件模板

量化支持网络单输入或多输入的场景。使用此配置文件时，可以使用配置文件模板中默认值，也可以根据实际情况将需要的参数改成合适的值。另外，配置文件中的参数分为必选参数和可选参数，必选参数您必须配置对应的参数值，可选参数您可以不配置。配置文件中各参数的含义、是否必选请参见**4.4.3 参数配置说明**。

配置文件中的内容必须是使用英文格式。

网络单输入场景

输入为图片，通过OMG模型转换命令指定量化配置，配置文件举例如下。

```
device:USE_CPU
quantize_algo:HALF_OFFSET
weight_type:VECTOR_TYPE
```

```
bin:150
type:KL
inference_with_data_quantized:false
inference_with_weight_quantized:true
super_parameter:
{
min_percentile:PERCENTILE_HIGH
max_percentile:PERCENTILE_MID
start_ratio:0.7
end_ratio:1.3
step_ratio:0.01
}
exclude_op:'fc1000'
batch_count:50
preprocess_parameter:
{
input_type:IMAGE
image_format:BGR
input_file_path:'calibration/image_set'
mean_value:104.0
mean_value:117.0
mean_value:123.0
standard_deviation:1.0
}
```

网络多输入场景

第一个输入为图片，第二个输入为二进制文件，通过OMG模型转换命令指定量化配置，配置文件举例如下。

```
device:USE_CPU
quantize_algo:HALF_OFFSET
weight_type:VECTOR_TYPE
bin:150
type:KL
inference_with_data_quantized:false
inference_with_weight_quantized:true
super_parameter:
{
min_percentile:PERCENTILE_HIGH
max_percentile:PERCENTILE_MID
start_ratio:0.7
end_ratio:1.3
step_ratio:0.01
}
exclude_op:'fc1000'
batch_count:50
preprocess_parameter:
{
input_type:IMAGE
image_format:BGR
input_file_path:'calibration/image_set'
mean_value:104.0
mean_value:117.0
mean_value:123.0
standard_deviation:1.0
}
preprocess_parameter:
{
input_type:BINARY
input_file_path:'calibration/img_info.bin'
}
```

4.4.3 参数配置说明

表 4-8 device 参数说明

作用	推理模式
类型	enum
取值范围	USE_CPU
参数说明	USE_CPU：使用CPU做推理。
推荐配置	USE_CPU
必选或可选	必选

表 4-9 quantize_algo 参数说明

作用	量化模式
类型	enum
取值范围	● NON_OFFSET ● HALF_OFFSET
参数说明	● NON_OFFSET：权重和数据都采用无偏移模式 ● HALF_OFFSET：数据采用有偏移模式，权重采用无偏移模式 根据权重和数据在量化时是否有偏移，量化的映射公式有两种： ● 有偏移，公式为：q_uint8 = round(d_float/scale)-offset ● 无偏移，公式为：q_int8 = round(d_float/scale)
推荐配置	HALF_OFFSET
必选或可选	必选

表 4-10 weight_type 参数说明

作用	权重量化模式
类型	enum
取值范围	● VECTOR_TYPE ● SCALAR_TYPE
参数说明	对于卷积算子，可能有多个卷积核，多个对应的量化参数可能不一样。 ● VECTOR_TYPE：表示一个卷积核对应一组量化参数 ● SCALAR_TYPE：表示多个卷积核使用同一组量化参数

推荐配置	VECTOR_TYPE
必选或可选	必选

表 4-11 preprocess_parameter 参数说明

作用	预处理及输入相关参数
类型	Struct
取值范围	无。
参数说明	指定预处理及输入相关参数。preprocess_parameter参数的数量与网络的输入算子数量必须相同。 在preprocess_parameter内部包含如下参数： ● input_type ● image_format ● input_file_path ● mean_value ● standard_deviation
推荐配置	无
必选或可选	必选

表 4-12 input_type 参数说明

作用	输入数据类型
类型	enum
取值范围	● IMAGE ● BINARY
参数说明	指定输入类型，当前支持图片（IMAGE）和二进制文件（BINARY）两种格式。 ● IMAGE：图片格式。 支持的图片格式包括：".bmp", ".dib", ".jpeg", ".jpg", ".jpe", ".jp2", ".png", ".webp",".pbm", ".pgm", ".ppm", ".sr", ".ras", ".tiff", ".tif",".BMP", ".DIB", ".JPEG", ".JPG", ".JPE", ".JP2", ".PNG", ".WEBP",".PBM", ".PGM", ".PPM", ".SR", ".RAS", ".TIFF", ".TIF"。 ● BINARY：二进制文件（Binary格式）格式，请参见**表4-13**。对于非四维（num、channels、height、width）的数据，需要补齐到四维，补齐的维度值为1。
推荐配置	无

必选或可选	必选

表 4-13 二进制文件格式

文件头/数据	地址偏移	type	value	description
文件头（共20字节）	0000	32bit int	510	magic number 当magic number = 510，用来校验文件的合法性。
	0004	32bit int	50	input num
	0008	32bit int	3	input channels
	0012	32bit int	28	input height
	0016	32bit int	28	input width
数据	…	float	126	数据数量等于num*channels*height*width

表 4-14 input_file_path 参数说明

作用	输入数据地址，即校准集图片所在的路径
类型	string
取值范围	无
参数说明	指定量化输入的目录或文件。路径不要有中文、特殊字符（包括**|;&$><`**）以及空格。 根据实际网络的输入配置。 ● 如果是图片，指定到目录 ● 如果是二进制文件，指定到文件
推荐配置	建议使用与应用场景相关的图片或二进制文件。
必选或可选	必选

表 4-15 image_format 参数说明

作用	图像输入数据三通道排序方式

类型	enum
取值范围	● BGR ● RGB
参数说明	模型训练时候的图片三通道排序格式。 当input_type参数值为IMAGE时需要配置该值。该参数根据实际网络训练时候的输入通道排序方式确定，通常网络训练时候的排序方式是BGR。
推荐配置	BGR
必选或可选	必选

表 4-16 mean_value 参数说明

作用	图像预处理的均值
类型	float
取值范围	[0, 255.0]
参数说明	图片预处理单个通道的均值。 当input_type参数值为IMAGE时需要配置该值。多个通道需要配置多个该参数，通常图片有RGB三个通道，则需要配置三个该参数，如下： ● mean_value：104.0 ● mean_value：117.0 ● mean_value：123.0
推荐配置	无
必选或可选	必选

表 4-17 standard_deviation 参数说明

作用	图像预处理的均方差
类型	float
取值范围	[0,FLOAT_MAX]
参数说明	图片预处理的方差。 当input_type参数值为IMAGE时需要配置该值。多个通道使用统一方差。 如果输入范围大于float型能表示的范围，模型量化精度不能保证。 如果0<=standard_deviation<=0.00001，standard_deviation取1.0。

推荐配置	1.0（图片取值区间大小不发生变化）或者255.0（对于图片区间大小为[0, 255]的场景，可将值压缩到[0, 1.0]）。
必选或可选	必选

表 4-18 bin 参数说明

作用	数据映射直方图范围
类型	uint32
取值范围	[0,1000]
参数说明	数据直方图统计的范围。 在计算散度过程中需要统计数据直方图，该参数的值决定了直方图统计的最大值。如果不配置或者配置为0，则使用默认配置150。
推荐配置	100/150/200/250。
必选或可选	可选

表 4-19 type 参数说明

作用	散度计算指标类型
类型	enum
取值范围	● KL ● SYMKL ● JSD
参数说明	● KL：Kullback-Leibler Divergence ● SYMKL：Symmetric Kullback-Leibler Divergence ● JSD：Jensen-Shannon Divergence。 不同的散度类型对应的计算方式不一样。默认为KL。
推荐配置	无
必选或可选	可选

表 4-20 inference_with_data_quantized 参数说明

作用	推理过程中是否使用量化后的输入数据
类型	bool

取值范围	● true ● false
参数说明	控制推理过程中的输入数据是否经过量化。 该参数开启，模拟了输入数据量化的过程。默认值为false。
推荐配置	false
必选或可选	可选

表 4-21 inference_with_weight_quantized 参数说明

作用	推理过程中是否使用量化后的权重数据
类型	bool
取值范围	● true ● false
参数说明	控制推理过程中的权重数据是否经过量化。 该参数开启，模拟了权重数据量化反量化的过程。默认值为true。
推荐配置	true
必选或可选	可选

表 4-22 super_parameter 参数说明

作用	搜索相关参数
类型	Struct
取值范围	无
参数说明	搜索相关参数。 super_parameter参数内部包含如下参数： ● min_percentile ● max_percentile ● start_ratio ● end_ratio ● step_ratio
推荐配置	无
必选或可选	可选

表 4-23 min_percentile 参数说明

作用	最小值搜索位置
类型	enum
取值范围	● PERCENTILE_HIGH ● PERCENTILE_MID ● PERCENTILE_LOW
参数说明	在从小到大排序的一组数中，决定取第多少小的数，比如有100个数，1.0表示取第100-100*1.0=0，对应的就是第一个小的数。 ● PERCENTILE_HIGH：1.0 ● PERCENTILE_MID：0.99999 ● PERCENTILE_LOW：0.9999
推荐配置	PERCENTILE_HIGH
必选或可选	可选

表 4-24 max_percentile 参数说明

作用	最大值搜索位置
类型	enum
取值范围	● PERCENTILE_HIGH ● PERCENTILE_MID ● PERCENTILE_LOW
参数说明	在从大到小排序的一组数中，决定取第多少大的数，比如有100个数，1.0表示取第100-100*1.0=0，对应的就是第一个大的数。 ● PERCENTILE_HIGH：1.0 ● PERCENTILE_MID：0.99999 ● PERCENTILE_LOW：0.9999
推荐配置	PERCENTILE_MID或PERCENTILE_HIGH
必选或可选	可选

表 4-25 start_ratio、end_ratio、step_ratio 参数说明

作用	参数说明
类型	float
取值范围	end_ratio>start_ratio>0 && step_ratio>0（如果输入范围大于float型能表示的范围，模型量化精度不能保证）

参数说明	● start_ratio：决定搜索开始的位置 ● end_ratio：决定搜索结束的位置 ● step_ratio：决定搜索步长 在算法中找到d_max/d_min之后，会根据此参数，取d_max/d_min前后多少范围之内的数，然后根据step_ratio决定，每次增加的步长，说明参考如下： 以d_max =100，start_ratio=0.8，end_ratio=1.2，step_ratio=0.01为例，其定义的d_max搜索空间为从100*0.8=80到100*1.2=120的范围，每次步进100*0.01=1，一共41个d_max搜索值。
推荐配置	推荐配置有两组 ● start_ratio:0.7 end_ratio:1.3 step_ratio:0.01 ● start_ratio:0.3 end_ratio:1.7 step_ratio:0.01
必选或可选	可选

表 4-26 batch_count 参数说明

作用	量化校准集图片处理的图片数量
类型	uint32
取值范围	[0,UINT32_MAX)
参数说明	决定量化的读取的校准集图片数量。 如果不配置或者配置为0，那么会把校准集所有图片都作为校准集数据。如果配置了大于0的数，那么会根据校准集路径下图片总数和该参数取较小值作为校准集的图片实际数量。建议校准集图片数量不超过500张。
推荐配置	50
必选或可选	可选

表 4-27 exclude_op 参数说明

作用	量化算子黑名单
类型	string
取值范围	算子名称

参数说明	配置该参数，算子不进行量化。 ● 只支持Convolution、Full Connection、ConvolutionDepthwise算子； ● 多个算子，需要配置多个该参数，每个参数一行，如：exclude_op:'aaa' exclude_op:'bbb'
推荐配置	无
必选或可选	可选

4.4.4 参数调优说明

按照模型转换时设置的量化参数对权重、偏置、数据进行量化，若精度不满足要求，您可以按照如下步骤调整量化参数值。

1. 按照**4.4.2 配置文件模板**中的默认参数值进行校准，完成模型数据和参数的量化。
2. 若按照**1**中的量化配置进行量化后，精度满足要求，则调参结束，否则继续**3**。
3. 手动调整以下量化参数后，再进行量化。
   a. 修改搜索范围参数。

   表 4-28 搜索范围参数

修改前	修改后
start_ratio:0.7 end_ratio:1.3 step_ratio:0.01	start_ratio:0.3 end_ratio:1.7 step_ratio:0.01

   b. 将bin参数值改大，从150改成200或者250。
   c. 如果是检测类网络，建议将max_percentile参数值设置为PERCENTILE_MID（表示0.99999）。
4. 若按照**3**中的量化配置进行量化后，精度满足要求，则调参结束，否则继续**5**。
5. 按照量化参数的可选配置项，遍历**表4-29**中的参数，并按“配置说明”配置参数（可根据calibration时间要求，调整搜索空间大小）。

   表 4-29 参数配置

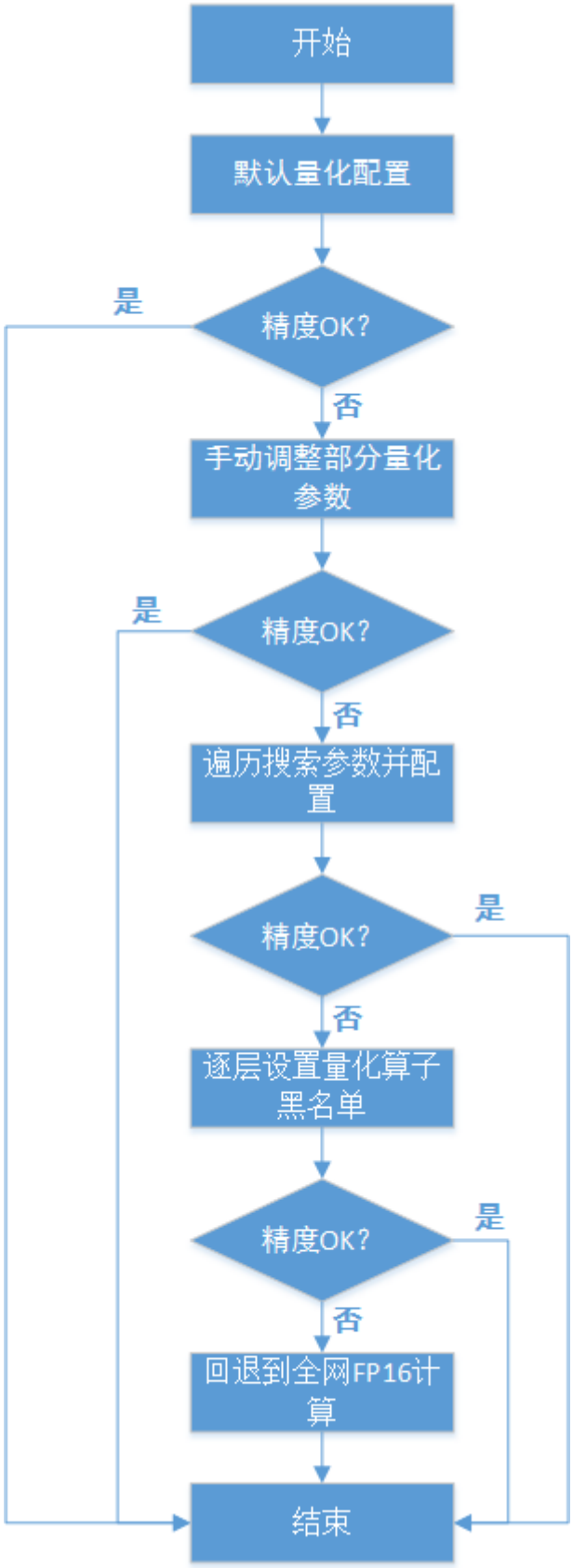
参数	配置说明
quantize_algo	配置为HALF_OFFSET，表示数据采用有偏移模式，权重采用无偏移模式。
weight_type	配置为VECTOR_TYPE，表示对于卷积算子，可能有多个卷积核的情况下，一个卷积核对应一组量化参数。

参数	配置说明
bin	在计算散度过程中需要统计数据直方图，该参数的值决定了直方图统计的最大值。如果不配置或者配置为0，则使用默认配置150。建议选100, 150, 200这三档。
type	不同的散度类型对应的计算方式不一样。默认为KL。 ● KL：Kullback-Leibler Divergence； ● SYMKL：Symmetric Kullback-Leibler Divergence； ● JSD：Jensen-Shannon Divergence。
● start_ratio ● end_ratio ● step_ratio	start_ratio决定搜索开始的位置，end_ratio决定搜索结束的位置，step_ratio决定搜索步长。 推荐配置有两组 ● start_ratio:0.7 end_ratio:1.3 step_ratio:0.01 ● start_ratio:0.3 end_ratio:1.7 step_ratio:0.01
max_percentile	在从大到小排序的一组数中，决定取第多少大的数。 建议配置为1.0, 0.99999这两档。
batch_count	决定量化的读取的校准集图片数量。 可配置为大于1的整数，建议选10, 20, 50这几档。

6. 若按照5中的量化配置进行量化后，精度满足要求，则调参结束，否则继续7。
7. 通过exclude_op参数设置量化算子黑名单，从首层开始逐层将算子设置成算子黑名单，设置成算子黑名单的算子不进行量化，通过排除法，判断是哪个算子的量化对精度有影响。

   除设置到量化算子黑名单的算子不进行量化，其它算子默认进行量化，这时会存在int8计算和FP16计算混合的情况。
8. 若按照7中的量化配置进行量化后，精度满足要求，则调参结束，否则表明量化对精度没有影响，无需设置量化，去除量化配置，退回全网FP16的计算。

图 4-5 调参流程



版权所有 © 华为技术有限公司

4.5 FAQ

4.5.1 DDK 安装用户为 HwHiAiUser 时，日志未输出到屏幕

DDK安装用户为HwHiAiUser，该场景下日志默认输入到host侧/var/dlog目录中，也可以通过设置如下环境变量，使日志直接输出到屏幕上。

```
export  SLOG_PRINT_TO_STDOUT=1
```

若DDK安装用户非HwHiAiUser，该场景下日志信息会输出到屏幕上。

4.5.2 DDK 所在服务器操作系统以及架构为 Arm（aarch64），模型转换耗时较长

若DDK所在服务器操作系统以及架构为Arm（aarch64），如果模型转换的耗时较长，可以使用numactl工具指定CPU核后进行模型转换，步骤如下：

1. 以DDK安装用户登录DDK所在服务器，执行**su root**命令切换到root用户。
2. 确保Host侧服务器已连接网络后，执行以下命令安装numactl工具。
   ```
   yum -y install numactl
   ```
3. 切换到DDK安装用户执行如下命令，通过**numactl -C**指定编号16到31的CPU核来处理模型转换操作。

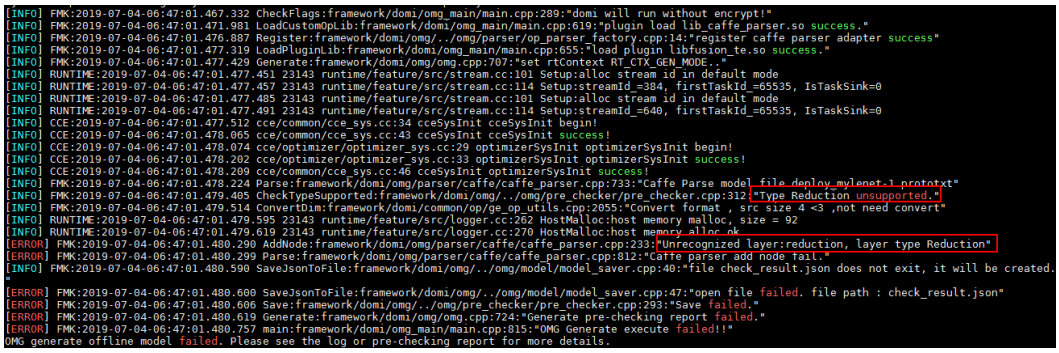
   此处建议指定编号16到31的CPU核，处理性能更好，用户也可以根据实际情况修改。
   ```
   numactl -C 16-31 --localalloc omg <args>
   ```

4.5.3 模型转换时提示 Unrecognized layer:xxx, layer type xxx 错误

问题描述

使用omg命令转换模型，模型转换失败时，提示"Type XXX unsupported"、"Unrecognized layer:xxx, layer type XXX"等信息，如图4-6所示。

图 4-6 模型转换失败提示信息



说明

图4-6中的提示信息是以Reduction类型的算子为例，请以实际使用的模型为准。

解决方法

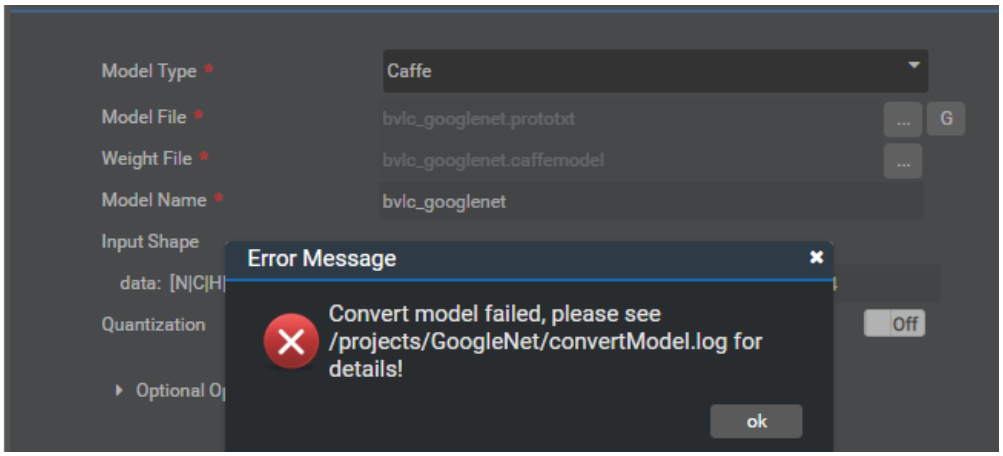
产生此错误原因是原始模型中有Ascend 310不识别的算子，需要用户根据《TE自定义算子开发指导》开发不识别的算子、开发算子插件、加载算子插件转换模型。

4.5.4 模型转换时提示 It is recommended to convert layers-structure to layer-structure by caffe tool 错误

问题描述

从https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet下载了googlenet的prototxt与caffemodel，导入时失败，如图4-7所示。

图 4-7 模型转化失败



“convertModel.log”日志报错如下：

```
[ERROR] FMK:2018-12-22-23:47:48.147.318 ConvertNetParameter:framework/domi/omg/parser/caffe/caffe_parser.cpp:776:"The weight file is consisted of layers-structure which is deprecated in caffe and unsupport in OMG. It is recommended to convert layers-structure to layer-structure by caffe tool. Error Code:0xFFFFFFFF(failed)"
```

解决方法

产生此错误原因为模型文件版本过低，需要用Caffe提供的工具将模型prototxt文件与caffemodel文件升级为最新版本。Caffe工具可从Link获取。

步骤1 用户自行下载Caffe的upgrade_net_proto_text工具与upgrade_net_proto_binary工具至Mind Studio所在服务器任一目录。

步骤2 参见Link中的说明自行编译获得**upgrade_net_proto_text**和**upgrade_net_proto_binary**工具。

步骤3 使用upgrade_net_proto_text工具升级prototxt。

upgrade_net_proto_text *model_old.prototxt model_new.prototxt*

步骤4 使用upgrade_net_proto_binary升级caffemodel。

upgrade_net_proto_binary *model_old.caffemodel model.new.caffemodel*

步骤5 使用转换后的protoxt文件与caffemodel文件重新进行模型导入。

----结束

5 算子开发

5.1 开发前必读
5.2 算子开发注意事项
5.3 环境准备
5.4 开发流程总览
5.5 配置环境变量
5.6 创建自定义算子开发工程
5.7 自定义算子开发
5.8 caffe.proto文件算子定义（可选）
5.9 自定义算子插件开发
5.10 加载插件转换模型
5.11 算子插件开发函数参考
5.12 算子清单

5.1 开发前必读

5.1.1 概述

Ascend 310中提供了深度学习框架Framework，可以将Caffe、Tensorflow开源框架模型转换成Ascend 310支持的模型，在进行模型转换时，如果模型中的算子在系统内置的算子库中未实现，则转换过程会报错。这时，未实现的算子就需要用户进行自定义，自定义算子可加入到自定义算子库中，使得模型转换过程可以正常进行。用户也可以优化当前模型中已有算子，并将优化后的算子加入到自定义算子库中，并加载优化后的算子进行模型转换。

Ascend 310提供了TE（Tensor Engine）算子开发框架，可以开发自定义算子。TE是基于TVM（Tensor Virtual Machine）的自定义算子开发框架，提供了基于Python语法的DSL语言供开发者开发编写自定义算子。TE自定义算子可运行于AI CPU和AI Core。

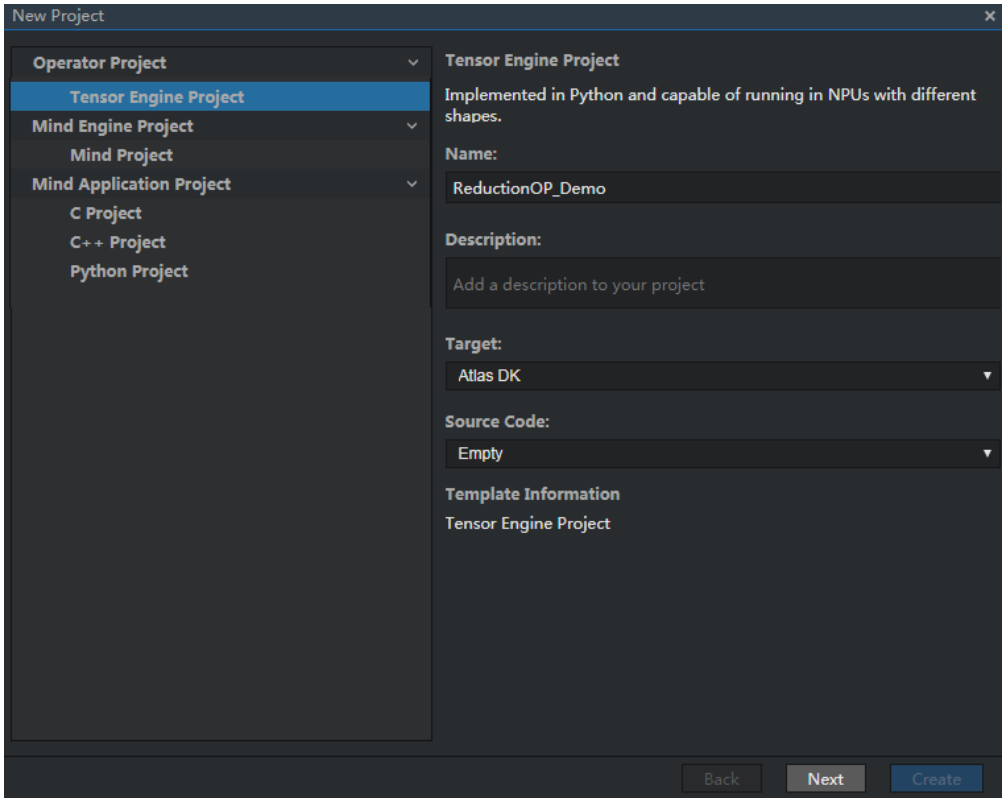
本文以Caffe网络模型Lenet-5内置Reduction算子的扩展自定义caffe_reduction_layer算子为例进行讲解，介绍如何通过后台命令行的方式进行TE自定义算子的开发、算子的验证、算子的注册及如何加载用户自定义的算子进行模型转化。

Ascend 310支持的TE自定义算子开发方式如**表5-1**所示，本文仅介绍“命令行开发方式”。

表 5-1 Ascend 310 支持的 TE 自定义算子开发方式

方式	工具依赖	描述	编译方式	对应文档
命令行开发方式	不依赖Mind Studio工具，仅安装DDK的情况就可进行TE自定义算子开发。 **说明** DDK（Device Development Kit）为用户提供基于NPU的数字开发者套件，内含工程代码样例，并集成了相关依赖库和头文件，用户可以通过makefile脚本编译相应的工程文件。	无UI界面，无自动生成的框架代码，所有算子实现代码及算子插件代码需要用户自行开发，或者基于DDK中代码样例进行编辑修改。	基于DDK代码样例编写Makefile编译脚本。	《TE自定义算子开发指导》
Mind Studio 开发方式	依赖Mind Studio工具，操作入口如下图所示。	框架代码基于工程创建时导入的模型文件及选择的算子自动生成，用户只需要重点关注算子的实现代码。	Makefile编译脚本通过Mind Studio界面配置生成，无需编写。	《Ascend 310 TE自定义算子开发指导（Mind Studio方式）》

图 5-1 Tensor Engine Project 创建



5.1.2 目标读者

本文档适用于使用Ascend 310进行算子开发的人员，通过本文档您可以达成：

- 通过本文档提供的算子开发样例进行算子开发，并将此算子集成到整网络中进行端到端的应用程序的运行，端到端了解自定义算子的开发流程。
- 参考文档中的Reduction算子样例，学习算子开发及算子插件开发的原理，能够进行其他自定义算子的开发。

掌握以下经验和技能可以更好地理解本文档：

- 具备Python/C++/C语言程序开发能力
- 理解数学表达式
- 对机器学习、深度学习有一定的了解
- 对Ascend 310架构有一定了解
- 对TVM及开源Caffe框架有一定的了解

5.2 算子开发注意事项

SSD 网络中带后处理节点的自定义算子开发注意事项

SSD网络的PriorBox算子与Detection Ouptut算子模型转换时会自动融合，若用户自定义实现了SSD网络中的PriorBox算子，则无法自动融合导致模型转换失败，此时需要用户删除网络模型文件中的Detection Output层算子，然后在Engine编排流程中自定义后处理节点实现Detection Output算子的功能。

版权所有 © 华为技术有限公司

其他

- 请勿修改框架内置算子的实现（路径：ddk/include/inc/custom下的所有文件），否则将可能导致未知风险，例如：系统启动失败、模型无法正常转换等。
- 客户在开发算子插件时需要对自己的源代码负责，避免植入后门。

5.3 环境准备

已完成DDK（Device Development Kit）的部署，DDK为用户提供基于Ascend 310的开发者套件，内含工程代码样例，并集成了相关依赖库和头文件，用户可以通过makefile编译相应的工程文件。

5.4 开发流程总览

TE自定义算子开发流程如图5-2所示。

图 5-2 TE 自定义算子开发流程

主要流程如下：

1. 创建自定义算子开发工程。
2. 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。

3. 自定义算子插件开发，将算子注册到Framework中，算子插件编译后，生成自定义算子的*.so插件文件。
4. 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。

5.5 配置环境变量

步骤1 以DDK安装用户*ascend*登录DDK所在服务器。

步骤2 设置环境变量。

- 若为Atlas 200 DK开发者板环境，在终端窗口执行如下命令设置环境变量。
```
export DDK_HOME=$HOME/tools/che/ddk/ddk
export LD_LIBRARY_PATH=$DDK_HOME/uihost/lib/
export PYTHONPATH=$DDK_HOME/site-packages/te-0.4.0.egg:$DDK_HOME/site-packages/topi-0.4.0.egg
export PATH=$PATH:$DDK_HOME/toolchains/ccec-linux/bin
export TVM_AICPU_LIBRARY_PATH=$DDK_HOME/uihost/lib/:$DDK_HOME/uihost/toolchains/ccec-linux/aicpu_lib
export TVM_AICPU_INCLUDE_PATH=$DDK_HOME/include/inc/tensor_engine
export TVM_AICPU_OS_SYSROOT=/usr/lib/aarch64-linux-gnu
```
- 若为ASIC环境，在终端窗口执行如下命令设置环境变量。
```
export DDK_HOME=$HOME/tools/che/ddk/ddk
export LD_LIBRARY_PATH=$DDK_HOME/uihost/lib/
export PYTHONPATH=$DDK_HOME/site-packages/te-0.4.0.egg:$DDK_HOME/site-packages/topi-0.4.0.egg
export PATH=$PATH:$DDK_HOME/toolchains/ccec-linux/bin
export TVM_AICPU_LIBRARY_PATH=$DDK_HOME/uihost/lib/:$DDK_HOME/uihost/toolchains/ccec-linux/aicpu_lib
export TVM_AICPU_INCLUDE_PATH=$DDK_HOME/include/inc/tensor_engine
export TVM_AICPU_OS_SYSROOT=$DDK_HOME/uihost/toolchains/aarch64-linux-gcc6.3/sysroot
```

说明

DDK_HOME的路径需要修改为实际的DDK安装路径。

用户可自行将设置环境变量的命令写入自定义脚本，方便后续执行。

----结束

5.6 创建自定义算子开发工程

步骤1 以DDK安装的普通用户*ascend*登录DDK所在服务器。

步骤2 创建工作空间目录。

mkdir $HOME/tools/projects

步骤3 创建自定义工程目录。

$ **mkdir $HOME/tools/projects/customop_te**

创建算子代码目录：

mkdir $HOME/tools/projects/customop_te/operator

创建算子插件代码目录：

mkdir $HOME/tools/projects/customop_te/plugin

步骤4 拷贝DDK中的样例代码到算子开发工程目录下。

说明

- 本示例使用ddk中内置的reduction算子代码及算子插件代码进行说明如何进行自定义算子的开发，用户可自己新建相关文件进行编辑。
- 本示例中DDK的安装路径为“$HOME/tools/che”，若实际安装路径与示例路径不一致，请自行替换示例命令中路径。

- 将ddk中的reduction算子样例代码reduction.py和算子验证的样例数据生成脚本data_gen.py拷贝到自定义工程的算子代码目录下。

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/python/reduction.py $HOME/tools/projects/customop_te/operator/

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/python/data_gen.py $HOME/tools/projects/customop_te/operator/

- 将ddk中的reduction算子插件样例代码文件及makefile文件拷贝到自定义工程的算子插件代码目录下。

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/caffe_reduction_layer.cpp $HOME/tools/projects/customop_te/plugin/

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/Makefile $HOME/tools/projects/customop_te/plugin/

- 将ddk中的本示例使用的caffe网络模型文件拷贝到自定义工程目录下。

  cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_caffe_demo/model/ $HOME/tools/projects/customop_te/

本示例所示的自定义算子开发工程目录结构如表5-2所示。

表 5-2 目录结构

样例目录	文件	说明
customop_te/operator	reduction.py	reduction算子样例代码
	data_gen.py	reduction算子样例数据生成文件
customop_te/plugin	caffe_reduction_layer.cpp	reduction算子插件样例代码（需要用户修改）
	Makefile	reduction算子插件编译规则文件（需要用户修改）
customop_te/model	deploy_mylenet-1.prototxt	mylenet网络的原始模型文件
	mylenet-1.caffemodel	mylenet网络的预训练模型文件

----结束

5.7 自定义算子开发

5.7.1 算子特性

深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称Op）。在Caffe中，算子对应层中的计算逻辑，例如：卷积层（Convolution Layer）中的卷积算法，是一个算子；全连接层（Fully-connected Layer， FC layer）中的权值求和过程，是一个算子。

下面介绍算子中常用的基本概念。

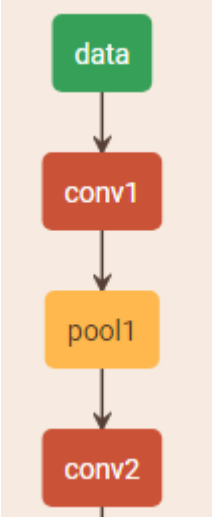
算子类型（Type）

算子的type，代表算子的类型，例如卷积算子的类型为Convolution，在一个网络中同一类型的算子可能存在多个。

算子名称（Name）

算子的name，用于标志网络中的某个算子，同一网络中算子的名称需要保持唯一。如下图所示conv1，pool1，conv2都是此网络中的算子名称，其中conv1与conv2算子的类型为Convolution，表示分别做一次卷积运算。

图 5-3 网络拓扑示例



张量（Tensor）

Tensor是TE算子中的数据，包括输入数据与输出数据，TensorDesc（Tensor描述符）是对输入数据与输出数据的描述，TensorDesc数据结构包含如下属性如表5-3所示。

表 5-3 TensorDesc 属性解释

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如（10,）或者（1024，1024）或者（2，3，4）等。详细介绍请参见 **形状（shape）**。 默认值：无 形式：(i1, i2,…in)，i1，i2，in为正整数
数据类型（dtype）	功能描述：指定Tensor对象的数据类型。 默认值：无 取值范围：float16, int8, int16, int32, uint8, uint16, bool。 **说明** ● 不同计算操作支持的数据类型不同，详细请参见《TE API参考》。 ● TE API同时支持float16与float32数据类型，但OMG进行模型转换的时候会将float32数据类型转换成float16，所以当前版本在进行自定义算子开发时，不支持使用float32的数据类型。
数据排布格式（format）	详细请参见 **数据排布格式（format）**。

- 形状（Shape）

  张量的形状，以(D0, D1, … ,Dn-1)的形式表示，D0到Dn是任意的正整数。

  如形状(3,4)表示第一维有3个元素，第二维有4个元素，(3,4)表示一个3行4列的矩阵。

  在形状的中括号中有多少个数字，就代表这个张量是多少维的张量。形状的第一个元素要看张量最外层的中括号中有几个元素，形状的第二个元素要看张量中从左边开始数第二个中括号中有几个元素，依此类推。例如：

  表 5-4 张量的形状举例

张量	形状
1	(0,)
[1,2,3]	(3,)
[[1,2],[3,4]]	(2,2)
[[[1,2],[3,4]], [[5,6],[7,8]]]	(2,2,2)

- 数据排布格式（format）

  在深度学习框架中，多维数据通过多维数组存储，比如卷积神经网络的特征图用四维数组保存，四个维度分别为批量大小（Batch，N）、特征图高度（Height，H）、特征图宽度（Width，W）以及特征图通道（Channels，C）。

由于数据只能线性存储，因为这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据，比如Caffe，排列顺序为[Batch, Channels, Height, Width]，即NCHW。Tensorflow中，排列顺序为[Batch,Height, Width, Channels]，即NHWC。

如图5-4所示，以一张格式为RGB的图片为例，NCHW实际存储的是“RRRGGGBBB”，同一通道的所有像素值顺序存储在一起，而NHWC实际存储的则是“RGBRGBRGB”，多个通道的同一位置的像素值顺序存储在一起。

图 5-4 NCHW 和 NHWC

昇腾AI软件栈中，为了提高数据访问效率，所有张量数据统一采用NC1HWC0的五维数据格式。其中C0与微架构强相关，等于AI Core中矩阵计算单元的大小，对于FP16类型为16，对于INT8类型则为32，这部分数据需要连续存储；C1=(C+C0-1)/C0。如果结果不整除，最后一份数据需要补零以对齐C0。

整个NHWC -> NC1HWC0的转换过程为：

a. 将NHWC数据在C维度进行分割，变成C1份NHWC0。

b. 将C1份NHWC0在内存中连续排列，由此变成NC1HWC0。

算子属性

不同的算子包含的属性值不同，下面介绍几个常见的算子属性。

- 轴（axis）

  axis代表张量中维度的下标，比如张量a是一个5行6列的二维数组，即shape是(5,6)，则axis=0表示是张量中的第一维，即行。axis=1表示是张量中的第二维，即列。

  例如张量数据[[[1,2],[3,4]], [[5,6],[7,8]]]，Shape为(2,2,2)，则轴0代表第一个维度的数据即[[1,2],[3,4]]与[[5,6],[7,8]]这两个矩阵，轴1代表第二个维度的数据即[1,2]、[3,4]、[5,6]、[7,8]这四个数组，轴2代表第三个维度的数据即1，2，3，4，5，6，7，8这八个数。

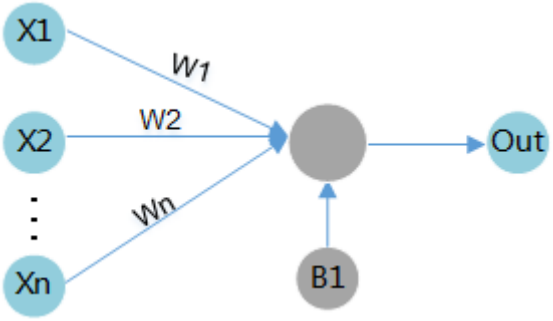
  轴axis可以为负数，此时表示是倒数第axis个维度。

- 偏差（bias）

  偏差是除了权重之外，另一个被应用于输入数据的线性分量。它被加到权重与输入数据相乘的结果中，用于改变权重与输入相乘所得结果的范围。

  如图5-5所示，假设输入数据为X1，与其相关联的权重为W1，偏差为B1，那么在通过计算单元后，数据变为了X1*W1+B1。

图 5-5 偏差计算示例

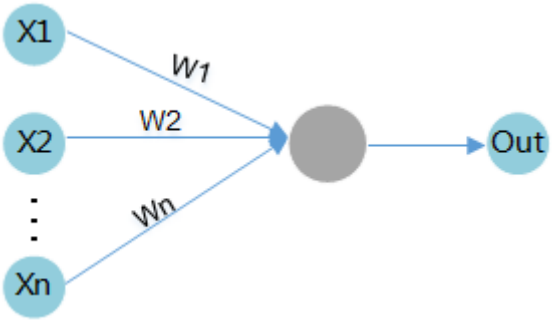


- 权重（weights）

  当输入数据进入计算单元时，会乘以一个权重。例如，如果一个算子有两个输入，则每个输入会分配一个关联权重，一般将认为较重要数据赋予较高的权重，不重要的数据赋予较小的权重，为零的权重则表示特定的特征是无需关注的。

  如图5-6所示，假设输入数据为X1，与其相关联的权重为W1，那么在通过计算单元后，数据变为了X1*W1。

图 5-6 权重计算示例



算子样例—Reduction

Reduction是caffe中的一个算子，将多维数组的指定轴及之后的数据做reduce操作。

- Reduction的属性
  - ReductionOp：算子支持的操作类型，包含四种类型。

    表 5-5 Reduction 算子操作类型

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。

  - axis：Reduction需要指定一个轴，会对此轴及其之后的轴进行reduce操作，取值范围为：[-N，N-1]。

比如，输入的张量的形状为(5,6,7,8)。

- 如果指定的轴是3，则输出tensor的形状为(5,6,7)。
- 如果指定的轴是2，则输出tensor的形状为(5,6)。
- 如果指定的轴是1，则输出tensor的形状为(5,)。
- 如果指定的轴是0，则输出tensor的形状为(1,)。

– coeff：标量，对结果缩放，如果值为1，则不进行缩放。

- Reduction算子的数据
  – 输入数据

    输入包含张量数据及针对张量数据描述。

    输入数据：张量x。

    张量x的描述包含如下属性：

表 5-6 Reduction 算子输入

输入参数	说明
x	张量的name，输入数据，形状由参数shape确定。
shape	输入数据的形状，维度为N。
dtype	输入的数据类型，支持如下类型： float16，float32。

  – 输出数据

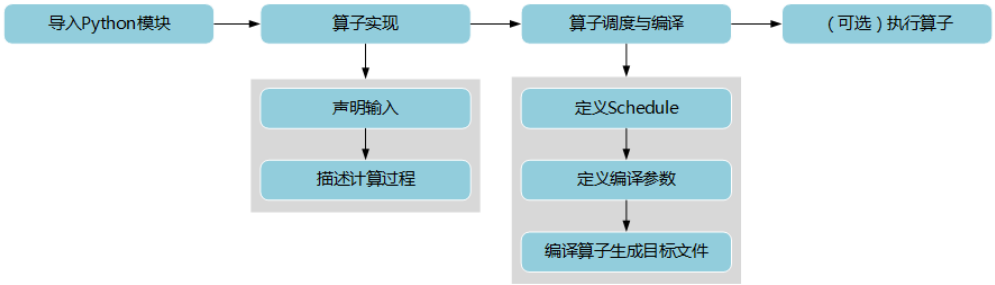
    y：与输入x数据类型相同的张量，其形状由输入张量形状和指定轴决定。

5.7.2 算子代码实现

5.7.2.1 实现流程

TE算子代码通过Python语言开发，实现流程如图5-7所示。

图 5-7 TE 自定义算子实现流程图



说明

- 支持的自定义算子的输入数据类型为：float16, int8, int16, int32, uint8, uint16, bool。
  - 不同计算操作支持的数据类型不同，详细请参见《TE API参考》。
  - TE API同时支持float16与float32数据类型，但OMG进行模型转换的时候会将float32数据类型转换成float16，所以当前版本在进行自定义算子开发时，不支持使用float32的数据类型。
- TE提供了一些已经自定义实现的算子样例代码，用户可参考或直接使用，代码存储路径为DDK安装目录下的“ddk/site-packages/topi-0.4.0.egg/topi/cce”。

5.7.2.2 导入 Python 模块

导入昇腾AI软件栈提供的Python模块，代码示例如下所示。

```
import te.lang.cce
from te import tvm
from topi import generic
```

其中：

- “te.lang.cce”：引入TE支持的特定域语言接口，包括常见的运算vmuls、vadds、matmul等。

  具体的接口定义可查看DDK安装路径下“/site-packages/te-0.4.0.egg/te/lang/cce/”目录下的python函数，使用方法请参见《TE API参考》。
- “te.tvm”：引入TVM后端代码生成机制。

  具体的接口定义可查看DDK安装路径下“/site-packages/te-0.4.0.egg/te/tvm”目录下的python函数，使用方法请参见**https://docs.tvm.ai/**。
- “topi.generic”：提供了自动算子调度接口。

  具体的接口定义可查看DDK安装路径下“/site-packages/topi-0.4.0.egg/topi/generic”目录下的所有python函数，使用方法请参见《TE API参考》。

5.7.2.3 算子实现

算子实现函数定义

如下所示，一个算子的实现函数中包含了输入张量的形状，数据类型，算子属性，内核名称，以及相应的编译、打印等配置。该函数会被插件代码调用，在离线模型生成器进行模型转换时执行。

```
def operationname(shape, dtype, attribute1, attribute2, ... , kernel_name="KernelName", need_build=True, need_print=False)
```

其中：

- shape：输入张量的形状，若算子有多个输入，且每个输入的shape不同，则此处需定义多个shape用于后续对每个输入张量占位，若多个输入张量的shape相同，则可以定义一个shape。
- dtype：输入张量的数据类型。
- *attribute1*，*attribute2*...：算子的属性，根据算子的实际定义进行代码编辑。
- kernel_name：算子在内核中的名称（即生成的二进制文件的名称），用户自定义，保持唯一，只能是大小写字母、数字、“_”的组合，且必须是字母或者“_”开头，长度小于或等于200个字符。

- 编译配置参数“need_build”：取值范围为**True**或者**False**，代表是否需要进行编译。
- 打印配置参数“need_print”：取值范围为**True**或者**False**，代表是否需要打印算子的中间表示（IR：Intermediate Representation）。

例如，

对于Reduction算子，实现函数定义如下：

```
def reduction(shape, dtype, axis, operation, coeff, kernel_name="Reduction", need_build=True,
need_print=False)
```

对于Matmul算子，实现函数定义如下：

```
def matmul(shape_a, shape_b, dtype, kernel_name="matmul", trans_a=False,
trans_b=False,need_build=False, need_print=False):
```

算子实现逻辑

TE的算子实现逻辑总体概括为：

定义好输入数据的张量占位符，然后调用**te.lang.cce**中的各种特定域语言接口进行计算过程的描述，如下代码示例所示：

```
data = tvm.placeholder(shape, name="data_input", dtype=inp_dtype)
with tvm.target.cce():
    cof = coeff
    data_tmp_input = te.lang.cce.vmuls(data, cof)        //对缩放参数进行处理，将输入张量乘上一个标量
    tmp = data_tmp_input
    res_tmp = te.lang.cce.sum(tmp, axis=axis)            //在轴axis上做求和操作
    res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag=True)     //进行数据类型的转换
```

其中：

- data为输入张量，使用TVM的**placeholder**接口进行定义，placeholder是一个占位符，返回一个Tensor对象，表示一组输入数据。

  若算子有多个输入张量，此处需要定义多个Tensor对象。例如：

  ```
  tensor_a = tvm.placeholder(shape_a, name='tensor_a', dtype=dtype)
  tensor_b = tvm.placeholder(shape_b, name='tensor_b', dtype=dtype)
  ```

- vmuls（向量乘），sum（求和）组成中间的计算逻辑。
- cast_to（转换数据类型）：输出张量需要与输入张量的数据类型保持一致，如果计算过程中对数据类型做了转换，需要使用cast_to接口将输出张量的数据类型转换为输入张量的数据类型。

  例如：如果输入张量的数据类型为int8，则执行vmuls操作时，会将数据类型转换为float16，则需要在计算逻辑结束后调用cast_to接口将输出张量的数据类型由float16转换为int8，由于使用vmuls将int8转换为float16的数值时，小数部分为0，所以“f1628IntegerFlag”设置为True，代码示例为：

  ```
  res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag=True)
  ```

  “te.lang.cce.cast_to”接口的详细使用方法请参见《TE API参考》中的“Compute接口”。

- res为输出张量，其数据类型与输入张量的数据类型一致。

用户在进行算子逻辑实现前，可以自定义实现代码对输入数据进行预处理，如下代码示例所示：

```
    # basic check
    check_list = ["float16", "float32"]
    if not (dtype.lower() in check_list):
```

```
        raise RuntimeError("Reduction only support %s while dtype is %s" % (
            ",".join(check_list), dtype))
    reduction_op = ("SUM", "ASUM", "SUMSQ", "MEAN")

    # axis parameter check
    if type(axis) != int:
        raise RuntimeError("type of axis value should be int")
    if axis >= len(shape) or axis < -len(shape):
        raise RuntimeError(
            "input axis is out of range, axis value can be from %d to %d" % (
                -len(shape), len(shape) - 1))
    # operation parameter check
    if operation not in reduction_op:
        raise RuntimeError("operation can only be one of SUM, ASUM, SUMSQ , MEAN")
    # coeff parameter check
    if type(coeff) != int and type(coeff) != float:
        raise RuntimeError("coeff must be a value")
    # Preprocess
    if axis < 0:
        axis = len(shape) + axis
    shape = list(shape)
    shape1 = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])]
    inp_dtype = dtype.lower()
```

5.7.2.4 算子调度与编译

如下代码所示，当定义完计算逻辑后，使用auto_schedule机制，便可以自动生成相应的调度，此处通过TVM的打印机制可以看到相应计算的中间表示。配置信息包括是否需要打印、编译以及算子内核名以及输入、输出张量。

```
sch = generic.auto_schedule(res)
config = {
    "print_ir": need_print,
    "need_build": need_build,
    "name": kernel_name,
    "tensor_list": [data, res]
}
te.lang.cce.cce_build_code(sch, config)
```

- 使用“generic”的“auto_schedule”接口，自动生成相应的调度（schedule），“auto_schedule”接口的参数为算子的输出张量。

  schedule可以理解为：描述的计算过程如何在硬件上高效执行。就是把相关的计算和硬件设备上的相关指令对应起来。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“need_print”参数把它打印出来进行查看。

- “tensor_list”（张量列表）中保存输入张量、输出张量，这个顺序需要严格按照算子本身的输入、输出数据顺序排列。

  例如："tensor_list": [tensor_a, tensor_b, res]，tensor_a与tensor_b是输入张量，res为输出张量。

- 根据调度和配置使用“te.lang.cce”提供的“cce_build_code”接口来进行算子编译，算子编译过程会根据输入的数据形状、类别、算子参数等编译出专用内核，这个过程在离线模型生成器转换模型时发生。
  - sch：生成的算子计算schedule对象。
  - config：编译参数配置的map。

  编译完成后，会生成算子目标文件.o文件（运行目标为AI Core的算子）或者.so文件（运行目标为AI CPU的算子）与算子描述文件.json文件。

5.7.2.5 执行算子

自定义算子代码编写完成后，可以在算子的*.py代码结尾加入调用算子的语句，如下所示，可以参照**5.7.4 算子运行验证**构造输入数据，验证算子执行结果是否正确。

例如：

```
if __name__ == "__main__":
    reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2,kernel_name = "Reduction")
```

5.7.3 代码样例

打开拷贝过来的算子样例文件reduction.py

详细的代码解释请参考如下注释。

```
#coding=utf-8

import te.lang.cce
from te import tvm
from topi import generic
from topi.cce import util
def reduction(shape, dtype, axis, operation, coeff, kernel_name="Reduction",
                              need_build=True, need_print=False):
    """
    Reduce a tensor on a certain axis, and scale output with coeff
    Parameters
    ----------
    shape : shape of data
    dtype : source data type, only support float16, float32
    axis : the first axis to reduce, may be negative to index from the end (e.g., -1 for the last axis).
           If axis == 0, the output Blob always has the empty shape (count 1), performing reduction across the
entire input.
    op : can only be one of "SUM, ASUM (sum of abs), SUMSQ (sum of sqr), MEAN"
    coeff : scale for output
    kernel_name : cce kernel name, default value is "cce_reductionLayer"
    need_buid : if need to build CCEC kernel, default value is False
    need_print : if need to print the ir, default value is False
    Returns
    -------
    None
    """
    #基本参数校验
    #shape参数的校验，check_shape_rule()函数的定义文件为ddk/ddk/site-packages/topi-0.4.0.egg/topi/cce/
util.py
    util.check_shape_rule(shape)
    check_list = ["float16", "float32"]
    if not (dtype.lower() in check_list):
        raise RuntimeError("Reduction only support %s while dtype is %s" % (",".join(check_list), dtype))

    reduction_op = ("SUM", "ASUM", "SUMSQ", "MEAN")

    # axis参数校验
    if type(axis) != int:
        raise RuntimeError("type of axis value should be int")
    if axis >= len(shape) or axis < -len(shape):
        raise RuntimeError(
            "input axis is out of range, axis value can be from %d to %d" % (-len(shape), len(shape) - 1))
   # op参数校验
    if operation not in reduction_op:
        raise RuntimeError("op can only be one of SUM, ASUM, SUMSQ , MEAN")
   # coeff参数校验
    if type(coeff) != int and type(coeff) != float:
        raise RuntimeError("coeff must be a value")
   # 参数预处理
    if axis < 0:
        axis = len(shape) + axis
    shape = list(shape)
```

```
    shape1 = shape[:axis] + [reduce(lambda x, y: x * y, shape[axis:])]
    inp_dtype = dtype.lower()
    # 定义输入数据Tensor，仅占位，未分配实际内存
    data = tvm.placeholder(shape1, name="data_input", dtype=inp_dtype)
    # 定义算子计算过程
    with tvm.target.cce():
        if operation == "ASUM":
            data_tmp_input = te.lang.cce.vabs(data)
            cof = coeff
            tmp = te.lang.cce.vmuls(data_tmp_input, cof)
        elif operation == "SUMSQ":
            data_tmp_input = te.lang.cce.vmul(data, data)
            cof = coeff
            tmp = te.lang.cce.vmuls(data_tmp_input, cof)
        elif operation == "MEAN":
            size = shape1[-1]
            cof = float(coeff) * (size ** (-0.5))
            tmp = te.lang.cce.vmuls(data, cof)
        elif operation == "SUM":
            cof = coeff
            data_tmp_input = te.lang.cce.vmuls(data, cof)
            tmp = data_tmp_input

        #按axis进行求和，降维
        res_tmp = te.lang.cce.sum(tmp, axis=axis)
        #进行数据类型的转换
        res = te.lang.cce.cast_to(res_tmp, inp_dtype, f1628IntegerFlag = True)
        if operation == "MEAN":
            size = shape1[-1]
            sqrt_size = size ** (-0.5)
            res = te.lang.cce.vmuls(res_tmp, sqrt_size)
        #生成算子计算的schedule对象
        sch = generic.auto_schedule(res)
    #定义编译参数
    config = {"print_ir": need_print,
              "need_build": need_build,
              "name": kernel_name,
              "tensor_list": [data, res]}
    #编译算子，生成目标文件
    te.lang.cce.cce_build_code(sch, config)

#以shape为(2,3,4)，datatype为float16，axis为1，op为SUM，coeff为2，算子名称为Reduction的参数来调用reduction算子
if __name__ == "__main__":
    reduction((2, 3, 4), "float16", 1, "SUM", coeff = 2,kernel_name = "Reduction")
```

5.7.4 算子运行验证

本节描述如何单独运行用户开发的算子，验证算子实现的正确性。

5.7.4.1 算子编译

对算子代码进行编译，生成算子的二进制文件及算子描述文件。

步骤1 算子编译依赖DDK的版本号，所以编译算子前需要获取当前使用的DDK的版本号。

在“$HOME/tools/che/ddk/ddk/ddk_info”文件中查看DDK版本号。

如图5-8所示，VERSION字段即为当前DDK版本。

图 5-8 DDK 版本号查看

```
{
    "VERSION": "1.3.T18.B850",
    "NAME": "DDK",
    "TARGET": "ASIC"
}
```

步骤2 执行如下命令设置版本号信息、编译算子。

1. 在customop_te/operator目录下执行如下命令进入python交互模式。

   python

2. 在python交互模式下依次执行如下命令设置DDK的版本号。

   from topi.cce import te_set_version

   import subprocess

   te_set_version("1.3.T18.B850")

   执行示例如图5-9所示。

   图 5-9 设置 DDK 版本号示例

```
ascend@ascend-HP-ProDesk-600-G4-PCI-MT:~/tools/projects/customop_te/operator$ python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from topi.cce import te_set_version
[06:54:41] src/tbe_lang/tune/plugin/plugin_mgr.cc:86: plugin type:tiling name:ml_tuning is registered successfully!
>>> import subprocess
>>> te_set_version("1.3.T18.B850")
'success'
```

3. 执行如下命令编译reduction.py文件，生成算子二进制文件与描述文件。

   subprocess.call("python reduction.py",shell=True)

   图 5-10 编译算子文件

```
>>> subprocess.call("python reduction.py",shell=True)
[06:55:46] src/tbe_lang/tune/plugin/plugin_mgr.cc:86: plugin type:tiling name:ml_tuning is registered successfully!
0
```

4. 退出python交互模式。

   quit(0)

步骤3 算子编译完成后，会在当前operator目录下生成kernel_meta文件夹，kernel_meta文件夹下为编译生成的算子二进制文件*.o文件（运行目标为AI Core的算子）或者*.so文件（运行目标为AI CPU的算子）及算子描述文件*.json文件。

- *.o或者*.so是算子的二进制文件。
- .json文件是对应的算子描述文件，用于定义算子属性及运行时所需要的资源。

  json文件解析如下所示：

```
{
"binFileName":"Reduction",          //生成的算子的二进制文件名称
"binFileSuffix":".o",           //生成的算子的二进制文件后缀，运行目标为AI CPU时后缀为".so"
"blockDim":1,                   //计算时用到的AI Core的数量
"kernelName":"Reduction__kernel0",   //算子的kernel函数名
"magic":"RT_DEV_BINARY_MAGIC_ELF",   //代表算子运行目标为AI Core，若magic值为
RT_DEV_BINARY_MAGIC_ELF_AICPU，代表运行目标是AI CPU
"sha256":"d5158df2ff2e64743eec7f527ddd9078b99c1d670f5adbd8b789224657ab0f91"  //对.o文件进行
加密得到的值
}
```

----结束

5.7.4.2 构造输入数据文件

运行单算子需要构造算子输入数据，并以二进制格式保存到算子工程下。

步骤1 以DDK安装用户进入自定义算子工程的算子代码目录下。

cd $HOME/tools/projects/customop_te/operator/

步骤2 执行样例数据生成脚本，生成reduction算子的样例数据文件。

python data_gen.py reduction

在算子工程中会生成如图5-11所示数据文件。

图 5-11 样例数据生成

```
Reduction_input_2_3_4_sum_axis_1.data
Reduction_input_2_3_4_sum_axis_1.txt
Reduction_output_2_3_4_sum_axis_1.data
Reduction_output_2_3_4_sum_axis_1.txt
```

数据文件说明如表5-7所示。

表 5-7 数据文件说明

数据文件	说明
Reduction_input_2_3_4_sum_axis_1.data	reduction算子的二进制格式输入数据文件。
Reduction_input_2_3_4_sum_axis_1.txt	将reduction算子的二进制格式文件以txt文件的方式显示出来，方便用户查看结果。
Reduction_output_2_3_4_sum_axis_1.data	reduction算子二进制格式输出数据验证文件，用于验证算子运行后的输出结果是否正确。
Reduction_output_2_3_4_sum_axis_1.txt	将reduction算子的二进制格式输出数据文件以txt文件的方式显示出来，方便用户查看结果。

----结束

5.7.4.3 运行单算子

步骤1 在DDK 样例**customop_runner**中生成单算子编译二进制文件。

1. 给customop_runner工程样例赋予写权限。

   chmod -R +w $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/

2. 进入DDK样例**customop_runner**工程的build目录。

   cd $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/build

3. 设置DDK_PATH的环境变量。

   export DDK_PATH=$HOME/tools/che/ddk/ddk

4. 执行如下命令生成Makefile文件。

- 对于Atlas 200 DK开发者板环境，执行如下命令。

  cmake -Dtarget=OI .

- 对于设备类型为ASIC的环境，执行如下命令。

  cmake .

5. 执行make命令，编译生成二进制文件。

   make

   在“$HOME/tools/che/ddk/ddk/sample/customop/customop_runner/out/”目录下会生成可执行文件**main和动态库libcustom_engine.so**

步骤2 把out目录拷贝到TE自定义算子工程目录。

cp -rf $HOME/tools/che/ddk/ddk/sample/customop/customop_runner/out $HOME/tools/projects/customop_te/

步骤3 配置输入数据、输出数据及验证描述文件。

进入TE自定义算子工程目录中的out文件夹。

cd $HOME/tools/projects/customop_te/out

- 在input.txt文件中配置输入数据文件路径及文件名。

  例如：

  ```
  dataPath=../operator/Reduction_input_2_3_4_sum_axis_1.data
  ```

  如果自定义算子有多个输入数据，需要自定义多个输入数据的txt文件，例如input1.txt、input2.txt等，分别按照input.txt的格式进行定义。

- 在output.txt中配置输出数据的相关信息

  例如：

  ```
  size=4
  dataPath=./output/out0.data
  dtype=1
  ```

  - size：预期的输出数据文件大小，单位字节。
  - dataPath：输出数据的路径及文件名。
  - dtype：输出数据类型，其中**1**代表**float1**6，**2**代表**float32**。

- 在expect.txt文件中配置期望生成数据的路径及文件名。

  例如：

  ```
  dataPath=../operator/Reduction_output_2_3_4_sum_axis_1.data
  ```

步骤4 将operator文件夹与out文件夹以HwHiAiUser用户拷贝到开发者板或者ASIC设备的Host侧，operator与out文件夹应放置在同一目录下。

例如拷贝到“/home/HwHiAiUser/projects”目录下。

步骤5 运行算子

以HwHiAiUser用户登录开发者板或者ASIC类型设备的Host侧，并进入out目录, 例如“/home/HwHiAiUser/projects/out” 。

1. 在当前目录下创建output文件夹，用于存储生成的数据文件。此文件路径及名称为output.txt文件中定义的。

   mkdir output

2. 执行如下命令为main文件添加可执行权限。

   chmod +x main

3. 执行如下命令，运行单算子并进行算子比对。

   ./main -i input.txt -o output.txt -e expect.txt -b ../operator/kernel_meta/Reduction.o -p *0.2* -d *0.2* -k *Reduction_kernel0* -t 0

   其中：

   - -i：输入数据配置文件，配置了输入数据的路径及文件名。

     如果自定义算子有多个输入数据，需要自定义多个输入数据的txt文件，例如input1.txt、input2.txt等，分别按照input.txt的格式进行定义。
   - -o：输出数据配置文件，配置了输出数据的大小、路径及文件名，输出数据的类型。
   - -e：期望数据配置文件，配置了期望输出数据的路径及文件名，用于做结果比对。
   - -b：算子实现的二进制文件*.o（运行目标为AI Core的算子）或者*.so（运行目标为AI CPU的算子）。
   - -p：精度偏差，表示数据相对误差的允许范围，取值为[0,1)，精度偏差越小表示精度越高。
   - -d：统计偏差，整个数据集中精度偏差不满足门限的数据量占比，数值范围（0，1），统计偏差越小表示精度越高。
   - -k：核函数名称，TE算子的核函数名称需要与.json文件中的kernelName保持一致；C++算子的内核名称需要与算子在Framework注册的opetype的名称保持一致。
   - -t：有三种取值，0代表TE_AI Core算子，1代表TE_AI CPU算子，2代表C++的自定义算子（AI CPU算子）。

     对于TE算子，是运行在AI Core上还是运行在AI CPU上可以通过算子编译完成后生成的****.json**文件中的“magic”参数值进行查看。

   执行成功后，会在当前目录的output文件夹下会生成名称为out0.data结果文件及名称为vertifyResult.txt的验证数据比对结果文件。

   vertifyResult.txt文件样例如下，代表实际输出数据与期望输出数据内容比对一致。

```
Output file ./output/out0.data compare result true
```

----结束

5.8 caffe.proto 文件算子定义（可选）

如果使用的是Caffe开源框架的模型，请参考本章节进行caffe.proto文件的定义；如果使用的Tensorflow开源框架的模型，此章节跳过。

算子开发完成后，需要在DDK内置的caffe.proto文件中增加自定义算子的相关定义（如果内置的caffe.proto文件中已存在此算子的定义则此章节跳过）。

若同一模型中存在多个不支持的自定义算子，请参考本章节一次性将相关算子定义进行添加。算子插件实现时会根据算子名称从caffe.proto文件读取相关的算子参数进行解析，进而将算子数据结构转换为适配昇腾AI处理器的离线模型支持的数据结构。

DDK中内置的caffe.proto文件的路径为：“$HOME/tools/che/ddk/ddk/include/inc/custom/proto/caffe/caffe.proto”，用户可基于此文件进行修改，添加自定义算子的定义。

此处以添加Reduction算子的定义为例（DDK中内置的caffe.proto已经添加了Reduction算子的定义，此处以此样例进行说明）。

步骤1 在LayerParameter中添加Reduction算子的定义。

如下所示，在LayerParameter中加入Reduction层的定义，并设置其ID。

```
message LayerParameter {
......
optional ReductionParameter reduction_param = 136;   //此ID需要保持唯一
......
}
```

步骤2 在caffe.proto文件中增加Reduction层的参数定义。

```
message ReductionParameter {
  enum ReductionOp {          //算子支持的操作类型
    SUM = 1;                    //对被reduce的所有轴求和
    ASUM = 2;                   //对被reduce的所有轴求绝对值后求和
    SUMSQ = 3;                  //对被reduce的所有轴求平方后再求和
    MEAN = 4;                   //对被reduce的所有轴求均值
  }
  optional ReductionOp operation = 1 [default = SUM];   // 定义算子的操作
  optional int32 axis = 2 [default = 0];                          //定义reduce的轴
  optional float coeff = 3 [default = 1.0]; // coefficient for output  //标量，对结果进行缩放的倍数
}
```

----结束

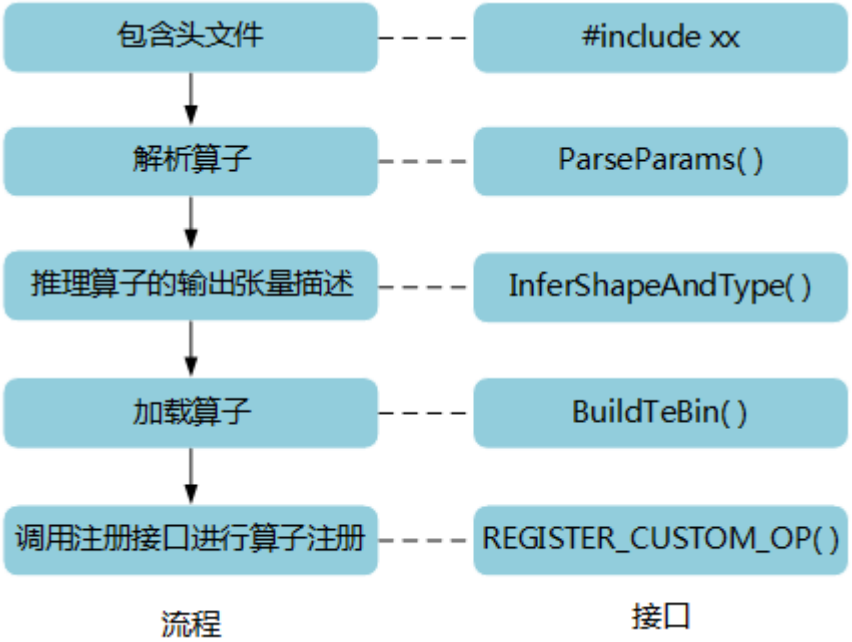
5.9 自定义算子插件开发

5.9.1 插件代码实现

5.9.1.1 插件实现流程

自定义算子开发完成后，离线模型生成器需要支持将自定义算子的属性值转换到离线模型，在离线模型执行阶段能够支持自定义算子的执行，所以用户需要开发自定义算子插件，进行算子属性的解析、形状类别的推断，并通过离线模型生成器提供的注册机制完成自定义算子的注册。

图 5-12 自定义算子插件实现流程图



5.9.1.2 包含头文件

应在插件实现文件的头部使用预编译命令“#include”将插件实现函数相关的头文件包含到插件实现源文件中。

```
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
```

表 5-8 头文件说明

头文件	目录	作用
custom/custom_op.h	DDK安装目录下的/include/inc/custom/custom_op.h。	包含该头文件，可调用自定义的编译函数及算子调测函数、算子校验函数。
framework/omg/register.h	DDK安装目录下的/include/inc/framework/omg/register.h。	包含该头文件，可使用算子注册类相关，调用算子注册相关的接口。
framework/omg/omg_types.h	DDK安装目录下的/include/inc/framework/omg/omg_types.h。	包含该头文件，可使用TE自定义算子信息结构体TEBinInfo。

头文件	目录	作用
proto/caffe/caffe.pb.h	算子插件编译时，在算子工程所在目录下生成proto/caffe/caffe.pb.h文件。	算子插件编译时，会自动编译DDK安装目录下的“/include/inc/custom/proto/caffe/caffe.proto”文件，并在算子工程所在目录下生成proto/caffe/caffe.pb.h文件供插件代码调用进行算子参数解析。
operator.h	DDK安装目录下的include/inc/graph/operator.h。	包含该头文件，可调用算子属性存取、输入/输出设置接口。
attr_value.h	DDK安装目录下的/include/inc/graph/attr_value.h。	包含该头文件，可使用AttrValue类下各数据类型。
memory	C++标准库。	包含该头文件，可使用C++标准库中的智能指针、内存分配器、临时性的申请与释放动态内存的函数、在内存上构造对象的函数等。
string	C++标准库。	包含该头文件，可使用string类构造对象并调用string相关接口。
vector	C++标准库。	包含该头文件，可使用vector类模板并调用vector相关接口。

算子插件编译前，由于工程中不存在proto/caffe/caffe.pb.h文件，所以工程中如果出现如下提示信息，请忽略。

图 5-13 caffe.pb.h 无法解析提示信息

```
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
```

5.9.1.3 解析算子

对于用户新开发的自定义算子，需要自定义实现解析算子中属性的函数，将原始模型中的算子属性定义转换为适配昇腾AI处理器的离线模型中的算子属性定义。如果用户重写昇腾AI处理器内置算子，则此步骤跳过，插件会自动实现对内置算子的解析。

函数声明

算子解析函数的声明如下所示：

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

- ParseParamsxx：函数名称，用户自定义，需要保持唯一。
- op_origin：源算子模型，为protobuf格式的数据结构，来源于Caffe模型的proto文件，存储路径为DDK安装目录下的“/include/inc/custom/proto/caffe/caffe.proto”，若用户自定义的算子在caffe.proto文件中未定义，则需要参考**5.8 caffe.proto文件算子定义（可选）**增加算子定义。

  算子插件实现时会根据算子名称从caffe.proto编译生成的proto/caffe/caffe.pb.h文件与caffe.pb.cc文件中读取相关的算子属性进行解析，进而将算子数据结构转换为离线模型支持的数据结构。

  DDK中内置的caffe.proto文件的路径为DDK安装目录下的：“/include/inc/custom/proto/caffe/caffe.proto”，用户可基于此文件进行修改，添加自定义算子的定义。
- op_dest：目标算子模型，适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息，Operator类的详细描述请参见《GE API参考》中的“Operator类接口”。

实现流程

ParseParamsxx函数的实现流程如下所示：

步骤1 定义指向LayerParameter的对象，并获取当前算子层的句柄。

```
const caffe::LayerParameter* layer =dynamic_cast<const caffe::LayerParameter*>(op_origin);
const caffe::xxxParameter& param = layer->reduction_param();
```

其中：

- param对象的类型caffe::xxxParameter中的xxxParameter需要与LayerParameter对象中声明的类型保持一致。
- layer对象的成员函数xxx_param()的名字需要与与LayerParameter对象中声明的对象名字保持一致。

例如caffe.proto中Reduction算子及Convolution算子的定义如下：

```
message LayerParameter {
optional ReductionParameter reduction_param = 136;
optional ConvolutionParameter convolution_param = 106;
...
}
```

则获取当前Reduction算子层及Convolution算子层的句柄代码定义如下：

const caffe::**ReductionParameter**& param = layer->**reduction_param**()

const caffe::**ConvolutionParameter**& param = layer->**convolution_param()**

步骤2 解析算子的每一个属性，并将其赋给Operator类型的对象op_dest。

开发者可调用CreateFrom<AttrValue::**T**>(**DT**&& val)接口将**DT**类型的参数转换为AttrValue类的**T**类型的参数，然后调用SetAttr(const string& name, const AttrValue& value)接口将转换后的AttrValue对象的值赋给op_dest对象的相应属性。

T类型是昇腾AI软件栈为了简化类型定义，对当前支持的数据类型做的重命名，T类型与原始数据类型的对应关系如表5-9所示，原型定义可参考DDK安装目录的“include/inc/graph/attr_value.h”文件。

表 5-9 T 类型与原始数据类型对应关系

T类型	原始数据类型
INT	int64_t
FLOAT	float
STR	std::string
TENSOR	TensorPtr
TENSOR_DESC	TensorDesc
GRAPH	ComputeGraphPtr
BYTES	Buffer
NAMED_ATTRS	NamedAttrs
BOOL	bool
LIST_INT	vector<INT>
LIST_FLOAT	vector<FLOAT>
LIST_BOOL	vector<BOOL>
LIST_STR	vector<STR>
LIST_TENSOR	vector<TENSOR>
LIST_TENSOR_DESC	vector<TENSOR_DESC>
LIST_GRAPH	vector<GRAPH>
LIST_BYTES	vector<BYTES>
LIST_NAMED_ATTRS	vector<NAMED_ATTRS>

SetAttr接口的详细描述请参见《GE API参考》。

下面给出几种常见类型的参数解析示例：

- int或者float类型参数

  例如，caffe.proto文件中算子参数定义如下所示：

  ```
  message ReductionParameter {
    ......
    optional int32 axis = 2 [default = 0];
    optional float coeff = 3 [default = 1.0];
  }
  ```

  调用SetAttr接口，将“param.axis()”的值赋给op_dest对象的axis属性，并将类型转换为INT；将“param.coeff()”的值赋给op_dest对象的coeff属性，并将类型转换为FLOAT，如下所示：

```
op_dest.SetAttr("axis", AttrValue::CreateFrom<AttrValue::INT>(param.axis()));
op_dest.SetAttr("coeff", AttrValue::CreateFrom<AttrValue::FLOAT>(param.coeff()));
```

其中CreateFrom<AttrValue::**T**>(**DT**&& val)是将**DT**类型的参数转换为AttrValue类的**T**类型的参数。

- enum类型参数

  例如，caffe.proto文件中算子属性定义如下所示：

```
message ReductionParameter {
  enum ReductionOp {
    SUM = 1;
    ASUM = 2;
    SUMSQ = 3;
    MEAN = 4;
  }
...}
```

  a. 首先，需要将enum类型参数转换为map类型参数。

```
std::map<caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
        { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
    { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
    { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
    { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
    };
```

  b. 调用SetAttr接口，将map类型参数*operation_map*赋值给op_dest对象的operation属性。

```
op_dest.SetAttr("operation",
AttrValue::CreateFrom<AttrValue::STR>(operation_map[param.operation()]));
```

  SetAttr接口的详细描述请参见《GE API参考》。

- repeated类型参数

  例如caffe.proto文件中算子参数定义如下所示：

```
message xxxParameter {
...
  repeated float min_size = 1;
  repeated uint32 offset = 2;
....
}
```

  a. 首先，将repeated float类型参数转换为vector<float>类型参数，将repeated uint32类型参数转换为vector<uint32>类型参数，并对vector类型的参数进行赋值。

```
vector<float> min_size;
vector<uint32> offset;
for(int i = 0; i < param.min_size_size(); ++i)
{
   min_size.push_back(param.min_size(i));   //调用vector类型对象的push_back函数为min_size对象赋值
}
for(int i = 0; i < param.offset_size(); ++i)
{
   offset.push_back(param.offset(i));      //调用vector对象的push_back函数为offset对象赋值
}
```

  b. 调用SetAttr接口，将vector<float>类型参数min_size赋值给op_dest对象的min_size属性；将vector<uint32>类型参数offset赋值给op_dest对象的offset属性，类型为LIST_INT。

```
op_dest.SetAttr("min_size",
ge::AttrValue::CreateFrom<ge::AttrValue::LIST_FLOAT>(min_size));
op_dest.SetAttr("offset", ge::AttrValue::CreateFrom<ge::AttrValue::LIST_INT>(offset));
```

SetAttr接口的详细描述请参见《GE API参考》。

----结束

5.9.1.4 推理算子输出张量描述

用户需要根据算子的输入张量描述、算子逻辑及算子属性，推理出算子的输出张量描述，包括张量的形状、数据类型及数据排布格式等信息。这样离线模型转换时就可以为所有的张量静态分配内存，避免动态内存分配带来的开销。

函数声明

函数的声明如下所示：

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

- InferShapeAndTypexx：函数名称，用户自定义，需要保持唯一。
- op：计算节点定义，存储输入张量描述及各种算子属性，ge::Operator类型的介绍请参见《GE API参考》中的“Operator类接口”。
- v_output_desc：存储该计算节点的输出张量描述，包括形状、数据排布格式及数据类型，关于TensorDesc类型的介绍请参见《GE API参考》中的“TensorDesc类接口”。

下面详细讲解不同场景下InferShapeAndTypexx函数的实现。

输出张量与输入张量形状相同的算子

对于输出张量与输入张量形状相同的算子，可以直接将输入张量的描述插入输出张量描述所在的向量空间中。

代码示例如下所示：

```
v_output_desc.push_back(op.GetInputDesc(0));
```

其中GetInputDesc是Operator类中根据算子Input名称或者Input索引获取输入张量描述的接口，详细的接口介绍请参见《GE API参考》中的“Operator类接口”。

降低维度的算子

对于Reduction、Reduce之类的常见降维操作，需要根据算子输入属性axis等信息计算出输出张量的形状（包含输出张量维度以及每一个维度有多少个元素），然后将输出张量的形状插入到v_output_desc向量中。

代码示例如下所示：

步骤1 获取算子的输入张量描述及输入张量的形状信息。

```
auto tensorDesc = op.GetInputDesc(0);        //输入张量描述，包括的形状、数据排布格式及数据类型
auto shape = tensorDesc.GetShape();          //获取输入张量的形状
```

GetShape接口的详细介绍请参见《GE API参考》中的“TensorDesc类接口”。

步骤2 根据计算逻辑获取算子的属性值，并计算算子输出张量的形状。

例如，对于Mylenet网络中的Reduction算子，由于Reduction的上一层为Softmax，Softmax的实际输出是2维，但在离线模型中会被补齐到4维，所以需要调整axis，将其指向2维的位置，进行reduce操作，并将调整后的Shape值赋给输出张量描述。

- 从operator对象中获取axis属性的键值对，然后从其中获取axis的属性值，并将其从INT类型转换为int64_t类型赋给名称为axis的变量，最后对axis的值进行校验、调整，将其指向轴1的位置，代码示例如下所示：

```
int64_t axis = -1;
ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("Get axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to
point to the original 2d.
    if (axis < 0) axis -= 2;

    if (axis < 0) axis += shape.GetDimNum();

    if (axis < 0 || axis >= shape.GetDimNum())
    {
        printf("invalid axis:%d, dim_size:%d\n", (int32_t)axis, (int32_t)shape.GetDimNum());
        return PARAM_INVALID;
    }
```

- 调整Shape，将轴1及其之后的维度设置为1，例如输入张量的Shape为(2,3,4,5)，则调整之后的Shape为(2,1,1,1)。

```
int32_t dimsize = (int32_t)shape.GetDimNum();
int32_t idx = 0;
for(idx=axis; idx<dimsize; idx++)
{
    shape.SetDim(idx, 1);
}
```

- 将调整后的Shape值设置到tensorDesc对象。

```
tensorDesc.SetShape(shape);
```

GetDimNum与SetDim接口的详细介绍请参见《GE API参考》中的“Shape类接口”。

步骤3 设置算子的输出张量描述。

```
v_output_desc.push_back(tensorDesc)
```

将tensorDesc赋给输出张量的描述对象v_output_desc。

----结束

网络存在 Type 相同的多个算子

一个网络中往往存在多层某个Type相同的算子，例如Convolution算子，开发者有时需要对某层算子进行形状的自定义（重定义已经存在的某层算子），此时在进行输出张量描述推理的时候需要根据网络的不同情况，根据num_output、kernel、stride、pad等属性判断是针对哪一层的算子进行的自定义，从而推断出对应算子的张量信息。

代码示例如下所示：

步骤1 将输入张量描述赋给输出张量描述，并获取算子的输入张量描述及输入张量的形状信息。

```
v_output_desc.push_back(op.GetInputDesc(0));  //将输入张量描述信息赋给输出张量描述对象，开发者也可以将后续推理出的shape直接赋给tensorDesc对象，再将tensorDesc对象赋给输出张量描述对象。
auto tensorDesc = op.GetInputDesc(0);        //输入张量描述，包括的形状、数据排布格式及数据类型
auto shape = tensorDesc.GetShape();          //获取输入张量的形状
```

GetShape接口的详细介绍请参见《GE API参考》中的“TensorDesc类接口”。

步骤2 根据计算逻辑获取算子的属性值，并根据算子的属性值、shape匹配算子，并计算算子输出张量的形状。

例如，对于某一网络中的Type为Convolution的算子（网络中存在多层Convolution），匹配num_outputs为128、shape.GetDim(0)为1、shape.GetDim(1)为128、shape.GetDim(2) 为28、shape.GetDim(3) 为28的Convolution，并对其输出描述张量的形状重新赋值。

- 获取num_outputs的属性值

```
ge::AttrValue num_outputsAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("num_output", num_outputsAttrValue)) ||
        (ge::GRAPH_SUCCESS != num_outputsAttrValue.GetValue<AttrValue::INT>(num_outputs)))
    {
        printf("GetOpAttr num_outputs failed!\n");
    }
```

- 匹配num_outputs为128、shape.GetDim(0)为1、shape.GetDim(1) 为128、shape.GetDim(2) 为28、shape.GetDim(3) 为28的Convolution，并对其输出张量描述的形状重新赋值。

```
 if(shape.GetDim(0) == 1 && shape.GetDim(1) == 128 &&
        shape.GetDim(2) == 28 && shape.GetDim(3) == 28 && num_outputs == 128)
    {
        shape.SetDim(0, 1);
        shape.SetDim(1, 128);
        shape.SetDim(2, 28);
        shape.SetDim(3, 28);
        v_output_desc[0].SetShape(shape);
        return SUCCESS;
        return FAILED;
    }
```

GetDimNum与SetDim接口的详细介绍请参见《GE API参考》中的“Shape类接口”。

说明

若需要使用op_name进行匹配，获取op_name的方法如下所示：

```
auto op_name = op.GetName();
```

其他属性kernel_w、kernel_h、stride_w、stride_h、pad_w、pad_h的获取方法与num_output相同，修改op.GetAttr中的key值即可。

----结束

5.9.1.5 编译算子

函数声明

算子编译函数的声明如下所示：

```
Status BuildTeBinxx(const ge::Operator& op, TEBinInfo& te_bin_info)
```

其中：

- BuildTeBinxx：函数名称，用户自定义，需要保持唯一。
- op：目标算子模型，适配昇腾AI处理器的离线模型的算子数据结构，保存算子信息，Operator类的详细描述请参见《GE API参考》中的“Operator类接口”。
- te_bin_info：存储自定义算子二进制文件路径、算子描述文件路径以及DDK的版本信息。TEBinInfo结构体的详细描述请参见《Framework API参考》中的“TEBinBuildFn函数”。

实现流程

算子编译函数在离线模型生成器进行模型转换时调用，主要完成以下流程：

- 获取算子张量描述信息以及算子属性，模型转换时，这些信息都需要是确定值，用于进行算子的匹配。

  例如，如下所示，模型转换时只匹配axis为1、输入张量Shape的Dim为4的Reduction算子。

```
//解析算子属性operation
ge::AttrValue operationAttrValue;
if ((ge::GRAPH_SUCCESS != op.GetAttr("operation", operationAttrValue)) || (ge::GRAPH_SUCCESS !=
operationAttrValue.GetValue<AttrValue::STR>(operation)))
{
    printf("GetOpAttr operation failed!\n");
}

// 解析算子属性axis ,并调整axis指向Mylenet网络中Reduction算子的上一层Softmax算子的实际输出维
度，即轴1的位置。
ge::AttrValue axisAttrValue;
if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
{
    printf("GetOpAttr axis failed!\n");
}
// In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to
point to the original 2d.
if(axis < 0)
    axis -= 2;

// 解析算子属性coeff
ge::AttrValue coeffAttrValue;
if ((ge::GRAPH_SUCCESS != op.GetAttr("coeff", coeffAttrValue)) || (ge::GRAPH_SUCCESS !=
coeffAttrValue.GetValue<AttrValue::FLOAT>(coeff)))
{
    printf("GetOpAttr coeff failed!\n");
}
// 获取算子输入张量描述信息
TensorDesc input_desc      = op.GetInputDesc(0);

// 解析Input shape并校验算子的Dim是否为4
if(input_desc.GetShape().GetDimNum() != 4)
{
    printf("The shape size is %d, which is not 4!", (int32_t)input_desc.GetShape().GetDimNum());
    return FAILED;
}
```

- 指定算子实现文件、算子实现函数以及内核名。

```
FilePath   = "project_path/operator/reduction";    //算子实现文件所在的绝对路径+算子py文件名称
FuncName   = "Reduction";                  //算子实现文件中算子实现函数的名称。
KernelName = "Reduction";                  //算子实现文件中算子实现函数中定义的kernel_name，即生成
的二进制文件的名称。
```

- 指定算子编译生成的算子描述文件（*.json）文件的路径，请保持如下固定配置。

```
te_bin_info.json_file_path = "./kernel_meta/" + KernelName + ".json";
```

  模型转换时，会从此路径中读取算子的二进制描述文件，获取算子相关信息。

  执行omg模型转换命令时，会根据FilePath中配置的算子实现的路径拷贝算子编译生成的kernel_meta文件夹到执行omg命令时所在的路径，所以*.json文件相对于执行omg命令所在目录的路径固定为“./kernel_meta”。

- 调用te::BuildCustomop函数来调用算子实现文件的python函数编译算子。

  BuildCustomop函数调用如下所示：

```
te::BuildTeCustomOp(te_bin_info.ddk_version, op.GetName(), FilePath, FuncName,"(i,i,i,i), s, i, s, f, s",
input_desc.GetShape().GetDim(0),input_desc.GetShape().GetDim(1),input_desc.GetShape().GetDim(2),
input_desc.GetShape().GetDim(3), "float16", axis, operation.c_str(), coeff,KernelName.c_str());
```

其中：

- te_bin_info.ddk_version：DDK的版本信息，固定参数。后续模型转换时会自动传入DDK版本信息。
- op.GetName( )：获取算子名称，固定参数。
- FilePath：前面定义的算子文件所在相对路径。
- FuncName：算子实现文件中的算子实现函数名称。
- *"(i,i,i,i), s, i, s, f, s"*：代表算子实现文件中实现函数的参数占位符，其中i表示整形，s表示字符串类型，f表示单精度浮点数类型，o表示PyObject*类型。占位符需要和后面紧跟的参数顺序和类型保持一致，同时需要和算子实现文件中的算子实现函数的定义保持一致。BuildCustomop会根据这些参数调用算子实现函数，再通过TVM机制生成内核。

5.9.1.6 算子注册

框架管理器（Framework）提供REGISTER_CUSTOM_OP宏，按照指定的算子名称完成算子的注册。

自定义算子的注册代码如下所示：

```
REGISTER_CUSTOM_OP("test_layer")
    .FrameworkType(CAFFE)
    .OriginOpType("Test")
    .ParseParamsFn(ParseParamsxx)
    .InferShapeAndTypeFn(InferShapeAndTypexx)
    .TEBinBuildFn(BuildTeBinxx)
    .ImplyType(ImplyType::TVM)
    .Formats({DOMI_TENSOR_NC1HWC0}, {DOMI_TENSOR_NC1HWC0})
    .WeightFormats({DOMI_TENSOR_FRACTAL_Z, DOMI_TENSOR_NC1HWC0});
```

其中：

- REGISTER_CUSTOM_OP：注册自定义算子，"test_layer"作为离线模型文件中的算子名称，可以任意命名但不能和已有的算子命名冲突。
- FrameworkType：不同框架的算子参数解析逻辑不同，因此对于不同框架模型需要有不同的插件，在插件的注册代码需要表明对应的框架。当前请配置为CAFFE。
- OriginOpType：算子类型，需要与Caffe Prototxt中定义的算子类型保持一致，否则无法正常解析，内置的caffe.proto文件路径为DDK安装路径下的“/include/inc/custom/proto/caffe/caffe.proto”。
- ParseParamsFn：用来注册解析模型的函数，“ParseParamsxx”即**5.9.1.3 解析算子**中实现的函数，针对Caffe框架开发插件时需要，若自定义的算子为重写昇腾AI处理器已经支持的算子，则此步骤可跳过；若自定义的算子为昇腾AI处理器不支持的算子，则此步骤必选。
- InferShapeAndTypeFn：用来注册形状和类别推断函数，“InferShapeAndTypexx”即**5.9.1.4 推理算子输出张量描述**中实现的函数。
- TEBinBuildFn：用来注册TE算子编译函数，“BuildTeBinxx”即**5.9.1.5 编译算子**中实现的函数。
- ImplyType：指定算子的实现方式，ImplyType::TVM表示该算子是TE算子。
- Formats：算子输入数据与输出数据的数据排布格式，其中第一个列表是输入数据格式列表，第二个列表为输出数据格式列表，若输入数据有多个，请在第一个列表中列出每一个输入数据的排布格式。例如若有两个输入数据，分别是NC1HWC0格式，则Formats函数调用方式如下：

```
.Formats({DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0},
{DOMI_TENSOR_NC1HWC0})
```

详细说明请参见《Framework API参考》中的“Formats函数”。

- WeightFormats：设置算子的权重数据的排布格式，支持的数据格式类型请参见《Framework API参考》中的“WeightFormats函数”。例如Convolution的filter的数据排布格式是fractal_Z，bias的数据排布格式是NC1HWC0。

  如果模型转换时开启量化开关，需要在此接口中增加Framework处理时新增的常量数据的格式，当前Framework支持的量化算子有Conv，FC与Depthwise Conv，若这几个算子在模型转换时开启了量化开关，则需要在WeightFormats接口的参数列表最后增加6个DOMI_TENSOR_NC1HWC0的数据格式（量化时Framework新增了6个数据排布格式为NC1HWC0的常量，此处不详细介绍），开启量化后的Convolution的WeightFormats接口示例如下：

```
.WeightFormats({DOMI_TENSOR_FRACTAL_Z, DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0,
DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0,DOMI_TENSOR_NC1HWC0,
DOMI_TENSOR_NC1HWC0, DOMI_TENSOR_NC1HWC0})
```

5.9.2 插件代码样例

步骤1 修改“customop_te/plugin/caffe_reduction_layer.cpp”代码文件。

- 修改“FilePath”为当前算子文件所在绝对路径+算子py文件名称。

  FilePath = "*project_path*/operator/reduction";

- 修改FuncName为reduction.py文件中算子实现的函数名称。

  FuncName ="reduction"

- 修改KernelName为reduction.py文件中设置的算子的KernelName。

  KernelName = "Reduction";

- 修改算子的二进制文件*.o及描述文件.json所在的路径。

  te_bin_info.bin_file_path = "./kernel_meta/" + KernelName + ".o";

  te_bin_info.json_file_path = "./kernel_meta/" + KernelName + ".json";

  此路径为相对于当前工程目录的相对路径，例如Reduction.o文件在当前工程目录的operator/kernel_meta目录下。

步骤2 修改后的reduction算子插件的代码样例如下所示：

```
#include <Python.h>
#include "custom/custom_op.h"
#include "framework/omg/register.h"
#include "framework/omg/omg_types.h"
#include "proto/caffe/caffe.pb.h"
#include "operator.h"
#include "attr_value.h"
#include <memory>
#include <string>
#include <vector>
using namespace ge;

namespace domi
{
// 用户自定义重写ParseParams函数，解析算子参数
Status CaffeReductionParseParams(const Message* op_origin, ge::Operator& op_dest)
{
    // 将op_origin转化为layer对象
    const caffe::LayerParameter* layer =
        dynamic_cast<const caffe::LayerParameter*>(op_origin);

    // 校验输入参数合法性
    if (nullptr == layer)
```

```
    {
        printf("Dynamic cast op_src to LayerParameter failed\n");
        return FAILED;
    }
    // 将算子参数中的枚举类型转换为map类型
    std::map<caffe::ReductionParameter_ReductionOp, std::string> operation_map = {
        { caffe::ReductionParameter_ReductionOp_SUM, "SUM" },
    { caffe::ReductionParameter_ReductionOp_ASUM, "ASUM" },
    { caffe::ReductionParameter_ReductionOp_SUMSQ, "SUMSQ" },
    { caffe::ReductionParameter_ReductionOp_MEAN, "MEAN" },
    };
    // 获取当前算子层的句柄
    const caffe::ReductionParameter& param = layer->reduction_param();
    // 将算子中所有参数赋值给离线模型算子数据结构op_dest对象
    if(param.has_axis())
    {
        op_dest.SetAttr("axis", AttrValue::CreateFrom<AttrValue::INT>(param.axis()));
    }
    if(param.has_coeff())
    {
        op_dest.SetAttr("coeff", AttrValue::CreateFrom<AttrValue::FLOAT>(param.coeff()));
    }
    if(param.has_operation())
    {
        op_dest.SetAttr("operation",
AttrValue::CreateFrom<AttrValue::STR>(operation_map[param.operation()]));
    }
    return SUCCESS;
}

// 自定义重写InferShapeAndType函数，获取算子的输出描述
Status CaffeReductionInferShapeAndType(const ge::Operator& op, vector<ge::TensorDesc>& v_output_desc)
{
    // 获取算子输入的TensorDesc对象
    auto tensorDesc      = op.GetInputDesc(0);
    // 获取输入shape
    auto shape = tensorDesc.GetShape();
    int64_t axis = -1;

    ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("Get axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to point to
the original 2d.
    if (axis < 0) axis -= 2;

    if (axis < 0) axis += shape.GetDimNum();

    if (axis < 0 || axis >= shape.GetDimNum())
    {
        printf("invalid axis:%d, dim_size:%d\n", (int32_t)axis, (int32_t)shape.GetDimNum());
        return PARAM_INVALID;
    }
    int32_t dimsize = (int32_t)shape.GetDimNum();
    int32_t idx = 0;
    for(idx=axis; idx<dimsize; idx++)
    {
        shape.SetDim(idx, 1);
    }
    // 将调整后的shape信息设置到tensorDesc
    tensorDesc.SetShape(shape);
    v_output_desc.push_back(tensorDesc);

    return SUCCESS;

}
```

```
// 编译算子，获取算子的二进制文件及算子描述文件
Status CaffeReductionBuildTeBin(const ge::Operator& op, TEBinInfo& te_bin_info)
{
    // 算子相关参数声明
    std::string FilePath   = "";
    std::string FuncName   = "";
    std::string KernelName = "";

    std::string operation  = "";
    int64_t     axis       = -1;
    float       coeff      = 1;
    // 获取算子参数值
    ge::AttrValue operationAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("operation", operationAttrValue)) || (ge::GRAPH_SUCCESS !=
operationAttrValue.GetValue<AttrValue::STR>(operation)))
    {
        // Add exception handling and maintenance information.
        printf("GetOpAttr operation failed!\n");
    }

    // Parse the axis parameter.
    ge::AttrValue axisAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("axis", axisAttrValue)) || (ge::GRAPH_SUCCESS !=
axisAttrValue.GetValue<AttrValue::INT>(axis)))
    {
        printf("GetOpAttr axis failed!\n");
    }
    // In the OM model, all shape are supplemented to 4d. In this case, axis needs to be repaired to point to
the original 2d.
    if(axis < 0)
        axis -= 2;

    // Parse the coeff parameter.
    ge::AttrValue coeffAttrValue;
    if ((ge::GRAPH_SUCCESS != op.GetAttr("coeff", coeffAttrValue)) || (ge::GRAPH_SUCCESS !=
coeffAttrValue.GetValue<AttrValue::FLOAT>(coeff)))
    {
        printf("GetOpAttr coeff failed!\n");
    }
    // 获取算子输入描述信息
    TensorDesc input_desc      = op.GetInputDesc(0);

    // 解析Input shape并校验算子的Dim是否为4
    if(input_desc.GetShape().GetDimNum() != 4)
    {
        printf("The shape size is %d, which is not 4!", (int32_t)input_desc.GetShape().GetDimNum());
        return FAILED;
    }
    // 设置算子文件路径
    FilePath   = "/home/ascend/tools/projects/customop_te/operator/reduction";
    // 设置实现算子的函数名
    FuncName   = "reduction";
    // 设置算子实现文件中定义的算子名称
    KernelName = "Reduction";

    // i => int; s => string; f => dobule; O => bool, and bool value is Py_True or Py_False，调用Tensor Engine
的编译接口编译算子
    te::BuildTeCustomOp(te_bin_info.ddk_version, op.GetName(), FilePath, FuncName,
                    "(i,i,i,i), s, i, s, f, s", input_desc.GetShape().GetDim(0), input_desc.GetShape().GetDim(1),
                    input_desc.GetShape().GetDim(2), input_desc.GetShape().GetDim(3), "float16", axis,
operation.c_str(), coeff,
                    KernelName.c_str());    //此处的参数顺序需要与算子实现函数中参数顺序保持一致。

    // set te op json to te_bin_info
    te_bin_info.bin_file_path  = "./kernel_meta/" + KernelName + ".o";
    te_bin_info.json_file_path = "./kernel_meta/" + KernelName + ".json";
```

```
    return SUCCESS;
}

REGISTER_CUSTOM_OP("custom_reduction") //####test_reduction is the type name of the operator in the
OM model. It can be specified randomly and cannot be the same as an existing type name. It is case
sensitive.
    .FrameworkType(CAFFE)  // Enumerated type. The options are as follows: CAFFE, TENSORFLOW
    .OriginOpType("Reduction")  // Reduction indicates the type name of the operator in the caffe
framework.
    .ParseParamsFn(CaffeReductionParseParams)  // Op parameters parse function
    .InferShapeAndTypeFn(CaffeReductionInferShapeAndType)       // Set output description and datatype
function
    .TEBinBuildFn(CaffeReductionBuildTeBin)           // Build Te op binary function
    .ImplyType(ImplyType::TVM);        // Implementation type. Enumerated type, The options are as follows:
TVM, AI_CPU.

}  // namespace domi
```

----结束

5.9.3 编译算子插件

步骤1 以DDK安装用户登录DDK所在服务器。

步骤2 在“projects/customop_te/plugin”目录下修改Makefile文件。

cd $HOME/tools/projects/customop_te/plugin/

vi Makefile

- 修改生成的算子插件名称。

```
ll : libcaffe_reduction_layer.so lib_caffe_parser.so
……bian
libcaffe_reduction_layer.so: $(OBJS_customop)
        $(CC) -c $(CC_FLAGS) -o proto/caffe/caffe.pb.o proto/caffe/caffe.pb.cc
        $(CC) $^ $(LNK_FLAGS) -o $@

lib_caffe_parser.so: $(OBJS_no_customop)
        $(CC) -c $(CC_FLAGS) -o proto/caffe/caffe.pb.o proto/caffe/caffe.pb.cc
        @if [ -f $(LOCAL_DIR)/proto/caffe/caffe.proto ]; then $(CC) $^ proto/caffe/caffe.pb.o $
(LNK_FLAGS) -o $@; fi;
```

  libcaffe_reduction_layer.so即为生成的算子插件名称，用户可自行修改。

  lib_caffe_parser.so是生成的解析caffe.proto文件的库文件，名称不可修改。若是Caffe算子，请确保同一模型中的所有不支持的自定义算子已在**5.8 caffe.proto文件算子定义（可选）**完成定义。

- 将“TOPDIR”指定到DDK的安装目录，例如

```
ifeq ($(DDK_PATH),)
TOPDIR      := $(HOME)/tools/che/ddk/ddk
else
TOPDIR := $(DDK_PATH)
endif
```

- Makefile中其他内容无需改动，为算子插件编译通用模板。

步骤3 执行编译操作。

在当前plugin目录下执行如下命令进行算子插件的编译。

make

编译完成后会在当前目录下生成算子插件文件**libcaffe_reduction_layer.so**。

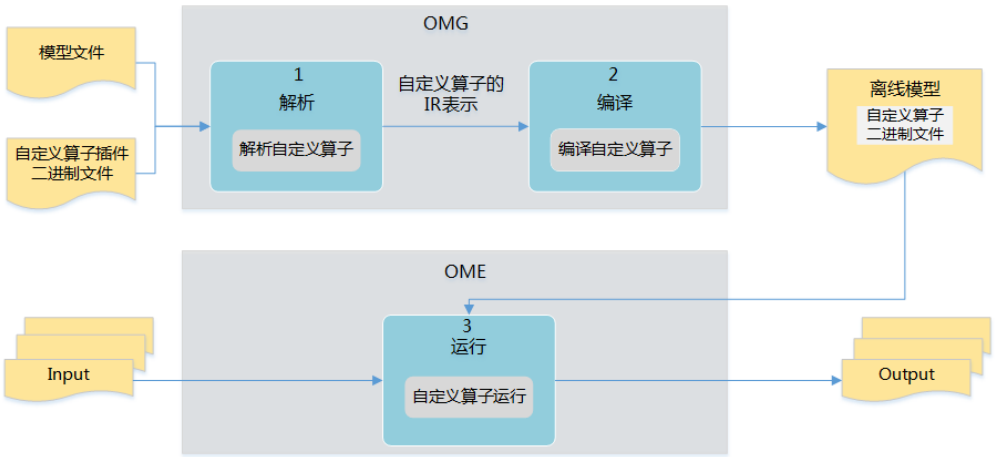
----结束

5.10 加载插件转换模型

原理说明

加载了用户自定义算子插件的模型转换流程如图5-14所示

图 5-14 加载插件进行模型转换流程



1. 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。
2. 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，并生成离线模型（*.om）文件
3. 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。

操作流程

步骤1 以DDK安装用户进入自定义算子开发工程根目录。

cd $HOME/tools/projects/customop_te/

步骤2 执行如下命令进行模型转换。

omg --model=*model/deploy_mylenet-1.prototxt* --weight=*model/mylenet-1.caffemodel* --framework=0 --plugin_path=*plugin* --output=*mylenet* --ddk_version=*1.3.T18.B850*

- --model项的值为mylenet网络的原始模型文件的相对路径。
- --weight项的值为mylenet网络的预训练模型文件的相对路径。
- --framework为原始框架类型。
  - 0：caffe
  - 3：tensorflow
- --plugin项的值为自定义算子插件所在目录。

- --output项的值为输出模型文件的名称，用户可自定义。
- --ddk_version项的值为自定义算子运行需要匹配DDK的版本号，可在“$HOME/tools/che/ddk/ddk/ddk_info”文件中查看DDK版本号。

说明

更多参数说明请参考《模型转换指导》。

----结束

5.11 算子插件开发函数参考

5.11.1 ParseParamsFn 函数

函数功能

参数解析函数。

函数原型

OpRegistrationData& ParseParamsFn(ParseParamFunc parseParamFn);

参数说明

参数	输入/输出	说明
parseParamFn	输入	回调函数ParseParamFunc，请参见**回调函数ParseParamFunc**。

回调函数 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。

函数原型

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

参数说明

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AI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”。

5.11.2 InferShapeAndTypeFn 函数

函数功能

维度推导函数。

函数原型

OpRegistrationData& InferShapeAndTypeFn(InferShapeFunc inferShapeFn);

参数说明

参数	输入/输出	说明
inferShapeFn	输入	回调函数InferShapeFunc，请参见**回调函数InferShapeFunc**。

回调函数 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。

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI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”。

5.11.3 TEBinBuildFn 函数

函数功能

build回调函数。

版权所有 © 华为技术有限公司

函数原型

OpRegistrationData& TEBinBuildFn(BuildTeBinFunc buildTeBinFn);

参数说明

参数	输入/输出	说明
buildTeBinFn	输入	回调函数BuildTeBinFunc，参见**回调函数BuildTeBinFunc**。

回调函数 BuildTeBinFunc

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

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI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;
		};

5.11.4 WeightFormats 函数

函数功能

设置算子权重支持的数据格式。

函数原型

```
OpRegistrationData& WeightFormats(
const std::initializer_list<domi::tagDomiTensorFormat>& weight_formats);
```

参数说明

参数	输入/输出	说明
weight_formats	输入	权重支持的数据格式列表： typedef enum tagDomiTensorFormat { DOMI_TENSOR_NCHW = 0, /**< NCHW */ DOMI_TENSOR_NHWC, /**< NHWC */ DOMI_TENSOR_ND, /**< Nd Tensor */ DOMI_TENSOR_NC1HWC0, /**< NC1HWC0 */ DOMI_TENSOR_FRACTAL_Z, /**< FRACTAL_Z */ DOMI_TENSOR_NC1C0HWPAD, DOMI_TENSOR_NHWC1C0, DOMI_TENSOR_FSR_NCHW, DOMI_TENSOR_FRACTAL_DECONV, DOMI_TENSOR_BN_WEIGHT, DOMI_TENSOR_CHWN, /*Android NN Depth CONV*/ DOMI_TENSOR_FILTER_HWCK, /* filter input tensor format */ DOMI_TENSOR_RESERVED } domiTensorFormat_t;

5.11.5 算子构建函数注册宏

宏功能

调用DOMI_REGISTER_OP完成算子注册。

宏原型

```
REGISTER_CUSTOM_OP(name)
```

参数说明

参数	输入/输出	说明
name	输入	算子在离线模型的类型名，可以随意指定，不可与已有类型名重复，区分大小写。

5.12 算子清单

5.12.1 参数说明

参数	含义说明
ni	批量的大小
ci/co	通道的数量
hi/ho/	高度
wi/wo	宽度
sh/sw	步长
kh/kw	卷积核的大小
window_h(window_y)/ window_w(window_x)	窗口的大小
dh(dilation_h)/ dw(dilation_w)	卷积膨胀系数
FilterHDilation/ FilterWDilation	卷积核膨胀完以后H/W维度的长度
FilterH/FilterW	卷积的权重参数的H/W维度的值
padWHead/padHHead	W/H维度补PAD后前端多出的一截
PadWTail/padHTail	W/H维度补PAD后尾端多出的一截
dilationsize	用户设置的膨胀系数的值
FilterSize	用户设置的卷积核的数目
INT32_MAX	数据类型int32能表示范围的最大值
ALIGN	向上对齐计算
CEIL	取整计算

5.12.2 算子边界

5.12.2.1 整体约束

对于caffe框架，当算子的输入维度不是4时，如果存在axis参数，不能使用负数

5.12.2.2 Caffe 算子边界

序号	算子	含义	边界
1	Absval	对输入求绝对值	【输入】 1个输入 【参数】 engine: 枚举型，默认为0（DEFAULT=0，CAFFE=1，CUDNN=2），可选 【约束】 无限制 【量化工具支持】 是
2	Argmax	返回输入的最大值对应的索引序号	【输入】 1个输入 【参数】 ● out_max_val: 布尔型，默认为false，可选 ● top_k：unit32，默认为1，可选 ● axis：int32，可选 【约束】 无限制 【量化工具支持】 否

序号	算子	含义	边界
3	BatchNorm	对输入做标准化 （x - avg(x)）/x的标准差	【输入】 1个输入 【参数】 ● use_global_stats: 布尔型，必须为true ● moving_average_fraction：float，默认为0.999，可选 ● eps：float，默认为1e-5，可选 【约束】 仅支持对C维度方向做归一化 【量化工具支持】 是
4	Concat	输入数据按维度拼接	【输入】 多个输入 【参数】 ● concat_dim: uint32，默认为1，取值：大于0，可选 ● axis：int32，默认1，可选；与concat_dim功能相同，二者择其一；axis==-1时，输入维度必须等于4，否则结果可能错误 【约束】 ● 输入的tensor，除了进行concat的维度外，其他维度的size必须相等 ● 输入tensor的个数范围不超过[1,1000] 【量化工具支持】 是

序号	算子	含义	边界
5	ConvolutionDepthwise	深度卷积	【输入】 1个输入，且filter为常量，维度为4 【参数】 ● num_output: uint32，可选 ● bias_term：布尔型，默认为true，可选 ● pad：uint32，默认为0；数组 ● kernel_size：uint32；数组 ● stride：uint32，默认为1；数组 ● dilation：uint32，只支持dilation=1；数组 ● pad_h：uint32，默认为0，可选（仅2D） ● pad_w：uint32，默认为0，可选（仅2D） ● kernel_h：uint32，可选（仅2D） ● kernel_w：uint32，可选（仅2D） ● stride_h：uint32，可选（仅2D） ● stride_w：uint32，可选（仅2D） ● group：uint32，默认为1，可选

序号	算子	含义	边界
			● weight_filler：不支持设置该参数 ● bias_filler：不支持设置该参数 ● engine：枚举型，默认为0（DEFAULT=0，CAFFE=1，CUDNN=2），可选 ● force_nd_im2col：布尔型，默认为false，可选 ● axis: int32，默认为1，可选 【约束】 filterN=inputC=group； (W+15)/16*16)*filter.W*32 <= 32*1024，其中W是算子输入的W，filter.W是filter的W。 【量化工具支持】 是

序号	算子	含义	边界
6	Convolution	卷积	【输入】 1个输入，且filter为常量，维度为4 【参数】 ● num_output: uint32，可选 ● bias_term：布尔型，默认为true，可选 ● pad：uint32，默认为0；数组 ● kernel_size：uint32；数组 ● stride：uint32，默认为1；数组 ● dilation：uint32，默认为1；数组 ● pad_h：uint32，默认为0，可选（仅2D） ● pad_w：uint32，默认为0，可选（仅2D） ● kernel_h：uint32，可选（仅2D） ● kernel_w：uint32，可选（仅2D） ● stride_h：uint32，可选（仅2D） ● stride_w：uint32，可选（仅2D） ● group：uint32，默认为1，可选 ● weight_filler：不支持设置该参数

序号	算子	含义	边界
			● bias_filler：不支持设置该参数 ● engine：枚举型，默认为0（DEFAULT=0，CAFFE=1，CUDNN=2），可选 ● force_nd_im2col：布尔型，默认为false，可选 ● axis: int32，默认为1，可选 【约束】 ● (inputW + padWHead + padWTail) >= (((FilterW- 1) * dilationW) + 1) ● (inputW + padWHead + padWTail) / StrideW + 1 <= 2147483647 ● (inputH + padHHead + padHTail) >= (((FilterH- 1) * dilationH) + 1) ● (inputH + padHHead + padHTail) / StrideH + 1 <= 2147483647 ● 0 <= Pad < 256, 0 < FilterSize < 256, 0 < Stride < 64, 1 <= dilationsize < 256 【量化工具支持】 是

序号	算子	含义	边界
7	Crop	截取	【输入】 2个输入 【参数】 ● axis: int32，默认为2，axis=-1，input dim必须等于4，可选 ● offset：uint32，数组 【约束】 无限制 【量化工具支持】 否

序号	算子	含义	边界
8	Deconvolution	反卷积	【输入】 1个输入，且filter为常量，维度为4 【参数】 ● num_output: uint32，可选 ● bias_term：布尔型，默认为true，可选 ● pad：uint32，默认为0；数组 ● kernel_size：uint32；数组 ● stride：uint32，默认为1；数组 ● dilation：uint32，默认为1；数组 ● pad_h：uint32，默认为0，可选（仅2D） ● pad_w：uint32，默认为0，可选（仅2D） ● kernel_h：uint32，可选（仅2D） ● kernel_w：uint32，可选（仅2D） ● stride_h：uint32，可选（仅2D） ● stride_w：uint32，可选（仅2D） ● group：uint32，默认为1，可选 ● weight_filler：不支持设置该参数

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			● bias_filler：不支持设置该参数 ● engine：枚举型，默认为0（DEFAULT=0，CAFFE=1，CUDNN=2），可选 ● force_nd_im2col：布尔型，默认为false，可选 ● axis: int32，默认为1，可选 【约束】 ● group = 1 ● dilation = 1 ● filterH - padHHead - 1 >= 0 ● filterW - padWHead - 1 >= 0 ● 还有一条约束涉及中间变量，公式如下： 1、a = ALIGN(filter_num, 16) * ALIGN(filter_c,16) * filter_h * filter_w * 2； 如果ALIGN(filter_c, 16)%32 = 0，a = a/2； 2、conv_input_width =(反卷积输入w - 1) * strideW + 1； 3、b =（conv_input_width）* filter_h * ALIGN(filter_num, 16) * 2 * 2；

序号	算子	含义	边界
			4、a + b <= 1024*1024。 【量化工具支持】是

序号	算子	含义	边界
9	DetectionOutput	检测结果输出FSR	【输入】 3个输入 【参数】 ● num_classes: 必选，类型：int32，要预测的类数 ● share_location: 可选，类型: bool，默认为true（表示不同类间共享bounding box） ● background_label_id: 可选，类型：int32，默认为0 ● nms_param: 可选，非最大抑制 ● save_output_param: 可选，用于保存检测结果 ● code_type: 可选，默认为CENTER_SIZE ● variance_encoded_in_target: 可选，类型：bool，默认为true；如果为true，方差编码在目标中，否则需要相应地调整预测偏移量 ● keep_top_k: 可选，类型：int32，在nms步骤后每个图像要保留的总bbox数； ● confidence_threshold: 可选，类型：float，仅考虑置信度大于阈值的检测；

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			如果没有设置，考虑所有的box ● nms_threshold: 可选，类型：float ● top_k: 可选，类型：int32 ● boxes: 可选，类型：int32，默认为1 ● relative: 可选，类型：bool，默认为true ● objectness_threshold，可选，类型：float，默认为0.5 ● class_threshold: 可选，类型：float，默认为0.5 ● biases: 数组 ● general_nms_param: 可选 【约束】 ● 支持fasterrcnn网络 ● 非极大值抑制比nmsThreshold∈(0,1) ● 概率阈值postConfThreshold∈(0,1) ● 类别>=2 ● 最大支持1024个框输入 ● 输出的w维度是16 【量化工具支持】 是

序号	算子	含义	边界
10	Eltwise	按元素操作层(求和、乘积、最大值)	【输入】 至少2个输入 【参数】 ● operation：可选，类型：枚举型（PROD = 0; SUM = 1; MAX = 2），默认为SUM ● coeff：数组；类型：float ● stable_prod_grad：可选，类型：bool，默认为true 【约束】 ● 最大支持4个输入 ● 与原生算子相比，不支持stable_prod_grad参数 ● 支持prod、sum、max三种模式 【量化工具支持】 是
11	Elu	激活函数	【输入】 1个输入 【参数】 alpha (optional)：类型：float，默认为1 【约束】 无限制 【量化工具支持】 否

序号	算子	含义	边界
12	Exp	对输入求以e为底的x次方值	【输入】 1个输入 【参数】 ● base：可选，类型：float，默认为-1.0 ● scale：可选，类型：float，默认为1.0 ● shift：可选，类型：float，默认为0.0 【约束】 无限制 【量化工具支持】 否
13	Flatten	输入n*c*h*w 变为向量 n* (c*h*w)	【输入】 1个输入 （top_size与bottom_size必须不一致，且不为1； 当axis=-1时，输入的dim必须等于4） 【参数】 ● axis：可选，类型：int32，默认为1 ● end_axis：可选，类型：int32, 默认为-1 【约束】 axis必须小于end axis 【量化工具支持】 是

序号	算子	含义	边界
14	FullConnection	全连接	【输入】 1个输入 【参数】 ● num_output: 可选，类型：uint32 ● bias_term: 可选，类型：bool，默认为true ● weight_filler: 不支持设置该参数 ● bias_filler: 不支持设置该参数 ● axis: 可选，类型：int32，默认为1 ● transpose: 可选，类型：bool，默认为false 【约束】 ● 仅支持transpose=false，axis=1； ● 量化场景下，Bais_C <= 59136；非量化场景下，Bais_C <= 118272； ● 如果客户要量化模型时，需要满足下列维度： ● 当N = 1，2 * CEIL(C,16) * 16 * xH * xW<= 1024 * 1024 ● 当N > 1，2 * 16 * CEIL(C,16) * 16 * xH * xW <= 1024 * 1024 【量化工具支持】 是

序号	算子	含义	边界
15	Interp	插值层	【输入】 1个输入 【参数】 ● height：可选，类型：int32，默认为0 ● width：可选，类型：int32, 默认为0 ● zoom_factor：可选，类型：int32, 默认为1 ● shrink_factor：可选，类型：int32, 默认为1 ● pad_beg：可选，类型：int32, 默认为0 ● pad_end：可选，类型：int32, 默认为0 说明： ● zoom_factor与shrink_factor不能同时存在 ● height与zoom_factor不能同时存在 ● height与shrink_factor不能同时存在 【约束】 （outputH*outputW）/（inputH*inputW）> 1/30 【量化工具支持】 否

序号	算子	含义	边界
16	Log	对输入进行log计算	【输入】 1个输入 【参数】 ● base：可选，类型：float，默认为-1.0 ● scale：可选，类型：float，默认为1.0 ● shift：可选，类型：float，默认为0.0 【约束】 无限制 【量化工具支持】 否

序号	算子	含义	边界
17	LRN	局部响应归一化层	【输入】 1个输入，不支持常量输入 【参数】 ● local_size：可选，类型：uint32，默认为5 ● alpha：可选，类型：float，默认为1. ● beta：可选，类型：float，默认为0.75 ● norm_region：可选，类型：枚举型，默认为ACROSS_CHANNELS，（ACROSS_CHANNELS=0;WITHIN_CHANNEL = 1） ● lrnk：可选，类型：float，默认为1. ● engine：可选，类型：枚举型（DEFAULT = 0;CAFFE = 1; CUDNN = 2） 【约束】 ● local_size >0，且必须为奇数 ● 通道间： 当local_size∈[1,15]时，lrnK>0.00001且beta>0.01，否则lrnK和beta为任意值；LrnK和alpha不同时为0；当C维度大于1776时，local_size <1728； 通道内： LrnK=1，

序号	算子	含义	边界
			local_size∈[1,15]，beta>0.01； 【量化工具支持】 是

序号	算子	含义	边界
18	LSTM	长短期记忆网络	【输入】 2个或3个输入 X: 时间序列数据(T*B*Xt)，按4D的NCHW格式排布传入，需要保证：N对应时间序列长度T，C对应Batch数B，H对应t时刻的输入数据Xt，W取1； Cont：序列连续性标志(T*B) Xs: 静态数据(B*Xt)，可选 【参数】 ● num_output：可选，类型：uint32，默认为0 ● weight_filler：不支持设置该参数 ● bias_filler：不支持设置该参数 ● debug_info：可选，类型：bool，默认为false ● expose_hidden：可选，类型：bool，默认为false 【约束】 ● 此约束涉及中间变量计算，公式如下，其中ht和output对应属性num_output： a = (ALIGN(xt,16) + ALIGN(output, 16)) * 16 * 2 * 2 b = (ALIGN(xt,16) + ALIGN(output,

序号	算子	含义	边界
			16)) * 16 * 4 * 2 * 2 d = 16 * ALIGN(ht, 16) * 2 e = batchNum* 4 则，约束为： a + b <= 1024 * 1024； d <= 256*1024 / 8； e <= 256*1024 / 32； ● B<=16，T<=768。 【量化工具支持】 否

序号	算子	含义	边界
19	Normalize	标准化层	【输入】 1个输入 【参数】 ● across_spatial：可选，类型：bool，默认为true ● scale_filler：不支持设置该参数 ● channel_shared：可选，类型：bool，默认为true ● eps：可选，类型：float，默认为1e-10 【约束】 ● eps应大于1e-7，并且小于等于0.1+(1e-6)； ● caffe框架中的参数across_spatial目前只支持true，按channel进行norm操作。 【量化工具支持】 是
20	Permute	将输入维度按照给定模式进行重排	【输入】 1个输入 【参数】 ● order: uint32，数组 【约束】 无约束 【量化工具支持】 是

序号	算子	含义	边界
21	Pooling	池化层	【输入】 1个输入 【参数】 ● pool: 池化方法，可选，类型：枚举，取值：MAX=0，AVE=1，STOCHASTIC = 2，默认为MAX ● pad: 可选，类型：uint32，默认为0 ● pad_h: 可选，类型：uint32，默认为0 ● pad_w: 可选，类型：uint32，默认为0 ● kernel_size: 可选，类型：uint32，kernel_size和kernel_h/kernel_w不能同时出现 ● kernel_h: 可选，类型：uint32 ● kernel_w: 可选，类型：uint32，kernel_h/kernel_w必须同时存在 ● stride: 可选，类型：uint32，默认为1 ● stride_h: 可选，类型：uint32 ● stride_w: 可选，类型：uint32 ● engine: 可选，类型：枚举，取

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			值：DEFAULT=0，CAFFE=1，CUDNN=2 ● global_pooling: 可选，类型：bool，默认值为false ● ceil_mode: 可选，类型：bool，默认为true ● round_mode: 可选，类型：枚举，取值：CEIL=0，FLOOR=1；默认为CEIL 【约束】 ● kernelH<=inputH+padTop+padBottom ● kernelW<=inputW+padLeft+padRight ● padTop<windowH ● padBottom<windowH ● padLeft<windowW ● padRight<windowW ● 除公共约束外，还需满足下列某一场景支持范围 global pool模式只支持以下范围 1) outputH==1 && outputW==1 && kernelH>=inputH && kernelW>=inputW

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			2) inputH*inputW<=10000 【量化工具支持】 是
22	Power	计算 y =（scale*x +shift）^ power	【输入】 1个输入 【参数】 ● power: 可选，类型：float，默认为1.0 ● scale: 可选，类型：float，默认为1.0 ● shift: 可选，类型：float，默认为0.0 【约束】 ● power!=1 ● scale*x+shift>0 【量化工具支持】 是
23	Prelu	激活函数	【输入】 1个输入 【参数】 ● filler: 不支持设置该参数 ● channel_shared: 可选，类型：bool，默认为false，是否跨channel共享斜率参数 【约束】 无限制 【量化工具支持】 是

序号	算子	含义	边界
24	PriorBox	从目标预选框获取真实目标的位置	【输入】 2个输入，包括： ● 模型的原始输入图片，数据格式：NCHW； ● 特征图，数据格式：NCHW。 【参数】 ● min_size: 必须设置，最小框大小（以像素为单位） ● max_size: 必须设置，最大框大小（以像素为单位） ● aspect_ratio: 数组；类型：float；各种宽高比，重复的比率将被忽略，如果没有提供，使用默认比率1 ● flip: 可选，类型：bool，默认为true；如果为true，将翻转每个宽高比，例如，如果有宽高比‘r’，也将生成宽高比‘1.0/r’ ● clip: 可选，类型：bool，默认为false；如果为true，将剪切先前的值，使其在[0, 1]范围内 ● variance: 数组；调整先前bbox的方差 ● img_size: 可选，类型：uint32，img_size与img_h/img_w不能同时存在

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			● img_h: 可选，类型：uint32 ● img_w: 可选，类型：uint32 ● step: 可选，类型：float，step与step_h/step_w不能同时存在 ● step_h: 可选，类型：float ● step_w: 可选，类型：float ● offset: 类型：float，默认为0.5 【约束】 只支持ssd网络。 输出维度：[n,2,检测框*4,1]。 【量化工具支持】 是

序号	算子	含义	边界
25	Proposal	将预选框通过（proposal, score）排序，通过nms获取topN proposal	【输入】 3个输入（scores，bbox_pred，im_info） 【参数】 ● feat_stride: 可选，类型：float ● base_size: 可选，类型：float ● min_size: 可选，类型：float ● ratio: 数组；类型：float ● scale: 数组；类型：float ● pre_nms_topn: 可选，类型：int32 ● post_nms_topn: 可选，类型：int32 ● nms_thresh: 可选，类型：float 【约束】 只用于fastercnn ● ProposalParameter、PythonParameter不能同时存在 1.preTopK范围为1~6144 2.postTopK范围为1~1024 3.scaleCnt*ratioCnt的最大值支持到64 4.nmsTresh:过滤框使用的阈值，0<nmsTresh<=1

序号	算子	含义	边界
			5.minSize:框的边长的最小值，所有小于此最小值的框将会被过滤掉 6.featStride:在生成默认框时，指定两个相邻的框延H或W的步长 7.baseSize:用来生成默认框用到的参数，表示框的基本大小 8.ratio & scale:生成默认框用到的参数 9.imgH&imgW:输入到网络的图片高和宽，其值必须大于0 ● input维度约束: clsProb：C=2*scaleCnt*ratioCnt bboxPred：C=4*scaleCnt*ratioCnt bboxPrior：N=clsProb.N，C=4*scaleCnt*ratioCnt imInfo：N=clsProb.N，C=3 【量化工具支持】 是

序号	算子	含义	边界
26	PSROIPooling	位置敏感的区域池化	【输入】 2个输入 【参数】 ● spatial_scale: 必须配置，类型: float ● output_dim: 必须配置，类型: int32，输出通道数 ● group_size: 必须配置，类型: int32，编码位置敏感分数图的组数 【约束】 用于RFCN网络 ● 输入Roi框的坐标信息为[roiN, roiC, roiH, roiW]，格式范围是：1<= roiN <=65535，roiC == 5，roiH == 1，roiW == 1； ● 输入的featuremap维度为[xN,xC,xH,xW] ● pooledH==pooledW==groupSize <= 128 [pooledH pooledW]表示pool框的长宽 输出的格式 y[yN, yC, yH, yW] ● poolingMode == avg pooling，pooledH == pooledW == groupSize，pooledH <= 128，spatialScale >

版权所有 © 华为技术有限公司

序号	算子	含义	边界
			0， groupSize > 0，outputDim > 0； ● 1<=xN<=65535， roisN % xN == 0； ● HW_LIMIT是xh、xw的约束 xHW = xH * xW， pooledHW = pooledH * pooledW， HW_LIMIT = (64*1024–8*1024) / 32， xH >= pooledH，xW >= pooledW， xHW >= pooledHW， xHW/pooledHW<=HW_LIMIT； ● 多batch场景时，每个batch的roi框个数相同，且roi的batch排列顺序与feature相同。 【量化工具支持】 是

序号	算子	含义	边界
27	Relu	激活函数，同时包含普通的relu和leaky relu，可通过参数指定	【输入】 1个输入 【参数】 ● negative_slope: 可选，类型：float，默认为0 ● engine: 可选，类型：枚举，取值：DEFAULT=0，CAFFE=1，CUDNN=2 【约束】 无限制 【量化工具支持】 是
28	Reshape	改变输入维度	【输入】 1个输入 【参数】 ● shape: 常量，类型：int64或int ● axis: 可选，类型：int32，默认为0 ● num_axes: 可选，类型：int32，默认为-1 【约束】 无限制 【量化工具支持】 是

序号	算子	含义	边界
29	ROIAlign	一种区域特征聚集的方式	【输入】 至少有2个输入 【参数】 ● pooled_h: 可选，类型：uint32，默认为0 ● pooled_w: 可选，类型：uint32，默认为0 ● spatial_scale: 可选，类型：float，默认为1 ● sampling_ratio: 可选，类型：int32，默认为-1 【约束】 主要用于maskrcnn FeatureMap（特征图）约束： ● H * W <= 5248 或 W * C < 40960； ● C <= 1280； ● ((C-1)/128+1)*pooled W <= 216； RoI（感兴区域）约束： ● C=5(caffe)，H=1，W=1； ● samplingRatio* pooledW <= 128且 samplingRatio* pooledH <= 128； ● H >= pooledH, W >= pooledW。 【量化工具支持】 是

序号	算子	含义	边界
30	ROIPooling	将“候选框”映射到特征图上	【输入】 至少有2个输入 【参数】 ● pooled_h: 必须设置，类型：uint32，默认为0 ● pooled_w: 必须设置，类型：uint32，默认为0 ● spatial_scale: 必须设置，类型：float；默认为1；乘法空间比例因子将ROI坐标从其输入比例转换为pool时使用的比例 【约束】 主要用于fasterrcnn ● 输入维度：H * W<=8160； 且H <=120； 且W<=120； ● 输出维度：pooledH <=20, 且pooledW <=20。 【量化工具支持】 是

序号	算子	含义	边界
31	Scale	out=alpha*Input +beta	【输入】 2个输入，input的dim 为4 【参数】 ● axis: 可选，类型：int32，默认为1，仅支持axis为1或者-3 ● num_axes: 可选，类型：int32，默认为1 ● filler: 不支持设置该参数 ● bias_term: 可选，类型：bool，默认为false；是否也学习bias（相当于ScaleLayer + BiasLayer，但可能更有效率 ● bias_filler: 不支持设置该参数 【约束】 scale,bias的shape只支持（n,c,1,1），且c维度与input的c维度相等； 【量化工具支持】 是
32	ShuffleChannel	帮助信息在特征通道交叉流动	【输入】 1个输入 【参数】 group: 可选，类型：uint32，默认为1 【约束】 无限制 【量化工具支持】 是

序号	算子	含义	边界
33	Sigmoid	激活函数	【输入】 1个输入 【参数】 engine: 可选，类型：枚举，取值：DEFAULT=0，CAFFE=1，CUDNN=2 【约束】 无限制 【量化工具支持】 是
34	Slice	将输入分解成多个输出	【输入】 1个输入 【参数】 ● slice_dim: 可选，类型：uint32，默认为1；axis和slice_dim不能同时存在 ● slice_point: 数组；类型：uint32 ● axis: 可选，类型: int32，默认为1（表示沿channel拼接）； 【约束】 无限制 【输出】 无限制 【量化工具支持】 是

序号	算子	含义	边界
35	Softmax	归一化逻辑函数	【输入】 1个输入 【参数】 ● engine: 可选，取值：DEFAULT=0，CAFFE=1，CUDNN=2 ● axis: 可选，类型：int32，默认为1；表示沿哪个axis作softmax 【约束】 输入4维时可以针对每一维做softmax： 根据分类的轴不同，计算边界分别为 axis=1时，C<=（(256*1024/4)-8*1024-256）/2； axis=0时，N<=(56*1024-256)/2； axis=2时，W=1，0<H<（1024*1024/32）； axis=3时，0<W<（1024*1024/32） 输入维度不足4维时，仅支持对最后一维做softmax计算，并且最后一维不超过46080。 【量化工具支持】 是

序号	算子	含义	边界
36	Tanh	激活函数	【输入】 1个输入 【参数】 engine: 可选，类型：枚举，取值：DEFAULT=0，CAFFE=1，CUDNN=2 【约束】 输入tensor元素个数不超过INT32_MAX 【量化工具支持】 是
37	Upsample	maxpool的反向传播过程	【输入】 2个输入 【参数】 scale: 可选，类型：int32，默认为1 【约束】 无限制 【量化工具支持】 是

序号	算子	含义	边界
38	SSDDetectionOutput	SSD网络检测输出	【输入】 3个输入 【参数】 ● num_classes: 必选，类型：int32，要预测的类数 ● share_location: 可选，类型: bool，默认为true（表示不同类间共享bounding box） ● background_label_id: 可选，类型：int32，默认为0 ● nms_param: 可选，非最大抑制 ● save_output_param: 可选，用于保存检测结果 ● code_type: 可选，默认为CENTER_SIZE ● variance_encoded_in_target: 可选，类型：bool，默认为true；如果为true，方差编码在目标中，否则需要相应地调整预测偏移量 ● keep_top_k: 可选，类型：int32，在nms步骤后每个图像要保留的总bbox数； ● confidence_threshold: 可选，类型：float，仅考虑置信度大于阈值的检测；

序号	算子	含义	边界
			如果没有设置，考虑所有的box ● nms_threshold: 可选，类型：float ● top_k: 可选，类型：int32 ● boxes: 可选，类型：int32，默认为1 ● relative: 可选，类型：bool，默认为true ● objectness_threshold，可选，类型：float，默认为0.5 ● class_threshold: 可选，类型：float，默认为0.5 ● biases: 数组 ● general_nms_param: 可选 【约束】 ● 只支持SSD网络 ● preTopK和postTopK的取值范围当前仅支持1~1024； ● shareLocation仅支持true； ● nmsEta 仅支持1； ● numClasses支持的范围是1~2048； ● code_type仅支持CENTER_SIZE； ● nms_threshold和confidence_threshold的范围为0.0~1.0

序号	算子	含义	边界
			【量化工具支持】 是
39	Reorg	实时物体检测	【输入】 1个输入 【参数】 ● stride: 可选，类型：uint32，默认为2 ● reverse: 可选，类型：bool，默认为false 【约束】 只用于YOLOV2 【量化工具支持】 否
40	Reverse	逆转	【输入】 1个输入 【参数】 ● axis: 可选，类型：int32，默认为1；控制需翻转的数据轴，内容的布局不会被颠倒 【约束】 无限制 【量化工具支持】 否
41	LeakyRelu	LeakyRelu激活函数	【输入】 1个输入 【参数】 同Relu 【约束】 无限制 【量化工具支持】 是

序号	算子	含义	边界
42	YOLODetectionOutput	YOLO网络检测输出	【输入】 4个输入 【参数】 ● num_classes: 必选，类型：int32，要预测的类数 ● share_location: 可选，类型: bool，默认为true（表示不同类间共享bounding box） ● background_label_id: 可选，类型：int32，默认为0 ● nms_param: 可选，非最大抑制 ● save_output_param: 可选，用于保存检测结果 ● code_type: 可选，默认为CENTER_SIZE ● variance_encoded_in_target: 可选，类型：bool，默认为true；如果为true，方差编码在目标中，否则需要相应地调整预测偏移量 ● keep_top_k: 可选，类型：int32，在nms步骤后每个图像要保留的总bbox数； ● confidence_threshold: 可选，类型：float，仅考虑置信度大于阈值的检测；

序号	算子	含义	边界
			如果没有设置，考虑所有的box ● nms_threshold: 可选，类型：float ● top_k: 可选，类型：int32 ● boxes: 可选，类型：int32，默认为1 ● relative: 可选，类型：bool，默认为true ● objectness_threshold，可选，类型：float，默认为0.5 ● class_threshold: 可选，类型：float，默认为0.5 ● biases: 数组 ● general_nms_param: 可选 【约束】 ● 只用于YOLOV2 ● classNUm<10240；anchorBox = 5； ● W <= 1536； ● yolodetectionoutput的上层必须为yoloregion算子。 【量化工具支持】 否

5.12.2.3 Tensorflow 算子边界

序号	Python API	C++ API	边界
1	tf.nn.avg_pool	AvgPool Type：Pooling	【参数】 ● value: 4-D tensor，格式：[batch, height, width, channels]，数据类型：float32 ● ksize: 包含四个int 的列表或元组，其中每个值对应相应维度的窗口大小 ● strides: 包含四个int 的列表或元组，其中每个值对应相应维度的滑动步长 ● padding: 类型：string，值必须为’VALID’或’SAME’ ● data_format: 类型：string，值为'NHWC'（默认值）或 'NCHW' ● name: 可选参数，类型：string 【约束】 ● kernelH<=inputH+padTop+padBottom ● kernelW<=inputW+padLeft+padRight ● padTop<windowH ● padBottom<windowH ● padLeft<windowW ● padRight<windowW ● 除公共约束外，还需满足下列某一场景支持范围 global pool模式只支持下列场景 1) outputH==1 && outputW==1 && kernelH>=inputH && kernelW>=inputW 2) inputH*inputW<=10000 【输出】 1个tensor，数据类型与value相同 【量化工具支持】 是

序号	Python API	C++ API	边界
2	tf.math.reduce_mean	Mean	【参数】 参数同AvgPool算子的参数 【约束】 Mean适配AvgPool，约束条件： ● keep_dims=true。如果keep_dims=false，后继算子必须是Reshape。 ● 只能对HW维度做平均，即format=NCHW时，axis = [2,3]；format=NHWC时，axis=[1,2]。 【输出】 1个tensor，数据类型与value相同 【是否支持量化推理】 是
3	tf.nn.max_pool	MaxPool	同tf.nn.avg_pool

序号	Python API	C++ API	边界
4	tf.nn.conv2d	Conv2D	【参数】 ● value: 4-D tensor，格式：[batch, height, width, channels] 数据类型：float32 ● filter: 1个常量tensor，数据类型与维度与value相同，[filter_height, filter_width, in_channels, out_channels] ● strides: 非空，包含四个int 的列表或元组，其中每个值对应相应维度的滑动步长 ● padding: 非空，类型：string，值必须为’VALID’ 或’SAME’ ● use_cudnn_on_gpu: 类型: bool，默认为’True’ ● data_format: 非空，类型：string，值为 'NHWC'（默认值）或 'NCHW' ● dilations: 可选参数，1个int列表（长度为4）；默认为[1,1,1,1]，对应输入的每一维；如果k>1，相应维度做filter间跳过k-1个单元；维度顺序由data_format决定；dilations的batch与depth维度上的值必须是1 ● name: 可选参数，类型：string 【约束】 ● (inputW + padWHead + padWTail) >= (((FilterW- 1) * dilationW) + 1) ● (inputW + padWHead + padWTail) /StrideW + 1 <= INT32_MAX ● (inputH + padHHead + padHTail) >= (((FilterH- 1) * dilationH) + 1) ● (inputH + padHHead + padHTail) /StrideH + 1 <= INT32_MAX ● 0 <= Pad < 256, 0 < FilterSize < 256, 0 < Stride < 64, 1 <= dilationsize < 256 【输出】 Tensor，数据类型与Value相同 【量化工具支持】 是

序号	Python API	C++ API	边界
5	tf.concat	Concat	【参数】 ● values: 输入，包含tensor对象的列表或单个tensor，除要拼接的维度外，其他维度上的值要一致 ● axis: 0-D tensor，类型：int32，指定要拼接的维度，范围在[-rank(values), rank(values)]；python中，索引以0开始，axis为正值时，表示对第axis维拼接；axis为负值时，对第axis +rank(values)维拼接； 【约束】 ● 输入的tensor，除了进行concat的维度外，其他维度的size必须相等 输入的的tensor个数范围属于[1,1000] 【输出】 1个tensor，为输入tensors拼接后的结果 【量化工具支持】 是

序号	Python API	C++ API	边界
6	tf.matmul	MatMul	【参数】 ● a: 非常量tensor，rank>=2，类型：float32 ● b: 常量tensor，类型与rank同a ● transpose_a: 如果属性为True，a在乘法前做转置 ● transpose_b: 如果属性为True，b在乘法前做转置；transpose_a属性为False，transpose_b属性也为False ● adjoint_a: 如果属性为True，a在乘法前被共轭和转置 ● adjoint_b: 如果属性为True，b在乘法前被共轭和转置 ● a_is_sparse: 如果属性为True，a被看做是稀疏矩阵 ● b_is_sparse: 如果属性为True，b被看做是稀疏矩阵 ● name: 可选参数 【约束】 ● weight的转置属性为false ● 不能支持两个tensor相乘，只能支持1个tensor乘以1个常量 【输出】 1个tensor，类型同a与b 【量化工具支持】 否

序号	Python API	C++ API	边界
7	tf.nn.fused_batch_norm	FusedBatchNorm	【参数】 ● x: 输入，4-D tensor，类型：float32 ● scale: 1-D tensor，用于缩放 ● offset: 1-D tensor，偏差 ● mean: 1-D tensor，用于推理总体均值 ● variance: 1-D tensor，用于推理总体方差 ● epsilon: 在x 的方差中添加的1个小的浮点数 ● data_format: x的数据格式，值为'NHWC'（默认）或 'NCHW' ● is_training: bool值，用于指定操作是用于训练还是推断 ● name: 操作的名称，可选参数 【约束】 scale, bias,mean,var的shape只支持(1,C,1,1)，且c维度与input的c维度相等。 【输出】 ● y: 标准化、缩放、偏移x的4-D tensor ● batch_mean: 1-D tensor，表示x的均值 ● batch_var: 1-D tensor，表示x的方差 【量化工具支持】 否
8	tf.abs	Abs	【参数】 ● x: tensor或稀疏tensor，类型： float32 ● name: 操作的名称，可选参数 【约束】 无限制 【输出】 返回x的绝对值，tensor或稀疏tensor，尺寸与类型同x 【量化工具支持】 是

序号	Python API	C++ API	边界
9	tf.image.resize_nearest_neighbor	ResizeNearestNeighbor	【参数】 ● images: 4维tensor，[batch, height, width, channels]；3维tensor，[height, width, channels]；类型：float32 ● size: 1维tensor，常量，2个元素（新的高宽） ● method: ResizeMethod.NEARESTNEIGHBOR ● align_corners: bool值，默认为False；如果为True，输入和输出的4个角像素的中心对齐，保留角像素处的值 【约束】 无限制 【输出】 shape同输入，类型：float 【量化工具支持】 否
10	tf.image.resize_bilinear	ResizeBilinear	【参数】 ● images: 4维非常量tensor，[batch, height, width, channels]；类型：float32 ● size: 1维tensor，常量，2个元素（新的高宽） ● method: ResizeMethod.BILINEAR ● align_corners: bool值，默认为False；如果为True，输入和输出的4个角像素的中心对齐，保留角像素处的值 【约束】 （outputH*outputW）/（inputH*inputW） > 1/7 【输出】 shape同输入，类型：float 【量化工具支持】 否

序号	Python API	C++ API	边界
11	tf.cast	Cast	【输入】 ● 类型：float32, int32, bool, int64, int16, int8, uint8, uint16, double 【参数】 ● x: tensor或 sparseTensor或indexedSlices ● dtype: 目标类型，同x支持的数据类型 ● name: 名称（可选） 【约束】 无限制 【输出】 tensor或 sparseTensor或indexedSlices， 同输入的dtype、shape 【量化工具支持】 无需支持
12	tf.nn.depthwise_conv2d	DepthwiseConv2dNative	【参数】 ● input: 4维 ● filter: 4维，常量，数据格式：[filter_height, filter_width, in_channels, channel_multiplier] ● strides: 输入的每个维度的滑动窗口的步长，属性必须是list(int)，且大小为4 ● padding: 类型：string，值为’VALID’或’SAME’ ● rate：1维，大小为2；等值卷积中在height和width维度上对输入值进行采样的扩张率；如果值大于1，则步长的所有值必须为1 ● data_format: 输入的数据格式，可以是NHWC（默认）或NCHW ● name: 名称（可选） 【约束】 (W+15)/16*16)*filter.W*32 <= 32*1024，其中W是算子输入的W，filter.W是filter的W。 【输出】 4维tensor，形状与data_format一致，例如，对于NHWC格式，形状是[batch, out_height, out_width, in_channels * channel_multiplier] 【量化工具支持】 是

序号	Python API	C++ API	边界
13	tf.reshape	Reshape	【参数】 ● tensor: 输入，1个tensor ● shape: 定义输出的shape，常量tensor，数据类型：int64, int ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 1个tensor，类型同输入 【量化工具支持】 是
14	tf.squeeze	Squeeze	【参数】 ● input: 1个tensor，非常量输入 ● axis: 1个int型列表，指定要移除的维度，默认为[]；不能指定非1的维度； ● name: 此操作的名称（可选） ● squeeze_dims: 不推荐使用的参数，axis和dim不能同时存在 【约束】 无限制 【输出】 1个tensor，与input的类型、数据相同，但删除了1个或多个值为1的维度 【量化工具支持】 是
15	tf.expand_dims	ExpandDims	【参数】 ● inuput: 输入，1个tensor ● axis: 0-D（标量），指定扩展input形状的维度索引； ● name: 输出tensor的名称 ● dim: 0-D（标量），相当于axis，被弃用 【约束】 无限制 【输出】 1个tensor，与input数据相同，shape增加了一维（值为1） 【量化工具支持】 是

序号	Python API	C++ API	边界
16	tf.greater	Greater	【输入】 2 输入 ● 1个输入是常量tensor，数据格式要求：NC1HWC0（华为自研5维数据格式）、4维输入数据（例如：NCHW） 在NC1HWC0数据格式中，C0与微架构强相关，该值等于cube单元的size，例如16；C1是将C维度按照C0切分：C1=C/C0， 若结果不整除，最后一份数据需要padding到C0。 ● 另一个输入是常量标量 【参数】 name: 此操作的名称（可选） 【约束】 ● 支持广播broadcast，对比x和y的shape，在同一纬度上，dim[x]只能相同或者一方为1或者一方缺失。 ● tf.greater的下一层算子只能有2个输入，其中一个输入tf.greater算子的输出，另一个输入是常量。 【输出】 1个常量tensor，类型：bool 【量化工具支持】 无需支持
17	tf.nn.relu	Relu	【参数】 ● features: 非常量输入，tensor ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 1个tensor，类型同features 【量化工具支持】 是

序号	Python API	C++ API	边界
18	tf.nn.relu6	Relu6	【参数】 ● features: 非常量输入，tensor ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 1个tensor，类型同features 【量化工具支持】 是
19	tf.nn.leaky_relu	/	【参数】 ● features: 非常量输入，tensor，表示预激活的值 ● alpha: x<0 时激活函数的斜率 ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 激活值 【量化工具支持】 是
20	tf.exp	exp	【参数】 ● x: 1个输入tensor，类型float32, double； ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 1个tensor，类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
21	tf.nn.conv2d_transpose	Conv2DBackpropInput	【输入】 ● value: 输入，4-D tensor，数据格式: NHWC（[batch, height, width, in_channels]）或NCHW（[batch, in_channel, height, width]） ● filter: 4-D tensor，Const类型，shape: [height, width, output_channels, in_channels] ● input_sizes：1-D tensor，Const类型 【参数】 ● output_shape: 1-D tensor，表示输出的shape ● strides: int型列表，非空，输入的每个维度的滑动窗口的步长 ● padding: 类型：string，值为’VALID’或’SAME’，非空 ● data_format: 类型: string，’NHWC’或’NCHW’，非空 ● name: 输出名（可选） 【约束】 ● group = 1 ● dilation = 1 ● filterH - padHHead - 1 >= 0 ● filterW - padWHead - 1 >= 0 ● 还有一条约束涉及中间变量，公式如下： 1、a = ALIGN(filter_num,16) * ALIGN(filter_c, 16) * filter_h * filter_w * 2； 如果ALIGN(filter_c,16)%32 = 0，a = a/2； 2、conv_input_width=(反卷积输入w - 1) * strideW + 1； 3、b =（conv_input_width）* filter_h * ALIGN(filter_num,16) * 2 * 2； 4、a + b <= 1024*1024。 【输出】 1个tensor，类型同value 【量化工具支持】 是

序号	Python API	C++ API	边界
22	tf.sigmoid	Sigmoid	【参数】 ● x: 1个tensor，输入 ● name: 此操作的名称（可选） 【约束】 无限制 【输出】 1个tensor，类型同value 【量化工具支持】 是
23	tf.add	Add	【参数】 ● x: 输入，1个tensor，类型：float32、int32 ● y: 输入，1个tensor，类型同x；两个输入为常量时，1个输入为标量 ● name: 此操作的名称（可选） 【约束】 支持两组输入的维度不一致，进行广播操作（广播即维度补齐）， 目前支持以下几种广播场景： NHWC+ NHWC, NHWC+scalar 或者NHWC +1 1 1 1， 或者NHWC+W 和HWC+W 和HW+W(备注, W维度做广播)， 或者NCHW + NH1C 和HWC + H1C 和HW + H1， 还有HWC + 1 WC(备注，H维度做广播)； 说明：两个tensor的输入顺序可以互换。 【输出】 1个tensor，类型同y 【量化工具支持】 是
24	tf.add_n	-	tf.add_n算子内部调用Eltwise算子实现所有输入tensor按元素相加，关于输入、约束、属性等说明，请参考**5.12.2.2 Caffe算子边界**中Eltwise算子。

序号	Python API	C++ API	边界
25	tf.multiply	Multiply Type：Mul	【参数】 ● x: 输入，1个tensor，类型：float32、int32 ● y: 输入，1个tensor，类型同x；两个输入为常量时，维度必须一致，为标量或者1-D tensor ● name: 此操作的名称（可选） 【约束】 支持两组输入的维度不一致，进行广播操作（广播即维度补齐）， 目前支持以下几种广播场景： NHWC+ NHWC, NHWC+scalar 或者NHWC +1 1 1 1， 或者NHWC+W 和HWC+W 和HW+W(备注, W维度做广播)， 或者NCHW + NH1C 和HWC + H1C 和HW + H1， 还有HWC + 1 WC(备注，H维度做广播)； 说明：两个tensor的输入顺序可以互换。 【输出】 1个tensor 【量化工具支持】 是

序号	Python API	C++ API	边界
26	tf.subtract	Subtract Type：Sub	【参数】 ● x: 输入，1个tensor ● y: 输入，1个tensor，类型同x；两个输入，可支持常量或非常量 ● name: 此操作的名称（可选） 【约束】 支持两组输入的维度不一致，进行广播操作（广播即维度补齐）， 目前支持以下几种广播场景： NHWC+ NHWC, NHWC+scalar 或者NHWC +1 1 1 1， 或者NHWC+W 和HWC+W 和HW+W(备注, W维度做广播)， 或者NCHW + NH1C 和HWC + H1C 和HW + H1， 还有HWC + 1 WC(备注，H维度做广播)； 说明：两个tensor的输入顺序可以互换。 【输出】 1个tensor 【量化工具支持】 是
27	tf.nn.bias_add	BiasAdd	【参数】 ● value: 输入，1个tensor，非常量 ● bias: 1-D tensor，常量，尺寸与value的最后一维一致；除非value为量化类型，否则类型需同value ● data_format: 类型：string，’NHWC’或’NCHW’ ● name: 此操作的名称（可选） 【约束】 ● C < 10000； ● input和bias的数据排布要一致； ● 当在c通道上加bias时，input和bias的C维度大小要一致。 【输出】 1个tensor，类型同value 【量化工具支持】 是

序号	Python API	C++ API	边界
28	tf.nn.lrn	LRN	【参数】 ● input: 输入，4-D tensor，类型：float32 ● depth_radius: 0-D int型，默认值为5，1-D标准化窗口的半宽 ● bias: 可选参数，float型，默认为1；偏移（通常为正值，以避免除以0） ● alpha: 可选参数，float型，默认为1；比例因子，通常为正值 ● beta: 可选参数，float型，默认为0.5；1个指数 ● name: 此操作的名称（可选） 【约束】 ● depth_radius >0，且必须为奇数； ● 通道间：当depth_radius∈[1,15]时，alpha >0.00001且beta>0.01，否则alpha和beta为任意值；当C维度大于1776时，depth_radius <1728 【输出】 1个tensor，类型同input 【量化工具支持】 是
29	tf.nn.elu	Elu	【参数】 ● features: 输入tensor，非常量 ● name: 名称（可选） 【约束】 无限制 【输出】 输出tensor，类型同features 【量化工具支持】 否
30	tf.rsqrt	Rsqrt	【参数】 ● x: 输入tensor ● name: 名称（可选） 【约束】 无限制 【输出】 输出tensor，类型同x 【量化工具支持】 是

序号	Python API	C++ API	边界
31	tf.log	Log	【参数】 ● x: 输入tensor，类型：float32 ● name: 名称（可选） 【约束】 无限制 【输出】 输出tensor，类型同x 【量化工具支持】 否
32	tf.tanh	Tanh	【参数】 ● x: 输入tensor，非常量 ● name: 名称（可选） 【约束】 无限制 【输出】 输出tensor，类型同x 【量化工具支持】 是
33	tf.slice	Slice	【参数】 ● input_: 输入tensor ● begin: tensor，类型：int32或int64 ● size: tensor，类型：int32或int64 ● name: 名称（可选） 【约束】 输入tensor元素个数不超过INT32_MAX 【输出】 输出tensor，类型同input_ 【量化工具支持】 是

序号	Python API	C++ API	边界
34	tf.split	Split	【参数】 ● value: 输入 Tensor； ● num_or_size_splits：不支持这个参数 ● axis：标准，整型，指定输出维度 ● name: string;名称(可选) 【约束】 无限制 【输出】 List，包含split 后的各Tensor 【量化工具支持】 是
35	tf.nn.softplus	Softplus	【参数】 ● features: 输入 Tensor；数据类型：`float32` ● name: string;名称(可选) 【约束】 无限制 【输出】 输出Tensor，数据类型与features相同 【量化工具支持】 否
36	tf.nn.softsign	Softsign	【参数】 ● features: 输入 Tensor；数据类型：`float32` ● name: string;名称(可选) 【约束】 无限制 【输出】 输出Tensor，数据类型与features相同 【量化工具支持】 否

序号	Python API	C++ API	边界
37	tf.pad	Pad/ MirrorPad/ PadV2	【参数】 ● tensor: 4-D Tensor；数据类型：`float32`，`int32` ● paddings: Tensor, 常量，数据类型：`int32` ● mode: string , 值为 "CONSTANT",或 "REFLECT", 或"SYMMETRIC" ● name: string;名称(可选) ● constant_values：Pad默认填充的值 标量，数据类型与tensor相同 【约束】 CONSTANT模式时，0=<PAD<=128，0<W<=3000， 【输出】 输出Tensor，数据类型与tensor相同 【量化工具支持】 是
38	tf.fake_quant_with_min_max_vars	FakeQuantWithMinMaxVars	【参数】 ● inputs : 输入Tensor；数据类型：`float32`， ● min: Tensor, 数据类型：`float32` ● max: Tensor, 数据类型：`float32` ● num_bits：标量，整型，默认值`8` ● narrow_range：bool（可选），默认值`False` ● name: string;名称(可选) 【约束】 -65504<=min<=65504，-65504<=max<=65504。 【输出】 输出Tensor，数据类型：`float32` 【量化工具支持】 否

序号	Python API	C++ API	边界
39	tf.reduce_max	Max	【参数】 ● input_tensor : 输入tensor；数据类型：`float32`， `int64`,`int32`,`uint8`, `uint16`，`int16`，`int8` ● axis: 1-D list 或 标量，数据类型整型 ● keepdims: bool, 是否保留维度为1 ● name: string;名称(可选) ● reduction_indices: axis 旧的别名(不推荐) ● keep_dims：keepdims 别名（不推荐） 【约束】 ● 当输入的tensor维数等于4时：输入axis={3,{1,2,3}}，keepDims=true，H*W*16*2 <= 16*1024； ● 当输入的tensor维数等于2时，输入axis={1,{1}}，keepDims=true，H*W*CEIL(C,16)*16*2 <= 16*1024。 【输出】 输出reduce 后的Tensor，数据类型同input_tensor 【量化工具支持】 否

序号	Python API	C++ API	边界
40	tf.strided_slice	StridedSlice	【参数】 ● input_: 1个Tensor ● begin: 1-D Tensor，数据类型: `int32` ● end: 1-D Tensor，数据类型: `int32` ● strides: 1-D Tensor，数据类型: `int32`; ● begin_mask: 标量，数据类型: `int32` ● end_mask: 标量，数据类型: `int32` ● ellipsis_mask: 标量，数据类型: `int32` ● new_axis_mask:：标量，数据类型: `int32` ● shrink_axis_mask: 标量，数据类型: `int32` ● var：与 input_ 或 None 对应的变量 ● name: string;名称(可选) 【约束】 strides不允许为0 【输出】 输出Tensor，数据类型同input_ 【量化工具支持】 否
41	tf.reverse	Reverse	【参数】 ● tensor: 1个包含Tensor的列表， ● axis: Tensor reverse轴向，数据类型：`int32`,`int64` ● name: string;名称(可选) 【约束】 无限制 【输出】 1个Tensor，数据类型同tensor 【量化工具支持】 否

版权所有 © 华为技术有限公司

序号	Python API	C++ API	边界
42	tf.realdiv	RealDiv	【参数】 ● x: 输入Tensor；数据类型: float32` ● y: 输入Tensor；数据类型: float32` ● name: string;名称(可选) 【约束】 无限制 【输出】 输出Tensor，数据类型同x 【量化工具支持】 是
43	tf.stack	Stack	【参数】 ● values: 包含tensor对象（形状与类型相同，类型：float32，int32）的列表 ● axis：整数，沿axis维度堆叠，默认是第一维，必须设置 ● name: 名称（可选） 【约束】 无限制 【输出】 堆叠后的tensor，类型同values 属性T、N和axis必须存在 【量化工具支持】 否
44	tf.transpose	Transpose	【参数】 ● a: 输入tensor ● perm: a的维数的排列 ● name: 名称（可选） ● conjugate: 可选，类型: bool，默认为False；设置为True时，等同于tf.conj(tf.transpose(input)) 【约束】 无限制 【输出】 被转置后的tensor 【量化工具支持】 是

序号	Python API	C++ API	边界
45	tf.space_to_batch_nd	SpaceToBatchND	【参数】 ● input：1个Tensor，是N维的并且具有形状input_shape = [batch] + spatial_shape + remaining_shape,其中spatial_shape有M维度；支持数据类型为：uint8，int8，int16，uint16，int32，int64，float32 ● block_shape：1个Tensor，必须是以下类型之一：int32,int64；1-D,shape为[M],所有值必须>=1 ● paddings：1个Tensor，必须是以下类型之一：int32,int64；二维, shape为[M, 2],所有值必须>=0. 【约束】 ● 当tensor维数为4时：blockShape的长度必须等于2，paddings的长度必须等于4. ● blockShape元素的大小必须要大于等于1，paddings元素值的大小必须大于等于0. ● padding后的h维度要能够被blockShape[0]整除，padding后的w维度要能够被blockShape[1]整除。 【输出】 1个Tensor，与input具有相同的类型 【量化工具支持】 否

序号	Python API	C++ API	边界
46	tf.batch_to_space_nd	BatchToSpaceND	【参数】 ● input：1个Tensor，是N维的并且具有形状input_shape = [batch] + spatial_shape + remaining_shape,其中spatial_shape有M维度；支持数据类型为：uint8，int8，int16，uint16，int32，int64，float32 ● block_shape：1个Tensor；必须是以下类型之一：int32,int64；1-D,shape为[M],所有值必须>=1 ● crops：1个Tensor；必须是以下类型之一：int32,int64；二维, shape为[M, 2],所有值必须>=0. 【约束】 ● blockShape和crops的元素值数据类型必须是int32，当tensor维数为4时：blockShape的长度必须等于2，crops的长度必须等于4. ● blockShape元素的大小必须要大于等于1，crops元素值的大小必须大于等于0，crops数组的大小必须满足crop_start[i] + crop_end[i] < block_shape[i] * input_shape[i+1]； 【输出】 1个Tensor，与images具有相同的类型 【量化工具支持】 否

序号	Python API	C++ API	边界
47	tf.extract_image_patches	ExtractImagePatches	【参数】 ● images: 1个Tensor，支持数据类型：float32，int32，int64，uint8，int8，uint16，int16；4-D Tensor shape：[batch, in_rows, in_cols, depth] ● ksizes: 1个整形 list，长度>=4 ● strides: 1个整形list，必须是：[1, stride_rows, stride_cols, 1] ● rate: 1个整形list，必须是：[1, rate_rows, rate_cols, 1] ● padding: string，取值：“VALID”或者“SAME”，“VALID”表示所取的patch区域必须完全包含在原始图中."SAME"表示取超出原始图像的部分，以0填充该部分 ● name: 名称(可选) 【约束】 无约束 【输出】 1个Tensor，与images具有相同的类型 【量化工具支持】 否
48	tf.floormod	FloorMod	【参数】 ● x: 1个Tensor，支持数据类型：float32，int32 ● y: 1个Tensor，与x具有相同类型 ● name: 名称(可选) 【约束】 由于支持广播broadcast，对比x和y的shape，在同一纬度上，dim[x]只能相同或者一方为1或者一方缺失。 【输出】 1个Tensor，与x具有相同的类型 【量化工具支持】 否

序号	Python API	C++ API	边界
49	tf.nn.softmax	Softmax	【参数】 ● logits: 1个非空的Tensor，支持数据类型：float32 ● axis:在其上执行维度softmax。默认值为-1，表示最后1个维度，不超过logits维度 ● name: 名称(可选) ● dim: axis的已弃用的别名 【约束】 ● 输入4维时可以针对每一维做softmax： 根据分类的轴不同，计算边界分别为： axis=1时，C<=（(256*1024/4)-8*1024-256）/2； axis=0时，N<=(56*1024-256)/2； axis=2时，W=1, 0<H<（1024*1024/32）； axis=3时，0<W<（1024*1024/32） ● 输入维度不足4维时，仅支持对最后一维做softmax计算， 并且最后一维不超过46080。 【输出】 1个Tensor，与logits具有相同的类型和shape 【量化工具支持】 是
50	tf.math.pow	Power	【参数】 ● x: 1个Tensor，支持数据类型：float32 ● y: 1个Tensor，支持数据类型：float32 ● name: 名称(可选) 【约束】 power!=1 scale*x+shift>0。 【输出】 1个 Tensor 【量化工具支持】 是

序号	Python API	C++ API	边界
51	tf.nn.leaky_relu	LeakyRelu	【参数】 ● features: 1个Tensor，支持数据类型： float32 ● alpha: x <0时激活函数的斜率. ● name: 名称(可选) 【约束】 无限制 【输出】 激活值 【量化工具支持】 是
52	tf.placeholder	-	【参数】 ● dype: 数据类型,(必须) ● shape: Tensor的维度和大小(必须) 【约束】 无限制 【输出】 1个Tensor 【量化工具支持】 否
53	tf.shape	Shape	【参数】 ● input: 1个Tensor 或 `SparseTensor` ● name: string;名称(可选) ● out_type 输出Tensor数据类型, `int32` 或 `int64` (可选,默 认为`int32`) 【约束】 无限制 【输出】 1个Tensor，输出数据类型为out_type 【量化工具支持】 无需支持

序号	Python API	C++ API	边界
54	tf.math.argmax	ArgMax	【参数】 ● input: 1个Tensor ,支持数据类型:`int8`,`uint8`,`int16`, `uint16`, `int32`,`int64`,`float32` ● axis: 1个Tensor, 数据类型`int32`,`int64` ● out_type 输出Tensor数据类型, `int32` 或 `int64` (可选,默 认为`int64`) ● name: string;名称(可选) 【约束】 无限制 【输出】 1个Tensor，输出数据类型为out_type 【量化工具支持】 否
55	tf.gather	Gather GatherV2	【参数】 ● params: 1个Tensor ，维度必须大于 `axis + 1` ● indices: 1个Tensor，数据类型`int32`,`int64`，范围[0, params.shape[axis]) ● axis: 输出Tensor数据类型， `int32` 或 `int64`，指定indices 选取的维度，rank=0 ● name: string，名称(可选) 【约束】 无限制 【输出】 1个Tensor,输出数据类型于params 相同 【量化工具支持】 否

序号	Python API	C++ API	边界
56	tf.gather_nd	GatherNd	【参数】 ● params: 1个Tensor，维度必须大于 `axis + 1` ● indices: 1个Tensor，数据类型`int32`,`int64` ● name: string;名称(可选) 【约束】 indices最后一维的大小不能超过params的维数 indices最后一维中的元素对应着params中的1个维度上的坐标，必须满足坐标规则 indices中对应维度上的坐标不能超过维度的大小 【输出】 1个Tensor，输出数据类型于params 相同 【量化工具支持】 是
57	tf.math.floordiv	FloorDiv	【参数】 ● x: 1个Tensor ,数据类型：`float32`,`int32`, ● y: 1个Tensor, 分母，数据类型：` float32`,` int32` ● name: string;名称(可选) 【约束】 由于支持广播broadcast，对比x和y的shape，在同一纬度上，dim[x]只能相同或者一方为1或者一方缺失 【输出】 1个Tensor ，floor（x/y） 【量化工具支持】 否

序号	Python API	C++ API	边界
58	tf.range	Range	【参数】 ● start: 开始标量，数据类型`float32`,`int32`, 必须为常量 ● limit: 结束标量，数据类型`float32`,`int32`, 必须为常量 ● delta: 步长标量，数据类型`float32`,`int32`, 必须为常量 ● dtype： 返回Tensor的数据类型 ● name: string;名称(可选) 【约束】 无限制 【输出】 1个1-D Tensor 【量化工具支持】 否
59	tf.tile	Tile	【参数】 ● input: 输入Tensor，数据类型： `int8`,`uint8`,`int16`, `uint16`, `int32`,`int64`,`float32` ● multiples: 1-D Tensor，长度须和input的秩相同，数据类型：`int32`, 必须为常量 ● name: string;名称(可选) 【约束】 无限制 【输出】 1个Tensor 【量化工具支持】 是
60	tf.size	Size	【参数】 ● input: 输入Tensor，数据类型：float32` ● name: string;名称(可选) ● out_type: 输出Tensor 数据类型，默认`int32` 【约束】 无限制 【输出】 1个Tensor，类型由out_type指定 【量化工具支持】 否

序号	Python API	C++ API	边界
61	tf.fill	Fill	【参数】 ● dims: 1-D Tensor, 数据类型：`int32`, ● value: 变量，数据类型：`int32`,`float32` ● name: string;名称(可选) 【约束】 支持Constant、GivenTensor、Range、Diagonal、Gaussian、MSRA、Uniform、UniformInt、UniqueUniform、XavierFill这些填充模式，在Uniform填充、UniformInt填充、UniqueUniform填充、xavier填充时，生成的数值区间最大范围介于[min,max)之间 【输出】 1个Tensor，类型同value数据类型 【量化工具支持】 否
62	tf.concat	Concat	【参数】 ● value: 1个包含Tensor的列表，数据类型：`int32`,`float32` ● axis: Tensor 拼接轴向，数据类型：`int32` ● name: string;名称(可选) 【约束】 ● 输入的tensor，除了进行concat的维度外，其他维度的size必须相等 输入的的tensor个数范围属于[1,1000] 【输出】 1个Tensor 【量化工具支持】 是
63	tf.reverse	Reverse	【参数】 ● tensor: 1个包含Tensor的列表， ● axis: Tensor reverse轴向，数据类型：`int32` ● name: string;名称(可选) 【约束】 无限制 【输出】 1个Tensor，数据类型同tensor 【量化工具支持】 否

序号	Python API	C++ API	边界
64	tf.reduce_sum	sum	【参数】 ● input_tensor: 输入Tensor， ● axis: Tensor sum轴向，数据类型：`int32` ● keepdims：bool值，是否保留维度 ● name: string;名称(可选) ● reduction_indices：axis 旧的string;名称 ● keep_dims：不推荐使用，参数keepdims 别名 【约束】 无约束 【输出】 输出Tensor，数据类型同tensor 【量化工具支持】 否
65	tf.math.maximum	Maximum	【参数】 ● x: Tensor 数据类型：`int32`，`int64`，`float32` ● y: Tensor 数据类型同x ● name: string;名称(可选) 【约束】 由于支持广播broadcast，对比x和y的shape，在同一纬度上，dim[x]只能相同或者一方为1或者一方缺失 【输出】 输出Tensor，返回值（x > y ? x : y） 数据类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
66	tf.math.minimum	Minimum	【参数】 ● x: Tensor 数据类型：`int32`，`int64`，`float32` ● y: Tensor 数据类型同x ● name: 名称（可选） 【约束】 广播场景只支持下面两种： NHWC+scaler,NHWC+NHWC 【输出】 输出Tensor，返回值（x < y ? x : y） 数据类型同x 【量化工具支持】 否
67	tf.clip_by_value	ClipByValue	【参数】 ● t: Tensor ● clip_value_min: clip 最小值 ● clip_value_max：clip 最大值 ● name: string;名称(可选) 【约束】 min值要小于或者等于max值 【输出】 输出Tensor，返回值范围【clip_value_min。clip_value_max】 【量化工具支持】 否
68	tf.math.logical_not	LogicalNot	【参数】 ● x: Tensor 数据类型bool ● name: string;名称(可选) 【约束】 无限制 【输出】 输出Tensor，数据类型bool 【量化工具支持】 否

序号	Python API	C++ API	边界
69	tf.math.logical_and	LogicalAnd	【参数】 ● x: Tensor 数据类型bool ● y: Tensor 数据类型bool ● name: string;名称(可选) 【约束】 BroadCast只支持如下几种维度的广播，NHWC和 [1,1,1,1],[N,H,W,C],[N,H,W,1],[1,H,W,C],[N,1,1,C] 【输出】 输出Tensor，数据类型bool 【量化工具支持】 否
70	tf.equal	Equal	【参数】 ● x: Tensor ● y: Tensor，数据类型同x ● name: string，名称(可选) 【约束】 由于支持广播broadcast，对比x和y的shape，在同一纬度上右对齐的情况下，xdim[i]和ydim[i]只能相同或者一方为1或者一方缺失 【输出】 输出Tensor，数据类型bool 【量化工具支持】 否
71	tf.square	Square	【参数】 ● x: Tensor ● name: string，名称(可选) 【约束】 无限制 【输出】 输出Tensor，数据类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
72	tf.image.crop_and_resize	CropAndResize	【参数】 ● image: 4-D Tensor; 数据类型： `float32`，`int8`, `int32`,`int64`, shape: `[num_boxes, 4]` ● boxes： 2-D Tensor;，数据类型: `float32`，shape：`[num_boxes]` ● box_ind：1-D Tensor，数据类型：`int32` ● crop_size：1-D Tensor ，包含2个元素，数据类型： `int32`， ● method: string,表示插值方法, 值为 "bilinear"(默认), "nearest" ： ● name: string;名称(可选) 【约束】 无限制 【输出】 ● Tensor，数据类型: `float32` 【量化工具支持】 否
73	tf.math.top_k	TopKV2	【参数】 ● input: Tensor，>=1-D,最后1个维度大小必须大于k; ● 数据类型： `float32`k： 标量，>=1; 数据类型：`int32` ● sorted：bool ● name: string;名称(可选) 【约束】 K一定要以常量传入 【输出】 ● values: Tensor 返回最后维度上的K个最大向量 ● indices: Tensor，values 在input中的索引位置 【量化工具支持】 否

序号	Python API	C++ API	边界
74	tf.invert_permutation	InvertPermutation	【参数】 ● x: 1-D Tensor，数据类型： `int32`,`int64`, ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，数据类型同x 【量化工具支持】 否
75	tf.multinomial	Multinomial	【参数】 ● logits ： 2-D Tensor，shape `[batch_size, num_classes]` ● num_samples： 标量 抽样个数 ● seed: 随机数种子，数据类型： `int32`,`int64`, ● name: string;名称(可选) ● output_dtype：输出Tensor 数据类型：整型默认`int64` 【约束】 seed为0时产生随机数是动态的 输出数据行数等于输出数据的行，输出数据的列数等num_samples 【输出】 ● Tensor，数据类型同output_dtype 【量化工具支持】 否

序号	Python API	C++ API	边界
76	tf.reverse_sequence	ReverseSequence	【参数】 ● input： 输入Tensor ● seq_lengths： 1-D Tensor； 数据类型：`int32`,`int64` ● seq_axis: 标量，,数据类型：整型 ● batch_axis: 标量（可选），数据类型：整型 ● name: string;名称(可选) 【约束】 ● seq_lengths的长度必须等于input在batchAxis的元素数。 ● seq_lengths的最大元素必须要小于等于seq_dim的元素数。 ● seqAxis、batchAxis、seqDim、batchDim必须是int64类型 ● seqAxis与seqDim不可同时指定，batchAxis与batchDim不可同时指定 ● batchAxis和batchDim是可选参数，缺省值为0权值个数需要为1， 【输出】 Tensor，数据类型同input 【量化工具支持】 否
77	tf.math.reciprocal	Reciprocal	【参数】 ● x： 输入Tensor 数据类型`float32` ● name: string;名称(可选) 【约束】 不支持输入数据中包含0 【输出】 Tensor，数据类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
78	tf.nn.selu	Selu	【参数】 ● features： 输入Tensor 数据类型`float32`, ● name: string;名称(可选) 【约束】 无约束 【输出】 Tensor，数据类型同features 【量化工具支持】 否
79	tf.math.acosh	Acosh	【参数】 ● x： 输入Tensor 数据类型`float32` ● name: string;名称(可选) 【约束】 无约束 【输出】 Tensor，数据类型同x 【量化工具支持】 否
80	tf.math.asinh	Asinh	【参数】 ● x： 输入Tensor 数据类型`float32` ● name: string;名称(可选) 【约束】 无约束 【输出】 Tensor，数据类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
81	tf.math.reduce_prod	Prod	【参数】 ● input_tensor： 输入Tensor ● axis : reduce 维度 ● keepdims： bool 是否需要保存reduce维度 ● name: string;名称(可选) ● reduction_indices： 参数axis 旧的别名（不推荐） ● keep_dims：参数keepdims 别名（不推荐） 【约束】 无约束 【输出】 Tensor，raduce 后的Tensor 【量化工具支持】 否
82	tf.math.sqrt	Sqrt	【参数】 ● x： 输入Tensor；数据类型: `float32` ● name: string;名称(可选) 【约束】 无约束 【输出】 Tensor，数据类型同x 【量化工具支持】 否
83	tf.math.reduce_all	All	【参数】 ● input_tensor： 输入Tensor；数据类型: bool ● axis： 指定reduce的维度 ● keepdims：bool值 ● name: string;名称(可选) ● reduction_indices：axis 旧的别名(不推荐) ● keep_dims：keepdims 别名（不推荐） 【约束】 无约束 【输出】 Tensor，数据类型同input_tensor 【量化工具支持】 否

序号	Python API	C++ API	边界
84	tf.nn.l2_normalize	L2Normalize	【参数】 ● x： 输入Tensor；数据类型: boolean ● axis： 指定normalize的维度轴向； 如果format为NCHW，则axis必须为1； 如果format为NHWC，则axis必须为3 ● epsilon：规范化的下限值，取值范围(1e-7, 0.1]。如果norm <sqrt(epsilon),将使用 sqrt(epsilon)作为除数. ● name: string;名称(可选) ● dim：axis 旧的别名(不推荐) 【约束】 H*W*2 < 256*1024/4； 【输出】 Tensor，数据类型同x 【量化工具支持】 否
85	tf.keras.backend.hard_sigmoid	Hardsigmoid	【参数】 ● x： 输入Tensor 【约束】 无限制 【输出】 输出Tensor，返回值：`0.` if `x < -2.5`, `1.` if `x > 2.5`. 当 `-2.5 <= x <= 2.5`, 返回 `0.2 * x + 0.5`. 【量化工具支持】 否
86	tf.keras.layers.ThresholdedReLU	ThresholdedReLU	【参数】 theta： 标量>=0,数据类型:；`float32` 【约束】 无限制 【输出】 Tensor 【量化工具支持】 否

序号	Python API	C++ API	边界
87	tf.math.acos	Acos	【参数】 ● x： 输入Tensor 数据类型 `float32`,`int32`,`int64` ● name: string;名称(可选) 【约束】 输入数据范围（-1<=x<=1），输出数据范围（0<=y<=π） 【输出】 Tensor，数据类型同x 【量化工具支持】 否
88	tf.math.atan	Arctan	【参数】 ● x： 输入Tensor 数据类型 `float32`,`int32`,`int64` ● name: string;名称(可选) 【约束】 输入数据范围（-65504<=x<=65504），输出数据范围（-π/2<y<π/2）。 【输出】 Tensor，数据类型同x 【量化工具支持】 否
89	tf.math.asin	Asin	【参数】 ● x： 输入Tensor 数据类型 `float32`,`int32`,`int64` ● name: string;名称(可选) 【约束】 输入数据范围（-1<=x<=1），输出数据范围（-π/2<=y<=π/2）。 【输出】 Tensor，数据类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
90	tf.math.atanh	Atanh	【参数】 ● x： 输入Tensor 数据类型 `float32`,`int32`,`int64` ● name: string;名称(可选) 【约束】 输入数据范围：x∈(-1,1) 【输出】 Tensor，数据类型同x 【量化工具支持】 否
91	tf.math.tan	Tan	【参数】 ● x： 输入Tensor 数据类型 `float32`,`int32`,`int64` ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，数据类型同x 【量化工具支持】 否
92	tf.math.logical_or	LogicalOr	【参数】 ● x： 输入Tensor 数据类型:`bool` ● y： 输入Tensor 数据类型:`bool` ● name: string;名称(可选) 【约束】 由于支持广播broadcast，对比x和y的shape，在同一纬度上右对齐的情况下，xdim[i]和ydim[i]只能相同或者一方为1或者一方缺失。 【输出】 Tensor，数据类型: `bool` 【量化工具支持】 否

序号	Python API	C++ API	边界
93	tf.math.reduce_min	ReduceMin	【参数】 ● input_tensor： 输入Tensor 数据类型:`float32`,`int64`,`int32`,`uint8`,`uint16`,`int8`,`int16`, ● axis: reduce 的维度轴向 ● keepdims：标量，bool类型， ● name: string;名称(可选) ● reduction_indices：axis 旧的别名 ● keep_dims：keepdims别名（不推荐） 【约束】 ● 当输入的tensor维数等于4时：输入axis={3,{1,2,3}}，keepDims=true，H*W*16*2 <= 16*1024； ● 当输入的tensor维数等于2时，输入axis={1,{1}}，keepDims=true，H*W*CEIL(C,16)*16*2 <= 16*1024 【输出】 Tensor，数据类型同input_tensor 【量化工具支持】 否
94	tf.math.negative	Neg	【参数】 ● x： 输入Tensor 数据类型:`float32`,`int64`,`int32`, ● name: string;名称(可选) 【约束】 输入数据范围（-65504<=x<=65504），输出数据范围（-65504<=y<=65504） 【输出】 Tensor，输出= -x 【量化工具支持】 否

序号	Python API	C++ API	边界
95	tf.math.greater_equal	Greaterequal	【参数】 ● x： 输入Tensor 数据类型: `float32`, `int64`,`int32`,`uint8`,`uint16`,`int8`,`int16`, ● y： 输入Tensor 数据类型同x, ● name: string;名称(可选) 【约束】 输入数据范围（-65504<=x<=65504） 【输出】 Tensor，输出类型bool 【量化工具支持】 否
96	tf.space_to_depth	SpaceToDepth	【参数】 ● input： 输入Tensor 数据类型: `float32`,`int64`,`int32`,`uint8`,`int8`, ● block_size：标量，整型，值>=2 ● data_format：数据类型：string 值: `"NHWC", "NCHW", "NCHW_VECT_C"`. 默认值是 `"NHWC" ● name: string;名称(可选) 【约束】 blockSize的大小必须大于等于1，且能被H和W整除 【输出】 Tensor，输出类型input 【量化工具支持】 否

序号	Python API	C++ API	边界
97	tf.depth_to_space	DepthToSpace	【参数】 ● input： 输入Tensor 数据类型: `float32`, `int64`,`int32`,`uint8`,`int8`, ● block_size： 标量，整型，值>=2 ● data_format：数据类型：string 值: `"NHWC", "NCHW", "NCHW_VECT_C"`. 默认值是 `"NHWC" ● name: string;名称(可选) 【约束】 blockSize必须大于等于1，且blockSize* blockSize必须能被C整除 【输出】 Tensor，输出类型input 【量化工具支持】 否
98	tf.math.round	Round	【参数】 ● x： 输入Tensor；数据类型: `float32`,`int64`,`int32` ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，输出类型同x，输出shape 同x 【量化工具支持】 否
99	tf.math.rint	Rint	【参数】 ● x： 输入Tensor；数据类型: `float32`,`int64`,`int32`, `uint8`,`int8` ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，输出类型同x，输出shape 同x 【量化工具支持】 否

序号	Python API	C++ API	边界
100	tf.math.less	Less	【参数】 ● x： 输入Tensor 数据类型: `float32`, `int64`,`int32`,`uint8`,`uint16`,`int8`,`int16`, ● y： 输入Tensor 数据类型同x, ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，输出类型bool 【量化工具支持】 否
101	tf.math.sinh	Sinh	【参数】 ● x： 输入Tensor 数据类型: `float32` ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，输出类型同x 【量化工具支持】 否
102	tf.math.cosh	Cosh	【参数】 ● x： 输入Tensor 数据类型: `float32` ● name: string;名称(可选) 【约束】 无限制 【输出】 Tensor，输出类型同x 【量化工具支持】 否

序号	Python API	C++ API	边界
103	tf.math.squared_difference	Squared_difference	【参数】 ● x： 输入Tensor; 数据类型: `float32`,`int64`,`int32` , ● y： 输入Tensor 数据类型同x, ● name: string;名称(可选) 【约束】 广播模式只支持下列场景： 第1个的tensor_format是NCHW，另1个的dim{}可以是[1,1,1,1]，[N,C,H,W]，[N,1,H,W]，[1,C,H,W]，[N,C,1,1],[1,C,1,1],[1,1,H,W],[N,1,1,1]这几种情况。 【输出】 Tensor，数据类型同x 【量化工具支持】 否

6 应用软件开发

6.1 开发前必读
6.2 软件开发说明（重要）
6.3 开发前准备
6.4 开发环境介绍
6.5 通用推理业务流程介绍
6.6 代码运行样例（HelloDavinci程序）
6.7 软件代码开发
6.8 Graph关键字

6.1 开发前必读

本章主要介绍在使用Atlas 200进行业务开发时要了解的基础知识、要求和注意事项。

建议开发人员仔细阅读本章节内容，确保了解各项要求和注意事项之后再启动开发。

6.1.1 适用场景

Atlas 200 模组根据不同的应用场景分为以下三种应用模式：

- 主处理器
- 协处理器
- PCIe从设备

说明

- 三种应用模式的区分请参考《Atlas 200 AI加速模块 1.0.0.SPC200以下 系统软件开发指南（型号 3000）》中的“应用模式”章节。
- 主处理器和协处理器的应用软件开发流程基本一致，本文档统称为SOC模式，PCIe从设备场景统称为EP模式。

6.1.2 关键概念

概念	解释
Ascend 310	Ascend 310是一款华为专门为图像识别、视频处理、推理计算及机器学习等领域设计的高性能、低功耗AI芯片。 芯片内置2个AI core，可支持128位宽的LPDDR4x，最高可提供16TOPS（Float16/INT8）的计算能力。
Atlas 200	Atlas 200是基于Ascend 310处理器的一个高性能的AI加速模块，可以实现高效的模型推理、图像识别及处理等工作。具有体积小、性能高、能效低，并且集成简单等，根据工作的场景可工作做SOC场景、协处理器场景、PCIe从设备的场景。
Mind Studio	Mind Studio是一套基于华为NPU（Neural-network Processing Unit）开发的AI全栈开发平台，包括基于芯片的算子开发、调试、调优以及自定义算子开发，同时还包括网络层的网络移植、优化和分析，另外在业务引擎层提供了一套可视化的AI引擎拖拽式编程服务。如果想进一步了解Mind Studio，请参考文档《Ascend 310 Mind Studio 快速入门》和《Ascend 310 Mind Studio 基础操作》。
DDK	数字开发套件（Digital Development Kit），DDK是Mind Studio解决方案提供的开发者套件包，Mind Studio通过安装DDK后获得Mind开发必需的API、库、工具链等开发组件。
Graph	Graph是HiAI框架中的概念，而非深度学习框架中计算图的概念。Graph是指HiAI框架中用于描述整个业务处理流程的图，是一个程序处理流程，由多个Engine组成。
HiAI Engine	HiAIEngine是一个通用业务流程执行引擎，主要包含Agent（运行在Host侧）和Manger（运行在Device侧）两个部分。每个Engine完成一个由用户代码实现的功能，即Engine的处理程序是由用户实现的。
Host侧	对于服务器+Atlas 200来说，Host侧为服务器侧CPU的操作系统。
Device侧	对于服务器+Atlas 200来说，Device侧为Atlas 200侧的操作系统。
DVPP	数字视觉预处理（Digital Vision Pre-Process），提供对特定格式的视频和图像进行解码、缩放等预处理操作，同时具有对处理后的视频、图像进行编码再输出的能力。
AIPP	AI（AI Pre Process）预处理，支持格式转换、Padding/Crop、CSC色域转换（YUV2RGB或者RGB2YUV）、Scale UP/Down、通道数据交换等。
OMG	离线模型生成（Offline Model Generator），用户使用Caffe/TensorFlow等框架训练好的模型，通过OMG将其转换为华为芯片支持的离线模型，实现算子调度的优化，权值数据重排、压缩，内存使用优化等可以脱离设备完成的模型优化功能。
OME	离线模型执行（Offline Model Inference Executor），转换完成的离线模型，使用OME进行模型的加载和推理。
Ctrl CPU	一个Ascend 310芯片中有4个Ctrl CPU，主要负责业务逻辑处理。

概念	解释
AI CPU	一个Ascend 310芯片中有4个AI CPU，主要用于算子任务调度、部分算子的实现。
AI Core	一个Ascend 310芯片中有2个AI Core，主要负责矩阵运算。
IPC	IP摄像机，提供RTSP数据流。

6.1.3 业务开发约束

相关约束请参考**表6-1**。

表 6-1 业务开发约束条例

编号	类型	约束描述	重要级别
1	兼容性	Atlas 200的文件系统目前只支持Ubuntu 16.04.3 Arm64。	高
2		当前交付版本支持g++5.4，低于该版本的编译链建议重新源码编译。	高
3	推理	Atlas 200为NPU，适合深度学习计算。	高
4		Atlas 200目前仅支持AI推理，不支持训练。推理平台支持Caffe以及Tensorflow深度学习框架。	高
5		支持大部分视频、图片推理领域的网络和算子，其它领域类（如语音、NLP等）的算子暂不支持，具体参考算子支持清单。	高
6		推理不支持动态batchsize，需要客户根据预定的batchsize推理，推荐8 batch。不支持动态batchsize场景，需要约束用户补齐batchsize数据。 例如：模型转换时batchsize为8，若用户使用时只有4个batch数据，则需要补充4个batch数据，拼成8个batch数据再进行推理。	高
7	编解码	VDEC视频解码，单芯片最多支持创建16路解码引擎。	高
8		JPEG图片解码，只支持huffman编码格式，不支持算术编码格式、渐进编码和jpeg2000格式，暂不支持YUV440采样格式。	高
9		PNG仅支持32bit位深rgba格式。	高
10		DVPP VDEC解码输出HFBC格式（华为内部格式）不能直接使用，需要通过VPC转换后使用。 VPC输出只支持YUV420SP格式，不支持YUV转RGB。	高

编号	类型	约束描述	重要级别
11		VPC输入输出图像内存限制：图像数据起始地址128字节对齐，图像宽128字节对齐，高16字节对齐，对齐后冗余的图像数据可通过AIPP裁剪掉。	高
12		VPC抠图最小为16*16像素。	低
13		AIPP必须以YUV420SP_U8格式输入，即NV12。	中
14		DVPP的JPEGD支持NV21格式，推理前必须通过VPC转换成NV12。	中
15		DVPP硬件限制，必须使用4G空间内地址，建议DVPP创建放在模型管家初始化之前，使DVPP在初始化前预留在对应的4G空间内。	高
16		VPC缩放最大放大16倍数，超过16倍放大需要软件多次放大实现。	高
17		视频解码最大480帧/s@1080P，单Graph最大320/s@1080P，320帧以上需要采用多进程方式。	高

6.2 软件开发说明（重要）

EP 模式下软件开发说明

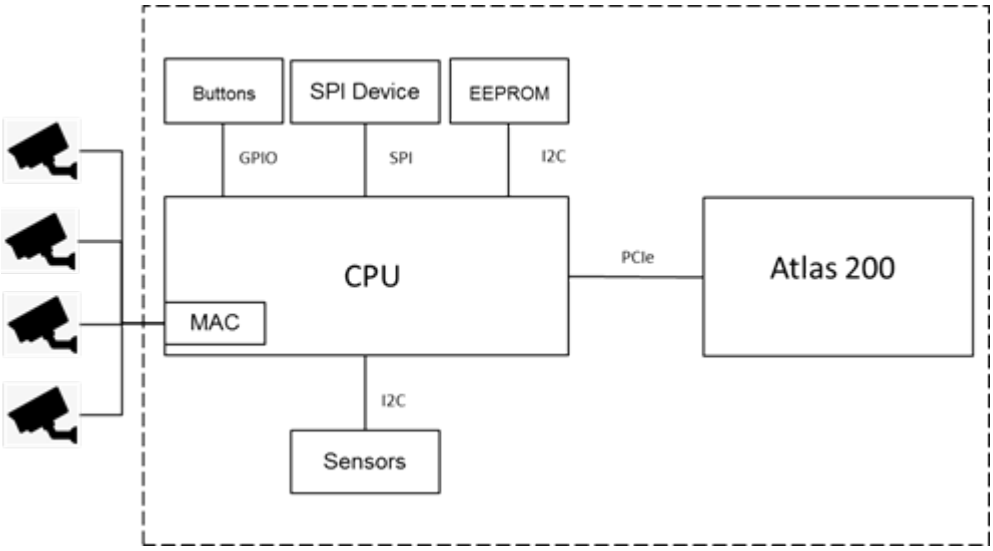
Atlas 200 EP模式是指Atlas 200作为PCIe从设备接入CPU系统工作的模式，整体示意图如图6-1所示。在EP模式下，业务软件运行在Host侧，通过PCIe通道与Atlas 200 进行交互，将AI任务卸载到Ascend 310芯片中运行。

在这种应用模式下，需要注意CPU选型，考虑CPU与Atlas 200 兼容。

Atlas 200的兼容性要求如下:

1. 支持PCIe MSI-X中断，至少可分配116个MSI-X中断。
2. BAR空间大小要求（3个BAR，大小分别为128KB，16MB，64MB）。

图 6-1 Atlas200 EP 模式示意图



与SOC模式不同，在EP模式下，CPU作为Host侧，Atlas 200作为Device侧，两者通过PCIe进行数据通信，因此在EP模式下，Host侧与Device侧将会发生数据的跨侧传输，所谓的跨侧传输为：传输的数据需要序列化为二进制，通过PCIE或DMA等硬件完成数据传输后，反序列为有效数据。因此对于自定义数据结构，需要用户自定义序列化与反序列化函数，Matrix框架主要提供两种序列化/反序列函数定义方式，分别为普通接口和高速接口，普通接口适合256K以下的数据传输，高速接口适合256K以上的数据传输（高速接口传输小内存块，速度与普通接口类似），详情请参考《Matrix API参考》中的“Graph::SetDataRecvFunctor”和“Engine::SetDataRecvFunctor”章节。

除此之外，由于部分代码运行于Host侧，部分代码运行于Device侧，因此代码需要分开进行编译，用户需要根据自己选取的CPU型号以及OS类型，使用合适的编译器进行代码编译，以便正确编译运行业务。

EP模式下的业务开发与Atlas 500十分相似，可以认为Atlas 500就是Atlas 200运行在EP模式下的例子。在Atlas 500中，Hi3559A作为主处理器，即**图6-1**中的CPU，Atlas 200作为PCIe从设备，由Hi3559A CPU对其进行控制，完成业务流。

因此，若用户在EP模式下进行业务软件开发，Atlas 200硬件底板设计参考《Atlas 200 AI加速模块 硬件开发指南（型号 3000）》，搭建开发环境与执行环境请参考《Atlas 200 AI加速模块 1.0.0.SPC200及以上 系统软件开发指南（型号 3000）》。业务软件开发可以参考Atlas 500的业务软件开发文档《Atlas 500 应用软件开发指南》。

SOC 模式下软件开发说明

Atlas 200 SOC模式是指Atlas 200作为主/协处理器的模式。如果Atlas 200作为主/协处理器，主处理器和协处理器的应用软件开发流程基本一致，业务软件开发的方法可以参见本文档的软件代码开发内容。

6.3 开发前准备

6.3.1 获取样例程序

- Sample代码

获取地址：**https://gitee.com/HuaweiAtlas/samples**
使用方法：请参考gitee各Sample README.md文件。

- Demo代码
  获取地址：**https://gitee.com/HuaweiAtlas**
  使用方法：请参考gitee各Demo README.md文件。

6.4 开发环境介绍

本章主要介绍安装后的开发环境，包含头文件、动态库、编译工具和其他工具。

6.4.1 DDK 目录分布

环境变量设置

安装完DDK，需要将DDK安装路径导出到环境变量DDK_HOME下，代码实现如下：

```
export DDK_HOME=/home/HwHiAiUser/tools/che/ddk/ddk
```

以上目录即为DDK安装目录，此处仅为示例，实际以现场为准。

目录结构

在$DDK_HOME中的目录结构以及主要目录介绍如下：

- include：头文件目录
- host/lib：Host侧库目录
- device/lib：Device侧库目录

```
├── bin
├── conf
├── ddk_info
├── device
│   └── lib -> ../lib/aarch64-linux-gcc5.4
├── host
│   └── lib -> ../lib/aarch64-linux-gcc5.4
├── include
│   ├── inc
│   ├── libc_sec
│   └── third_party
├── lib
│   ├── aarch64-linux-gcc5.4
│   └── x86_64-linux-gcc5.4
├── packages
├── python3.5
├── sample
├── scripts
├── site-packages
├── toolchains
└── uihost
    ├── bin -> ../bin/x86_64-linux-gcc5.4
    ├── lib -> ../lib/x86_64-linux-gcc5.4
    └── toolchains -> ../toolchains
```

6.4.2 头文件与库介绍

头文件

include中包含如下三个头文件：

```
├─── include
│   ├─── inc
│   ├─── libc_sec
│   └─── third_party
```

- include/inc：包含DDK主要功能头文件
- include/libc_sec：包含安全函数头文件
- include/third_party：包含第三方库头文件

动态库

由于Atlas 200没有主从之分，程序统一使用aarch64-linux所属的编译器进行交叉编译。因此Atlas 200的DDK中的“host/lib”和“device/lib”目录都指向了同一个文件夹“lib/aarch64-linux-gcc5.4”，在这个文件夹中分别存放了交叉编译需要的库文件，其中也包含了protobuf，opencv等第三方库文件。

6.4.3 编译工具介绍

对于Atlas 200，程序需要采用aarch64的gcc编译链工具进行交叉编译。交叉编译工具aarch64-linux-gcc5.4的安装方式参考《Atlas 200 AI加速模块 1.0.0.SPC200及以上 系统软件开发指南（型号 3000）》的"安装工具链"章节。（推荐的交叉编译工具的安装路径为“/opt/atlas200_cimpiler/bin”）

```
├─── /opt/atlas200_compiler/aarch64-linux-gnu
│   ├─── bin
│   ├─── include
│   ├─── lib
│   ├─── lib64
│   └─── libc
├─── /opt/atlas200_compiler/bin
│   ├─── aarch64-linux-gnu-addr2line
│   ├─── aarch64-linux-gnu-ar
│   ├─── aarch64-linux-gnu-as
│   ├─── aarch64-linux-gnu-c++
│   ├─── aarch64-linux-gnu-c++filt
│   ├─── aarch64-linux-gnu-cpp
│   ├─── aarch64-linux-gnu-elfedit
│   ├─── aarch64-linux-gnu-g++
│   ├─── aarch64-linux-gnu-gcc
│   ├─── aarch64-linux-gnu-gcc-5.4.1
│   ├─── aarch64-linux-gnu-gcc-ar
│   ├─── aarch64-linux-gnu-gcc-nm
│   ├─── aarch64-linux-gnu-gcc-ranlib
│   ├─── aarch64-linux-gnu-gcov
│   ├─── aarch64-linux-gnu-gcov-dump
│   ├─── aarch64-linux-gnu-gcov-tool
│   ├─── aarch64-linux-gnu-gprof
│   ├─── aarch64-linux-gnu-ld
│   ├─── aarch64-linux-gnu-ld.bfd
│   ├─── aarch64-linux-gnu-nm
│   ├─── aarch64-linux-gnu-objcopy
│   ├─── aarch64-linux-gnu-objdump
│   ├─── aarch64-linux-gnu-ranlib
│   ├─── aarch64-linux-gnu-readelf
│   ├─── aarch64-linux-gnu-size
│   ├─── aarch64-linux-gnu-strings
│   └─── aarch64-linux-gnu-strip
├─── /opt/atlas200_compiler/include
├─── /opt/atlas200_compiler/lib
├─── /opt/atlas200_compiler/libexec
└─── /opt/atlas200_compiler/share
```

6.4.4 开发工具介绍

DDK工具所在目录：“$DDK_HOME/bin/x86_64-linux-gcc5.4”

工具	描述	说明
IDE-daemon-client	IDE Daemon命令工具集	具体说明请参考《IDE-daemon-client命令参考》。
omg	模型转换工具	可将Caffe或者Tensorflow模型转换为DDK支持的om模型文件，具体说明请参考《模型转换指导》。
protoc	第三方库protobuf的工具	可将proto文件转换为各个语言支持的协议。

6.5 通用推理业务流程介绍

本章主要介绍推理业务典型的业务流程。

6.5.1 软件业务流程介绍

图 6-2 业务流程图

图6-2展示了软件业务流程，具体流程说明请参考表6-2。

表 6-2 业务流程说明

过程	描述	说明
数据源	视频或图片来源	可以使IPC摄像机提供RTSP数据流，或者磁盘离线视频文件，图片文件等。
数据获取	实现数据获取	可以选择开源FFmpeg函数库拉流，或自行实现代码拉流、读文件。运行在HostCPU。
数据预处理	实现图像解码、缩放、色域转换等图片预处理功能	• 选择软解码时，调用开源OpenCV函数库，运行在HostCPU或CtrlCPU。 • 选择硬解码时，调用DVPP解码，缩放AIPP色域接口，实现预处理，再在Device侧运行对应硬件模块。
推理	实现模型推理功能	运行在Device侧AI Core和AI CPU。

过程	描述	说明
后处理	实现模型结果后处理	可以选择运行在Device侧Ctrl CPU或Host CPU。
输出	实现结果呈现	-

6.5.2 软件模块与硬件模块对应关系示例

Device侧的CtrlCPU为Arm处理器，由于处理能力有限，用户可根据业务流程、Host-Device数据传输及CtrlCPU负载情况，在Device侧实现通用的逻辑运算（预处理、后处理等）。

图 6-3 软件模块图

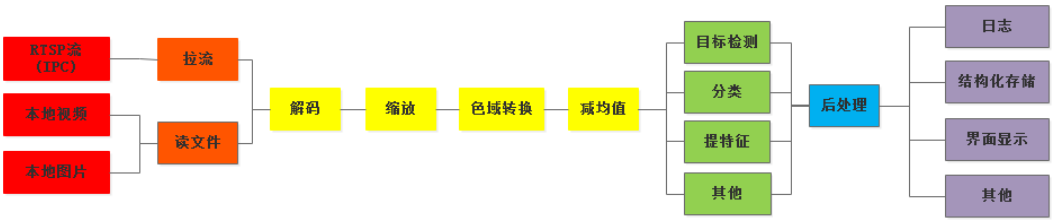
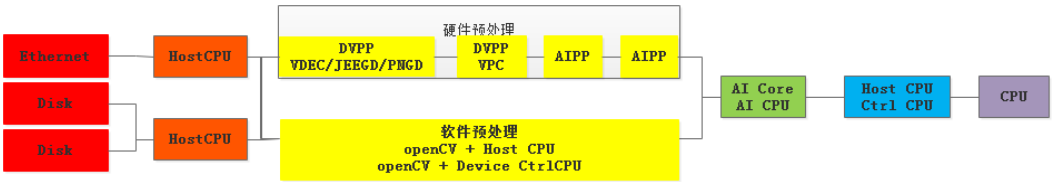


图 6-4 硬件模块图



6.6 代码运行样例（HelloDavinci 程序）

本章主要介绍入门程序HelloDavinci。用户可以通过这个入门样例了解基于Atlas产品的应用开发的过程，同时验证编译环境和运行环境等配置的正确性。

6.6.1 获取 HelloDavinci 代码

获取地址：**https://gitee.com/HuaweiAtlas/samples**。

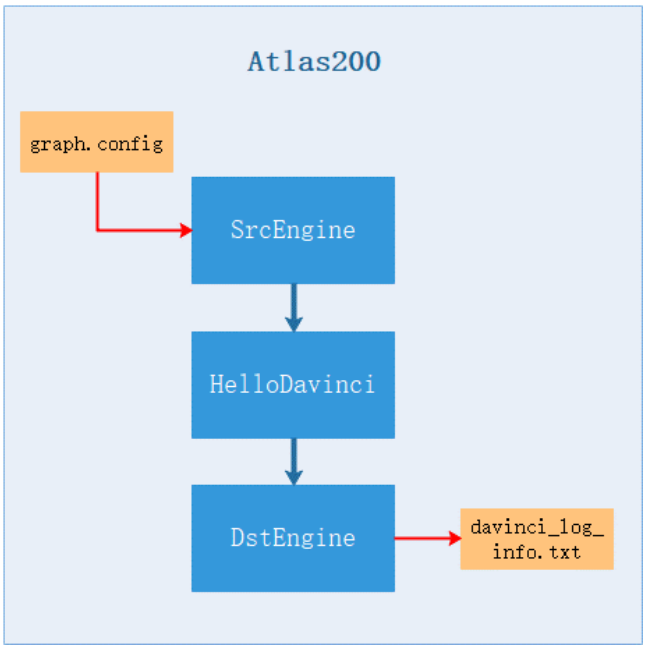
6.6.2 HelloDavinci 流程框架介绍

本开发样例以HelloDavinci程序为例子，主要演示各个引擎Engine间的数据传递。

说明

如无需了解本章节内容可跳过本节到**6.6.3 HelloDavinci编译运行**直接进行编译运行，查看运行结果。

图 6-5 运行流程



运行过程说明如下：

1. 程序运行从main开始，向SrcEngine发数据。
2. SrcEngine收到数据之后，读取graph.config中的配置转发给HelloDavinci。
3. HelloDavinci生成字符串，并发送到DstEngine。
4. DstEngine收到数据后将信息保存在“dacvinci_log_info.txt”下，并向main发送结束信号。
5. main函数收到结束信号后，销毁graph，在终端打印结束信息。
6. 退出程序。

6.6.3 HelloDavinci 编译运行

前提条件

1. 环境安装成功。
2. 已安装CMake（2.8.4版本以上）。
3. 已安装DDK。
4. 已配置环境变量（DDK_HOME：export DDK_HOME=<DDK install Path>/ddk/）。

编译步骤

步骤1 将获取到的samples文件拷贝至开发主机的某一个目录下。

步骤2 进入“Samples/HelloDavinci”目录。

步骤3 执行命令**bash build.sh A200**进行编译。编译成功后，将在“out/”目录下生成可执行文件main、so文件和配置文件graph.config等。

步骤4 执行命令**scp -r out HwHiAiUser@A200IP:~/**（A200IP表示Atlas 200系统的IP地址），将文件传输至Atlas 200系统上。

----结束

运行步骤

步骤1 在Atlas 200系统上，执行如下命令（ip为Atlas 200系统的IP地址）：

ssh HwHiAiUser@ip

步骤2 输入Atlas 200系统的密码，登录Atlas 200系统。

步骤3 在out目录下，执行**./main**命令查看结果输出，如返回如下结果，则表示运行成功。

- 运行成功显示如下：
```
HwHiAiUser@davinci-mini:~/out$ ./main
Hello Davinci!
The sample end!!
```
- 查看同一目录下的dacvinci_log_info.txt文件，显示如下：
```
HwHiAiUser@davinci-mini:~/out$ cat davinci_log_info.txt
This message is from HelloDavinci
```

----结束

6.7 软件代码开发

Atlas 200利用内部集成的Ascend 310芯片提供的算力，实现神经网络的推理加速。

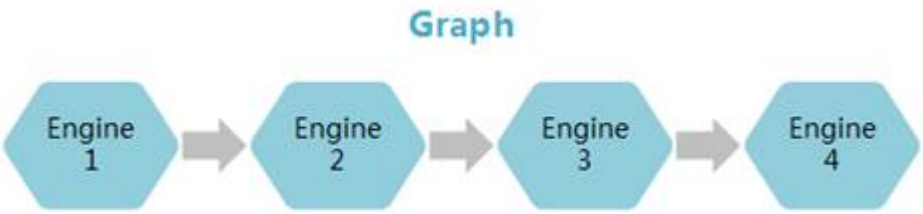
6.7.1 配置 Matrix 框架

提供Matrix框架是为了帮助开发者快速完成推理业务迁移，有助于开发者高效使用Ascend 310芯片的算力。Matrix框架核心为Graph和Engine。

Engine为业务的基本功能单元，允许用户自定义Engine的实现（输入图片数据、对图片进行分类处理、输出对图片数据的分类预测结果等）。每个Engine默认对应一个线程运行处理。

Graph管理若干Engine组成的业务流。每个Graph在Ascend 310拉起一个进程实现对应业务。Graph描述了Engine及Engine的数据传输关系。

图 6-6 Graph 与 Engine 关系

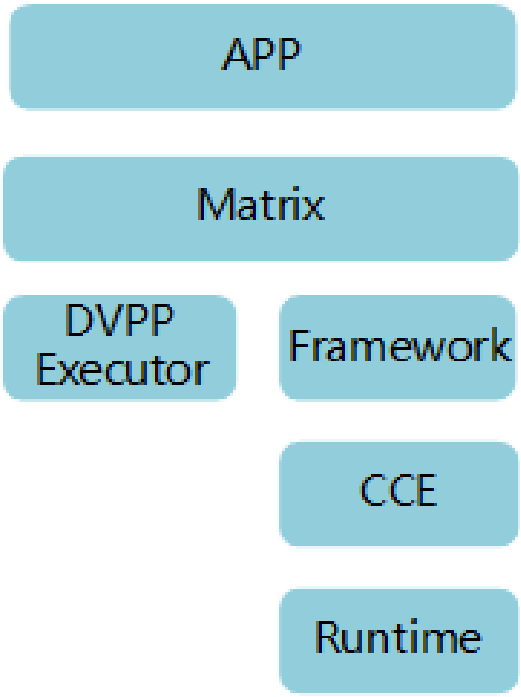


业务软件的逻辑框架如图6-7所示：

- Matrix为通用业务流程执行引擎，提供Ascend 310芯片功能接口、完成与APP进行控制命令和处理数据的交互、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、Engine调用DVPP的API接口实现媒体预处理以及Engine调用模型管家的API接口实现模型推理。

- DVPP Excutor提供了API接口实现视频编解码、图片编解码以及视觉预处理（包括裁剪、缩放等）功能。
- Framework为离线模型框架，提供接口实现离线模型转换、离线模型执行以及自定义算子实现。
- CCE为加速库，通过API的方式，为上层应用（Framework或者Matrix）提供加速。
- Runtime运行于APP进程空间，为APP提供了Ascend 310设备的Memory管理、Device管理、Stream管理、Event管理以及Kernel执行等功能。

图 6-7 业务软件逻辑图



6.7.1.1 配置、创建与销毁 Graph

Graph主要功能是描述业务包含的Engine和Engine间的数据传输关系，Matrix框架通过Protobuf定义了Graph的数据结构，用户以配置文件的方式来定义Graph配置。

说明

Graph关键字含义请参考**6.8 Graph关键字**，详情请参考《Matrix API参考》。

下文提供了一个简单的graph配置文件，该graph文件会创建一个id为1000的graph业务流，该业务流包含三个Engine，Engine间的数据传输关系如**图6-8**所示。

图 6-8 Engine 间的数据传输关系

```
graphs {
    graph_id: 1000          # 支持多graph，graph id需大于0，而且不能重复
```

```
    device_id: "0"          # graph运行的芯片id
    priority: 1             # 优先级

    # 用户自定义Engine，可实例化多个Engine，以id区分
    engines {
        id: 2001                                # 引擎ID
        engine_name: "ObjectDetectionEngine"     # 用户自定义Engine的类名
        side: DEVICE                            # 指定引擎运行在Host/Device侧
        so_name: "./libObjectDetectionEngine.so" # 动态链接库名称及路径
        # 用户可自定义配置参数
        ai_config{
            items{
                name: "model"                   # 模型名称
                value: "./FaceDetection.om"     # 模型的路径
            }
            items{
                name: "mode"
                value: "test"
            }
        }
    }
    engines {
        id: 1000
        engine_name: "DecodeEngine"
        side: DEVICE
        so_name: "./libDecodeEngine.so"
    }
    engines {
        id: 2002
        engine_name: "PostProcess"
        side: HOST
    }

    # 描述Engine间的端口连接
    connects {
        src_engine_id: 1000                     #数据传输源引擎ID
        src_port_id: 0                          #数据传输源端口号
        target_engine_id: 2001                  #数据传输目的引擎ID
        target_port_id: 0                       #数据传输目的端口号
    }
    connects {
        src_engine_id: 2001
        src_port_id: 0
        target_engine_id: 2002
        target_port_id: 0
    }
}
```

Matrix框架会将Device侧Engine依赖的文件动态链接库（由so_name指定），模型文件（在ai_config内添加）等拷贝到Device侧，供Engine线程使用。Matrix提供的三个常用Graph接口如**表6-3**所示，详细介绍请参考《Matrix API参考》。

表 6-3 接口说明

接口	说明
HIAI_StatusT HIAI_Init(uint32_t deviceID)	Ascend 310芯片初始化，请注意此处的芯片ID编号是Ascend 310芯片编号。 Atlas 200只有一个芯片，只能是0，这个地方需要注意：和Atlas 300 AI加速卡（型号 3010）（型号 3000）可能不一致。
static HIAI_StatusT Graph::CreateGraph(const std::string& configFile)	读取Graph配置文件，初始化Engine，创建线程和数据传输通道，完成业务流初始化。
static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)	销毁Graph，运行Engine的析构函数等。

6.7.1.2 配置 Engine

Engine是Matrix框架定义的业务软件基本功能单元，用户可继承Matrix定义的Engine模板类，创建业务中各功能模块的Engine（读取输入文件，图像预处理，神经网络推理，推理结果后处理，host/device数据传输等），详情请参考《Matrix API参考》。每一个Engine定义了函数Init()和Process()，在Graph初始化时，会自动运行Init()，从而实现Engine的参数初始化（包含内存分配和模型加载）。Process()接口实现数据的传输和业务逻辑。

- 常用接口介绍如表6-4所示。

表 6-4 接口说明

接口	描述	说明
HIAI_DEFINE_PROCESS（inputPortNum, outputPortNum）	端口数量（Engine的数据输入和输出数据通道）	Matrix框架为每一端口创建了队列用于缓存数据，Engine间端口的传输关系在Graph配置文件内指定，请参考6.8 Graph关键字。
HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription>&modelDesc)	Engine初始化	在创建Graph时此接口会被调用，初始化Engine，Graph配置文件中定义的ai_config会传给函数入参config，用户可在配置文件内添加自定义item，并在初始化函数内使用此item。

接口	描述	说明
HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)	Engine的Process	由数据驱动，即输入端口收到数据后，由框架启动Process运行，如果设置了多个输入端口，每一个输入端口接收到数据后，都会触发一次Process运行，因此如果业务处理依赖多个输入，用户需自行实现多输入的同步逻辑。框架已将Process封装成宏定义，用户实现该宏定义即可。

- Engine读取输入端口的数据，框架支持最大16个输入端口（arg0~arg15），用户可直接使用，从业务应用角度看，数据传输的是共享指针，传输代码示例如下：
```
// 发送Engine：传输自定义数据USER_DEFINE_TYPE到目标engine
std::shared_ptr<USER_DEFINE_TYPE > streamData= std::make_shared< USER_DEFINE_TYPE >();
// 对streamData进行赋值后，调用Senddata发送，发送需要将共享指针转换为void类型
hiai::Engine::SendData(0, " USER_DEFINE_TYPE ", std::static_pointer_cast<void>(deviceStreamData));
// 接收Engine：结构数据，并将数据转换为用户自定义类型USER_DEFINE_TYPE
std::shared_ptr< USER_DEFINE_TYPE > inputArg = std::static_pointer_cast< USER_DEFINE_TYPE >(arg0);
```

6.7.1.3 配置数据传输

Matrix框架定义的数据传输，根据业务应用分为以下三类，接口类似，传输的对象都为共享指针，但是根据场景不同接口使用方式有明显区别。

- Graph外传输数据到engine的输入端口：请参考《Matrix API参考》中的“Graph::SendData”章节。
- Graph内的engine输出端口传输数据到graph外：请参考《Matrix API参考》中的“Graph::SetDataRecvFunctor”和“Engine::SetDataRecvFunctor”章节。
- Graph内engine间的数据传输：请参考《Matrix API参考》中的“Engine::SendData”章节。

Engine 间的数据传输

在Atlas 200下Engine间的数据传输只有同侧传输，具体的定义为：传输的数据即为共享指针的地址，并未拷贝，由于Engine是线程运行，该方式即为线程间用共享内存的方式实现数据传输，但用户需要保证不出现非法访问。Matrix框架推荐用于传输的共享指针，在传输给下一个Engine后，本Engine不再修改共享指针。

Graph 外传数据到 engine 的输入端口

- 由Engine组成的业务流，由数据驱动，数据从Graph外传输到Graph内，即通过该接口实现。
- 以下接口可实现跨侧传输和同侧传输，要求与Engine间的数据传输一致，详情请参考《Matrix API参考》中的“Graph::SendData”章节。
```
HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const std::string& messageName, const std::shared_ptr<void>& dataPtr，const uint32_t timeOut = 500)
```

Graph 内的 Engine 输出端口传输数据到 Graph 外

- Matrix采用了以下两种设置回调函数的方式将Engine输出端口的数据传输到graph外。
  - 业务流节点Engine输出端口回调，请参考《Matrix API参考》的“Graph::SetDataRecvFunctor”章节。
  - Engine输出端口回调，请参考《Matrix API参考》中的“Engine::SetDataRecvFunctor”章节。
- 框架提供了回调函数的模板类DataRecvInterface。

6.7.2 调用 DVPP 接口

DVPP是Ascend 310芯片提供的图像预处理硬件加速模块，该模块集成的六个功能如下所示，接口介绍和使用方法请参考《DVPP API参考》。

- 格式转换，抠图与缩放（VPC）
- H264/H265视频解码（VDEC）
- H264/H265视频编码 （VENC）
- Jpeg图片解码（JPEGD）
- Jpeg图片编码（JPEGE）
- Png图片解码（PNGD）

6.7.2.1 使用 DVPP 接口

DVPP采用句柄方式提供接口，主要包含三类接口：创建，使用，销毁。

VPC、JPEGE、JPEGD、PNGD共用一套接口，输入的参数有差异。VDEC和VENC各使用一套接口。

- VDEC解码使用的三个接口如下所示，用户以异步方式调用，使用VdecCtl接口传入配置（包含回调函数）、H264/H265数据等，硬件解码后，Matrix框架调用回调函数将结果返回。

  说明

  VDEC解码后的数据为HFBC（内部格式），用户需要使用VPC将HFBC转换为YUV420SP格式，详情请参考《DVPP API参考》的“实现 VDEC 功能”章节中相关示例。

  ```
  int CreateVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
  int VdecCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG, int singleton)
  int DestroyVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
  ```

- VPC、JPEGE、JPEGD、PNGD使用如下所示相同的接口，不同功能传输的配置参数不同，详情请用户参考《DVPP API参考》中的“VPC/JPEGE/JPEGD/PNGD功能接口”章节。

  ```
  int CreateDvppApi(IDVPPAPI *&pIDVPPAPI)
  int DvppCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG)
  int DestroyDvppApi(IDVPPAPI *&pIDVPPAPI)
  ```

- 由于硬件限制，DVPP使用过程中存在部分限制。为了加快读写速度，图片长宽需要对齐到指定大小，但不影响有效区域，采用向左、向下填充0的方式，对齐到指定大小。

  例如：对于300*300的YUV420SP_UV图片，需要对齐到304*300（宽要求16对齐，高要求2对齐），有效区域仍为[0, 0]至[300, 300]，需要用户在图片右侧补零，对齐到304列。

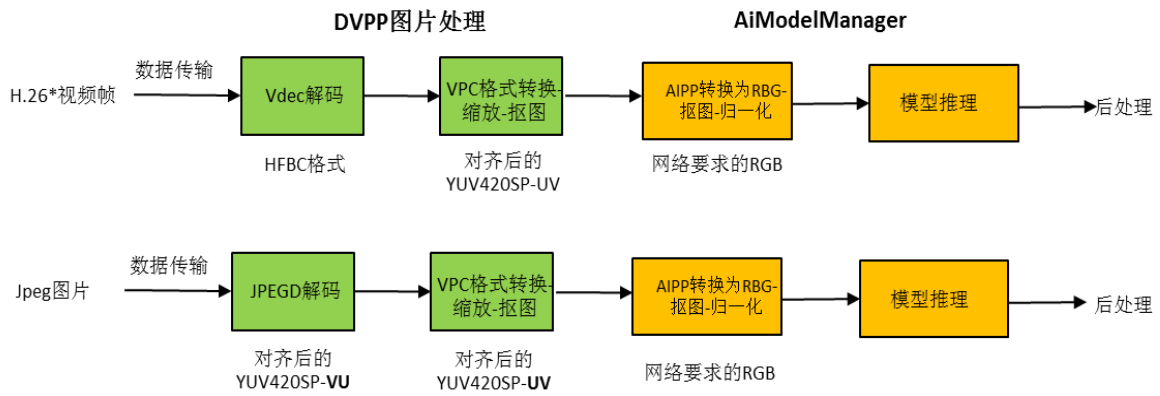
- 当用户使用DVPP的JPEGD、VDEC、PNGD组件读取输入图片时，解码后的图片需要满足长宽对齐要求（实际长宽），此时用户需要注意，以对齐后的图片大小申请输出图片内存。

  例如：对于300*300的YUV420SP_UV图片，需要申请304*300*3/2Byte（YUV420SP一个像素需要1.5Byte存储）。

  - VPC输入和输出内存地址，16字节对齐。
  - VPC输出图片的宽度，16字节对齐。
  - VPC输出图片的高度，2字节对齐。
  - VPC输入图片的宽度，16字节对齐。
  - VPC输入图片的高度，2字节对齐。
  - JPEGD输出图片的宽度，128字节对齐。
  - JPEGD输出图片的高度，16字节对齐。
- DVPP各组件基于处理速度和内存占用量的考虑，对输出图片有诸多限制，如输出图片需要长宽对齐，输出格式必须为YUV420SP等，但模型输入通常为RGB或BGR，且输入图片尺寸各异。因此，Ascend310芯片提供AIPP（Ai Preprocess），用于图片格式转换及抠图功能，具体说明请参考《模型转换指导》。

  例如：Jpeg图片输入和H26*视频输入的处理流程如图6-9所示。

图 6-9 视频与图片输入处理流程



6.7.2.2 申请 DVPP 的内存

Matrix针对DVPP提供了内存申请接口HIAI_DVPP_DMalloc和释放接口HIAI_DVPP_DFree，申请接口用于申请满足DVPP对齐要求的内存。用户在使用时注意该接口需要成对使用，为有效预防内存泄露问题，推荐使用共享指针管理申请的内存，实现代码如下：

```
uint8_t* buffer = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(dataSize,  (void*&) buffer);
std::shared_ptr<uint8_t> dataBuffer =  std::shared_ptr<uint8_t>( buffer, \
[](std::uint8_t* data) hiai::HIAIMemory::HIAI_DVPP_DFree(data);});
```

此外，该接口仅在Device侧使用，Host侧无该接口。HIAI_DVPP_DMalloc申请的内存可用于Device到Host的高速数据传输，在此方式下，内存由Matrix框架释放，详情请参考sample代码DvppDecodeResize。

6.7.3 离线模型推理

Ascend 310芯片支持Caffe、Tensorflow框架的推理加速。用户在完成训练后，需要先将训练后的模型转换成Ascend 310支持的模型文件om文件，之后，编写业务代码，调用Matrix框架提供的接口，完成业务功能。Matrix框架将AIPP（Ai Preprocess）和模型推理功能，封装到一个模块内。用户调用推理接口，Matrix框架会先调用AIPP，完成输入图片的预处理，之后将预处理后的图片，输入到模型推理模块，返回推理后的结果。

6.7.3.1 配置 AIPP

AIPP是Ascend 310提供的硬件图像预处理功能，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。

AIPP分为静态AIPP和动态AIPP：

- 静态AIPP：在模型转换时设置参数，模型推理过程采用固定的AIPP预处理（无法修改），静态AIPP的关键字含义和配置文件模板请参考《模型转换指导》“配置文件模板”章节。
- 动态AIPP：在模型转换时，设置AIPP为动态模式，每次模型推理前，根据需求，设置AIPP预处理参数。动态AIPP在根据业务要求改变预处理参数的场合下使用（如不同摄像头采用不同的归一化参数，输入图片格式需要兼容YUV420和RGB等）。动态AIPP的使用方法请参考《Matrix API参考》中的“AIPP配置接口”章节。

须知

- AIPP的输入格式为“YUV420SP_U8”（默认为“YUV420SP_UV”格式），若格式为“YUV420SP_VU”，请修改参数“rbuv_swap_switch”，否则会影响输出结果。
- 模型输入为“RGB_U8”或“BGR_U8”，对应不同的色域转换矩阵。

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。

例如：模型要求输入为300*300的RGB图片，用户调用DVPP接口处理（如Jpeg解码和缩放）后，由于对齐要求，输出的图片为384*304（图片有效区域为300*300，向右向下填充零补齐）的YUV420SP_UV格式图片。

静态AIPP的配置如下所示，文件配置了抠图的起始坐标，抠图的长宽默认为模型输入，图像归一化参数，即均值和方差的倒数（减均值后，乘以该系数）。

```
aipp_op{
# AIPP为静态模式
aipp_mode: static
# 使能抠图
crop: true
# 输入图片的格式及大小
input_format : YUV420SP_U8
```

```
src_image_size_w : 384
src_image_size_h : 304
# 抠图的起始坐标，抠图的宽与高默认为模型输入
load_start_pos_h : 0
load_start_pos_w : 0

# 使能格式转换，该转换矩阵将YUV420SP_UV转换为RGB888
csc_switch : true
matrix_r0c0 : 298
matrix_r0c1 : 516
matrix_r0c2 : 0
matrix_r1c0 : 298
matrix_r1c1 : -100
matrix_r1c2 : -208
matrix_r2c0 : 298
matrix_r2c1 : 0
matrix_r2c2 : 409
input_bias_0 : 16
input_bias_1 : 128
input_bias_2 : 128

#开启数据归一化，配置均值和方差的倒数
mean_chn_0 : 125
mean_chn_1 : 125
mean_chn_2 : 125
var_reci_chn_0 : 0.0039
var_reci_chn_1 : 0.0039
var_reci_chn_2 : 0.0039
}
```

6.7.3.2 转换离线模型

训练后的Caffe和Tensorflow模型，有两种初步评估的方法评估Ascend 310芯片是否支持用户模型，评估方法如下：

1. 模型使用的算子是否全部包含在算子清单内，包含即支持。
2. 使用模型转换工具直接转换训练好的模型，确认是否可以转换成功，转换成功即支持，不支持的算子将返回错误信息。

说明

1. 算子清单可参考《算子清单》。
2. 如存在不支持的算子，请参考《TE自定义算子开发指导》开发不支持的算子。

离线模型转换工具包含在DDK工具包内，位于目录“<$DDK_HOME>/uihost/bin/omg”下，omg为命令行工具（可通过**-h**获得参数信息），用于实现Caffe和Tensorflow模型转换成Ascend 310支持的om文件。Omg工具的使用说明请参考《模型转换指导》的“使用omg工具转换模型”章节。

1. Caffe模型的转换：
```
#omg --framework 0 --model <model.prototxt> --weight <model.caffemodel> --output <output name> --insert_op_conf <aipp.cfg>
```
2. Tensorflow模型的转换：
```
#omg --framework 3 --model <model.pb> --input_shape “input_name:1,112,112,3” --output <output_name> --insert_op_conf <aipp.cfg>
```

表 6-5 参数说明

参数	说明
framework 0	指定为Caffe模型。
framework 3	指定为Tensorflow模型。
model	指定模型文件。
weight	指定Caffe的权重文件。
output	支持输出om文件的文件名。
input_shape	指定输入层的名字和大小，Tensorflow默认为“input_layer_name: n, h, w, c”。
insert_op_conf	指定AIPP的配置文件。

6.7.3.3 执行模型推理

Matrix框架提供AIModelManager类，实现模型加载和推理功能，详情请参考《Matrix API参考》。

模型推理初始化

步骤1 在自定义推理模型Engine的Graph配置文件内设置模型路径（在ai_config内，添加items，设置模型在host路径）。

步骤2 使用Matrix框架将模型文件传输到Device侧。

步骤3 用户在自定义Engine内解析自定义items，获得模型在Device侧的路径。

步骤4 调用AIModelManager::Init()完成初始化，具体实现如下：

```
/* 用户自定义Engine的成员变量 */
std::shared_ptr<hiai::AIModelManager> modelManager;
/* 自定义Engine Init函数内实现AIModelManager初始化 */
std::vector<hiai::AIModelDescription> model_desc_vec;
hiai::AIModelDescription model_desc_;
......
/* 从Graph配置文件的ai_config结构内，解析出模型路径 */
model_desc_.set_path(model_path);// 设置模型路径
model_desc_vec.push_back(model_desc_);
ret = modelManager->Init(config, model_desc_vec);// config无意义，将Engine::Init的入参传入即可
```

----结束

设置模型推理输入与输出

- Matrix框架定义IAITensor类，用于管理模型推理的输入与输出矩阵。为了便于使用，Matrix框架基于IAITensor，派生出了AISimpleTensor和AINeuralNetworkBuffer。
- 模型推理的输入和输出采用HIAI_DMalloc接口申请，可减少一次内存拷贝。
- Matrix框架可自动释放AISimpleTensor管理的内存，建议由用户申请和释放，防止出现内存泄露或重复释放。

- 模型转换过程中，如启用了AIPP的抠图、格式转换、图片归一化等功能，则输入数据需经AIPP模块处理，再将处理后的数据进行真正的模型推理。

输入与输出的具体实现

输入输出的具体实现代码如下：

```
/* 获取推理模型的输入输出Tensor描述 */
std::vector<hiai::TensorDimension> inputTensorDims;
std::vector<hiai::TensorDimension> outputTensorDims;
ret = modelManager->GetModelIOTensorDim(modelName, inputTensorDims,
outputTensorDims);

/* 设置输入, 如果设置多个输入，需要依次创建并设置 */
std::shared_ptr<hiai::AISimpleTensor> inputTensor =
std::shared_ptr<hiai::AISimpleTensor>(new hiai::AISimpleTensor());
inputTensor->SetBuffer(<输入数据的内存地址>, <输入数据的长度>);
inputTensorVec.push_back(inputTensor);

/* 设置输出 */
for (uint32_t index = 0; index < outputTensorDims.size(); index++) {
hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
uint8_t* buf = (uint8_t*)HIAI_DMalloc(outputTensorDims[index].size);
......
std::shared_ptr<hiai::IAITensor> outputTensor = hiai::AITensorFactory::GetInstance()-
>CreateTensor(
outputTensorDesc, buf, outputTensorDims[index].size);
outputTensorVec.push_back(outputTensor);
}
```

模型推理

- Matrix框架提供了同步推理和异步推理两种方式，默认为同步推理，可通过设置AIContext配置项，使用回调函数实现异步推理。

  ```
  /* 模型推理 */
  hiai::AIContext aiContext;
  HIAI_StatusT ret = modelManager->Process(aiContext, inputTensorVec, outputTensorVec,
  0);
  ```

- AIModelManager对象加载多个模型，可通过设置AIContext配置项，指定模型（初始化阶段，指定的模型名），详情请参考《Matrix API参考》中的“离线模型管家”章节。

模型推理后处理

模型推理的结果矩阵，以内存+描述信息的方式保存在IAITensor对象内，用户需要根据模型的实际输出格式（数据类型和数据顺序），将内存解析成有效的输出。

```
/* 推理结果解析 */
for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
shared_ptr<hiai::AINeuralNetworkBuffer> resultTensor =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outputTensorVec[i]);
// resultTensor->GetNumber()  -- N
// resultTensor->GetChannel() -- C
// resultTensor->GetHeight()  -- H
// resultTensor->GetWidth()   -- W
// resultTensor->GetSize()    -- 内存大小
// resultTensor->GetBuffer()  -- 内存地址
}
```

说明

常见的分类模型后处理可参考示例代码“InferClassification”，SSD目标检测模型的后处理可参考示例代码“InferObjectDetection”。

6.7.4 优化软件性能

软件业务流开发过程中，需要考虑性能优化（内存使用，数据传输，高性能算子选择等），高性能软件编程建议请参考《高性能应用编程用户手册》。

6.7.5 软件日志调测

本章节介绍Atlas平台软件开发过程中如何编写日志打印和日志查看的代码，这些代码用于软件开发过程中打印调测日志，软件运行过程中记录异常。

6.7.5.1 日志注册

在使用日志接口之前，先进行错误码注册， 使用日志注册相关接口在头文件"hiaiengine/status.h"中定义。其详细步骤如下：

步骤1 自定义日志模块ID。

```
#define USE_DEFINE_ERROR 0x6001
```

自定义日志模块ID，0x6001为自定义值，不能跟status.h文件中的MODID_GRAPH等重复即可。

步骤2 自定义错误码的名称。

```
enum{
        HIAI_INVALID_INPUT_MSG_CODE = 0 // 错误码的名称，0为自定义值
};
```

步骤3 注册错误码。

函数格式：

```
HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)
```

表 6-6 参数说明

参数	说明
moduleId	模块ID
logLevel	错误级别
codeName	错误码的名称
codeDesc	错误码的描述，字符串

调用示例：

```
HIAI_DEF_ERROR_CODE(USE_DEFINE_ERROR, HIAI_ERROR, HIAI_INVALID_INPUT_MSG, "invalid input message pointer");
```

----结束

6.7.5.2 日志打印

通过HIAI_ENGINE_LOG来根据传入的参数调用不同的HIAI_ENGINE_LOG_IMPL函数，打印相关的接口在log.h中定义。日志打印格式有八种格式，本章结合日志注册的内容，介绍其中的一种。

- 函数格式

```
#define HIAI_ENGINE_LOG(...) \
HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,
const uint32_t  errorCode, const char* format, ...);
```

表 6-7 参数说明

参数	说明
errorCode	错误码
format	日志描述
...	format中的可变参数，根据日志内容添加

- 调用示例

  HIAI_ENGINE_LOG（HIAI_INVALID_INPUT_MSG, "RUNNING OK"）;

6.7.5.3 日志查看

更改日记级别

框架提供5种日志级别：ERROR>WARNING>INFO>DEBUG>EVENT

EVENT级别的日志输出全系统最关键日志，需要单独设置。对于其余级别，设置了日志对应级别后，此级别及高于此级别的日志都能打印出来。

1. 登录Atlas 200系统。
2. 更改“/etc/slog.conf”文件中“global_level”的值，即修改日志的等级。各个日志等级的对应“global_level”的值如表6-8所示。

   例如：若“global_level”的值为“3”，则打印的日志等级为ERROR，此时只有ERROR级别的日志被打印，Host侧和Device侧需要单独配置 。

表 6-8 日志等级说明

参数值	级别
0	DEBUG
1	INFO
2	WARNING
3	ERROR

代码实现示例如下：

```
# Global log level
global_level=3
# User
user=HwHiAiUser
# Share memory size of node 512k * 32 biggest support
maxNodeSize=524272
# Share memory count of queue
maxQueueCount=40
# log-agent-host #
logAgentMaxFileNum=8
# set host one log file max size, range is (0, 104857600]
logAgentMaxFileSize=10485760
# set host log dir
logAgentFileDir=/var/dlog
...
```

查看日志

- 日志路径：“/var/dlog”
- 日志命名格式：host-x_*.log（x代表芯片号）
- 日志压缩：日志文件默认是压缩的，无法直接查看，若需要直接查看，需要设置slog.conf文件或者联系产品支持技术人员申请获取专用解压工具进行解压再查看。用户需要在“/etc/slog.conf”文件中， 查看“zip_switch”的值是否为“0”，若不为“0”，将其值设置为“0”后重启slogd进程即可生效直接查看日志功能，slogd进程重启方法请参考**6.7.5.4 日志重启**。

  “/etc/slog.conf”文件内容：

```
...
#log server send log to IDE
#IP_address to bind: if you want to set ip_address manually, please open the item and add your ip.
#log_server_ip = 127.0.0.1
# Port: Please select a port number between 18000 and 18080([18000, 18080]).
daemon_socket_port=18080
#enable print event log, 0:disable, 1:enable
enableEvent=0
#zip switch, default : not zip
#0 : not zip
#1 : zip
zip_switch = 0
#zip level: 0 ~ 9, others err and exit, default:0
#zip_level=0
...
```

6.7.5.4 日志重启

步骤1 输入如下命令查找slogd和IDE-daemon-host进程ID：

```
HwHiAiUser@davinci-mini:/var/dlog$ ps aux |grep slogd
HwHiAiU+    667  0.0  0.0 237652  4952 ?        Ssl  15:37   0:00 /var/slogd
syslog     1732  0.0  0.0 256400  4320 ?        Ssl  Aug14   0:11 /usr/sbin/rsyslogd -n
HwHiAiU+  44956  0.0  0.0  21292   944 pts/14   S+   15:51   0:00 grep --color=auto slogd
HwHiAiUser@ davinci-mini:/var/dlog$ ps aux |grep IDE
HwHiAiU+  34640  0.0  0.0  39864  6304 ?        Ss   15:48   0:00 /var/IDE-daemon-device
HwHiAiU+  61918  0.0  0.0  21292  1024 pts/14   S+   15:56   0:00 grep --color=auto IDE
HwHiAiU+ 215378  0.2  0.0 1401252 18340 ?       Ssl  15:33   0:03 /var/IDE-daemon-host
```

步骤2 执行如下命令将用户切换为该进程的所属用户（HwHiAiUser）。

su HwHiAiUser

步骤3 执行如下命令结束slogd和IDE-daemon-host进程。

kill 667

kill 215378

步骤4 执行如下命令重新启动这两个进程，重新启动后即生效。

cd /var

./slogd

./IDE-daemon-host

----结束

6.7.6 业务软件编译

业务软件需要使用aarch64的编译工具和动态链接库进行交叉编译。推荐将主函数文件编译为一个可执行文件，所有的引擎Engine编译为一个动态连接库，故用户可以根据需要将业务划分为一个可执行文件+多个动态链接库，即一个Engine对应一个动态链接库。

例如：将main编译为可执行文件，将每一个Engine编译成动态链接库。

说明

业务软件的编译系统推荐采用Cmake搭建。CMake的编译系统，请参考示例工程HelloDavinci。

6.8 Graph 关键字

参数		说明	必选（M）/可选（O）
graph_id	-	Graph ID，为正整数。	O
priority	-	优先级，无需调整。	O
device_id	-	设备ID。不同Graph可以运行在一个或多个芯片上，也可以运行在不同PCIe卡的芯片上，如果运行在不同芯片上，需要在graph的配置中增加device_id项，用于指定Graph运行的device id。如果不指定，默认运行在device id为0的芯片上。device id的id值从0开始，到N-1结束（N表示Device个数）。	O
engines	id	Engine ID。	M
	engine_name	Engine名称。	M
	side	Engine运行target，取值为“HOST”或“DEVICE”，需要注意根据业务需求，配置Engine运行在HOST或DEVICE。	M

参数		说明	必选（M）/可选（O）
说明 多路解码时推荐配置多个Engine，一个Engine对应一个线程，如果一个Engine对应多个线程，解码时无法保证顺序。	so_name	Engine运行时需要从Host侧拷贝动态库so文件名到Device侧。如果FrameworkerEngine运行时，需要依赖第三方库文件或自定义库文件时，依赖的so文件也需要配置在graph文件中（请将以下示例中的“xxx”替换为实际依赖库名。） so_name: "./libFrameworkerEngine.so" so_name: "./libxxx.so" so_name: "./libxxx.so"	O
	thread_num	线程数量。多路解码时，该参数值推荐设置为“1”，如果“thread_num”的值大于“1”，线程之间的解码将无法保证顺序。	M
	thread_priority	线程优先级。取值范围从“1”到“99”，用于设置engine对应的数据处理线程的优先级，采用SCHED_RR调度策略，再根据策略将对应Engine设置为较高优先级。	O
	queue_size	队列大小，默认为“200”。 用户需要根据业务负载的波动、Engine接收数据大小、系统内存进行合理配置。	O

	ai_config	配置示例：
		ai_config{
		items{
		name: "model_path"
		value: "./test_data/model/resnet18.om"
		}
		}
		● name值无需配置。
		● value值配置为模型文件所在的路径，包含文件名（名称中只允许包含数字、字母、下划线和小数点），可以将参数值配置为单个模型文件的路径（“./test_data/model/resnet18.om”）；也可以将模型文件打包成tar包后，将参数值配置为tar包所在的路径（“./test_data/model/resnet18.tar”）。若存在多个AIConfigItem，在配置tar包路径时，不允许在同一个目录下同时存在名称相同但格式不同的文件，例如“./test_data/model/test.zip”和“./test_data/model/test.tar”。
			O

参数		说明	必选（M）/可选（O）

	ai_model	配置示例：
		ai_model{
		name: "" //模型名称
		type："" //模型类型
		version："" //模型版本
		size："" //模型大小
		path："" //模型路径
		sub_path："" //辅助模型路径
		key："" //模型密钥
		sub_key："" //辅助模型密钥
		frequency："UNSET" //设备频率，取值UNSET或0表示取消设置；LOW或1表示低频；MEDIUM或2表示中频；HIGH或3表示高频。
		device_frameworktype："NPU" //模型运行设备类型，取值为NPU/IPU/MLU/CPU。
		framework："OFFLINE" //执行框架，取值为OFFLINE/CAFFE/TENSORFLOW。
		}
			O

	oam_config	配置dump算法数据的方式。当推理不准确时，可以查看某些层或全部层的算法结果。
		配置示例：
		oam_config{
		items{
		model_name: "" //相对路径+模型名称+后缀
		is_dump_all："" //是否全部dump，取值为"true"或"false"。
		layer: "" //层
		}
		dump_path："" //dump路径
		}
			O

	internal_so_name	内置在Device侧的动态库文件，用户可直接使用，无需从Host侧拷贝到Device侧。	O
	wait_inputdata_max_time	当前收到数据后等待下一个数据的最大超时时间。 ● 单位：毫秒。 ● 默认为“0”，表示数据不超时。	O
	is_repeat_timeout_flag	针对Engine未收到数据，是否进行循环超时处理（唤醒），与wait_inputdata_max_time配合使用，取值说明如下： ● 0：不重复，默认为0。 ● 1：重复。	O
	holdModelFileFlag	是否保留本Engine的模型文件，取值说明如下： ● 0：不保留。 ● 非0：保留。	O

参数		说明	必选（M）/可选（O）
connects	src_engine_id	源Engine ID。	M

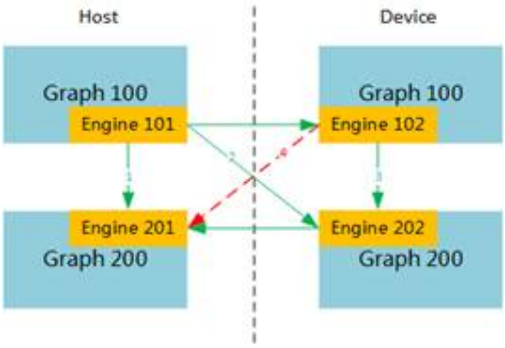
	src_port_id	源Engine的发送端口号。
		对于某个Engine，其端口号从“0”开始顺次增加。端口个数定义一般在相应的头文件中以HIAI_DEFINE_PROCESS宏来定义，代码实现如下：
		#define SOURCE_ENGINE_INPUT_SIZE 1
		#define SOURCE_ENGINE_OUTPUT_SIZE 1
		class SrcEngine : public Engine {
		
		HIAI_DEFINE_PROCESS(SOURCE_ENGINE_INPUT_SIZE, SOURCE_ENGINE_OUTPUT_SIZE)
		}
			M

	target_engine_id	目的Engine ID。	M
	target_port_id	目的Engine的接收端口号。	M
	target_graph_id	目的Graph ID。 Engine支持跨Graph串接，在该场景下，需要在connects的配置中增加“target_graph_id”项来表示接收端的Graph ID，如不配置，则默认在同一个graph中串接。 在有多个Ascend 310芯片情况下，可以使用跨Graph串接将各个模型运行在不同的Ascend 310芯片上，故尽量将相同的模型放在一个芯片上进行推理，减少不必要的内存消耗。 【场景说明】 ● 如表格下图6-10所示，Engine在以下场景中支持跨Graph串接： 1. Graph A的Host Engine直接给Graph B的Host Engine发送数据。 2. Graph A的Host Engine直接给Graph B的Device Engine发送数据。 3. Graph A的Device Engine直接给Graph B的Device Engine发送数据。 ● Engine在以下场景中不支持跨Graph串接：Graph A的Device Engine直接给Graph B的Host Engine发送数据，如表格下图6-10所示。	O

版权所有 © 华为技术有限公司

参数		说明	必选（M）/可选（O）
	receive_memory_without_dvpp	• 参数默认值为“0”，表示Deivce上运行的目标Engine的接收内存需要满足4G地址空间限制； • 若将参数值配置为“1”，表示Device上运行的目标Engine的接收内存不用满足4G地址空间限制。 • 用户可在Graph配置文件中将所有运行在Device上的目标Engine的参数“receive_memory_without_dvpp”（在connects属性下）设置为“1”，也可以将所有connects属性下的“receive_memory_without_dvpp”参数都配置为“1”，由于在Host上运行的目标Engine即使设置了该参数也无影响，故Matrix接收内存池无4G地址空间限制，从而提高系统内存使用率。 **说明** 当前DVPP中，VPC、JPEGE、JPEGD、PNGD功能的输入内存需要满足4G地址空间限制，VDEC、VENC功能的输入内存可以无4G地址空间限制。	O

图 6-10 Engine 跨 Graph 串接场景



7 高性能应用编程

7.1 简介
7.2 要点介绍
7.3 算子使用建议
7.4 示例说明
7.5 FAQ

7.1 简介

目的

本文档详细描述在Ascend 310上实现高性能应用的约束和建议，帮助用户快速理解样例，使用户能够快速构建起自己的高性能应用。

范围

本文档主要描述Ascend 310上编程实现高性能的要点，适合在Ascend 310上使用芯片硬件解码、推理、以及通用图像处理来完成应用的用户。

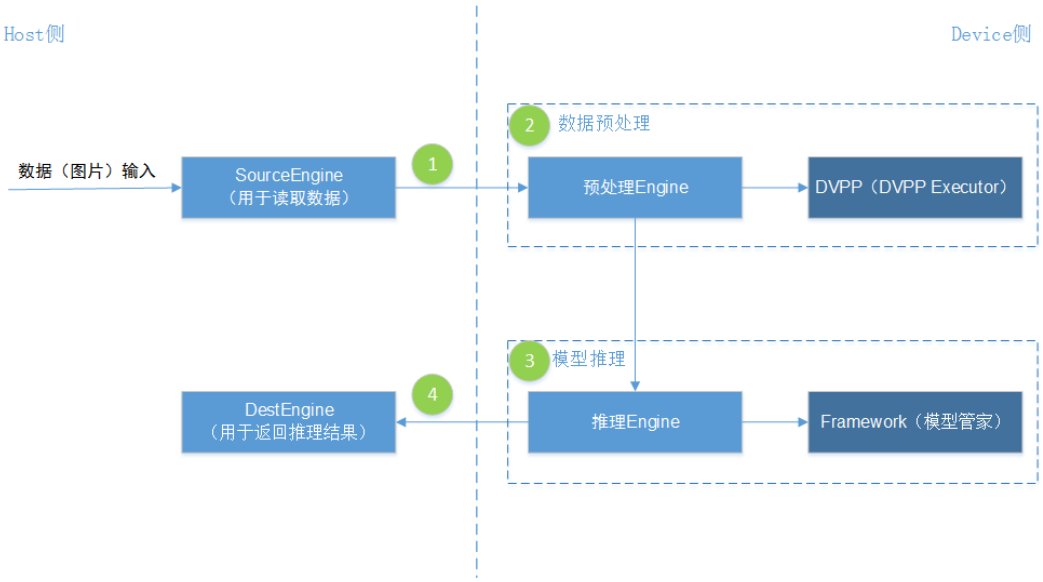
7.2 要点介绍

应用开发过程中，Matrix对数据的处理流程分为以下几个阶段：

1. Matrix调用接口将数据（图片）从Host侧搬运到Device侧。
2. Matrix调用DVPP的接口对数据进行编解码、缩放等处理。
3. Matrix调用Framwork提供的模型管家接口对数据进行推理。
4. Matrix调用接口将推理结果从Device侧回传到Host侧。

版权所有 © 华为技术有限公司

图 7-1 处理流程图



在Matrix的数据处理过程中，主要的消耗在搬运数据、DVPP处理以及模型推理上，本文档主要介绍这几个处理环节及它们之间的衔接环节如何进行较高性能的处理。

关于各接口的详细说明，请分别参见《Matrix API参考》、《DVPP API参考》。

7.2.1 内存管理

Ascend 310支持以下两种内存管理方式：原生语言的内存管理接口和Matrix框架提供的内存管理接口。

7.2.1.1 原生语言的内存管理接口

原生语言的内存管理接口包括malloc、free、memcpy、memset、new、delete等接口，支持C/C++等语言，由此类接口申请的内存，用户可以自行管理和控制内存使用的生命周期。用户申请内存空间小于256k时，使用原生语言的内存接口与Matrix框架提供的内存管理接口在性能上区别不大，基于简单便捷考虑，建议使用原生语言的内存管理接口。

使用原生语言的内存管理接口代码示例：

```
// Use malloc to alloc buffer
unsigned char* inbuf = (unsigned char*)malloc( fileLen );
// free buffer
free(inbuf);
inbuf = nullptr;
```

7.2.1.2 Matrix 框架提供的内存管理接口

框架单独提供了一套内存分配和释放接口，包括HIAI_DMalloc/HIAI_DFree、HIAI_DVPP_DMalloc/HIAI_DVPP_DFree，支持C/C++语言。其中，HIAI_DMalloc/HIAI_DFree接口主要用于申请内存，再配合SendData接口从Host侧搬运数据到Device侧；HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口主要用于申请Device侧DVPP使用的内存。通过调用HIAI_DMalloc/HIAI_DFree、 HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口申请内存，能够尽量少拷贝，减少流程处理时间。

版权所有 © 华为技术有限公司

接口描述

关于HIAI_DMalloc/HIAI_DFree、 HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口的功能，如表7-1所示。

表 7-1 接口描述

接口名称	接口功能
HIAIMemory::HIAI_DMalloc（C++专用接口）	申请内存接口， 该内存类似普通内存，在用跨侧传输（Host-Device/Device-Host）以及模型推理处理时，性能更优。
HIAIMemory::HIAI_DFree（C++专用接口）	与HIAIMemory::HIAI_DMalloc配合使用， 用于释放HIAIMemory::HIAI_DMalloc分配的内存。 如果在调用HIAIMemory::HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAIMemory::HIAI_DFree，对端接收到数据后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAIMemory::HIAI_DFree释放内存。
HIAI_DMalloc（C语言与C++通用接口）	申请内存接口， 该内存类似普通内存，在用跨侧传输（Host-Device/Device-Host）以及模型推理处理时，性能更优。
HIAI_DFree（C语言与C++通用接口）	与HIAI_DMalloc配合使用， 用于释放HIAI_DMalloc分配的内存。 如果在调用HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAI_DFree，程序运行结束后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAI_DFree释放内存。
HIAIMemory::HIAI_DVPP_DMalloc（C++专用接口）	申请内存接口，该接口主要给Device端分配DVPP使用的内存。
HIAIMemory::HIAI_DVPP_DFree（C++专用接口）	用于释放内存， 该接口主要用于释放HIAIMemory::HIAI_DVPP_DMalloc接口分配的内存。

接口名称	接口功能
HIAI_DVPP_DMalloc（C语言与C++通用接口）	申请内存接口，该接口主要给Device端分配DVPP使用的内存。
HIAI_DVPP_DFree（C语言与C++通用接口)	用于释放内存， 该接口主要用于释放HIAI_DVPP_DMalloc接口分配的内存。

接口调用流程

图 7-2 接口调用流程

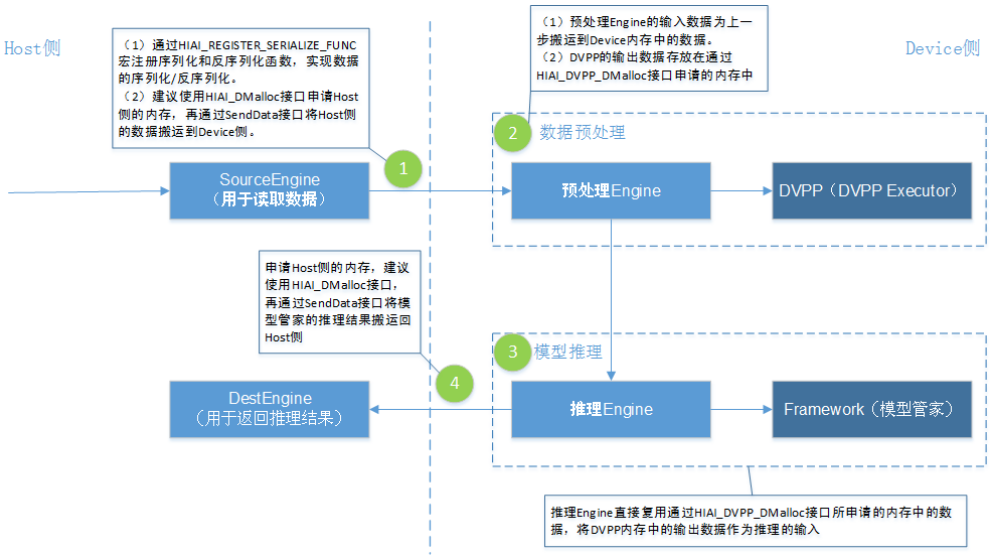


图7-2中关于接口使用的说明如下：

- 通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请的内存可用于端到端数据传输以及模型推理时使用。调用HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口，同时配合使用HIAI_REGISTER_SERIALIZE_FUNC宏（对用户自定义数据类型进行序列化或反序列化），可使数据传输效率更高，性能更优。

  通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存，**有以下优势**：
  - 申请的内存是可以直接给到HDC做数据搬运的， 这样可以避免Matrix与HDC间的数据拷贝。
  - 申请的内存可以直接使能模型推理零拷贝机制，减少数据拷贝时间。
- 通过HIAI_DVPP_DMalloc或HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存可以给DVPP使用， 同时也可以在DVPP使用完后透传给模型推理时使用。如果不需要做模型推理，HIAI_DVPP_DMalloc接口申请的内存中的数据可以直接回传给Host侧。
- 通过HIAI_DMalloc、HIAIMemory::HIAI_DMalloc、HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存兼容原生语言的内存管理接口， 可以当做普通内存使用，但是不能使用free、delete等来释放。一般来说，通过HIAI_DMalloc、HIAIMemory::HIAI_DMalloc、HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存，需要分别通过调用HIAI_DFree、HIAIMemory::HIAI_DFree、HIAI_DVPP_DFree、HIAIMemory::HIAI_DVPP_DFree接口释放内存。

如果在调用HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口时，将flag参数值设置为MEMORY_ATTR_AUTO_FREE，且通过SendData接口发送数据到对端，则无需调用HIAI_DFree或HIAIMemory::HIAI_DFree接口，对端接收到数据后，内存会自动释放；如果仅将flag参数值设置为MEMORY_ATTR_AUTO_FREE，但没有通过SendData接口发送数据到对端或者通过SendData接口发送数据失败，则需要调用HIAIMemory::HIAI_DFree释放内存。

- 通过HIAI_DVPP_DMalloc、HIAIMemory::HIAI_DVPP_DMalloc接口申请的内存是满足DVPP要求的内存，所以资源较为有限，建议仅当DVPP处理场景时使用。

接口使用要点

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存，关于内存管理，**请注意以下要点**：

- 申请自动释放内存，用于Host到Device或Device到Host的数据传输时，如果是智能指针，由于Matrix框架自动释放内存，所以智能指针指定的析构器必须是空的；如果非智能指针，则Matrix框架自动释放。
- 申请手动释放内存，用于Host到Device或Device到Host的数据传输时，如果是智能指针，则需要指定析构器为HIAI_DFree或HIAIMemory::HIAI_DFree；如果非智能指针，则数据发送完成后需要调用HIAI_DFree或HIAIMemory::HIAI_DFree释放内存。
- 申请自动释放内存，对于该内存中的数据，不允许多次调用SendData接口发送数据。
- 申请手动释放内存时，如果用于Host到Device或Device到Host的数据传输时，在内存释放前，不可复用内存中的数据；如果用于Host到Host或Device到Device的数据传输时，在内存释放前，可以复用内存中的数据。
- 申请手动释放内存时，如果调用SendData接口异步传输数据，发送数据后，不允许修改内存中的数据。

如果调用HIAI_DVPP_MAlloc或HIAIMemory::HIAI_DVPP_DMalloc接口申请内存，用于Device到Host的数据传输时，由于HIAI_DVPP_MAlloc或HIAIMemory::HIAI_DVPP_DMalloc没有自动释放标签，所以一定需要调用HIAI_DVPP_DFree或HIAIMemory::HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址， 必须指定析构器为HIAI_DVPP_DFree或HIAIMemory::HIAI_DVPP_DFree。

接口调用示例

```
(1)使用性能优化方案传输数据，必须对发送数据的接口进行手动序列化和反序列化：
// 注：序列化函数在发送端使用，反序列化在接收端使用，所以这个注册函数最好在接收端和发送端都注册一遍；
//数据结构
typedef struct
{
    uint32_t left_offset = 0;
    uint32_t right_offset = 0;
    uint32_t top_offset = 0;
    uint32_t bottom_offset = 0;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(left_offset,right_offset,top_offset,bottom_offset);
    }
} crop_rect;
```

```
// 注册Engine将流转的结构体
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // 传输Buffer
    uint32_t buffer_size = 0;                         // 传输Buffer大小
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
}EngineTransNewT;

//序列化函数
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : data_ptr         结构体指针
* @param [out]：struct_str       结构体buffer
* @param [out]：data_ptr         结构体数据指针buffer
* @param [out]：struct_size      结构体大小
* @param [out]：data_size        结构体数据大小
*/
void GetTransSearPtr(void* data_ptr, std::string& struct_str,
    uint8_t*& buffer, uint32_t& buffer_size)
{
    EngineTransNewT* engine_trans = (EngineTransNewT*)data_ptr;
    uint32_t dataLen = engine_trans->buffer_size;
    uint32_t dataLen_extend = engine_trans->buffer_size_extend;
    // 获取结构体buffer和size
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // 序列化处理
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}

//反序列化函数
/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,            反序列化Trans数据
* @param [in] : ctrl_ptr             结构体指针
* @param [in] : data_ptr             结构体数据指针
* @param [out]：std::shared_ptr<void> 传给Engine的指针结构体指针
*/
std::shared_ptr<void> GetTransDearPtr(
    const char* ctrlPtr, const uint32_t& ctrlLen,
    const uint8_t* dataPtr, const uint32_t& dataLen)
{
    if(ctrlPtr == nullptr) {
        return nullptr;
    }
    std::shared_ptr<EngineTransNewT> engine_trans_ptr = std::make_shared<EngineTransNewT>();
    // 给engine_trans_ptr赋值
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engine_trans_ptr));
    uint32_t offsetLen = engine_trans_ptr->buffer_size;
    if(dataPtr != nullptr) {
        (engine_trans_ptr->trans_buff).reset((const_cast<uint8_t*>(dataPtr)), ReleaseDataBuffer);
        // 因为trans_buff和trans_buff_extend指向的是一块以dataPtr为首地址的连续内存空间，
```

```
        // 因此只需要trans_buff挂载析构器释放一次即可
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}

// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);

(2) 在发送数据时，需要使用注册的数据类型，另外配合使用HIAI_DMalloc分配数据内存，可以使性能更优
     注：在从host侧向Device侧搬运数据时，使用HIAI_DMalloc方式将很大的提供传输效率，建议优先使用
HIAI_DMalloc，该内存接口目前支持0 – (256M Bytes - 96 Bytes)的数据大小，如果数据超出该范围，则需要使
用malloc接口进行分配；
     // 使用Dmalloc接口申请数据内存，10000为时延，为10000毫秒，表示如果内存不足，等待10000毫秒；
     HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(width*align_height*3/2,(void*&)align_buffer, 10000);

     // 发送数据，调用该接口后无需调用HIAI_DFree接口，10000为时延
     graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```

7.2.2 Host-Device 数据传输

```
// EngineTransNewT结构体
typedef struct EngineTransNew
{
    std::shared_ptr<uint8_t> trans_buff = nullptr;    // 传输Buffer
    uint32_t buffer_size = 0;                         // 传输Buffer大小
    std::shared_ptr<uint8_t> trans_buff_extend = nullptr;
    uint32_t buffer_size_extend = 0;
    std::vector<crop_rect> crop_list;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(buffer_size, buffer_size_extend, crop_list);
    }
}EngineTransNewT;

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : dataPtr          结构体指针
* @param [out]：structStr        结构体buffer
* @param [out]：buffer           结构体数据指针buffer
* @param [out]：buffSize         结构体数据大小
*/
void GetTransSearPtr(void* data_ptr, std::string& struct_str,
    uint8_t*& buffer, uint32_t& buffer_size)
{
    EngineTransNewT* engine_trans = (EngineTransNewT*)data_ptr;
    uint32_t dataLen = engine_trans->buffer_size;
    uint32_t dataLen_extend = engine_trans->buffer_size_extend;
    // 获取结构体buffer和size
    buffer_size = dataLen + dataLen_extend;
    buffer = (uint8_t*)engine_trans->trans_buff.get();

    // 序列化处理
    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engine_trans));
    struct_str = outputStr.str();
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,            反序列化Trans数据
* @param [in] : ctrlPtr              结构体指针
* @param [in] : ctrlLen              数据结构中控制信息大小
* @param [in] : dataPtr              结构体数据指针
```

```
* @param [in] : dataLen            结构中数据信息存储空间大小，仅用于校验，不表示原始数据信息大小
* @param [out]：std::shared_ptr<void> 传给Engine的指针结构体指针
*/
std::shared_ptr<void> GetTransDearPtr(
    const char* ctrlPtr, const uint32_t& ctrlLen,
    const uint8_t* dataPtr, const uint32_t& dataLen)
{
    if(ctrlPtr == nullptr) {
        return nullptr;
    }
    std::shared_ptr<EngineTransNewT> engine_trans_ptr = std::make_shared<EngineTransNewT>();
    // 给engine_trans_ptr赋值
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engine_trans_ptr));
    uint32_t offsetLen = engine_trans_ptr->buffer_size;
    if(dataPtr != nullptr) {
        (engine_trans_ptr->trans_buff).reset((const_cast<uint8_t*>(dataPtr)), ReleaseDataBuffer);
        // 因为trans_buff和trans_buff_extend指向的是一块以dataPtr为首地址的连续内存空间，
        // 因此只需要trans_buff挂载析构器释放一次即可
        (engine_trans_ptr->trans_buff_extend).reset((const_cast<uint8_t*>(dataPtr + offsetLen)),
SearDeleteNothing);
    }
    return std::static_pointer_cast<void>(engine_trans_ptr);
}
// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

在较大图像数据或码流传输的情况下，用HIAI_REGISTER_SERIALIZE_FUNC对自定义数据类型进行序列化/反序列化，可以实现高性能数据传输，节省传输时间。

Matrix通过“控制信息 + 数据信息”的形式描述要传输的数据，控制信息指用户自定义的数据类型，数据信息指需要传输的数据内容。为保证Host和Device之间的数据传输，Matrix提供如下机制：

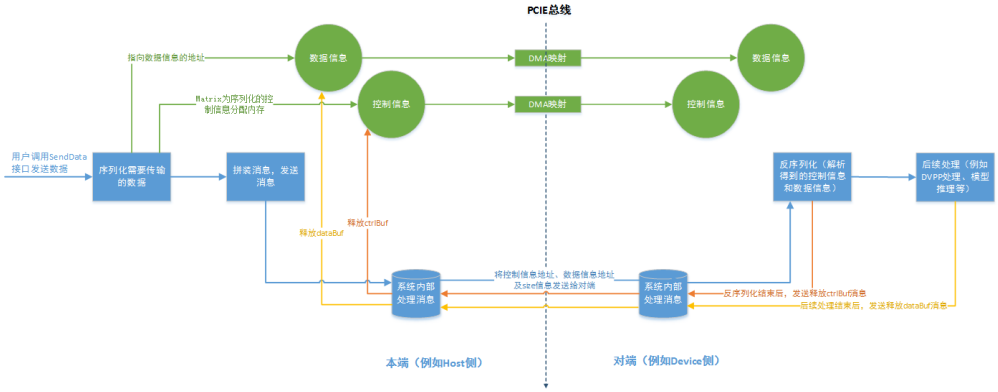
1. 在传输数据前，用户可调用HIAI_REGISTER_SERIALIZE_FUNC宏注册用户自定义数据类型、用户自定义序列化函数、用户自定义反序列化函数。
2. 用户在本端调用SendData接口发送数据后，Matrix会做如下处理，处理流程如图7-3所示：
   a. 调用用户自定义的序列化函数对控制信息序列化，将序列化后的控制信息放入内存（ctrlBuf）。
   b. 通过DMA（Direct Memory Access）映射将控制信息拷贝一份存放到对端的内存中，并保持本端与对端之间控制信息的映射关系。

      指向数据信息的内存（dataBuf）指针已通过SendData接口的入参传入，dataBuf是由用户调用HIAI_DMalloc/HIAI_DVPP_DMalloc接口申请的，申请该内存后系统将本端的数据信息通过DMA映射拷贝一份存放到对端的内存中，并保持本端与对端之间数据信息的映射关系。
   c. 拼装消息发送给对端，主要将ctrlBuf的地址及大小、dataBuf的地址及大小发送到对端。
   d. 对端收到消息后，Matrix会调用用户自定义的反序列化函数解析对端已获取到的控制信息和数据信息，并将解析后的数据发送给对应的接收Engine进行处理。
   e. 对端解析数据后，控制信息已使用完成，因此可以释放掉存放控制信息的内存（ctrlBuf），但由于存放控制消息的内存是在本端申请的，因此对端需要给本端发送释放ctrlBuf的消息。
   f. 本端收到消息后，释放ctrlBuf。
   g. Engine在收到数据并完成所有处理后，便可以释放dataBuf了，但由于Matrix并不知道用户何时会使用完dataBuf，因此需要用户在实现反序列化函数时，

dataBuf以智能指针返回并绑定析构器hiai::Graph::ReleaseDataBuffer。当智能指针结束生命周期析构时，便会自动调用析构器给本端发送释放dataBuf内存消息。

h. 本端收到消息后，释放dataBuf。

图 7-3 数据处理流程



示例场景：以下代码声明了序列化/反序列化函数，自定义了数据类型（结构体），并将自定义数据类型、序列化函数、反序列化函数通过HIAI_REGISTER_SERIALIZE_FUNC宏注册。在数据传输发起侧的数据传输前，框架调用注册的序列化函数，在数据接收侧的数据传输后，调用注册的反序列化函数。

```
// EngineTransNewT结构体
struct EngineTransNewT
{
    std::shared_ptr<uint8_t> transBuff;
    uint32_t bufferSize;   // buffer大小
    std::string url;
    //下面serialize函数用于序列化结构体
    template <class Archive>
    void serialize(Archive & ar)
    {
        ar(url);
    }
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,        序列化Trans数据
* @param [in] : dataPtr         结构体指针
* @param [out]:  structStr        结构体buffer
* @param [out]:  buffer           结构体数据指针buffer
* @param [out]:  buffSize         结构体数据大小
*/
void GetTransSearPtr(void* dataPtr, std::string& structStr, uint8_t*& buffer, uint32_t& bufferSize)
{
    EngineTransNewT* engineTrans = (EngineTransNewT *)dataPtr;
    std::shared_ptr<uint8_t> transBuff = ((EngineTransNewT *)dataPtr)->transBuff;
    buffer = (uint8_t*)engineTrans->transBuff.get();
    bufferSize = engineTrans->bufferSize;

    engineTrans->transBuff = nullptr;
    engineTrans->buffSize = 0;

    std::ostringstream outputStr;
    cereal::PortableBinaryOutputArchive archive(outputStr);
    archive((*engineTrans));
    struct_str = outputStr.str();

    ((EngineTransNewT*)dataPtr)->transBuff = transBuff;
```

```
    engineTrans->buffSize = bufferSize;
}

/**
* @ingroup hiaiengine
* @brief GetTransSearPtr,            反序列化Trans数据
* @param [in] : ctrlPtr             结构体指针
* @param [in] : ctrlLen             数据结构中控制信息大小
* @param [in] : dataPtr             结构体数据指针
* @param [in] : dataLen             结构中数据信息存储空间大小，仅用于校验，不表示原始数据信息大小
* @param [out]：std::shared_ptr<void> 传给Engine的指针结构体指针
*/
std::shared_ptr<void> GetTransDearPtr(char* ctrlPtr, const uint32_t& ctrlLen, uint8_t* dataPtr, const
uint32_t& dataLen)
{
    std::shared_ptr<EngineTransNewT> engineTransPtr = std::make_shared<EngineTransNewT>();
    std::istringstream inputStream(std::string(ctrlPtr, ctrlLen));
    cereal::PortableBinaryInputArchive archive(inputStream);
    archive((*engineTransPtr));
    engineTransPtr->bufferSize = dataLen;
    engineTransPtr->transBuff.reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
    return std::static_pointer_cast<void>(engineTransPtr);
}
// 注册EngineTransNewT
HIAI_REGISTER_SERIALIZE_FUNC("EngineTransNewT", EngineTransNewT, GetTransSearPtr, GetTransDearPtr);
```

7.2.3 DVPP 使用

7.2.3.1 图像/视频编解码

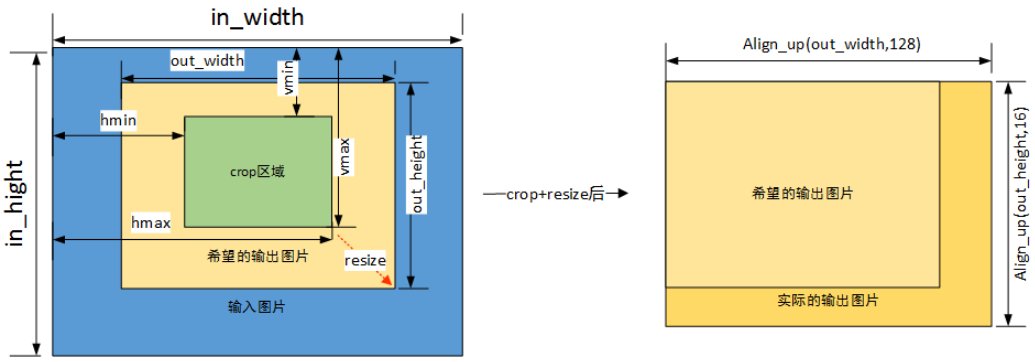
框架提供了图像处理单元以及视频编解码能力的调用接口，用户可以根据实际情况，将图像的解码/视频的解码放到Device上，以减少从Host到Device传输的数据量，同时降低数据传输时间开销和带宽压力。

在Host侧，通过调用Matrix框架提供HIAI_DMalloc申请Device侧的内存，作为图像/视频编解码的输入使用，数据存放的内存位置建议起始地址128对齐。在Device侧，DVPP完成图像/视频预处理后，调用Matrix框架提供HIAI_DVPP_DMalloc申请内存，作为图像预处理后的输出使用。

7.2.3.2 图像 Crop/Resize

在Ascend 310上编程，图像crop/resize推荐使用DVPP来实现。

图 7-4 crop/resize 运行示意图



crop/resize运行示意图如图7-4所示，它可以完成在图像中对ROI区域进行截图并使用这个截图进行重采样的过程。截图称为crop，重采样称之为resize。当resize系数为1时，相当于只做crop。当crop为原图时，相当于只做resize。

crop/resize对输入的限制如下：

- 输入数据地址（os虚拟地址）：16字节对齐。
- 输入图像宽度内存限制：16字节对齐。
- 输入图像高度内存限制：2字节对齐。

基于上述限制，高性能的编程方式要实现“0拷贝”则需要满足从Device（接收端）给用户的内存地址开始就满足限制。一般的做法根据输入不同分为以下两种做法。

- 方法一：在Host进行解码或者在其他硬件进行解码的应用，在Host发送端将数据就做好裁剪或者padding，满足16*2对齐，这样框架在数据接收端会自动的申请满足上述限制的数据内存。

说明

示例的输入数据是YUV图片，所以计算图片SIZE都是通过(ImageWidth * ImageHeight) * 3/2的方式，其他格式的数据，根据图片格式自行调整。

```
static const uint32_t ALIGN_W = 16;
static const uint32_t ALIGN_H = 2;
uint32_t imageWidth = 500;
uint32_t imageHeight = 333;
uint32_t imageSize = imageWidth * imageHeight *3/2;
uint32_t align_width = image_width, align_height = image_height;

// 进行宽高对齐
alignWidth  = (alignWidth % ALIGN_W) ? alignWidth : (imageWidth + ALIGN_W)/ALIGN_W*
ALIGN_W;
alignHeight = (alignHeight % ALIGN_H) ? alignHeight : (imageHeight + ALIGN_H)/ALIGN_H *
ALIGN_H;
uint32_t alignSize = alignWidth * alignHeight *3/2;

// 读取文件数据， 拷贝对齐内存
FILE *fpIn = fopen(filePath.data(), "rb");
Uint8_t* imageBuffer = (uint8_t*) malloc(imageSize);
HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(fileLen, (void*&)alignBuffer, 10000);

size_t size = fread(imageBuffer, 1, imageSize, fpIn);

// 进行拷贝
Uint32_t tempLen = imageWidth * imageHeight;
if (alignWidth == imageWidth)
{
    if(alignHeight > imageHeight)
    {
        memcpy_s(alignBuffer, tempLen, imageBuffer, tempLen);
        memcpy_s(alignBuffer + alignHeight * alignWidth,
            tempLen/2, imageBuffer + tempLen, tempLen/2);
    }
} else {
    for(int32_t n=0;n<imageHeight;n++)
    {
       memcpy_s(alignBuffer+n*alingWidth,imageWidth,
           imageBuffer+n*imageWidth,imageWidth);
    }
    for(int32_t n=0;n<imageHeight/2;n++)
    {
        memcpy_s(alignBuffer+n*alignWidth+alignHeight* alignWidth,
            imageWidth, imageBuffer+n* imageWidth + imageHeight*imageWidth,imageWidth);//UV
     }
}
```

- 方法二：在Device进行图片解码、视频解码以后输出为16*2对齐，可直接作为DVPP VPC(Crop&&Resize)输入。

7.2.4 模型转换预处理配置

从**图1 crop/resize运行示意图**中可以看到，crop/resize输出的图像是经过align_up对齐的，这种对齐会导致部分图像是经过padding的，就不是原始模型需要的输入。为了得到希望输出的图片，可以经过将这部分数据拷贝出来，放在一个新的缓冲区输入到模型推理模块，但这样引入了数据拷贝的开销。为了降低这类开销，框架提供了机制，允许输入到模型管家（modelManager）的图像带padding边，模型管家的AIPP模块会根据用户设定的宽高（在模型转换时设置），对图像进行crop，输出满足模型输入要求的图片，送到模型推理，不需要进行数据拷贝，性能得到了较大提高。

如下，假定模型推理需要输入的图像为224*224，而从DVPP的获取的数据是128*16对齐的，即256*224。

7.2.5 Batch 和超时

对于大部分模型，特别是小模型，一个批量的输入组成一个batch交给芯片做模型的推理可获得性能收益。使用batch推理将大大提高数据的吞吐率，同时也将提高芯片的利用率，在损失一定的时延情况下提升了整体的性能。因此，构建一个高性能应用应当在时延允许的情况下尽可能使用大batch。

框架为了用户能更方便、更灵活的使用batch，引入了超时机制，用户在config文件配置“is_repeat_timeout_flag”设置是否重复做超时处理，配置“wait_inputdata_max_time”设置超时时间。如果设置了超时参数，在超时后，系统调用Process函数时传入的全部是空指针，需要由用户自行编写超时处理的代码逻辑。用户编写代码逻辑时，可以在接收到数据时，将数据存储在队列中，等到足够的数据组成batch后再一并进行推理。这个队列可以使用框架提供的hiai::MultiTypeQueue。为了防止数据“饿死”在队列中，用户使用超时的设置接口，根据应用对时延的要求设置超时时间。超时时间到达时，框架会主动再次调用engine的主要处理流程。此时用户将队列中的数据取出处理，这样，数据就不会“饿死”在队列中了。

如果模型的输入是多Batch且用户分批发送各Batch的数据给模型管家（推理Engine）做推理，则需要用户添加如下代码逻辑：

1. 需要用户在Device侧单独申请缓存空间存放各Batch的数据。
2. 当Device侧的推理Engine接收到各Batch的数据后，用户需要将各Batch的数据拼接起来存放**1**申请的缓存空间中。
3. 等Device侧的推理Engine接收的Batch个数与模型推理需要的Batch个数相等后，用户才可以使用缓存空间中的多Batch数据进行推理。

7.2.6 算法推理输入输出数据处理

为了避免算法推理内部可能出现的内存拷贝，在调用模型管家Process接口时，建议输入数据（输入数据一般可直接使用框架传入的内存，该内存是由框架通过HIAI_DMalloc申请得到）及输出数据都通过HIAI_DMalloc接口申请，这样就能够使能算法推理的零拷贝机制，优化Process时间。如果在推理前需要进行DVPP处理，DVPP的输入内存使用框架传入的内存，输出内存可通过HIAI_DVPP_DMalloc接口分配，并将输出内存传给推理Engine当做输入内存。

7.2.7 回传数据优化处理

当推理计算完成后，需要将推理结果或者推理结束信号发送给Host端，如果在推理Engine内部调用SendData回传数据到Host端，将会消耗推理Engine的时间。建议单独

开一个专门负责回传数据的Engine（例如：DataOptEngine），当推理结束后，推理Engine将处理数据透传给DataOptEngine，由DataOptEngine负责将数据回传给Host侧，再由Host侧的Engine（例如：DstEngine）负责接收传过来的推理结果。

```
//Device侧的DataOptEngine负责将数据回传给Host侧
HIAI_IMPL_ENGINE_PROCESS("DataOptEngine", DataOptEngine, 1)
{
    HIAI_StatusT hiaiRet = HIAI_OK;
    if (arg0 == nullptr) {
        HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "get inference result timeout");
        return HIAI_INVALID_INPUT_MSG;
    }
    hiaiRet = SendData(0, "EngineTransNewT", arg0);
}
// Host侧的DstEngine负责接收从Device侧传过来的推理结果
HIAI_IMPL_ENGINE_PROCESS("DstEngine", DstEngine, 1)
{
    HIAI_StatusT ret = HIAI_OK;
    if (nullptr != arg0) {
        printf("dest engine had receive data already\n");
        std::shared_ptr<std::string> result =
            std::static_pointer_cast<std::string>(arg0);
        ret = SendData(0, "string", result);
        if (HIAI_OK != ret) {
            HIAI_ENGINE_LOG(ret, "DstEngine SendData to recv failed");
            return ret;
        }
    }
    else {
        HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "DestEngine Fail to receive data");
        printf("destengine do not receive data arg0 is null\n");
        return HIAI_INVALID_INPUT_MSG;
    }
    return HIAI_OK;
}
```

7.3 算子使用建议

总体原则

基于Ascend 310芯片的特点，要提升算法的性能，就要尽量提升Cube的使用效率，相应的需减小数据搬移和Vector运算的比例。总体原则有以下几点。

1. 网络结构
   - 推荐使用主流的网络拓扑，包括ResNet、MobileNet，性能已做过调优。
   - 不推荐使用早期的网络拓扑，包括VGG、AlexNet，网络模型偏大，带宽压力大。
   - 矩阵乘法的MKN，尽量取16的倍数。算法上可以考虑适当增加channel个数，而不是分group的方式减少channel数量。
   - 增加数据复用率：一个参数的利用次数越多带宽的瓶颈越小，所以算法上可以考虑增加filter的复用次数，比如增加feature map大小，避免过大的stride或dilation。
2. Conv算子
   - 非量化模式下，Conv的输入和输出通道数建议采用16的整数倍。
   - 量化模式下，Conv的输入和输出通道数建议采用32的整数倍。
   - 量化模式下，多个Conv算子之间，建议少插入Pooling算子。
3. FC（FullConnection）算子

当网络存在FC算子，尽量使用多batch同时推理。

4. Concat算子
   - 非量化模式下，Concat的输入通道建议采用16的整数倍。
   - 量化模式下，Concat的输入通道建议采用32的整数倍。
5. Conv融合算子

   推荐使用Conv+BatchNorm+Scale+Relu/Relu6的组合，性能已做过调优。
6. Norm算子
   - 推荐使用BatchNorm算子，使用预训练的Norm参数。
   - 不推荐使用需要在线计算Norm参数的算子，比如LRN等。
7. 检测算子

   建议使用主流的检测网络拓扑，包括FasterRCNN、SSD，性能已做过调优。

部分算子使用技巧

1. Conv+（BatchNorm+Scale）+Relu性能较Conv+(BatchNorm+Scale)+Tanh等激活算子好；尽量避免过于复杂的激活函数。
2. Concat算子在C维度进行拼接时，输入Tensor的Channel数均为16倍数时，性能较好。
3. FC算子在Batch数为16倍数时，性能较好。
4. 连续卷积结构性能较好，如果卷积层间反复插入较多Vector算子（如Pooling），则性能较差；这点在INT8模型中较明显。
5. 在早期AlexNet、 GoogleNet中使用了LRN作为normalization算子，该算子计算十分复杂，在算法演进过程中也逐渐被替换为BatchNorm等其他算子，在目前ResNet、Inception等主流网络结构中不再使用。针对Ascend310平台，推荐在网络中替换为BatchNorm等算子。

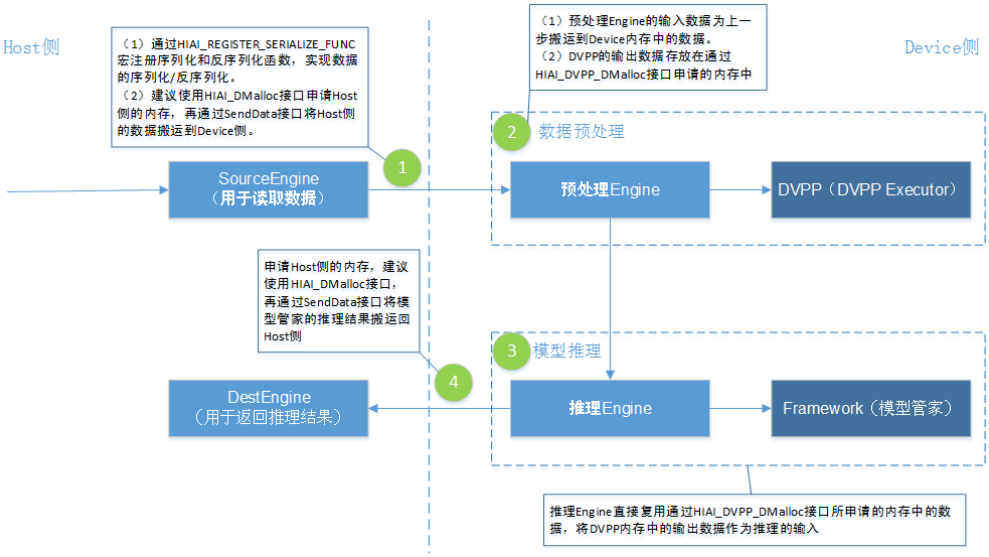
7.4 示例说明

需要同时运行多个模型推理时，不建议使用多Graph，建议单Graph多batch或者单Graph多Engine。

7.4.1 数据流

示例代码完成了通过分类网络（resnet-18）处理图片的功能，数据流如**图7-5**所示。

图 7-5 接口调用流程



7.4.2 ”0“拷贝

示例代码的整个流程展示了“0”拷贝的思想。所谓的“0”拷贝，指用户在整个流程中对图像数据不做任何显式的拷贝动作。为了实现“0”拷贝，需要执行以下操作。

1. 通过HIAI_REGISTER_SERIALIZE_FUNC注册序列化函数（GetSerializeFunc）和反序列化函数（GetDeserializeFunc），实现数据类型的序列化/反序列化。
2. 使用Matrix框架提供的HIAI_Dmalloc接口申请内存，再调用SendData接口，用于将Host侧数据向Device侧搬运。
3. 通过HIAI_DMalloc接口申请的内存，作为图像/视频编解码的输入使用，无需进行数据拷贝。
4. 使用Matrix框架提供的HIAI_DVPP_DMalloc接口申请的内存，内存地址满足DVPP的输入/输出要求，可直接作为图像/视频输出的使用。调用HIAI_DVPP_DMalloc接口申请内存后，必须使用HIAIMemory::HIAI_DVPP_DFree接口释放内存。

   在DVPP内部，VPC模块的输入可直接复用内存中JPEGD模块的输出数据。
5. 通过HIAI_DVPP_DMalloc接口申请的内存，可直接作为模型推理首层的输入，无需进行数据拷贝。

7.5 FAQ

7.5.1 申请自动释放内存时使用智能指针，但析构器不为空，导致异常

问题现象

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_AUTO_FREE（表示自动释放内存），使用智能指针存放申请的内存地址，但不定义析构器（调用默认析构器）或调用其他释放析构器（如：HIAI_DFree，会重复释放），会导致程序异常、内存泄露等情况。

版权所有 © 华为技术有限公司

解决方法

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_AUTO_FREE（表示自动释放内存），使用智能指针存放申请的内存地址，由于Matrix框架自动释放内存，所以指定智能指针的析构器为空即可。

示例代码如下：

```
* @申请自动释放内存，如果使用智能指针，则析构器必须是空的（框架自动释放）
**/
int main()
{
    // 使用DMalloc申请自动释放
    unsigned char* inbuf = nullptr;
    uint32_t bufferLen = 1080*1920;
    // 默认是自动释放方式， flag = hiai::HIAI_MEMORY_ATTR::MEMORY_ATTR_ATUO_FREE
    HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(bufferLen, (void*&)inbuf, 10000);
    // todo getRet 异常判断
    // todo 把需要传输的数据放到 inbuf中，大小为bufferLen。
    //组装发送数据的结构体engineTranData
    EngineTransNewT    engineTranData;
    engineTranData->bufferSize = bufferLen;
    engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
        {
            // 析构器不做任何操作，由框架自动释放内存，如果不定义析构器，则调用默认析构器，会异常
           // 如果调用其他释放析构器，HIAI_DFree，则重复释放，也是异常
        });
    //SandData 发送数据到对端
    HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
// todo sendRet异常判断
}
```

7.5.2 申请手动释放内存时使用智能指针， 但未将析构器指定为HIAI_DFree，导致异常

问题现象

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_MANUAL_FREE（表示手动释放内存），使用智能指针存放申请的内存地址，若指定的智能指针析构器不对，会导致程序异常、内存泄露等情况。

解决方法

通过HIAI_DMalloc或HIAIMemory::HIAI_DMalloc接口申请内存（用于Host到Device或Device到Host的数据传输），将flag参数值设置为MEMORY_ATTR_MANUAL_FREE（表示手动释放内存），使用智能指针存放申请的内存地址，需指定智能指针析构器为HIAI_DFree或HIAIMemory::HIAI_DFree。

示例代码为：

```
/**
* @申请手动释放内存，如果使用智能指针， 需要指定析构器为 HIAI_DFree
**/
int main()
{
    // 使用DMalloc手动释放内存
    unsigned char* inbuf = nullptr;
    uint32_t bufferLen = 1080*1920;
    HIAI_StatusT getRet = hiai::HIAIMemory::HIAI_DMalloc(bufferLen, (void*&)inbuf,
```

```
10000,hiai::HIAI_MEMORY_ATTR::MEMORY_ATTR_MANUAL_FREE);
    // todo getRet 异常判断

    // todo 把需要传输的数据放到 inbuf中，大小为bufferLen。
    //组装发送数据的结构体engineTranData
    EngineTransNewT    engineTranData;
    engineTranData->bufferSize = bufferLen;
    engineTranData->transBuff.reset(std::static_pointer_cast<uint8_t *> (inbuf), [](uint8_t *)(addr )
        {
            // 析构器必须调用HIAI_DFree，否则内存泄漏
          hiai::HIAIMemory::HIAI_DFree(void * addr)
        });
    //SandData 发送数据到对端
    HIAI_StatusT sendRet =
SendData(DEFAULT_PORT,"EngineTransNewT",std::static_pointer_cast<void>(engineTranData));
    // todo sendRet 异常判断
}
```

8 API 参考

8.1 Framework API参考
8.2 Matrix API参考
8.3 DVPP API参考
8.4 GE API参考
8.5 TE API参考

8.1 Framework API 参考

8.1.1 简介

本文介绍编译自定义算子时所依赖的Framework组件的接口（这部分接口是内部接口，用户不需要调用）、自定义算子插件开发涉及的接口，接口主要包括非类成员函数、类成员函数，同时本文还介绍了与接口相关的宏。

关于如何使用这些接口进行自定义算子插件的开发以及代码的编译运行指导，请参见《TE自定义算子开发指导》。

您可以在DDK包的安装目录下的“ddk/include/inc”目录下查看接口的定义文件。如果通过引导安装的方式同时安装Mind Studio和DDK，您可以使用Mind Studio安装用户登录Mind Studio服务器，在“~/tools/che/ddk/ddk/include/inc”路径下查看接口定义文件。每个接口对应的定义文件请参见具体的接口描述。

8.1.2 非类成员函数

8.1.2.1 custom_op_run 函数

函数功能

自定义算子的单算子调测入口函数。该函数在“custom\custom_op.h”中定义。

custom_op_run接口的返回值是ErrorInfo类型，包含错误码、错误描述。当错误码为0时，表示成功；当错误码为其它值时，表示失败。ErrorInfo类型的定义如下：

```
struct ErrorInfo
  {
      uint32_t    error_code;
      std::string error_msg;
  };
```

函数原型

ErrorInfo custom_op_run(const std::string& name, int32_t type, const std::string& bin_file,

const std::vector< std::string >& in_files,

const std::vector< std::string >& out_files,

const std::vector< uint32_t >& out_buf_sizes,

const std::vector< uint32_t >& workspace_sizes = std::vector<uint32_t>(),

const std::string& op_cfg_file = "", void *param = nullptr, int param_len = 0);

参数说明

参数	输入/输出	说明
name	输入	算子kernel名称。
type	输入	算子类型： ● 0 - TE_AICORE ● 1 -TE_AICPU ● 2 - AI CPU
bin_file	输入	算子二进制文件路径，包含文件名。 路径支持配置相对路径，此时二进制文件需存放在程序所在的目录下，如：**./*.bin**。
in_files	输入	输入数据文件路径，包含文件名。 路径支持配置相对路径，此时数据文件需存放在程序所在的目录下。
out_files	输入	输出数据文件路径，包含文件名。 路径支持配置相对路径，此时数据文件需存放在程序所在的目录下。
out_buf_sizes	输入	输出数据大小，单位是字节。
workspace_sizes	输入	workspace大小，单位是字节。
op_cfg_file	输入	用户指定的算子的输入输出描述信息。
param	输入	用户指定的算子参数结构体地址信息。
param_len	输入	用户指定的算子参数结构体的长度，使用sizeof(param)取结构体长度，单位是字节。

8.1.2.2 custom_op_compare 函数

函数功能

自定义算子的单算子校验接口，用于校验数据精度。该函数在“custom \custom_op.h”中定义。

custom_op_compare接口的返回值是ErrorInfo类型，包含错误码、错误描述。当错误码为0时，表示成功；当错误码为其它值时，表示失败。ErrorInfo类型的定义如下：

```
struct ErrorInfo
    {
        uint32_t    error_code;
        std::string error_msg;
    };
```

函数原型

ErrorInfo custom_op_compare(const std::string& expect_file, const std::string& actual_file,

int32_t data_type, float precision_deviation,

float statistical_discrepancy, bool& compare_result);

参数说明

参数	输入/输出	说明
expect_file	输入	期望数据文件，包含文件名
actual_file	输入	实际数据文件，包含文件名
data_type	输入	算子类型，目前支持FP32（0）、FP16（1）
precision_deviation	输入	单数据精度偏差，数值范围（0,1），偏差越小，精度越高。
statistical_discrepancy	输入	整个数据集的统计偏差，数值范围（0,1），偏差越小，精度越高。
compare_result	输出	比较结果， true：对比成功；false：对比失败

8.1.2.3 GetContext 函数

函数功能

获取OMG上下文，并返回。该函数在“framework\omg\omg_types.h”中定义。

GetContext接口的返回值是OmgContext类型，OmgContext类型的定义请参见“framework\omg\omg_types.h”。

函数原型

OmgContext& GetContext();

参数说明

无。

8.1.2.4 AutoMappingFn 函数

函数功能

自动映射回调函数。该函数在“framework\omg\register.h”中定义。

函数原型

Status AutoMappingFn(const google::protobuf::Message* op_src, ge::Operator& op);

参数说明

参数	输入/输出	说明
op_src	输入	原始转换前算子
op	输入	映射的算子

8.1.2.5 aicpu_run_func 函数

函数功能

AI CPU自定义算子的函数指针，函数名称可以自定义。该函数在“cce\customize.h”中定义。

函数原型

void (*aicpu_run_func)(opTensor_t **, void **, int32_t,

opTensor_t **, void **, int32_t, void *, rtStream_t);

参数说明

参数	输入/输出	说明

opTensor_t **	输入	输入描述符Tensor。
		typedef struct tagOpTensor
		{
		// real dim info
		opTensorFormat_t format;
		//char align1[4];
		opDataType_t data_type;
		//char align2[4];
		int32_t dim_cnt;
		int32_t mm; //lint !e148
		int32_t dim[CC_DEVICE_DIM_MAX]; //lint !e148
		} opTensor_t;

void **	输入	输入描述符Tensor的地址
int32_t	输入	输入描述符个数
opTensor_t **	输入	输出描述符Tensor
void **	输入	输出描述符Tensor的地址
int32_t	输入	输出描述符个数
void *	输入	属性handle地址
rtStream_t	输入	stream 标识

8.1.3 类成员函数

8.1.3.1 StatusFactory 类

该类函数在该函数在“custom\common\fmk_error_codes.h”中定义。

8.1.3.1.1 Instance 函数

函数功能

返回Status工厂类的实例对象。

函数原型

static StatusFactory* Instance();

参数说明

无

8.1.3.1.2 RegisterErrorNo 函数

函数功能

注册错误码。

函数原型

void RegisterErrorNo(uint32_t err, const std::string& desc);

参数说明

参数	输入/输出	说明
err	输入	错误码
desc	输入	错误码描述字符串

8.1.3.1.3 GetErrDesc 函数

函数功能

获取错误码描述。

函数原型

std::string GetErrDesc(uint32_t err);

参数说明

参数	输入/输出	说明
err	输入	错误码

8.1.3.1.4 StatusFactory 函数

函数功能

构造函数。

函数原型

StatusFactory();

参数说明

无

8.1.3.1.5 ~StatusFactory 函数

函数功能

析构函数。

函数原型

~StatusFactory();

参数说明

无

8.1.3.2 ErrorNoRegisterar 类

该类函数在该函数在“custom\common\fmk_error_codes.h”中定义。

8.1.3.2.1 ErrorNoRegisterar 函数

函数功能

构造函数。

函数原型

ErrorNoRegisterar(uint32_t err, const std::string& desc);

参数说明

参数	输入/输出	说明
err	输入	错误码
desc	输入	错误码描述字符串

8.1.3.2.2 ~ErrorNoRegisterar 函数

函数功能

析构函数。

函数原型

~ErrorNoRegisterar();

参数说明

无。

8.1.3.3 OpTypeContainer 类

该类函数在该函数在“custom\common\op_types.h”中定义。

8.1.3.3.1 Instance 函数

函数功能

获取OpTypeContainer工厂类的实例对象。

函数原型

```
static OpTypeContainer * Instance();
```

参数说明

无。

8.1.3.3.2 Register 函数

函数功能

将算子的类型保存到一个自定义的集合中。

函数原型

```
void Register(const std::string& op_type);
```

参数说明

参数	输入/输出	说明
op_type	输入	需要保存的算子类型

8.1.3.3.3 IsExisting 函数

函数功能

判断一个算子类型是否存在。

函数原型

```
bool IsExisting(const std::string& op_type);
```

参数说明

参数	输入/输出	说明
op_type	输入	需要查询的算子类型

8.1.3.3.4 OpTypeContainer 函数

函数功能

构造函数。

函数原型

OpTypeContainer();

参数说明

无。

8.1.3.4 OpTypeRegistrar 类

该类函数在该函数在“custom\common\op_types.h”中定义。

8.1.3.4.1 OpTypeRegistrar 函数

函数功能

构造函数，将一个算子的类型注册到Container中。

函数原型

OpTypeRegistrar(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	需要注册的算子类型

8.1.3.4.2 ~OpTypeRegistrar 函数

函数功能

析构函数。

函数原型

~OpTypeRegistrar();

参数说明

无

8.1.3.5 OpRegistrationData 类

该类函数在该函数在“framework\omg\register.h”中定义。

8.1.3.5.1 OpRegistrationData 函数

函数功能

构造函数。

函数原型

OpRegistrationData(const std::string& om_optype);

参数说明

参数	输入/输出	说明
om_optype	输入	算子类型

8.1.3.5.2 ~OpRegistrationData

函数功能

析构函数。

函数原型

~OpRegistrationData();

参数说明

无。

8.1.3.5.3 FrameworkType 函数

函数功能

设置框架类型。

函数原型

OpRegistrationData& FrameworkType(domi::FrameworkType fmk_type);

参数说明

参数	输入/输出	说明
fmk_type	输入	框架类型 0：caffe 3：tensorflow

8.1.3.5.4 OriginOpType 函数

函数功能

设置原始模型的算子类型或算子类型列表。

函数原型

- 设置算子类型列表：

  OpRegistrationData& OriginOpType (const std::initializer_list<std::string> & ori_optype_list);
- 设置算子类型：

  OpRegistrationData& OriginOpType (const std::string& ori_optype);

参数说明

参数	输入/输出	说明
ori_optype_list	输入	原始模型算子类型列表
ori_optype	输入	原始模型算子类型

8.1.3.5.5 ParseParamsFn 函数

函数功能

参数解析函数。

函数原型

OpRegistrationData& ParseParamsFn(ParseParamFunc parseParamFn);

参数说明

参数	输入/输出	说明
parseParamFn	输入	回调函数ParseParamFunc，请参见**回调函数ParseParamFunc**。

回调函数 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。

函数原型

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

参数说明

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AI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”。

8.1.3.5.6 InferShapeAndTypeFn 函数

函数功能

维度推导函数。

函数原型

OpRegistrationData& InferShapeAndTypeFn(InferShapeFunc inferShapeFn);

参数说明

参数	输入/输出	说明
inferShapeFn	输入	回调函数InferShapeFunc，请参见**回调函数InferShapeFunc**。

回调函数 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。

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI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”。

8.1.3.5.7 UpdateOpDescFn 函数

函数功能

算子描述更新函数。

函数原型

OpRegistrationData& UpdateOpDescFn(UpdateOpDescFunc updateOpDescFn);

参数说明

参数	输入/输出	说明
updateOpDescFn	输入	回调函数UpdateOpDescFunc，参见**回调函数UpdateOpDescFunc**。

回调函数 UpdateOpDescFunc

用户自定义并实现UpdateOpDescFunc类函数，用于对算子信息数据进行更新。

函数原型

static Status UpdateOpDescFunc(ge::Operator& op);

参数说明

参数	输入/输出	说明
op	输入/输出	适配昇腾AI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”。

8.1.3.5.8 GetWorkspaceSizeFn 函数

函数功能

获取work 空间大小。

函数原型

OpRegistrationData& GetWorkspaceSizeFn(GetWorkspaceSizeFunc getWorkspaceSizeFn);

参数说明

参数	输入/输出	说明
getWorkspaceSizeFn	输入	回调函数GetWorkspaceSizeFunc，参见**回调函数GetWorkspaceSizeFunc**。

回调函数 GetWorkspaceSizeFunc

用户自定义并实现GetWorkspaceSizeFunc类函数，完成算子对应的工作空间大小计算，并输出。

函数原型

Status GetWorkspaceSizeFn(domi::Status (const ge::Operator& op, std::vector<int64_t>& size));

参数说明

参数	输入/输出	说明
op	输入	需要计算工作空间的算子对象。 关于Operator类，请参见《GE API参考》中的“Operator类接口”。
size	输出	工作空间大小，该值需要用户计算并提供。

8.1.3.5.9 TEBinBuildFn 函数

函数功能

build回调函数。

函数原型

OpRegistrationData& TEBinBuildFn(BuildTeBinFunc buildTeBinFn);

参数说明

参数	输入/输出	说明
buildTeBinFn	输入	回调函数BuildTeBinFunc，参见**回调函数BuildTeBinFunc**。

回调函数 BuildTeBinFunc

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

函数原型

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

参数说明

参数	输入/输出	说明
op	输入	适配昇腾AI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;
		};

版权所有 © 华为技术有限公司

8.1.3.5.10 ImplyType 函数

函数功能

设置算子执行方式。

函数原型

OpRegistrationData& ImplyType(domi::ImplyType imply_type);

参数说明

参数	输入/输出	说明

imply_type	输入	算子执行方式。
		enum class ImplyType : unsigned int
		{
		BUILDIN = 0,// 内置算子，由OME正常执行
		TVM, // 编译成tvm bin文件执行
		CUSTOM, // 由用户自定义计算逻辑，通过CPU执行
		AI_CPU, // AICPU 自定义算子类型
		INVALID = 0xFFFFFFFF,
		};

8.1.3.5.11 Formats 函数

函数功能

设置算子支持的输入/输出数据格式。

函数原型

```
OpRegistrationData& Formats(
const std::initializer_list<domi::tagDomiTensorFormat>& input_formats,
const std::initializer_list<domi::tagDomiTensorFormat>& output_formats);
OpRegistrationData& Formats(
const domi::tagDomiTensorFormat& input_format,
const domi::tagDomiTensorFormat& output_format);
```

参数说明

参数	输入/输出	说明
input_format	输入	输入支持的数据格式
output_format	输入	输出支持的数据格式

8.1.3.5.12 WeightFormats 函数

函数功能

设置算子权重支持的数据格式。

函数原型

```
OpRegistrationData& WeightFormats(
const std::initializer_list<domi::tagDomiTensorFormat>& weight_formats);
```

参数说明

参数	输入/输出	说明
weight_formats	输入	权重支持的数据格式列表： typedef enum tagDomiTensorFormat { DOMI_TENSOR_NCHW = 0, /**< NCHW */ DOMI_TENSOR_NHWC, /**< NHWC */ DOMI_TENSOR_ND, /**< Nd Tensor */ DOMI_TENSOR_NC1HWC0, /**< NC1HWC0 */ DOMI_TENSOR_FRACTAL_Z, /**< FRACTAL_Z */ DOMI_TENSOR_NC1C0HWPAD, DOMI_TENSOR_NHWC1C0, DOMI_TENSOR_FSR_NCHW, DOMI_TENSOR_FRACTAL_DECONV, DOMI_TENSOR_BN_WEIGHT, DOMI_TENSOR_CHWN, /*Android NN Depth CONV*/ DOMI_TENSOR_FILTER_HWCK, /* filter input tensor format */ DOMI_TENSOR_RESERVED } domiTensorFormat_t;

8.1.3.5.13 Finalize 函数

函数功能

创建所有支持的算子的parser和builder函数，成功则返回true，否则返回false。

函数原型

bool Finalize();

参数说明

无。

8.1.3.6 OpRegistry 类

该类函数在该函数在“framework\omg\register.h”中定义。

8.1.3.6.1 Instance 函数

函数功能

获取OpRegistry类对象的实例。

函数原型

static OpRegistry* Instance();

参数说明

无。

8.1.3.6.2 Register 函数

函数功能

注册一个OpRegistrationData，成功则返回true，否则返回false。

函数原型

bool Register(const OpRegistrationData& reg_data);

参数说明

参数	输入/输出	说明
reg_data	输入	算子注册信息

8.1.3.6.3 GetImplyType 函数

函数功能

获取算子执行类型。

函数原型

domi::ImplyType GetImplyType(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	需要查询的算子类型

8.1.3.6.4 GetOpTypeByImplyType 函数

函数功能

查询指定执行类型的所有算子的名称。

函数原型

```
void GetOpTypeByImplyType(std::vector<std::string>& vec_op_type,const
domi::ImplyType& imply_type);
```

参数说明

参数	输入/输出	说明
vec_op_type	输出	保存查询的算子名称信息
imply_type	输入	算子的执行类型

8.1.3.6.5 GetFormats 函数

函数功能

查询op_type这个算子支持的输入输出数据类型。

函数原型

```
void GetFormats(const std::string& op_type,
std::vector<domi::tagDomiTensorFormat>&
input_format_vector,std::vector<domi::tagDomiTensorFormat>&
output_format_vector);
```

参数说明

参数	输入/输出	说明
op_type	输入	算子类型名称
input_format_vector	输出	支持的输入数据类型列表
output_format_vector	输出	支持的输出数据类型列表

8.1.3.6.6 GetWeightFormats 函数

函数功能

查询op_type这个算子支持的权重数据类型。

函数原型

void GetWeightFormats(const std::string& op_type,

std::vector<domi::tagDomiTensorFormat>& format_vector);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型名称
format_vector	输出	支持的权重数据类型列表

8.1.3.6.7 GetParseParamFunc 函数

函数功能

获取参数解析回调函数。

函数原型

domi::ParseParamFunc GetParseParamFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.6.8 GetInferShapeFunc 函数

函数功能

获取推导维度回调函数。

函数原型

domi::InferShapeFunc GetInferShapeFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.6.9 GetGetWorkspaceSizeFunc 函数

函数功能

获取workspace大小回调函数。

函数原型

domi::GetWorkspaceSizeFunc GetGetWorkspaceSizeFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.6.10 GetUpdateOpDescFunc 函数

函数功能

获取更新算子描述回调函数。

函数原型

domi::UpdateOpDescFunc GetUpdateOpDescFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.6.11 GetBuildTeBinFunc 函数

函数功能

获取build回调函数。

函数原型

domi::BuildTeBinFunc GetBuildTeBinFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.6.12 GetTransWeightFunc 函数

函数功能

获取转换权值回调函数。

函数原型

domi::TransWeightFunc GetTransWeightFunc(const std::string& op_type);

参数说明

参数	输入/输出	说明
op_type	输入	算子类型

8.1.3.7 OpReceiver 类

该类函数在该函数在“framework\omg\register.h”中定义。

8.1.3.7.1 OpReceiver 函数

函数功能

构造函数。

函数原型

OpReceiver(OpRegistrationData& reg_data);

参数说明

参数	输入/输出	说明
reg_data	输入	需要注册的算子信息

8.1.3.7.2 ~OpReceiver 函数

函数功能

析构函数。

函数原型

~OpReceiver();

参数说明

无。

8.1.4 宏定义

8.1.4.1 错误码生成宏

宏功能

调用DEF_ERRORNO生成指定模块的错误码。

宏原型

DEF_ERRORNO(sysid, modid, name, value, desc)

参数说明

参数	输入/输出	说明
sysid	输入	系统id
modid	输入	模块id
name	输入	错误码名称
value	输入	错误码的实际值
desc	输入	错误码描述字符串

8.1.4.2 COMMON 模块错误码生成宏

宏功能

调用DEF_ERRORNO_COMMON生成common模块的错误码。

宏原型

DEF_ERRORNO_COMMON(name, value, desc)

参数说明

参数	输入/输出	说明
name	输入	错误码名称
value	输入	错误码的实际值
desc	输入	错误码描述字符串

8.1.4.3 OMG 模块错误码生成宏

宏功能

调用DEF_ERRORNO_OMG生成OMG模块的错误码。

宏原型

DEF_ERRORNO_OMG(name, value,desc)

参数说明

参数	输入/输出	说明
name	输入	错误码名称
value	输入	错误码的实际值
desc	输入	错误码描述字符串

8.1.4.4 OME 模块错误码生成宏

宏功能

调用DEF_ERRORNO_OME生成OME模块的错误码。

宏原型

DEF_ERRORNO_OME(name, value,desc)

参数说明

参数	输入/输出	说明
name	输入	错误码名称
value	输入	错误码的实际值
desc	输入	错误码描述字符串

8.1.4.5 CALIBRATION 模块错误码生成宏

宏功能

调用DEF_ERRORNO_CALIBRATION生成量化模块的错误码。

宏原型

DEF_ERRORNO_CALIBRATION(name, value, desc)

参数说明

参数	输入/输出	说明
name	输入	错误码名称
value	输入	错误码的实际值
desc	输入	错误码描述字符串

8.1.4.6 获取错误码描述宏

宏功能

调用GET_ERRORNO_STR获取错误码描述。

宏原型

GET_ERRORNO_STR(value)

参数说明

参数	输入/输出	说明
value	输入	错误码的值

8.1.4.7 算子类型注册宏

宏功能

对一种算子类型进行注册，主要是声明一个常量字符串,该字符串就是算子类型的名称，然后将其保存到一个set中。

宏原型

REGISTER_OPTYPE_DECLARE(var_name, str_name)

REGISTER_OPTYPE_DEFINE(var_name, str_name)

参数说明

参数	输入/输出	说明
var_name	输入	常量字符串名称
str_name	输入	算子类型名称

8.1.4.8 算子类型是否存在宏

宏功能

检查一个算子类型是否存在。

宏原型

IS_OPTYPE_EXISTING(str_name)

参数说明

参数	输入/输出	说明
str_name	输入	算子类型名称

8.1.4.9 算子注册宏

8.1.4.9.1 REGISTER_CUSTOM_OP 宏

宏功能

按指定名称注册算子。

宏原型

REGISTER_CUSTOM_OP(name)

参数说明

参数	输入/输出	说明
name	输入	算子名称

8.1.4.10 对外接口标记宏

8.1.4.10.1 FMK_FUNC_HOST_VISIBILITY

宏功能

标记Host侧对外暴露接口，使用该宏标记接口在动态链接库中可见并可被调用。

宏原型

FMK_FUNC_HOST_VISIBILITY

参数说明

无

8.1.4.10.2 FMK_FUNC_DEV_VISIBILITY

宏功能

标记Device侧对外暴露接口，使用该宏标记接口在动态链接库中可见并可被调用。

宏原型

FMK_FUNC_DEV_VISIBILITY

参数说明

无

8.2 Matrix API 参考

8.2.1 简介

Matrix运行于操作系统之上，业务应用之下。屏蔽操作系统差异，为应用提供统一的标准化接口。Matrix具有多节点调度能力和多进程管理，可以根据配置文件完成业务流程的建立和运行，以及相关的统计信息汇总等。

Matrix总体逻辑包含3个主要的部分，Matrix Agent（运行在Host侧）、Matrix Daemon（运行在Device侧）和Matrix Service（运行在Device侧）。

说明

- Host指与Device相连接的X86服务器、ARM服务器或者WindowsPC，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侧链接，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等开源框架模型转换成Mind Studio支持的模型。
- CCE（cube-based computing engine）加速库通过API的方式，为上层应用（为机器学习提供的各种framework或者Application）提供加速。
- Runtime运行于APP进程空间，为APP提供了Ascend 310芯片的Memory管理、Device管理、Stream管理、Event管理、Kernel执行等功能。

Matrix为通用业务流程执行引擎，主要包含：

- 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进程。
- Matrix Service：运行在Device侧，其功能如下。
  - Engine调用DVPP的API接口实现媒体预处理。
  - Engine调用模型管家（AIModelManger）的API接口实现模型推理。

您可以在DDK包的安装目录下的“ddk/include/inc/hiaiengine”目录下查看接口的定义文件。如果通过引导安装的方式同时安装Mind Studio和DDK，您可以使用Mind Studio安装用户登录Mind Studio服务器，在“~/tools/che/ddk/ddk/include/inc/hiaiengine”路径下查看接口定义文件。每个接口对应的定义文件请参见具体的接口描述。

8.2.2 流程编排接口

8.2.2.1 流程串接接口（C 语言）

8.2.2.1.1 HIAI_Init（与 C++共用）

初始化Matrix的HDC（Host Device Communication）通讯模块。如果Graph配置文件中已经指定运行Graph的设备编号，则以配置文件为准，否则用此处设置的deviceID为准。该接口在init.h中定义。

函数格式

HIAI_StatusT HIAI_Init(uint32_t deviceID)

说明

HIAI_StatusT结构体的定义请参见**8.2.7.2.2 Matrix自定义数据类型**。

参数说明

参数	说明	取值范围
deviceID	设备编号，默认值为0。	0~63

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_INIT_FAIL	hiai init fail

8.2.2.1.2 HIAI_CreateGraph

创建并启动整个Graph，该接口支持单个配置文件中的graph之间的串联，不支持多个配置文件之间的graph串联。只允许在Host侧或Ascend 310 RC上被调用。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_CreateGraph(const char* configFile, size_t len)

参数说明

参数	说明	取值范围
configFile	配置文件路径。 请确保传入的文件路径是正确路径。	单个配置文件中最大支持2048个graph。但受系统资源限制，根据硬件配置不同，以及graph大小不同，实际最多可能只支持几个或十几个graph。
len	配置文件名的长度。	1~255

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_PROTO_FILE_PARSE_FAILED	ParseFromString return failed

8.2.2.1.3 HIAI_DestroyGraph

销毁指定的Graph，只允许在Host侧或Ascend 310 RC上被调用。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_DestroyGraph (uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	指定的Graph ID。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIMEOUT	destroy timeout
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	engine destroy failed

8.2.2.2 流程串接接口（C++语言）

8.2.2.2.1 Graph::GetInstance

获取Graph实例，在创建Graph成功后可以正常使用该接口。该接口在graph.h中定义。

函数格式

static std::shared_ptr<Graph> Graph::GetInstance(uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	目标graph的id。	配置文件里配置的Graph编号。

返回值

Graph的智能指针。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_PROTO_FILE_PARSE_FAILED	ParseFromString return failed

8.2.2.2.2 Graph::CreateGraph（根据配置文件创建 Graph）

创建并启动整个Graph，该接口支持单个配置文件中的graph之间的串联，不支持多个配置文件之间的graph串联。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::CreateGraph(const std::string& configFile)

参数说明

参数	说明	取值范围
configFile	整个Graph的配置文件路径。 请确保传入的文件路径是正确路径。	单个配置文件中最大支持2048个graph。但受系统资源限制，根据硬件配置不同，以及graph大小不同，实际最多可能只支持几个或十几个graph。

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE_NULL	get instance null

8.2.2.2.3 Graph::CreateGraph（根据配置文件创建 Graph，并将生成的 Graph 回写到 list 中）

重载8.2.2.2.2 Graph::CreateGraph（根据配置文件创建Graph），Matrix根据配置文件创建Graph，并将生成的Graph回写到list中，供用户使用。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::CreateGraph(const std::string& configFile, std::list<std::shared_ptr<Graph>>& graphList)

参数说明

参数	说明	取值范围
configFile	整个Graph的配置文件路径。 请确保传入的文件路径是正确路径。	单个配置文件中最大支持2048个graph。但受系统资源限制，根据硬件配置不同，以及graph大小不同，实际最多可能只支持几个或十几个graph。
graphList	用户定义的lsit，Matrix将生成的Graph回写到list，以供用户使用。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE_NULL	get instance null

8.2.2.2.4 Graph::CreateGraph（根据 Protobuf 数据格式创建 Graph）

创建并启动整个Graph，该接口支持根据Protobuf数据格式创建Graph。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::CreateGraph(const GraphConfigList& graphConfig)

参数说明

参数	说明	取值范围
graphConfig	Protobuf数据格式。 用户在编写代码时，可以选择以下定义**graphConfig**参数的方式： ● 定义一个**GraphConfigList**类型的参数，作为Graph::CreateGraph(const GraphConfigList& graphConfig)接口的入参graphConfig。 ● 调用**8.2.5.1.12 解析Matrix配置文件**的接口后，获取graphConfigList参数值，作为Graph::CreateGraph(const GraphConfigList& graphConfig)接口的入参graphConfig。	graph中的Engine数量必须小于等于512。

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_GET_INSTANCE_NULL	get instance null

8.2.2.2.5 Graph::DestroyGraph

销毁整个Graph。该接口在graph.h中定义。

只允许在Host侧或Ascend 310 RC上被调用。

函数格式

static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)

参数说明

参数	说明	取值范围
graphID	需要销毁的graph的id	配置文件里配置的Graph编号

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_DESTROY_TIMEOUT	destroy graph timeout
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	destroy graph failed

8.2.2.2.6 Graph::UpdateEngineConfig

通过HiAI接口更新指定Engine的参数。该接口在graph.h中定义。

使用场景：

通过该接口，更新流程运行过程中指定的运行参数。

例如：视频抓拍图场景，需要根据白天黑夜的变化更新运行流程，则需要通过UpdataEngineConfig函数将白天/黑夜的数值更新入Engine。

函数格式

static HIAI_StatusT Graph::UpdateEngineConfig(const uint32_t& graphId, const uint32_t& engineId, const AIConfig& aiConfig, const bool& syncFlag = false)

参数说明

参数	说明	取值范围
graphId	Graph ID	-
engineId	engine ID	-
aiConfig	配置参数	-
syncFlag	同步/异步执行，同步则等待返回结果	true/false

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARGER	failed to malloc buffer due to the size larger than 256M
5	HIAI_MEMORY_POOL_UPDATE_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL_INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

8.2.2.3 数据传输接口（C 语言）

8.2.2.3.1 HIAI_SetDataRecvFunctor

设置Graph中某Engine接收消息的回调函数。该接口在c_graph.h中定义。

说明

回调函数类型：**typedef HIAI_StatusT (*HIAI_RecvDataCallbackT)(void*)**

函数格式

HIAI_StatusT HIAI_SetDataRecvFunctor(HIAI_PortID_t targetPortConfig, HIAI_RecvDataCallbackT recvCallback)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID 和Port ID。	-
recvCallback	设置用户自定义的结果接收方式。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

8.2.2.3.2 HIAI_C_SendData

用户从外部向Matrix发送数据（或消息）。该接口在c_graph.h中定义。

说明

- 用户自定义的消息类型需要调用8.2.5.2 数据类型序列化和反序列化（C++语言）先注册。
- 该消息指针dataPtr控制权为调用者所有，Matrix不管理该指针，由调用者负责释放。
- 在跨侧（Host->Device或Device->Host）传输数据时，该接口采用DMA（Direct Memory Access）传送方式，可能会影响CPU对中断请求的及时响应与处理，例如影响调用new或者malloc分配内存时的性能。

函数格式

HIAI_StatusT HIAI_C_SendData(HIAI_PortID_t targetPortConfig, const char* messageName, size_t len, void* dataPtr)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID和Port ID。	-

参数	说明	取值范围
messageName	当前发送的消息名。	-
len	消息名称的长度。	-
dataPtr	指向具体的消息指针（该消息指针控制权为调用者所有，Matrix不管理该指针，由调用者负责释放）。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

8.2.2.4 数据传输接口（C++语言）

8.2.2.4.1 Graph::SetDataRecvFunctor

设置Graph中某Engine接收消息的回调函数。调用该接口的代码的运行环境必须与Engine一致，例如，Engine在Device侧运行，那么调用该函数的代码也必须运行在Device。该接口在graph.h中定义。

该接口需要与**DataRecvInterface::RecvData**接口配合使用，一般用于将推理结果返回给用户，详细原理如下：

1. 定义一个DataRecvInterface类的子类（例如DdkDataRecvInterface），初始化一个子类的对象。
2. 调用Graph::SetDataRecvFunctor接口将DdkDataRecvInterface类的对象设置成回调函数。
3. 调用Engine::SendData函数将对应的输入数据发送到对应的输出端口。
4. 调用DdkDataRecvInterface类的RecvData函数返回推理结果。

函数格式

static HIAI_StatusT Graph::SetDataRecvFunctor(const EnginePortID& targetPortConfig, const std::shared_ptr<DataRecvInterface>& dataRecv);

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID 和Port ID。	-
dataRecv	用户自定义的数据接收回调函数。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

8.2.2.4.2 Graph::SendData

用户从外部向Matrix发送void类型的数据到指定的端口。该接口在graph.h中定义。

说明

在跨侧（Host->Device或Device->Host）传输数据时，该接口采用DMA（Direct Memory Access）传送方式，可能会影响CPU对中断请求的及时响应与处理，例如影响调用new或者malloc分配内存时的性能。

函数格式

HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const std::string& messageName, const std::shared_ptr<void>& dataPtr， const uint32_t timeOut = 500)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID 和Port ID。	-
messageName	消息的名称。	-

参数	说明	取值范围
dataPtr	消息的指针。	-
timeOut	调用该接口发送数据时的一次超时时间，不指定timeout参数时，默认的一次超时时间为500ms。 若发送数据超时，系统后台会再次尝试发送数据，最大尝试16次。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

8.2.2.4.3 HIAI_SendData

用户从外部向Matrix发送void类型的数据到指定的端口。该接口在c_graph.h中定义。

说明

在跨侧（Host->Device或Device->Host）传输数据时，该接口采用DMA（Direct Memory Access）传送方式，可能会影响CPU对中断请求的及时响应与处理，例如影响调用new或者malloc分配内存时的性能。

函数格式

HIAI_StatusT HIAI_SendData (HIAI_PortID_t targetPortConfig, const std::string& messageName, std::shared_ptr<void>& dataPtr, uint32_t timeOut=0)

参数说明

参数	说明	取值范围
targetPortConfig	数据接收方的Graph ID、Engine ID和Port ID。	-
messageName	消息的名称。	-
dataPtr	消息的指针。	-
timeOut	发送数据时用户可以添加time_out作为时延，单位为毫秒(ms)。如果出现发送队列满或者内存池不足的情况，根据时延作为阻塞。默认为0，即不阻塞。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
3	HIAI_GRAPH_NOT_EXIST	graph not exist
4	HIAI_GRAPH_NO_USEFUL_MEMORY	no useful memory
5	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
6	HIAI_GRAPH_INVALID_VALUE	graph invalid value

8.2.2.5 接口使用示例

8.2.2.5.1 C 语言接口使用示例

```
/**
* @file graph_c_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
* @version 1.0
*
```

```
*/
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include "hiaiengine/c_api.h"
#include "user_def_data_type.h"

/*************************************************************************
* Normally the mandatory steps to setup the whole HIAI graph include:
*  1) call HIAI_Init() API to perform global system initialization;
*  2) create HIAI_CreateGraph to create and start graph
*  3) set profile config if necessary
*  4) call HIAI_DestoryGraph to destroy the graph finally
*
*************************************************************************/


// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)

static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000;  // 定义在graph.prototxt
static uint32_t dest_engine_id = 1002;  // 定义在graph.prototxt
static const char graph_config_proto_file[] = "llt/hiaiengine/st/examples/config/graph.prototxt";
static const char graph_config_len = sizeof(graph_config_proto_file);

// Define End

// 用户自定义数据接收（recv_data 由框架自动释放）
HIAI_StatusT UserDefDataRecvCallBack(void* recv_data)
{
    // 处理接收到的数据
    return HIAI_OK;
}


/*************************************************************************
* HIAIEngine Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
*************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph()
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = HIAI_CreateGraph(graph_config_proto_file, graph_config_len);
    if (status != HIAI_OK)
    {
        return status;
    }

    // Step3: 设置回调函数用于接收数据
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = dest_engine_id;
    engine_id.port_id = 0;
    HIAI_SetDataRecvFunctor(engine_id, UserDefDataRecvCallBack);

    return HIAI_OK;
}


// User Application Main Example
int main()
```

```
{
    HIAI_StatusT ret = HIAI_OK;

    // 创建流程
    ret = HIAI_InitAndStartGraph();

    if (HIAI_OK != ret)
    {
        return -1;
    }

    // 读取数据，往流程灌输数据
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    int is_end_of_data = 0;
    while (!is_end_of_data)
    {
        UserDefDataTypeT* user_def_msg = (UserDefDataTypeT*)malloc(sizeof(UserDefDataTypeT));

        // 读取数据并填充 UserDefDataType，此数据类型注册见数据类型注册章节

        const char * message_name = "UserDefDataType";
        size_t msg_len = strlen(message_name);
        if (HIAI_OK != HIAI_C_SendData(engine_id, message_name, msg_len, user_def_msg))
        {
            // 队列满时返回发送数据失败，由业务逻辑处理是否重发或丢弃
            break;
        }

        free(user_def_msg);
        user_def_msg = nullptr;
    }

    // 处理结束时，删除整个Graph
    HIAI_DestroyGraph(g_graph_id);
    return 0;
}
```

8.2.2.5.2 C++接口使用示例（根据配置文件创建 Graph）

```
/**
* @file graph_cplusplus_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
* @version 1.0
*
*/
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>

#include "hiaiengine/api.h"
#include "user_def_data_type.h"

/*****************************************************************************
* Normally the mandatory steps to setup the whole HIAI graph include:
*  1) call HIAI_Init() API to perform global system initialization;
*  2) create HIAI_CreateGraph or Graph::CreateGraph to create and start graph
*  3) set profile config if necessary
*  4) call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally
*
*****************************************************************************/
```

```
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)

static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000;  // 定义在graph.prototxt
static uint32_t dest_engine_id = 1002;  // 定义在graph.prototxt
static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End

namespace hiai {

class GraphDataRecvInterface: public DataRecvInterface
{
public:
    GraphDataRecvInterface()
    {
    }

    /**
    * @ingroup hiaiengine
    * @brief 读取数据，保存
    * @param [in]输入数据
    * @return HIAI Status
    */
    HIAI_StatusT RecvData(const std::shared_ptr<void>& message)
    {
        // 转换成具体的消息类型，并处理相关消息， 例如：
        // shared_ptr<std::string> data =
        // std::static_pointer_cast<std::string>(message);
        return HIAI_OK;
    }
private:
};
}

/**************************************************************************
* HIAIEngine Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
**************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph()
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = hiai::Graph::CreateGraph(graph_config_proto_file);
    if (status != HIAI_OK)
    {
        HIAI_ENGINE_LOG(status, "Fail to start graph");
        return status;
    }

    // Step3: 设置回调函数用于接收数据
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return status;
    }

    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = g_graph_id;
    target_port_config.engine_id = dest_engine_id;
```

```
    target_port_config.port_id = 0;

    graph->SetDataRecvFunctor(target_port_config,
        std::shared_ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));

    return HIAI_OK;
}


// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;

    // 创建流程
    ret = HIAI_InitAndStartGraph();

    if(HIAI_OK != ret)
    {
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    }

    // 读取数据，往流程灌输数据
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return -1;
    }

    hiai::EnginePortID engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    bool is_end_of_data = false;
    while (!is_end_of_data)
    {
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);

        // 读取数据并填充 UserDefDataType，此数据类型注册见数据类型注册章节
        // 往Graph灌输数据
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            std::static_pointer_cast<void>(user_def_msg)))
        {
            // 队列满时返回发送数据失败，由业务逻辑处理是否重发或丢弃
            HIAI_ENGINE_LOG("Fail to send data to the graph-%u", g_graph_id);
            break;
        }
    }

    // 处理结束时，删除整个Graph
    hiai::Graph::DestroyGraph(g_graph_id);
    return 0;
}
```

8.2.2.5.3 C++接口使用示例（根据配置文件创建 Graph，并将生成的 Graph 回写到 list 中）

```
/**
* @file graph_cplusplus_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
```

```
* @version 1.0
*
*/
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>
#include "hiaiengine/api.h"
#include "user_def_data_type.h"
/*****************************************************************************
* Normally the mandatory steps to setup the whole HIAI graph include:
*  1) call HIAI_Init() API to perform global system initialization;
*  2) create HIAI_CreateGraph or Graph::CreateGraph to create and start graph
*  3) set profile config if necessary
*  4) call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally
*
*****************************************************************************/
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)
static uint32_t src_engine_id = 1000;  // 定义在graph.prototxt
static uint32_t dest_engine_id = 1002;  // 定义在graph.prototxt
static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End
namespace hiai {
class GraphDataRecvInterface: public DataRecvInterface
{
public:
    GraphDataRecvInterface()
    {
    }
    /**
    * @ingroup hiaiengine
    * @brief 读取数据，保存
    * @param [in]输入数据
    * @return HIAI Status
    */
    HIAI_StatusT RecvData(const std::shared_ptr<void>& message)
    {
        // 转换成具体的消息类型，并处理相关消息， 例如：
        // shared_ptr<std::string> data =
        // std::static_pointer_cast<std::string>(message);
        return HIAI_OK;
    }
private:
};
}
/*****************************************************************************
* Matrix Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
*****************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph(std::list<std::shared_ptr<hiai::Graph>>& graphList)
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);
    // Step2: Create and Start the Graph
    status = hiai::Graph::CreateGraph(graph_config_proto_file, graphList);
    if (status != HIAI_OK)
    {
        HIAI_ENGINE_LOG(status, "Fail to start graph");
        return status;
    }
    // Step3: 设置回调函数用于接收数据
    // 如果在一个配置文件中配置了多个graph，可以遍历GraphList取相应graph执行期望操作
    std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
```

```
    {
        HIAI_ENGINE_LOG("Fail to get the graph");
        return status;
    }
    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = (*iter)->GetGraphId();
    target_port_config.engine_id = dest_engine_id;
    target_port_config.port_id = 0;
    graph->SetDataRecvFunctor(target_port_config,
        std::shared_ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));
    return HIAI_OK;
}
// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;
    // 存储用户创建的graph
    std::list<std::shared_ptr<hiai::Graph>> graphList;
    // 创建流程
    ret = HIAI_InitAndStartGraph(graphList);
    if(HIAI_OK != ret)
    {
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    }
    // 读取数据，往流程灌输数据
    std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph");
        return -1;
    }
    hiai::EnginePortID engine_id;
    engine_id.graph_id = (*iter)->GetGraphId();
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;
    bool is_end_of_data = false;
    while (!is_end_of_data)
    {
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);
        // 读取数据并填充 UserDefDataType，此数据类型注册见数据类型注册章节
        // 往Graph灌输数据
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            std::static_pointer_cast<void>(user_def_msg)))
        {
            // 队列满时返回发送数据失败，由业务逻辑处理是否重发或丢弃
            HIAI_ENGINE_LOG("Fail to send data to the graph-%u", (*iter)->GetGraphId());
            break;
        }
    }
    // 处理结束时，删除所有Graph
    std::list<std::shared_ptr<hiai::Graph>>::iterator graphIter;
    for (graphIter = graphList.begin(); graphIter != graphList.end(); ++graphIter) {
        hiai::Graph::DestroyGraph((*graphIter)->GetGraphId());
    }
    return 0;
}
```

8.2.3 Engine 实现接口（C++语言）

8.2.3.1 Engine 的构造函数与析构函数

该接口在engine.h中定义。

函数格式

Engine()

virtual ~Engine()

说明

该构造函数和析构函数接口是可选的，用户根据实际确定是否需要重载实现。

8.2.3.2 Engine::Init

初始化Engine实例的相关配置。该接口在engine.h中定义。

说明

该Init接口是可选的，用户根据实际情况决定是否需要重载实现。

函数格式

HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription> &modelDesc)

参数说明

参数	说明	取值范围
config	Engine配置项。	-
modelDesc	模型描述。	-

返回值

返回的错误码是由用户提前注册的，具体注册方式见**8.2.5.5.1 错误码注册**。

错误码示例

本API（Application Programming Interface）的错误码由用户进行注册。

重载实现示例

初始化推理Engine，在初始化过程中通过模型管家（AIModelManager）的**8.2.4.1.1 AIModelManager::Init**接口加载模型。

```
HIAI_StatusT FrameworkerEngine::Init(const hiai::AIConfig& config,
   const  std::vector<hiai::AIModelDescription>& model_desc)
{
    hiai::AIStatus ret = hiai::SUCCESS;
    // init ai_model_manager_
    if (nullptr == ai_model_manager_)
    {
        ai_model_manager_ = std::make_shared<hiai::AIModelManager>();
    }
    std::cout<<"FrameworkerEngine Init"<<std::endl;
    HIAI_ENGINE_LOG("FrameworkerEngine Init");
```

```
    for (int index = 0; index < config.items_size(); ++index)
    {

        const ::hiai::AIConfigItem& item = config.items(index);
        // loading model
        if(item.name() == "model_path")
        {
            const char* model_path = item.value().data();
            std::vector<hiai::AIModelDescription> model_desc_vec;
            hiai::AIModelDescription model_desc_;
            model_desc_.set_path(model_path);
            model_desc_vec.push_back(model_desc_);
            ret = ai_model_manager_->Init(config, model_desc_vec);

            if (hiai::SUCCESS != ret)
            {
                HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL, "[DEBUG] fail to init ai_model");
                return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
            }
        }
    }
    HIAI_ENGINE_LOG("FrameworkerEngine Init success");
    return HIAI_OK;
}
```

8.2.3.3 宏:HIAI_DEFINE_PROCESS

用户直接调用该宏定义Engine的输入与输出端口数。该宏在engine.h中定义。

本宏封装用到了以下函数：

```
HIAI_StatusT Engine::InitQueue(const uint32_t& in_port_num, const uint32_t& out_port_num);
```

相关宏：

在HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)之前调用此宏。

宏格式

HIAI_DEFINE_PROCESS（inputPortNum, outputPortNum）

参数说明

参数	说明	取值范围
inputPortNum	Engine的输入端口数。	-
outputPortNum	Engine的输出端口数。	-

调用示例

```
#define FRAMEWORK_ENGINE_INPUT_SIZE  1
#define FRAMEWORK_ENGINE_OUTPUT_SIZE  1

* @[in]: 定义一个输入端口，一个输出端口*/
HIAI_DEFINE_PROCESS(FRAMEWORK_ENGINE_INPUT_SIZE, FRAMEWORK_ENGINE_OUTPUT_SIZE)
```

8.2.3.4 宏:HIAI_IMPL_ENGINE_PROCESS

用户需要重载实现该宏，用于定义Engine的具体实现。该宏在engine.h中定义。

本宏封装用到了以下函数：

```
static HIAIEngineFactory* GetInstance();
HIAI_StatusT HIAIEngineFactory::RegisterEngineCreator(const std::string&
engine_name,HIAI_ENGINE_FUNCTOR_CREATOR engineCreatorFunc);
HIAI_StatusT HIAIEngineFactory::UnRegisterEngineCreator(const std::string& engine_name);
```

相关宏：

在HIAI_DEFINE_PROCESS（inputPortNum, outputPortNum）之后调用本宏。

宏格式

HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)

参数说明

参数	说明	取值范围
name	Config的Engine名称。	-
engineClass	Engine的实现类名称。	-
inPortNum	输入的端口数。	-

返回值

返回的错误码由用户提前注册。

错误码示例

该API的错误码由用户进行注册。

重载实现样例

定义推理Engine的实现，在实现过程中通过模型管家（AIModelManager）的**8.2.4.1.3 AIModelManager::Process**接口执行模型推理。

```
HIAI_IMPL_ENGINE_PROCESS("FrameworkerEngine", FrameworkerEngine,
FRAMEWORK_ENGINE_INPUT_SIZE)
{
    hiai::AIStatus ret = hiai::SUCCESS;
    HIAI_StatusT hiai_ret = HIAI_OK;
    // receive data
    //arg0表示Engine的输入端口（编号为0），如果有多个输入端口，可依次通过arg1（编号为1的输入端口）、
arg2（编号为2的输入端口）等来对应输入端口。通过输入端口获取上一个Engine发送的数据。
    std::shared_ptr<std::string> input_arg =
        std::static_pointer_cast<std::string>(arg0);
    if (nullptr == input_arg)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "[DEBUG] input arg is invalid");
        return HIAI_INVALID_INPUT_MSG;
    }
    std::cout<<"FrameworkerEngine Process"<<std::endl;

    //  prapare for calling the process of ai_model_manager_
    std::vector<std::shared_ptr<hiai::IAITensor>> input_data_vec;

    uint32_t len = 75264;
```

```
    HIAI_ENGINE_LOG("FrameworkerEngine:Go to Process");
    std::cout << "HIAIAippOp::Go to process" << std::endl;
    std::shared_ptr<hiai::AINeuralNetworkBuffer> neural_buffer =
std::shared_ptr<hiai::AINeuralNetworkBuffer>(new hiai::AINeuralNetworkBuffer());//
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(input_data);
    neural_buffer->SetBuffer((void*)(input_arg->c_str()), (uint32_t)(len));
    std::shared_ptr<hiai::IAITensor> input_data = std::static_pointer_cast<hiai::IAITensor>(neural_buffer);
    input_data_vec.push_back(input_data);

    //  call Process and inference
    hiai::AIContext ai_context;
    std::vector<std::shared_ptr<hiai::IAITensor>> output_data_vec;
    ret = ai_model_manager_->CreateOutputTensor(input_data_vec, output_data_vec);
    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }

    ret = ai_model_manager_->Process(ai_context, input_data_vec, output_data_vec, 0);

    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }
    std::cout<<"[DEBUG] output_data_vec size is "<< output_data_vec.size()<<std::endl;
    for (uint32_t index = 0; index < output_data_vec.size(); index++)
    {
        // send data of inference to destEngine
        std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(output_data_vec[index]);
        std::shared_ptr<std::string> output_string_ptr = std::shared_ptr<std::string>(new
std::string((char*)output_data->GetBuffer(), output_data->GetSize()));

        hiai_ret = SendData(0, "string", std::static_pointer_cast<void>(output_string_ptr));
        if (HIAI_OK != hiai_ret)
        {
            HIAI_ENGINE_LOG(this, HIAI_SEND_DATA_FAIL, "fail to send data");
            return HIAI_SEND_DATA_FAIL;
        }
    }
    return HIAI_OK;
}
```

8.2.3.5 Engine::SetDataRecvFunctor

设置Engine接收消息的回调函数。该接口在engine.h中定义。

该接口需要与**DataRecvInterface::RecvData**接口配合使用，详细原理如下：

1. 定义一个DataRecvInterface类的子类（例如DdkDataRecvInterface），初始化一个子类的对象。
2. 调用Engine::SetDataRecvFunctor接口将DdkDataRecvInterface类的对象设置成回调函数。
3. 调用Engine::SendData函数将对应的输入数据发送到对应的输出端口。
4. 调用DdkDataRecvInterface类的RecvData函数返回数据。

函数格式

HIAI_StatusT Engine::SetDataRecvFunctor(const uint32_t portId, const shared_ptr<DataRecvInterface>& userDefineDataRecv)

参数说明

参数	说明	取值范围
portId	端口ID。	-
userDefineDataRecv	用户自定义的数据接收回调函数。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_PORT_ID_ERROR	port id error

8.2.3.6 Engine::SendData

将数据从本Engine发送到指定port_id。该接口在engine.h中定义。

说明

在跨侧（Host->Device或Device->Host）传输数据时，该接口采用DMA（Direct Memory Access）传送方式，可能会影响CPU对中断请求的及时响应与处理，例如影响调用new或者malloc分配内存时的性能。

函数格式

HIAI_StatusT Engine::SendData(uint32_t portId, const std::string& messageName, const shared_ptr<void>& dataPtr， uint32_t timeOut = TIME_OUT_VALUE);

参数说明

参数	说明	取值范围
portId	Engine的输出端口号。	-
messageName	当前发送的消息名（该消息必须已经调用HiAI提供的宏已经注册过）。	-
dataPtr	指向具体的消息指针。	-

参数	说明	取值范围
timeOut	调用该接口发送数据时的一次超时时间，不指定timeout参数时，默认的一次超时时间为500ms。 若发送数据超时，系统后台会再次尝试发送数据，最大尝试16次。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_NULL_POINTER	null pointer
3	HIAI_GRAPH_ENGINE_NOT_EXIST	engine not exist
4	HIAI_GRAPH_SRC_PORT_NOT_EXIST	src port not exist

8.2.3.7 调用示例

```
/**
* @file multi_input_output_engine_example.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/


#ifndef MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_

#include "hiaiengine/engine.h"
#include "hiaiengine/data_type.h"
#include "hiaiengine/multitype_queue.h"

#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE 3
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE 2

namespace hiai {
// Define New Engine
class HIAIMultiEngineExample : public Engine {

public:
    HIAIMultiEngineExample() :
```

```
        input_que_(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE) {}

    // 重载父类Engine的Init
    HIAI_StatusT Init(const AIConfig& config,
        const std::vector<AIModelDescription>& model_desc)
    {
        return HIAI_OK;
    }

    /**
    * @ingroup hiaiengine
    * @brief HIAI_DEFINE_PROCESS
    * @[in]: 定义一个输入端口，一个输出端口
    */
    HIAI_DEFINE_PROCESS(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE)
private:

    // 私有实现一个成员变量，用来缓存输入队列
    hiai::MultiTypeQueue input_que_;
};

}

#endif //MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
```

```
/**
* @file multi_input_output_engine_example.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/

#include "multi_input_output_engine_example.h"
#include "use_def_data_type.h"
#include "use_def_errorcode.h"

namespace hiai {

/**
* @ingroup hiaiengine
* @brief HIAI_DEFINE_PROCESS : 实现多端口输入输出处理流程
* @[in]: 定义一个输入端口，一个输出端口，
*        并该Engine注册，其名为“HIAIMultiEngineExample”
*/
HIAI_IMPL_ENGINE_PROCESS("HIAIMultiEngineExample", HIAIMultiEngineExample,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE)
{
    // This Engine has three input args and two output

    // 每个端口接收到的数据可能不是同一个时刻，因此某些输入端口可能是空的
    input_que_.PushData(0,arg0);
    input_que_.PushData(1,arg1);
    input_que_.PushData(2,arg2);

    std::shared_ptr<void> input_arg1;
    std::shared_ptr<void> input_arg2;
    std::shared_ptr<void> input_arg3;

    // 仅当三个端口都有输入数据时才继续后续处理，方式一
    if (!input_que_.PopAllData(input_arg1, input_arg2, input_arg3))
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
        return HIAI_INVALID_INPUT_MSG;
    }

    // 仅当三个端口都有输入数据时才继续后续处理，方式二
```

```
/*
    if (!(input_que_.FrontData(0, input_arg1) && input_que_.FrontData(1, input_arg2) &&
input_que_.FrontData(2, input_arg3)))
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
        return HIAI_INVALID_INPUT_MSG;
    } else {
         input_que_.PopData(0, input_arg1);
         input_que_.PopData(1, input_arg2);
         input_que_.PopData(2, input_arg3);
    }
*/
    //
    // 中间业务逻辑可以直接调用 DVPP API或AIModelManger API 处理。
    //

    // 分配内存用来保存结果
    std::shared_ptr<UseDefDataTypeT> output1 =
        std::make_shared<UseDefDataTypeT>();

    std::shared_ptr<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>> output2 =
        std::make_shared<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>>();

    // 填充输出数据结构，例如简单赋值
    *output1 = *input_arg2;
    *output2 = *input_arg3;

    // 将输出端口发送到端口0
    hiai::Engine::SendData(0, "UseDefDataTypeT", std::static_pointer_cast<void>(output1));

    // 将输出端口发送到端口1
    hiai::Engine::SendData(1, "UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
        std::static_pointer_cast<void>(output2));

    return HIAI_OK;
}

}
```

8.2.4 模型管家接口（C++语言）

8.2.4.1 离线模型管家

通过如下宏标识的接口，属于当前版本的未实现接口，用户不能直接调用：__ANDROID__、ANDROID、__LITE__。

8.2.4.1.1 AIModelManager::Init

初始化接口，主要完成模型加载。

函数格式

virtual AIStatus AIModelManager::Init(const AIConfig &config, const std::vector<AIModelDescription> &model_descs = {}) override;

参数说明

参数	说明	取值范围
config	配置信息。 关于AIConfig数据类型的定义，请参见**8.2.4.3.2 AIConfig**。	-
model_descs	模型描述信息列表，支持从内存、文件加载模型，内存加载模型优先级高于从文件加载模型。 关于AIModelDescription数据类型的定义，请参见**8.2.4.3.6 AIModelDescription**。	-

返回值

SUCCESS 初始化成功/FAILED 初始化失败。

调用示例

请注意，MODEL_NAME仅支持大小写字母、数字、下划线和点，MODEL_PATH仅支持大小写字母、数字、下划线。

- 从文件加载模型：

```
AIModelManager model_mngr;
AIModelDescription model_desc;
AIConfig config;
/* 校验输入文件路径是否合法
   路径部分：支持大小写字母、数字、下划线
   文件名部分：支持大小写字母、数字、下划线和点(.)  */
model_desc.set_path(MODEL_PATH);
model_desc.set_name(MODEL_NAME);
model_desc.set_type(0);
vector<AIModelDescription> model_descs;
model_descs.push_back(model_desc);
// AIModelManager Init
AIStatus ret = model_mngr.Init(config, model_descs);
if (SUCCESS != ret)
{
    printf("AIModelManager Init failed. ret = %d\n", ret);
    return -1;
}
```

- 从内存加载模型：

```
AIModelManager model_mngr;
AIModelDescription model_desc;
vector<AIModelDescription> model_descs;
AIConfig config;
model_desc.set_name(MODEL_NAME);
model_desc.set_type(0);
char *model_data = nullptr;
uint32_t model_size = 0;
ASSERT_EQ(true, Utils::ReadFile(MODEL_PATH.c_str(), model_data, model_size));
model_desc.set_data(model_data,model_size);
model_desc.set_size(model_size);
```

```
    AIStatus ret = model_mngr.Init(config, model_descs);
    if (SUCCESS != ret)
    {
        printf("AIModelManager Init failed. ret = %d\n", ret);
        return -1;
    }
}
```

说明

model_size必须与模型实际大小保持一致。

8.2.4.1.2 AIModelManager::SetListener

设置模型管理回调函数。

说明

如果不调用此接口或调用此接口时listener设置为nullptr，表示**Process**接口为同步调用，否则为异步。

函数格式

virtual AIStatus AIModelManager::SetListener(std::shared_ptr<IAIListener> listener) override;

参数说明

参数	说明	取值范围
listener	回调函数。 关于IAIListener数据类型的定义，请参见**8.2.4.3.19 IAIListener**。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

8.2.4.1.3 AIModelManager::Process

计算接口。单模型推理时，process的缓冲队列长度限制为2048。推理队列长度超过2048，将返回失败。

函数格式

virtual AIStatus AIModelManager::Process(AIContext &context, const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout);

参数说明

参数	说明	取值范围
context	context运行时上下文信息，包含engine运行时的一些可变参数配置。 关于AIContext数据类型的定义，请参见**8.2.4.3.17 AIContext**。	-
in_data	模型输入tensor列表。 关于IAITensor数据类型的定义，请参见**8.2.4.3.20 IAITensor**。	-
out_data	模型输出tensor列表。 关于IAITensor数据类型的定义，请参见**8.2.4.3.20 IAITensor**。	-
timeout	计算超时时间，预留参数，默认值为0，配置无效。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

8.2.4.1.4 AIModelManager::CreateOutputTensor

创建输出的Tensor列表。

说明

如果用户使用该接口来创建tensor，该接口返回的内存首地址满足512对齐要求。建议用户使用AITensorFactory::CreateTensor接口来创建tensor，详见**8.2.4.1.10 AITensorFactory::CreateTensor**。

函数格式

```
virtual AIStatus AIModelManager::CreateOutputTensor(const
std::vector<std::shared_ptr<IAITensor>> &in_data,
std::vector<std::shared_ptr<IAITensor>> &out_data) override;
```

参数说明

参数	说明	取值范围
in_data	输入tensor列表。 关于IAITensor数据类型的定义，请参见**8.2.4.3.20 IAITensor**。	-
out_data	输出tensor列表。 关于IAITensor数据类型的定义，请参见**8.2.4.3.20 IAITensor**。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

8.2.4.1.5 AIModelManager::CreateInputTensor

创建输入的Tensor列表。

说明

如果用户使用该接口来创建tensor，该接口返回的内存首地址满足512对齐要求。建议用户使用AITensorFactory::CreateTensor接口来创建tensor，详见**8.2.4.1.10 AITensorFactory::CreateTensor**。

函数格式

```
virtual AIStatus
AIModelManager::CreateInputTensor(std::vector<std::shared_ptr<IAITensor>>
&in_data);
```

参数说明

参数	说明	取值范围
in_data	输入tensor列表。 关于IAITensor数据类型的定义，请参见**8.2.4.3.20 IAITensor**。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

8.2.4.1.6 AIModelManager::IsPreAllocateOutputMem

是否可以预分配输出内存。

相关函数：

AIStatus AIModelManager::CreateOutputTensor(const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data);

函数格式

virtual bool AIModelManager::IsPreAllocateOutputMem() override ;

参数说明

无。

返回值

- True：可调用CreateOutputTensor申请输出内存申请。
- False：不可调用CreateOutputTensor。

说明

只有模型管家加载一个模型时才返回true。

8.2.4.1.7 AIModelManager::GetModelIOTensorDim

获取已加载模型的输入输出尺寸。

函数格式

virtual AIStatus AIModelManager::GetModelIOTensorDim(const std::string& model_name, std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>& output_tensor) ;

参数说明

参数	说明	取值范围
model_name	模型名称。	-
input_tensor	模型输入尺寸列表。 关于TensorDimension数据类型的定义，请参见**8.2.4.3.18 TensorDimension**。	-
output_tensor	模型输出尺寸列表。 关于TensorDimension数据类型的定义，请参见**8.2.4.3.18 TensorDimension**。	-

返回值

SUCCESS初始化成功/FAILED初始化失败。

示例

例如获取resnet50模型的输入输出Tensor描述如下。

若output_tensor的size不为0，则获取到的output_tensor数据会追加在原output_tensor数据之后。

```
input_tensor
{
name = "data"                #输入层的name
data_type = 0                #预留数据类型，暂时不用
size = 20                    #内存大小，单位字节
format = 0                   #预留Tensor排布格式，暂时不用
dims = {1,3,224,224}
}
output_tensor
{
name =  "output_0_prob_0"    #输出Tensor的name，格式为：output_{数字}_{输出节点name}_{输出节点输出索引}
data_type = 0                #预留数据类型，暂时不用
size = 20                    #内存大小，单位字节
format = 0                   #预留Tensor排布格式，暂时不用
dims = {1,1000,1,1}
}
```

8.2.4.1.8 AIModelManager::GetMaxUsedMemory

根据模型名字查询模型使用内存大小。

函数格式

int32_t AIModelManager::GetMaxUsedMemory(std::string model_name);

参数说明

参数	说明	取值范围
model_name	模型名称。	-

返回值

模型使用内存大小。

8.2.4.1.9 AISimpleTensor::SetBuffer

设置tensor数据地址。

函数格式

void SetBuffer(void *data, const int32_t size, bool isown=false);

参数说明

参数	说明	取值范围
data	数据地址。	-
size	数据长度。 **说明** 单位为字节。size必须与数据实际大小一致。	-
isown	是否在tensor生命周期结束后，由tensor释放data地址的内存。 ● false：默认值，data地址的内存在tensor生命周期结束后，由用户进行free操作。 ● true：在tensor生命周期结束后tensor释放该内存，用户不能进行free操作，否则导致重复释放。	-

返回值

无。

8.2.4.1.10 AITensorFactory::CreateTensor

创建模型tensor。该接口在ai_tensor.h中定义。

函数格式

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc, void *buffer, int32_t size);

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);

std::shared_ptr<IAITensor> CreateTensor(const std::string &type);

参数说明

参数	说明	取值范围
tensor_desc	tensor描述。 关于AITensorDescription数据类型的定义，请参见**8.2.4.3.4 AITensorDescription**。	-

参数	说明	取值范围
buffer	数据地址。 **说明** ● 建议输入数据及输出数据都通过HIAI_DMalloc接口申请，这样就能够使能算法推理的零拷贝机制， 优化Process时间。 ● 由客户申请内存地址，使用结束后，由客户释放内存地址。	-
size	数据长度。 **说明** 单位为字节，size必须与数据实际大小一致。	-
type	注册Tensor时的类型。 说明： 该接口不分配内存，在非预分配内存情况下使用。请参见**8.2.4.3.4 AITensorDescription**中type描述。	-

返回值

创建模型tensor成功，则返回tensor指针。创建模型tensor失败，则返回空指针。

8.2.4.1.11 调用示例

示例 1 同步调用示例

当前示例是单模型情况下推理Engine的实现代码。

在多模型情况下，如果需要参考该示例，您需要注意以下几点：

- 声明preOutBuffer变量时，不能带有static关键字，您需要按如下方式定义：
```
bool preOutBuffer = false;
```
- 在调用Process函数前，需要调用AddPara函数分别设置每个模型的名称：
```
ai_context.AddPara("model_name", modelName);//多模型，一定要分别设置模型名称
ret = ai_model_manager_->Process(ai_context,
        inDataVec, outDataVec_, 0);
```

示例代码如下：

```
// 推理Engine Process函数实现
HIAI_IMPL_ENGINE_PROCESS("ClassifyNetEngine", ClassifyNetEngine, CLASSIFYNET_ENGINE_INPUT_SIZE)
{
    HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine Process");
    HIAI_StatusT ret = HIAI_OK;
    static bool preOutBuffer = false;
    std::vector<std::shared_ptr<hiai::IAITensor>> inDataVec;

    // 获取从上个Engine传入的数据
    std::shared_ptr<EngineTransNewT> input_arg =
        std::static_pointer_cast<EngineTransNewT>(arg0);
    // 如果传入数据为空指针，直接返回
    if (nullptr == input_arg)
    {
```

```
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG,
            "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }

    // 准备输出数据，输出数据使用HIAI_DMalloc接口分配内存， 并通过CreateTensor给到算法推理，如果使用
同步的机制
    // 调用推理的Process，则只需要分配一次输出内存
    if (preOutBuffer == false) {
        std::vector<hiai::TensorDimension> inputTensorVec;
        std::vector<hiai::TensorDimension> outputTensorVec;
        ret = ai_model_manager_->GetModelIOTensorDim(modelName, inputTensorVec, outputTensorVec);
        if (ret != hiai::SUCCESS)
        {
            HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL,
                "hiai ai model manager init fail");
            return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
        }
        // allocate OutData in advance
        HIAI_StatusT hiai_ret = HIAI_OK;
        for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
            hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
            uint8_t* buffer = nullptr;
            // HIAI_Dmalloc分配内存，该内存主要是给算法进行推理，需要调用HIAI_DFree释放内存
            // Engine析构时释放
            hiai_ret = hiai::HIAIMemory::HIAI_DMalloc(outputTensorVec[index].size, (void*&)buffer, 1000);
            if (hiai_ret != HIAI_OK || buffer == nullptr) {
                std::cout<<"HIAI_DMalloc failed"<< std::endl;
                continue;
            }
            outData_.push_back(buffer);
            shared_ptr<hiai::IAITensor> outputTensor =
                hiai::AITensorFactory::GetInstance()->CreateTensor(outputTensorDesc, buffer,
outputTensorVec[index].size);
            outDataVec_.push_back(outputTensor);
        }
        preOutBuffer = true;
    }

    // Transfer buffer to Framework directly, only one inputsize
    hiai::AITensorDescription inputTensorDesc =
        hiai::AINeuralNetworkBuffer::GetDescription();
    shared_ptr<hiai::IAITensor> inputTensor =
        hiai::AITensorFactory::GetInstance()->CreateTensor(inputTensorDesc,
        input_arg->trans_buff.get(), input_arg->buffer_size);
    // AIModelManager. fill in the input data.
    inDataVec.push_back(inputTensor);

    hiai::AIContext ai_context;
    // Process work
    ret = ai_model_manager_->Process(ai_context,
        inDataVec, outDataVec_, 0);
    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL,
            "Fail to process ai model manager");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }

    // Convert the generated data to the buffer of the string type and send the data.
    for (uint32_t index = 0; index < outDataVec_.size(); index++)
    {
        HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine SendData");
        std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outDataVec_[index]);
        std::shared_ptr<std::string> output_string_ptr =
            std::shared_ptr<std::string>(new std::string((char*)output_data->GetBuffer(), output_data-
>GetSize()));
        hiai::Engine::SendData(0, "string",
```

```
            std::static_pointer_cast<void>(output_string_ptr));
    }
    inDataVec.clear();
    return HIAI_OK;
}

ClassifyNetEngine::~ClassifyNetEngine() {
    // 释放outData预分配内存
    HIAI_StatusT ret = HIAI_OK;
    for (auto buffer : outData_) {
        if (buffer != nullptr) {
            ret = hiai::HIAIMemory::HIAI_DFree(buffer);
            buffer = nullptr;
        }
    }
 }
```

示例 2 异步调用示例

请参见DDK样例中的“ddk安装目录/ddk/sample/customop/customop_app/main.cpp”。

8.2.4.2 AIPP 配置接口

8.2.4.2.1 简介

AIPP（AI Preprocessing）用于在AI Core上完成图像预处理，包括改变图像尺寸（Crop裁剪或Padding补边）、色域转换（转换图像格式）、减均值/乘系数（改变图像像素）。

对于动态AIPP，模型转换时仅设置AIPP模式为动态，每次模型推理前需要在推理Engine的代码中设置动态AIPP参数值，然后在模型推理时可使用不同的AIPP参数。在推理Engine的代码中设置动态AIPP参数值，就需要调用本章提供的AIPP配置相关接口，这部分接口在ai_tensor.h中定义。

8.2.4.2.2 SetDynamicInputIndex

函数格式

void SetDynamicInputIndex(uint32_t dynamicInputIndex = 0);

功能说明

指定对于模型的第几个原始输入做AIPP处理。

参数说明

参数名	输入/输出	类型	描述
dynamicInputIndex	输入	uint32_t	模型原始输入的下标，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，则将dynamicInputIndex配置为1。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.3 SetDynamicInputEdgeIndex

函数格式

void SetDynamicInputEdgeIndex(uint32_t dynamicInputEdgeIndex = 0);

功能说明

如果一个模型输入为多个算子共有，即Data算子后面跟着多个算子，通过调用该接口，对Data算子的不同的输出边做不同的AIPP处理。

参数说明

参数名	输入/输出	类型	描述
dynamicInputEdgeIndex	输入	uint32_t	某个模型输入对应的输出边的下标，从0开始，默认为0。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.4 SetInputFormat

函数格式

AIStatus SetInputFormat(AippInputFormat inputFormat);

功能说明

用于设置模型的原始输入的类型。

参数说明

参数名	输入/输出	类型	描述

inputFormat	输入	AippInputFormat	模型原始输入类型。
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

返回值

参数名	类型	描述
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.5 SetCscParams（设置默认参数值）

函数格式

AIStatus SetCscParams(AippInputFormat srcFormat,

AippModelFormat dstFormat,

ImageFormat imageFormat = BT_601NARROW);

功能说明

用户可以调用此接口，实现色域转换功能，根据入参处指定的原始输入类型、目标输入类型及图片类型，自动生成AIPP中CSC（color space convert）色域转换参数的一组默认值，同时根据原始输入类型、目标输入类型打开或关闭RB/UV通道交换开关。

参数说明

参数名	输入/输出	类型	描述

srcFormat	输入	AippInputFormat	模型的原始输入类型。
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

dstFormat	输入	AippModelFormat	AIPP转换之后的目标输入类型。
			enum AippModelFormat
			{
			MODEL_RGB888_U8 = 1,
			MODEL_BGR888_U8,
			MODEL_GRAY,
			MODEL_YUV444SP_U8,
			MODEL_YVU444SP_U8
			};

imageFormat	输入	ImageFormat	图片类型，此接口之处的图片类型，当前仅支持配置为JPEG和BT_601NARROW。
			enum ImageFormat
			{
			BITMAP,
			PNG,
			JPEG,
			BT_601NARROW
			};

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

此接口支持对CSC色域转换相关参数进行快捷的配置，系统内置了一组默认的CSC配置参数，请参见《模型转换指导》中的“AIPP配置”。

支持的AIPP转换前和转换后的图片输入格式如下：

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_RGB888_U8

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_BGR888_U8

AippInputFormat::YUV420SP_U8 到 AippModelFormat::MODEL_GRAY

AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_YUV444SP_U8

AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::XRGB8888_U8 到 AippModelFormat::MODEL_GRAY

AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_BGR888_U8

AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_YUV444SP_U8

AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::RGB888_U8 到 AippModelFormat::MODEL_GRAY

如果支持的图片类型，或者图片处理格式无法满足需求，请调用另一个**8.2.4.2.5 SetCscParams（设置默认参数值）**接口设置CSC色域转换相关参数。

8.2.4.2.6 SetCscParams（根据需求设置参数值）

函数格式

void SetCscParams(bool csc_switch = false,

int16_t cscMatrixR0C0 = 0,

int16_t cscMatrixR0C1 = 0,

int16_t cscMatrixR0C2 = 0,

int16_t cscMatrixR1C0 = 0,

int16_t cscMatrixR1C1 = 0,

int16_t cscMatrixR1C2 = 0,

int16_t cscMatrixR2C0 = 0,

int16_t cscMatrixR2C1 = 0,

int16_t cscMatrixR2C2 = 0,

uint8_t cscOutputBiasR0 = 0,

uint8_t cscOutputBiasR1 = 0,

uint8_t cscOutputBiasR2 = 0,

uint8_t cscInputBiasR0 = 0,

uint8_t cscInputBiasR1 = 0,

uint8_t cscInputBiasR2 = 0);

功能说明

用户可以调用此接口，根据需求对AIPP中CSC色域转换相关参数进行灵活的配置。

参数说明

参数名	输入/输出	类型	描述
csc_switch	输入	bool	色域转换开关，默认false。
cscMatrixR0C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR0C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR0C2	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR1C2	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C0	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C1	输入	int16_t	色域转换矩阵参数。
cscMatrixR2C2	输入	int16_t	色域转换矩阵参数。
cscOutputBiasR0	输入	uint8_t	RGB转YUV时的输出偏移，默认值为0，可以只配置部分配置项。
cscOutputBiasR1	输入	uint8_t	RGB转YUV时的输出偏移，默认值为0，可以只配置部分配置项。

参数名	输入/输出	类型	描述
cscOutputBiasR2	输入	uint8_t	RGB转YUV时的输出偏移，默认值为0，可以只配置部分配置项。
cscInputBiasR0	输入	uint8_t	YUV转RGB时的输入偏移，默认值为0，可以只配置部分配置项。
cscInputBiasR1	输入	uint8_t	YUV转RGB时的输入偏移，默认值为0，可以只配置部分配置项。
cscInputBiasR2	输入	uint8_t	YUV转RGB时的输入偏移，默认值为0，可以只配置部分配置项。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.7 SetRbuvSwapSwitch

函数格式

void SetRbuvSwapSwitch(bool rbuvSwapSwitch = false);

功能说明

设置CSC色域转换前，R通道与B通道交换开关，或者U通道与V通道交换开关。

参数说明

参数名	输入/输出	类型	描述
rbuvSwapSwitch	输入	bool	true表示支持交换，false表示不支持。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.8 SetAxSwapSwitch

函数格式

void SetAxSwapSwitch(bool axSwapSwitch = false);

功能说明

设置CSC色域转换前，GBA->ARGB或者YUVA->AYUV交换开关。

参数说明

参数名	输入/输出	类型	描述
axSwapSwitch	输入	bool	true表示支持交换，false表示不支持。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.9 SetSrcImageSize

函数格式

void SetSrcImageSize(int32_t srcImageSizeW = 0, int32_t srcImageSizeH = 0);

功能说明

设置AIPP处理前，原始图片的宽和高信息。

参数说明

参数名	输入/输出	类型	描述
srcImageSizeW	输入	int32_t	原始图片的宽。
srcImageSizeH	输入	int32_t	原始图片的高。

返回值

无。

异常处理

无。

约束说明

无。

8.2.4.2.10 SetCropParams

函数格式

AIStatus SetCropParams(bool cropSwitch,

int32_t cropStartPosW, int32_t cropStartPosH,

int32_t cropSizeW, int32_t cropSizeH,

uint32_t batch_index = 0);

功能说明

设置Crop裁剪参数信息。动态AIPP支持为每个batch配置不同的Crop参数，batchIndex表示对第几个batch设置Crop参数，batchIndex取值范围[0, batchNum)，如果超出这个范围，调用接口后会返回失败。

参数说明

参数名	输入/输出	类型	描述
cropSwitch	输入	Bool	true表示支持裁剪，false表示不支持。
cropStartPosW	输入	int32_t	裁剪时，坐标点起始位置在图中横向的坐标。
cropStartPosH	输入	int32_t	裁剪时，坐标点起始位置在图中纵向的坐标。
cropSizeW	输入	int32_t	裁剪的宽度。

参数名	输入/输出	类型	描述
cropSizeH	输入	int32_t	裁剪的高度。
batchIndex	输入	uint32_t	对第几个batch做裁剪，默认为0，取值范围[0, batchNum)。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.11 SetPaddingParams

函数格式

AIStatus SetPaddingParams(int8_t paddingSwitch,

int32_t paddingSizeTop, int32_t paddingSizeBottom,

int32_t paddingSizeLeft, int32_t paddingSizeRight,

uint32_t batch_index = 0);

功能说明

设置Padding补边参数。paddingSizeTop、paddingSizeBottom，paddingSizeLeft与paddingSizeRight四个参数分别表示在图片的上下左右填充的值。

参数说明

参数名	输入/输出	类型	描述
paddingSwitch	输入	Bool	true表示支持padding，false表示不支持。

参数名	输入/输出	类型	描述
paddingSizeTop	输入	int32_t	在图片上方填充的值。
paddingSizeBottom	输入	int32_t	在图片下方填充的值。
paddingSizeLeft	输入	int32_t	在图片左方填充的值。
paddingSizeRight	输入	int32_t	在图片右方填充的值。
batchIndex	输入	uint32_t	对第几个batch做补边，默认为0，取值范围[0, batchNum）。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.12 SetDtcPixelMean

函数格式

AIStatus SetDtcPixelMean(int16_t dtcPixelMeanChn0 = 0,

int16_t dtcPixelMeanChn1 = 0,

int16_t dtcPixelMeanChn2 = 0,

int16_t dtcPixelMeanChn3 = 0,

uint32_t batch_index = 0);

功能说明

设置DTC通道均值参数。

参数说明

参数名	输入/输出	类型	描述
dtcPixelMeanChn0	输入	int16_t	通道0均值。
dtcPixelMeanChn1	输入	int16_t	通道1均值。
dtcPixelMeanChn2	输入	int16_t	通道2均值。
dtcPixelMeanChn3	输入	int16_t	通道3均值。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道均值，默认为0，取值范围[0, batchNum）。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.13 SetDtcPixelMin

函数格式

```
AIStatus SetDtcPixelMin(float dtcPixelMinChn0 = 0,
float dtcPixelMinChn1 = 0,
float dtcPixelMinChn2 = 0,
float dtcPixelMinChn3 = 0,
uint32_t batch_index = 0);
```

功能说明

设置DTC通道最小值参数。

参数说明

参数名	输入/输出	类型	描述
dtcPixelMinChn0	输入	float	DTC通道0最小值。
dtcPixelMinChn1	输入	float	DTC通道1最小值。
dtcPixelMinChn2	输入	float	DTC通道2最小值。
dtcPixelMinChn3	输入	float	DTC通道3最小值。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道最小值，默认为0，取值范围[0, batchNum）。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.14 SetPixelVarReci

函数格式

```
AIStatus SetPixelVarReci(float dtcPixelVarReciChn0 = 0,
float dtcPixelVarReciChn1 = 0,
float dtcPixelVarReciChn2 = 0,
float dtcPixelVarReciChn3 = 0,
uint32_t batch_index = 0);
```

功能说明

设置DTC通道方差或（max-min）的倒数。

参数说明

参数名	输入/输出	类型	描述
dtcPixelVarReciChn0	输入	float	DTC通道0方差。
dtcPixelVarReciChn1	输入	float	DTC通道1方差。
dtcPixelVarReciChn2	输入	float	DTC通道2方差。
dtcPixelVarReciChn3	输入	float	DTC通道3方差。
batchIndex	输入	uint32_t	对第几个batch设置DTC通道方差，默认为0，取值范围[0, batchNum）。

返回值

参数名	类型	描述
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.15 SetInputDynamicAIPP

函数格式

```
AIStatus SetInputDynamicAIPP(std::vector<std::shared_ptr<IAITensor>>& inData,
std::shared_ptr<AippDynamicParaTensor> aippParms);
```

功能说明

将生成的动态AIPP参数AippDynamicParaTensor插入到输入Tensor列表中。插入的位置由AIPP Tensor中的dynamicInputIndex和dynamicInputEdgeIndex两个属性决定，这两个属性由前面提到的**8.2.4.2.2 SetDynamicInputIndex**和**8.2.4.2.15 SetInputDynamicAIPP**两个接口设置。

AIPP Tensor将会被插入到原始输入的Tensor之后。

如果需要插入多个AippDynamicParaTensor，需要多次调用此接口，此时如果和Tensor列表中已有的AippDynamicParaTensor相比，新插入的AippDynamicParaTensor的dynamicInputIndex较小，或者dynamicInputIndex相同但dynamicInputEdgeIndex较小，则会被插入在已有的AippDynamicParaTensor前；如果dynamicInputIndex和dynamicInputEdgeIndex与列表中已有的AippDynamicParaTensor相同，则新插入的AippDynamicParaTensor覆盖已有的AippDynamicParaTensor。

注意，调用此接口时，并不会校验dynamicInputIndex和dynamicInputEdgeIndex是否合法，即dynamicInputIndex配置大于了模型的输出个数，或者dynamicInputEdgeIndex大于了对应输入的输出边的个数，接口不会返回错误。

参数说明

参数名	输入/输出	类型	描述
inData	输入	std::vector<std::shared_ptr<IAITensor>>&	输入Tensor列表
aippParms	输入	std::shared_ptr<AippDynamicParaTensor>	动态AIPP Tensor

返回值

参数名	类型	描述
-	AIStatus	设置成功返回0，如果输入非法，返回其它值。 AIStatus类型的定义如下： AIStatus = uint32_t

异常处理

无。

约束说明

无。

8.2.4.2.16 GetDynamicInputIndex

函数格式

uint32_t GetDynamicInputIndex() const;

功能说明

获取SetDynamicInputIndex设置的模型原始输入下标。

参数说明

无。

返回值

参数名	类型	描述
-	uint32_t	模型原始输入下标。

异常处理

无。

约束说明

无。

8.2.4.2.17 GetDynamicInputEdgeIndex

函数格式

uint32_t GetDynamicInputEdgeIndex() const;

功能说明

获取SetDynamicInputEdgeIndex设置的模型输入对应的输出边的下标。

参数说明

无。

返回值

参数名	类型	描述
-	uint32_t	模型输入对应的输出边的下标。

异常处理

无。

约束说明

无。

8.2.4.2.18 调用示例

```
/**
* @file mngr_sample.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <assert.h>
#include "hiaiengine/ai_model_manager.h"
using namespace std;
using namespace hiai;
// 图片数据路径
static const std::string IMAGE_FILE_PATH = "/data/input_zebra.bin";
static const char* MODEL_PATH = "/data/ResNet.davincimodel";
static const char* MODEL_NAME = "resnet18";
/*
*** brief：读取输入数据
*/
char* ReadBinFile(const char *file_name, uint32_t *fileSize)
{
    std::filebuf *pbuf;
    std::ifstream filestr;
    size_t size;
    filestr.open(file_name, std::ios::binary);
    if (!filestr)
    {
        return nullptr;
    }
    pbuf = filestr.rdbuf();
    size = pbuf->pubseekoff(0, std::ios::end, std::ios::in);
    pbuf->pubseekpos(0, std::ios::in);
    char * buffer = (char*)malloc(size);
    if (nullptr == buffer)
    {
        return nullptr;
    }
    pbuf->sgetn(buffer, size);
    *fileSize = size;
    filestr.close();
    return buffer;
}
int main(int argc, char* argv[])
{
    vector<shared_ptr<IAITensor>> model_input;
    vector<shared_ptr<IAITensor>> model_output;
    AIModelManager model_mngr;
    AIModelDescription model_desc;
    AIConfig config;
    AIContext context;
    model_desc.set_path(MODEL_PATH);
    model_desc.set_name(MODEL_NAME);
    model_desc.set_type(0);
    vector<AIModelDescription> model_descs;
    model_descs.push_back(model_desc);
    // AIModelManager Init
    AIStatus ret = model_mngr.Init(config, model_descs);
    if (SUCCESS != ret)
    {
        printf("AIModelManager Init failed. ret = %d\n", ret);
        return -1;
```

```
    }
    // 输入tensor
    // 输入tensor会在读取图片数据后重新设置，这里的作用只是初始化
    AITensorDescription tensor_desc = AINeuralNetworkBuffer::GetDescription();
    shared_ptr<IAITensor> input_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
    if (nullptr == input_tensor)
    {
        printf("Create input_tensor failed.\n");
        return -1;
    }
    // 读取图片数据
    uint32_t image_data_size = 0;
    float* image_data = (float*)ReadBinFile(IMAGE_FILE_PATH.c_str(), &image_data_size);
    if (nullptr == image_data)
    {
        printf("ReadBinFile failed bin file path= %s \n", IMAGE_FILE_PATH.c_str());
        return -1;
    }
    // 将图片数据地址指针及长度，设置给input_simple_tensor
    shared_ptr<AISimpleTensor> input_simple_tensor = static_pointer_cast<AISimpleTensor>(input_tensor);
    input_simple_tensor->SetBuffer((void*)image_data, image_data_size);
    model_input.push_back(input_tensor);
    // 创建输出tensor
    if(model_mngr.IsPreAllocateOutputMem())
    {
        ret = model_mngr.CreateOutputTensor(model_input, model_output);
        if (SUCCESS != ret)
        {
            printf("CreateOutputTensor failed.ret = %d\n", ret);
            delete image_data;
            return -1;
        }
    }
    else
    {
        // 用户创建tensor
        ret = model_mngr.GetModelIOTensorDim(MODEL_NAME, input_tensor_dims, output_tensor_dims);
        std::vector<TensorDimension> input_tensor_dims;
        std::vector<TensorDimension> output_tensor_dims;
        for(TensorDimension & dims : output_tensor_dims)
        {
            shared_ptr<IAITensor> output_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
            shared_ptr<AISimpleTensor> output_simple_tensor =
static_pointer_cast<AISimpleTensor>(output_tensor);
            output_simple_tensor->setBuffer((void*) new char[dims.size], dims.size);
            model_output.push_back(output_tensor)
        }
    }

    bool dynamic_aipp_flag = true;
    if (dynamic_aipp_flag)
    {
        const int batch_number = 2;
        AITensorDescription desc = AippDynamicParaTensor::GetDescription(std::to_string(batch_number));
        shared_ptr<IAITensor> tensor = AITensorFactory::GetInstance()->CreateTensor(desc);
        shared_ptr<AippDynamicParaTensor> aipp_params_tensor =
static_pointer_cast<AippDynamicParaTensor>(tensor);
        //设置模型的原始输入的类型。
        aipp_params_tensor->SetInputFormat(hiai::YUV420SP_U8);
        //设置色域矩阵参数值
        aipp_params_tensor->SetCscParams(hiai::YUV420SP_U8, hiai::MODEL_BGR888_U8, hiai::JPEG);
        //设置原始图片的宽和高信息
        aipp_params_tensor->SetSrcImageSize(256, 224);
        //如果有多batch的情况下，为每个batch设置AIPP参数。
        for (int i = 0; i < batch_number; i++) {
            //设置抠图参数值
            aipp_params_tensor->SetCropParams(true, 0, 0, 200, 200, i);
            //设置Padding参数值
            aipp_params_tensor->SetPaddingParams(true, 12, 12, 12, 12, i);
```

```
            //设置DTC通道均值参数
            aipp_params_tensor->SetDtcPixelMean(104, 117, 123, 0, i);
            //设置DTC通道方差或（max-min）的倒数
            aipp_params_tensor->SetPixelVarReci(1.0, 1.0, 1.0, 1.0, i);
        }

    model_mngr.SetInputDynamicAIPP(model_input, aipp_params_tensor);
    }

    // 启动模型推理
    printf("Start process.\n");
    ret = model_mngr.Process(context, model_input, model_output, 0);
    if (SUCCESS != ret)
    {
        printf("Process failed.ret = %d\n", ret);
        return -1;
    }
    // 因未设置监听器，所以模型推理为同步调用，可直接取出模型推理输出数据
    shared_ptr<AISimpleTensor> result_tensor = static_pointer_cast<AISimpleTensor>(model_output[0]);
    printf("Get Result, buffsize is %d",result_tensor->GetSize());
    for(TensorDimension & dims : output_tensor_dims)
    {
    }
    printf("predict ok.\n");
    return 0;
}
```

8.2.4.3 数据类型

8.2.4.3.1 AIConfigItem

AIConfig中配置项描述，详细实现参考ai_type.proto。

```
message AIConfigItem
{
    string name = 1;   // 配置项名称
    string value = 2;  // 配置项值
    repeated AIConfigItem sub_items = 3;  // 配置子项
};
```

8.2.4.3.2 AIConfig

调用模型管家Init接口时作为入参，详细实现参考ai_types.proto。

```
message AIConfig
{
    repeated AIConfigItem items = 1; // 配置项列表
};
```

8.2.4.3.3 AITensorParaDescription

tensor参数描述，详细实现参考ai_types.proto。

```
message AITensorParaDescription
{
    string name = 1;   // 参数名称
    string type = 2;   // 参数类型
    string value = 3;  // 参数值
    string desc = 4;   // 参数描述
    repeated AITensorParaDescription sub_paras = 5; // 子参数列表
};
```

8.2.4.3.4 AITensorDescription

tensor描述，主要用于描述模型的输入输出信息，详细实现参考ai_types.proto。

```
// Tensor描述
message AITensorDescription
{
    string name = 1;   // Tensor名称
    string type = 2;   // Tensor类型
    repeated string compatible_type = 3; // 指定可以兼容的所有父类类型
    repeated AITensorParaDescription paras = 4; // 参数列表
};
```

8.2.4.3.5 AITensorDescriptionList

tensor描述列表，主要用于描述模型的输入输出信息列表，详细实现参考ai_types.proto。

```
message AITensorDescriptionList
{
    repeated AITensorDescription tensor_descs = 1; // tensor列表
}
```

8.2.4.3.6 AIModelDescription

调用Init接口时作为输入，用于描述模型，详细实现参考ai_types.proto。

```
messAIPPDynaage AIModelDescription
{
    string name = 1;  // 模型名称，支持大小写字母、数字、下划线和点(.)
    int32 type = 2;   // 模型类型，当前仅支持 DAVINCI_OFFLINE_MODEL类型，值为0
                      // 模型管家已新增模型解析能力，因此该字段不设置亦无碍，为了保持向前兼容，特此保
留--2018/11/24
    string version = 3;  // 模型版本
    int32 size = 4;  // 模型大小
    string path = 5;  // 模型路径，支持大小写字母、数字、下划线
    repeated string sub_path = 6;  // 辅助模型路径，用于模型文件为多个的情况，比如caffe在线模型
                                   // 为了向前兼容，我们没有对path字段做修改（修改为repeated），而是新增了
sub_path字段
    string key = 7;  // 模型秘钥
    repeated string sub_key = 8; // 辅助模型秘钥，用于模型秘钥为多个的情况，比如caffe在线模型
                                 // 为了向前兼容，我们没有对key字段做修改（修改为repeated），而是新增了sub_key字
段
    enum Frequency
    {
        UNSET   =0;
        LOW     =1;
        MEDIUM  =2;
        HIGH    =3;
    }
    Frequency frequency = 9;
    enum DeviceType
    {
        NPU  = 0;
        IPU  = 1;
        MLU  = 2;
        CPU  = 3;
        NONE = 255;
    }
    DeviceType device_type = 10;
    enum Framework
    {
        OFFLINE    =0;
        CAFFE      =1;
        TENSORFLOW =2;
    }
    Framework framework = 11;
    bytes data = 100;  // 模型数据
    repeated AITensorDescription inputs = 12;  // 输入Tensor描述
    repeated AITensorDescription outputs = 13; // 输出Tensor描述
};
```

8.2.4.3.7 AINNNodeDescription

NN（neural network） Node描述，主要用于描述NN Node需要的模型、输入tensor、输出tensor等，详细实现参考ai_types.proto。

```
message AINNNodeDescription
{
    string name = 1;  // NNNode名称
    string desc = 2;  // NNNode描述
    bool isPreAllocateOutputMem = 3;  // 是否预分配输出内存
    AIConfig config = 4; // 配置参数
    repeated AIModelDescription model_list = 5; // nnnode需要的模型列表
    repeated AITensorDescription inputs = 6;   // 输入Tensor描述
    repeated AITensorDescription outputs = 7;  // 输出Tensor描述
    bool need_verify = 8; // 串联时是否需要校验Tensor匹配
    repeated string ignored_check_aitensor = 9; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor列表
};
```

8.2.4.3.8 AINNNodeDescriptionList

NN（neural network） Node描述列表，主要用于描述NN Node需要的模型、输入tensor、输出tensor等，详细实现参考ai_types.proto。

```
message AINNNodeDescriptionList
{
    repeated AINNNodeDescription nnnode_descs = 1; // nnnode列表
}
```

8.2.4.3.9 AIAPIDescription

API描述，主要用于描述API的名称、输入tensor、输出tensor等，详细实现请参考ai_types.proto。

```
message AIAPIDescription
{
    string name = 1;  // API名称
    string desc = 2;  // API描述
    bool isPreAllocateOutputMem = 3;  // 是否预分配输出内存
    AIConfig config = 4; // 配置参数
    repeated AITensorDescription inputs = 5;   // 输入Tensor描述
    repeated AITensorDescription outputs = 6;  // 输出Tensor描述
    bool need_verify = 7; // 串联时是否需要校验Tensor匹配
    repeated string ignored_check_aitensor = 8; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor列表
};
```

8.2.4.3.10 AIAPIDescriptionList

API描述列表，主要用于描述API的名称、输入tensor、输出tensor等，详细实现请参考ai_types.proto。

```
message AIAPIDescriptionList
{
    repeated AIAPIDescription api_descs = 1; // API列表
}
```

8.2.4.3.11 AIOPDescription

该类型在ai_types.proto中定义。

```
//AI Operation描述
message AIOPDescription
{
    enum OP_Type
    {
        AI_API = 0;
```

```
        AI_NNNODE = 1;
    }

    OP_Type type = 1;
    AINNNodeDescription nnnode_desc = 2;
    AIAPIDescription api_desc = 3;
};
```

8.2.4.3.12 AIOPDescriptionList

该类型在ai_types.proto中定义。

```
//AI Operation描述列表
message AIOPDescriptionList
{
    repeated AIOPDescription op_descs = 1; //  AI Operation列表
}
```

8.2.4.3.13 NodeDesc

该类型在ai_types.proto中定义。

```
message NodeDesc
{
   string name=1; // IAINNNode或ALG_API 名字
   AIConfig config=2;  //IAINNNode或ALG_API 需要的初始化参数
   repeated  AIModelDescription mode_desc=3; //IAINNNode 需要的初始化参数
}
```

8.2.4.3.14 EngineDesc

该类型在ai_types.proto中定义。

```
message EngineDesc
{
    enum RunSide
    {
        DEVICE=0;
        HOST=1;
    }
    enum EngineType
    {
        NORMAL=0;
        SOURCE=1;
        DEST=2;
    }
    uint32 id =1;   //Engine ID (节点)
    EngineType type=2;
    string name =3; //Engine 节点名字
    repeated string so_name=4; //需要的所有动态库so文件名列表
    RunSide side=5; //部署在Host侧还是Device侧
    int32 priority=6; //节点优先级
    uint32 instance_cnt=7; //实例个数（相当于线程个数）
    repeated uint32 next_node_id=8; //下一个节点列表
    bool user_input_cb=9;   //IDE 可以忽略
    bool user_output_cb=10; //IDE 可以忽略
    repeated NodeDesc oper=11; //HIAIEngine Node 列表
}
```

8.2.4.3.15 GraphInitDesc

该类型在ai_types.proto中定义。

```
message GraphInitDesc
{
    int32 priority=1; //Graph 整个进程的优先级
}
```

8.2.4.3.16 GeneralFileBuffer

该类型在ai_types.proto中定义。

```
message GeneralFileBuffer
{
    bytes raw_data = 1;
    string file_name = 2;
}
```

8.2.4.3.17 AIContext

在使用异步调用模型管家Process接口时，用于保存process上下文，AIContext中保存string类型键值对。该数据类型在ai_types.h中定义。

```
class AIContext
{
public:
    /*
    * @brief 获取参数
    * @param [in] key  参数对应的key
    * @return sring key对应的值，如果不存在，则返回空字符串
    */
    const std::string GetPara(const std::string &key) const;

    /*
    * @brief 设置参数
    * @param [in] key    参数对应的key
    * @param [in] value  参数对应的value
    */
    void AddPara(const std::string &key, const std::string &value);

    /*
    * @brief 删除参数
    * @param [in] key 待删除参数对应的key
    */
    void DeletePara(const std::string &key);

    /*
    * @brief 获取所有参数
    * @param [out] keys 所有已设置的参数key
    */
    void GetAllKeys(std::vector<std::string> &keys);

#if defined( __ANDROID__) || defined(ANDROID)
        std::string  Serialize();

        AIStatus  Deserialize(std::string str);
#endif

private:
    std::map<std::string, std::string> paras_; /** 参数的名值对定义 */
};
```

8.2.4.3.18 TensorDimension

调用**AIModelManager::GetModelIOTensorDim**接口可获取模型输入/输出尺寸的信息，主要包含tensor的dim信息，数据类型，内存大小，tensor名字。该数据类型在ai_types.h中定义。

```
/*
    * 描述Tensor尺寸
    */
    struct TensorDimension
    {
        uint32_t n;  //批量大小Batch
        uint32_t c;  //特征图通道Channels
        uint32_t h;  //特征图高度Height
        uint32_t w;  //特征图宽度Width
```

```
    uint32_t data_type;
    uint32_t size;
    std::string  name;
};
```

- 若执行模型转换时，未配置AIPP（AI Preprocessing，用于在AI Core上完成图像预处理）相关参数，则通过**AIModelManager::GetModelIOTensorDim**接口获取到的模型输入/输出尺寸的信息如下：
  - n、c、h、w就是转换前模型的n、c、h、w。
  - data_type枚举值的定义如下：

```
enum DataType
{
DT_UNDEFINED = 0; // Used to indicate a DataType field has not been set.
DT_FLOAT = 1; // float type，数据大小为4字节
DT_FLOAT16 = 2; // fp16 type，数据大小为2字节
DT_INT8 = 3; // int8 type，数据大小为1字节
DT_UINT8 = 4; // uint8 type，数据大小为1字节
DT_INT16 = 5; // int16 type，数据大小为2字节
DT_UINT16 = 6; // uint16 type，数据大小为2字节
DT_INT32 = 7; // int32 type，数据大小为4字节
DT_INT64 = 8; // int64 type，数据大小为8字节
DT_UINT32 = 9; // unsigned int32，数据大小为4字节
DT_UINT64 = 10; // unsigned int64，数据大小为8字节
DT_BOOL = 11; // bool type，数据大小为1字节
DT_DOUBLE = 12; // double type，数据大小为8字节
}
```

  - size=n*c*h*w*各类型数据的字节数
- 若执行模型转换时，配置AIPP相关参数，则通过**AIModelManager::GetModelIOTensorDim**接口获取到的模型输入/输出尺寸的信息如下：
  - n、c、h、w依然是转换前模型的n、c、h、w，并不是在AIPP处配置的h（对应src_image_size_h参数）、w（对应src_image_size_w参数）。
  - data_type值固定为3，表示DT_INT8类型。
  - size的值为AIPP处理后的图片数据的大小，不同输入格式对应的size不同，具体见**表8-1**。

表 8-1 size 公式

input_format	size
YUV400_U8	n * src_image_size_w * src_image_size_h
YUV420SP_U8	n * src_image_size_w * src_image_size_h * 1.5
XRGB8888_U8	n * src_image_size_w * src_image_size_h * 4
RGB888_U8	n * src_image_size_w * src_image_size_h * 3

说明

关于模型转换或AIPP的说明，请参见《模型转换指导》。

8.2.4.3.19 IAIListener

异步调用Process时需要配置IAIListener，用于模型执行结束后，回调通知，具体实现如下。该类在ai_nn_node.h中定义。

```
/*
* 异步回调接口，由调用方实现
*/
class IAIListener
{
public:
    virtual ~IAIListener(){}
    /*
    * @brief 异步回调接口
    * @param [in] context 运行时上下文信息，包含nnnode运行时的一些可变参数配置
    * @param [in] result 执行结束时任务状态
    * @param [in] out_data 执行结束时的输出数据
    */
    virtual void OnProcessDone(const AIContext &context, int result,-
        const std::vector<std::shared_ptr<IAITensor>> &out_data) = 0;
    / *
     * @brief 服务死亡回调接口，当客户端到服务端挂死时，通知应用
     */
    virtual void OnServiceDied() {};
};
```

8.2.4.3.20 IAITensor

Porcess函数时使用IAITensor作为模型的输入输出。该类在ai_tensor.h中定义。

```
/**
* Tensor，用于表示输入输出的数据
*/
class IAITensor
{
public:
    IAITensor() {}
    virtual ~IAITensor() {};
    /*
     * @brief 通过AITensorDescription设置参数
     * @param [in] tensor_desc tensor描述
     * @return true: init成功
     * @return false: init失败，失败的可能原因比如tensor_desc与当前tensor不符
    */
    virtual bool Init(const AITensorDescription &tensor_desc) = 0;
   /*
    * @brief 获取类型名称
    */
    virtual const char* const GetTypeName() = 0;
    /*
    * @brief  获取序列化后字节长度
    */
    virtual uint32_t ByteSizeLong() = 0;
    virtual void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share) = 0;

    virtual bool IsSupportZerocpy() = 0;

#if defined( __ANDROID__) || defined(ANDROID)
        /*
        * @brief 序列化到buffer，用于数据跨进程交互
        * @param [in] buffer 序列化输出的内存区域地址，由调用方分配
        * @param [in] size 输出内存区域的大小
        * @return SUCCESS: 成功
        *         FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
        */
        virtual AIStatus SerializeTo(void* buffer, const uint32_t size) = 0;

        /*
        * @brief 从buffer反序列化回tensor，用于数据跨进程交互
        * @param [in] buffer 输入的内存区域地址
        * @param [in] size 输入内存区域的大小
        * @return SUCCESS: 成功
        *         FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
        */
```

```
    virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) = 0;
#endif
};
```

8.2.4.4 其他用于编译依赖的接口

以下接口属于编译依赖接口，不推荐直接调用。

8.2.4.4.1 AIAlgAPIFactory

ALG的API注册工厂类。该类在ai_alg_api.h中定义。

```
class AIAlgAPIFactory
{
public:
    static AIAlgAPIFactory* GetInstance();

    /*
    * @brief 获取API
    * @param [in] name api名称
    * @return API原型指针
    */
    AI_ALG_API GetAPI(const std::string &name);

    /*
    * @brief 注册API
    * @param [in] desc api描述
    * @param [in] func api定义
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus RegisterAPI(const AIAPIDescription &desc, AI_ALG_API func);

    /*
    * @brief 获取所有API描述
    * @param [in] api_desc_list api描述列表
    */
    void GetAllAPIDescription(AIAPIDescriptionList &api_desc_list);

    /*
    * @brief 卸载注册API
    * @param [in] api_desc api描述
    */
    AIStatus UnRegisterApi(const AIAPIDescription &api_desc);

    /*
    * @brief 获取API描述
    * @param [in] name api名称
    * @param [in] api_desc api描述
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus GetAPIDescription(const std::string &name, AIAPIDescription &api_desc);

private:
    std::map<std::string, AI_ALG_API> func_map_;
    std::map<std::string, AIAPIDescription> desc_map_;
    std::mutex api_reg_lock_;
};
```

8.2.4.4.2 AIAlgAPIRegisterar

对AIAlgAPIFactory工厂类的注册封装。该类在ai_alg_api.h中定义。

```
class AIAlgAPIRegisterar
{
public:
```

```
    AIAlgAPIRegisterar(const AIAPIDescription &desc, AI_ALG_API func)
    {
        AIAlgAPIFactory::GetInstance()->RegisterAPI(desc, func);
        api_desc_ = desc;
    }

    ~AIAlgAPIRegisterar()
    {
        AIAlgAPIFactory::GetInstance()->UnRegisterApi(api_desc_);
    }
private:
    AIAPIDescription api_desc_;
};
```

8.2.4.4.3 REGISTER_ALG_API_UNIQUE

API注册宏，在API实现中使用。该宏在ai_alg_api.h中定义。

```
#define REGISTER_ALG_API_UNIQUE(desc, ctr, func) \
    AIAlgAPIRegisterar g_##ctr##_api_registerar(desc, func)
```

8.2.4.4.4 REGISTER_ALG_API

对REGISTER_ALG_API_UNIQUE的封装宏。该宏在ai_alg_api.h中定义。

```
#define REGISTER_ALG_API(name, desc, func) \
    REGISTER_ALG_API_UNIQUE(desc, name, func)
```

8.2.4.4.5 AIModelManager

模型管理类。该类在ai_model_manager.h中定义。

```
class AIModelManager : public IAINNNode
{
public:
    AIModelManager();

    /*
     * @brief 设置动态batch接口，用户不需要调用该接口。
     * @param [in] inputDim 模型输入尺寸
     * @param [in] input 输入数据
     * @param [out] inputIndex 需要设置动态batch输入的序号, 从0开始
     * @param [in] batchNumber 动态batch数
     * @return SUCCESS 成功
     * @return 其他 失败
     */
     AIStatus SetInputDynamicBatch(const vector<TensorDimension>& inputDim, std::vector
<std::shared_ptr <IAITensor> > &input,
        uint32_t inputIndex, uint32_t batchNumber);

    /*
     * @brief 是否可以预分配输出内存，该接口由业务NNNode实现，默认值为true.
     */
     virtual bool IsPreAllocateOutputMem() override;

    /*
     * @brief 获取AINNNodeDescription对象.
     */
     static AINNNodeDescription GetDescription();

     /*
     * @brief 获取已加载模型的输入输出尺寸
     * @param [in] model_name 模型名字
     * @param [out] input_tensor 模型输入尺寸
     * @param [out] output_tensor 模型输出尺寸
     * @return SUCCESS 成功
     * @return 其他 失败
     */
```

```
    AIStatus GetModelIOTensorDim(const std::string& model_name,
        std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>& output_tensor);

    /*
    * @brief 设置线程推理请求ID
    * @param [in] request_id 模型名字
    * @return 无
    */
    static void SetRequestId(uint64_t request_id);

    ~AIModelManager();

#ifdef __LITE__
    /*
    * @brief 闲置超时调用，释放资源。实现时调用的是UnloadModels。
    */
    virtual AIStatus IdleRelease();
    /*
    * @brief 超时后恢复，包含打开设备，加载模型等。实现时调用的是LoadModels。
    */
    virtual AIStatus IdleResume();
#endif // __LITE__

  private:
    AIModelManagerImpl* impl_;
  };
```

8.2.4.4.6 getModelInfo

解析一个模型文件，获取模型信息。该接口在ai_model_parser.h中定义。

```
    /**
    * @ingroup hiai
    * @brief 解析一个模型文件,获取模型信息
    * @param [in] model_path 模型文件路径
    * @param [in] key 模型解密秘钥
    * @param [out] model_desc 模型信息
    * @return Status 运行结果
    */

    hiai::AIStatus getModelInfo(const std::string & model_path,
        const std::string & key, AIModelDescription & model_desc);
```

8.2.4.4.7 IAINNNode

NN Node接口，业务提供方实现。IAINNNode类在ai_nn_node.h中定义。

```
  class IAINNNode
  {
  public:
    virtual  ~IAINNNode(){}
    /*
    * @brief 初始化接口，业务在该接口中实现模型加载或其他初始化动作
    * @param [in] model_desc 模型信息，如果不需要模型，则传入空的vector
    * @param [in] config 配置参数
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus Init(const AIConfig &config,
        const std::vector<AIModelDescription> &model_descs = {}) = 0;

    /*
    * @brief 设置监听
    * @parm [in] 如果listener设置为nullptr，表示process接口为同步调用,否则为异步
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus SetListener(std::shared_ptr<IAIListener> listener) = 0;
```

```
    /*
    * @brief 计算接口
    * @param [in] context 运行时上下文信息，包含nnnode运行时的一些可变参数配置
    * @param [in] in_data 输入数据
    * @param [out] out_data 输出数据
    * @param [in] timeout 超时时间，同步调用时无效
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus Process(AIContext &context,
        const std::vector<std::shared_ptr<IAITensor>> &in_data,
        std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout) = 0;

    /*
    * @brief 创建输出的Tensor列表
    * @param [in] in_data 输入tensor列表，计算输出时可能使用
    * @param [out] out_data 输出的tensor列表
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    virtual AIStatus CreateOutputTensor(
        const std::vector<std::shared_ptr<IAITensor>> &in_data,
        std::vector<std::shared_ptr<IAITensor>> &out_data) { (void)in_data;(void)out_data;return SUCCESS; }

    /*
    * @brief 是否可以预分配输出内存，该接口由业务NNNode实现，默认值为true.
    */
    virtual bool IsPreAllocateOutputMem() { return true; }

    /*
    * @brief 判断nnnode是否有效
    */
    virtual AIStatus IsValid() { return SUCCESS; }

    /*
    * @brief 查询node 支持的同步方式
    * @return BOTH    支持同步异步
    * @return ASYNC   仅支持异步
    * @return SYNC    仅支持同步
    */
    virtual AI_NODE_EXEC_MODE GetSupportedExecMode() { return AI_NODE_EXEC_MODE::BOTH; }

#ifdef __LITE__
    /*
    * @brief nnnode闲置超时，需要释放的资源，由用户自定义实现，适用于lite场景。如果使用了modelmanger，需要在实现该函数时调用相应modelmanager释放资源的方法IdleRelease
    */
    virtual AIStatus IdleRelease() { return SUCCESS; }
    /*
    * @brief nnnode超时后恢复，包含打开设备，加载模型等，由用户自定义实现，适用于lite场景。如果使用了modelmanger，需要在实现该函数时调用相应modelmanager的方法IdleResume
    */
    virtual AIStatus IdleResume() { return SUCCESS; }
    /*
    * @brief 设置最大闲置时间，如果超过该时间则自动销毁释放资源，未设置情况下，默认60s
    *        业务构造函数中调用
    *        在AIServiceBase中实现
    * @param [in] time 最大闲置时间，单位ms
    */
    virtual void SetMaxIdleTime(const int32_t time) { (void)time;}
#endif // __LITE__

    /*
    * @brief 获取业务最大使用内存
    * @return 业务最大使用内存大小
    */
    virtual uint32_t GetMaxUsedMemory() { return 0; }
};
```

8.2.4.4.8 AINNNodeFactory

支持业务引擎自注册，提供创建NN Node接口，提供已注册NN Node描述信息查询（按名称查询以及查询全部）。AINNNodeFactory类在ai_nn_node.h中定义。

```
class AINNNodeFactory
{
public:
    static AINNNodeFactory* GetInstance();

    /*
    * @brief 根据nnnode name创建NNNode
    * @param [in] name nnnode名称
    * @return std::shared_ptr<IAINNNode> name对应的NNNode对象指针，如果返回nullptr，表示找不到对应的nnnode
    */
    std::shared_ptr<IAINNNode> CreateNNNode(const std::string &name);

    /*
    * @brief 获取所有已注册nnnode的描述信息
    * @param [out] nnnode_desc 所有已注册的nnnode描述信息
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    void GetAllNNNodeDescription(AINNNodeDescriptionList &nnnode_desc);

    /*
    * @brief 根据nnnode name获取nnnode的描述信息
    * @param [in] name nnnode名称
    * @param [out] engin_desc nnnode描述信息
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus GetNNNodeDescription(const std::string &name, AINNNodeDescription &engin_desc);

    /*
    * @brief 注册NNNode创建函数
    * @param [in] nnnode_desc nnnode描述信息
    * @param [in] create_func nnnode创建函数
    * @return SUCCESS 成功
    * @return 其他    失败
    */
    AIStatus RegisterNNNodeCreator(const AINNNodeDescription &nnnode_desc,
        AINNNODE_CREATE_FUN create_func);
    AIStatus RegisterNNNodeCreator(const string nnnode_str,
        AINNNODE_CREATE_FUN create_func);

    /*
    * @brief 注销NNNode
    * @param [in] name nnnode名称
    * @return SUCCESS 成功
    */
    AIStatus UnRegisterNNNode(const AINNNodeDescription &nnnode_desc);
    AIStatus UnRegisterNNNode(const string nnnode_str);

private:
    std::map<std::string, AINNNODE_CREATE_FUN> create_func_map_;
    std::map<std::string, AINNNodeDescription> nnnode_desc_map_;
    std::mutex  map_lock_;
};
```

8.2.4.4.9 AINNNodeRegisterar

NN Node注册类。AINNNodeRegisterar类在ai_nn_node.h中定义。

```
class AINNNodeRegisterar
{
public:
```

```
    AINNNodeRegisterar(const AINNNodeDescription &nnnode_desc, AINNNODE_CREATE_FUN
create_func)
    {
        AINNNodeFactory::GetInstance()->RegisterNNNodeCreator(nnnode_desc, create_func);
        nnnode_desc_ = nnnode_desc;
        nnnode_str_.erase(nnnode_str_.begin(),nnnode_str_.end());
    }

    ~AINNNodeRegisterar(){
            AINNNodeFactory::GetInstance()->UnRegisterNNNode(nnnode_desc_);
    }
private:
    AINNNodeDescription nnnode_desc_;
    string nnnode_str_;
};
```

8.2.4.4.10 REGISTER_NN_NODE

NN Node注册宏。

```
 /*
 * @brief NNNode 注册宏，业务NNNode在实现类中使用
 * 直接使用 REGISTER_ENGINE(desc, clazz)
 * @param [in] desc nnnode描述信息对象
 * @param [in] clazz nnnode类名
 */
#define REGISTER_NN_NODE(desc, name) \
    std::shared_ptr<IAINNNode> NNNode_##name##_Creator() \
    { \
        return std::make_shared<name>(); \
    } \
    AINNNodeRegisterar g_nnnode_##name##_creator(desc,NNNode_##name##_Creator)
```

8.2.4.4.11 AITensorGetBytes

获取Tensor字节大小。该接口在ai_tensor.h中定义。

```
extern uint32_t AITensorGetBytes(int32_t data_type);
```

8.2.4.4.12 AITensorFactory

Tensor工厂类，用来创建Tensor。AITensorFactory类在ai_tensor.h中定义。

```
class AITensorFactory
{
public:
    static AITensorFactory* GetInstance();

    /*
    * @brief 通过参数创建Tensor，包含分配内存
    * @param [in] tensor_desc 包含Tensor参数的描述信息
    * @return shared_ptr<IAITensor> 创建完成的Tensor指针，如果创建失败，则返回nullptr
    */
    std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);

    /*
    * @brief 通过type创建Tensor，不分配内存，非预分配情况下使用
    * @param [in] type Tensor注册时的类型
    * @return shared_ptr<IAITensor> 创建完成的Tensor指针，如果创建失败，则返回nullptr
    */
    std::shared_ptr<IAITensor> CreateTensor(const std::string &type);


#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief 通过参数、buffer创建Tensor，内容从buffer反序列化得到
    * @param [in] tensor_desc 包含Tensor参数的描述信息
    * @param [in] buffer 已存在的数据缓存
```

```
    * @param [in] size buffer的大小
    * @return shared_ptr<IAITensor> 创建完成的Tensor指针，如果创建失败，则返回nullptr
    */
    std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
        const void *buffer, const int32_t size);
#else
    std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
        void *buffer, int32_t size);
#endif

    /*
    * @brief 注册Tensor
    * @param [in] tensor_desc tensor描述
    * @param [in] create_func tensor创建函数
    */
    AIStatus RegisterTensor(const AITensorDescription &tensor_desc,
        CREATE_TENSOR_FUN create_func);
    AIStatus RegisterTensor(const string tensor_str, CREATE_TENSOR_FUN create_func)
    {
        // 加锁，防止多线程并发
        AITensorDescription tensor_desc;
        tensor_desc.set_type(tensor_str);
        return RegisterTensor(tensor_desc, create_func);
    }

    /*
    * @brief 卸载注册Tensor
    * @param [in] tensor_desc tensor描述
    */
    AIStatus UnRegisterTensor(const AITensorDescription &tensor_desc);
    AIStatus UnRegisterTensor(const string tensor_str)
    {
        AITensorDescription tensor_desc;
        tensor_desc.set_type(tensor_str);
        return UnRegisterTensor(tensor_desc);
    }

    /*
    * @brief 获取所有的Tensor列表
    * @param [out] tensor_desc_list 输出Tensor描述列表
    */
    void GetAllTensorDescription(AITensorDescriptionList &tensor_desc_list);

    /*
    * @brief 获取tensor描述
    */
    AIStatus GetTensorDescription(const std::string& tensor_type_name,
        AITensorDescription &tensor_desc);

  private:
    std::map<std::string, AITensorDescription> tensor_desc_map_;
    std::map<std::string, CREATE_TENSOR_FUN> create_func_map_;
    std::mutex tensor_reg_lock_;
  };
```

8.2.4.4.13 REGISTER_TENSOR_CREATER_UNIQUE

Tensor注册宏。该宏在ai_tensor.h中定义。

```
#define REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func) \
  AITensorRegisterar g_##name##_tensor_registerar(desc, func)
```

8.2.4.4.14 REGISTER_TENSOR_CREATER

对Tensor注册宏的封装宏。该宏在ai_tensor.h中定义。

```
#define REGISTER_TENSOR_CREATER(name, desc,func) \
    REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func)
```

8.2.4.4.15 AISimpleTensor

简单类型的数据Tensor。该类在ai_tensor.h中定义。

```
class AISimpleTensor : public IAITensor
{
 public:
    AISimpleTensor();

    ~AISimpleTensor();

    /*
    * @brief 获取类型名称
    */
    virtual const char* const GetTypeName() override;

   /*获取数据地址*/
    void* GetBuffer();

    /*
    @brief 设置数据地址
    @param [in] data 数据指针
    @param [in] size 大小
    @param [in] isown 是否在tensor生命周期结束后，由tensor释放data地址内存
    */
    void SetBuffer(void *data, const int32_t size, bool isown=false);

    /*
    @brief 获取数据占用空间大小
    */
    uint32_t GetSize();

    /*
    @brief 分配数据空间
    */
    void* MallocDataBuffer(uint32_t size);

    /*
    * @brief  获取序列化后字节长度
    */
    virtual uint32_t ByteSizeLong() override;

    /*
    * @brief 通过AITensorDescription设置参数
    * @param [in] tensor_desc tensor描述
    * @return true: init成功
    * @return false: init失败，失败的可能原因比如tensor_desc与当前tensor不符
    */
    virtual bool Init(const AITensorDescription &tensor_desc) override;

#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief 序列化到buffer，用于数据跨进程交互
    * @param [in] buffer 序列化输出的内存区域地址，由调用方分配
    * @param [in] size 输出内存区域的大小
    * @return SUCCESS: 成功
    *        FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    virtual AIStatus SerializeTo(void* buffer, const uint32_t size) override;

    /*
    * @brief 从buffer反序列化回tensor，用于数据跨进程交互
    * @param [in] buffer 输入的内存区域地址
    * @param [in] size 输入内存区域的大小
    * @return SUCCESS: 成功
    *        FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) override;
#endif
```

```
    /*
    @brief 构建tensor描述
    */
    static AITensorDescription BuildDescription(const std::string& size = "");

    /*
    @brief 根据描述创建tensor
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    bool IsSupportZerocpy();
    /*
    @brief 设置数据地址
    @param [in] data 数据指针
    @param [in] size 大小
    @param [in] isown 是否在tensor生命周期结束后，由tensor释放data地址内存
    @param [in] is_data_support_mem_share_ 是否在tensor生命周期结束后，由tensor释放data地址内存
    */
    void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share) override;

private:
    void* data_;
    uint32_t size_;
    bool isowner_ ;

    static std::string type_name_;
}
```

8.2.4.4.16 AINeuralNetworkBuffer

NN通用Buffer定义。该类在ai_tensor.h中定义。

```
class AINeuralNetworkBuffer : public AISimpleTensor
{
public:
    AINeuralNetworkBuffer()
    {
        data_type_ = 0;
        number_    = 1;
        channel_   = 1;
        height_    = 1;
        width_     = 1;
        name_      = "";
    };

    ~AINeuralNetworkBuffer() {};
    /*
    * @brief 获取类型名称
    */
    const char* const GetTypeName();

    /*
    @brief 获取size字节大小
    */
    uint32_t ByteSizeLong();

    /*
    @brief 初始化
    @param [in] tensor_desc tensor描述
    */
    bool Init(const AITensorDescription &tensor_desc);

#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief 序列化到buffer，用于数据跨进程交互
    * @param [in] buffer 序列化输出的内存区域地址，由调用方分配
    * @param [in] size 输出内存区域的大小
    * @return SUCCESS: 成功
```

```
    *       FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    AIStatus SerializeTo(void* buffer, uint32_t size_);

    /*
    * @brief 从buffer反序列化回tensor，用于数据跨进程交互
    * @param [in] buffer 输入的内存区域地址
    * @param [in] size 输入内存区域的大小
    * @return SUCCESS: 成功
    *       FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
    /*
    * @brief 获取描述信息
    */
    static AITensorDescription GetDescription(
        const std::string &size = "0",const std::string &data_type="0",
        const std::string &number="0", const std::string &channel="0",
        const std::string &height="0", const std::string &width="0");

    /*
    @brief 创建tensor
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    /*
    @brief 获取number数量
    */
    int32_t GetNumber();

    /*
    @brief 设置number
    */
    void SetNumber(int32_t number);

    /*
    @brief 获取channel数量
    */
    int32_t GetChannel();

    /*
    @brief 设置channel
    */
    void SetChannel(int32_t channel);

    /*
    @brief 获取height
    */
    int32_t GetHeight();

    /*
    @brief 设置height
    */
    void SetHeight(int32_t height);

    /*
    @brief 获取width
    */
    int32_t GetWidth();

    /*
    @brief 设置width
    */
    void SetWidth(int32_t width);

    /*
    @brief 获取数据类型
    */
```

```
    int32_t GetData_type();

    /*
    @brief 设置数据类型
    */
    void SetData_type(int32_t data_type);

    /*
    @brief 获取数据类型
    */
    const std::string& GetName() const;

    /*
    @brief 设置数据类型
    */
    void SetName(const std::string& value);

private:
    int32_t data_type_;
    int32_t number_;
    int32_t channel_;
    int32_t height_;
    int32_t width_;
    std::string name_;
};
```

8.2.4.4.17 AIImageBuffer

图像的通用Buffer定义。该类在ai_tensor.h中定义。

```
class AIImageBuffer : public AISimpleTensor
{
public:
    AIImageBuffer()
    {
        format_ = JPEG;
        width_  = 0;
        height_ = 0;
    };

    /*
    * @brief 获取类型名称
    */
    const char* const GetTypeName();

    /*
    @brief 获取size字节大小
    */
    uint32_t ByteSizeLong();

    /*
    @brief 初始化
    @param [in] tensor_desc tensor描述
    */
    bool Init(const AITensorDescription &tensor_desc);

#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief 序列化到buffer，用于数据跨进程交互
    * @param [in] buffer 序列化输出的内存区域地址，由调用方分配
    * @param [in] size 输出内存区域的大小
    * @return SUCCESS: 成功
    *        FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    AIStatus SerializeTo(void* buffer, uint32_t size_);

    /*
    * @brief 从buffer反序列化回tensor，用于数据跨进程交互
    * @param [in] buffer 输入的内存区域地址
```

```
    * @param [in] size 输入内存区域的大小
    * @return SUCCESS: 成功
    *        FAILED: 失败，如果该Tensor不支持跨进程，则不需要实现此接口，直接返回失败
    */
    AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
    /*
    * @brief 获取描述信息
    */
    static AITensorDescription GetDescription(
        const std::string &size = "0",const std::string &height="0",
        const std::string &width="0", const std::string format="JPEG");

    /*
    @brief 创建tensor
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    /*
    @brief 获取图像格式
    */
    ImageFormat GetFormat();
    /*
    @brief 设置图像格式
    */
    void SetFormat(ImageFormat format);

    /*
    @brief 获取height
    */
    int32_t GetHeight();

    /*
    @brief 设置height
    */
    void SetHeight(int32_t height);

    /*
    @brief 获取width
    */
    int32_t GetWidth();

    /*
    @brief 设置width
    */
    void SetWidth(int32_t width);
  private:
    ImageFormat format_;
    int32_t width_;
    int32_t height_;
  };
```

8.2.4.4.18 HIAILog

HiAI的日志类。该类在log.h中定义。

```
  class HIAILog {
  public:
    /**
    * @ingroup hiaiengine
    * @brief 获取HIAILog指针
    */
    HIAI_LIB_INTERNAL static HIAILog* GetInstance();

    /**
    * @ingroup hiaiengine
    * @brief : 是否初始化
    * @return : 是否初始化 true 是
    */
```

```
    HIAI_LIB_INTERNAL bool IsInit() { return isInitFlag; }

    /**
    * @ingroup hiaiengine
    * @brief 获取log输出级别
    * @return :log输出级别
    */
    HIAI_LIB_INTERNAL uint32_t HIAIGetCurLogLevel();

    /**
    * @ingroup hiaiengine
    * @brief 判断该条log是否需要输出
    * @param [in]errorCode:消息的错误码
    * @return 是否能被输出
    */
    HIAI_LIB_INTERNAL bool HIAILogOutputEnable(const uint32_t errorCode);

    /**
    * @ingroup hiaiengine
    * @brief log对应的级别名称
    * @param [in]logLevel:宏定义的log级别
    */
    HIAI_LIB_INTERNAL std::string HIAILevelName(const uint32_t logLevel);

    /**
    * @ingroup hiaiengine
    * @brief 输出log
    * @param [in]moudleID:enum定义的组件id,
    * @param [in]logLevel:宏定义的log级别,
    * @param [in]strLog:输出的日志内容
    */
    HIAI_LIB_INTERNAL void HIAISaveLog(const int32_t moudleID,
        const uint32_t logLevel, const char* strLog);
  protected:
    /**
    * @ingroup hiaiengine
    * @brief HIAILog构造函数
    */
    HIAI_LIB_INTERNAL HIAILog();
    HIAI_LIB_INTERNAL ~HIAILog() {}
  private:
    /**
    * @ingroup hiaiengine
    * @brief HIAILog初始化函数
    */
    HIAI_LIB_INTERNAL void Init();
    HIAI_LIB_INTERNAL HIAILog(const HIAILog&) = delete;
    HIAI_LIB_INTERNAL HIAILog(HIAILog&&) = delete;
    HIAI_LIB_INTERNAL HIAILog& operator=(const HIAILog&) = delete;
    HIAI_LIB_INTERNAL HIAILog& operator=(HIAILog&&) = delete;
  private:
    HIAI_LIB_INTERNAL static std::mutex mutexHandle;
    uint32_t outputLogLevel;
    std::map<uint32_t, std::string> levelName;
    std::map<std::string, uint32_t> levelNum;
    static bool isInitFlag;
};
```

8.2.4.4.19 HIAI_ENGINE_LOG

日志封装宏，对HIAI_ENGINE_LOG_IMPL宏进行封装。该宏在log.h中定义。

```
#define HIAI_ENGINE_LOG(...) \
    HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
```

8.2.4.5 异常处理

当调用方出现调用离线模型管家接口失败，也即调用该函数返回值为FAILED时，可通过Mind Studio界面的Log窗口查看日志。根据Time列的时间查看最新日志，并根据日志的提示排查异常调用错误。

示例：调用方在使用Init接口时，加载的模型不存在：

```
Init:hiaiengine/node/ai_model_manager_impl.cpp:163:"Load models failed!"
```

说明

关于日志查看的详细操作，可参见《Ascend 310 Mind Studio开发辅助工具》中的“日志工具 > 基本操作 > 日志查看”章节。

8.2.5 辅助接口

8.2.5.1 数据获取接口（C++语言）

通过以下辅助接口可以获取相关参数。数据获取的接口大部分为C语言与C++通用的接口，C++专用的接口在函数格式中进行了说明。

8.2.5.1.1 获取 Device 数目

获取device个数。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_GetDeviceNum(uint32_t *devCount);

参数说明

参数	说明	取值范围
devCount	输出device个数的指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVNUM_ERROR	get device number error

调用示例

```
uint32_t dev_count;
HIAI_GetDeviceNum(&dev_count);
```

8.2.5.1.2 获取第一个 DeviceID

获取第一个device的id。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_GetFirstDeviceId(uint32_t *firstDevID);

参数说明

参数	说明	取值范围
firstDevID	第一个device的id的指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVID_ERROR	get device id error

调用示例

```
uint32_t first_dev_id;
HIAI_GetFirstDeviceId(&first_dev_id);
```

8.2.5.1.3 获取第一个 GraphID

获取GraphID。该接口在c_graph.h中定义。

函数格式

C语言与C++通用接口：**HIAI_StatusT HIAI_GetFirstGraphId(uint32_t *firstGraphID);**

参数说明

参数	说明	取值范围
firstGraphID	第一个graph的id指针。	-

返回值

C语言与C++通用接口：错误码。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_GRAPHID_ERROR	get graph id error

调用示例

```
uint32_t first_graph_id;
HIAI_GetFirstGraphId(&first_graph_id);
```

8.2.5.1.4 获取下一个 DeviceID

获取下一个device的id。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_GetNextDeviceId(const uint32_t curDevID,uint32_t *nextDevID);

参数说明

参数	说明	取值范围
curDevID	当前device的id号。	-
nextDevID	下一个device的id号的指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVID_ERROR	get device id error

调用示例

```
uint32_t cur_dev_id = 1;
uint32_t next_dev_id;
HIAI_GetNextDeviceId(cur_dev_id, &next_dev_id);
```

8.2.5.1.5 获取下一个 GraphID

获取下一个GraphID。该接口在c_graph.h中定义。

函数格式

C语言与C++通用接口：**HIAI_StatusT HIAI_GetNextGraphId(const uint32_t curGraphID,uint32_t *nextGraphID);**

参数说明

参数	说明	取值范围
curGraphID	当前graph id。	-
nextGraphID	下一个graph的id指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_GRAPHID_ERROR	get graph id error

调用示例

```
uint32_t cur_graph_id = 1;
uint32_t next_graph_id;
HIAI_GetNextGraphId(cur_graph_id, &next_graph_id);
```

8.2.5.1.6 获取 Engine 指针

根据engineID查找Engine对象指针。该接口在graph.h中定义。

函数格式

std::shared_ptr<Engine> Graph::GetEngine(uint32_t engineID);

参数说明

参数	说明	取值范围
engineID	目标Engine的id	-

返回值

Engine的智能指针。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
```

8.2.5.1.7 获取 Graph 的 GraphId

获取当前graph的id。该接口在graph.h中定义。

函数格式

uint32_t Graph::GetGraphId();

返回值

graph的id。

8.2.5.1.8 获取 Graph 的 DeviceID

获取当前graph在device侧的id。该接口在graph.h中定义。

函数格式

uint32_t Graph::GetDeviceID();

返回值

当前graph在device侧的id。

调用示例

```
auto graphPtr = Graph::GetInstance(100);
auto deviceId = graphPtr->GetDeviceID();
```

8.2.5.1.9 获取 Engine 的 GraphId

获取Engine的GraphId。该接口在engine.h中定义。

函数格式

uint32_t Engine::GetGraphId();

返回值

Engine的GraphId。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto graphID= enginePtr->GetGraphId();
```

8.2.5.1.10 获取 Engine 队列最大大小

获取Engine的Queue最大Size。该接口在engine.h中定义。

函数格式

const uint32_t Engine::GetQueueMaxSize();

返回值

该Engine的Queue最大Size。

调用示例

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueMaxSize();
```

8.2.5.1.11 获取 Engine 指定端口当前队列大小

获取Engine的某端口号Queue的Size。该接口在engine.h中定义。

函数格式

const uint32_t Engine::GetQueueCurrentSize(const uint32_t portID);

参数说明

参数	说明	取值范围
portID	端口号	-

返回值

该Engine的portID端口号Queue的Size。

调用示例

```
uint32_t engineID= 1000;
uint32_t portID = 0;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueCurrentSize(portID);
```

8.2.5.1.12 解析 Matrix 配置文件

解析配置文件。Matrix解析配置文件，并将生成的Graph回写到list中，供用户使用。该接口在graph.h中定义。

函数格式

static HIAI_StatusT Graph::ParseConfigFile(const std::string& configFile, GraphConfigList& graphConfigList)

参数说明

参数	说明	取值范围
configFile	配置文件路径。 请确保传入的文件路径是正确路径。	-
graphConfigList	Protobuf数据格式。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	-	HIAI_GRAPH_GET_INSTANCE_NULL	-

8.2.5.1.13 获取 PCIe 的 Info

获取PCIe（Peripheral Component Interconnect Express）的信息。该接口在c_graph.h中定义。

函数格式

static HIAI_StatusT HIAI_GetPCIeInfo(const uint32_t devId,int32_t* bus, int32_t* dev, int32_t* func);

参数说明

参数	说明	取值范围
devId	需要查询的devId	-
bus	返回PCIe总线号	-
dev	返回PCIe设备号	-
func	返回PCIe功能号	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码

序号	错误码级别	错误码	错误码描述
1	HIAI_INFO	HIAI_OK	running ok
2	HIAI_ERROR	HIAI_GRAPH_GET_PCIEINFO_ERROR	bus,dev,func中有空指针，或者get pcie info error

调用示例

```
uint32_t devId= 1;
uint32_t bus;
uint32_t dev;
uint32_t func;
HIAI_GetPCIeInfo(devId, &bus, &dev, &func);
```

8.2.5.1.14 获取版本号

获取API版本信息。该接口在c_graph.h中定义。

函数格式

HIAI_API_VERSION HIAI_GetAPIVersion();

返回值

版本信息的enum信息。

调用示例

```
HIAI_API_VERSION HIAI_GetAPIVersion();
```

8.2.5.1.15 获取 OamConfig 智能指针

获取OamConfig智能指针，用于获取oam配置信息，包括模型名称、是否打开Dump开关、需要Dump模型中哪些层的算子信息。该接口在engine.h文件中定义。

函数格式

OAMConfigDef* GetOamConfig();

参数说明

无。

返回值

OamConfig智能指针。OAMConfigDef类型的定义请参见**8.2.7.2.1 Protobuffer数据类型**。

8.2.5.2 数据类型序列化和反序列化（C++语言）

为用户自定义的各种数据类型提供自动化序列化和反序列化机制。

其中相关宏封装用到了以下接口：

```
static HIAIDataTypeFactory* HIAIDataTypeFactory::GetInstance();
```

8.2.5.2.1 宏:HIAI_REGISTER_DATA_TYPE

为用户自定义的数据结构类型提供自动化序列化和反序列化机制。该宏在data_type_reg.h中定义。

说明

- 该接口需要在host和Device端同时注册。
- 因为cereal的限制， 结构体中应避免使用long double类型，为了保证跨平台（如Windows到Linux）运行， 结构体成员尽量使用可移植的类型，如 int32_t等;
- 因为cereal的限制， 结构体中需使用共享指针。

函数格式

HIAI_REGISTER_DATA_TYPE(name, type)

参数说明

参数	说明	取值范围
name	用户自定义的数据结构类型名字（不同的数据类型要保证名字唯一）。	-
type	用户自定义的数据结构类型。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NULL	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIST	hiai engine function is existed

8.2.5.2.2 宏:HIAI_REGISTER_TEMPLATE_DATA_TYPE

为用户自定义的模板的数据结构类型提供自动化序列化和反序列化机制。该宏在data_type_reg.h中定义。

说明

该接口需要在host和Device端同时注册该类。

函数格式

HIAI_REGISTER_TEMPLATE_DATA_TYPE(name, type, basictype1, basictype2, …)

参数说明

参数	说明	取值范围
name	用户自定义的数据结构类型名字（不同的数据结构类型要保证名字唯一）。	-
type	用户自定义的模板的数据结构类型。	-
basicType1	用户自定义的数据类型。	-
basicType2	用户自定义的数据类型。	-
…	…	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NULL	hiai engine function is null

序号	错误码	错误码描述
3	HIAI_ENGINE_FUNCTOR_EXIST	hiai engine function is existed

8.2.5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC

为用户自定义的数据类型提供自定义的序列化和反序列化机制。该宏在data_type_reg.h中定义。

即发送时将用户传输的结构体指针转化为结构体buffer和数据buffer，接受数据时，将从框架获取的结构体buffer和数据buffer反转为结构体。

使用场景：

用于从host侧到Device侧快速搬运数据，如果用户需要提升传输性能，必须使用该接口注册函数。

说明

- 该接口用于从host侧到Device侧快速搬运数据使用。
- 如果用户想使用高性能传输接口，必须使用该注册函数注册序列化和反序列化函数。

函数格式

HIAI_REGISTER_SERIALIZE_FUNC(name, type, hiaiSerializeFunc, hiaiDeSerializeFunc)

参数说明

参数	说明	取值范围
name	注册的消息名字。	-
type	自定义数据结构的类型。	-
hiaiSerializeFunc	序列化函数。函数形式为 typedef void(*__hiaiSerializeFunc__)(void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t& dataLen); 各参数说明如下： ● inputPtr：结构体指针 ● ctrlStr：结构体buffer ● dataPtr：结构体数据指针buffer ● dataLen：结构体数据大小	-

参数	说明	取值范围
hiaiDeSerializeFunc	反序列化函数。函数形式为 typedef std::shared_ptr<void>(***hiaiDeSerializeFunc**)(const char* ctrlPtr, const uint32_t& ctrlLen, const uint8_t* dataPtr, const uint32_t& dataLen); 各参数说明如下： ● ctrlPtr：结构体指针 ● ctrlLen：数据结构中控制信息大小 ● dataPtr：结构体数据指针 ● dataLen：结构中数据信息存储空间大小，仅用于校验，不表示原始数据信息大小	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_ENGINE_FUNCTOR_NULL	hiai engine function is null
3	HIAI_ENGINE_FUNCTOR_EXIST	hiai engine function is existed

8.2.5.2.4 Graph::ReleaseDataBuffer

在数据进行反序列化过程中，数据内存赋值给智能指针时，将本函数注册为删除器。该接口在graph.h中定义。

函数格式

static void Graph::ReleaseDataBuffer(void* ptr)

参数说明

参数	说明	取值范围
ptr	内存指针	-

8.2.5.2.5 接口使用示例

```
/*****************************************************************************
 *                    CopyRight (C) Hisilicon Co., Ltd.
 *
 *      Filename:  user_def_datatype.h
 *   Description:  User defined data type
 *
 *       Version:  1.0
 *       Created:  2018-01-08 10:15:18
 *        Author:
 *
 *      Revision:  initial draft;
 *****************************************************************************/
#ifndef USE_DEF_DATA_TYPE_H_
#define USE_DEF_DATA_TYPE_H_

#ifdef __cplusplus
#include <vector>
#include <map>
#endif

// 用户自定义的数据类型（C 语言风格）
typedef struct UseDefDataType
{
    int data1;
    float data2;
}UseDefDataTypeT;

#ifdef __cplusplus
// 用户自定义的数据类型（带模板类型）
template<typename T1, typename T2, typename T3>
class UseDefTemplateDataType
{
  public:
    std::vector<T1> data_vec_;
    std::map<T2, T3> data_map_;
};
#endif

#endif
/*****************************************************************************
 *                    CopyRight (C) Hisilicon Co., Ltd.
 *
 *      Filename:  use_def_data_type_reg.cpp
 *   Description:  User defined Data Type Register
 *
 *       Version:  1.0
 *       Created:  2018-01-08 10:15:18
 *        Author:  h00384043
 *
 *      Revision:  initial draft;
 *****************************************************************************/

#include "use_def_data_type.h"
#include "hiaiengine/data_type_reg.h"

// 用户自定义的数据类型必须先注册到Matrix，才能作为Engine之间的通信接口正常使用
template<class Archive>
void serialize(Archive& ar, UseDefDataTypeT& data)
```

```
{
    ar(data.data1, data.data2);
}

// 注册UseDefDataTypeT
HIAI_REGISTER_DATA_TYPE("UseDefDataTypeT", UseDefDataTypeT)


// 模板类型数据的注册
// 对于模板类型，对于所有需要使用的类型必须都注册
template<class Archive, typename T1, typename T2, typename T3>
void serialize(Archive& ar, UseDefTemplateDataType<T1, T2, T3>& data)
{
    ar(data.data_vec_, data.data_map_);
}

// 注册 UseDefTemplateDataType<int, float>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_float_string",
UseDefTemplateDataType, uint64_t, float, std::string);

//... 注册其他类型

// 注册 UseDefTemplateDataType<int, int>
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
UseDefTemplateDataType, uint64_t, uint64_t, uint64_t);
// 注册结构体的序列化和反序列化
typedef struct tagST_SC_IMAGE
{
    INT32  iWidth;                  // image width
    INT32  iHeight;                 // image height
    INT32  iWidthStep;                // Size of aligned image row in bytes
    INT32  iChannel;                 // channels
    INT32  iDepth;                  // depth in bits
    UINT32 uiTimeStampH;
    UINT32 uiTimeStampL;
    UINT32 iSize;
    UINT64 uiID;
    EN_SC_IMAGE_TYPE eType;           // image type
    ST_SC_IMAGE_ROI *roi;            // Image ROI. If NULL, the whole image is selected
    UINT8*  pucImageData;             // Image data
    UINT64 uiPhyAddr;
}ST_SC_IMAGE;

std::shared_ptr<ST_SC_IMAGE> st_image = std::make_shared<ST_SC_IMAGE>();
st_image->iWidth = 1080;
st_image->iHeight = 1080;
st_image->iChannel = 1;
st_image->iWidthStep = 1080;
st_image->iSize = 1080*1080*3/2 ;
st_image->eType = SC_IMAGE_YUV420SP;
st_image->roi = (ST_SC_IMAGE_ROI*)malloc(sizeof(ST_SC_IMAGE_ROI));
st_image->pucImageData = nullptr;
uint8_t* align_buffer = nullptr;
HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(st_image->iSize,(void*&)align_buffer,10000);
hiai::EnginePortID engine_id;
engine_id.graph_id = 1;
engine_id.engine_id = 1;
engine_id.port_id = 0;
HIAI_SendData(engine_id, "ST_SC_IMAGE", std::static_pointer_cast<void>(st_image),100000);

void GetStScImageSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t& dataLen)

{
    // 如果结构体内有除了image之外的多个指针， 则进行拼接
    ST_SC_IMAGE* st_image = (ST_SC_IMAGE*)inputPtr;

    ctrlStr = std::string((char*)inputPtr, sizeof(ST_SC_IMAGE));
    if(nullptr != st_image->roi)
    {
```

```
        std::string image_roi_str = std::string((char*)st_image->roi, sizeof(ST_SC_IMAGE_ROI));
        ctrlStr += image_roi_str;
    }

    dataPtr = (UINT8*)st_image->pucImageData;
    dataLen= st_image->iSize;
}

std::shared_ptr<void> GetStScImageDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen, const uint8_t*
dataPtr, const uint32_t& dataLen)
{
    ST_SC_IMAGE* st_sc_image = (ST_SC_IMAGE*)ctrlPtr;
    std::shared_ptr<hiai::BatchImagePara<uint8_t>> ImageInfo(new hiai::BatchImagePara<uint8_t>);
    hiai::ImageData<uint8_t> image_data;
    image_data.width = st_sc_image->iWidth;
    image_data.height = st_sc_image->iHeight;
    image_data.channel = st_sc_image->iChannel;
    image_data.width_step = st_sc_image->iWidthStep;
    if (st_sc_image->eType == SC_IMAGE_U8C3PLANAR)
    {
        image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
        image_data.format = hiai::BGR888;
    }
    else if (SC_IMAGE_U8C3PACKAGE == st_sc_image->eType)
    {
        image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
        image_data.format = hiai::BGR888;
    }
    else if (st_sc_image->eType == SC_IMAGE_YUV420SP)
    {
        image_data.size = st_sc_image->iSize;//st_sc_image->iWidth * st_sc_image->iHeight * 3 / 2;
        image_data.format = hiai::YUV420SP;
    }
    image_data.data.reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
    ImageInfo->v_img.push_back(image_data);
    ImageInfo->b_info.frame_ID.push_back(0);
    ImageInfo->b_info.batch_size = ImageInfo->b_info.frame_ID.size();

    return std::static_pointer_cast<void>(ImageInfo);
}

HIAI_REGISTER_SERIALIZE_FUNC("ST_SC_IMAGE", ST_SC_IMAGE, GetStScImageSearPtr,
GetStScImageDearPtr);
```

8.2.5.3 内存管理（C 语言）

接口使用要点

- 通过HIAI_DMalloc接口申请内存，关于内存管理，**请注意以下要点**：
  - 申请自动释放内存，用于跨侧数据传输时，如果是智能指针，由于Matrix框架自动释放内存，所以智能指针指定的析构器必须是空的；如果非智能指针，则Matrix框架自动释放。
  - 申请手动释放内存，用于跨侧数据传输时，如果是智能指针，则需要指定析构器为HIAI_DFree；如果非智能指针，则数据发送完成后需要调用HIAI_DFree释放内存。
  - 申请自动释放内存时，调用SendData接口用于跨侧数据传输后，不允许读写内存中的数据。
  - 申请手动释放内存时，如果调用SendData接口用于跨侧数据传输时，在内存释放前，不允许修改内存中的数据，但是可以读取内存中的数据；如果调用SendData接口用于同侧数据传输时，在内存释放前，可以读写内存中的数据。

  - 申请自动释放内存，对于该内存中的数据，不允许多次调用SendData接口发送数据。
  - 申请内存后，调用SendData接口用于跨侧数据传输时，只能从起始地址处发送数据，不可截取该内存中的某一段数据发送，否则会失败。
- 如果调用HIAI_DVPP_MAlloc接口申请内存，用于Device到Host的数据传输时，由于HIAI_DVPP_MAlloc没有自动释放标签，所以一定需要调用HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址， 必须指定析构器为HIAI_DVPP_DFree。

说明

跨侧数据传输是指Host->Device或Device->Host的数据传输。

同侧数据传输是指Host->Host或Device->Device的数据传输。

8.2.5.3.1 HIAI_DMalloc

通过Matrix接口申请Host侧或Device侧的内存块，配合高效数据传输使用。该接口在c_graph.h中定义。

若要给Device端分配DVPP使用的内存，您需要参考**8.2.5.3.3 HIAI_DVPP_DMalloc**。

使用场景：

通过Matrix框架，将大数据从Host端搬运到Device端，该接口需要配合**8.2.5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC**使用。

例如：需要将1080P或者4K图像发送到Device端，如果需要提升传输性能，则必须通过HIAI_REGISTER_SERIALIZE_FUNC注册结构体转换和反转函数，另外，大数据块内存则通过HIAI_DMalloc接口申请内存，使用该方式，传输性能将得到很大提升。

说明

该接口主要用于Host与Device的搬运大数据的性能问题，如无性能要求或非传输场景，不建议使用本接口。

函数格式

void* HIAI_DMalloc (const uint32_t dataSize, const uint32_t timeOut, uint32_t flag)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	● 如果调用该接口申请Host侧内存，则取值范围是：0~256M Bytes，不包含0。推荐使用256K Bytes~256M Bytes。 ● 如果调用该接口申请Device侧内存，则取值范围是：0~(256M Bytes-96 Bytes)，不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。Device侧模型管家会占用96 Bytes。
timeOut	当内存申请失败时，提供时延进行阻塞等待有空余内存，默认值为500毫秒。	-

flag	● MEMORY_ATTR_AUTO_FREE：默认值， 如果设置了此值，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAI_DFree，程序运行结束后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端，则需要调用HIAI_DFree释放内存。
	● MEMORY_ATTR_MANUAL_FREE：如设置此值，表示无论何种情况，必须调用HIAI_DFree释放内存。
	● MEMORY_ATTR_NONE和MEMORY_ATTR_MAX当前未使用。
		typedef enum {
		MEMORY_ATTR_NONE = 0,
		//框架自行释放DMalloc内存
		MEMORY_ATTR_AUTO_FREE = (0x1 << 1),
		//需要手动调用DFree释放内存
		MEMORY_ATTR_MANUAL_FREE = (0x1 << 2),
		MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

返回值

使用HIAI_DMalloc接口申请到的内存地址。如果申请失败，则返回空指针。

8.2.5.3.2 HIAI_DFree

释放通过HiAI接口申请的内存。该接口在c_graph.h中定义。

使用场景：

当用户通过**8.2.5.3.1 HIAI_DMalloc**申请内存后并且没有调用SendData接口发送数据，必须通过该接口释放内存，如果已经调用SendData接口，则不需要调用该接口释放内存。

函数格式

HIAI_StatusT HIAI_DFree (void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存指针。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEMORY	can't find the memory

8.2.5.3.3 HIAI_DVPP_DMalloc

用于分配内存， 该接口主要用于分配内存给Device端的DVPP使用。该接口在c_graph.h中定义。调用该接口申请内存后，必须使用HIAI_DVPP_DFree接口释放内存。

函数格式

void* HIAI_DVPP_DMalloc(const uint32_t dataSize)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	0~(256M Bytes-96 Bytes)，不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。 Device侧模型管家会占用96 Bytes。

返回值

使用HIAI_DVPP_DMalloc接口申请到的内存地址。如果申请失败，则返回空指针。

使用说明

HIAI_DVPP_DMalloc是专门为了给DVPP分配内存而新增的一个接口，通过该接口申请的内存能够满足DVPP对内存的各种限制及要求。

如果您在旧版本中使用HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)方式为DVPP申请内存，则需要修改为使用HIAI_DVPP_DMalloc接口申请内存。关于HIAI_DMalloc接口，请参见**8.2.5.3.1 HIAI_DMalloc**。

使用范例：

```
// 使用HIAI_DVPP_DMalloc申请内存
uint32_t allocSize = 1000;
char* allocBuffer = (char*)HIAI_DVPP_DMalloc(allocSize);
if (allocBuffer == nullptr) {
    // 分配失败，进行异常处理
}
```

8.2.5.3.4 HIAI_DVPP_DFree

用于释放内存， 该接口主要用于释放HIAI_DVPP_DMalloc接口分配的内存。该接口在c_graph.h中定义。

函数格式

HIAI_StatusT HIAI_DVPP_DFree(void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存地址。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

版权所有 © 华为技术有限公司

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEMORY	can't find the memory

8.2.5.4 内存管理（C++语言）

接口使用要点

- 通过HIAIMemory::HIAI_DMalloc接口申请内存，关于内存管理，**请注意以下要点**：
  - 申请自动释放内存，用于跨侧数据传输时，如果是智能指针，由于Matrix框架自动释放内存，所以智能指针指定的析构器必须是空的；如果非智能指针，则Matrix框架自动释放。
  - 申请手动释放内存，用于跨侧数据传输时，如果是智能指针，则需要指定析构器为HIAIMemory::HIAI_DFree；如果非智能指针，则数据发送完成后需要调用HIAIMemory::HIAI_DFree释放内存。
  - 申请自动释放内存时，调用SendData接口用于跨侧数据传输后，不允许读写内存中的数据。
  - 申请手动释放内存时，如果调用SendData接口用于跨侧数据传输时，在内存释放前，不允许修改内存中的数据，但是可以读取内存中的数据；如果调用SendData接口用于同侧数据传输时，在内存释放前，可以读写内存中的数据。
  - 申请自动释放内存，对于该内存中的数据，不允许多次调用SendData接口发送数据。
  - 申请内存后，调用SendData接口用于跨侧数据传输时，只能从起始地址处发送数据，不可截取该内存中的某一段数据发送，否则会失败。
- 如果调用HIAIMemory::HIAI_DVPP_DMalloc接口申请内存，用于Device到Host的数据传输时，由于HIAIMemory::HIAI_DVPP_DMalloc没有自动释放标签，所以一定需要调用HIAIMemory::HIAI_DVPP_DFree接口手动释放内存。如果使用智能指针存放申请的内存地址， 必须指定析构器为HIAIMemory::HIAI_DVPP_DFree。

说明

跨侧数据传输是指Host->Device或Device->Host的数据传输。

同侧数据传输是指Host->Host或Device->Device的数据传输。

8.2.5.4.1 HIAIMemory::HIAI_DMalloc

通过Matrix接口申请Host侧或Device侧的内存块，配合高效数据传输使用。该接口在ai_memory.h中定义。

若要给Device端分配DVPP使用的内存，您需要参考**8.2.5.4.4 HIAIMemory::HIAI_DVPP_DMalloc**。

使用场景：

通过Matrix框架，将大数据从Host端搬运到Device端，该接口需要配合**8.2.5.2.3 宏:HIAI_REGISTER_SERIALIZE_FUNC**使用。

例如：需要将1080P或者4K图像发送到Device端，如果需要提升传输性能，则必须通过HIAI_REGISTER_SERIALIZE_FUNC注册结构体转换和反转函数，另外，大数据块内存则通过HIAI_DMalloc接口申请内存，使用该方式，传输性能将得到很大提升。

说明

- 该接口主要用于Host与Device的搬运大数据的性能问题，不推荐当做普通的malloc使用。
- 出于性能考虑，该接口会预申请内存，实际占用内存大小与申请内存大小存在一定差异，该接口的运行时间也会存在波动。

函数格式

static HIAI_StatusT HIAIMemory::HIAI_DMalloc (const uint32_t dataSize, void*& dataBuffer, const uint32_t timeOut = MALLOC_DEFAULT_TIME_OUT, uint32_t flag = MEMORY_ATTR_AUTO_FREE)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	• 如果调用该接口申请Host侧内存，则取值范围是：0~256M Bytes，不包含0。推荐使用256K Bytes~256M Bytes。 • 如果调用该接口申请Device侧内存，则取值范围是：0~(256M Bytes-96 Bytes)，不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。Device侧模型管家会占用96 Bytes。
dataBuffer	内存指针。	-
timeOut	当内存申请失败时，提供时延进行阻塞等待有空余内存，默认值为MALLOC_DEFAULT_TIME_OUT（表示500毫秒）。	-

参数	说明	取值范围

flag	● MEMORY_ATTR_AUTO_FREE：默认值， 如果设置了此值，表示如果分配了内存，且通过SendData接口发送数据到对端，则无需调用HIAIMemory::HIAI_DFree，程序运行结束后，内存会自动释放；如果分配了内存，但没有通过SendData接口发送数据到对端，则需要调用HIAIMemory::HIAI_DFree释放内存。
	● MEMORY_ATTR_MANUAL_FREE：如设置此值，表示无论何种情况，必须调用HIAIMemory::HIAI_DFree释放内存。
	● MEMORY_ATTR_NONE和MEMORY_ATTR_MAX当前未使用。
		typedef enum {
		MEMORY_ATTR_NONE = 0,
		//框架自行释放DMalloc内存
		MEMORY_ATTR_AUTO_FREE = (0x1 << 1),
		//需要手动调用DFree释放内存
		MEMORY_ATTR_MANUAL_FREE = (0x1 << 2),
		MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	memory pool is not existed
4	HIAI_GRAPH_MALLOC_LARGER	failed to malloc buffer due to the size larger than 256M
5	HIAI_MEMORY_POOL_UPDATE_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL_INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

8.2.5.4.2 HIAIMemory::HIAI_DFree

释放通过HiAI接口申请的内存。该接口在ai_memory.h中定义。

使用场景：

当用户通过8.2.5.4.1 HIAIMemory::HIAI_DMalloc申请内存后并且没有调用SendData接口发送数据，必须通过该接口释放内存，如果已经调用SendData接口，则不需要调用该接口释放内存。

函数格式

static HIAI_StatusT HIAIMemory::HIAI_DFree (void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	需要释放的内存指针	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEMORY	can't find the memory

8.2.5.4.3 HIAIMemory:: IsDMalloc

用户可调用该接口判断内存是否是由DMalloc申请。该接口在ai_memory.h中定义。

函数格式

static bool IsDMalloc(const void* dataBuffer, const uint32_t& dataSize)

参数说明

参数	说明	取值范围
dataBuffer	需要判断的内存指针	-

参数	说明	取值范围
dataSize	内存地址的大小，单位是字节	-

返回值

true：由DMalloc申请内存。

false：不由DMalloc申请内存

8.2.5.4.4 HIAIMemory::HIAI_DVPP_DMalloc

用于分配内存， 该接口主要用于分配内存给Device端的DVPP使用。该接口在ai_memory.h中定义。调用该接口申请内存后，必须使用HIAIMemory::HIAI_DVPP_DFree接口释放内存。

函数格式

HIAI_StatusT HIAI_DVPP_DMalloc(const uint32_t dataSize, void*& dataBuffer)

参数说明

参数	说明	取值范围
dataSize	内存块大小。	0~(256M Bytes-96 Bytes)，不包含0。推荐使用(256K Bytes-96 Byte)~(256M Bytes-96 Bytes)。 Device侧模型管家会占用96 Bytes。
dataBuffer	分配出来的内存地址，如果分配失败，为nullptr。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	memory pool is not existed

版权所有 © 华为技术有限公司

序号	错误码	错误码描述
4	HIAI_GRAPH_MALLOC_LARGER	failed to malloc buffer due to the size larger than 256M
5	HIAI_MEMORY_POOL_UPDATE_FAILED	failed to update memory pool
6	HIAI_GRAPH_SENDMSG_FAILED	hdc send msg failed
7	HIAI_GRAPH_MEMORY_POOL_INITED	memory pool has inited
8	HIAI_GRAPH_NO_MEMORY	no memory

使用说明

HIAIMemory::HIAI_DVPP_DMalloc是专门为了给DVPP分配内存而新增的一个接口，通过该接口申请的内存能够满足DVPP对内存的各种限制及要求。

如果您在旧版本中使用HIAIMemory::HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)方式为DVPP申请内存，则需要修改为使用HIAIMemory::HIAI_DVPP_DMalloc接口申请内存。关于HIAIMemory::HIAI_DMalloc接口，请参见**8.2.5.4.1 HIAIMemory::HIAI_DMalloc**。

使用范例：

```
// 使用HIAI_DVPP_DMalloc申请内存
uint32_t allocSize = 1000;
char* allocBuffer  = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(allocSize, (void*&)allocBuffer);
if (ret != HIAI_OK || allocBuffer == nullptr) {
    // 分配失败，进行异常处理
}
```

8.2.5.4.5 HIAIMemory::HIAI_DVPP_DFree

用于释放内存， 该接口主要用于释放HIAIMemory::HIAI_DVPP_DMalloc接口分配的内存。该接口在ai_memory.h中定义。

函数格式

HIAI_StatusT HIAI_DVPP_DFree(void* dataBuffer)

参数说明

参数	说明	取值范围
dataBuffer	被释放的内存地址。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_GRAPH_NOT_EXIST	graph not exist
3	HIAI_GRAPH_INVALID_VALUE	graph invalid value
4	HIAI_GRAPH_NOT_FIND_MEMORY	can't find the memory

8.2.5.5 日志（C++语言）

8.2.5.5.1 错误码注册

?.1. 宏 HIAI_DEF_ERROR_CODE

注册错误码。该宏在status.h中定义。

本宏封装用到了以下函数：

```
static StatusFactory* StatusFactory::StatusFactory::GetInstance();
void RegisterErrorNo(const uint32_t err, const std::string& desc);
```

函数格式

HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)

参数说明

参数	说明	取值范围
moduleId	模块ID。	-
logLevel	错误级别。	-
codeName	错误码的名称。	-
codeDesc	错误码的描述，字符串。	-

返回值

无。

?.2. 接口使用示例

```
/*****************************************************************************************
*                    CopyRight (C) Hisilicon Co., Ltd.
*
*      Filename:  user_def_errorcode.h
*   Description:  User defined Error Code
```

```
 *
 *        Version:  1.0
 *        Created:  2018-01-08 10:15:18
 *         Author:
 *
 *       Revision:  initial draft;
 ****************************************************************************************/
#ifndef USE_DEF_ERROR_CODE_H_
#define USE_DEF_ERROR_CODE_H_

#include "hiaiengine/status.h"
#define USE_DEFINE_ERROR 0x6001   // 自定义日志模块ID，0x6001为自定义值，不能跟status.h文件中的
MODID_GRAPH等重复即可
enum
{
    HIAI_INVALID_INPUT_MSG_CODE = 0   // 错误码的名称，0为自定义值
};
HIAI_DEF_ERROR_CODE(USE_DEFINE_ERROR, HIAI_ERROR, HIAI_INVALID_INPUT_MSG, "invalid input
message pointer");
#endif
```

8.2.5.5.2 日志打印

通过HIAI_ENGINE_LOG来根据传入的参数调用不同的HIAI_ENGINE_LOG_IMPL函数。在调用格式化函数时，format中参数的类型与个数必须与实际参数类型一致。日志打印相关的接口在log.h中定义。

调用HIAI_ENGINE_LOG后，系统将日志记录日志文件中，日志文件在Host侧或开发者板的“/var”目录下。Device侧的日志被记录在文件名称以device-*id*开头的日志文件中，Host侧的日志被记录在文件名称以host-0开头的日志文件中。

?.1. 日志打印格式 1

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
graph	Graph对象指针	-
errorCode	错误码	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG（graph.get(), HIAI_OK, "RUNNING OK"）;
```

?.2. 日志打印格式 2

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
engine	Engine对象指针	-
errorCode	错误码	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
// 在HIAI_IMPL_ENGINE_PROCESS里调用此接口
HIAI_ENGINE_LOG（this, HIAI_OK, "RUNNING OK"）;
```

?.3. 日志打印格式 3

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
errorCode	错误码	-

参数	说明	取值范围
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
HIAI_ENGINE_LOG（HIAI_OK, "RUNNING OK"）;
```

?.4. 日志打印格式 4

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const char* format, ...);

参数说明

参数	说明	取值范围
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
HIAI_ENGINE_LOG（"RUNNING OK"）;
```

?.5. 日志打印格式 5

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
graph	Graph对象指针	-
errorCode	错误码	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG（MODID_OTHER, graph.get(), HIAI_OK, "RUNNING OK"）;
```

?.6. 日志打印格式 6

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID,hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
engine	Engine对象指针	-
errorCode	错误码	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
// 在HIAI_IMPL_ENGINE_PROCESS里调用此接口
HIAI_ENGINE_LOG（MODID_OTHER, this, HIAI_OK, "RUNNING OK"）;
```

?.7. 日志打印格式 7

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
HIAI_ENGINE_LOG（MODID_OTHER, "RUNNING OK"）;
```

?.8. 日志打印格式 8

函数格式

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID,const uint32_t errorCode, const char* format, ...);

参数说明

参数	说明	取值范围
moudleID	模块名枚举ID	-
errorCode	错误码	-
format	log描述	-
...	format中的可变参数，根据日志内容添加	-

调用示例

```
HIAI_ENGINE_LOG（MODID_OTHER, HIAI_OK, "RUNNING OK"）;
```

8.2.5.6 队列管理 MultiTypeQueue 接口（C++语言）

支持多种类型的消息队列保存。队列管理的相关接口在multitype_queue.h中定义。

8.2.5.6.1 MultiTypeQueue 构造函数

MultiTypeQueue构造函数。

函数格式

MultiTypeQueue(uint32_t queNum, uint32_t maxQueLen = 1024, uint32_t durationMs = 0);

参数说明

参数	说明	取值范围
queNum	指定队列个数。	-
maxQueLen	指定每个队列的最大长度。	-
durationMs	指定队列成员从入队到出队最大时间间隔。当某个成员从入队(Push)起，超过duration_ms时间后还未出队（Pop），则该成员会被自动删除。	-

返回值

无。

8.2.5.6.2 PushData

往指定队列插入数据。

函数格式

bool MultiTypeQueue::PushData(uint32_t qIndex, const std::shared_ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ queNum - 1
dataPtr	队列函数指针	-

返回值

成功则返回true，否则返回false（例如队列满等）。

8.2.5.6.3 FrontData

读取指定队列头部数据（并不从队列中删除）

函数格式

bool MultiTypeQueue::FrontData(uint32_t qIndex, std::shared_ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ queNum - 1
dataPtr	队列函数指针	-

返回值

成功则返回true，否则返回false（例如队列为空等）。

8.2.5.6.4 PopData

读取指定队列头部数据，并从队列中删除。

函数格式

bool MultiTypeQueue::PopData(uint32_t qIndex, std::shared_ptr<void>& dataPtr);

参数说明

参数	说明	取值范围
qIndex	队列编号	0 ~ que_num-1
dataPtr	队列函数指针	-

返回值

成功则返回true，否则返回false（例如队列为空等）。

8.2.5.6.5 PopAllData

PopAllData读取所有队列头部数据。仅当所有队列头部都有数据时才读取成功，并删除队列头部数据。否则，返回失败但并不删除任何数据。

函数格式

```
template<typename T1>
bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1);

template<typename T1, typename T2>
bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1,
std::shared_ptr<T2>& arg2);

......

template<typename T1, typename T2, typename T3, typename T4, typename
T5, typename T6, typename T7, typename T8, typename T9, typename T10,
typename T11, typename T12, typename T13, typename T14, typename T15,
typename T16>
bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1,
std::shared_ptr<T2>& arg2, std::shared_ptr<T3>& arg3,
std::shared_ptr<T4>& arg4, std::shared_ptr<T5>& arg5,
std::shared_ptr<T6>& arg6, std::shared_ptr<T7>& arg7,
std::shared_ptr<T8>& arg8, std::shared_ptr<T9>& arg9,
std::shared_ptr<T16>& arg16)
```

参数说明

参数	说明	取值范围
arg1	队列函数指针	-
...	队列函数指针	-
arg16	队列函数指针	-

返回值

仅当所有队列头部都有数据时才读取成功并返回true，否则返回false。

8.2.5.6.6 接口使用示例

见8.2.3.7 调用示例。

8.2.5.7 事件注册接口（C++语言）

8.2.5.7.1 Graph::RegisterEventHandle

用户调用RegisterEventHandle接口订阅感兴趣的事件，当前支持Host-Device断开连接事件，用户订阅该事件，当连接断开时，用户可接收到该事件，并在注册的回调函数中处理断开逻辑（如停掉主程序的等待）。该接口在graph.h中定义。

函数格式

HIAI_StatusT Graph::RegisterEventHandle(const HIAIEvent& event, const std::function<HIAI_StatusT(void)>&callBack)

参数说明

参数	说明	取值范围
event	订阅的事件	typedef enum { HIAI_DEVICE_DISCONNECT_EVENT // 断开消息 }HIAIEvent;
callBack	事件回调函数	用户订阅的回调函数

返回值

成功返回HIAI_OK，否则为失败。

8.2.5.8 其他（C++语言）

8.2.5.8.1 DataRecvInterface::RecvData

用户需要定义一个DataRecvInterface类的子类，初始化一个子类的对象，作为数据接收回调函数，传入**8.2.2.4.1 Graph::SetDataRecvFunctor**或**8.2.3.5 Engine::SetDataRecvFunctor**。同时，用户需要在子类中重载实现RecvData接口。该接口在data_recv_interface.h中定义。

函数格式

virtual HIAI_StatusT DataRecvInterface::RecvData(const std::shared_ptr<void>& message)

参数说明

参数	说明	取值范围
message	回调消息	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

8.2.5.8.2 Graph::SetPublicKeyForSignatureEncryption

该接口用于设置公钥，该公钥用于对Device侧Matrix要加载的*.so文件进行数字签名验证。需要在创建Graph之前调用该接口。该接口在graph.h中定义。

数字签名采用的算法为**SHA256withRSA**，本组件只提供数字签名的验证功能，加签的功能需要由用户自己基于该算法进行对需要的SO进行签名，签名文件在host的存放路径需要与SO在统一目录，并且签名文件的名称为SO名称+“.signature”，比如SO的名称为“libhosttodevice.so”，签名文件的名称应为“libhosttodevice.so.signature”。

注意

如果不调用Graph::SetPublicKeyForSignatureEncryption接口或者调用Graph::SetPublicKeyForSignatureEncryption接口的返回值不是HIAI_OK，则Matrix无法设置公钥，Device侧不会启用签名校验机制，因此无法通过数字签名识别出可能被篡改的*.so文件。

函数格式

static HIAI_StatusT Graph::SetPublicKeyForSignatureEncryption(const std::string& publicKey)

参数说明

参数	说明	取值范围
publicKey	数字签名验证需要的公钥	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok

```
std::string publicKey =
"-----BEGIN RSA PUBLIC KEY-----\n"
```

```
"MIIBCAKCAQEAt+kdmu8CdViw2xHlh/JWXOl/0AitDgYGd+9RadULGZmj0tt/UQLv\n"
"EYPZaXC8E0a9e97kqfg/ZHinu04XG5RhXv2J0kwkhuBAeCCllefvrQy5OBRqdLYq\n"
"EZe8wUO0adSMDsLbPL52b+NdO9/zX+MUzvsBzitWJbiH96s4xCiEERX87/uQfr6F\n"
"lHLtrNtooeF2VmxOm3n0yzh4kcSouLgb/O0+v0I+fnR+CTNG95IvzDNOYLWD6ZkL\n"
"c7JyIYFZA0CNx9SlPhqbrfjOV5XSG3g3CW0TopUDfHyhAgZt5vACMpeDDx+89tg2\n"
"RfT4M9DH/qKzkLlOURvsMShRMD6/PwzsPwIBAw==\n"
"-----END RSA PUBLIC KEY-----";
auto ret = Graph::SetPublicKeyForSignatureEncryption(publicKey);
```

8.2.6 Python 接口

如下接口不建议用户使用，若用户需要使用，请联系华为技术支持获取接口详细使用方法。

表 8-2 Basic Process Connection APIs

分类	API
Class Graph	Graph.__init__
	Graph.__del__
	Graph.get_id
	Graph.set_device_id
	Graph.get_graph
	Graph.destroy
	Graph.destroy_graph
	Graph.as_default
	Graph.create_graph
Class Engine	Engine.__init__
	Engine.update_config
	Engine.as_default
Class EngineConfig	EngineConfig.__init__
	EngineConfig.engine_id
	EngineConfig.engine_name
	EngineConfig.so_names
	EngineConfig.side
	EngineConfig.thread_num
	EngineConfig.thread_priority
	EngineConfig.queue_size
	EngineConfig.ai_config
	EngineConfig.internel_so_names

分类	API
	EngineConfig.wait_inputdata_max_time
	EngineConfig.hold_model_file_flag
Class GraphConfig	GraphConfig.__init__
	GraphConfig.graph_id
	GraphConfig.priority
	GraphConfig.device_id
Function APIs	get_default_graph
	get_default_engine
	crop
	resize
	inference

表 8-3 Basic Data Types

分类	API
Class NNTensor	NNTensor. __init__
	NNTensor.height
	NNTensor.width
	NNTensor.channel
	NNTensor.batch_size
Class NNTensorList	NNTensorList.__init__
	NNTensorList.__getitem__
	NNTensorList.get_tensor_num
	NNTensorList.next
	NNTensorList.__iter__

表 8-4 Data Pre-processing

分类	API
Class Engine	Engine.crop
	Engine.resize

版权所有 © 华为技术有限公司

分类	API
	Engine.padding
Class CropConfig	CropConfig.__init__
Class ResizeConfig	ResizeConfig.__init__

表 8-5 Offline Model Inference

分类	API
Class Engine	Engine.inference
Class AIModelDescription	AIModelDescription.__init__
	AIModelDescription.name
	AIModelDescription.model_type
	AIModelDescription.version
	AIModelDescription.size
	AIModelDescription.path
Class AIConfig	AIConfig.__init__
Class AIConfigItem	AIConfigItem.__init__

表 8-6 Data TX/RX

分类	API
Class Graph	Graph.proc

8.2.7 附录

8.2.7.1 后续版本会删除的接口

后续版本会删除的接口，用户无需调用，调用后无法保证后续兼容性。

8.2.7.1.1 低功耗接口（C++语言）

?.1. PowerState::SubscribePowerStateEvent

用户注册低功耗回调函数。该接口在power_state.h中定义。

函数格式

HIAI_StatusT SubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

参数说明

参数	说明	取值范围
notifyCallback	用户希望注册的回调函数。	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_POWER_STATE_CALLBACK_ERROR	App callback function is nullptr

?.2. PowerState::UnsubscribePowerStateEvent

用户注销先前注册的低功耗回调函数。该接口在power_state.h中定义。

函数格式

HIAI_StatusT UnsubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

参数说明

参数	说明	取值范围
notifyCallback	用户希望注销的回调函数	-

返回值

返回的部分错误码请参见“错误码示例”中的“错误码”列。

错误码示例

序号	错误码	错误码描述
1	HIAI_OK	running ok
2	HIAI_POWER_STATE_CALLBACK_ERROR	App callback function is nullptr

8.2.7.1.2 数据类型

?.1. 枚举：PowerState::DEVICE_POWER_STATE

```
enum DEVICE_POWER_STATE {
    DEVICE_POWER_STATE_S0 = 0,       // 系统进入S0，设备可用，唤醒APP
    DEVICE_POWER_STATE_SUSPEND,      // 系统即将进入S3/S4，APP释放通道、内存等资源
    DEVICE_POWER_STATE_D0,           // NPU设备进入D0，设备可用，唤醒APP（matrix业务无需处理）
    DEVICE_POWER_STATE_D3,           // NPU设备即将进入D3，APP停止与device侧通信（matrix业务无需处理）
    DEVICE_POWER_STATE_ENABLE,       // NPU设备驱动安装/启用, NPU可用
    DEVICE_POWER_STATE_DISABLE,      // NPU设备驱动卸载/禁用, NPU不可用
    DEVICE_POWER_STATE_MAX           // 状态最大个数，并不代表实际状态
};
```

?.2. PowerState::PowerStateNotifyCallbackT

```
typedef HIAI_StatusT (*PowerStateNotifyCallbackT)(uint32_t, DEVICE_POWER_STATE);
```

8.2.7.2 Matrix 数据类型

8.2.7.2.1 Protobuffer 数据类型

关于Protobuffer的使用，请参见**https://developers.google.com/protocol-buffers/docs/reference/proto3-spec**。

Matrix用到的数据类型Protobuffer格式：

ai_types.proto

```
syntax = "proto3";IAITensor

package hiai;

// Tensor参数描述
message AITensorParaDescription
{
    string name = 1;   // 参数名称
    string type = 2;   // 参数类型
    string value = 3;  // 参数值
    string desc = 4;   // 参数描述
    repeated AITensorParaDescription sub_paras = 5; // 子参数列表
};

// Tensor描述
message AITensorDescription
{
    string name = 1;   // Tensor名称
    string type = 2;   // Tensor类型
    repeated string compatible_type = 3; // 指定可以兼容的所有父类类型
    repeated AITensorParaDescription paras = 4; // 参数列表
};
```

```
// Tensor描述列表
message AITensorDescriptionList
{
    repeated AITensorDescription tensor_descs = 1; // tensor列表
}

// 通用配置项
// 如果存在sub_items，则当前节点为父节点，value值无效
message AIConfigItem
{
    string name = 1;   // 配置项名称
    string value = 2;  // 配置项值，此处配置模型文件所在的路径时，包含文件名（名称中只允许有数字、字母、下划线和点号），可以将参数值配置为单个模型文件的路径（如：./test_data/model/resnet18.om）；也可以将模型文件打包成tar包后，将参数值配置为tar包所在的路径（如：./test_data/model/resnet18.tar）。若存在多个AIConfigItem，在配置tar包路径时，不允许在同一个目录下同时存在名称相同但格式不同的文件，例如./test_data/model/test、./test_data/model/test.tar。创建Graph接口包含Graph::CreateGraph或HIAI_CreateGraph。
    repeated AIConfigItem sub_items = 3;  // 配置子项
};

// nnnode/api运行时配置参数
message AIConfig
{
    repeated AIConfigItem items = 1; // 配置项列表
};

// 模型描述
message AIModelDescription
{
    string name = 1;  // 模型名称
    int32 type = 2;   // 模型类型，当前仅支持DAVINCI_OFFLINE_MODEL类型，值为0
    string version = 3;  // 模型版本
    int32 size = 4;  // 模型大小
    string path = 5;  // 模型路径
    string key = 8;  // 模型秘钥
    bytes data = 100;  // 模型数据
    repeated AITensorDescription inputs = 6;  // 输入Tensor描述
    repeated AITensorDescription outputs = 7; // 输出Tensor描述
};

// NNNode描述
message AINNNodeDescription
{
    string name = 1;  // NNNode名称
    string desc = 2;  // NNNode描述
    bool isPreAllocateOutputMem = 3;  // 是否预分配输出内存
    AIConfig config = 4; // 配置参数
    repeated AIModelDescription model_list = 5; // NNNode需要的模型列表
    repeated AITensorDescription inputs = 6;   // 输入Tensor描述
    repeated AITensorDescription outputs = 7;  // 输出Tensor描述
    bool need_verify = 8; // 串联时是否需要校验Tensor匹配
    repeated string ignored_check_aitensor = 9; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor列表
};

// NNNode描述列表
message AINNNodeDescriptionList
{
    repeated AINNNodeDescription nnnode_descs = 1; // NNNode列表
}

// API描述
message AIAPIDescription
{
    string name = 1;  // API名称
    string desc = 2;  // API描述
    bool isPreAllocateOutputMem = 3;  // 是否预分配输出内存
    AIConfig config = 4; // 配置参数
    repeated AITensorDescription inputs = 5;   // 输入Tensor描述
```

```
    repeated AITensorDescription outputs = 6;  // 输出Tensor描述
    bool need_verify = 7; // 串联时是否需要校验Tensor匹配
    repeated string ignored_check_aitensor = 8; // 指定在串联时不与Inputs的Tensor做匹配校验的Tensor列表
};

// API描述列表
message AIAPIDescriptionList
{
    repeated AIAPIDescription api_descs = 1; // API列表
}

// AI Operation 描述
message AIOPDescription
{
    enum OP_Type
    {
        AI_API = 0;
        AI_NNNODE = 1;
    }

    OP_Type type = 1;
    AINNNodeDescription nnnode_desc = 2;
    AIAPIDescription api_desc = 3;
};

// AI Operation 描述列表
message AIOPDescriptionList
{
    repeated AIOPDescription op_descs = 1; //  AI Operation列表
}

// IDE传给Matrix的接口

message NodeDesc
{
   string name=1; // IAINNNode或ALG_API 名字
   AIConfig config=2;  //IAINNNode或ALG_API 需要的初始化参数
   repeated  AIModelDescription mode_desc=3; //IAINNNode 需要的初始化参数
}

message EngineDesc
{
    enum RunSide
    {
        DEVICE=0;
        HOST=1;
    }
    enum EngineType
    {
        NORMAL=0;
        SOURCE=1;
        DEST=2;
    }
    uint32 id =1;   //Engine ID (节点)
    EngineType type=2; //
    string name =3; //Engine 节点名字
    repeated string so_name=4; //需要拷贝的所有动态库so文件名列表，会先按照用户配置的顺序先进行加载，
加载失败再按照其他的加载顺序进行尝试加载
    RunSide side=5; //部署在host侧还是Device侧
    int32 priority=6; //节点优先级
    uint32 instance_cnt=7; //实例个数（相当于线程个数）
    repeated uint32 next_node_id=8; //下一个节点列表
    bool user_input_cb=9;   //IDE 可以忽略
    bool user_output_cb=10; //IDE 可以忽略
    repeated NodeDesc oper=11; //Matrix Node 列表
}

message GraphInitDesc
{
```

```
    int32 priority=1; //Graph 整个进程的优先级
    //Runside side = 2; //当前假定Graph 配置在host侧，不可配置
}

message GeneralFileBuffer
{
    bytes raw_data = 1;
    string file_name = 2;
}
```

graph_config.proto

```
syntax = "proto3";
import "ai_types.proto";
package hiai;

message DumpDef
{
    bytes model_name = 1;      //模型名称
    bytes is_dump_all = 2;    //是否打开Dump开关，用于Dump模型中每层算子的信息
    repeated bytes layer = 3;    //需要Dump模型中哪些层的算子信息
}
message OAMConfigDef
{
    repeated DumpDef dump_list = 1;
    bytes dump_path = 2;    //dump文件存放的路径
}
message EngineConfig
{
    enum RunSide
    {
        DEVICE=0;   //运行在DEVICE侧
        HOST=1;      //运行在host侧
    }

    uint32 id =1;   //Engine ID (节点)
    string engine_name =2; //Engine 节点名字
    repeated string so_name=3; //需要拷贝的所有动态库so文件名列表，会先按照用户配置的顺序先进行加载，加载失败再按照其他的加载顺序进行尝试加载
    RunSide side=4; //部署在host侧还是Device侧
    uint32 thread_num = 5;//线程数量。多路解码时，则该参数值推荐设置为1，如果thread_num值大于1，则线程之间的解码无法保证顺序。
    uint32 thread_priority = 6;//线程优先级
    uint32 queue_size = 7;//队列大小
    AIConfig ai_config = 8; //Aiconfig配置文件
    repeated AIModelDescription ai_model = 9; //AIModelDescription
    repeated string internal_so_name=10; //不需要拷贝的所有动态库so文件名列表
    uint32 wait_inputdata_max_time = 12; //当前已经收到数据后等待下一个数据的最大超时时间，单位为毫秒
    uint32 holdModelFileFlag = 13; //是否保留本engine的模型文件，0不保留（默认为0），非0保留
    OAMConfigDef oam_config = 14; // OAM Config配置
    bool is_repeat_timeout_flag = 15; //engine未收到数据是否重复做超时处理，0不重复，1重复。例如多路数据分析时，数据量较大的情况下，建议将该参数值设置为1，做重复超时处理。与wait_inputdata_max_time参数配合使用，如果is_repeat_timeout_flag设置为0，则通过wait_inputdata_max_time参数设置单次超时时长；如果is_repeat_timeout_flag设置为1，则通过wait_inputdata_max_time参数设置周期超时时长。
}
message ConnectConfig
{
    uint32 src_engine_id=1;  //发送端的EngineID
    uint32 src_port_id = 2;  //发送端的PortID
    uint32 target_graph_id=3; //接收端的GraphID
    uint32 target_engine_id=4; //接收端的PortID
    uint32 target_port_id=5;  //接收端的PortID
    bool receive_memory_without_dvpp=6; //参数默认值为0，表示Deivce上运行的目标Engine的接收内存需要满足DVPP的内存要求；若将参数值配置为1，表示Device上运行的目标Engine的接收内存无限制。参数的生效场景及使用场景请参见“关于receive_memory_without_dvpp参数的说明”。
}
message GraphConfig
{
    uint32 graph_id = 1;  //GraphID
    int32 priority = 2;  //优先级
```

```
    string device_id = 3;//设备id配置，例如"0"
    repeated EngineConfig engines = 3; //所有的engine。多路解码时推荐配置多个Engine，一个Engine对应一个线程，如果一个Engine对应多个线程，解码时无法保证顺序。
    repeated ConnectConfig connects = 4;  //连接方式
}
message GraphConfigList
{
    repeated GraphConfig graphs = 1;  //单个Graph的配置参数
}

message ProfileConfig
{
    string matrix_profiling = 1;  //Matrix的性能统计开关，“on”为开
    string ome_profiling = 2;  //OME的性能统计开关，“on”为开
    string cce_profiling =3;  //CCE的性能统计开关，“on”为开
    string runtime_profiling = 4;  //RUNTIME的性能统计开关，“on”为开
    string PROFILER_TARGET = 5;  //透传Profiling的参数
    string PROFILER_JOBCTX = 6;
    string src_path = 7;
    string dest_path = 8;
    string runtime_config = 9;
    string RTS_PATH = 10;
    string profiler_jobctx_path = 11;
    string profiler_target_path = 12;
}

message GraphUpdateConfig
{
    GraphConfig updataGraphConfig = 1;
    repeated uint32 del_engine_ids = 2;
    repeated ConnectConfig del_connects = 3;
}
```

说明

关于receive_memory_without_dvpp参数的说明：

- 生效场景：使用**HIAI_DMalloc**接口申请内存，同时使用**HIAI_REGISTER_SERIALIZE_FUNC**宏对自定义数据类型进行序列化/反序列化时，配置文件中的该参数才生效。
- 使用场景：

  当前，在DVPP中：

  - 对VPC、JPEGE、JPEGD、PNGD功能的输入内存有限制，保持receive_memory_without_dvpp参数的默认值（默认值为0），不用配置，内存满足DVPP的要求；
  - 对VDEC、VENC功能的输入内存无限制，可将receive_memory_without_dvpp参数配置为1，使用普通内存，提高系统普通内存的使用率，减少“满足DVPP要求的内存”的占用；也可以不配置receive_memory_without_dvpp参数（默认值为0）。
- 配置方法有两种：
  - 在Graph配置文件中，在connects属性下，将所有在Device上运行的目标Engine的receive_memory_without_dvpp参数配置为1。
  - 在Graph配置文件中，将所有connects属性下的receive_memory_without_dvpp参数都配置为1，因为在Host上运行的目标Engine即使配置了该参数也无影响。

8.2.7.2.2 Matrix 自定义数据类型

- HIAI_StatusT：用于定义返回码的数据类型为uint32_t。
- EnginePortID：Matrix中用于定义端口的结构体。

```
struct EnginePortID {
    uint32_t graph_id;
    uint32_t engine_id;
    uint32_t port_id;
};
```

8.2.7.3 Matrix 已经注册的数据结构

Matrix中传输的数据结构需要先进行注册，以下数据结构Matrix已经进行了注册，可直接使用。

```
namespace hiai {
    // message name  <<<<<<==================>>>>>>> message type
    // name:"BatchInfo"                        type:BatchInfo
    // name:"FrameInfo"                        type:FrameInfo
    // name:"IMAGEFORMAT"                      type:IMAGEFORMAT
    // name:"Angle"                            type:Angle
    // name:"Point2D"                          type:Point2D
    // name:"Point3D"                          type:Point3D
    // name:"ROICube"                          type:ROICube
    // name:"RLECode"                          type:RLECode
    // name:"IDTreeNode"                       type:IDTreeNode
    // name:"IDTreePara"                       type:IDTreePara
    // name:"BatchIDTreePara"                  type:BatchIDTreePara
    // name:"RetrievalResult"                  type:RetrievalResult
    // name:"RetrievalResultTopN"              type:RetrievalResultTopN
    // name:"RetrievalResultPara"              type:RetrievalResultPara
    // name:"RawDataBuffer"                    type:RawDataBuffer
    // name:"BatchRawDataBuffer"               type:BatchRawDataBuffer
    // name:"string"                           type:string


    // name:"vector_uint8_t"                   type:vector<uint8_t>
    // name:"vector_float"                     type:vector<float>


    // name:"ImageData_uint8_t"                type:ImageData<uint8_t>
    // name:"ImageData_float"                  type:ImageData<float>


    // name:"ImagePara_uint8_t"                type:ImagePara<uint8_t>
    // name:"ImagePara_float"                  type:ImagePara<float>


    // name:"BatchImagePara_uint8_t"           type:BatchImagePara<uint8_t>
    // name:"BatchImagePara_float"             type:BatchImagePara<float>


    // name:"Line_uint8_t"                     type:Line<uint8_t>
    // name:"Line_float"                       type:Line<float>


    // name:"Rectangle_uint8_t"                type:Rectangle<uint8_t>
    // name:"Rectangle_float"                  type:Rectangle<float>


    // name:"Polygon_uint8_t"                  type:Polygon<uint8_t>
    // name:"Polygon_float"                    type:Polygon<float>


    // name:"MaskMatrix_uint8_t"               type:MaskMatrix<uint8_t>
    // name:"MaskMatrix_float"                 type:MaskMatrix<float>


    // name:"ObjectLocation_uint8_t_uint8_t"   type:ObjectLocation<uint8_t, uint8_t>
    // name:"ObjectLocation_float_float"       type:ObjectLocation<float, uint8_t>


    // name:"DetectedObjectPara_uint8_t_uint8_t"   type:DetectedObjectPara<uint8_t, uint8_t>
    // name:"DetectedObjectPara_float_float"       type:DetectedObjectPara<float, float>


    // name:"BatchDetectedObjectPara_uint8_t_uint8_t"
type:BatchDetectedObjectPara<uint8_t,uint8_t>
    // name:"BatchDetectedObjectPara_float_float"          type:BatchDetectedObjectPara<float,float>
```

```
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_int32_t"
type:BatchDetectedObjectPara<Rectangle<Point2D>, int32_t>
    // name:"BatchDetectedObjectPara_Rectangle_Point2D_float"
type:BatchDetectedObjectPara<Rectangle<Point2D>, float>


    // name:"RegionImage_uint8_t"                type:RegionImage<uint8_t>
    // name:"RegionImage_float"                  type:RegionImage<float>


    // name:"RegionImagePara_uint8_t"            type:RegionImagePara<uint8_t>
    // name:"RegionImagePara_float"              type:RegionImagePara<float>


    // name:"BatchRegionImagePara_uint8_t"       type:BatchRegionImagePara<uint8_t>
    // name:"BatchRegionImagePara_float"         type:BatchRegionImagePara<float>


    // name:"Attribute_uint8_t"                  type:Attribute<uint8_t>
    // name:"Attribute_float"                    type:Attribute<float>


    // name:"AttributeTopN_uint8_t"              type:AttributeTopN<uint8_t>
    // name:"AttributeTopN_float"                type:AttributeTopN<float>


    // name:"AttributeVec_uint8_t"               type:AttributeVec<uint8_t>
    // name:"AttributeVec_float"                 type:AttributeVec<float>


    // name:"AttributeResultPara_uint8_t"        type: AttributeVec<uint8_t>
    // name:"AttributeResultPara_float"          type:AttributeVec<float>


    // name:"BatchAttributeResultPara_uint8_t"   type:AttributeVec<uint8_t>
    // name:"BatchAttributeResultPara_float"     type:AttributeVec<float>


    // name:"Feature_uint8_t"                    type:AttributeVec<uint8_t>
    // name:"Feature_float"                      type:AttributeVec<float>


    // name:"FeatureList_uint8_t"                type:FeatureList<uint8_t>
    // name:"FeatureList_float"                  type:FeatureList<float>


    // name:"FeatureVec_uint8_t"                 type:FeatureVec<uint8_t>
    // name:"FeatureVec_float"                   type:FeatureVec<float>


    // name:"FeatureResultPara_uint8_t"          type:FeatureResultPara<uint8_t>
    // name:"FeatureResultPara_float"            type:FeatureResultPara<float>


    // name:"BatchFeatureResultPara_uint8_t"     type:BatchFeatureResultPara<uint8_t>
    // name:"BatchFeatureResultPara_float"       type:BatchFeatureResultPara<float>


    // name:"FeatureRecord_uint8_t"              type:FeatureRecord<uint8_t>
    // name:"FeatureRecord_float"                type:FeatureRecord<float>


    // name:"RetrievalSet_uint8_t_uint8_t"       type:RetrievalSet<uint8_t, uint8_t>
    // name:"RetrievalSet_float_float"           type:RetrievalSet<float, float>


    // name:"EvaluationResult_uint8_t"           type:EvaluationResult<uint8_t>
    // name:"EvaluationResult_float"             type:EvaluationResult<float>
```

```
// name:"EvaluationResultVec_uint8_t"          type:EvaluationResultVec<uint8_t>
// name:"EvaluationResultVec_float"            type:EvaluationResultVec<float>


// name:"EvaluationResultPara_uint8_t"         type:EvaluationResultPara<uint8_t>
// name:"EvaluationResultPara_float"           type:EvaluationResultPara<float>


// name:"BatchEvaluationResultPara_uint8_t"    type:BatchEvaluationResultPara<uint8_t>
// name:"BatchEvaluationResultPara_float"      type:BatchEvaluationResultPara<float>


// name:"Classification_uint8_t"               type:Classification<uint8_t>
// name:"Classification_float"                 type:Classification<float>


// name:"ClassificationTopN_uint8_t"           type:ClassificationTopN<uint8_t>
// name:"ClassificationTopN_float"             type:ClassificationTopN<float>


// name:"ClassificationVec_uint8_t"            type:ClassificationVec<uint8_t>
// name:"ClassificationVec_float"              type:ClassificationVec<float>


// name:"ClassificationResultPara_uint8_t"     type:ClassificationResultPara<uint8_t>
// name:"ClassificationResultPara_float"       type:ClassificationResultPara<float>


// name:"BatchClassificationResultPara_uint8_t" type:BatchClassificationResultPara<uint8_t>
// name:"BatchClassificationResultPara_float"   type:BatchClassificationResultPara<float>


/////////////////////////////////////////////////////
///////////////     batch信息     ////////////////////
/////////////////////////////////////////////////////
struct BatchInfo {
    bool is_first = false;              // 是否为第一个batch
    bool is_last = false;               // 是否为最后一个batch
    uint32_t batch_size = 0;            // 当前batch的实际大小
    uint32_t max_batch_size = 0;        // batch预设大小（最大容量）
    uint32_t batch_ID = 0;              // 当前batch ID号
    uint32_t channel_ID = 0;            // 处理当前batch的通道ID号
    uint32_t processor_stream_ID = 0;   // 处理器计算流ID号


    std::vector<uint32_t> frame_ID;     // batch中图像帧ID号
    std::vector<uint32_t> source_ID;    // batch中图像源ID号


    std::vector<uint64_t> timestamp;    // batch中图像的时间戳
};


template<class Archive>
void serialize(Archive& ar, BatchInfo& data) {
    ar(data.is_first, data.is_last, data.batch_size,
        data.max_batch_size, data.batch_ID, data.channel_ID,
        data.processor_stream_ID, data.frame_ID, data.source_ID,
        data.timestamp);
}


/////////////////////////////////////////////////////
///////////////     frame信息     ////////////////////
/////////////////////////////////////////////////////
struct FrameInfo {
    bool is_first = false;              // 是否为第一个frame
    bool is_last = false;               // 是否为最后一个frame
```

```
    uint32_t channel_ID = 0;              // 处理当前frame的通道ID号
    uint32_t processor_stream_ID = 0;     // 处理器计算流ID号
    uint32_t frame_ID = 0;                // 图像帧ID号
    uint32_t source_ID = 0;               // 图像源ID号
    uint64_t timestamp = 0;               // 图像的时间戳
};


template<class Archive>
void serialize(Archive& ar, FrameInfo& data) {
    ar(data.is_first, data.is_last, data.channel_ID,
        data.processor_stream_ID, data.frame_ID, data.source_ID,
        data.timestamp);
}


//////////////////////////////////////////////////////
//////////////////  1. image tensor ////////////////////
//////////////////////////////////////////////////////


// 图像格式
enum IMAGEFORMAT {
    RGB565,          //  Red 15:11, Green 10:5, Blue 4:0
    BGR565,          //  Blue 15:11, Green 10:5, Red 4:0
    RGB888,          //  Red 24:16, Green 15:8, Blue 7:0
    BGR888,          //  Blue 24:16, Green 15:8, Red 7:0
    BGRA8888,        //  Blue 31:24, Green 23:16, Red 15:8, Alpha 7:0
    ARGB8888,        //  Alpha 31:24, Red 23:16, Green 15:8, Blue 7:0
    RGBX8888,
    XRGB8888,
    YUV420Planar,    //  I420
    YVU420Planar,    //  YV12
    YUV420SP,        //  NV12
    YVU420SP,        //  NV21
    YUV420Packed,    //  YUV420 Interleaved
    YVU420Packed,    //  YVU420 Interleaved
    YUV422Planar,    //  Three arrays Y,U,V.
    YVU422Planar,
    YUYVPacked,      //  422 packed per payload in planar slices
    YVYUPacked,      //  422 packed
    UYVYPacket,      //  422 packed
    VYUYPacket,      //  422 packed
    YUV422SP,        //  422 packed
    YVU422SP,
    YUV444Interleaved,  //  Each pixel contains equal parts YUV
    Y8,
    Y16,
    RAW
};


enum OBJECTTYPE {
    OT_VEHICLE,
    OT_HUMAN,
    OT_NON_MOTOR,
    OT_FACE,
    OT_FACE_BODY,
    OT_PLATE
};


template<class T>   // T: uint8_t  float
struct ImageData {
    IMAGEFORMAT format;   // 图像格式


    uint32_t width  = 0;       // 图像宽
    uint32_t height = 0;       // 图像高
```

```
    uint32_t channel = 0;     // 图像通道数
    uint32_t depth = 0;       // 位深
    uint32_t height_step = 0;  // 对齐高度
    uint32_t width_step = 0;  // 对齐宽度
    uint32_t size = 0;        // 数据大小（Byte）
    std::shared_ptr<T> data;   // 数据指针
};


template<class Archive, class T>
void serialize(Archive& ar, ImageData<T>& data) {
    ar(data.format, data.width, data.height, data.channel,
        data.depth, data.height_step, data.width_step, data.size);
    if (data.size > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.size]);
    }
    ar(cereal::binary_data(data.data.get(), data.size * sizeof(T)));
}


// 参数
template<class T>
struct ImagePara {
    FrameInfo f_info;          // frame信息
    ImageData<T> img;          // 图像
};


template<class Archive, class T>
void serialize(Archive& ar, ImagePara<T>& data) {
    ar(data.f_info, data.img);
}


// 参数
template<class T>
struct BatchImagePara {
    BatchInfo b_info;                  // batch信息
    std::vector<ImageData<T> > v_img;   // batch中的图像
};


template<class Archive, class T>
void serialize(Archive& ar, BatchImagePara<T>& data) {
    ar(data.b_info, data.v_img);
}




// 角度
typedef struct {
    float x;
    float y;
    float z;
}Angle;


template<class Archive>
void serialize(Archive& ar, Angle& data) {
    ar(data.x, data.y, data.z);
}


// 2D点
struct Point2D {
    int32_t x;
    int32_t y;
};
```

```
template<class Archive>
void serialize(Archive& ar, Point2D& data) {
    ar(data.x, data.y);
}


//////////////////////////////////////////////////////
///////////////     ROI信息     ////////////////////
//////////////////////////////////////////////////////


struct RoiPolygon {
    uint32_t          uiPtNum;       //多边形顶点数
    std::vector<Point2D> astPts;    //多个检索结果（TopN）
};


template<class Archive>
void serialize(Archive &ar, RoiPolygon &data) {
    ar(data.uiPtNum, data.astPts);
}


struct ArgsRoiPolygon {
    RoiPolygon stRoiPolygon; // 配置项Roi
    uint32_t iMinFace; // 最小人脸26*26
    uint32_t iMaxFace; // 最大人脸300*300
    uint32_t imgWidth;  // 视频帧图像宽
    uint32_t imgHeight; // 视频帧图像高
};


template<class Archive>
void serialize(Archive &ar, ArgsRoiPolygon &data) {
    ar(data.stRoiPolygon, data.iMinFace, data.iMaxFace, data.imgWidth, data.imgHeight, data.iMinFace);
}
// 3D点
struct Point3D {
    int32_t x;
    int32_t y;
    int32_t z;
};


template<class Archive>
void serialize(Archive& ar, Point3D& data) {
    ar(data.x, data.y, data.z);
}


// 2D或3D空间上的线
template<class T>   // T: Point2D  Point3D
struct Line {
    T start;
    T end;
};


template<class Archive, class T>
void serialize(Archive& ar, Line<T>& data) {
    ar(data.start, data.end);
}


// 2D或3D空间上的矩形平面
template<class T>  // T: Point2D  Point3D
struct Rectangle {
```

```
    T anchor_lt;
    T anchor_rb;
};


template<class Archive, class T>
void serialize(Archive& ar, Rectangle<T>& data) {
    ar(data.anchor_lt, data.anchor_rb);
}


// 2D或3D空间上的多边形平面
template<class T>  // T: Point2D  Point3D
struct Polygon {
    std::vector<T> anchors;
};


template<class Archive, class T>
void serialize(Archive& ar, Polygon<T>& data) {
    ar(data.anchors);
}


// 立方体
struct ROICube {
    Point3D anchor;
    uint32_t length;
    uint32_t width;
    uint32_t height;
};


template<class Archive>
void serialize(Archive& ar, ROICube& data) {
    ar(data.anchor, data.length, data.width, data.height);
}


// 掩膜矩阵
template<class T>    // T: int32_t float etc.
struct MaskMatrix {
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<T> data;
};


template<class Archive, class T>
void serialize(Archive& ar, MaskMatrix<T>& data) {
    ar(data.cols, data.rows);
    if (data.cols*data.rows > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.cols*data.rows]);
    }
    ar(cereal::binary_data(data.data.get(),
        data.cols * data.rows * sizeof(T)));
}


// RLE编码
struct RLECode {
    uint32_t len;
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<uint32_t> data;
};


template<class Archive>
```

```
void serialize(Archive& ar, RLECode& data) {
    ar(data.len, data.cols, data.rows);
    if (data.len > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint32_t[data.len]);
    }
    ar(cereal::binary_data(data.data.get(),
        data.len * sizeof(uint32_t)));
}


template<class T1, class T2>    // T1: Point2D Rectangle RLECode etc.
struct ObjectLocation {          // T2: int8_t float etc.
    std::vector<uint32_t> v_obj_id;
    std::vector<T1> range;           // 目标范围描述
    std::vector<Angle> angle;        // 角度信息
    std::vector<T2> confidence;      // 置信度
    std::vector<uint32_t> label;     // 目标类型标签
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, ObjectLocation<T1, T2>& data) {
    ar(data.v_obj_id, data.range, data.angle,
       data.confidence, data.label);
}


// 参数
template<class T1, class T2>
struct DetectedObjectPara {
    FrameInfo f_info;                // frame信息
    ObjectLocation<T1, T2> location;     // 多个目标位置
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, DetectedObjectPara<T1, T2>& data) {
    ar(data.f_info, data.location);
}


// 参数
template<class T1, class T2>
struct BatchDetectedObjectPara {
    //  batch信息
    BatchInfo b_info;
    //  对应每帧图像的多个目标位置
    std::vector<ObjectLocation<T1, T2> > v_location;
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, BatchDetectedObjectPara<T1, T2>& data) {
    ar(data.b_info, data.v_location);
}






//////////////////////////////////////////////////////////
/////////////// 3. region image tensor ////////////////
//////////////////////////////////////////////////////////


template<class T>
struct RegionImage {
    std::vector<ImageData<T> > region;
```

```
    std::vector<uint32_t> v_obj_id;
};


template<class Archive, class T>
void serialize(Archive& ar, RegionImage<T>& data) {
    ar(data.v_obj_id, data.region);
}


// 参数
template<class T>
struct RegionImagePara {
    FrameInfo f_info;             // frame信息
    RegionImage<T> region;        // 每帧对应的小图
};


template<class Archive, class T>
void serialize(Archive& ar, RegionImagePara<T>& data) {
    ar(data.f_info, data.region);
}


// 参数
template<class T>
struct BatchRegionImagePara {
    // batch信息
    BatchInfo b_info;
    // 每帧图像若干region image，用一个指向Image数组的指针来引用
    std::vector<RegionImage<T>> v_region;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchRegionImagePara<T>& data) {
    ar(data.b_info, data.v_region);
}




// 递归ID树节点
struct IDTreeNode {
    std::vector<uint32_t>  nodeID;     // 节点上的ID值
    std::vector<bool> is_leaf;         // 是否为叶子节点
    std::vector<IDTreeNode> node;     // 子节点结构指针
};


template <class Archive>
void serialize(Archive& ar, IDTreeNode& data) {
    ar(data.nodeID, data.is_leaf, data.node);
}


// 参数
struct IDTreePara {
    FrameInfo f_info;    // frame信息
    IDTreeNode tree;     // 图像中的ID Tree
};


template <class Archive>
void serialize(Archive& ar, IDTreePara& data) {
    ar(data.f_info, data.tree);
}
```

```
// 参数
struct BatchIDTreePara {
    BatchInfo b_info;                    // batch信息
    std::vector<IDTreeNode> v_tree;     // 每帧图像中的ID Tree
};


template <class Archive>
void serialize(Archive& ar, BatchIDTreePara& data) {
    ar(data.b_info, data.v_tree);
}




template<class T>
struct Attribute {
    std::string attr_value;
    T score;
};


template <class Archive, class T>
void serialize(Archive& ar, Attribute<T>& data) {
    ar(data.attr_value, data.score);
}


template<class T>
struct AttributeTopN {
    uint32_t obj_ID;
    std::map<std::string, std::vector<Attribute<T>> > attr_map;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeTopN<T>& data) {
    ar(data.obj_ID, data.attr_map);
}


template<class T>
struct AttributeVec {
    std::vector<AttributeTopN<T>> obj_attr;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeVec<T>& data) {
    ar(data.obj_attr);
}


// ²ÎÊý
template<class T>
struct AttributeResultPara {
    FrameInfo f_info;
    AttributeVec<T> attr;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeResultPara<T>& data) {
    ar(data.f_info, data.attr);
}
```

```
// ²ÎÊý
template<class T>
struct BatchAttributeResultPara {
    BatchInfo b_info;
    std::vector<AttributeVec<T>> v_attr;
};


template <class Archive, class T>
void serialize(Archive& ar, BatchAttributeResultPara<T>& data) {
    ar(data.b_info, data.v_attr);
}


template<class T>
struct Feature {
    int32_t len;                // 特征向量长度
    std::shared_ptr<T> feature;      // 特征向量指针
};


template<class Archive, class T>
void serialize(Archive& ar, Feature<T>& data) {
    ar(data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}




template<class T>
struct FeatureList {
    uint32_t obj_ID;                // 目标ID
    std::vector<Feature<T>> feature_list;    // 目标的多个特征
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureList<T>& data) {
    ar(data.obj_ID, data.feature_list);
}


template<class T>
struct FeatureVec {
    std::vector<FeatureList<T>> obj_feature;   // 多个目标的属性
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureVec<T>& data) {
    ar(data.obj_feature);
}


// 参数
template<class T>
struct FeatureResultPara {
    FrameInfo f_info;         // frame信息
    FeatureVec<T> feature;     // 图像的多个目标的特征
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureResultPara<T>& data) {
    ar(data.f_info, data.feature);
```

```
}


// 参数
template<class T>
struct BatchFeatureResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的多个目标的特征
    std::vector<FeatureVec<T>> v_feature;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchFeatureResultPara<T>& data) {
    ar(data.b_info, data.v_feature);
}




template<class T>
struct FeatureRecord {
    uint32_t ID;            // 特征图ID
    uint32_t len;           // 特征向量长度
    std::shared_ptr<T> feature;  // 特征向量指针
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureRecord<T>& data) {
    ar(data.ID, data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}


template<class T1, class T2>
struct RetrievalSet {
    uint32_t TopN;       // TopN结果设置
    T1 threshold;        // 相似度阈值
    std::vector<FeatureRecord<T2>> record;    // 特征集合中所有的记录
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSet<T1, T2>& data) {
    ar(data.TopN, data.threshold, data.record);
}


// 参数
template<class T1, class T2>
struct RetrievalSetPara {
    RetrievalSet<T1, T2> set;   // 参数和特征集合
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSetPara<T1, T2>& data) {
    ar(data.set);
}


struct RetrievalResult {
    uint32_t ID;        // 特征图ID
    float similarity;   // 相似度
```

```
};


template<class Archive>
void serialize(Archive& ar, RetrievalResult& data) {
    ar(data.ID, data.similarity);
}


struct RetrievalResultTopN {
    std::vector<RetrievalResult> result_list;    // 多个检索结果（TopN）
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultTopN& data) {
    ar(data.result_list);
}


// 参数
struct RetrievalResultPara {
    FrameInfo f_info;               // frame信息
    RetrievalResultTopN result;     // 检索结果
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultPara& data) {
    ar(data.f_info, data.result);
}


// 参数
struct BatchRetrievalResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的检索结果
    std::vector<RetrievalResultTopN> v_result;
};


template<class Archive>
void serialize(Archive& ar, BatchRetrievalResultPara& data) {
    ar(data.b_info, data.v_result);
}


template<class T>
struct EvaluationResult {
    uint32_t obj_ID;            // 目标ID
    std::string description;    // 描述
    T score;                    // 评价分数
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResult<T>& data) {
    ar(data.obj_ID, data.description, data.score);
}


template<class T>
struct EvaluationResultVec {
    std::vector<EvaluationResult<T>> result;    // 多个目标的评价结果
};


template <class Archive, class T>
```

```
void serialize(Archive& ar, EvaluationResultVec<T>& data) {
    ar(data.result);
}


// 参数
template<class T>
struct EvaluationResultPara {
    FrameInfo f_info;                // frame信息
    EvaluationResultVec<T> result;    // 评价结果
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResultPara<T>& data) {
    ar(data.f_info, data.result);
}


// 参数
template<class T>
struct BatchEvaluationResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的评价结果
    std::vector<EvaluationResultVec<T>> v_result;
};


template <class Archive, class T>
void serialize(Archive& ar, BatchEvaluationResultPara<T>& data) {
    ar(data.b_info, data.v_result);
}




template<class T>
struct Classification {
    std::string class_value;    // 类别取值
    T score;                    // 置信度
};


template<class Archive, class T>
void serialize(Archive& ar, Classification<T>& data) {
    ar(data.class_value, data.score);
}


template<class T>
struct ClassificationTopN {
    uint32_t obj_ID;                            // 目标ID
    std::vector<Classification<T>> class_result;   // 分类结果（TopN）
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationTopN<T>& data) {
    ar(data.obj_ID, data.class_result);
}


template<class T>
struct ClassificationVec {
    std::vector<ClassificationTopN<T>> class_list;   // 多个目标的分类
};
```

```
template<class Archive, class T>
void serialize(Archive& ar, ClassificationVec<T>& data) {
    ar(data.class_list);
}


// 参数
template<class T>
struct ClassificationResultPara {
    FrameInfo f_info;                       // frame信息
    ClassificationVec<T> classification;       // 多个目标的分类
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationResultPara<T>& data) {
    ar(data.f_info, data.classification);
}


// 参数
template<class T>
struct BatchClassificationResultPara {
    // batch信息
    BatchInfo b_info;
    // 对应每帧图像的多个目标的分类
    std::vector<ClassificationVec<T>> v_class;
};




template<class Archive, class T>
void serialize(Archive& ar, BatchClassificationResultPara<T>& data) {
    ar(data.b_info, data.v_class);
}


struct RawDataBuffer {
    uint32_t len_of_byte;  // size length
    std::shared_ptr<uint8_t> data;  // 一块buffer，可转为用户自定义类型
};


template<class Archive>
void serialize(Archive& ar, RawDataBuffer& data) {
    ar(data.len_of_byte);
    if (data.len_of_byte > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint8_t[data.len_of_byte]);
    }
    ar(cereal::binary_data(data.data.get(), data.len_of_byte *
        sizeof(uint8_t)));
}


// common raw databuffer struct
struct BatchRawDataBuffer {
    BatchInfo b_info;  // batch info
    std::vector<RawDataBuffer> v_info;
};


template<class Archive>
void serialize(Archive& ar, BatchRawDataBuffer& data) {
    ar(data.b_info, data.v_info);
}
```

```
    template<typename T> const char* TypeName(void);


#define REGISTER_TYPE_DEFINITION(type) \
    template<> inline const char* TypeName<type>(void) { return #type; }


    REGISTER_TYPE_DEFINITION(int8_t);
    REGISTER_TYPE_DEFINITION(uint8_t);
    REGISTER_TYPE_DEFINITION(int16_t);
    REGISTER_TYPE_DEFINITION(uint16_t);
    REGISTER_TYPE_DEFINITION(int32_t);
    REGISTER_TYPE_DEFINITION(uint32_t);
    REGISTER_TYPE_DEFINITION(int64_t);
    REGISTER_TYPE_DEFINITION(uint64_t);
    REGISTER_TYPE_DEFINITION(float);
    REGISTER_TYPE_DEFINITION(double);
    REGISTER_TYPE_DEFINITION(Point2D);
    REGISTER_TYPE_DEFINITION(Point3D);
    REGISTER_TYPE_DEFINITION(Line<Point2D>);
    REGISTER_TYPE_DEFINITION(Line<Point3D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point2D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point3D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point2D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point3D>);
    REGISTER_TYPE_DEFINITION(ROICube);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int8_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int32_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<float>);
    REGISTER_TYPE_DEFINITION(RLECode);
    REGISTER_TYPE_DEFINITION(RoiPolygon);
    REGISTER_TYPE_DEFINITION(ArgsRoiPolygon);
}
```

8.2.7.4 示例

8.2.7.4.1 编排配置示例

Graph创建配置文件（graph.prototxt）为proto格式，其示例如下（该示例同时创建两个Graph，每个Graph创建多个Engine并配置映射关系）。

须知

如果用户想要传送文件，必须满足下列条件，否则系统会默认传递的是字符串或数字：

- Linux环境下，文件要包含相对路径或者绝对路径，例如：“/home/1.txt”或者“../test/2.txt”。注意如果字符串中使用了"\"或"/"符号可能会被误判为文件，请避免使用。
- Windows环境下，文件要包含相对路径或者绝对路径，例如：“c:\1.txt”，或者“..\test\2.txt”。注意如果字符串中使用了\"或"/"符号可能会被误判为文件，请避免使用

```
graphs {
  graph_id: 100
  device_id: "0"
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
```

```
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    so_name: "./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: 200
  device_id: "1"
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    internal_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
  }
}
```

8.2.7.4.2 性能优化传输示例

```
(1)使用性能优化方案传输数据，必须对要传输的数据进行手动序列化和反序列化：
// 注：序列化函数在发送端使用，反序列化在接收端使用，所以这个注册函数最好在接收端和发送端都注册一遍；

  结构体
typedef struct
{
    uint32_t frameId;
    uint8_t  bufferId;
    uint8_t* image_data;
    uint8_t  image_size;
}TEST_STR;

// 序列化TEST_STR结构体，该函数只需要调动注册函数进行注册，参数说明：
// 输入：inputPtr，TEST_STR结构体指针
// 输出：ctrlStr，控制信息地址
        imageData，数据信息指针
        imageLen，数据信息大小
void GetTestStrSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& imageData, uint32_t& imageLen)
{
    // 获取结构体buffer
    TEST_STR* test_str = (TEST_STR*)inputPtr;
    ctrlStr = std::string((char*)inputPtr, sizeof(TEST_STR));

    // 获取数据信息，直接将结构体的数据指针赋值传递
    imageData = (uint8_t*)test_str->image_data;
    imageLen = test_str->image_size;
}

// 反序列化结构体，回传回来的将是结构体buffer和数据块buffer
// 输入：ctrlPtr，控制信息地址
        ctrlLen，控制信息大小
        imageData，数据信息指针
        imageLen，数据信息大小
// 输出：std::shared_ptr<void>，指向结构体的智能指针
std::shared_ptr<void> GetTestStrDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen,const uint8_t*
imageData, const uint32_t& imageLen)
{
    // 获取结构体
    TEST_STR* test_str = (TEST_STR*)ctrlPtr;

    // 获取传输过来的大内存数据
    // 注：大内存数据最好赋值给智能指针，并注册删除器，如果不是使用智能指针或者没有注册删除器，则需要
在释放该内存时手动调用hiai::Graph::ReleaseDataBuffer(void* ptr);
    std::shared_ptr<TEST_STR<uint8_t>> shared_data = std::make_shared<TEST_STR<uint8_t>>();
    shared_data->frameId = test_str->frame_ID;
    shared_data->bufferId= test_str->bufferId;
    shared_data->image_size = imageLen;
    shared_data->image_data.reset(imageData, hiai::Graph::ReleaseDataBuffer);

    // 返回智能指针给到Device端Engine使用
    return std::static_pointer_cast<void>(shared_data);
}
// 注册序列化反序列化函数
HIAI_REGISTER_SERIALIZE_FUNC("TEST_STR", TEST_STR, GetTestStrSearPtr, GetTestStrDearPtr);

(2) 在发送数据时，需要使用注册的数据类型，另外配合使用HIAI_DMalloc分配数据内存，可以使性能更优
    注：在从host侧向Device侧搬运数据时，使用HIAI_DMalloc方式将很大的提供传输效率，建议优先使用
HIAI_DMalloc，该内存接口目前支持0 – (256M Bytes - 96 Bytes)的数据大小，如果数据超出该范围，则需要使
用malloc接口进行分配；
    // 使用Dmalloc接口申请数据内存，10000为时延，为10000毫秒，表示如果内存不足，等待10000毫秒；
    HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(width*align_height*3/2,(void*&)align_buffer, 10000);

    // 发送数据，调用该接口后无需调用HIAI_Dfree接口，10000为时延
    graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```

8.3 DVPP API 参考

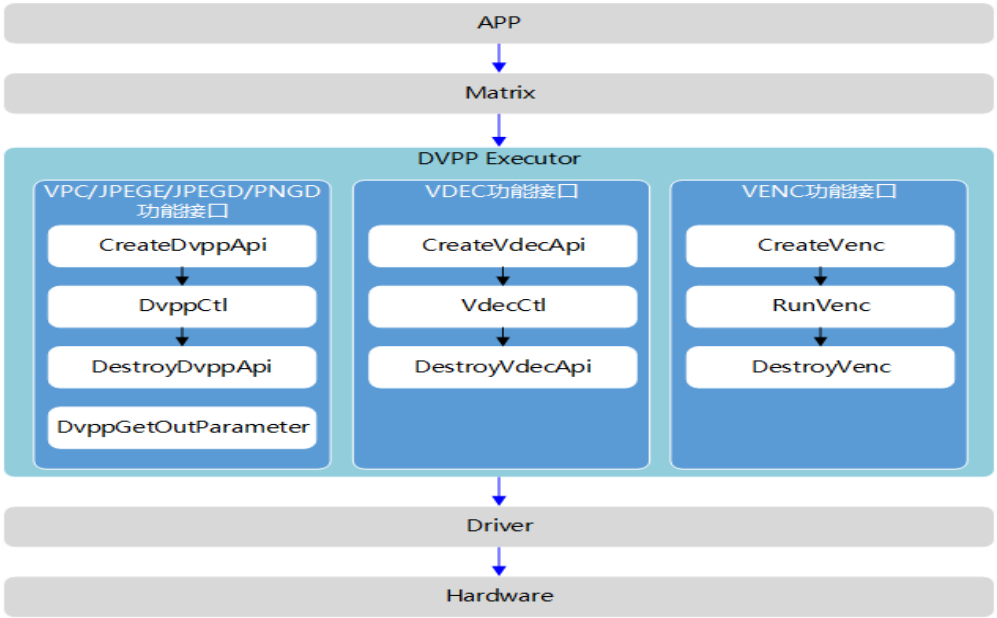
8.3.1 概述

8.3.1.1 接口简介

本文档详细描述了DVPP（Digital Vision Pre-Processing）执行器对外提供的接口，接口描述包括：接口函数描述、接口调用说明、整体示例等，适合开发人员、测试人员。

开发人员在开发APP时调用DVPP接口的流程如下：

图 8-1 流程图



- 当前版本调用**CreateDvppApi**、**DvppCtl**、**DestroyDvppApi**接口实现VPC功能有两种方式：
  - 方式一，在**DvppCtl**接口中传入入参：DVPP_CTL_VPC_PROC命令字和**VpcUserImageConfigure结构体**的参数。**推荐使用。**
  - 方式二，在**DvppCtl**接口中传入入参：DVPP_CTL_VPC_PROC命令字和**resize_param_in_msg结构体**的参数。对时延要求不高、处理的图片分辨率低、数量多的场景，可以在**DvppCtl**接口中传入入参：DVPP_CTL_CMDLIST_PROC和**IMAGE_CONFIG结构体**的参数。这种方式是为了兼容旧版本中的功能，后续版本会删除，**不推荐使用**。
- 当前版本实现VENC功能有两种方式：
  - 方式一，调用**CreateVenc**、**RunVenc**、**DestroyVenc**接口实现VENC功能，**推荐使用**。
  - 方式二，调用**CreateDvppApi**、**DvppCtl**、**DestroyDvppApi**接口实现VENC功能，在**DvppCtl**接口中传入入参：DVPP_CTL_VENC_PROC命令字和

venc_in_msg结构体的参数，这种方式在为了兼容旧版本中的功能，后续版本会删除，**不推荐使用**。

8.3.1.2 接口列表

您可以在DDK（Device Development Kit）的安装目录下的“ddk/include/inc/dvpp/”目录下查看接口的头文件。若需要调用DVPP提供的接口，在代码中可以包含idvppapi.h、Venc.h、Vpc.h（定义数据类型的头文件，请参见8.3.7.1 VpcUserImageConfigure中的结构体）。

表 8-7 接口列表

分类	接口	功能说明	定义接口的头文件
实现VPC/JPEGE/JPEGD/PNGD功能	CreateDvppApi	创建dvppapi实例，相当于获取DVPP执行器句柄，调用方可以使用申请到的dvppapi实例调用DvppCtl接口处理图片，可以跨函数调用，跨线程调用。	idvppapi.h
	DvppCtl	使用CreateDvppApi接口创建的实例来调用**DvppCtl**接口，控制DVPP各模块执行，模块主要包括VPC（Vision Pre-processing Core）、JPEGE、JPEGD、PNGD等。	
	DestroyDvppApi	销毁由CreateDvppApi接口创建的dvppapi实例，关闭DVPP执行器。	
	DvppGetOutParameter	获取JPEGD/JPEGE/PNGD模块的输出内存大小。	
实现VDEC功能	CreateVdecApi	获取vdecapi实例，相当于vdec执行器句柄。调用方可以使用申请到的vdecapi实例调用CreateVdecApi接口进行视频解码，可以跨函数调用，跨线程调用。	idvppapi.h

分类	接口	功能说明	定义接口的头文件
	VdecCtl	使用CreateVdecApi接口创建的实例调用VdecCtl接口，控制DVPP执行器进行视频解码。	
	DestroyVdecApi	释放由CreateVdecApi接口创建的vdecapi实例，关闭VDEC执行器。	
实现VENC功能	CreateVenc	获取VENC编码实例，相当于获取VENC执行器句柄。调用方可以使用申请到的VENC编码实例调用RunVenc接口进行图片编码。	Venc.h
	RunVenc	使用CreateVenc接口创建的实例调用RunVenc接口，控制DVPP执行器进行视频编码。	
	DestroyVenc	释放由CreateVenc接口创建的VENC编码实例，关闭VENC执行器。	

8.3.1.3 VPC 功能

功能说明

VPC功能包括：

- **抠图**，从输入图片中抠出需要用的图片区域。
- **缩放**
  - 针对不同分辨率的图像，VPC的处理方式可分为非8K缩放、8K缩放。
    - 8K缩放，用于处理输入图像宽度在4096~8192范围内或高度在4096~8192范围内的图片。
    - 非8K缩放，用于处理输入图像分辨率在32*6~4096*4096范围内的图片。
  - 从是否抠多张图的维度，可分为单图裁剪缩放（支持非压缩格式和HFBC压缩格式）、一图多框裁剪缩放（支持非压缩格式和HFBC压缩格式）。

    HFBC，是VDEC输出的一种压缩图像格式，使用这种方式压缩图像，VDEC的处理性能更优。
  - 其它缩放方式，如：原图缩放、等比例缩放。

- **叠加**，从输入图片中抠出来的图，对抠出的图进行缩放后，放在用户输出图片的指定区域，输出图片可以是空白图片（由用户申请的空输出内存产生的），也可以是已有图片（由用户申请输出内存后将已有图片读入输出内存），只有当输出图片是已有图片时，才表示叠加。
- **拼接**，从输入图片中抠多张图片，对抠出的图进行缩放后，放到输出图片的指定区域。
- **格式转换**
  - 将RGB格式/YUV422格式/YUV444格式的图片转为YUV420格式的图片。
  - 图像灰度化，对输出图像数据只取Y分量的数据。

约束说明

- 针对不同分辨率的图像，VPC的处理方式包括8K缩放、非8K缩放，如下表所示。

输入图像分辨率	输入图像格式	输入图像宽stride*高stride对齐要求	VPC功能	输出图像分辨率	输出图像格式	输出图像宽stride*高stride对齐要求
宽度在4096~8192范围内或高度在4096~8192范围内（不包括4096）	YUV420SP（NV12、NV21）	2*2对齐	8K缩放	16*16~4096*4096	请参见表8-10处的outputFormat参数	2*2对齐
32*6~4096*4096（包括4096）	请参见表8-10处的inputFormat参数	• 宽stride的对齐：请参见表8-10处的widthStride参数 • 高stride是2对齐	非8K缩放	32*6~4096*4096	请参见表8-10处的outputFormat参数	16*2对齐

- 针对缩放功能，宽高缩放比例范围：[1/32, 16]。
- 对于8K缩放功能，支持缩放、支持YUV420SP NV12与YUV420SP NV21之间的格式转换、不支持抠图。

功能示意图

图 8-2 VPC 功能示意图（抠图+缩放+叠加）

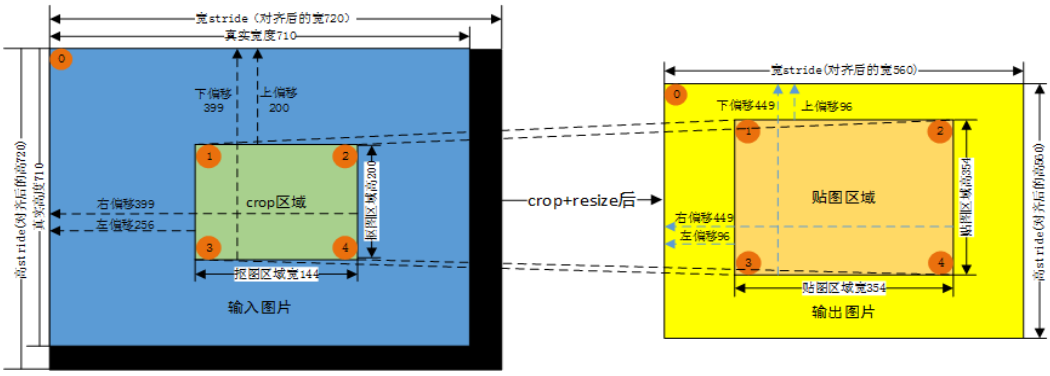


图 8-3 VPC 功能示意图（拼接）

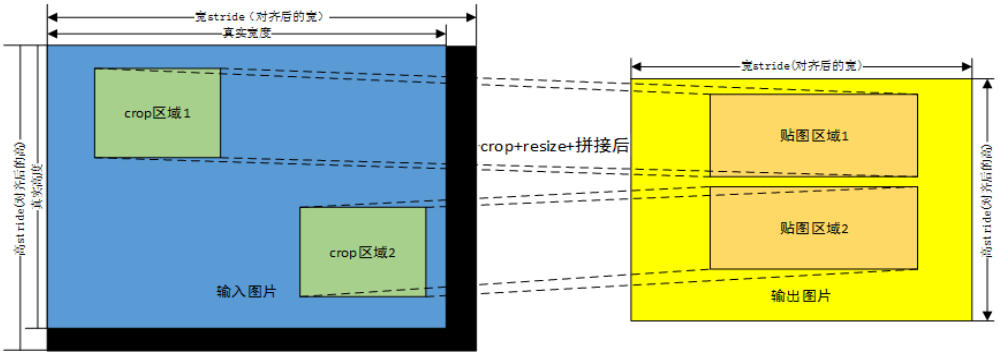


表 8-8 概念解释

概念	描述
宽stride	指一行图像步长，表示输入图片对齐后的宽，RGB格式或YUV格式的宽stride计算方式不一样。宽stride最小为32，最大为4096 * 4（即宽是4096的argb格式的图像）。 ● yuv400sp、yuv420sp、yuv422sp、yuv444sp：输入图像的宽对齐到16。 ● yuv422packed：输入图像的宽*2后再对齐到16后。 ● yuv444packed、rgb888：输入图像的宽对齐*3后再对齐到16。 ● xrgb8888：输入图像的宽*4后再对齐到16。 ● HFBC格式：输入图像的宽。
高stride	指图像在内存中的行数，表示输入图片对齐后的高。 取值为：输入图像的高对齐到2。高stride最小为6，最大为4096。

概念	描述
上/下/左/右偏移	通过配置上偏移、下偏移、左偏移、右偏移可以实现两个功能：指定抠图区域或贴图区域的位置；控制抠图或贴图区域的宽、高，右偏移-左偏移+1=宽，下偏移-上偏移+1=高。 ● 左偏移：输入/输出图片中，抠图/贴图区域1、3两个点相对于0点水平向左偏移的值。 ● 右偏移：输入/输出图片中，抠图/贴图区域2、4两个点相对于0点水平向左偏移的值。 ● 上偏移：输入/输出图片中，抠图/贴图区域1、2两个点相对于0点垂直向上偏移的值。 ● 下偏移：输入/输出图片中，抠图/贴图区域3、4两个点相对于0点垂直向上偏移的值。
抠图区域	指用户指定的需裁剪的图片区域。 抠图区域最小分辨率为10*6，最大分辨率为4096*4096。
贴图区域	指在输出图片中用户指定的区域，贴图区域最小分辨率为10*6，最大分辨率为4096*4096。 约束如下： ● 贴图区奇数、偶数限制为：左偏移和上偏移为偶数、右偏移和下偏移为奇数。 ● 抠图区域不超出输入图片，贴图区域不超出输出图片。 ● 贴图时可直接放置在输出图片的最左侧，即相对输出图片的左偏移为0。 ● 最大贴图个数为256个。 ● 贴图区域相对输出图片的左偏移16对齐。 ● 输出图片的贴图宽度建议16对齐，如果不是16对齐，会多写一段无效数据使其16对齐。

性能指标说明

- 对于**非8K缩放**，VPC性能由于涉及到抠图、缩放等不同的场景，当处理图片的过程中分辨率改变时，以大分辨率来计算性能。例如，对于缩放，缩放后图片的分辨率大于缩放前图片的分辨率，则以缩放后图片的分辨率来计算性能指标；对于抠图，贴图区域的分辨率大于抠图区域的分辨率，则以贴图区域的分辨率来计算性能指标。对于YUV420 SP格式的图片，典型场景性能指标参考如下：

场景举例	总帧率
1080p * n路（n<4，1路对应一个线程）	n*360fps
1080p * n路（n>=4，1路对应一个线程）	1440fps
4k * n路（n<4，1路对应一个线程）	n*90fps

场景举例	总帧率
4k * n路（n>=4，1路对应一个线程）	360fps

- 对于**8K缩放**，VPC性能与输出分辨率强相关，输出分辨率越大，处理耗时越久，性能越低，典型场景（输出分辨率为1080p、4k的场景，图片格式为YUV420 SP）性能指标参考如下：

场景举例	总帧率
1080p * n路（n=1，1路对应一个线程）	4fps
1080p * n路（n>=4，1路对应一个线程）	16fps
4k * n路（n=1，1路对应一个线程）	1fps
4k * n路（n>=4，1路对应一个线程）	4fps

参考说明

RBG、YUV格式图像的各分量排布示意图。示例：sp图像以yuv420sp为例，packed和rgb图像以argb图像为例。

格式	分辨率	宽stride	高stride	buffer大小					
yuv420sp	4*4	4	4	24					
内存排列									
y11	y12	y13	y14						
y21	y22	y23	y24						
y31	y32	y33	y34						
y41	y42	y43	y44						
u11	v11	u13	v13						
u31	v31	u33	v33						
格式	分辨率	宽stride	高stride	buffer大小					
yuv420sp	4*4	6	6	54					
内存排列	x为无效数据								
y11	y12	y13	y14	x	x				
y21	y22	y23	y24	x	x				
y31	y32	y33	y34	x	x				
y41	y42	y43	y44	x	x				
x	x	x	x	x	x				
x	x	x	x	x	x				
u11	v11	u13	v13	x	x				
u31	v31	u33	v33	x	x				
x	x	x	x	x	x				
格式	分辨率	宽stride	高stride	buffer大小					
argb	2*2	10	4	40					
内存排列	x为无效数据								
a11	r11	g11	b11	a12	r12	g12	b12	x	x
a21	r21	g21	b21	a22	r22	g22	b22	x	x
x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x

8.3.1.4 JPEGE 功能

功能及约束说明

JPEGE功能：实现编码生成.jpg图片。

- JPEGE支持以下格式输入：
  - YUV422 packed（yuyv,yvyu,uyvy,vyuy）
  - YUV420 Semi-planar（NV12，NV21）
- JPEGE支持的分辨率为：
  最大分辨率：8192 * 8192，最小分辨率：32 * 32
- JPEGE输出格式为jpeg，只支持huffman编码，不支持算术编码，不支持渐进编码。

性能指标说明

场景举例	总帧率
1080p * n路（n>=1）	64fps
4k * n路（n>=1）	16fps

8.3.1.5 JPEGD 功能

功能说明

实现.jpg、.jpeg、.JPG、.JPEG图片的解码，对于硬件不支持的格式，会使用软件解码。

解码后，输出如下格式的图片：

jpeg(444) -> yuv444 / yuv420半平面V在前U在后。

jpeg(422) -> yuv422 / yuv420半平面V在前U在后。

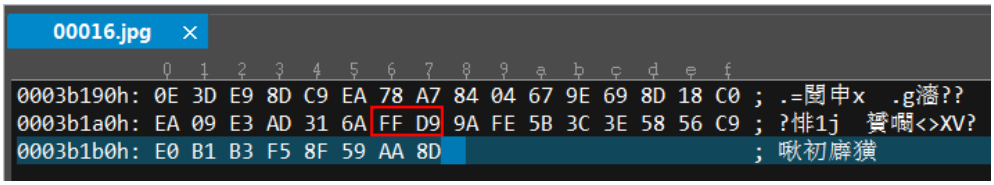
jpeg(420) -> yuv420半平面V在前U在后。

jpeg(400) -> yuv420，uv数据采用0 x 80填充。

须知

- JPEGD由于编程规范要求驼峰风格命名方式，原内核风格的入参和出参继续支持，同时提供新的驼峰风格参数供调用方使用，所以JPEGD暂时支持两套参数供用户使用，推荐使用驼峰风格参数。
- 若图片内EOI（End Of Image，标记代码为0XFFD9）之后，还有用户自定义的数据，则JPEGD在对图片进行解码时，会直接清零EOI之后的8字节数据，若用户需要保留这些自定义的数据，则将图片数据读入内存之后，需要提前备份这部分数据，再传给JPEGD处理。

  若需要查看图片内EOI之后是否存在自定义数据，可以使用二进制查看工具打开图片查看，例如下图中的FFD9标记符之后就存在自定义数据。



约束说明

- 关于输入图片的约束：
  - 最大分辨率：8192 * 8192，最小分辨率：32 * 32
  - 只支持Huffman编码，码流的colorspace为YUV，码流的subsample为444/422/420/400；
  - 不支持算术编码；
  - 不支持渐进JPEG格式；
  - 不支持JPEG2000格式；
- 关于硬件约束：
  - 最多支持4张Huffman表，其中包括2 张DC（直流）表和2 张AC（交流）表；
  - 最多支持3张量化表；
  - 只支持8bit采样精度；

  - 只支持对顺序式编码的图片进行解码；
  - 只支持基于DCT（Discrete Cosine Transform）变换的JPEG 格式解码；
  - 只支持一个SOS（Start of Scan）标志的图片解码。
- 关于软件约束：
  - 支持3个SOS标志的图片解码；
  - 支持mcu（Minimum Coded Unit）数据不足的异常图片解码。

性能指标说明

JPEGD性能指标是基于硬件解码的性能，JPEGD硬件解码不支持3个SOS的图片解码，对于硬件不支持的格式，会使用软件解码，软件解码性能参考为1080P*1路 15fps。

场景举例	总帧率
1080p * 1路	128fps
1080p * n路（n>=2）	256fps
4k * 1路	32fps
4k * n路（n>=2）	64fps

8.3.1.6 PNGD 功能

功能及约束说明

PNGD功能：实现PNG格式图片的硬件解码。

- PNGD支持以下两种输入格式：
  RGBA、RGB
- PNGD输出格式为：
  RGBA、RGB
- PNGD支持的分辨率
  最大分辨率4096 * 4096，最小分辨率32 * 32。

性能指标说明

场景举例	总帧率
1080p * 1路	4fps
1080p * 2路	8fps
1080p * 3路	12fps
1080p * 4路	16fps
1080p * 5路	20fps
1080p * n路（n>= 6）	24fps

场景举例	总帧率
4k * 1路	1fps
4k * 2路	2fps
4k * 3路	3fps
4k * 4路	4fps
4k * 5路	5fps
4k * n路（n>= 6）	6fps

8.3.1.7 VDEC 功能

功能及约束说明

实现视频的解码。

- 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支持的分辨率
  最大分辨率4096 * 4096，最小分辨率128 * 128。
- VDEC输出格式为：yuv420sp压缩后的HFBC数据。
  HFBC，是VDEC输出的一种压缩图像格式，使用这种方式压缩图像，VDEC的处理性能更优。
- 若码流中有坏帧、缺帧等情况，解码器VDEC可能会丢帧。
- 通过隔行扫描方式编码出来的码流，VDEC不支持解码。
- VDEC比较特殊，由于当前内部不是单例模式，所以其对外提供C接口函数与其他不同。

性能指标说明

场景举例	总帧率
1080p n路（n>=2且n<=16）	480fps
1080p *1路	240fps
4k * n路（n>=2且n<=16）	120fps
4k * 1路	60fps

如果用户使用Matrix模块编排应用的流程，那么在使用DVPP对多路（Channel）输入的视频流进行解码时，由于视频流中每帧的数据之间有关联关系，因此为保证同一个视频流中每帧数据的顺序，要求每个VDEC固定对应一路视频流，否则DVPP中的VDEC无法解码视频流。

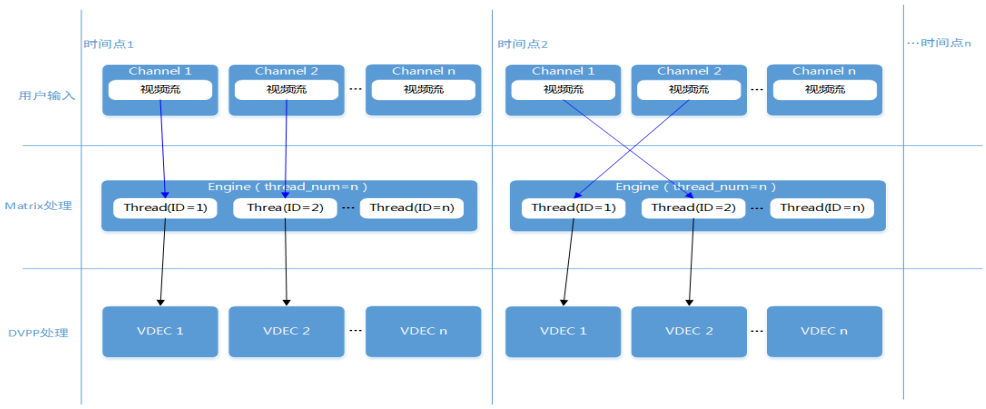
- 多路视频流解码时，为保证视频流中每帧数据的顺序，可能存在多种实现方式，推荐使用以下配置：
  - 在Graph配置文件中，在graphs配置段内，配置多个进行视频解码的Engine，一路视频流对应一个Engine。
  - 在Graph配置文件中，在视频解码的engine配置段内，将thread_num配置为1，一个Engine对应一个线程。

  说明

  - 关于Matrix的功能及其Graph配置文件的配置，请参见《 Matrix API参考 》。
  - 一个Graph（一个进程）最大可以对应16路的视频流。

- 如果在多路视频流解码时，在Graph配置文件中，仅配置一个视频解码的Engine，并将thread_num配置为n（n的值就是视频流channel的数量），在不同时刻，在Matrix中各路视频流可能会在不同的Thread处理，但是在DVPP中，一个VDEC需要固定一路视频流数据，一路视频流数据对应一个线程，这样可能导致VDEC解码时无法保证视频流中每帧数据的顺序，如图所示。

图 8-4 VDEC 处理流程



8.3.1.8 VENC 功能

VENC将编码分成实例化、编码、释放资源三小步。

功能及约束说明

实现YUV420/YVU420图片数据的编码。

- VENC支持以下格式输入：

  YUV420 semi-planar NV12/NV21-8bit

- VENC支持的分辨率

  最大分辨率1920 * 1920，最小分辨率128 * 128。

- VENC输出格式为：

  H264 BP/MP/HP

  H265 MP（仅支持Slice码流）

性能指标说明

场景举例	总帧率
1080p * n路（一个进程对应一路）	30fps

注：其它分辨率可以等量估算。

8.3.1.9 关于输入输出内存的说明

关于HIAI_DMalloc/HIAI_DFree接口、HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口的说明，以及Graph配置文件（receive_memory_without_dvpp参数）的说明，请参见《Matrix API参考》。

表 8-9 内存要求说明

功能模块	输入内存	输出内存
VPC	请使用Matrix提供的接口申请/释放内存： ● 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址16对齐）； ● 使用HIAI_DVPP_DFree接口释放内存。	请使用Matrix提供的接口申请/释放内存： ● 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址16对齐）； ● 使用HIAI_DVPP_DFree接口释放内存。
JPEGE和JPEGD	请使用Matrix提供的接口申请/释放内存： ● 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址128对齐）； ● 使用HIAI_DVPP_DFree接口释放内存。	● 由用户指定输出内存时，由用户自行释放内存 。请使用Matrix提供的接口申请/释放内存： – 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址128对齐）。在申请内存前，可以调用**DvppGetOutParameter**接口获取输出内存大小。 – 使用HIAI_DVPP_DFree接口释放内存。 ● 不由用户指定输出内存时，DVPP内部申请内存，需由用户调用JPEGE/JPEGD出参结构体中的cbFree()回调函数释放内存，并将内存地址指针置为空。

功能模块	输入内存	输出内存
PNGD	请使用Matrix提供的接口申请/释放内存： ● 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址128对齐）； ● 使用HIAI_DVPP_DFree接口释放内存。	由用户指定输出内存时，由用户自行释放内存 。请使用Matrix提供的接口申请/释放内存： ● 使用HIAI_DVPP_DMalloc接口申请内存，内存满足DVPP的要求（内存起始地址128对齐）。在申请内存前，可以调用**DvppGetOutParameter**接口获取输出内存大小。 ● 使用HIAI_DVPP_DFree接口释放内存。
VDEC和VENC	对内存无要求，支持调用malloc/free、new/delete等原生接口申请/释放内存，也支持调用Matrix提供的HIAI_DMalloc/HIAI_DFree接口、HIAI_DVPP_DMalloc/HIAI_DVPP_DFree接口申请/释放内存，还支持通过Matrix提供的receive_memory_without_dvpp参数控制内存是否给DVPP使用。	VDEC输出的HFBC格式数据直接作为VPC的输入。 VENC输出内存是DVPP内部管理，用户在使用时可以拷贝输出内存中的数据。

8.3.2 VPC/JPEGE/JPEGD/PNGD 功能接口

8.3.2.1 CreateDvppApi

函数原型	**int CreateDvppApi(IDVPPAPI*& pIDVPPAPI)**
功能	创建dvppapi实例，相当于获取DVPP执行器句柄，调用方可以使用申请到的dvppapi实例调用**DvppCtl接口**处理图片，可以跨函数调用，跨线程调用。
输入说明	输入为“IDVPPAPI”类型指针引用，输入指针必须为“NULL”。
输出说明	输出为“IDVPPAPI”类型指针引用，获取失败则为NULL，获取成功则不为NULL。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	调用方创建**“IDVPPAPI”**对象指针，初始化为NULL，调用**“CreateDvppApi”**函数将**“IDVPPAPI”**对象指针传入。如果申请成功，**“CreateDvppApi”**函数会返回dvppapi实例，否则返回NULL。调用方需要对返回值进行校验。

使用约束	调用方负责dvppapi实例的生命周期，即申请与释放，申请使用**CreateDvppApi**接口，释放使用**DestroyDvppApi接口**。

调用示例

```
IDVPPAPI *pidvppapi = NULL;
CreateDvppApi(pidvppapi);
```

8.3.2.2 DvppCtl

8.3.2.2.1 接口说明

函数原型	**int DvppCtl(IDVPPAPI*& pIDVPPAPI, int CMD, dvppapi_ctl_msg* MSG)**
功能	使用**CreateDvppApi**接口创建的实例来调用**DvppCtl**接口，控制DVPP各模块执行，模块主要包括VPC（Vision Pre-processing Core）、JPEGE、JPEGD、PNGD等。
输入说明	• **“IDVPPAPI”**类型指针引用 • **CMD控制命令字** • **dvppapi_ctl_msg**类型的DVPP执行器配置信息MSG。DVPP各个模块的配置信息不同，因此调用各模块功能时需传入相应的配置信息。
输出说明	根据DVPP各模块的功能，出参也不一样，请参见**dvppapi_ctl_msg**。
返回值说明	• 返回值“0”代表成功。 • 返回值“-1”代表失败。
使用说明	调用方调用**“DvppCtl”**函数，传入**“IDVPPAPI”**对象指针、传入正确的**CMD控制命令字**、配置好相应功能的**dvppapi_ctl_msg**。
使用约束	无。

CMD 控制命令字

CMD控制命令字在DDK安装目录下的“ddk/include/inc/dvpp/DvppCommon.h”文件中定义。

成员变量	说明	取值范围
DVPP_CTL_VPC_PROC	VPC功能控制命令字。	0
DVPP_CTL_PNGD_PROC	PNGD功能控制命令字。	1
DVPP_CTL_JPEGE_PROC	JPEGE功能控制命令字。	2

成员变量	说明	取值范围
DVPP_CTL_JPEGD_PROC	JPEGD功能控制命令字。	3
DVPP_CTL_VENC_PROC	预留命令字。	5
DVPP_CTL_DVPP_CAPABILITY	查询DVPP能力控制命令字。	6
DVPP_CTL_CMDLIST_PROC	预留命令字。	7
DVPP_CTL_TOOL_CASE_GET_RESIZE_PARAM	预留命令字。	8

dvppapi_ctl_msg

该类型在DDK安装目录下的“ddk/include/inc/dvpp/DvppCommon.h”文件中定义。

成员变量	说明	取值范围
int32_t in_size	入参大小。	-
int32_t out_size	出参大小。	-
void *in	入参。	• VPC功能入参：**VpcUserImageConfigure** • JPEGE功能入参：**sJpegeIn** • JPEGD功能入参：**JpegdIn**（Jpegd输入结构体的驼峰风格版本，推荐使用） • JPEGD功能入参：**jpegd_raw_data_info**（jpegd输入结构体的内核风格版本，不推荐使用） • PNGD功能入参：**PngInputInfoAPI** • 查询DVPP引擎入参：**device_query_req_stru**
void *out	出参。	• VPC功能出参：无 • JPEGE功能出参：**sJpegeOut** • JPEGD功能出参：**JpegdOut**（Jpegd输出结构体的驼峰风格版本，推荐使用） • JPEGD功能出参：**jpegd_yuv_data_info**（jpegd输出结构体的内核风格版本，不推荐使用） • PNGD功能出参：**PngOutputInfoAPI** • 查询DVPP引擎出参：**dvpp_engine_capability_stru**

8.3.2.2.2 VPC 参数说明

入参：VpcUserImageConfigure

表 8-10 入参 VpcUserImageConfigure

成员变量	说明	取值范围
uint8_t* bareDataAddr	非压缩格式输入图片地址，当图像为压缩格式时，不需要填，默认为NULL。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址16对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。	-
uint32_t bareDataBufferSize	非压缩格式输入图片内存大小，其中大小是根据图像的宽高stride计算出来的，bareDataAddr指向内存大小应该等于bareDataBufferSize，当图像为压缩格式时，不需要填，默认为0。	非压缩格式，对于不同图像格式，内存大小的计算公式如下： ● yuv400sp、yuv420sp：widthStride*heightStride*3/2 ● yuv422sp：widthStride*heightStride*2 ● yuv444sp：widthStride*heightStride*3 ● yuv422packed：widthStride*heightStride ● yuv444packed、rgb888：widthStride*heightStride ● xrgb8888：widthStride*heightStride

成员变量	说明	取值范围
uint32_t widthStride	图像宽度方向的步长。	宽stride最小为32，最大为4096 * 4（即宽是4096的argb格式的图像）。 对于8K缩放，要求宽stride对齐到2；对于非8K缩放，对于不同图像格式，widthStride的计算公式不同： ● yuv400sp、yuv420sp、yuv422sp、yuv444sp：输入图像的宽对齐到16。 ● YUV422packed：输入图像的宽对齐到16后，再乘以2（宽16对齐，每个像素占2个字节）。 ● YUV444packed、RGB888：输入图像的宽对齐到16，再乘以3（宽16对齐，每个像素占3个字节）。 ● XRGB8888：输入图像的宽对齐到16后，再乘以4（宽16对齐，每个像素占4个字节）。 ● HFBC格式：输入图像的宽。
uint32_t heightStride	图像高度方向的步长，对于yuv sp图像根据该参数计算uv数据的起始地址。	取值为：输入图像的高对齐到2。高stride最小为6，最大为4096。

成员变量	说明	取值范围
enum VpcInputFormat inputFormat	输入图像格式。	enum VpcInputFormat { INPUT_YUV400, // 0 INPUT_YUV420_SEMI_PLANNER_UV, // 1 INPUT_YUV420_SEMI_PLANNER_VU, // 2 INPUT_YUV422_SEMI_PLANNER_UV, // 3 INPUT_YUV422_SEMI_PLANNER_VU, // 4 INPUT_YUV444_SEMI_PLANNER_UV, // 5 INPUT_YUV444_SEMI_PLANNER_VU, // 6 INPUT_YUV422_PACKED_YUYV, // 7 INPUT_YUV422_PACKED_UYVY, // 8 INPUT_YUV422_PACKED_YVYU, // 9 INPUT_YUV422_PACKED_VYUY, // 10 INPUT_YUV444_PACKED_YUV, // 11 INPUT_RGB, // 12，输入图像格式为RGB888 INPUT_BGR, // 13，输入图像格式为BGR888 INPUT_ARGB, // 14，此格式的输入图像在存储时各分量的排列顺序类似RGB888，其中，A表示透明度 INPUT_ABGR, // 15，此格式的输入图像在存储时各分量的排列顺序类似BGR888，其中，A表示透明度 INPUT_RGBA, // 16，此格式的输入图像在存储时各分量的排列顺序类似RGB888，其中，A表示透明度 INPUT_BGRA, // 17，此格式的输入图像在存储时各分量的排列顺序类似BGR888，其中，A表示透明度

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
		INPUT_YUV420_SEMI_PLANNER_UV_10BIT, // 18 INPUT_YUV420_SEMI_PLANNER_VU_10BIT, // 19 };

enum VpcOutputFormat outputFormat	输出图像格式。	enum VpcOutputFormat {
		OUTPUT_YUV420SP_UV,
		OUTPUT_YUV420SP_VU
		};

VpcUserRoiConfigure* roiConfigure	抠图区域配置，详细请参见 **•VpcUserRoiConfigure结构体**。	-
bool isCompressData	是否是视频解码输出的HFBC压缩图片数据格式。	当图片来源于VDEC时，需要配置为true，其他格式为false，默认为false。 单图裁剪缩放和一图多框裁剪缩放均支持HFBC压缩格式。
VpcCompressDataConfigure compressDataConfigure	视频解码输出的HFBC压缩图片数据配置。详细请参见 **•VpcCompressDataConfigur...**。	-
bool yuvSumEnable	是否需要计算yuvSum值，当需要计算该值时只支持一图一框。	当需要计算yuvSum值时配置true，其他为false，默认为false。 统计yuvSum值的约束条件： 1，输入图像分辨率小于等于4096*4096 2，贴图区域起始点是（0,0）位置 3，输入只有一图一框
VpcUserYuvSum yuvSum	yuvSum计算配置。详细请参见 **•VpcUserYuvSum结构体**。	-
VpcUserPerformanceTuningParameter tuningParameter	预留参数，用于参数调优，详细请参见 **•VpcUserPerformanceTunin...**。	-

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
uint32_t* cmdListBufferAddr	预留参数。	-
uint32_t cmdListBufferSize	预留参数。	-
uint64_t yuvScalerParaSetAddr	预留参数，滤波参数集文件路径地址和文件名数组。 **说明** ● 参数集须在Device设备上。 ● 请确保传入的文件路径是正确路径。	参数集文件路径为device侧的绝对文件路径＋文件名，如{"/root/share/vpc/YUVScaler_pra.h"}
uint16_t yuvScalerParaSetSize	预留参数，滤波参数集地址指向的参数集数组元素个数（默认为1）。	1<=yuvscaler_paraset_size<=10
uint16_t yuvScalerParaSetIndex	预留参数，yuvScalerParaSetIndex变量对应yuvScalerParaSetAddr的索引（默认为0）。	0<=yuvScalerParaSetIndex<10
uint8_t isUseMultiCoreAccelerate	预留参数。	-
uint8_t reserve1	预留参数。	-
uint16_t reserve2;	预留参数。	-

出参说明

暂无出参。

8.3.2.2.3 JPEGE 参数说明

入参：sJpegeIn

表 8-11 入参 sJpegeIn

成员变量	说明
eEncodeFormat format	输入YUV数据的类型，支持YUV422 packed（yuyv,yvyu,uyvy,vyuy）和YUV420 Semi-planar（NV12，NV21） typedef enum { JPGENC_FORMAT_UYVY = 0x0, JPGENC_FORMAT_VYUY = 0x1, JPGENC_FORMAT_YVYU = 0x2, JPGENC_FORMAT_YUYV = 0x3, JPGENC_FORMAT_NV12 = 0x10, JPGENC_FORMAT_NV21 = 0x11, } eEncodeFormat;
unsigned char* buf	yuv输入数据，需要调用方申请，需要合理对齐，见参数stride，heightAligned。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数bufSize的参数值一致。
uint32_t bufSize	输入buf长度，指宽高对齐后的数据长度。
uint32_t width	输入图片的宽度，范围[32,8192]。
uint32_t height	输入图片的高度，范围[32,8192]。
uint32_t stride	输入图片对齐后宽度，对齐到16，兼容对齐到16的倍数如128。对于YUV422packed数据，stride应该为width的两倍对齐到16。
uint32_t heightAligned	1. 支持与高度height相同，2.支持输入图片对齐后高度（Matrix异侧传输后的对齐），3.兼容高度向16对齐的数。
uint32_t level	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。

出参：sJpegeOut

表 8-12 出参 sJpegeOut

成员变量	说明
unsigned char* jpgData	输出缓冲区中jpg数据的起始地址。 如果由用户指定内存，使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数jpgSize的参数值一致。
uint32_t jpgSize	编码后的jpg图片数据长度。
JpegCalBackFree cbFree	释放输出内存的回调函数。 ● 由用户指定输出内存时，由用户自行释放内存 。 ● 不由用户指定输出内存时，DVPP内部申请内存，需由用户调用cbFree()回调函数释放内存，并将jpgData置为空指针。调用示例请参见**8.3.6.2 实现JPEGE功能**。

8.3.2.2.4 JPEGD 参数说明

入参：JpegdIn

表 8-13 入参 JpegdIn

成员变量	说明
unsigned char* jpegData	输入jpg图片数据，起始地址128对齐，2M大页表方式申请。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数jpegDataSize的参数值一致。
uint32_t jpegDataSize	输入内存的长度。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存大小为“实际数据大小+8 byte”，8 byte为硬件约束要求。

成员变量	说明
bool isYUV420Need	是否需要输出yuv420 semi-planar格式的数据。 ● true：是 ● false：否，保持源格式输出。 JPEGD支持raw格式（包括yuv420sp、yuv422sp、yuv444sp）和降采样为yuv420的半平面格式输出。其中灰度图片输出的yuv420为fake420形式。
bool isVBeforeU	该参数值只能配置为true，v在u前，预留字段。

出参：JpegdOut

表 8-14 出参 JpegdOut

成员变量	说明
unsigned char* yuvData	输出yuv图片数据buf，该图片的宽高为128*16对齐后的宽高。 如果由用户指定内存，使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《Matrix API参考》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数yuvDataSize的参数值一致。
uint32_t yuvDataSize	输出yuv数据的长度，数据长度由对齐后的宽高计算，也可以通过调用**DvppGetOutParameter接口**获取数据长度。
uint32_t imgWidth	输出yuv图片的宽度。
uint32_t imgHeight	输出yuv图片的高度。
uint32_t imgWidthAligned	输出图片的对齐后的宽度，对齐到128。
uint32_t imgHeightAligned	输出图片的对齐后的高度，对齐到16。
JpegCalBackFree cbFree	释放输出内存的回调函数。 ● 由用户指定输出内存时，由用户自行释放内存。 ● 不由用户指定输出内存时，DVPP内部申请内存，需由用户调用cbFree()回调函数释放内存，并将yuvData置为空指针。调用示例请参见**8.3.6.3 实现JPEGD功能**。

成员变量	说明

jpegd_color_space outFormat	输出yuv数据格式：
	enum jpegd_color_space{
	DVPP_JPEG_DECODE_OUT_UNKNOWN = -1,
	DVPP_JPEG_DECODE_OUT_YUV444 = 0,
	DVPP_JPEG_DECODE_OUT_YUV422_H2V1 = 1, /* YUV422 */
	DVPP_JPEG_DECODE_OUT_YUV422_H1V2 = 2, /* YUV440 */
	DVPP_JPEG_DECODE_OUT_YUV420 = 3,
	DVPP_JPEG_DECODE_OUT_YUV400 = 4,
	DVPP_JPEG_DECODE_OUT_FORMAT_MAX,
	};

入参：jpegd_raw_data_info

表 8-15 入参 jpegd_raw_data_info

成员变量	说明
unsigned char* jpeg_data	输入jpg图片数据，起始地址128对齐，2M大页表方式申请。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数jpeg_data_size的参数值一致。
uint32_t jpeg_data_size	输入内存的长度。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存大小为“实际数据大小+8 byte”，8 byte为硬件约束要求。

成员变量	说明

jpegd_raw_format in_format	输入图片中yuv的采样格式，不需要填充，默认值即可。
	enum jpegd_raw_format{
	DVPP_JPEG_DECODE_RAW_YUV_UNSUPPORT = -1,
	DVPP_JPEG_DECODE_RAW_YUV444 = 0,
	DVPP_JPEG_DECODE_RAW_YUV422_H2V1 = 1, // 422
	// yuv440 不再支持，保留字段
	DVPP_JPEG_DECODE_RAW_YUV422_H1V2 = 2, // 440
	DVPP_JPEG_DECODE_RAW_YUV420 = 3,
	DVPP_JPEG_DECODE_RAW_YUV400 = 4,
	DVPP_JPEG_DECODE_RAW_MAX,
	};

bool IsYUV420Need	是否需要输出yuv420 semi-planar格式的数据。 ● true：是 ● false：否，保持源格式输出。 JPEGD支持raw格式（包括yuv420sp、yuv422sp、yuv444sp）和降采样为yuv420的半平面格式输出。其中灰度图片输出的yuv420为fake420形式。
bool isVBeforeU	该参数值只能配置为true，v在u前，预留字段。

出参：jpegd_yuv_data_info

表 8-16 出参 jpegd_yuv_data_info

成员变量	说明
unsigned char* yuv_data	输出yuv图片数据buf，该图片的宽高为对齐后的宽高。 **须知** 该内存是由DVPP内部申请并管理，不能由用户指定。
uint32_t yuv_data_size	输出yuv数据的长度，数据长度由对齐后的宽高计算。
uint32_t img_width	输出yuv图片的宽度。
uint32_t img_height	输出yuv图片的高度。
uint32_t img_width_aligned	输出图片的对齐后的宽度，对齐到128。

成员变量	说明
uint32_t img_height_aligned	输出图片的对齐后的高度，对齐到16。
JpegCalBackFree cbFree	释放输出内存的回调函数。 DVPP内部申请内存，需由用户调用cbFree()回调函数释放内存，并将yuv_data置为空指针。调用示例请参见**8.3.6.3 实现JPEGD功能**。

enum jpegd_color_space out_format	输出yuv数据格式：
	enum jpegd_color_space {
	DVPP_JPEG_DECODE_OUT_UNKNOWN = -1,
	DVPP_JPEG_DECODE_OUT_YUV444 = 0,
	DVPP_JPEG_DECODE_OUT_YUV422_H2V1 = 1,// 422
	// yuv440 不再支持，保留字段
	DVPP_JPEG_DECODE_OUT_YUV422_H1V2 = 2,// 440
	DVPP_JPEG_DECODE_OUT_YUV420 = 3,
	DVPP_JPEG_DECODE_OUT_YUV400 = 4,
	DVPP_JPEG_DECODE_OUT_FORMAT_MAX,
	};

8.3.2.2.5 PNGD 参数说明

入参：PngInputInfoAPI

表 8-17 入参 PngInputInfoAPI

成员变量	说明
void* inputData	输入图像数据的地址。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数inputSize的参数值一致。
uint64_t inputSize	输入内存长度，用于校验输入数据。
void* address	输入图像数据的地址。当前版本不使用该参数。

版权所有 © 华为技术有限公司

成员变量	说明
uint64_t size	输入内存长度。当前版本不使用该参数。
int32_t transformFlag	转换标志，1表示RGBA转换到RGB，0保留原格式。

出参：PngOutputInfoAPI

表 8-18 出参 PngOutputInfoAPI

成员变量	说明
void* outputData	输出图像数据的地址。当前版本不使用该参数。
uint64_t outputSize	输出图像数据的地址。当前版本不使用该参数。
void* address	输出内存地址。 如果由用户指定内存，使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址128对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考 》。 **须知** 使用HIAI_DVPP_DMalloc接口申请内存时，则由用户保证申请的内存大小与输入参数outputSize的参数值一致。
uint64_t size	内存大小。
int32_t format	输出图像格式： ● 2表示RGB输出。 ● 6表示RGBA输出。
uint32_t width	输出图像宽度。
uint32_t high	输出图像高度。
uint32_t widthAlign	宽度内存对齐，图片一行数据占用的内存大小进行128位对齐。若输出格式为RGB，则width*3后128位对齐，若输出格式为RGBA，则width*4后128位对齐。
uint32_t highAlign	高度对齐，目前为16位对齐。

8.3.2.2.6 查询 DVPP 引擎参数说明

功能

主要用于查询DVPP引擎的能力，包括各个模块的能力，各个模块的分辨率限制及性能参数等。

入参：device_query_req_stru

表 8-19 入参 device_query_req_stru

成员变量	说明	取值范围
uint32_t module_id	模块的ID。	固定为1
uint32_t engine_type	DVPP引擎类型。	VDEC：0 JPEGD：1 PNGD：2 JPEGE：3 VPC：4 VENC：5

出参：dvpp_engine_capability_stru

表 8-20 出参 dvpp_engine_capability_stru

成员变量	说明	取值范围
int32_t engine_type	DVPP引擎类型。	VDEC：0 JPEGD：1 PNGD：2 JPEGE：3 VPC：4 VENC：5
struct dvpp_resolution_stru max_resolution	最大分辨率。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。
struct dvpp_resolution_stru min_resolution;	最小分辨率。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
uint32_t protocol_num;	引擎所支持的标准协议类型数量。	VDEC：5 JPEGD：1 PNGD：1 JPEGE：1 VPC：0 VENC：4

uint32_t protocol_type[DVPP_PROTOCOL_TYPE_MAX];	引擎所支持的标准协议类型表格。	*enum* dvpp_proto_type {
		dvpp_proto_unsupport =-1,
		dvpp_itu_t81,
		iso_iec_15948_2003,
		h265_main_profile_level_5_1_hightier,
		h265_main_10_profile_level_5_1_hightier,
		h264_main_profile_level_5_1,
		h264_baseline_profile_level_5_1,
		h264_high_profile_level_5_1,
		h264_high_profile_level_4_1,
		h264_main_profile_level_4_1,
		h264_baseline_profile_level_4_1,
		h265_main_profile_level_4_1
		};

uint32_t input_format_num;	支持的输入格式数量。	VDEC：2 JPEGD：1 PNGD：1 JPEGE：6 VPC：51 VENC：2
struct dvpp_format_unit_stru engine_input_format_table[DVPP_VADIO_FORMAT_MAX];	引擎所支持的输入格式表。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
uint32_t output_format_num;	支持的输出格式数量。	VDEC：4 JPEGD：4 PNGD：2 JPEGE：1 VPC：2 VENC：2
struct dvpp_format_unit_stru engine_output_format_table[DVPP_VADIO_FORMAT_MAX];	引擎所支持的输出格式表结构体。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。
uint32_t performance_mode_num;	性能模式的数量。	固定为1。
struct dvpp_perfomance_unit_stru performance_mode_table[DVPP_PERFOMANCE_MODE_MAX];	模块的性能结构体。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。
struct dvpp_pre_contraction_stru pre_contraction;	预缩小信息结构体。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。
struct dvpp_pos_scale_stru pos_scale;	后缩放信息结构体。	详细见**8.3.7.4 dvpp_engine_capability_stru中的结构体**。
uint32_t spec_input_num	输入格式所属类型的数目，目前支持两种类型： ● HFBC ● YUV或RGB	-
struct dvpp_vpc_data_spec_stru spec_input[DVPP_DATA_INPUT_SPEC_TYPE_MAX]	输入格式所属类型的描述。	详细见**•dvpp_vpc_data_spec_stru...**。

8.3.2.3 DestroyDvppApi

函数原型	**int DestroyDvppApi(IDVPPAPI*& pIDVPPAPI)**
功能	销毁由**CreateDvppApi**接口创建的dvppapi实例，关闭DVPP执行器。

版权所有 © 华为技术有限公司

输入说明	输入为“**IDVPPAPI**”类型指针引用。
输出说明	无。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	无。
使用约束	一旦调用**DestroyDvppApi**接口，如果还想在继续调用DVPP，需要调用**CreateDvppApi接口**重新创建dvppapi实例。

调用示例

```
DestroyDvppApi(pidvppapi);
```

8.3.2.4 DvppGetOutParameter

函数原型	**int32_t DvppGetOutParameter(void* in, void* out, int32_t cmd)**
功能	获取JPEGD/JPEGE/PNGD模块的输出内存大小。
输入说明	● in指针，各模块对应的指针不同，请参见**表8-21**。 ● out指针，各模块对应的指针不同，请参见**表8-21**。 ● **cmd控制命令字**
输出说明	各模块输出内存的大小： ● JPEGE功能，取**sJpegeOut**结构体中的jpgSize参数值。 ● JPEGD功能，取**JpegdOut**结构体中的yuvDataSize参数值。 ● PNGD功能，取**PngOutputInfoAPI**结构体中的outputSize参数值。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	调用方调用“**DvppGetOutParameter**”函数，传入in和out指针，并传入正确的**cmd控制命令字**。 调用示例如下： ● JPEGD模块，请参见**8.3.6.3 实现JPEGD功能**。 ● JPEGE模块，请参见**8.3.6.2 实现JPEGE功能**。 ● PNGD模块，请参见**8.3.6.4 实现PNGD功能**。
使用限制	无。

表 8-21 入参说明

入参	说明	取值范围
void* in	输入结构体指针	• JPEGE功能入参：**sJpegeIn** • JPEGD功能入参：**JpegdIn** • PNGD功能入参：**PngInputInfoAPI** **须知** 该函数不支持JPEGD 的**jpegd_raw_data_info**结构体。
void* out	输出结构体指针	• JPEGE功能出参：**sJpegeOut** • JPEGD功能出参：**JpegdOut** • PNGD功能出参：**PngOutputInfoAPI** **须知** 该函数不支持JPEGD 的**jpegd_yuv_data_info**结构体。
int32_t cmd	获取输出参数控制命令字	• JPEGE功能的命令字：GET_JPEGE_OUT_PARAMETER • JPEGD功能的命令字：GET_JPEGD_OUT_PARAMETER • PNGD功能的命令字：GET_PNGD_OUT_PARAMETER

8.3.3 VDEC 功能接口

8.3.3.1 总体说明

对于一个视频码流，调用一次**CreateVdecApi**接口创建实例后，必须使用同一个实例调用**VdecCtl**接口进行视频解码，最后再调用一次**DestroyVdecApi**接口释放实例。

在视频解码时，如果需要从一个视频码流切换到另一个视频码流，则需要先调用**DestroyVdecApi**接口释放前一个码流的实例，再调用**CreateVdecApi**接口创建新的实例，用于处理新的视频码流。

8.3.3.2 CreateVdecApi

函数原型	**int CreateVdecApi(IDVPPAPI*& pIDVPPAPI, int singleton)**
功能	获取vdecapi实例，相当于vdec执行器句柄。调用方可以使用申请到的vdecapi实例调用**CreateVdecApi接口**进行视频解码，可以跨函数调用，跨线程调用。
输入说明	• **IDVPPAPI**类型指针引用，输入指针必须为NULL。 • singleton为内部保留使用，为以后实现pIDVPPAPI单例预留，建议调用方当前设置为0。

版权所有 © 华为技术有限公司

输出说明	输出为**IDVPPAPI**类型指针引用，输出可能为NULL，可能不为NULL，获取失败则为NULL，成功则不为NULL。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	调用方创建**IDVPPAPI**对象指针，初始化为NULL，调用**CreateVdecApi**函数将**IDVPPAPI**对象指针传入。如果申请成功，**CreateVdecApi**接口会返回DvppApi实例，否则返回NULL。调用方需要对返回值进行校验。
使用约束	调用方负责vdecapi实例的生命周期，即申请与释放，申请使用**CreateVdecApi**接口，释放使用**DestroyVdecApi接口**。

8.3.3.3 VdecCtl

函数原型	**int VdecCtl(IDVPPAPI*& pIDVPPAPI, int CMD, dvppapi_ctl_msg* MSG, int singleton)**
功能	使用**CreateVdecApi**接口创建的实例调用**VdecCtl**接口，控制DVPP执行器进行视频解码。
输入说明	● **IDVPPAPI**类型指针引用。 ● 控制命令字CMD(VDEC为DVPP_CTL_VDEC_PROC)。 ● **dvppapi_ctl_msg**类型的vdec执行器配置信息MSG。其中此结构体中的in请见**入参：vdec_in_msg**。 ● 输入singleton为内部保留使用，为以后实现pIDVPPAPI单例预留，建议调用方当前设置为0。
输出说明	输出为MSG的配置信息中的输出buffer，以及VDEC的输出状态信息(都存储于MSG中)。
返回值说明	● 返回值“0”代表接口调用成功，并不代表解码成功（由于VDEC解码是异步方式）。 ● 返回值“-1”代表接口调用失败。
使用说明	调用方调用**VdecCtl**接口，传入**IDVPPAPI**对象指针，配置好相应功能的**dvppapi_ctl_msg**，并传入正确的控制命令字CMD。VDEC解码为异步方式，调用**VdecCtl**接口会将用户码流buffer中的数据拷贝到VDEC内部输入buffer后即返回，故调用**VdecCtl**接口后并不代表解码成功（仅表示数据拷贝成功），并且此接口返回后，用户码流buffer即可释放。
使用约束	对于一个视频码流，调用一次**CreateVdecApi接口**创建实例后，必须使用同一个实例调用**VdecCtl**接口进行视频解码，最后再调用一次**DestroyVdecApi接口**释放实例。

入参：vdec_in_msg

表 8-22 入参 vdec_in_msg

成员变量	说明
char video_format[10]	输入视频格式，“h264”或者“h265”，默认是“h264”， 仅支持yuv420sp(NV12、NV21)编码后的h264和h265码流。
char image_format[10]	输出帧格式，“nv12”或者“nv21”，默认是“nv12”。
void (*call_back)(FRAME* frame,void * hiai_data)	调用方回调函数，FRAME为vdec解码后的输出结构体，详见**8.3.7.2 vdec_in_msg中的结构体和类FRAME结构体**。用户可以根据该指针获取输出结果。 建议用户在回调函数内仅调用**DvppCtl**接口输出yuv格式的图片数据，其它功能不建议放在回调函数内实现，因为回调函数内实现的功能太多，可能会耗时过长，导致解码过程阻塞等待资源。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；帧率=25fps，则允许最大耗时=1/（25fps）=0.04s。
char* in_buffer	输入视频码流内存，此码流为h264或h265裸码流。 用户将存放解码前码流的内存（由用户自行申请）赋值给in_buffer，调用**VdecCtl**接口有返回后，就可以释放存放解码前码流的buffer。
int32_t in_buffer_size	输入视频码流内存大小。
void * hiai_data	解码后输出结果帧回调函数的形参指针，指针指向对象由调用方定义具体的结构。 **须知** VDEC内部仅调用第一次传给VDEC的hiai_data，若要想多次使用hiai_data传送不同对象，请用下面智能指针对象。

版权所有 © 华为技术有限公司

成员变量	说明
std::shared_ptr<HIAI_DATA_SP> hiai_data_sp	如果不涉及帧序号的配置，可不设置该参数，默认为NULL。如果需要设置帧序号，使用方法如下。 调用方回调函数形参指针，其中HIAI_DATA_SP为VDEC内部定义的父类，具体类HIAI_DATA_SP结构请见**8.3.7.2 vdec_in_msg中的结构体和类HIAI_DATA_SP类**，用户可以继承衍生子类，使用方式可以参考**8.3.6.5 实现VDEC功能**。此指针指向的类对象必须设置帧序号信息，只支持一帧（必须包含I帧或P帧或B帧）对应一个帧序号，不支持多帧对应同一个帧序号，并且帧序号需以等差数列方式设置，从1开始，间隔大小为1。 **使用注意点：** 1. 在用户自定义子类hiai_data_sp对象中，不能包含需申请过大内存空间的成员变量。因为针对每路视频码流的解码，VDEC内部最多支持100个hiai_data_sp对象，如果成员变量申请的内存空间过大，可能会导致内存消耗过大； 2. 用户自定义子类hiai_data_sp对象中若有需要申请内存空间的成员变量，在申请内存并使用完成后，需在析构函数中释放内存，避免内存泄露； 3. 使用VDEC进行视频码流解码时，如果使用了hiai_data_sp对象，但视频码流中存在异常帧，VDEC会直接将异常帧的hiai_data_sp对象丢弃，因此建议用户在析构函数的实现代码中对这个异常情况做处理。 4. 使用VDEC进行视频码流解码时，如果使用了hiai_data_sp对象，仅支持码流参考帧间距不超出30帧，例如解码第30帧时可以参考第1帧，但解码第31帧时不能再参考第1帧。 5. VDEC会将用户传入的hiai_data_sp对象缓存至内部队列，队列最大长度为100。若某帧解码失败则对应的hiai_data_sp对象最快会在之后30帧解码全部返回的时刻被丢弃，例如第1帧解码失败，第1帧对应的hiai_data_sp对象会在第31帧解码结束后被丢弃；若码流中全是异常帧，则第1帧对应的hiai_data_sp对象会在第101帧开始解码时被丢弃。 6. hiai_data_sp和hiai_data只能选其一，如果使用hiai_data_sp，则必须与channelId参数配合使用。 7. 若码流中含有B帧，帧序号只支持按显示顺序输出，不支持按解码顺序输出。
int32_t channelId	输入码流对应的解码通道的ID，不同码流同时解码需设置不同的值，取值范围0~15。

版权所有 © 华为技术有限公司

成员变量	说明
void (*err_report)(VDECERR* vdecErr)	错误上报回调函数，用于将解码过程中出现的异常通知用户，比如码流错误、硬件问题、解码器状态错误等，以便用户决定如何处理。其中形参VDECERR为VDEC内部定义结构体，具体请见**8.3.7.2 vdec_in_msg中的结构体和类 VDECERR结构体**。
bool isEOS	码流结束标志，标识本路解码结束。用户可以不用关心此标志，默认在**DestroyVdecApi**接口中会将isEOS设置true，并在内部实现结束本路解码同时释放资源。若用户主动配置isEOS为true，则在调用VdecCtl接口中优先使用用户配置，结束本路解码并释放资源。具体使用请见**8.3.6.5 实现VDEC功能**。
int32_t isOneInOneOutMode	预留参数，请保持默认值0。

8.3.3.4 DestroyVdecApi

函数原型	**int DestroyVdecApi(IDVPPAPI*& pIDVPPAPI, int singleton)**
功能	释放由**CreateVdecApi**接口创建的vdecapi实例，关闭VDEC执行器。
输入说明	输入为**IDVPPAPI**类型指针引用。 输入singleton为内部保留使用，为以后实现pIDVPPAPI单例预留，建议调用方当前设置为0。
输出说明	无输出。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	无特殊说明。
使用约束	一旦调用**DestroyVdecApi**接口，如果还想在继续调用VDEC，需要重新创建vdecapi实例。

8.3.4 VENC 功能接口

8.3.4.1 总体说明

若需要将多张图片编码成一个视频，则调用一次**CreateVenc**接口创建实例后，必须使用同一个实例调用**RunVenc**接口进行视频编码，最后再调用一次**DestroyVenc**接口释放实例。

版权所有 © 华为技术有限公司

8.3.4.2 CreateVenc

函数原型	**int32_t CreateVenc(struct VencConfig* vencConfig)**
功能	获取VENC编码实例，相当于获取VENC执行器句柄。调用方可以使用申请到的VENC编码实例调用**RunVenc接口**进行图片编码。
输入说明	输入为结构体VencConfig类型指针，VencConfig参见说明入参：**入参：VencConfig**。其中需要配置回调函数，用于处理编码结果。
输出说明	无输出。
返回值说明	● 返回值非负数代表创建VENC编码实例成功 ● 返回值“-1”代表创建VENC编码实例失败。
使用说明	调用方创建VencConfig对象指针，调用**CreateVenc**接口将VencConfig对象指针传入。如果申请成功，**CreateVenc**接口会返回VENC编码实例句柄号，否则返回-1。调用方需要对返回值进行校验。
使用约束	调用方负责VENC编码实例的生命周期，即申请与释放，申请使用CreateVenc接口，释放使用**DestroyVenc接口**。

入参：VencConfig

该入参是初始化VENC模块时使用，结构体所有成员变量必须初始化后再使用。

表 8-23 入参 VencConfig

成员变量	说明	取值范围
uint32_t width	图像宽度。	128~1920，且为偶数。
uint32_t height	图像高度。	128~1920，且为偶数。
uint32_t codingType	视频编码协议H265-main level（0）、H264-baseline level（1）、H264-main level(2)、H264-high level（3）	0~3 ● 0：H265 main level（仅支持Slice码流）。 ● 1：H264 baseline level。 ● 2：H264 main level。 ● 3：H264 high level。
uint32_t yuvStoreType	YUV图像存储格式。	0或者1 ● 0：YUV420 semi-planner（nv12） ● 1：YVU420 semi-planner（nv21）
uint32_t keyFrameInterval	I帧间隔	大于0小于65535

成员变量	说明	取值范围
VencOutMsgCallBack vencOutMsgCallBack	回调函数，用于处理编码结果。 DVPP（VENC模块）处理每一路图片数据的编码时，对于首帧图片数据，会调用两次回调函数，第一次调用回调函数处理文件头信息，第二次调用回调函数处理本帧的图片数据	typedef void (*VencOutMsgCallBack) (struct VencOutMsg* vencOutMsg, void* userData);不可为空
void* userData	用户记录想要传递的信息，随回调函数返回给用户	可以为NULL

8.3.4.3 RunVenc

函数原型	**int32_t RunVenc(int32_t vencHandle, struct VencInMsg* vencInMsg)**
功能	使用**CreateVenc**接口创建的实例调用**RunVenc**接口，控制DVPP执行器进行视频编码。
输入说明	输入为int32_t句柄号和VencInMsg指针。vencHandle为CreateVenc返回值。VencInMsg参见入参：**入参：VencInMsg**。VENC执行器配置信息VencInMsg，该结构体用于将要编码的视频信息传递给执行器进行编码。
输出说明	无输出。
返回值说明	● 返回值“0”代表成功。 ● 返回值“-1”代表失败。
使用说明	调用方调用RunVenc函数，传入编码实例句柄号和VencInMsg对象指针，配置好相应功能的vencInMsg。
使用约束	若需要将多张图片编码成一个视频，则调用一次**CreateVenc接口**创建实例后，必须使用同一个实例调用**RunVenc**接口进行视频编码，最后再调用一次**DestroyVenc**接口释放实例。

入参：VencInMsg

该入参是调用VENC模块执行编码时使用，所有结构体成员变量必须初始化后再使用。

表 8-24 入参 VencInMsg

成员变量	说明	取值范围
void* inputData	输入数据地址	非空
uint32_t inputDataSize	输入数据大小	不能大于输入数据buffer大小，推荐值和输入数据buffer大小一样
uint32_t keyFrameInterval	I帧间隔	大于等于0小于65535，0表示这个参数不起作用
uint32_t forceIFrame	强制重新开始I帧间隔，0：不强制；1：强制重新开始I帧	0或者1
uint32_t eos	是否为结束帧，0：不是；1：是结束帧	0或者1

出参：VencOutMsg

表 8-25 出参 VencOutMsg

成员变量	说明	取值范围
void* outputData	输出数据地址	由VENC内部申请
uint32_t outputDataSize	输出数据大小	由VENC内部填写
uint32_t timeStamp	记录调用回调函数的时序	由VENC内部填写

8.3.4.4 DestroyVenc

函数原型	**int32_t DestroyVenc(int32_t vencHandle)**
功能	释放由**CreateVenc**接口创建的VENC编码实例，关闭VENC执行器。
输入说明	输入为int32_t类型VENC编码实例句柄号，为调用函数CreateVenc返回值。
输出说明	无输出。
返回值说明	• 返回值“0”代表成功。 • 返回值“-1”代表失败。
使用说明	无特殊说明。
使用约束	一旦调用**DestroyVenc**接口，如果还想在继续调用VENC，需要重新创建VENC编码实例。

8.3.5 兼容旧版本的功能

8.3.5.1 兼容性说明

为了兼容旧版本中的功能，当前版本支持以下方式实现VPC功能和VENC功能：

- 实现VPC功能：
  - 在**DvppCtl**接口中传入入参：DVPP_CTL_VPC_PROC命令字和**resize_param_in_msg结构体**的参数。
  - 对时延要求不高、处理的图片分辨率低、数量多的场景，可以在**DvppCtl**接口中传入入参：DVPP_CTL_CMDLIST_PROC和**IMAGE_CONFIG结构体**的参数。

  这种方式是为了兼容旧版本中的功能，后续版本会删除，不推荐使用。如果使用了不推荐使用的功能，建议迁移到推荐的VPC功能，请参见**8.3.2 VPC/JPEGE/JPEGD/PNGD功能接口**。
- 实现VENC功能：调用**CreateDvppApi**、**DvppCtl**、**DestroyDvppApi**接口实现VENC功能，在**DvppCtl**接口中传入入参：DVPP_CTL_VENC_PROC命令字和**venc_in_msg结构体**的参数。

  这种方式在为了兼容旧版本中的功能，后续版本会删除，不推荐使用。如果使用了不推荐使用的功能，建议迁移到推荐的VENC功能，请参见**8.3.4 VENC功能接口**。

8.3.5.2 VPC 功能及参数说明

不推荐使用旧版本中的VPC功能，后续版本会删除。旧版本中的VPC功能和CMDLIST功能均迁移到VPC功能，请参见8.3.2.2.2 VPC参数说明处的参数说明进行开发。

旧版本中的VPC功能：主要用于处理媒体图像转换，包括缩放、色域转换、降bit数处理、格式转换、区块切割，叠加拼接，CMDLIST。

- VPC路数无限制，其性能指标如下表。

  VPC性能由于涉及到抠图、缩放等不同的场景，当处理图片过程中的分辨率改变时，性能会对应改变。当处理过程中图像分辨率都没有比原图分辨率大时，典型场景性能指标如下：

场景	总帧率
1080p*32路	1440fps
4k*4路	360fps

  当处理过程中图像分辨率大于原图分辨率时，总帧率计算公式参考如下：

  总帧率=(3840*2160*360)/(W*H) fps

  其中：W和H为VPC处理过程中最大的宽度和高度，例如1080p抠图到960*540，再缩放到3840*2160，此时W为3840，H为2160。

  在处理过程中，缩小比例的增大会导致VPC处理性能有略微下降。
- VPC输入输出图像宽高限制：
  - 输入图像宽高都需要2对齐（偶数）。

– 输出图像宽高都需要2对齐（偶数）。

须知

此处所说的图像宽高是指图片的有效区域，实际上对于送进VPC的内存需要128*16对齐。

- VPC输入输出图像内存限制：
  – VPC输入内存限制
    - 输入数据地址（os虚拟地址）：128字节对齐。
    - 输入图像宽度内存限制：128字节对齐。
    - 输入图像高度内存限制：16字节对齐。

说明

对于yuv400sp格式图片，需要调用方申请和yuv420sp格式相同大小的空间，即：width_align_128*high_align_16*1.5。

  – VPC输出内存限制
    - 输出数据地址（os虚拟地址）：128字节对齐。
    - 输出图像宽内存限制：128字节对齐。
    - 输出图像高内存限制：16字节对齐。

说明

在使用VPC实现图像裁剪与缩放时，共需要调用两次DvppCtl，第一次调用的输入参数为resize_param_in_msg，输出参数为resize_param_out_msg，第二次调用的输入参数为vpc_in_msg，详细信息请参见**入参：resize_param_in_msg**~**入参：vpc_in_msg**。

入参：resize_param_in_msg

成员变量	说明	取值范围
int src_width	原图的宽度（2对齐）。 **须知** 原图指图片的有效区域，实际上对于送进VPC的内存需要128*16对齐。	最小分辨率： • 128（来源VDEC） • 16（来源非VDEC） 最大分辨率： 4096 必填

成员变量	说明	取值范围
int src_high	原图的高度（2对齐）。 **须知** 原图指图片的有效区域，实际上对于送进VPC的内存需要128*16对齐。	最小分辨率： ● 128（来源VDEC） ● 16（来源非VDEC） 最大分辨率： 4096 必填
int hmax	与原点在水平方向的最大偏移。	奇数，具体参见**入参：vpc_in_msg**。
int hmin	与原点在水平方向的最小偏移。	偶数，具体参见**入参：vpc_in_msg**。
int vmax	与原点在垂直方向的最大偏移。	奇数，具体参见**入参：vpc_in_msg**。
int vmin	与原点在垂直方向的最小偏移。	偶数，具体参见**入参：vpc_in_msg**。
int dest_width	输出图像宽度。	偶数。
int dest_high	输出图像高度。	偶数。

出参：resize_param_out_msg

成员变量	说明	取值范围
int dest_width_stride	输出图像的宽度对齐值。	128
int dest_high_stride	输出图像的高度对齐值。	16
int hmax	与原点在水平方向的最大偏移。	奇数，具体参见**入参：vpc_in_msg**。
int hmin	与原点在水平方向的最小偏移。	偶数，具体参见**入参：vpc_in_msg**。
int vmax	与原点在垂直方向的最大偏移。	奇数，具体参见**入参：vpc_in_msg**。
int vmin	与原点在垂直方向的最小偏移。	偶数，具体参见**入参：vpc_in_msg**。
double hinc	水平缩放系数。	具体参见**入参：vpc_in_msg**。
double vinc	垂直缩放系数。	具体参见**入参：vpc_in_msg**。

入参：vpc_in_msg

成员变量	说明	取值范围
int format	图像格式，调用方可以自行定义，只要后面的数字对应正确即可。例如：yuv420_semi_plannar对应的是0，调用方声明为yuv420sp，那么也必须对应为0。	yuv420_semi_plannar = 0,//0 yuv400_semi_plannar = 0,//0 yuv422_semi_plannar,//1 yuv444_semi_plannar,//2 yuv422_packed,//3 yuv444_packed,//4 rgb888_packed,//5 xrgb8888_packed,//6 必填
int rank	图像排列方式，调用方可以自行定义，只要后面的数字对应正确即可。	具体参见**表8-26** 必填
int bitwidth	位深，默认是8bit。10bit只有在图像格式为YUV/YVU420 Semi-Planar且为HFBC压缩场景下使用。	8或者10 选填
int cvdr_or_rdma	输入图像走CVDR通道还是RDMA，默认走CVDR通道，RDMA通道的输入仅为VDEC输出的HFBC格式数据。	CVDR：1，RDMA：0。 选填
int width	输入原图像宽度，2对齐（偶数）。 **须知** 原图指图片的有效区域，实际上对于送进VPC的内存需要128*16对齐。	最小分辨率： ● 128（来源VDEC）。 ● 16（来源非VDEC）。 最大分辨率： 4096 必填
int high	输入原图像高度，2对齐（偶数）。 **须知** 原图指图片的有效区域，实际上对于送进VPC的内存需要128*16对齐。	最小分辨率： ● 128（来源VDEC）。 ● 16（来源非VDEC）。 最大分辨率： 4096 必填

成员变量	说明	取值范围
int stride	图像步长。 **须知** 对于不同输入格式，其stride不同。	yuv400sp、yuv420sp、yuv422sp、yuv444sp：width对齐到128。 yuv422packed：width * 2后对齐到128 yuv444packed、rgb888：width * 3后对齐到128 xrgb8888：width * 4后对齐到128 必填
double hinc	水平放大倍数，aicore输出通道配置。	[0.03125, 1) or (1, 4] 缩放系数配置，超出配置范围，VPC会提示出错，如果不要缩放填1。 必填
double vinc	垂直放大倍数，aicore输出通道配置。	[0.03125, 1) or (1, 4] 缩放系数配置，超出配置范围，VPC会提示出错，如果不要缩放填1。 必填
double jpeg_hinc	水平放大倍数，jpege输出通道配置。	[0.03125, 1) or (1, 4] 缩放系数配置，超出配置范围，VPC会提示出错。 选填(目前不适用)
double jpeg_vinc	垂直放大倍数，jpege输出通道配置。	[0.03125, 1) or (1, 4] 缩放系数配置，超出配置范围，VPC会提示出错。 选填(目前不适用)
int hmax	与原点在水平方向的最大偏移。	此值必须为奇数,如果不用裁剪,此值配置成输入宽度-1。 必填
int hmin	与原点在水平方向的最小偏移。	此值必须为偶数。 如果不用裁剪，此值配置成0。 必填
int vmax	与原点在垂直方向的最大偏移。	此值必须为奇数。 如果不用裁剪，此值配置成输入宽度-1。 必填
int vmin	与原点在垂直方向的最小偏移。	此值必须为偶数。 如果不用裁剪，此值配置成0。

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
int out_width	输出图像宽度。	此值必须为偶数(2对齐)。 当使用智能指针申请输出buffer时，不需要填，当使用自己申请的内存作为输出buffer时，必填。
int out_high	输出图像高度。	此值必须为偶数(2对齐)。 当使用智能指针申请输出buffer时，不需要填，当使用自己申请的内存作为输出buffer时，必填。
int h_stride	图像高度步长，同int: stride描述。此值无需配置。	此值默认为16，即默认输入高度为16对齐。 选填
char* in_buffer	输入buffer。	调用方需保证输入buffer的大小与in message中配置的长度与宽度一致，即：如果输入是yuv420 semi planner nv12格式的图片，其图像大小为：宽*高*1.5，而输入的buffer大小与如上计算结果不符，则会导致VPC调用失败。 必填
int in_buffer_size	输入buffer大小。	此值用来进行输入buffer校验。 必填
shared_ptr<AutoBuffer> auto_out_buffer_1	输出buffer1。	使用VPC基础功能（裁剪，缩放）需要配置此buffer，此buffer为调用方申请智能指针当做参数传入DVPP，由DVPP内部申请输出buffer。用户可读取该buffer获取输出图片。 与下一个参数out_buffer选填一个即可。
char* out_buffer	输出buffer。	使用VPC基础功能（裁剪，缩放）需要配置此buffer，，此buffer为调用方申请的内存，需保证首地址128对齐。用户可读取该buffer获取输出图片。 与上一个参数auto_out_buffer_1选填一个即可。
int out_buffer_1_size	输出buffer1大小。	此值用来进行输出buffer校验。 如果采用shared_ptr<AutoBuffer> auto_out_buffer_1申请输出内存则选填，如果采用 char* out_buffer申请输出内存则必填。

成员变量	说明	取值范围
shared_ptr<AutoBuffer> auto_out_buffer_2	此智能指针为vpc双通道的另外一个通道输出，为预留接口	--
int out_buffer_2_size	此buffersize为auto_out_buffer_2智能指针输出buffer2大小，为预留接口	--
int use_flag	使用vpc功能的标志。	● 0：vpc基础功能。 ● 1：Rawdata8k缩放。 ● 2：Rawdata叠加基础功能，预留功能。 ● 3：Rawdata拼接基础功能，预留功能。 默认为0。 选填
VpcOverLayReverse overlay	叠加结构体，预留。	预留
VpcCollageReverse collage	拼接结构体，预留。	预留
RDMACHANNEL rdma	rdma参数为HFBC数据专用配置结构体。	具体请参见**8.3.7.5.1 Rdma通道结构体RDMACHANNEL**。
VpcTurningReverse turningReverse1	Vpc优化预留接口。	具体请参见**8.3.7.5.2 Vpc内部优化结构体VpcTurningReverse**。
bool isVpcUseTurning1	标志是否使用Vpc优化，预留接口，此标志与turningReverse1配合使用。	--
VpcTurningReverse turningReverse2	Vpc优化预留接口。	具体请参见**8.3.7.5.2 Vpc内部优化结构体VpcTurningReverse**。
bool isVpcUseTurning2	标志是否使用Vpc优化，预留接口，此标志与turningReverse2配合使用。	--

版权所有 © 华为技术有限公司

成员变量	说明	取值范围
string *yuvscaler_paraset	yuvscaler_paraset指针接收VPC滤波参数集文件的路径和文件名数组。 **说明** ● 参数集须在Device设备上。 ● 请确保传入的文件路径是正确路径。	参数集文件路径为device侧的绝对文件路径＋文件名，如{"/home/HwHiAiUser/matrix/YUVScaler_pra.h"}。
unsigned int yuvscaler_paraset_size	yuvscaler_paraset指针指向的string数组元素个数（默认为1）。	yuvscaler_paraset_size<=10
unsigned int index	index变量对应yuvscaler_paraset的数组索引。	0<=index<10

表 8-26 输入图片格式与输出图片格式对应 rank 配置表

输入图片格式	VPC的rank参数配置	输出图片格式
YUV420sp NV12	NV12 = 0	YUV420sp NV21
YUV420sp NV21	NV21 = 1	YUV420sp NV21
YUV420sp NV12	NV12 = 1	YUV420sp NV12
YUV420sp NV21	NV21 = 0	YUV420sp NV12
YUV422sp NV16	NV16 = 1	YUV420sp NV12
YUV422sp NV61	NV61 = 0	YUV420sp NV12
YUV422sp NV16	NV16 = 0	YUV420sp NV21
YUV422sp NV61	NV61 = 1	YUV420sp NV21
YUV444sp 444spUV	444spUV = 1	YUV420sp NV12
YUV444sp 444spVU	444spVU= 0	YUV420sp NV12
YUV444sp 444spUV	444spUV = 0	YUV420sp NV21
YUV444sp 444spVU	444spVU= 1	YUV420sp NV21
YUV422packet YUYV	YUYV = 2	YUV420sp NV12
YUV422packet YVYU	YVYU = 3	YUV420sp NV21
YUV422packet UYVY	UYVY = 4	YUV420sp NV12
YUV444packet YUV	YUV = 5	YUV420sp NV12
RGB888 RGB	RGB = 6	YUV420sp NV12

输入图片格式	VPC的rank参数配置	输出图片格式
RGB888 BGR	BGR = 7	YUV420sp NV21
RGB888 RGB	BGR = 7	YUV420sp NV21
RGB888 BGR	RGB = 6	YUV420sp NV12
XRGB8888 RGBA	RGBA = 8	YUV420sp NV12
XRGB8888 RGBA	BGRA = 9	YUV420sp NV21

调用示例

- VPC基础功能调用示例

  gXxxxx为配置参数，参数配置与VPC调用分为两个阶段。

  Step1：通过DVPP_CTL_TOOL_CASE_GET_RESIZE_PARAM命令字调用DvppCtl，获取配置参数，DvppCtl的入参为resize_paramo_in_msg，出参为resize_param_out_msg。即通过resize_param_in_msg将原始图片的宽高，裁剪区域(hmax,hmin,vamx,vmin)，以及输出宽高等参数传给DVPP，DVPP通过计算将重新生成的裁剪区域(hmax,hmin,vamx,vmin)以及缩放比(hinc,vinc)返回给调用方。

  Step2：调用方将第一步返回的参数通过vpc_in_msg结构体传递给DVPP，并将DvppCtl调用命令字改为DVPP_CTL_VPC_PROC，执行并获得处理结果。

```
dvppapi_ctl_msg dvppApiCtlMsg;
vpc_in_msg vpcInMsg;
resize_param_in_msg resize_in_param;
resize_param_out_msg resize_out_param;
resize_in_param.src_width = gWidth;
resize_in_param.src_high = gHigh;
resize_in_param.hmax = gHmax;
resize_in_param.hmin = gHmin;
resize_in_param.vmax = gVmax;
resize_in_param.vmin = gVmin;
resize_in_param.dest_width = floor(gHinc * (gHmax - gHmin + 1) + 0.5);
resize_in_param.dest_high = floor(gVinc * (gVmax - gVmin + 1) + 0.5);
printf("width  =%d\n", gWidth);
printf("high   =%d\n", gHigh);
printf("hmax   =%d\n", gHmax);
printf("hmin   =%d\n", gHmin);
printf("vmax   =%d\n", gVmax);
printf("vmin   =%d\n", gVmin);
printf("out width   =%d\n", resize_in_param.dest_width);
printf("out hight   =%d\n", resize_in_param.dest_high );

dvppApiCtlMsg.in = (void *)(&resize_in_param);
dvppApiCtlMsg.out = (void *)(&resize_out_param);
IDVPPAPI *pidvppapi = NULL;
CreateDvppApi(pidvppapi);
if (DvppCtl(pidvppapi, DVPP_CTL_TOOL_CASE_GET_RESIZE_PARAM, &dvppApiCtlMsg) != 0) {
    printf("call DVPP_CTL_TOOL_CASE_GET_RESIZE_PARAM process faild!\n");
    DestroyDvppApi(pidvppapi);
    return;
}

int bufferSize        = 0;
vpcInMsg.format       = gFormat;
vpcInMsg.cvdr_or_rdma = gcvdr_or_rdma;
vpcInMsg.bitwidth = gBitwidth;
vpcInMsg.rank = gRank;
```

```
vpcInMsg.width = gWidth;
vpcInMsg.high = gHigh;
vpcInMsg.stride = gStride;
vpcInMsg.hmax = resize_out_param.hmax;
vpcInMsg.hmin = resize_out_param.hmin;
vpcInMsg.vmax = resize_out_param.vmax;
vpcInMsg.vmin = resize_out_param.vmin;
vpcInMsg.vinc = resize_out_param.vinc;
vpcInMsg.hinc = resize_out_param.hinc;
printf("resize_out_param.hinc = %lf\n", resize_out_param.hinc);
printf("resize_out_param.vinc = %lf\n", resize_out_param.vinc);
printf("resize_out_param.hmax = %d\n", resize_out_param.hmax);
printf("resize_out_param.hmin = %d\n", resize_out_param.hmin);
printf("resize_out_param.vmax = %d\n", resize_out_param.vmax);
printf("resize_out_param.vmin = %d\n", resize_out_param.vmin);

printf("format =%d\n", vpcInMsg.format);
printf("rank   =%d\n", vpcInMsg.rank);
printf("width  =%d\n", vpcInMsg.width);
printf("high   =%d\n", vpcInMsg.high);
printf("stride =%d\n", vpcInMsg.stride);
printf("hmax   =%d\n", vpcInMsg.hmax);
printf("hmin   =%d\n", vpcInMsg.hmin);
printf("vmax   =%d\n", vpcInMsg.vmax);
printf("vmin   =%d\n", vpcInMsg.vmin);
printf("vinc   =%F\n", vpcInMsg.vinc);
printf("hinc   =%F\n", vpcInMsg.hinc);
if (vpcInMsg.cvdr_or_rdma == 0) {
    printf("rdma channel\n");
    int buffersize ;
    char * payloadY ;
    char * payloadC ;
    char * headY ;
    char * headC ;
    FILE * rdmaimage = fopen(in_file_name, "rb");

    if (vpcInMsg.bitwidth == 10) {
        buffersize                    = vpcInMsg.width  * vpcInMsg.high * 2 + 512 * 1024;
        payloadY                       = (char*)memalign(128, buffersize);
        headY                         = payloadY + vpcInMsg.width  * vpcInMsg.high * 2;
        headC                         = headY + 256 * 1024;
        vpcInMsg.rdma.luma_payload_addr   = (long)payloadY;
        vpcInMsg.rdma.chroma_payload_addr = (long)payloadY + 640;
    } else {
        buffersize                    = vpcInMsg.width * vpcInMsg.high * 1.5 + 512 * 1024;
        payloadY                       = (char*)memalign(128, buffersize);
        payloadC                       = payloadY + vpcInMsg.width * vpcInMsg.high;
        headY                         = payloadC + vpcInMsg.width * vpcInMsg.high / 2;
        headC                         = headY + 256 * 1024;
        vpcInMsg.rdma.luma_payload_addr   = (long)payloadY;
        vpcInMsg.rdma.chroma_payload_addr = (long)payloadC;
    }

    fread(payloadY, 1, buffersize, rdmaimage);
    vpcInMsg.rdma.luma_head_addr        = (long)headY;
    vpcInMsg.rdma.chroma_head_addr      = (long)headC;
    vpcInMsg.rdma.luma_head_stride      = vpcInMsg.stride;
    vpcInMsg.rdma.chroma_head_stride    = vpcInMsg.stride;
    vpcInMsg.rdma.luma_payload_stride   = gpYStride;
    vpcInMsg.rdma.chroma_payload_stride = gpUVStride;
    fclose(rdmaimage);
    printf("luma payload stride= %d\n", vpcInMsg.rdma.luma_payload_stride );
    printf("chroma payload stride= %d\n", vpcInMsg.rdma.chroma_payload_stride );
} else {
    alloc_buffer(vpcInMsg.width, vpcInMsg.high, vpcInMsg.stride, bufferSize);
    if (open_image(in_file_name, vpcInMsg.width, vpcInMsg.high, vpcInMsg.stride) != 0) {
        printf("open file failed~\n");
        free(in_buffer);
        return;
```

```
    }
    vpcInMsg.in_buffer      = in_buffer;
    vpcInMsg.in_buffer_size = bufferSize;
}
//alloc in and out buffer
shared_ptr<AutoBuffer> auto_out_buffer = make_shared<AutoBuffer>();
vpcInMsg.auto_out_buffer_1              = auto_out_buffer;
dvppApiCtlMsg.in                        = (void *)(&vpcInMsg);
dvppApiCtlMsg.in_size                   = sizeof(vpc_in_msg);

if (pidvppapi != NULL) {

    for (int i = 0; i < gLoop; i++) {
        if (DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg) != 0) {
            printf("call dvppctl process faild!\n");
            DestroyDvppApi(pidvppapi);
            free(in_buffer);
            return;
        }
    }

} else {
    printf("pidvppapi is null!\n");
}

free(in_buffer);

FILE *fp = NULL;
fp = fopen(out_file_name, "wb+");
if (fp == NULL) {
    printf("open file %s failed~\n", out_file_name);
    free(in_buffer);
    return;
}

fwrite(vpcInMsg.auto_out_buffer_1->getBuffer(), 1, vpcInMsg.auto_out_buffer_1->getBufferSize(),
fp);
fflush(fp);
fclose(fp);
DestroyDvppApi(pidvppapi);
return;
```

- RawData8k基础功能调用示例

  参数配置：

```
dvppapi_ctl_msg dvppApiCtlMsg;
vpc_in_msg vpcInMsg;
vpcInMsg.width = gWidth;
vpcInMsg.high = gHigh;
vpcInMsg.stride = gStride;
vpcInMsg.out_width = gOutWidth;
vpcInMsg.out_high = gOutHigh;
printf("width     =%d\n", vpcInMsg.width);
printf("high      =%d\n", vpcInMsg.high);
printf("stride    =%d\n", vpcInMsg.stride);
printf("out_width =%d\n", vpcInMsg.out_width);
printf("out_high  =%d\n", vpcInMsg.out_high);
//alloc in and out buffer
char *in_buffer = (char *)malloc(vpcInMsg.width * vpcInMsg.high * 1.5);
if (in_buffer = NULL) {
    printf("malloc fail\n");
    return;
}

shared_ptr<AutoBuffer> auto_out_buffer = make_shared<AutoBuffer>();
FILE *yuvimage = fopen(in_file_name, "rb");
if (yuvimage == NULL) {
    printf("no such file!,file_name=[%s]\n", in_file_name);
    free(in_buffer);
    return;
```

```
    } else {
        fread(in_buffer, 1, vpcInMsg.width * vpcInMsg.high * 3 / 2, yuvimage);
    }

    vpcInMsg.rank = gRank;
    vpcInMsg.in_buffer = in_buffer;
    vpcInMsg.in_buffer_size = vpcInMsg.width * vpcInMsg.high * 1.5;
    vpcInMsg.auto_out_buffer_1 = auto_out_buffer;
    vpcInMsg.use_flag = 1;
    dvppApiCtlMsg.in = (void *)(&vpcInMsg);
    dvppApiCtlMsg.in_size = sizeof(vpc_in_msg);
    IDVPPAPI *pidvppapi = NULL;
    CreateDvppApi(pidvppapi);
    if (pidvppapi != NULL) {
        if (DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg) != 0) {
            printf("call dvppctl process faild!\n");
            DestroyDvppApi(pidvppapi);
            free(in_buffer);
            fclose(yuvimage);
            return;
        }
    } else {
        printf("pidvppapi is null!\n");
    }

    DestroyDvppApi(pidvppapi);
    FILE *fp = NULL;
    fp = fopen(out_file_name, "wb+");
    if (fp != NULL) {
        fwrite(vpcInMsg.auto_out_buffer_1->getBuffer(), 1, vpcInMsg.auto_out_buffer_1-
>getBufferSize(), fp);
        fflush(fp);
        fclose(fp);
    } else {
        fclose(yuvimage);
        return;
    }

    free(in_buffer);
    fclose(yuvimage);
    return;
```

- Rawdata叠加基础功能调用实例

```
    dvppapi_ctl_msg dvppApiCtlMsg;
    vpc_in_msg vpcInMsg;
    //配置RawData叠加参数
    vpcInMsg.use_flag = 2;
    vpcInMsg.overlay.image_type = 0;
    vpcInMsg.overlay.image_rank_type = 0;
    vpcInMsg.overlay.bit_width = 8;
    vpcInMsg.overlay.in_width_image = 1000;
    vpcInMsg.overlay.in_high_image = 1000;
    vpcInMsg.overlay.in_width_text = 500;
    vpcInMsg.overlay.in_high_text = 390;
    vpcInMsg.overlay.image_width_stride = 2;
    vpcInMsg.overlay.image_high_stride = 2;
    vpcInMsg.overlay.text_width_stride = 2;
    vpcInMsg.overlay.text_high_stride = 2;
    vpcInMsg.overlay.in_buffer_image = NULL;
    vpcInMsg.overlay.in_buffer_text = NULL;
    vpcInMsg.overlay.auto_overlay_out_buffer = make_shared<AutoBuffer>();
    //初始化图像内存
    if(NULL == (vpcInMsg.overlay.in_buffer_image =
(char*)malloc(vpcInMsg.overlay.in_width_image*vpcInMsg.overlay.in_high_image*3/2)))    {
        printf("in_buffer_image alloc failed!\n");
        return ;
    }
    //打开图片至申请的内存中
    char image_file[50] = "yuv_data_0";
    FILE * yuvimage = fopen(image_file,"rb");
```

```
    if(NULL == yuvimage)    {
        printf("VPC_TEST: yuv image open faild!");
        return ;
    }   else    {
        fread(vpcInMsg.overlay.in_buffer_image,
1,vpcInMsg.overlay.in_width_image*vpcInMsg.overlay.in_high_image*3/2,yuvimage);
        fclose(yuvimage);
        yuvimage = NULL;
}

    //初始化文字内存
    if(NULL == (vpcInMsg.overlay.in_buffer_text =
(char*)malloc(vpcInMsg.overlay.in_width_text*vpcInMsg.overlay.in_high_text*3/2)))    {
        printf("in_buffer_text alloc failed!");
        free(vpcInMsg.overlay.in_buffer_image);
        return ;
    }

    //打开text图片至申请的内存中
    char text_file[50] = "text_0";
    FILE * yuvtext = fopen(text_file,"rb");
    if(NULL == yuvtext)    {
        printf("VPC_TEST: yuv text image open faild!");
        free(vpcInMsg.overlay.in_buffer_image);
        return ;
    }   else    {
        fread(vpcInMsg.overlay.in_buffer_text,
1,vpcInMsg.overlay.in_width_text*vpcInMsg.overlay.in_high_text*3/2, yuvtext);
        fclose(yuvtext);
        yuvtext = NULL;
    }

    dvppApiCtlMsg.in = (void*)(&vpcInMsg);
    dvppApiCtlMsg.in_size = sizeof(vpc_in_msg);
    IDVPPAPI *pidvppapi = NULL;
    CreateDvppApi(pidvppapi);
    if(pidvppapi!=NULL)    {
        if(DvppCtl(pidvppapi,DVPP_CTL_VPC_PROC,&dvppApiCtlMsg)!= 0)        {
            printf("call dvppctl process faild!\n");
            DestroyDvppApi(pidvppapi);
            free(vpcInMsg.overlay.in_buffer_image);
            free(vpcInMsg.overlay.in_buffer_text);
            return ;
        }
        DestroyDvppApi(pidvppapi);
        FILE * fp   = fopen("./out_overlay_image_share","wb+");
        fwrite(vpcInMsg.overlay.auto_overlay_out_buffer->getBuffer(),
1,vpcInMsg.overlay.in_width_image*vpcInMsg.overlay.in_high_image*3/2,fp);
        fflush(fp);
        fclose(fp);
        fp=NULL;
        return ;
    }   else    {
        printf("pidvppapi is null!\n");
        return ;
    }
    if(NULL != vpcInMsg.overlay.in_buffer_image)    {
        free(vpcInMsg.overlay.in_buffer_image);
        vpcInMsg.overlay.in_buffer_image = NULL;
    }
    if(NULL != vpcInMsg.overlay.in_buffer_text)    {
        free(vpcInMsg.overlay.in_buffer_text);
        vpcInMsg.overlay.in_buffer_text = NULL;
    }
```

- Rawdata拼接基础功能调用实例

```
    dvppapi_ctl_msg dvppApiCtlMsg;
    vpc_in_msg vpcInMsg;
     //配置RawData拼接参数
    vpcInMsg.use_flag = 3;
```

```
    vpcInMsg.collage.image_type = 0;
    vpcInMsg.collage.image_rank_type = 0;
    vpcInMsg.collage.bit_width = 8;
    vpcInMsg.collage.in_width = 1000;
    vpcInMsg.collage.in_high = 1000;
    vpcInMsg.collage.width_stride = 2;
    vpcInMsg.collage.high_stride = 2;
    vpcInMsg.collage.collage_type = 0;
    vpcInMsg.collage.auto_out_buffer = make_shared<AutoBuffer>();
    char image_file[4][50] = {"yuv_data_0","yuv_data_1","yuv_data_2","yuv_data_3"};
    //初始化图像内存without stride************begin************
    for(int i=0; i<4; i++) {
        if(NULL == (vpcInMsg.collage.in_buffer[i] =
(char*)malloc(vpcInMsg.collage.in_width*                                    vpcInMsg.collage.in_high*3/2)))
        {
            printf("in_buffer_image alloc failed!\n");
            for(int j=0; j<i; j++) {
                free(vpcInMsg.collage.in_buffer[j]);
            }
            return ;
        }
        //打开图片至申请的内存中
        FILE * yuvimage = fopen(image_file[i],"rb");
        if(NULL == yuvimage)        {
            printf("PVC_TEST: yuv image open faild!");
            return ;
        }        else        {
            fread(vpcInMsg.collage.in_buffer[i], 1,vpcInMsg.collage.in_width*
                vpcInMsg.collage.in_high*3/2,yuvimage);
            fclose(yuvimage);
            yuvimage = NULL;
        }
    }

    dvppApiCtlMsg.in = (void*)(&vpcInMsg);
    dvppApiCtlMsg.in_size = sizeof(vpc_in_msg);
    IDVPPAPI *pidvppapi = NULL;
    CreateDvppApi(pidvppapi);
    if(pidvppapi!=NULL)    {
        if(DvppCtl(pidvppapi,DVPP_CTL_VPC_PROC,&dvppApiCtlMsg)!= 0)        {
            printf("call dvppctl process faild!\n");
            DestroyDvppApi(pidvppapi);
            for(int i=0; i<4; i++) {
                free(vpcInMsg.collage.in_buffer[i]);
            }
            return ;
        }
        DestroyDvppApi(pidvppapi);
        FILE * fp   = fopen("./out_collage_image_share","wb+");
        fwrite(vpcInMsg.collage.auto_out_buffer->getBuffer(),
1,vpcInMsg.collage.in_width*vpcInMsg.collage.in_high*6,fp);
        fflush(fp);
        fclose(fp);
        fp=NULL;
        return ;
    }   else    {
        printf("pidvppapi is null!\n");
        return ;
    }
    for(int i=0; i<4; i++) {
        if(NULL != vpcInMsg.collage.in_buffer[i])        {
            free(vpcInMsg.collage.in_buffer[i]);
            vpcInMsg.collage.in_buffer[i] = NULL;
        }
    }
```

- VPCAPI调用

```
IDVPPAPI *pidvppapi = NULL;
CreateDvppApi(pidvppapi);
if(pidvppapi!=NULL)
```

```
{
 if(0 != DvppCtl(pidvppapi,DVPP_CTL_VPC_PROC,&dvppApiCtlMsg))
 {
  printf("call dvppctl process faild!\n");
  DestroyDvppApi(pidvppapi);
  return -1;
 }
}
else
printf("pidvppapi is null!\n");
DestroyDvppApi(pidvppapi);
```

- VPC Resize参数配置获取功能调用示例

  gXxxxx为配置参数

```
dvppapi_ctl_msg dvppApiCtlMsg;
vpc_in_msg vpcInMsg;
resize_param_in_msg resize_in_param;
resize_param_out_msg resize_out_param;
resize_in_param.src_width = gWidth;
resize_in_param.src_high = gHigh;
resize_in_param.hmax = gHmax;
resize_in_param.hmin = gHmin;
resize_in_param.vmax = gVmax;
resize_in_param.vmin = gVmin;
resize_in_param.dest_width = floor(gHinc * (gHmax - gHmin + 1) + 0.5);
resize_in_param.dest_high = floor(gVinc * (gVmax - gVmin + 1) + 0.5);
dvppApiCtlMsg.in = (void *)(&resize_in_param);
dvppApiCtlMsg.out = (void *)(&resize_out_param);
IDVPPAPI *pidvppapi = NULL;
CreateDvppApi(pidvppapi);
if (0 != DvppCtl(pidvppapi, DVPP_CTL_TOOL_CASE_GET_RESIZE_PARAM, &dvppApiCtlMsg))
{
  printf("call dvppctl process faild!\n");
  DestroyDvppApi(pidvppapi);
  return;
}
int bufferSize = 0;
vpcInMsg.format = gFormat;
vpcInMsg.cvdr_or_rdma = gcvdr_or_rdma;
vpcInMsg.bitwidth = gBitwidth;
vpcInMsg.rank = gRank;
vpcInMsg.width = gWidth;
vpcInMsg.high = gHigh;
vpcInMsg.stride = gStride;
vpcInMsg.hmax = resize_out_param.hmax;
vpcInMsg.hmin = resize_out_param.hmin;
vpcInMsg.vmax = resize_out_param.vmax;
vpcInMsg.vmin = resize_out_param.vmin;
vpcInMsg.vinc = resize_out_param.vinc;
vpcInMsg.hinc = resize_out_param.hinc;
```

8.3.5.3 CMDLIST 功能及参数说明

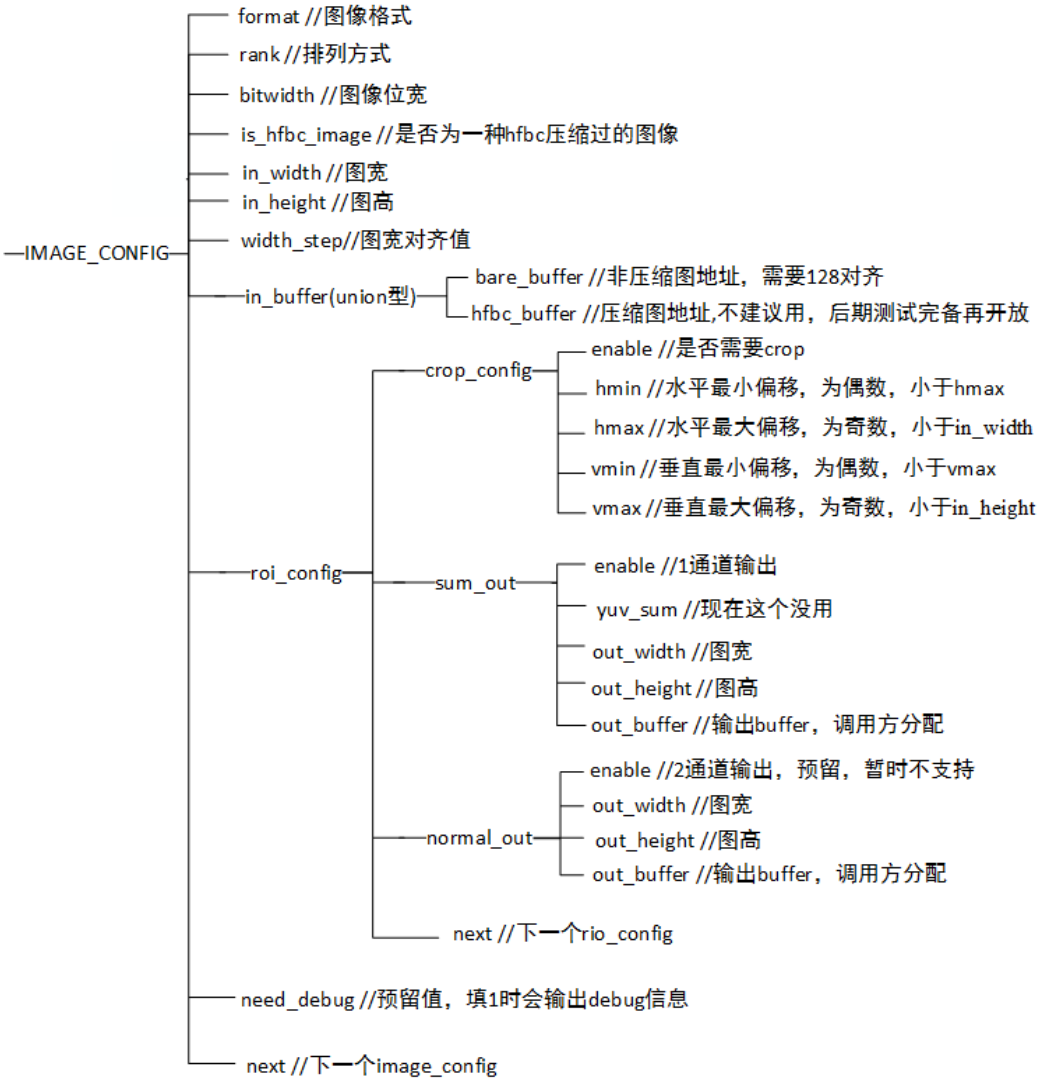
不推荐使用CMDLIST功能，后续版本会删除。旧版本中的VPC功能和CMDLIST功能均迁移到VPC功能，请参见8.3.2.2.2 VPC参数说明处的参数说明进行开发。

CMDLIST是VPC的一个扩展功能，其概念为将原来需要多次启动VPC执行的功能，合并到一次的“启动”与“完成中断返回”间完成。简单来说，CMDLIST适用于对时延要求不高，处理的图片分辨率低，数量多的场景。

入参

CMDLIST输入为结构体，详细参数如图8-5、表8-27所示。

图 8-5 CMDLIST 结构体



详细信息以及取值范围请参见**表8-27**。

表 8-27 结构体说明

成员变量	说明	取值范围
unsigned int: format	输入图像类型。	typedef enum { yuv420_semi_plannar =0,//0 yuv422_semi_plannar,//1 yuv444_semi_plannar,//2 yuv422_packed,//3 yuv444_packed,//4 rgb888_packed,//5 xrgb8888_packed,//6 yuv400_semi_plannar,//7 invalid_image_type,//20 }Imge_Type;
unsigned int: rank	输出图像格式(NV12或NV21)。	详见**表8-26**。
unsigned int: bitwidth	位深，通常为8bit。10bit只有在图像格式为YUV/YVU420 Semi-Planar且为HFBC压缩场景下使用。	8：8bit 10：10bit
int: is_hfbc_image	输入图像通道，通常设置为1，走CVDR通道，仅当VDEC输出的HFBC格式数据作为输入时走RDMA通道。	0：RDMA 1：CVDR
unsigned int: in_width	输入图像宽度，必须128对齐。	128~4096
unsigned int: in_height	输入图像高度，必须16对齐。	16~4096
unsigned int: width_step	图像步长。	yuv400sp、yuv420sp、yuv422sp、yuv444sp：width对齐到128。 yuv422packed：width * 2后对齐到128。 yuv444packed、rgb888：width * 3后对齐到128。 xrgb8888：width * 4后对齐到128。
unsigned int: need_debug	内部调试预留值，通常设置为0。	0：普通模式。 1：debug模式。

版权所有 © 华为技术有限公司

成员变量	说明	取值范围

CMDLIST_IN_BUFFER：in_buffer	输入图像内存地址指针，是一个联合体，普通非压缩图像选择bare_buffer，需要128对齐。HFBC压缩图像地址选择hfbc_buffer。	union CMDLIST_IN_BUFFER
		{
		char* bare_buffer;
		RDMACHANNEL* hfbc_buffer;
		};
		与vpc_in_msg中RDMACHANNEL相同，
		详见**Rdma通道结构体RDMACHANNEL**。

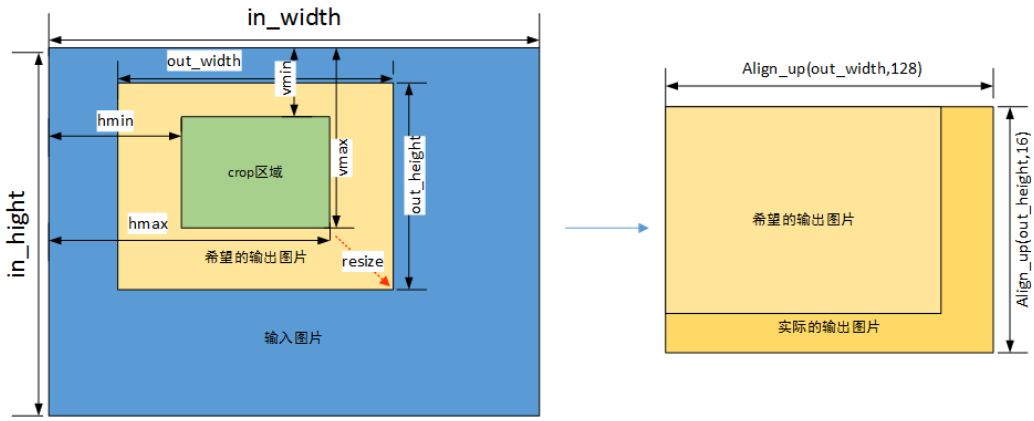
ROI_CONFIG：roi_config	输出参数配置结构体。	详见**ROI_CONFIG结构体**
IMAGE_CONFIG*：next	下一个IMAGE_CONFIG结构体指针。多图时配置，单图时配置为NULL。	--

说明

结构体详细定义请见DDK包下include/inc/dvpp/dvpp_config.h。

可以通过**图8-6**所示，直观了解输入参数信息。

图 8-6 输入参数示例说明



此图将crop和resize操作归一化到一个图中，理解如下。

- 只做crop时，相当于resize系数为1。
- 只做resize时，相当于crop大小为原图。

版权所有 © 华为技术有限公司

说明

- 输入图片的宽128对齐，高16对齐，且输入输出图片最大分辨率为4K。
- 内存地址对齐，需要用到如下接口。
  输入输出图片内存地址128对齐，需要用HIAI_DVPP_DMalloc的申请大页内存。
  申请接口：HIAI_DVPP_DMalloc(size)。
- 根据输出图片比上crop后图片计算缩放系数，需要在[0.03125,4]范围内。
- 输出内存大小需要根据输出宽128对齐，高16对齐后的分辨率计算。
- 调用一次cmdlist接口，最多支持32张图，每图最多支持256个框。

出参

暂无，其中输出图片在sum_out中的out_buffer中，用户可读取该内存获取图片。

输入图片格式与输出图片格式对应rank配置表请参见表8-27。

调用示例

```
/*****************************************************************************************/
/*****用例说明：*****************************************************************************
此用例以一张1920x1088 yuv420 nv12图片作为输入
输入图片文件名： "file1_1920x1088_nv12.yuv"
对输入图片的操作：抠出5张子图，均缩放到224x224；
 ****************************************************************************************
 *****************************************************************************************/
int main()
{
    int ret = 0;
    //读取输入文件
    char image_file[128] = "file1_1920x1088_nv12.yuv";
    ifstream in_stream(image_file);
    if (!in_stream.is_open()) {
        printf("can not open %s.\n", image_file);
        return -1;
    }
    in_stream.seekg(0, ios::end);
    int file_len = in_stream.tellg();
    char* in_buffer = (char *)HIAI_DVPP_DMalloc(file_len);
    in_stream.seekg(0, ios::beg);
    in_stream.read(in_buffer, file_len);
    in_stream.close();
    //开始添加第一个image的配置
    IMAGE_CONFIG* image_config = (IMAGE_CONFIG*)malloc(sizeof(IMAGE_CONFIG));
    image_config->in_buffer.bare_buffer = in_buffer; //目前用到的都是非压缩图
    image_config->format = 0;
    image_config->rank = 1;//输入nv12输出也为nv12，rank填1
    image_config->bitwidth = 8;//目前用到的都是8bit一个单位的
    image_config->in_width = 1920;
    image_config->in_height = 1088;
    image_config->width_step = 1920;//这个值同宽
    //开始添加第一个抠图的配置,该例子抠图区域:(0,0),(511,511)围成的区域
    ROI_CONFIG* roi_config = &image_config->roi_config;
    //开始配置crop参数
    roi_config->crop_config.enable = 1; //注：当只要resize的时候，crop参数只要这个配置为0
    roi_config->crop_config.hmin = 0;
    roi_config->crop_config.hmax = 511;
    roi_config->crop_config.vmin = 0;
    roi_config->crop_config.vmax = 511;
    //开始配置输出通道参数
    roi_config->sum_out.enable = 1; //开启第一通道输出
    roi_config->sum_out.out_width = 224;
    roi_config->sum_out.out_height = 224;
    int out_buffer_size = AlignUp(224,128)*AlignUp(224,16)*3/2;
```

```
    roi_config->sum_out.out_buffer = (char*)HIAI_DVPP_DMalloc(out_buffer_size);
    ROI_CONFIG* last_roi = roi_config;
    //开始添加第2到第5个抠图的配置
    for (int i = 0 ; i < 4; i++) {
        ROI_CONFIG* roi_config = (ROI_CONFIG*)malloc(sizeof(ROI_CONFIG));
        //开始配置crop参数
        roi_config->crop_config.enable = 1;
        roi_config->crop_config.hmin = 100*i;
        roi_config->crop_config.hmax = 299 + 100*i;
        roi_config->crop_config.vmin = 100*i;
        roi_config->crop_config.vmax = 299 + 100*i;
        //开始配置输出通道参数
        roi_config->sum_out.enable = 1;
        roi_config->sum_out.out_width = 224;
        roi_config->sum_out.out_height = 224;
        out_buffer_size = AlignUp(224,128)*AlignUp(224,16)*3/2;
        roi_config->sum_out.out_buffer = (char*)HIAI_DVPP_DMalloc(out_buffer_size);
        roi_config->next = nullptr;
        last_roi->next = roi_config;
        last_roi = roi_config;
    }
    //开始调用dvpp的cmdlist接口
    IDVPPAPI *pidvppapi = NULL;
    //无论后面调用多少次，createdvppapi只要一次调用就好
    ret = CreateDvppApi(pidvppapi);
    if (ret != 0) {
        printf("creat dvpp api faild!\n");
        return -1;
    }
    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = (void *)(image_config);
    dvppApiCtlMsg.in_size = sizeof(IMAGE_CONFIG);
    ret = DvppCtl(pidvppapi, DVPP_CTL_CMDLIST_PROC, &dvppApiCtlMsg);
    if (0 != ret) {
        printf("call cmdlist dvppctl process faild!\n");
    } else {
        printf("cmdlist success.\n");
    }
    ret += DestroyDvppApi(pidvppapi);
    roi_config = image_config->roi_config.next;
    while (roi_config != nullptr) {
        ROI_CONFIG* next_roi = roi_config->next;
        UNMAP(roi_config->sum_out.out_buffer, out_buffer_size);
        free(roi_config);
        roi_config = next_roi;
    }
    UNMAP(image_config->roi_config.sum_out.out_buffer, out_buffer_size);
    free(image_config);
    UNMAP(in_buffer, file_len);
    return ret;
}
```

8.3.5.4 VENC 功能及参数说明

不推荐使用旧版本中的VENC功能，后续版本会删除。旧版本中的VENC功能已迁移到VENC新接口，请参见8.3.4 VENC功能接口进行开发。

功能

实现YUV420/YVU420图片数据的编码。

- VENC支持以下格式输入：
  YUV420 semi-planner NV12/NV21-8bit
- VENC输出格式为：
  H264 BP/MP/HP

H265 MP

VENC 性能指标

场景	总帧率
1080p * 1路（不支持多路）	30fps

入参：venc_in_msg

成员变量	说明	取值范围
Int width	图像宽度。	128~1920，且为偶数。
Int height	图像高度。	128~1920，且为偶数。
Int coding_type	视频编码协议H265-main level（0）、H264-baseline level（1）、H264-main level(2)、H264-high level（3）	0~3 ● 0：H265 main level。 ● 1：H264 baseline level。 ● 2：H264 main level。 ● 3：H264high level。
Int YUV_store_type	YUV图像存储格式。	0或者1 ● 0：YUV420 semi-planner ● 1：YVU420 semi-planner
char* input_data	输入图像数据地址。	不能为NULL。
Int input_data_size	输入图像数据大小。	正数。
shared_ptr<AutoBuffer> output_data_queue	输出编码码流数据地址。	需要配置此buffer，此buffer为调用方申请智能指针当做参数传入DVPP。用户可读取该buffer获取输出图片。 必填

出参

暂无。

调用示例

```
void TEST_3() //venc demo
{
    int read_file_size;
    int unit_file_size;
```

```
    FILE *fp = fopen(in_file_name, "rb");
    if (fp == NULL) {
        printf("open file: %s failed.\n", in_file_name);
        return;
    }
    printf("open yuv success \n");
    fseek(fp, 0L, SEEK_END);
    int file_size = ftell(fp);
    fseek(fp, 0L, SEEK_SET);

    venc_in_msg venc_msg;
    venc_msg.width = gWidth;
    venc_msg.height = gHigh;
    venc_msg.coding_type = gFormat;
    venc_msg.YUV_store_type = gBitwidth;
    venc_msg.output_data_queue = make_shared<AutoBuffer>();
    unit_file_size = gWidth * gHigh * 3 / 2 * MAX_FRAME_NUM_VENC; //单次文件大小为16帧
    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = (void *)(&venc_msg);
    dvppApiCtlMsg.in_size = sizeof(venc_in_msg);
    IDVPPAPI *pidvppapi = NULL;
    CreateDvppApi(pidvppapi);
    char out_filename[] = "venc.bin";
    FILE *outputBufferFile;
    outputBufferFile = fopen (out_filename, "wb+");

    do{
        read_file_size = file_size>unit_file_size? unit_file_size:file_size; //每次读取文件大小不能超过16帧
        venc_msg.input_data = (char *)malloc(read_file_size);
        int read_len = fread(venc_msg.input_data, 1, read_file_size, fp);
        printf("file size is %d,read len is %d.\n", read_file_size, read_len);
        venc_msg.input_data_size = read_len;

        if (pidvppapi != NULL) {
            if (DvppCtl(pidvppapi, DVPP_CTL_VENC_PROC, &dvppApiCtlMsg) != 0) {
                printf("call dvppctl process faild!\n");
                DestroyDvppApi(pidvppapi);
                fclose(fp);
                fclose(outputBufferFile);
                return;
            }
            if (venc_msg.output_data_queue->getBufferSize() > 100) { //防止编码结果只含有头部信息
                char* out_buf = venc_msg.output_data_queue->getBuffer();
                int out_buf_size = venc_msg.output_data_queue->getBufferSize();
                int write_size = fwrite(out_buf, 1, out_buf_size, outputBufferFile);
                fflush(outputBufferFile);
            } else {
                printf("venc output data is too small : %d \n", venc_msg.output_data_queue->getBufferSize());
            }

        } else {
            printf("pidvppapi is null!\n");
        }

        if (venc_msg.input_data != NULL) {
            free(venc_msg.input_data);
            venc_msg.input_data = NULL;
        }

        file_size = file_size - unit_file_size;
    } while (file_size > 0);

    DestroyDvppApi(pidvppapi);
    fclose(outputBufferFile);
    fclose(fp);
    return;
}
```

8.3.6 调用示例

8.3.6.1 实现 VPC 功能

概念说明

关于抠图、缩放、叠加、上偏移、下偏移、左偏移、右偏移等概念，请参见**8.3.1.3 VPC功能**。

示例 1：仅原图缩放

关键参数设置如下：

- 抠图区域宽高跟输入图片的真实宽高相同，抠图区域各偏移值的设置如下：
  - 左偏移leftOffset=0
  - 上偏移upOffset=0
  - 右偏移rightOffset - 左偏移leftOffset+1=输入图片真实宽
  - 下偏移downOffset - 上偏移upOffset+1=输入图片真实高
- 贴图区域宽高是缩放后图片的宽高，可指定贴图区域的位置，例如：
  若指定贴图区域的位置在输出图片的左上角，则贴图区域各偏移值的设置如下：
  - 左偏移leftOffset=0
  - 上偏移upOffset=0
  - 右偏移rightOffset - 左偏移leftOffset+1=缩放后图片的宽
  - 下偏移downOffset - 上偏移upOffset+1=缩放后图片的高
- 如果是8K的原图缩放，则inputFormat和outputFormat只支持依次设置为INPUT_YUV420_SEMI_PLANNER_UV、OUTPUT_YUV420SP_UV或设置为INPUT_YUV420_SEMI_PLANNER_VU、OUTPUT_YUV420SP_VU；如果是非8K的原图缩放，inputFormat和outputFormat的设置可参考**表8-10**。

示例代码：

本示例是将yuv420sp格式的图片从1080p缩放到720p。

```
void NewVpcTest1()
{
    uint32_t inWidthStride = 1920;
    uint32_t inHeightStride = 1080;
    uint32_t outWidthStride = 1280;
    uint32_t outHeightStride = 720;
    uint32_t inBufferSize = inWidthStride * inHeightStride * 3 / 2; // 1080P yuv420sp Image
    uint32_t outBufferSize = outWidthStride * outHeightStride * 3 / 2; // 720P yuv420sp bImage
    uint8_t* inBuffer  = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(inBufferSize)); // Construct an input picture.
    if (inBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc input buffer");
        return;
    }
    uint8_t* outBuffer = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(outBufferSize)); // Construct an output
picture.
    if (outBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc output buffer");
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        return;
    }
```

```
FILE* fp = fopen("dvpp_vpc_1920x1080_nv12.yuv", "rb+");
if (fp == nullptr) {
    HIAI_ENGINE_LOG(HIAI_ERROR, "fopen file failed");
    DFreeInOutBuffer(inBuffer, outBuffer);
    return;
}

fread(inBuffer, 1, inBufferSize, fp);
fclose(fp);
fp = nullptr;
// Construct the input picture configuration.
std::shared_ptr<VpcUserImageConfigure> imageConfigure(new VpcUserImageConfigure);
imageConfigure->bareDataAddr = inBuffer;
imageConfigure->bareDataBufferSize = inBufferSize;
imageConfigure->widthStride = inWidthStride;
imageConfigure->heightStride = inHeightStride;
imageConfigure->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV;
imageConfigure->outputFormat = OUTPUT_YUV420SP_UV;
imageConfigure->yuvSumEnable = false;
imageConfigure->cmdListBufferAddr = nullptr;
imageConfigure->cmdListBufferSize = 0;
std::shared_ptr<VpcUserRoiConfigure> roiConfigure(new VpcUserRoiConfigure);
roiConfigure->next = nullptr;
VpcUserRoiInputConfigure* inputConfigure = &roiConfigure->inputConfigure;
// Set the drawing area, [0,0] in the upper left corner of the area,
// and [1919,1079] in the lower right corner.
inputConfigure->cropArea.leftOffset  = 0;
inputConfigure->cropArea.rightOffset = inWidthStride - 1;
inputConfigure->cropArea.upOffset    = 0;
inputConfigure->cropArea.downOffset  = inHeightStride - 1;
VpcUserRoiOutputConfigure* outputConfigure = &roiConfigure->outputConfigure;
outputConfigure->addr = outBuffer;
outputConfigure->bufferSize = outBufferSize;
outputConfigure->widthStride = outWidthStride;
outputConfigure->heightStride = outHeightStride;
// Set the map area, coordinate [0,0] in the upper left corner of the map area,
// and [1279,719] in the lower right corner.
outputConfigure->outputArea.leftOffset  = 0;
outputConfigure->outputArea.rightOffset = outWidthStride - 1;
outputConfigure->outputArea.upOffset    = 0;
outputConfigure->outputArea.downOffset  = outHeightStride - 1;

imageConfigure->roiConfigure = roiConfigure.get();

IDVPPAPI *pidvppapi = nullptr;
int32_t ret = CreateDvppApi(pidvppapi);
if (ret != 0) {
    HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "create dvpp api fail.");
    DFreeInOutBuffer(inBuffer, outBuffer);
    return;
}
dvppapi_ctl_msg dvppApiCtlMsg;
dvppApiCtlMsg.in = static_cast<void*>(imageConfigure.get());
dvppApiCtlMsg.in_size = sizeof(VpcUserImageConfigure);
ret = DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg);
if (ret != 0) {
    HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "call vpc dvppctl process faild!");
    ret = DestroyDvppApi(pidvppapi);
    DFreeInOutBuffer(inBuffer, outBuffer);
    return;
} else {
    HIAI_ENGINE_LOG(HIAI_OK, "NewVpcTest1::call vpc dvppctl process success!");
}

FILE* outImageFp = fopen("NewVpcTest1Out.yuv", "wb+");
if (outImageFp == nullptr) {
    HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "open NewVpcTest1Out.yuv faild!");
    ret = DestroyDvppApi(pidvppapi);
    DFreeInOutBuffer(inBuffer, outBuffer);
```

```
        return;
    }

    fwrite(outBuffer, 1, outBufferSize, outImageFp);

    ret = DestroyDvppApi(pidvppapi);
    DFreeInOutBuffer(inBuffer, outBuffer);
    fclose(outImageFp);
    outImageFp = nullptr;
    return;
}
```

示例 2：抠一张图+缩放

关键参数设置如下：

- 抠图区域宽高是缩放前图片的宽高，可指定抠图区域的位置，例如：
  若指定抠图区域的位置在输入图片的中间，则抠图区域各偏移值的设置如下：
  - 左偏移leftOffset=100
  - 右偏移rightOffset=499
  - 上偏移upOffset=100
  - 下偏移downOffset=399
  - 右偏移rightOffset - 左偏移leftOffset+1=抠图区域图片的宽
  - 下偏移downOffset - 上偏移upOffset+1=抠图区域图片的高
- 贴图区域宽高是缩放后图片的宽高，可指定贴图区域的位置，例如：
  若指定贴图区域的位置在输出图片的中间，则贴图区域各偏移值的设置如下：
  - 左偏移leftOffset=256
  - 右偏移rightOffset=399
  - 上偏移upOffset=200
  - 下偏移downOffset=399
  - 右偏移rightOffset - 左偏移leftOffset+1=缩放后图片的宽
  - 下偏移downOffset - 上偏移upOffset+1=缩放后图片的高

示例代码：

本示例是从yuv420sp格式、1080p的图片中抠出一张图，经过缩放后，将缩放后的图片贴到720p的输出图片（由用户申请的空输出内存产生的空白图片）中。如果需要将已有图片作为输出图片，用户需在申请输出内存后，将已有图片读入输出内存，代码示例请参考**示例5：叠加**。

```
void NewVpcTest2()
{
    uint32_t inWidthStride = 1920;
    uint32_t inHeightStride = 1080;
    uint32_t outWidthStride = 1280;
    uint32_t outHeightStride = 720;
    uint32_t inBufferSize = inWidthStride * inHeightStride * 3 / 2; // 1080P yuv420sp Image
    uint32_t outBufferSize = outWidthStride * outHeightStride * 3 / 2; // 720P yuv420sp Image
    uint8_t* inBuffer  = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(inBufferSize)); //  Construct an input
picture.
    if (inBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc input buffer");
        return;
    }
    uint8_t* outBuffer = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(outBufferSize)); //  Construct an output
picture.
```

```
    if (outBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc output buffer");
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        return;
    }
    (void)memset_s(outBuffer, outBufferSize, 0x80, outBufferSize);

    FILE* fp = fopen("dvpp_vpc_1920x1080_nv12.yuv", "rb+");
    if (fp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "fopen file failed");
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }

    fread(inBuffer, 1, inBufferSize, fp);
    fclose(fp);
    fp = nullptr;
    // Construct the input picture configuration
    std::shared_ptr<VpcUserImageConfigure> imageConfigure(new VpcUserImageConfigure);
    imageConfigure->bareDataAddr = inBuffer;
    imageConfigure->bareDataBufferSize = inBufferSize;
    imageConfigure->widthStride = inWidthStride;
    imageConfigure->heightStride = inHeightStride;
    imageConfigure->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV;
    imageConfigure->outputFormat = OUTPUT_YUV420SP_UV;
    imageConfigure->yuvSumEnable = false;
    imageConfigure->cmdListBufferAddr = nullptr;
    imageConfigure->cmdListBufferSize = 0;
    std::shared_ptr<VpcUserRoiConfigure> roiConfigure(new VpcUserRoiConfigure);
    roiConfigure->next = nullptr;
    VpcUserRoiInputConfigure* inputConfigure = &roiConfigure->inputConfigure;
    // Set the drawing area, [100,100] in the upper left corner of the area, and [499,499] in the lower right
corner.
    inputConfigure->cropArea.leftOffset  = 100;
    inputConfigure->cropArea.rightOffset = 499;
    inputConfigure->cropArea.upOffset    = 100;
    inputConfigure->cropArea.downOffset  = 499;
    VpcUserRoiOutputConfigure* outputConfigure = &roiConfigure->outputConfigure;
    outputConfigure->addr = outBuffer;
    outputConfigure->bufferSize = outBufferSize;
    outputConfigure->widthStride = outWidthStride;
    outputConfigure->heightStride = outHeightStride;
    // Set the map area, [256,200] in the upper left corner of the map area, and [399,399] in the lower right
corner.
    outputConfigure->outputArea.leftOffset  = 256; // The offset value must be 16-pixel-aligned.
    outputConfigure->outputArea.rightOffset = 399;
    outputConfigure->outputArea.upOffset    = 200;
    outputConfigure->outputArea.downOffset  = 399;

    imageConfigure->roiConfigure = roiConfigure.get();

    IDVPPAPI *pidvppapi = nullptr;
    int32_t ret = CreateDvppApi(pidvppapi);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "create dvpp api fail.");
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }
    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = static_cast<void*>(imageConfigure.get());
    dvppApiCtlMsg.in_size = sizeof(VpcUserImageConfigure);
    ret = DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "call vpc dvppctl process faild!");
        ret = DestroyDvppApi(pidvppapi);
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    } else {
```

```
        HIAI_ENGINE_LOG(HIAI_OK, "NewVpcTest2::call vpc dvppctl process success!");
    }

    FILE* outImageFp = fopen("NewVpcTest2Out.yuv", "wb+");
    if (outImageFp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "open NewVpcTest2Out.yuv failed");
        DestroyDvppApi(pidvppapi);
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }
    fwrite(outBuffer, 1, outBufferSize, outImageFp);

    ret = DestroyDvppApi(pidvppapi);
    DFreeInOutBuffer(inBuffer, outBuffer);
    fclose(outImageFp);
    outImageFp = nullptr;
    return;
}
```

示例 3：抠多张图+缩放+拼接

关键参数设置：抠图区域上偏移、下偏移、左偏移、右偏移以及贴图区域上偏移、下偏移、左偏移、右偏移的配置，请参见**示例2：抠一张图+缩放**中的说明。

示例代码：本示例是从yuv420sp格式、1080p的图片中抠出多张图，经过缩放后，将缩放后的多张图片拼接到720p的输出图片（由用户申请的空输出内存产生的空白图片）中，贴图区域拼接的位置由上偏移、下偏移、左偏移、右偏移的值决定。如果需要将已有图片作为输出图片，用户需在申请输出内存后，将已有图片读入输出内存，代码示例请参考**示例5：叠加**。

```
void NewVpcTest3()
{
    uint32_t inWidthStride = 1920;
    uint32_t inHeightStride = 1080;
    uint32_t outWidthStride = 1280;
    uint32_t outHeightStride = 720;
    uint32_t inBufferSize = inWidthStride * inHeightStride * 3 / 2; // 1080P yuv420sp Image
    uint32_t outBufferSize = outWidthStride * outHeightStride * 3 / 2; // 720P yuv420sp Image
    uint8_t* inBuffer = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(inBufferSize)); // Construct an input picture.
    if (inBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc input buffer");
        return;
    }

    FILE* fp = fopen("dvpp_vpc_1920x1080_nv12.yuv", "rb+");
    if (fp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "fopen file failed");
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        return;
    }

    fread(inBuffer, 1, inBufferSize, fp);
    fclose(fp);
    fp == nullptr;
    // Construct the input picture configuration.
    std::shared_ptr<VpcUserImageConfigure> imageConfigure(new VpcUserImageConfigure);
    imageConfigure->bareDataAddr = inBuffer;
    imageConfigure->bareDataBufferSize = inBufferSize;
    imageConfigure->widthStride = inWidthStride;
    imageConfigure->heightStride = inHeightStride;
    imageConfigure->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV;
    imageConfigure->outputFormat = OUTPUT_YUV420SP_UV;
    imageConfigure->yuvSumEnable = false;
    imageConfigure->cmdListBufferAddr = nullptr;
    imageConfigure->cmdListBufferSize = 0;
    std::shared_ptr<VpcUserRoiConfigure> lastRoi;
```

```
    std::vector<std::shared_ptr<VpcUserRoiConfigure>> roiVector;
    for (uint32_t i = 0; i < 5; i++) {
        std::shared_ptr<VpcUserRoiConfigure> roiConfigure(new VpcUserRoiConfigure);
        roiVector.push_back(roiConfigure);
        roiConfigure->next = nullptr;
        VpcUserRoiInputConfigure* inputConfigure = &roiConfigure->inputConfigure;
        // Set the drawing area.
        inputConfigure->cropArea.leftOffset  = 100 + i * 16;
        inputConfigure->cropArea.rightOffset = 499 + i * 16;
        inputConfigure->cropArea.upOffset    = 100 + i * 16;
        inputConfigure->cropArea.downOffset  = 499 + i * 16;
        VpcUserRoiOutputConfigure* outputConfigure = &roiConfigure->outputConfigure;
        uint8_t* outBuffer = static_cast<uint8_t*>(HIAI_DVPP_DMalloc(outBufferSize)); // Construct an input
picture
        if (outBuffer == nullptr) {
            HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc output buffer");
            HIAI_DVPP_DFree(inBuffer);
            inBuffer = nullptr;
            FreeMultipleRio(imageConfigure->roiConfigure);
            return;
        }
        (void)memset_s(outBuffer, outBufferSize, 0x80, outBufferSize);
        outputConfigure->addr = outBuffer;
        outputConfigure->bufferSize = outBufferSize;
        outputConfigure->widthStride = outWidthStride;
        outputConfigure->heightStride = outHeightStride;
        // Set the map area.
        outputConfigure->outputArea.leftOffset  = 256 + i * 16; // The offset value must be 16-pixel-aligned.
        outputConfigure->outputArea.rightOffset = 399 + i * 16;
        outputConfigure->outputArea.upOffset    = 256 + i * 16;
        outputConfigure->outputArea.downOffset  = 399 + i * 16;
        if (i == 0) {
            imageConfigure->roiConfigure = roiConfigure.get();
            lastRoi = roiConfigure;
        } else {
            lastRoi->next = roiConfigure.get();
            lastRoi = roiConfigure;
        }
    }

    IDVPPAPI *pidvppapi = nullptr;
    int32_t ret = CreateDvppApi(pidvppapi);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "create dvpp api fail.");
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        FreeMultipleRio(imageConfigure->roiConfigure);
        return;
    }
    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = static_cast<void*>(imageConfigure.get());
    dvppApiCtlMsg.in_size = sizeof(VpcUserImageConfigure);
    ret = DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "call vpc dvppctl process faild!");
        ret = DestroyDvppApi(pidvppapi);
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        FreeMultipleRio(imageConfigure->roiConfigure);
        return;
    } else {
        HIAI_ENGINE_LOG(HIAI_OK, "NewVpcTest3::call vpc dvppctl process success!");
    }
    FILE* outImageFp = nullptr;
    uint32_t imageCount = 0;
    char fileName[50] = {0};

    while (imageConfigure->roiConfigure != nullptr) {
        int32_t safeFuncRet = sprintf_s(fileName, sizeof(fileName), "NewVpcTest3_%dOut.yuv", imageCount);
```

```
        if (safeFuncRet == -1) {
            HIAI_ENGINE_LOG(HIAI_ERROR, "sprintf_s fail, ret = %d", safeFuncRet);
            DestroyDvppApi(pidvppapi);
            HIAI_DVPP_DFree(inBuffer);
            inBuffer = nullptr;
            FreeMultipleRio(imageConfigure->roiConfigure);
            return;
        }
        outImageFp = fopen(fileName, "wb+");
        if (outImageFp == nullptr) {
            HIAI_ENGINE_LOG(HIAI_ERROR, "open %s faild!", fileName);
            DestroyDvppApi(pidvppapi);
            HIAI_DVPP_DFree(inBuffer);
            inBuffer = nullptr;
            FreeMultipleRio(imageConfigure->roiConfigure);
            return;
        }
        fwrite(imageConfigure->roiConfigure->outputConfigure.addr, 1,
            imageConfigure->roiConfigure->outputConfigure.bufferSize, outImageFp);
        fclose(outImageFp);
        outImageFp = nullptr;
        imageConfigure->roiConfigure = imageConfigure->roiConfigure->next;
        imageCount++;
    }
    ret = DestroyDvppApi(pidvppapi);
    HIAI_DVPP_DFree(inBuffer);
    inBuffer = nullptr;
    FreeMultipleRio(imageConfigure->roiConfigure);
    return;
}
```

示例 4，8K 缩放功能

对于**8K缩放**功能，支持缩放、支持YUV420SP NV12与YUV420SP NV21之间的格式转换、不支持抠图。

示例代码：本示例是将yuv420sp格式的图片从8129*8192缩放至4000*4000。

```
void NewVpcTest4()
{
    uint32_t inWidthStride = 8192; // No need for 128 byte alignment
    uint32_t inHeightStride = 8192; // No need for 16 byte alignment
    uint32_t outWidthStride = 4000; // No need for 128 byte alignment
    uint32_t outHeightStride = 4000; // No need for 16 byte alignment
    uint32_t inBufferSize = inWidthStride * inHeightStride * 3 / 2;
    uint32_t outBufferSize = outWidthStride * outHeightStride * 3 / 2; // Construct dummy data
    uint8_t* inBuffer = (uint8_t*)HIAI_DVPP_DMalloc(inBufferSize); // Construct input image
    if (inBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc roiOutput buffer");
        return;
    }
    uint8_t* outBuffer = (uint8_t*)HIAI_DVPP_DMalloc(outBufferSize); // Construct output image
    if (outBuffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc roiOutput buffer");
        HIAI_DVPP_DFree(inBuffer);
        inBuffer = nullptr;
        return;
    }

    FILE* fp = fopen("dvpp_vpc_8192x8192_nv12.yuv", "rb+");
    if (fp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "fopen file failed");
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }
    fread(inBuffer, 1, inBufferSize, fp);
    fclose(fp);
    fp = nullptr;
```

```
    std::shared_ptr<VpcUserImageConfigure> imageConfigure(new VpcUserImageConfigure);
    imageConfigure->bareDataAddr = inBuffer;
    imageConfigure->bareDataBufferSize = inBufferSize;
    imageConfigure->isCompressData = false;
    imageConfigure->widthStride = inWidthStride;
    imageConfigure->heightStride = inHeightStride;
    imageConfigure->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV;
    imageConfigure->outputFormat = OUTPUT_YUV420SP_UV;
    imageConfigure->yuvSumEnable = false;
    imageConfigure->cmdListBufferAddr = nullptr;
    imageConfigure->cmdListBufferSize = 0;
    std::shared_ptr<VpcUserRoiConfigure> roiConfigure(new VpcUserRoiConfigure);
    roiConfigure->next = nullptr;
    VpcUserRoiInputConfigure* inputConfigure = &roiConfigure->inputConfigure; // Set the roi area
    inputConfigure->cropArea.leftOffset  = 0;
    inputConfigure->cropArea.rightOffset = inWidthStride - 1;
    inputConfigure->cropArea.upOffset = 0;
    inputConfigure->cropArea.downOffset  = inHeightStride - 1;
    VpcUserRoiOutputConfigure* outputConfigure = &roiConfigure->outputConfigure;
    outputConfigure->addr = outBuffer;
    outputConfigure->bufferSize = outBufferSize;
    outputConfigure->widthStride = outWidthStride;
    outputConfigure->heightStride = outHeightStride; // Set the map area
    outputConfigure->outputArea.leftOffset  = 0;
    outputConfigure->outputArea.rightOffset = outWidthStride - 1;
    outputConfigure->outputArea.upOffset = 0;
    outputConfigure->outputArea.downOffset  = outHeightStride - 1;
    imageConfigure->roiConfigure = roiConfigure.get();
    IDVPPAPI *pidvppapi = nullptr;
    int32_t ret = CreateDvppApi(pidvppapi);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "create dvpp api fail.");
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }
    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = static_cast<void*>(imageConfigure.get());
    dvppApiCtlMsg.in_size = sizeof(VpcUserImageConfigure);
    ret = DvppCtl(pidvppapi, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "call vpc dvppctl process faild!");
        ret = DestroyDvppApi(pidvppapi);
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    } else {
        HIAI_ENGINE_LOG(HIAI_OK, "NewVpcTest4::call vpc dvppctl process success!");
    }

    FILE* outImageFp = fopen("NewVpcTest4Out.yuv", "wb+");
    if (outImageFp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "open NewVpcTest4Out.yuv faild!");
        ret = DestroyDvppApi(pidvppapi);
        DFreeInOutBuffer(inBuffer, outBuffer);
        return;
    }
    fwrite(outBuffer, 1, outBufferSize, outImageFp);
    fclose(outImageFp);
    outImageFp = nullptr;
    ret = DestroyDvppApi(pidvppapi);
    DFreeInOutBuffer(inBuffer, outBuffer);
    return;
}
```

示例 5：叠加

如果用户需要将已有图片读入内存用于存放输出图片，将贴图区域叠加在输出图片上，则需要编写代码逻辑将图片读入内存，在申请输出内存代码**uint8_t* outBuffer =**

static_cast<uint8_t*>(HIAI_DVPP_DMalloc(outBufferSize));之后，增加如下代码：

```
FILE* fpOut = fopen("vpcOut.yuv", "rb+");
if  (fpOut == nullptr) {
    HIAI_ENGINE_LOG(HIAI_ERROR, "fopen file  failed.");
    fclose(fpOut);
    return;
}
fread(outBuffer, 1,  outBufferSize, fpOut);
fclose(fpOut);
```

8.3.6.2 实现 JPEGE 功能

- 使用**DvppGetOutParameter**接口获取内存大小，由用户指定输出内存，由用户自行释放内存，调用示例如下 。

```
void TEST_JPEGE_CUSTOM_MEMORY()
{
    sJpegeIn  inData;
    sJpegeOut outData;

    inData.width         = g_width;
    inData.height        = g_high;
    inData.heightAligned = g_high; // no need to align
    inData.format        = (eEncodeFormat)g_format;
    inData.level         = 100;

    inData.stride  = ALIGN_UP(inData.width * 2, 16);
    inData.bufSize = inData.stride * inData.heightAligned;
    if (JPGENC_FORMAT_YUV420 == (inData.format & JPGENC_FORMAT_BIT)) {
        inData.stride  = ALIGN_UP(inData.width, 16);
        inData.bufSize = inData.stride * inData.heightAligned * 3 / 2;
    }

    void* addrOrig = HIAI_DVPP_DMalloc(inData.bufSize);
    if (addrOrig == nullptr) {
        HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "can not alloc input buffer");
        return;
    }
    inData.buf = reinterpret_cast<unsigned char*>(addrOrig);

    unsigned char* tmpAddr = nullptr;

    do {
        // load img file
        FILE* fpIn = fopen(g_inFileName, "rb");
        if (nullptr == fpIn) {
            HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open input file");
            break;
        }
        // only copy valid image data, other part is pending
        if (JPGENC_FORMAT_YUV420 == (inData.format & JPGENC_FORMAT_BIT)) {
            // for yuv420semi-planar format, like nv12 / nv21
            HIAI_ENGINE_LOG("input yuv 420 data");
            // for y data
            for (uint32_t j = 0; j < inData.height; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width, fpIn);
            }
            // for uv data
            for (uint32_t j = inData.heightAligned; j < inData.heightAligned + inData.height / 2; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width, fpIn);
            }
        } else {
            // for yuv422packed format, like uyvy / vyuy / yuyv / yvyu
            HIAI_ENGINE_LOG("input yuv 422 data");
            for (uint32_t j = 0; j < inData.height; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width * 2, fpIn);
            }
        }
```

```
        fclose(fpIn);
        fpIn = nullptr;

        int32_t ret = DvppGetOutParameter((void*)(&inData), (void*)(&outData),
GET_JPEGE_OUT_PARAMETER);
        if (ret != 0) {
            HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "call DvppGetOutParameter process failed");
            break;
        }
        // 此处获得的jpgSize为估算值，实际数据长度可能要小于这个值
        // 最终jpgSize大小，以调用DvppCtl接口以后，此字段才为真正的编码后的jpgSize大小
        HIAI_ENGINE_LOG("outdata size is %d", outData.jpgSize);
        tmpAddr = (unsigned char*)HIAI_DVPP_DMalloc(outData.jpgSize);
        if (tmpAddr == nullptr) {
            HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "can not alloc output buffer");
            break;
        }
        uint32_t size = outData.jpgSize;

        // call jpege process
        dvppapi_ctl_msg dvppApiCtlMsg;
        dvppApiCtlMsg.in = (void *)&inData;
        dvppApiCtlMsg.in_size = sizeof(inData);
        dvppApiCtlMsg.out = (void *)&outData;
        dvppApiCtlMsg.out_size = sizeof(outData);

        if(!IsHandleJpegeCtlSucc(dvppApiCtlMsg, tmpAddr, size, &outData)) {
            break;
        }
    } while (0); // for resource inData.buf
    if (addrOrig != nullptr) {
        HIAI_DVPP_DFree(addrOrig);
        addrOrig = nullptr;
    }
    if (tmpAddr != nullptr) { // 注意：释放内存时需要使用tmpAddr，因为outData.jpgData在JPEGE处理后
会有地址偏移
        HIAI_DVPP_DFree(tmpAddr);
        tmpAddr = nullptr;
    }
}
```

- 不由用户指定输出内存时，DVPP内部申请内存，需由用户调用cbFree()回调函数释放内存，调用示例如下。

```
void JpegeProcess()
{
    sJpegeIn  inData;
    sJpegeOut outData;

    inData.width         = g_width;
    inData.height        = g_high;
    inData.heightAligned = g_high; // no need to align
    inData.format        = (eEncodeFormat)g_format;
    inData.level         = 100;

    if (JPGENC_FORMAT_YUV420 == (inData.format & JPGENC_FORMAT_BIT)) {
        inData.stride  = ALIGN_UP(inData.width, 16);
        inData.bufSize = inData.stride * inData.heightAligned * 3 / 2;
    } else {
        inData.stride  = ALIGN_UP(inData.width * 2, 16);
        inData.bufSize = inData.stride * inData.heightAligned;
    }

    void* addrOrig = HIAI_DVPP_DMalloc(inData.bufSize);

    if (addrOrig == nullptr) {
        HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "can not alloc input buffer");
        return;
    }
    inData.buf = reinterpret_cast<unsigned char*>(addrOrig);
```

```
    do {
        // load img file
        FILE* fpIn = fopen(g_inFileName, "rb");
        if (fpIn == nullptr) {
            HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open input file");
            break;
        }
        // only copy valid image data, other part is pending
        if (JPGENC_FORMAT_YUV420 == (inData.format & JPGENC_FORMAT_BIT)) {
            // for yuv420semi-planar format, like nv12 / nv21
            HIAI_ENGINE_LOG("input yuv 420 data");
            // for y data
            for (uint32_t j = 0; j < inData.height; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width, fpIn);
            }
            // for uv data
            for (uint32_t j = inData.heightAligned; j < inData.heightAligned + inData.height / 2; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width, fpIn);
            }
        } else {
            // for yuv422packed format, like uyvy / vyuy / yuyv / yvyu
            HIAI_ENGINE_LOG("input yuv 422 data");
            for (uint32_t j = 0; j < inData.height; j++) {
                fread(inData.buf + j * inData.stride, 1, inData.width * 2, fpIn );
            }

        }
        fclose(fpIn);
        fpIn = nullptr;

        // call jpege process
        dvppapi_ctl_msg dvppApiCtlMsg;
        dvppApiCtlMsg.in = (void*)&inData;
        dvppApiCtlMsg.in_size = sizeof(inData);
        dvppApiCtlMsg.out = (void*)&outData;
        dvppApiCtlMsg.out_size = sizeof(outData);

        IDVPPAPI *pidvppapi = nullptr;
        CreateDvppApi(pidvppapi);
        if (pidvppapi == nullptr) {
            HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "can not open dvppapi engine");
            break;
        }

        JpegeProcessBranch(pidvppapi, dvppApiCtlMsg, outData);

        DestroyDvppApi(pidvppapi);

    } while (0); // for resource inData.buf
    if (addrOrig != nullptr) {
        HIAI_DVPP_DFree(addrOrig);
        addrOrig = nullptr;
    }
}

/*
 * only handle jpege process.
 */
void JpegeProcessBranch(IDVPPAPI*& pidvppapi, dvppapi_ctl_msg& dvppApiCtlMsg, sJpegeOut&
outData)
{
    do {
        for (int i = 0;i < g_loop; i++) { // same picture loop test
            if (DvppCtl(pidvppapi, DVPP_CTL_JPEGE_PROC, &dvppApiCtlMsg)) {
                HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "call jpeg encoder fail");
                break;
            }
            if (i < g_loop - 1) {
                outData.cbFree();
```

```
            outData.jpgData = nullptr;
        }
    }
    stringstream outFile;
    outFile << g_outFileName << "_t" << std::this_thread::get_id() << ".jpg";

    FILE* fpOut = fopen(outFile.str().c_str(), "wb");
    if (fpOut != nullptr) {
        fwrite(outData.jpgData, 1, outData.jpgSize, fpOut);
        fflush(fpOut);
        fclose(fpOut);
        fpOut = nullptr;
    } else {
        HIAI_ENGINE_LOG(HIAI_JPEGE_CTL_ERROR, "call not save result file %s ", outFile.str().c_str());
    }

    outData.cbFree();
    outData.jpgData = nullptr;
    HIAI_ENGINE_LOG(HIAI_OK, "jpeg encode process completed");
  } while (0); // for resource pddvppapi
}
```

8.3.6.3 实现 JPEGD 功能

- 使用**DvppGetOutParameter**接口获取内存大小，由用户指定输出内存，由用户自行释放内存，调用示例如下 。

```
void TEST_JPEGD_CUSTOM_MEMORY()
{
    struct JpegdIn jpegdInData;
    struct JpegdOut jpegdOutData;

    if (g_rank) {
        jpegdInData.isYUV420Need = false;
    }

    FILE *fpIn = fopen(g_inFileName, "rb");
    if (fpIn == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open file %s.", g_inFileName);
        return;
    }

    do { // for resource fpIn
        fseek(fpIn, 0, SEEK_END);
        // the buf len should 8 byte larger, the driver asked
        uint32_t fileLen   = ftell(fpIn);
        jpegdInData.jpegDataSize = fileLen + 8;
        fseek(fpIn, 0, SEEK_SET);

        void* addrOrig = HIAI_DVPP_DMalloc(jpegdInData.jpegDataSize);
        if (addrOrig == nullptr) {
            HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "can not alloc input buffer");
            fclose(fpIn);
            fpIn = nullptr;
            break;
        }

        jpegdInData.jpegData = reinterpret_cast<unsigned char*>(addrOrig);
        do { // for resource inBuf
            fread(jpegdInData.jpegData, 1, fileLen, fpIn);
            fclose(fpIn);
            fpIn = nullptr;
            int32_t ret = DvppGetOutParameter((void*)(&jpegdInData), (void*)(&jpegdOutData),
GET_JPEGD_OUT_PARAMETER);
            if (ret != 0) {
                HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "call DvppGetOutParameter process failed");
                break;
            }
            HIAI_ENGINE_LOG("jpegd out size is %d", jpegdOutData.yuvDataSize);
            jpegdOutData.yuvData = (unsigned char*)HIAI_DVPP_DMalloc(jpegdOutData.yuvDataSize);
```

```
        if (jpegdOutData.yuvData == nullptr) {
            HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "can not alloc output buffer");
            break;
        }
        HIAI_ENGINE_LOG("jpegdOutData.yuvData is %p", jpegdOutData.yuvData);

        dvppapi_ctl_msg dvppApiCtlMsg;
        dvppApiCtlMsg.in = (void *)&jpegdInData;
        dvppApiCtlMsg.in_size = sizeof(jpegdInData);
        dvppApiCtlMsg.out = (void *)&jpegdOutData;
        dvppApiCtlMsg.out_size = sizeof(jpegdOutData);

        IDVPPAPI *pidvppapi = nullptr;
        CreateDvppApi(pidvppapi);

        if (pidvppapi != nullptr) {
            for (int i = 0;i < g_loop; i++) {
                if (0 != DvppCtl(pidvppapi, DVPP_CTL_JPEGD_PROC, &dvppApiCtlMsg)) {
                    HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "call dvppctl process failed");
                    break;
                }
            }
            DestroyDvppApi(pidvppapi);
        } else {
            HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "can not create dvpp api");
            break;
        }
        WriteTestJpegdResultToFile(&jpegdOutData);
    } while (0); // for resource inBuf

    if (addrOrig != nullptr) {
        HIAI_DVPP_DFree(addrOrig);
        addrOrig = nullptr;
    }
    if (jpegdOutData.yuvData != nullptr) {
        HIAI_DVPP_DFree(jpegdOutData.yuvData);
        jpegdOutData.yuvData = nullptr;
    }
} while (0); // for resource fpIn
}
```

- 不由用户指定输出内存时，DVPP内部申请内存，需由用户调用cbFree()回调函数释放内存，调用示例如下。

```
void JpegdProcess()
{
    struct jpegd_raw_data_info jpegdInData;
    struct jpegd_yuv_data_info jpegdOutData;

    if (g_rank) {
        jpegdInData.IsYUV420Need = false;
    }

    FILE *fpIn = fopen(g_inFileName, "rb");
    if (fpIn == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open file %s.", g_inFileName);
        return;
    }

    do { // for resource fpIn
        fseek(fpIn, 0, SEEK_END);
        // the buf len should 8 byte larger, the driver asked
        uint32_t fileLen   = ftell(fpIn);
        jpegdInData.jpeg_data_size = fileLen + 8;
        fseek(fpIn, 0, SEEK_SET);

        void* addrOrig = HIAI_DVPP_DMalloc(jpegdInData.jpeg_data_size);
        if (addrOrig == nullptr) {
            HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "can not alloc input buffer");
            break;
        }
```

```
        jpegdInData.jpeg_data = reinterpret_cast<unsigned char*>(addrOrig);

        do { // for resource inBuf
            fread(jpegdInData.jpeg_data, 1, fileLen, fpIn);
            dvppapi_ctl_msg dvppApiCtlMsg;
            dvppApiCtlMsg.in = (void*)&jpegdInData;
            dvppApiCtlMsg.in_size = sizeof(jpegdInData);
            dvppApiCtlMsg.out = (void*)&jpegdOutData;
            dvppApiCtlMsg.out_size = sizeof(jpegdOutData);

            IDVPPAPI *pidvppapi = nullptr;
            CreateDvppApi(pidvppapi);

            if (pidvppapi != nullptr) {
                for (int i = 0;i < g_loop; i++) { // same picture loop test
                    if (DvppCtl(pidvppapi, DVPP_CTL_JPEGD_PROC, &dvppApiCtlMsg) != 0) {
                        HIAI_ENGINE_LOG(HIAI_JPEGD_CTL_ERROR, "call dvppctl process failed");
                        break;
                    }
                    if (i < g_loop - 1) {
                        jpegdOutData.cbFree();
                        jpegdOutData.yuv_data = nullptr;
                    }
                }
                DestroyDvppApi(pidvppapi);
            } else {
                HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "can not create dvpp api");
                break;
            }

            WriteJpegdProcessResultToFile(&jpegdOutData);
            jpegdOutData.cbFree();
            jpegdOutData.yuv_data = nullptr;

        } while (0); // for resource inBuf

        if (addrOrig != nullptr) {
            HIAI_DVPP_DFree(addrOrig);
            addrOrig = nullptr;
        }

    } while (0); // for resource fpIn
    fclose(fpIn);
    fpIn = nullptr;
}
```

8.3.6.4 实现 PNGD 功能

使用**DvppGetOutParameter**接口获取内存大小，由用户指定输出内存，由用户自行释放内存，调用示例如下 。

```
void TEST_PNGD_CUSTOM_MEMORY()
{
    FILE* fpIn = fopen(g_inFileName, "rb");
    if (nullptr == fpIn) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open file %s.", g_inFileName);
        return;
    }

    fseek(fpIn, 0, SEEK_END);
    uint32_t fileLen = ftell(fpIn);
    fseek(fpIn, 0, SEEK_SET);
    void* inbuf = HIAI_DVPP_DMalloc(fileLen);
    if (inbuf == nullptr) {
        HIAI_ENGINE_LOG(HIAI_PNGD_CTL_ERROR, "can not alloc input buffer");
        fclose(fpIn);
        fpIn = nullptr;
        return;
```

```
    }

    // prepare msg
    struct PngInputInfoAPI inputPngData; // input data
    inputPngData.inputData = inbuf; // input png data
    inputPngData.inputSize = fileLen; // the size of png data

    HIAI_ENGINE_LOG("inputPngData.address: %p", inputPngData.address);
    HIAI_ENGINE_LOG("inputPngData.size: %u", inputPngData.size);
    HIAI_ENGINE_LOG("inputPngData.inputData: %p", inputPngData.inputData);
    HIAI_ENGINE_LOG("inputPngData.inputSize: %u", inputPngData.inputSize);

    fread(inputPngData.inputData, 1, fileLen, fpIn);
    fclose(fpIn);
    fpIn = nullptr;

    if (g_transform == 1) { // Whether format conversion
        inputPngData.transformFlag = 1; // RGBA -> RGB
    } else {
        inputPngData.transformFlag = 0;
    }

    struct PngOutputInfoAPI outputPngData; // output data
    int32_t ret = DvppGetOutParameter((void*)(&inputPngData), (void*)(&outputPngData),
GET_PNGD_OUT_PARAMETER);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_PNGD_CTL_ERROR, "call DvppGetOutParameter process failed");
        HIAI_DVPP_DFree(inbuf);
        inbuf = nullptr;
        return;
    }
    HIAI_ENGINE_LOG("pngd out size is  %d", outputPngData.size);
    outputPngData.address = HIAI_DVPP_DMalloc(outputPngData.size);
    if (outputPngData.address == nullptr) {
        HIAI_ENGINE_LOG(HIAI_PNGD_CTL_ERROR, "can not alloc output buffer");
        HIAI_DVPP_DFree(inbuf);
        inbuf = nullptr;
        return;
    }

    dvppapi_ctl_msg dvppApiCtlMsg; // call the interface msg
    dvppApiCtlMsg.in = (void*)(&inputPngData);
    dvppApiCtlMsg.in_size = sizeof(struct PngInputInfoAPI);
    dvppApiCtlMsg.out = (void*)(&outputPngData);
    dvppApiCtlMsg.out_size = sizeof(struct PngOutputInfoAPI);

    // use interface
    IDVPPAPI *pidvppapi = nullptr;
    CreateDvppApi(pidvppapi); // create dvppapi

    if (pidvppapi != nullptr) { // use DvppCtl interface to handle DVPP_CTL_PNGD_PROC
        for (int i = 0; i < g_loop; i++) {
            if (-1 == DvppCtl(pidvppapi, DVPP_CTL_PNGD_PROC, &dvppApiCtlMsg)) {
                HIAI_ENGINE_LOG(HIAI_PNGD_CTL_ERROR, "call dvppctl process failed");
                HIAI_DVPP_DFree(inbuf);
                inbuf = nullptr;
                HIAI_DVPP_DFree(outputPngData.address);
                outputPngData.address = nullptr;
                DestroyDvppApi(pidvppapi); // destory dvppapi
                return;
            }
        }
    } else {
        HIAI_ENGINE_LOG(HIAI_PNGD_CTL_ERROR, "can not get dvpp api");
    }

    DestroyDvppApi(pidvppapi);

    HIAI_ENGINE_LOG("output result");
```

```
    char* pAddr = (char*)outputPngData.outputData;
    FILE* fpOut = fopen(g_outFileName, "wb");
    if (fpOut == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "can not open file %s.", g_outFileName);
        HIAI_DVPP_DFree(inbuf);
        inbuf = nullptr;
        HIAI_DVPP_DFree(outputPngData.address);
        outputPngData.address = nullptr;
        return;
    }

    int size = outputPngData.width * 4;
    if (outputPngData.format == PNGD_RGB_FORMAT_NUM) {
        size = outputPngData.width * 3;
    } else if (outputPngData.format == PNGD_RGBA_FORMAT_NUM) {
        size = outputPngData.width * 4;
    }
    // copy valid image data from every line.
    for (int i = 0; i < outputPngData.high; i++) {
        fwrite(pAddr + (int)(i * outputPngData.widthAlign), size, 1, fpOut);
    }

    fclose(fpOut);
    fpOut = nullptr;
    HIAI_DVPP_DFree(inbuf);
    inbuf = nullptr;
    HIAI_DVPP_DFree(outputPngData.address);
    outputPngData.address = nullptr;
    return;
}
```

8.3.6.5 实现 VDEC 功能

对于一个视频码流，调用一次**CreateVdecApi**接口创建实例后，必须使用同一个实例调用**VdecCtl**接口进行视频解码，最后再调用一次**DestroyVdecApi**接口释放实例。

在视频解码时，如果需要从一个视频码流切换到另一个视频码流，则需要先调用**DestroyVdecApi**接口释放前一个码流的实例，再调用**CreateVdecApi**接口创建新的实例，用于处理新的视频码流。

此调用示例是读取文件名为“test_file”的h264码流文件，调用VDEC功能，将解码结果存放到“output_dir”目录。

```
// 自定义子类
class HIAI_DATA_SP_SON: public HIAI_DATA_SP {
public:
    ~ HIAI_DATA_SP_SON ()
    {
        //destruct here;
    }
    // 用户自定义新增一些成员函数，以用户需求为准，下述新增成员函数仅为示例
    uint8_t GetTotalFrameNum()
    {
        return info_.totalFrameNum;
    }

private:
    // 用户自定义一些成员变量，以用户需求为准，下述新增成员变量结构体info_仅为示例
    struct INFO {
        uint8_t totalFrameNum;
        uint8_t frameRate;
    }info_;
};

IDVPPAPI * pidvppapi_vpc = NULL;
IDVPPAPI * pidvppapi_vdec = NULL;
```

```
//回调函数举例，调用方需根据自己的需求来重新定义回调函数
void FrameReturn(FRAME* frame, void* hiaiData)
{
    static int32_t imageCount = 0;
    imageCount++;
    HIAI_ENGINE_LOG("call save frame number:[%d], width:[%d], height:[%d]", imageCount, frame->width,
frame->height);
    // The image directly output by vdec is an image of hfbc compression format, which cannot be directly
displayed.
    // It is necessary to call vpc to convert hfbc to an image of uncompressed format to display.
    HIAI_ENGINE_LOG("start call vpc interface to translate hfbc.");
    IDVPPAPI* dvppHandle = nullptr;
    int32_t ret = CreateDvppApi(dvppHandle);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "creat dvpp api fail.");
        return;
    }

    // Construct vpc input configuration.
    std::shared_ptr<VpcUserImageConfigure> userImage(new VpcUserImageConfigure);
    // bareDataAddr should be null which the image is hfbc.
    userImage->bareDataAddr = nullptr;
    userImage->bareDataBufferSize = 0;
    userImage->widthStride = frame->width;
    userImage->heightStride = frame->height;
    // Configuration input format
    string imageFormat(frame->image_format);
    if (frame->bitdepth == 8) {
        if (imageFormat == "nv12") {
            userImage->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV;
        } else {
            userImage->inputFormat = INPUT_YUV420_SEMI_PLANNER_VU;
        }
    } else {
        if (imageFormat == "nv12") {
            userImage->inputFormat = INPUT_YUV420_SEMI_PLANNER_UV_10BIT;
        } else {
            userImage->inputFormat = INPUT_YUV420_SEMI_PLANNER_VU_10BIT;
        }
    }
    userImage->outputFormat = OUTPUT_YUV420SP_UV;
    userImage->isCompressData = true;
    // Configure hfbc input address
    VpcCompressDataConfigure* compressDataConfigure = &userImage->compressDataConfigure;
    uint64_t baseAddr = (uint64_t)frame->buffer;
    compressDataConfigure->lumaHeadAddr = baseAddr + frame->offset_head_y;
    compressDataConfigure->chromaHeadAddr = baseAddr + frame->offset_head_c;
    compressDataConfigure->lumaPayloadAddr = baseAddr + frame->offset_payload_y;
    compressDataConfigure->chromaPayloadAddr = baseAddr + frame->offset_payload_c;
    compressDataConfigure->lumaHeadStride = frame->stride_head;
    compressDataConfigure->chromaHeadStride = frame->stride_head;
    compressDataConfigure->lumaPayloadStride = frame->stride_payload;
    compressDataConfigure->chromaPayloadStride = frame->stride_payload;

    userImage->yuvSumEnable = false;
    userImage->cmdListBufferAddr = nullptr;
    userImage->cmdListBufferSize = 0;
    // Configure the roi area and output area
    std::shared_ptr<VpcUserRoiConfigure> roiConfigure(new VpcUserRoiConfigure);
    roiConfigure->next = nullptr;
    userImage->roiConfigure = roiConfigure.get();
    VpcUserRoiInputConfigure* roiInput = &roiConfigure->inputConfigure;
    roiInput->cropArea.leftOffset = 0;
    roiInput->cropArea.rightOffset = frame->width - 1;
    roiInput->cropArea.upOffset = 0;
    roiInput->cropArea.downOffset = frame->height - 1;
    VpcUserRoiOutputConfigure* roiOutput = &roiConfigure->outputConfigure;
    roiOutput->outputArea.leftOffset = 0;
```

```
    roiOutput->outputArea.rightOffset = frame->width - 1;
    roiOutput->outputArea.upOffset = 0;
    roiOutput->outputArea.downOffset = frame->height - 1;
    roiOutput->bufferSize = ALIGN_UP(frame->width, 16) * ALIGN_UP(frame->height, 2) * 3 / 2;
    roiOutput->addr = (uint8_t*)HIAI_DVPP_DMalloc(roiOutput->bufferSize);
    if (roiOutput->addr == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VPC_CTL_ERROR, "can not alloc roiOutput buffer");
        DestroyDvppApi(dvppHandle);
        return;
    }
    roiOutput->widthStride = ALIGN_UP(frame->width, 16);
    roiOutput->heightStride = ALIGN_UP(frame->height, 2);

    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = (void*)(userImage.get());
    dvppApiCtlMsg.in_size = sizeof(VpcUserImageConfigure);
    ret = DvppCtl(dvppHandle, DVPP_CTL_VPC_PROC, &dvppApiCtlMsg);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "call dvppctl fail");
        HIAI_DVPP_DFree(roiOutput->addr);
        roiOutput->addr = nullptr;
        DestroyDvppApi(dvppHandle);
        return;
    }
    ret = FrameReturnSaveVpcResult(frame, roiOutput->addr, imageCount);
    if (ret != 0) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "save vpc result fail");
    }
    HIAI_DVPP_DFree(roiOutput->addr);
    roiOutput->addr = nullptr;
    DestroyDvppApi(dvppHandle);
    return;
}

/*
*错误上报回调函数，用户若不需要错误信息，可不定义此函数
*/
void ErrReport(VDECERR* vdecErr)
{
    if (vdecErr != nullptr) {
        HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "vdec error code is %d, channelId = %u\n", vdecErr-
>errType, vdecErr->channelId);
    }
}

//测试函数主入口
void TEST_VDEC()
{
    char outputDir[20] = {0};
    int32_t safeFuncRet = memset_s(outputDir, sizeof(outputDir), 0, 20);
    if (safeFuncRet != EOK) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "memset_s fail");
        return;
    }
    safeFuncRet = strncpy_s(outputDir, sizeof(outputDir), "output_dir", strlen("output_dir"));
    if (safeFuncRet != EOK) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "strncpy_s fail");
        return;
    }
    if (access(outputDir, F_OK) == -1) {
        if (mkdir(outputDir, 0700) < 0) { // directory authority: 0700
            HIAI_ENGINE_LOG(HIAI_VDEC_CTL_ERROR, "create dir failed.");
            return;
        }
    } else if (access(outputDir, R_OK | W_OK | X_OK) == -1) {
        HIAI_ENGINE_LOG(HIAI_VDEC_CTL_ERROR, "output directory authority is not correct, use 700
instead.");
        return;
    }
```

```
IDVPPAPI *pidvppapi = nullptr;
int32_t ret = CreateVdecApi(pidvppapi, 0);
if (ret != 0) {
    HIAI_ENGINE_LOG(HIAI_CREATE_DVPP_ERROR, "create dvpp api fail.");
    return;
}
if (pidvppapi != nullptr) {
    FILE* fp = fopen(g_inFileName, "rb");
    if (fp == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "open file: %s failed.", g_inFileName);
        DestroyVdecApi(pidvppapi, 0);
        return;
    }
    fseek(fp, 0L, SEEK_END);
    int file_size = ftell(fp);
    fseek(fp, 0L, SEEK_SET);
    int rest_len = file_size;
    int len = file_size;

    vdec_in_msg vdec_msg;
    vdec_msg.call_back = FrameReturn;
    vdec_msg.err_report = ErrReport; // can be unassigned, if user does not need error info
    vdec_msg.hiai_data = nullptr;
    int32_t safeFuncRet = 0;
    if (g_format == 0) {
        safeFuncRet = memcpy_s(vdec_msg.video_format, sizeof(vdec_msg.video_format), "h264", 4);
        if (safeFuncRet != EOK) {
            HIAI_ENGINE_LOG(HIAI_ERROR, "memcpy_s fail");
            DestroyVdecApi(pidvppapi, 0);
            fclose(fp);
            fp = nullptr;
            return;
        }
    } else {
        safeFuncRet = memcpy_s(vdec_msg.video_format, sizeof(vdec_msg.video_format), "h265", 4);
        if (safeFuncRet != EOK) {
            HIAI_ENGINE_LOG(HIAI_ERROR, "memcpy_s fail");
            DestroyVdecApi(pidvppapi, 0);
            fclose(fp);
            fp = nullptr;
            return;
        }
    }
    vdec_msg.in_buffer = (char*)malloc(len);
    if (vdec_msg.in_buffer == nullptr) {
        HIAI_ENGINE_LOG(HIAI_ERROR, "malloc fail");
        fclose(fp);
        fp = nullptr;
        DestroyVdecApi(pidvppapi, 0);
        return;
    }

    dvppapi_ctl_msg dvppApiCtlMsg;
    dvppApiCtlMsg.in = (void*)(&vdec_msg);
    dvppApiCtlMsg.in_size = sizeof(vdec_in_msg);

    while (rest_len > 0) {
        int read_len = fread(vdec_msg.in_buffer, 1, len, fp);
        HIAI_ENGINE_LOG("rest_len is %d,read len is %d.", rest_len, read_len);
        vdec_msg.in_buffer_size = read_len;
        fclose(fp);
        fp = nullptr;
        if (VdecCtl(pidvppapi, DVPP_CTL_VDEC_PROC, &dvppApiCtlMsg, 0) != 0) {
            HIAI_ENGINE_LOG(HIAI_VDEC_CTL_ERROR, "call dvppctl process faild!");
            free(vdec_msg.in_buffer);
            vdec_msg.in_buffer = nullptr;
            DestroyVdecApi(pidvppapi, 0);
            return;
        }
```

```
            HIAI_ENGINE_LOG(HIAI_OK, "call vdec process success!");
            rest_len = rest_len - len;
        }
        free(vdec_msg.in_buffer);
        vdec_msg.in_buffer = nullptr;
    } else {
        HIAI_ENGINE_LOG(HIAI_VDEC_CTL_ERROR, "pidvppapi is null!");
    }
    DestroyVdecApi(pidvppapi, 0);
    return;
}
```

8.3.6.6 实现 VENC 功能

若需要将多张图片编码成一个视频，则调用一次**CreateVenc**接口创建实例后，必须使用同一个实例调用**RunVenc**接口进行视频编码，最后再调用一次**DestroyVenc**接口释放实例。

本调用示例是调用VENC接口将yuv图片编码成h265或者h264格式的码流。

```
std::string vencOutFileName("venc.bin");
std::shared_ptr<FILE> vencOutFile(nullptr);
void VencCallBackDumpFile(struct VencOutMsg* vencOutMsg, void* userData)
{
    if (vencOutFile.get() == nullptr) {
        HIAI_ENGINE_LOG(HIAI_VENC_CTL_ERROR, "get venc out file fail!");
        return;
    }
    fwrite(vencOutMsg->outputData, 1, vencOutMsg->outputDataSize, vencOutFile.get());
    fflush(vencOutFile.get());
}

/*
 * venc new interface to achieve venc basic functions.
 */
void TEST_VENC()
{
    std::shared_ptr<FILE> fpIn(fopen(g_inFileName, "rb"), fclose);
    vencOutFile.reset(fopen(vencOutFileName.c_str(), "wb"), fclose);

    if (fpIn.get() == nullptr || vencOutFile.get() == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "open open venc in/out file failed.");
        return;
    }

    fseek(fpIn.get(), 0, SEEK_END);
    uint32_t fileLen = ftell(fpIn.get());
    fseek(fpIn.get(), 0, SEEK_SET);

    struct VencConfig vencConfig;
    vencConfig.width = g_width;
    vencConfig.height = g_high;
    vencConfig.codingType = g_format;
    vencConfig.yuvStoreType = g_yuvStoreType;
    vencConfig.keyFrameInterval = 16;
    vencConfig.vencOutMsgCallBack = VencCallBackDumpFile;
    vencConfig.userData = nullptr;

    int32_t vencHandle = CreateVenc(&vencConfig);
    if (vencHandle == -1) {
        HIAI_ENGINE_LOG(HIAI_VENC_CTL_ERROR, "CreateVenc fail!");
        return;
    }

    // input 16 frames once
    uint32_t inDataLenMaxOnce = g_width * g_high * 3 / 2;
    std::shared_ptr<char> inBuffer(static_cast<char*>(malloc(inDataLenMaxOnce)), free);
```

```
    if (inBuffer.get() == nullptr) {
        HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "alloc input buffer failed");
        DestroyVenc(vencHandle);
        return;
    }

    uint32_t inDataUnhandledLen = fileLen;

    auto start = std::chrono::system_clock::now();
    uint32_t frameCount = 0;

    while (inDataUnhandledLen > 0) {
        uint32_t inDataLen     = inDataUnhandledLen;
        if (inDataUnhandledLen > inDataLenMaxOnce) {
            inDataLen          = inDataLenMaxOnce;
        }
        inDataUnhandledLen    -= inDataLen;
        uint32_t readLen       = fread(inBuffer.get(), 1, inDataLen, fpIn.get());
        if (readLen != inDataLen) {
            HIAI_ENGINE_LOG(HIAI_OPEN_FILE_ERROR, "error in read input file");
            DestroyVenc(vencHandle);
            return;
        }

        struct VencInMsg vencInMsg;
        vencInMsg.inputData = inBuffer.get();
        vencInMsg.inputDataSize = inDataLen;
        vencInMsg.keyFrameInterval = 16;
        vencInMsg.forceIFrame = 0;
        vencInMsg.eos = 0;

        if (RunVenc(vencHandle, &vencInMsg) == -1) {
            HIAI_ENGINE_LOG(HIAI_VENC_CTL_ERROR, "call video encode fail");
            break;
        }
        ++frameCount;
    }

    auto end = std::chrono::system_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    size_t timeCount = static_cast<size_t>(duration.count());
    HIAI_ENGINE_LOG("Total frame count: %u", frameCount);
    HIAI_ENGINE_LOG("Time cost: %lu us", timeCount);
    HIAI_ENGINE_LOG("FPS: %lf", frameCount * 1.0 / (timeCount / 1000000.0));

    DestroyVenc(vencHandle);
    return;
}
```

8.3.7 数据类型

8.3.7.1 VpcUserImageConfigure 中的结构体

该结构体在DDK安装目录下的“ddk/include/inc/dvpp/Vpc.h”文件中定义。

- VpcUserRoiConfigure结构体

成员变量	说明
VpcUserRoiInputConfigure inputConfigure	用户ROI输入配置，详细见**•VpcUserRoiInputConfigur...**。若实现8K缩放功能，不用配置该参数。

成员变量	说明
VpcUserRoiOutputConfigure outputConfigure	用户ROI输出配置，详细见 **•VpcUserRoiOutputConfigu...**。
VpcUserRoiConfigure* next	用户下一个ROI配置，当需要使用一图多框时配置，否则为NULL，默认为NULL。
uint64_t reserve1	预留参数。

- VpcCompressDataConfigure结构体

成员变量	说明
uint64_t lumaHeadAddr	y分量头地址。
uint64_t chromaHeadAddr	uv分量头地址。
uint32_t lumaHeadStride	y分量头stride，与 **FRAME结构体**中的stride_head参数值保持一致。
uint32_t chromaHeadStride	uv分量头stride，与 **FRAME结构体**中的stride_head参数值保持一致。
uint64_t lumaPayloadAddr	y分量数据的地址。
uint64_t chromaPayloadAddr	uv分量数据的地址。
uint32_t lumaPayloadStride	y分量数据的stride，与 **FRAME结构体**中的stride_payload参数值保持一致。
uint32_t chromaPayloadStride	uv分量数据的stride，与 **FRAME结构体**中的stride_payload参数值保持一致。

- VpcUserYuvSum结构体

成员变量	说明
uint32_t ySum	y分量总和。
uint32_t uSum	u分量总和。
uint32_t vSum	v分量总和。
uint64_t reserve1	预留参数。

- VpcUserPerformanceTuningParameter结构体

成员变量	说明
uint64_t reserve1	预留参数1。
uint64_t reserve2	预留参数2。

成员变量	说明
uint64_t reserve3	预留参数3。
uint64_t reserve4	预留参数4。
uint64_t reserve5	预留参数5。

- VpcUserRoiInputConfigure 结构体

成员变量	说明
VpcUserCropConfigure cropArea	用户抠图部分的输入数据配置，详细见**•VpcUserCropConfigure 结构...**。
uint64_t reserve1	预留参数。

- VpcUserRoiOutputConfigure结构体

成员变量	说明
uint8_t* addr	输出图片的首地址。 使用Matrix提供的HIAI_DVPP_DMalloc接口申请内存，申请到的内存满足DVPP的要求（首地址16对齐）。HIAI_DVPP_DMalloc接口的说明，请参见《 Matrix API参考》。
uint32_t bufferSize	输出buffer的大小，根据yuv420sp计算。
uint32_t widthStride	输出图片的宽步长，需要16对齐；宽stride最小为32，最大为4096。
uint32_t heightStride	输出图片的高步长，需要2对齐；高stride最小为6，最大为4096。 输出为yuv420sp图像，需要根据heightStride计算出uv数据的起始地址。
VpcUserCropConfigure outputArea	用户指定输出区域坐标，详细见**•VpcUserCropConfigure 结构...**。 若实现8K缩放功能，不用配置该参数。
uint64_t reserve1	预留参数。

- VpcUserCropConfigure 结构体

  关于上偏移、下偏移、左偏移、右偏移各概念的解释请参见**表8-8**。

成员变量	说明
uint32_t leftOffset	左偏移，必须为偶数。 贴图区域相对输出图片的左偏移16对齐。
uint32_t rightOffset	右偏移，必须为奇数。
uint32_t upOffset	上偏移，必须为偶数。
uint32_t downOffset	下偏移，必须为奇数。
uint64_t reserve1	预留参数。

8.3.7.2 vdec_in_msg 中的结构体和类

该结构体在DDK安装目录下的“ddk/include/inc/dvpp/Vdec.h”文件中定义

- FRAME结构体

成员变量	说明
int height	输出图像的高（对齐后的值，H264为16对齐，H265为64对齐）。
int width	输出图像的宽（对齐后的值，H264为16对齐，H265为64对齐）。
int realHeight	真实图像的高。
int realWidth	真实图像的宽。
unsigned char* buffer	输出图像的内存地址。
int buffer_size	输出图像的内存大小。
unsigned int offset_payload_y	输出图像payload的Y分量偏移量，payload的Y分量地址=buffer + offset_payload_y。
unsigned int offset_payload_c;	输出图像payload的C分量偏移量，payload的C分量地址=buffer + offset_payload_c。
unsigned int offset_head_y;	输出图像head的Y分量偏移量，head的Y分量地址=buffer + offset_head_y。
unsigned int offset_head_c;	输出图像head的C分量偏移量，head的C分量地址=buffer + offset_head_c。
unsigned int stride_payload;	输出图像payload的stride。
unsigned int stride_head;	输出图像head的stride。
unsigned short bitdepth;	输出图像的位深。

成员变量	说明
char video_format[10];	输入视频的格式，为“h264”或“h265”。
char image_format[10];	输出图像的格式，为“nv12”或“nv21”。

- HIAI_DATA_SP类

成员变量或函数	说明
unsigned long long frameIndex	帧序号。
void * frameBuffer	用户申请用于存放输出帧的内存。
unsigned int frameSize	用户申请用于存放输出帧的内存大小。
void setFrameIndex(unsigned long long index)	设置帧序号函数。
unsigned long long getFrameIndex()	获取帧序号函数。
void setFrameBuffer(void * frameBuff)	设置frameBuffer地址。
void * getFrameBuffer()	获取frameBuffer地址。
void setFrameSize(unsigned int size)	设置frameSize大小。
unsigned int getFrameSize()	获取frameSize大小。

- VDECERR结构体

成员变量	说明

ERRTYPE errType	错误类型。
	enum ERRTYPE{
	//VDEC解码器状态异常错误，用户需要销毁解码实例，再重新创建实例
	ERR_INVALID_STATE = 0x10001,
	//硬件错误，包含解码器启动、执行、停止等异常，用户需要销毁解码实例，再重新创建实例
	ERR_HARDWARE,
	//将视频码流分解成多帧图片异常，用户需要检查输入的视频流数据是否正确
	ERR_SCD_CUT_FAIL,
	//解码某一帧异常，用户需要检查输入的视频流数据是否正确
	ERR_VDM_DECODE_FAIL,
	//暂未使用，预留
	ERR_ALLOC_MEM_FAIL,
	//包括输入视频分辨率超范围（用户需要检查输入视频流的分辨率）、内部动态申请内存失败（用户检查系统是否有可用内存）等异常
	ERR_ALLOC_DYNAMIC_MEM_FAIL,
	//系统内部申请VDEC的输入、输出buffer异常，用户检查系统是否有可用内存
	ERR_ALLOC_IN_OR_OUT_PORT_MEM_FAIL,
	//码流错误，暂未使用，预留
	ERR_BITSTREAM,
	//输入视频格式错误，暂未使用，预留
	ERR_VIDEO_FORMAT,
	//输出格式配置错误，暂未使用，预留
	ERR_IMAGE_FORMAT,
	//回调函数为空错误，暂未使用，预留
	ERR_CALLBACK,
	//输入buffer为空错误，暂未使用，预留
	ERR_INPUT_BUFFER,
	//输入buffer大小<=0错误，暂未使用，预留
	ERR_INBUF_SIZE,
	//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，用户需要检查系统

成员变量	说明

	中资源（例如：线程、内存等）是否可用
	ERR_THREAD_CREATE_FBD_FAIL,
	//创建解码实例失败，用户需要重新创建解码实例
	ERR_CREATE_INSTANCE_FAIL,
	//初始化解码器失败，例如解码实例个数超出范围（最大16），用户需要释放部分解码实例后再重新创建实例
	ERR_INIT_DECODER_FAIL,
	//系统内部获取某路视频流的解码句柄失败，用户需要重新创建解码实例
	ERR_GET_CHANNEL_HANDLE_FAIL,
	//系统内部设置解码实例异常，用户需要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_COMPONENT_SET_FAIL,
	//系统内部设置解码实例名称异常，用户需要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,
	//其它错误
	ERR_OTHER
	};

unsigned short channelId	解码错误的通道。

8.3.7.3 IMAGE_CONFIG 中的结构体

- ROI_CONFIG结构体

成员变量	说明	取值范围
CROP_CONFIG：crop_config	裁剪参数配置结构体。	详细见**•CROP_CONFIG结构体**。
YUV_SUM_OUT_CONFIG：sum_out	输出参数配置结构体1。	详见**•YUV_SUM_OUT_CONFIG结构体**。
NORMAL_OUT_CONFIG：normal_out	输出参数配置结构体2(预留，暂时不支持)。	详见**•NORMAL_OUT_CONFIG结构体**。

成员变量	说明	取值范围
ROI_CONFIG*：next	下一个ROI_CONFIG结构体指针。	多框时配置，单框时配置为NULL。

- CROP_CONFIG结构体

成员变量	说明	取值范围
int：enable	是否需要进行裁剪操作。	0：不需要裁剪。 1：需要裁剪。
unsigned int：hmin	水平最小偏移。	偶数，且小于hmax。
unsigned int：hmax	水平最大偏移。	奇数，且小于输入宽度in_width。
unsigned int：vmin	垂直最小偏移。	偶数，且小于vmax。
unsigned int：vmax	垂直最大偏移。	奇数，且小于输入高度in_height。

- YUV_SUM_OUT_CONFIG结构体

成员变量	说明	取值范围
int：enable	是否启用该通道进行输出。	0：不启用。 1：启用。
unsigned int：out_width	输出图像宽度。	偶数。128~4096，缩放系数需满足[0.03125, 4]。
unsigned int：out_height	输出图像高度。	偶数。16~4096，缩放系数需满足[0.03125, 4]。
char*：out_buffer	输出图像内存地址指针。	输出内存大小需要根据输出宽128对齐，高16对齐后的分辨率计算。
unsigned int：yuv_sum	输出图像所有yuv值的和。	预留，暂不支持。

- NORMAL_OUT_CONFIG结构体

成员变量	说明	取值范围
int：enable	是否启用该通道进行输出。	0：不启用。 1：启用。

成员变量	说明	取值范围
unsigned int：out_width	输出图像宽度。	偶数。128~4096，缩放系数需满足[0.03125, 4]。
unsigned int：out_height	输出图像高度。	偶数。16~4096，缩放系数需满足[0.03125, 4]。
char*：out_buffer	输出图像内存地址指针。	输出内存大小需要根据输出宽128对齐，高16对齐后的分辨率计算。

8.3.7.4 dvpp_engine_capability_stru 中的结构体

该结构体在DDK安装目录下的“ddk/include/inc/dvpp/DvppCapability.h”文件中定义

- dvpp_resolution_stru结构体

成员变量	说明	取值范围
uint32_t resolution_high;	高度分辨率。	最大值： VDEC：4096 JPEGD：8192 PNGD：4096 JPEGE：8192 VPC：4096 VENC：1920 最小值： VDEC：128 JPEGD：32 PNGD：32 JPEGE：32 VPC：16 VENC：128

成员变量	说明	取值范围
uint32_t resolution_width;	宽度分辨率。	最大值： VDEC：4096 JPEGD：8192 PNGD：4096 JPEGE：8192 VPC：4096 VENC：1920 最小值： VDEC：128 JPEGD：32 PNGD：32 JPEGE：32 VPC：16 VENC：128

- dvpp_format_unit_stru 结构体

成员变量	说明	取值范围

enum dvpp_color_format color_format;	支持图片格式	*enum* dvpp_color_format {
		//YUV444 in different ordering of YUV Semi-Planar/Packed，8 bit，Linear。
		DVPP_COLOR_YUV444_YUV_P_8BIT_LIN,
		DVPP_COLOR_YUV444_YVU_P_8BIT_LIN,
		DVPP_COLOR_YUV444_UYV_P_8BIT_LIN,
		DVPP_COLOR_YUV444_UVY_P_8BIT_LIN,
		DVPP_COLOR_YUV444_VYU_P_8BIT_LIN,
		DVPP_COLOR_YUV444_VUY_P_8BIT_LIN,
		DVPP_COLOR_YUV444_UV_SP_8BIT_LIN,
		DVPP_COLOR_YUV444_VU_SP_8BIT_LIN,
		/*422*/
		DVPP_COLOR_YUYV422_YUYV_P_8BIT_LIN,
		DVPP_COLOR_YUYV422_YVYU_P_8BIT_LIN,
		DVPP_COLOR_YUYV422_UYVY_P_8BIT_LIN,
		DVPP_COLOR_YUYV422_VYUY_P_8BIT_LIN,
		DVPP_COLOR_YUV422_UV_SP_8BIT_LIN,
		DVPP_COLOR_YUV422_VU_SP_8BIT_LIN,
		/*420*/
		DVPP_COLOR_YUV420_SP_8BIT_LIN,
		DVPP_COLOR_YVU420_SP_8BIT_LIN,
		DVPP_COLOR_YUV420_SP_8BIT_HFBC,
		DVPP_COLOR_YVU420_SP_8BIT_HFBC,
		DVPP_COLOR_YUV420_SP_10BIT_HFBC,
		DVPP_COLOR_YVU420_SP_10BIT_HFBC,
		DVPP_COLOR_YUV420_P_8BIT_LIN,

成员变量	说明	取值范围
		/*400*/ DVPP_COLOR_YVU400_SP_8BIT, /*rgb888*/ DVPP_COLOR_RGB888_RGB_P_8BIT_LIN, DVPP_COLOR_RGB888_RBG_P_8BIT_LIN, DVPP_COLOR_RGB888_GBR_P_8BIT_LIN, DVPP_COLOR_RGB888_GRB_P_8BIT_LIN, DVPP_COLOR_RGB888_BRG_P_8BIT_LIN, DVPP_COLOR_RGB888_BGR_P_8BIT_LIN, /*argb888*/ DVPP_COLOR_ARGB8888_ARGB_P_8BIT_LIN, DVPP_COLOR_ARGB8888_ARBG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_AGBR_P_8BIT_LIN, DVPP_COLOR_ARGB8888_AGRB_P_8BIT_LIN, DVPP_COLOR_ARGB8888_ABRG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_ABGR_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RAGB_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RABG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RGBA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RGAB_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RBAG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_RBGA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BRGA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BRAG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BGAR_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BGRA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BAR

成员变量	说明	取值范围
		G_P_8BIT_LIN, DVPP_COLOR_ARGB8888_BAGR_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GRAG_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GRBA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GABR_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GARB_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GBRA_P_8BIT_LIN, DVPP_COLOR_ARGB8888_GBAR_P_8BIT_LIN, PIC_JPEG, PIC_PNG, VIO_H265, VIO_H264 };

uint32_t compress_type;	压缩类型	enum dvpp_compress_type {
		arithmetic_code =0,
		huffman_code
		};

uint32_t stride_size;	步长大小	VDEC：128 JPEGD：128 PNGD：128 JPEGE：0 VPC：128 VENC：0

enum dvpp_high_align_type high_alignment;	高度对齐类型	enum dvpp_high_align_type {
		pix_random = 0,
		two_pix_alignment = 2,
		four_pix_alignment = 4,
		eight_pix_alignment = 8,
		sixteen_pix_alignment = 16
		};

版权所有 © 华为技术有限公司

成员变量	说明	取值范围

enum dvpp_high_align_type width_alignment;	宽度对齐类型	enum dvpp_high_align_type {
		pix_random = 0,
		two_pix_alignment = 2,
		four_pix_alignment = 4,
		eight_pix_alignment = 8,
		sixteen_pix_alignment = 16
		};

uint32_t out_mem_alignment;	输出内存对齐参数	-

- dvpp_perfomance_unit_stru结构体

成员变量	说明	取值范围
uint32_t resolution_high;	高度分辨率	VDEC：1920 JPEGD：1920 PNGD：1920 JPEGE：1920 VPC：3840 VENC：1920
uint32_t resolution_width;	宽度分辨率	VDEC：1080 JPEGD：1080 PNGD：1080 JPEGE：1080 VPC：2160 VENC：1080
uint32_t stream_num;	流大小	VDEC：16 JPEGD：0 PNGD：0 JPEGE：0 VPC：0 VENC：1
unsigned long fps;	帧率	VDEC：30 JPEGD：256 PNGD：24 JPEGE：64 VPC：90 VENC：30

版权所有 © 华为技术有限公司

- dvpp_pre_contraction_stru结构体

成员变量	说明	取值范围

enum dvpp_support_type is_support;	预缩小是否支持	VPC：support
		others：no support
		其中，support取值范围如下：
		enum dvpp_support_type {
		no_support =0, //no support
		do_support //support
		};

uint32_t contraction_types;	缩小类型	VPC：3 others：0
uint32_t contraction_size[DVPP_PRE_CONTRATION_TYPE_MAX];	预缩小固定比例	VPC：2/4/8 others：0
enum dvpp_support_type is_horizontal_support;	是否支持水平方向预缩小	VPC：support others：no support
enum dvpp_support_type is_vertical_support;	是否支持垂直方向预缩小	VPC：support others：no support

- dvpp_pos_scale_stru结构体

成员变量	说明	取值范围
enum dvpp_support_type is_support;	后缩放是否支持	VPC：support others：no support
uint32_t min_scale;	最小缩放系数	VPC：1 others：1
uint32_t max_scale;	最大缩放系数	VPC：4 others：1
enum dvpp_support_type is_horizontal_support;	是否支持水平方向后缩放	VPC：support others：no support
enum dvpp_support_type is_vertical_support;	是否支持垂直方向后缩放	VPC：support others：no support

- dvpp_vpc_data_spec_stru结构体

成员变量	说明	取值范围
uint32_t input_type;	输入格式所属类型	包括以下两种： ● 0：表示HFBC类型 ● 1：表示YUV或RGB类型
struct dvpp_resolution_stru min_resolution;	最小分辨率。	请参见**•struct dvpp_resolution_...**
struct dvpp_resolution_stru max_resolution;	最大分辨率	请参见 **struct dvpp_resolution_...**

enum dvpp_align_type high_alignment;	高度对齐类型。	enum dvpp_high_align_type {
		//1像素对齐
		pix_random = 0,
		//2像素对齐
		two_pix_alignment = 2,
		//4像素对齐
		four_pix_alignment = 4,
		//8像素对齐
		eight_pix_alignment = 8,
		//16像素对齐
		sixteen_pix_alignment = 16
		};

版权所有 © 华为技术有限公司

成员变量	说明	取值范围

enum dvpp_align_type width_alignment;	宽度对齐类型。	enum dvpp_high_align_type {
		//1像素对齐
		pix_random = 0,
		//2像素对齐
		two_pix_alignment = 2,
		//4像素对齐
		four_pix_alignment = 4,
		//8像素对齐
		eight_pix_alignment = 8,
		//16像素对齐
		sixteen_pix_alignment = 16
		};

8.3.7.5 vpc_in_msg 中的结构体

8.3.7.5.1 Rdma 通道结构体 RDMACHANNEL

成员变量	说明	取值范围
Long luma_head_addr	Y分量的头地址。	-
Long chroma_head_addr	UV分量的头地址。	-
unsigned int luma_head_stride	Y分量的头stride。	-
nsigned int chroma_head_stride;	UV分量的头stride。	-
long luma_payload_addr	Y分量数据的地址。	-
long chroma_payload_addr	UV分量数据的地址。	-
unsigned int luma_payload_stride	Y分量数据的头stride。	-
unsigned int chroma_payload_stride;	UV分量数据的头stride。	-

8.3.7.5.2 Vpc 内部优化结构体 VpcTurningReverse

成员变量	说明	取值范围
unsigned int reverse1	内部优化预留接口1。	--
unsigned int reverse2	内部优化预留接口2。	--
unsigned int reverse3	内部优化预留接口3。	--
unsigned int reverse4	内部优化预留接口4。	--

8.3.8 附录

8.3.8.1 辅助功能接口

- JpegCalBackFree类辅助接口

class JpegCalBackFree：

JpegCalBackFree() :go

~JpegCalBackFree()

JpegCalBackFree(JpegCalBackFree& others)

void setBuf(void* addr4Free)

void setBuf(void* addr4Free, size_t size4Free)

void operator() ()

const JpegCalBackFree& operator= (JpegCalBackFree& others)

- AutoBuffer类辅助接口

class AutoBuffer：

AutoBuffer()

~AutoBuffer()

void Reset()

char* allocBuffer(int size)

char* getBuffer()

int getBufferSize()

AutoBuffer(const AutoBuffer&)

const AutoBuffer& operator= (const AutoBuffer&)

- HIAI_DATA_SP类辅助接口

class HIAI_DATA_SP：

HIAI_DATA_SP()

virtual ~HIAI_DATA_SP()

void setFrameIndex(unsigned long long index)

unsigned long long getFrameIndex()

void setFrameBuffer(void * frameBuff)

void * getFrameBuffer()

void setFrameSize(unsigned int size)

unsigned int getFrameSize()

8.3.8.2 不推荐使用接口列表

- class IDVPPCAPABILITY：

virtual ~IDVPPCAPABILITY(void)

virtual int process(dvppapi_ctl_msg *MSG) = 0

virtual int init() = 0

virtual int start() = 0

virtual int stop() = 0

- class IJPEGDAPI：

virtual ~IJPEGDAPI(void)

virtual int process(dvppapi_ctl_msg* MSG) = 0

virtual int init() = 0

virtual int start() = 0

virtual int stop() = 0

- class IJPEGEAPI：

virtual ~IJPEGEAPI(){}

virtual int process(dvppapi_ctl_msg* MSG) = 0

virtual int init() = 0

virtual int start() = 0

virtual int stop() = 0

- class IVDECAPI：

virtual ~IVDECAPI(void)

virtual int process(dvppapi_ctl_msg* MSG) = 0

virtual int init() = 0

virtual int start() = 0

virtual int stop() = 0

- class IVENCAPI：

virtual ~IVENCAPI(void)

virtual int process(dvppapi_ctl_msg* MSG) = 0

virtual int init() = 0

virtual int start() = 0

- class IVPCAPI：

virtual ~IVPCAPI(void)

virtual int process(dvppapi_ctl_msg* MSG) = 0

virtual int init() = 0

virtual int start() = 0

virtual int stop() = 0

8.3.8.3 DVPP 执行器样例使用说明

DVPP执行器工具集成了DVPP的六个基础功能，包括VPC、JPEGE、JPEGD、VENC、VDEC以及PNGD。

8.3.8.3.1 环境准备

步骤1 将编译生成的可执行文件“sample_dvpp_hlt_ddk”复制到Device侧任意路径。

说明

依赖SO为： libc_sec.so， libDvpp_api.so， libDvpp_jpeg_decoder.so， libDvpp_jpeg_encoder.so， libDvpp_png_decoder.so， libDvpp_vpc.so， libOMX_hisi_video_decoder.so， libOMX_hisi_video_encoder.so， libslog.so

这些SO默认已存在deivce侧的“/usr/lib64”目录，用户无需单独拷贝。

步骤2 输入数据文件放入到可执行文件所在路径下，运行如下命令，其中，**paramList**是8.3.8.3.2 DVPP执行器样例入参说明中介绍的参数。

./sample_dvpp_hlt_ddk paramList。

----结束

8.3.8.3.2 DVPP 执行器样例入参说明

参数	描述
img_width	输入图片的宽度。
img_height	输入图片的高度。

参数	描述
in_format	● 对于VPC，输入图片的格式如下： INPUT_YUV400, // 0 INPUT_YUV420_SEMI_PLANNER_UV, // 1 INPUT_YUV420_SEMI_PLANNER_VU, // 2 INPUT_YUV422_SEMI_PLANNER_UV, // 3 INPUT_YUV422_SEMI_PLANNER_VU, // 4 INPUT_YUV444_SEMI_PLANNER_UV, // 5 INPUT_YUV444_SEMI_PLANNER_VU, // 6 INPUT_YUV422_PACKED_YUYV, // 7 INPUT_YUV422_PACKED_UYVY, // 8 INPUT_YUV422_PACKED_YVYU, // 9 INPUT_YUV422_PACKED_VYUY, // 10 INPUT_YUV444_PACKED_YUV, // 11 INPUT_RGB, // 12，输入图像格式为RGB888 INPUT_BGR, // 13，输入图像格式为BGR888 INPUT_ARGB, // 14，此格式的输入图像在存储时各分量的排列顺序类似RGB888，其中，A表示透明度 INPUT_ABGR, // 15，此格式的输入图像在存储时各分量的排列顺序类似BGR888，其中，A表示透明度 INPUT_RGBA, // 16，此格式的输入图像在存储时各分量的排列顺序类似RGB888，其中，A表示透明度 INPUT_BGRA, // 17，此格式的输入图像在存储时各分量的排列顺序类似BGR888，其中，A表示透明度 INPUT_YUV420_SEMI_PLANNER_UV_10BIT, // 18 INPUT_YUV420_SEMI_PLANNER_VU_10BIT, // 19 ● 对于VDEC，输入码流的格式如下： – 0：h264格式的码流 – 1：h265格式的码流 ● 对于VENC，输出码流的编码格式如下： – 0：H265 main level（仅支持Slice码流） – 1：H264 baseline level – 2：H264 main level – 3：H264 high level ● 对于JPEGE，输入图片的格式如下： – 0：JPGENC_FORMAT_UYVY – 1：JPGENC_FORMAT_VYUY – 2：JPGENC_FORMAT_YVYU – 3：JPGENC_FORMAT_YUYV – 16：JPGENC_FORMAT_NV12

参数	描述
	– 17：JPGENC_FORMAT_NV21
out_format	VPC输出图片格式如下： ● 0：OUTPUT_YUV420SP_UV ● 1：OUTPUT_YUV420SP_VU
yuvStoreType	VENC输入yuv数据的存储格式如下： ● 0：YUV420 semi-planner ● 1：YVU420 semi-planner
inWidthStride	VPC输入图像宽度方向的步长。详细见**入参：VpcUserImageConfigure**中widthStride的描述。
inHighStride	VPC输入图像高度方向的步长。详细见**入参：VpcUserImageConfigure**中heightStride的描述。
outWidthStride	VPC输出图片的宽步长，需要16对齐。
outHighStride	VPC输出图片的高步长，需要2对齐；
hmin	VPC用户指定抠图区域左偏移，必须为偶数。
hmax	VPC用户指定抠图区域右偏移，必须为奇数。
vmin	VPC用户指定抠图区域上偏移，必须为偶数。
vmax	VPC用户指定抠图区域下偏移，必须为奇数。
outLeftOffset	VPC用户指定输出区域左偏移，必须为偶数。
outRightOffset	VPC用户指定输出区域右偏移，必须为奇数。
outUpOffset	VPC用户指定输出区域上偏移，必须为偶数。
outDownOffset	VPC用户指定输出区域下偏移，必须为奇数。
out_width	输出图像宽度。
out_high	输出图像高度。
in_image_file	输入图像文件的路径，包含文件名。
out_image_file	输出图像文件的路径，包含文件名。
rank	● 0：JPEGD输出yuv420 semi-planar数据； ● 1：JPEGD输出yuv原格式数据。
transform	PNGD处理图像时的转换标志： ● 1：表示RGBA转换到RGB； ● 0：保留原格式。

版权所有 © 华为技术有限公司

参数	描述
test_type	输入数据：1、2、3、4、5、6、7、8、9、10、12。 ● 1：VPC基础功能1（抠图、缩放）。 ● 2：VPC基础功能2（原图缩放、抠图缩放、一图多框和8k图像缩放）。 ● 3：VDEC。 ● 4：VENC。 ● 5：JPEGE。 ● 6：JPEGE用户指定输出内存。 ● 7：JPEGD。 ● 8：JPEGD用户指定输出内存。 ● 9：PNGD。 ● 10：PNGD用户指定输出内存。 ● 12：JPEGD+VPC+JPEGE。

8.3.8.3.3 VPC 使用说明

?.1. VPC 基础功能 1

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_vpc_1920x1080_nv12.yuv --out_image_file vpc_out.yuv --inWidthStride 1920 --inHighStride 1080 --in_format 1 --out_format 0 --hmax 1919 --hmin 0 --vmax 999 --vmin 0 --outLeftOffset 0 --outRightOffset 719 --outUpOffset 0 --outDownOffset 719 --outWidthStride 720 --outHighStride 720 --test_type 1
```

示例描述：调用VPC实现图像的裁剪、缩放。

- 输入图像：
  - 宽1920像素，高1080像素、格式为yuv420sp、名称为"dvpp_vpc_1920x1080_nv12.yuv”的图像。
  - 与可执行文件放在同一目录下。
- 输出图像：
  - 宽720像素、高720像素、格式为yuv420sp（nv12）、名称为“vpc_out.yuv”的图像
  - 生成的图像与可执行文件在同一目录下。
- 用户可以使用图像显示软件打开vpc_out.yuv图像进行正确性验证。

?.2. VPC 基础功能 2

```
./sample_dvpp_hlt_ddk --test_type 2
```

示例描述：本命令实现4个功能，调用VPC实现原图缩放、抠图缩放、一图多框和8k图像缩放功能。

- 输入图像：
  - 宽1920像素、高1080像素、格式为yuv420sp nv12、名称为“dvpp_vpc_1920x1080_nv12.yuv”的图像。

  - 宽8192像素、高8192像素、格式为yuv420sp nv12、名称为“dvpp_vpc_8192x8192_nv12.yuv”的图像。
  - 与可执行文件放在同一目录下。
- 输出图像：
  - NewVpcTest1函数实现1080p yuv420sp图像缩小至1280*720 yuv420sp（nv12），输出文件名为“NewVpcTest1Out.yuv”。
  - NewVpcTest2函数实现1080p yuv420sp图像中抠一张子图，并贴图至输出图片的指定位置，输出文件名为“NewVpcTest2Out.yuv”，该文件像素是1280*720，格式是yuv420sp。
  - NewVpcTest3函数实现1080p yuv420sp图像中抠五张子图，并贴图至输出图片的指定位置，输出文件名为“NewVpcTest3_*Out.yuv”。该文件像素是1280*720，格式是yuv420sp。
  - NewVpcTest4函数实现8192*8192的 yuv420sp图像缩放至4000*4000的yuv420sp图像，输出文件名为“NewVpcTest4Out.yuv”。
  - 生成的图像与可执行文件在同一目录下。
- 用户可以使用图像显示软件打开图像进行正确性验证。

8.3.8.3.4 VDEC 使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_vdec_h264_1frame_bp_51_1920x1080.h264 --out_image_file h264_image --in_format 0 --test_type 3
./sample_dvpp_hlt_ddk --in_image_file dvpp_vdec_h265_1frame_mp_51_1920x1080.h265 --out_image_file h265_image --in_format 1 --test_type 3
```

示例描述：本命令用于调用VDEC解码*.h264和*.h265格式的码流，并调用VPC对解码后图像进行解压缩成yuv格式图像。

- 输入视频：
  - 宽1920像素、高1080像素、名称为“dvpp_vdec_h264_1frame_bp_51_1920x1080.h264”的码流。
  - 宽1920像素、高1080像素、名称为“dvpp_vdec_h265_1frame_mp_51_1920x1080.h265”的码流。
  - 与可执行文件放在同一目录下。
- 输出图像：
  - 宽1920像素、高1080像素、格式为YUV420sp的图像。
  - 针对h264码流的解码，在“output_dir”目录下生成名称为“h264_image*”的图片，当前视频中只有一帧，所以只输出一张图像，如果视频中有多帧，则输出多张图像。
  - 针对h265码流的解码，在“output_dir”目录下生成名称为“h265_image*”的图片，当前视频中只有一帧，所以只输出一张图像，如果视频中有多帧，则输出多张图像。
- 用户可以使用图像显示软件打开“h264_image*”、“h265_image*”图像进行正确性验证。

8.3.8.3.5 VENC 使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_venc_128x128_nv12.yuv --img_width 128 --img_height 128 --in_format 0 --yuvStoreType 0 --test_type 4
```

示例描述：本命令用于调用VENC将yuv图片编码成h265格式的码流。

- 输入图像：
  - 宽128像素、高128像素、格式为yuv420sp nv12、名称为“dvpp_venc_128x128_nv12.yuv”的图像。
  - 与可执行文件放在同一目录下。
  - in_format对应到**入参：VencConfig**中coding_type。
  - yuvStoreType对应到**入参：VencConfig**中YUV_store_type。
- 输出视频：
  - 输出名称为“venc.bin”的码流文件。
  - 与可执行文件放在同一目录下。
- 用户可以使用ffmpeg解码venc.bin文件，使用命令“ffmpeg.exe -i venc.bin venc.jpg”解码，然后使用图像显示软件打开图像进行正确性校验。ffmpeg工具的官网路径为：http://ffmpeg.org/

8.3.8.3.6 JPEGE 使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_jpege_444_1920x1080_1920_1080_nv12.yuv --img_width 1920 --img_height 1080 --in_format 16 --test_type 5
./sample_dvpp_hlt_ddk --in_image_file dvpp_jpege_444_1920x1080_1920_1080_nv12.yuv --img_width 1920 --img_height 1080 --in_format 16 --test_type 6
```

示例描述：分别使用dvpp内部申请输出内存（test_type 5）和客户申请输出内存（test_type 6）两种调用方式，将yuv格式图像编码成jpg图像。

- 输入图像：
  - 宽1920像素、高1080像素、格式为yuv420sp nv12、名称为“dvpp_jpege_444_1920x1080_1920_1080_nv12.yuv”的图像。
  - 与可执行文件放在同一目录下。
- 输出图像：
  - 宽1920像素、高1080像素、输出名称为“outfile_t*.jpg”（*为线程id）的图像。
  - 与可执行文件放在同一目录下。

8.3.8.3.7 JPEGD 使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_jpegd_decode_1920x1080.jpg --test_type 7 --rank 0
./sample_dvpp_hlt_ddk --in_image_file dvpp_jpegd_decode_1920x1080.jpg --test_type 8 --rank 0
```

示例描述：分别使用dvpp内部申请输出内存（test_type 7）和客户申请输出内存（test_type 8）两种调用方式，将jpg图像解码成yuv图像。

- 输入图像：
  - 宽1920像素、高1080像素、名称为“dvpp_jpegd_decode_1920x1080.jpg”的图像。
  - 与可执行文件放在同一目录下。
  - -- rank为0或不设置rank参数，**输出NV21格式数据**。-- rank不为0输出原格式的半平面数据。
- 输出图像：
  - 宽1920像素、高1080像素、输出名称为“dvppapi_jpegd_wxx_hxx_fx_tx.yuv”的图像。
  - 与可执行文件放在同一目录下。

- 用户可以使用图像显示软件打开输出的yuv图像进行正确性验证。

8.3.8.3.8 PNGD 使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_pngd_1920x1080_RGBA.png --out_image_file
pngd_out_RGBAtoRGB_1.rgb --test_type 9  --transform 1
./sample_dvpp_hlt_ddk --in_image_file dvpp_pngd_1920x1080_RGBA.png --out_image_file
pngd_out_RGBAtoRGB_2.rgb --test_type 10 --transform 1
```

示例描述：分别使用dvpp内部申请输出内存（test_type 9）和客户申请输出内存（test_type 10）两种调用方式，将png图像解码成rgb888格式图像。

- 输入图像：
  - 宽1920像素、高1080像素、名称为“dvpp_pngd_1920x1080_RGBA.png”的图像。
  - 与可执行文件放在同一目录下。
  - 参数transform为1时，表示将rgba格式的png图片转成rgb888格式输出。
- 输出图像：
  - 宽1920像素、高1080像素、分别输出名称为“pngd_out_RGBAtoRGB_1.rgb”（test_type 9）和“pngd_out_RGBAtoRGB_2.rgb”（test_type 10）的图像。
  - 与可执行文件放在同一目录下。
- 用户可以使用图像显示软件打开pngd_out_RGBAtoRGB_1.rgb和pngd_out_RGBAtoRGB_2.rgb图像进行正确性验证。

8.3.8.3.9 JPEGD+VPC+JPEGE 串联使用说明

```
./sample_dvpp_hlt_ddk --in_image_file dvpp_jpegd_vpc_jpege.jpg --out_image_file
out_jpegd_vpc_jpege_new.jpg --hmax 1919 --hmin 0 --vmax 999 --vmin 0 --outLeftOffset 0 --
outRightOffset 719 --outUpOffset 0 --outDownOffset 719 --outWidthStride 720 --outHighStride 720 --
test_type 12
```

示例描述：JPEGD+VPC+JPEGE三个·模块串联使用的调用实例。其中中test_type 12使用的是客户申请输出内存的调用方式。此命令的功能是调用JPEGD将一张jpg图片进行解码生成yuv420 sp图片，送入VPC进行裁剪、缩放，然后将VPC的输出送给JPEGE进行编码，输出jpg图片。

- 输入图像：
  - 宽1920像素、高1080像素、名称为“dvpp_jpegd_vpc_jpege.jpg”的图像。
  - 与可执行文件放在同一目录下。
- 输出图像：
  - 宽720像素、高720像素、名称为“out_jpegd_vpc_jpege_new.jpg”的图像。
  - 与可执行文件放在同一目录下。

8.3.8.4 异常处理

当调用方出现调用**DvppCtl**或者**DvppGetOutParameter**失败时，也即调用该函数返回值为-1时，可通过Mind Studio界面的Log窗口查看日志，在**ModuleName**参数处选择DVPP，然后单击Search查询日志。根据Time列的时间查看最新日志，并根据日志的提示排查异常调用错误。

示例：调用方在使用VPC功能时，输入图片宽高不是128x16对齐，则在Content列的日志内容处提示如下信息。

```
The input image width should be divided by 128, high should be divided by 16, now width 512, high 456.
```

说明

关于日志查看的详细操作，可参见《Ascend 310 Mind Studio开发辅助工具》中的“日志工具 > 基本操作 > 日志查看”章节。

8.3.8.5 缩略语和术语一览

表 8-28 缩略语和术语

缩略语	英文全名
VDEC	video decoder
VENC	video encoder
JPEGD	JPEG decoder
JPEGE	JPEG encoder
PNGD	PNG decoder
VPC	vision preprocessing core
DVPP	digital vision pre-processing
HFBC	Hisi Frame Buffer Compress

8.4 GE API 参考

8.4.1 简介

Graph Engine（GE）提供了安全易用的构图接口集合，用户可以调用这些接口构建网络模型，设置模型所包含的图、图内的算子、以及模型和算子的属性。该集合包含如下几个模块：

- 接口类型
  - Operator

    Operator表达了基础的算子类型，提供了算子的属性存取接口、输入/输出的设置接口。
  - Graph

    Graph表达了由多个Operator通过连接关系构建起来图。该模块提供了整图Input/Output的设置以及图的有效性校验接口。
  - Model

    Model表达了由一个Graph构成的模型。该模块提供了模型版本号、模型属性的存取接口，以及模型内部的Graph的存取接口。
- 辅助（数据）类型
  - AttrValue

- NamedAttr
  - 属性的实例，按照key-value的格式存储。
- AttrHolder
  - 用于存放Operator的属性值。

– Tensor

- TensorDesc：用于描述Tensor的一些信息，包括：
  - Shape：Tensor的形状的描述。
  - DataType：Tensor所存储的数据的类型。
  - Format：Tensor的format信息。

您可以在DDK包的安装目录下的“ddk/include/inc/graph”目录下查看接口的定义文件。如果通过引导安装的方式同时安装Mind Studio和DDK，您可以使用Mind Studio安装用户登录Mind Studio服务器，在“~/tools/che/ddk/ddk/include/inc/graph”路径下查看接口定义文件。每个接口对应的定义文件请参见具体的接口描述。

8.4.2 属性类接口

该类接口在attr_value.h、detail/attributes_holder.h中定义。

8.4.2.1 SetAttr

函数原型

graphStatus SetAttr(const string& name, const AttrValue& value);

功能说明

设置属性值。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称
value	输入	const AttrValue&	属性值

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	成功返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.2.2 GetAttr

函数原型

graphStatus GetAttr(const string& name, AttrValue& value) const;

功能说明

获取属性值。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称
value	输出	AttrValue&	属性值

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	成功返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.2.3 HasAttr

函数原型

bool HasAttr(const string& name) const;

功能说明

判断是否有对应名称的属性

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	true：有名称为name的属性 false：无名称为name的属性

异常处理

无。

约束说明

无。

8.4.2.4 DelAttr

函数原型

graphStatus DelAttr(const string& name);

功能说明

删除属性。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	成功返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.2.5 SetName

函数原型

void SetName(const std::string& name);

功能说明

设置属性的名称。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称

返回值

无。

异常处理

无。

约束说明

无。

8.4.2.6 GetName

函数原型

string GetName() const;

功能说明

获取属性的名称。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	string	属性名称

异常处理

无。

约束说明

无。

8.4.2.7 GetItem

函数原型

AttrValue GetItem(const string& key) const;

功能说明

获取对应名称的属性值。

参数说明

参数名	输入/输出	类型	描述
key	输入	const string&	属性名称

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AttrValue	获取到的属性值

异常处理

无。

约束说明

无。

8.4.2.8 GetValueType

函数原型

ValueType GetValueType() const;

功能说明

获取当前属性值的类型；

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	ValueType	返回对应的属性类型

异常处理

无。

约束说明

无。

8.4.2.9 IsEmpty

函数原型

bool IsEmpty() const;

功能说明

判断属性值是否设置。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	false：未设置值 true：已设置属性值

异常处理

无。

约束说明

无。

8.4.2.10 Copy

函数原型

AttrValue Copy() const;

功能说明

复制属性值，返回一个属性对象。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	AttrValue	属性对象

异常处理

无。

约束说明

无。

8.4.2.11 MutableTensor

函数原型

graphStatus MutableTensor(TensorPtr& tensor);

功能说明

获取可以修改的tensor引用。

参数说明

参数名	输入/输出	类型	描述
tensor	输出	TensorPtr&	tensor引用。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	成功返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.2.12 MutableListTensor

函数原型

graphStatus MutableListTensor(vector<TensorPtr>& list_tensor);

功能说明

获取可以修改的Tensor列表的引用。

参数说明

参数名	输入/输出	类型	描述
list_tensor	输出	vector<TensorPtr>&	Tensor列表的引用

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	成功返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.2.13 InitDefault

函数原型

void InitDefault();

功能说明

初始化默认值。

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无

8.4.3 缓存类接口

该类接口在buffer.h中定义。

8.4.3.1 ClearBuffer

函数原型

void ClearBuffer();

功能说明

清空Buffer, 将保存数据的Buffer指针置为null。

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无

8.4.3.2 GetData

函数原型

```
std::uint8_t* GetData();
const std::uint8_t* GetData() const;
```

功能说明

获取Buffer中存放的数据的指针。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	uint8_t*	返回Buffer中存放的数据的指针。

异常处理

异常场景	说明
Buffer为空	返回空指针

约束说明

无。

8.4.3.3 GetSize

函数原型

std::size_t GetSize() const;

功能说明

获取Buffer中存放的数据的大小。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	返回Buffer中存放的数据的大小。

异常处理

异常场景	说明
Buffer为空	返回数据大小为0。

约束说明

无。

8.4.3.4 CopyFrom

函数原型

static Buffer CopyFrom(std::uint8_t* data, std::size_t bufferSize);

功能说明

根据data中指针所指向的地址，拷贝bufferSize大小的数据，并将数据地址存放到Buffer中。

参数说明

参数名	输入/输出	类型	描述
data	输入	uint8_t*	存放要拷贝的数据指针。
bufferSize	输入	size_t	拷贝数据的大小。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Buffer	用于存放拷贝数据的Buffer对象。

异常处理

无。

约束说明

无。

8.4.4 Graph 类接口

该类接口在graph.h中定义。

8.4.4.1 SetInputs

函数原型

```
Graph& SetInputs(std::vector<Operator>& inputs);
```

功能说明

设置Graph内的输入算子。

参数说明

参数名	输入/输出	类型	描述
inputs	输入	std::vector<Operator>&	Graph内的输入算子。

返回值

参数名	类型	描述
-	Graph&	返回调用者本身

异常处理

无。

约束说明

无。

8.4.4.2 SetOutputs

函数原型

Graph& SetOutputs(std::vector<Operator>& outputs);

功能说明

设置Graph关联的输出算子。

参数说明

参数名	输入/输出	类型	描述
outputs	输入	std::vector<Operator>&	与Graph关联的输出算子。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Graph&	返回调用者本身

异常处理

无。

约束说明

无。

8.4.4.3 IsValid

函数原型

bool IsValid() const;

功能说明

判断Graph对象是否有效。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	● true，构建Graph对象有效，非空。 ● false，构建Graph对象无效，空指针。

异常处理

无。

约束说明

无。

8.4.4.4 AddOp

函数原型

graphStatus AddOp(ge::Operator& op);

功能说明

在Graph中增加算子。

参数说明

参数名	输入/输出	类型	描述
op	输入	ge::Operator&	需增加的算子

返回值

参数名	类型	描述
-	graphStatus	GRAPH_SUCCESS：操作成功 GRAPH_FAILED： 操作失败

异常处理

无。

约束说明

无。

8.4.4.5 FindOpByName

函数原型

ge::Operator FindOpByName(const string& name) const;

功能说明

根据算子的名称，返回Graph中的算子实例。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	需指定的算子名称。

返回值

参数名	类型	描述
-	ge::Operator	如果Graph中包含对应名称算子，返回Graph中的这个算子；否则返回type为“NULL”的算子。

异常处理

无。

约束说明

无。

8.4.4.6 CheckOpByName

函数原型

graphStatus CheckOpByName(const string& name) const;

功能说明

在Graph中检查指定名称的算子是否存在。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	需指定的算子名称。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	GRAPH_SUCCESS： 指定名称的算子存在 GRAPH_FAILED：指定名称的算子不存在

异常处理

无。

约束说明

无。

8.4.4.7 GetAllOpName

函数原型

graphStatus GetAllOpName(std::vector<string>& opName) const ;

功能说明

返回Graph中所有算子的名称。

版权所有 © 华为技术有限公司

参数说明

参数名	输入/输出	类型	描述
opName	输出	std::vector<string>&	返回Graph中的所有算子的名称。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	GRAPH_SUCCESS：操作成功 GRAPH_FAILED： 操作失败

异常处理

无。

约束说明

无。

8.4.5 Model 类接口

8.4.5.1 SetName

函数原型

void SetName(const string& name);

功能说明

设置模型的名称。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	模型名称。

返回值

无。

异常处理

无。

约束说明

无。

8.4.5.2 GetName

函数原型

string GetName() const;

功能说明

获取模型的名称。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	string	返回模型的名称。

异常处理

无。

约束说明

无。

8.4.5.3 SetVersion

函数原型

void SetVersion(uint32_t version)

功能说明

设置模型的版本号。

参数说明

参数名	类型	描述（参数说明、取值范围等）
version	uint32_t	模型的版本号。

返回值

无。

异常处理

无。

约束说明

无。

8.4.5.4 GetVersion

函数原型

uint32_t GetVersion() const;

功能说明

获取模型的版本号。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	uint32_t	返回模型的版本号。

异常处理

无。

约束说明

无。

8.4.5.5 SetPlatformVersion

函数原型

void SetPlatformVersion(string version)

功能说明

设置用户自定义模型的版本号。

参数说明

参数名	类型	描述（参数说明、取值范围等）
version	string	用户自定义模型的版本号。

返回值

无。

异常处理

无。

约束说明

无。

8.4.5.6 GetPlatformVersion

函数原型

std::string GetPlatformVersion() const;

功能说明

获取用户自定义模型的版本号，版本号是常量。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	string	模型版本号。

异常处理

无。

约束说明

无。

8.4.5.7 GetGraph

函数原型

Graph GetGraph() const;

功能说明

获取模型中的Graph对象。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Graph	模型中的Graph对象。

异常处理

无。

约束说明

无。

8.4.5.8 SetGraph

函数原型

void SetGraph(const Graph& graph);

功能说明

设置模型的Graph对象。

参数说明

参数名	输入/输出	类型	描述
graph	输入	const Graph&	graph 对象。

返回值

无。

异常处理

无。

约束说明

无。

8.4.5.9 Save

函数原型

graphStatus Save(Buffer& buffer) const;

功能说明

将模型对象序列化。

参数说明

参数名	输入/输出	类型	描述
buffer	输入	Buffer&	序列化输出的对象的引用。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	序列化成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.5.10 Load

函数原型

static graphStatus Load(const uint8_t* data, size_t len, Model& model);

功能说明

加载序列化数据，反序列化构建模型对象。

参数说明

参数名	输入/输出	类型	描述
data	输入	const uint8_t *	序列化数据指针。
len	输入	size_t	序列化数据长度。
model	输出	Model &	承载反序列化后的模型对象。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	执行成功，返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.5.11 IsValid

函数原型

bool IsValid() const;

功能说明

判断模型中Graph对象是否有效，如果Graph对象为空指针，则无效。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	Model中Graph对象是否有效，true表示有效，false表示无效。

异常处理

无。

约束说明

无。

8.4.6 Operator 类接口

该类接口在operator.h中定义。

8.4.6.1 Operator 构造函数

函数原型

Operator();

功能说明

Operator构造函数。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Operator	算子对象。

异常处理

无。

约束说明

无。

8.4.6.2 GetName

函数原型

string GetName() const;

功能说明

获取算子名称。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	string	算子名称。

异常处理

无。

约束说明

无。

8.4.6.3 SetInput（srcOprt 只存在一个输出）

函数原型

Operator& SetInput(const string& dstName, const Operator& srcOprt);

功能说明

设置当前算子的Input。

参数说明

参数名	输入/输出	类型	描述
dstName	输入	const string&	当前算子的Input名称。

参数名	输入/输出	类型	描述
srcOprt	输入	const Operator&	Input名称为dstName的输入算子对象。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Operator&	当前调度者本身。

异常处理

无。

约束说明

要求Operator& srcOprt只存在一个输出。

8.4.6.4 SetInput（srcOprt 存在多个输出）

函数原型

Operator& SetInput(const string& dstName, const Operator& srcOprt, const string &name);

功能说明

如果srcOprt存在多个输出，则根据srcOprt的Output名称设置当前算子的Input。

参数说明

参数名	输入/输出	类型	描述
dstName	输入	const string&	当前算子的Input名称。
srcOprt	输入	const Operator&	Input名称为dstName的输入算子对象。
name	输入	const string&	srcOprt的Ouput名称。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Operator&	当前调度者本身。

异常处理

无。

约束说明

无。

8.4.6.5 GetInputDesc

函数原型

TensorDesc GetInputDesc(const string& name) const;

TensorDesc GetInputDesc(uint32_t index) const;

功能说明

根据算子Input名称或Input索引获取Input的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子Input名称。 无算子Input名称，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。
index	输入	uint32_t	算子Input索引。 无算子Input索引，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述（参数说明、取值范围等）
TensorDesc	TensorDesc	算子Input的TensorDesc。

异常处理

无。

约束说明

无。

8.4.6.6 TryGetInputDesc

函数原型

bool TryGetInputDesc(const string& name, TensorDesc& tensorDesc) const;

功能说明

根据算子Input名称获取Input的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	Input名称
tensorDesc	输出	TensorDesc&	TensorDesc对象

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	获取TensorDesc成功， 返回true，否则，返回false。

异常处理

无。

约束说明

无。

8.4.6.7 UpdateInputDesc

函数原型

graphStatus UpdateInputDesc(const string& name, const TensorDesc& tensorDesc);

功能说明

根据算子Input名称更新Input的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子Input名称
tensorDesc	输入	const TensorDesc&	TensorDesc对象

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.6.8 GetOutputDesc

函数原型

TensorDesc GetOutputDesc(const string& name) const;

TensorDesc GetOutputDesc(uint32_t index) const;

功能说明

根据算子Output名称或Output索引获取Output的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子Output名称。
index	输入	uint32_t	算子Output索引。 无算子Output索引，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	TensorDesc	获取TensorDesc成功，返回对应TensorDesc， 否则，返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

8.4.6.9 UpdateOutputDesc

函数原型

graphStatus UpdateOutputDesc (const string& name, const TensorDesc& tensorDesc);

功能说明

根据算子Output名称更新Output的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子Output名称。
tensorDesc	输入	const TensorDesc&	TensorDesc对象

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	更新TensorDesc成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.6.10 GetDynamicInputDesc

函数原型

TensorDesc GetDynamicInputDesc(const string& name, const unsigned int index) const;

功能说明

根据name和index的组合获取算子动态Input的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子动态Input的名称
index	输入	const unsigned int	算子动态Input编号，编号从1开始

返回值

参数名	类型	描述（参数说明、取值范围等）
TensorDesc	TensorDesc	获取TensorDesc成功，则返回算子动态Input的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

8.4.6.11 UpdateDynamicInputDesc

函数原型

graphStatus UpdateDynamicInputDesc(const string& name, const unsigned int index, const TensorDesc& tensorDesc);

功能说明

根据name和index的组合更新动态Input的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子动态Input的名称
index	输入	const unsigned int	算子动态Input编号，编号从1开始
tensorDesc	输入	const TensorDesc&	TensorDesc对象

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	更新动态Input成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.6.12 GetDynamicOutputDesc

函数原型

TensorDesc GetDynamicOutputDesc (const string& name, const unsigned int index) const;

功能说明

根据name和index的组合获取算子动态Output的TensorDesc；

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子动态Output的名称
index	输入	const unsigned int	算子动态Output编号，编号从1开始

返回值

参数名	类型	描述（参数说明、取值范围等）
TensorDesc	TensorDesc	获取TensorDesc成功，则返回算子动态Output的TensorDesc；获取失败，则返回TensorDesc默认构造的对象，其中，主要设置DataType为DT_FLOAT（表示float类型），Format为FORMAT_NCHW（表示NCHW）。

异常处理

无。

约束说明

无。

8.4.6.13 UpdateDynamicOutputDesc

函数原型

graphStatus UpdateDynamicOutputDesc (const string& name, const unsigned int index, const TensorDesc& tensorDesc);

功能说明

根据name和index的组合更新算子动态Output的TensorDesc。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	算子动态Output的名称
index	输入	const unsigned int	算子动态Output编号
tensorDesc	输入	const TensorDesc&	TensorDesc对象

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	更新动态Output成功，返回GRAPH_SUCCESS， 否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.6.14 SetAttr

函数原型

Operator& SetAttr(const string& name, AttrValue&& attrValue);

Operator& SetAttr(const string& name, const UsrQuantizeFactorParams& attrValue);

功能说明

设置算子属性的属性值。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称。
attrValue	输入	AttrValue&&	需设置的属性值。
attrValue	输入	const UsrQuantizeFactorParams&	量化因子参数。 关于UsrQuantizeFactorParams，请参见**数据类型说明**。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Operator&	对象本身

异常处理

无。

约束说明

无。

数据类型说明

- UsrQuantizeFactorParams结构体：

```
struct UsrQuantizeFactorParams
{
    uint32_t quantize_algo{0};
    uint32_t scale_type{0};
    UsrQuantizeFactor quantize_param;
    UsrQuantizeFactor dequantize_param;
    UsrQuantizeFactor requantize_param;
```

```
    UsrQuantizeCalcFactor quantizecalc_param;
};
```

- UsrQuantizeFactor结构体：

```
struct UsrQuantizeFactor
{
    //QuantizeScaleMode scale_mode;
    uint32_t scale_mode{0};
    std::vector<uint8_t> scale_value;
    int64_t scale_offset{0};
    std::vector<uint8_t> offset_data_value;
    int64_t offset_data_offset{0};
    std::vector<uint8_t> offset_weight_value;
    int64_t offset_weight_offset{0};
    std::vector<uint8_t> offset_pad_value;
    int64_t offset_pad_offset{0};
};
```

- UsrQuantizeCalcFactor结构体：

```
struct UsrQuantizeCalcFactor
{
    std::vector<uint8_t> offsetw;
    int64_t offsetw_offset{0};
    std::vector<uint8_t> offsetd;
    int64_t offsetd_offset{0};
    std::vector<uint8_t> scalereq;
    int64_t scaledreq_offset{0};
    std::vector<uint8_t> offsetdnext;
    int64_t offsetdnext_offset{0};
};
```

8.4.6.15 GetAttr

函数原型

graphStatus GetAttr(const string& name, AttrValue& attrValue) const;

graphStatus GetAttr(const string& name, UsrQuantizeFactorParams& attrValue) const;

功能说明

根据属性名称获取其属性值。

参数说明

参数名	输入/输出	类型	描述
name	输入	const string&	属性名称。
attrValue	输入	AttrValue&	属性值。
attrValue	输入	UsrQuantizeFactorParams&	量化因子参数。 关于UsrQuantizeFactorParams，请参见**数据类型说明**。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	获取属性值成功，返回GRAPH_SUCCESS，否则返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.7 Shape 类接口

该类接口在tensor.h中定义。

8.4.7.1 GetDimNum

函数原型

size_t GetDimNum() const;

功能说明

获取Shape的维度个数。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	size_t	Tensor Shape的维度个数。

异常处理

无。

约束说明

无。

8.4.7.2 GetDim

函数原型

```
int64_t GetDim(size_t idx) const;
```

功能说明

获取Shape第idx维的长度。

参数说明

参数名	输入/输出	类型	描述
idx	输入	size_t	维度索引，索引从0开始。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	int64_t	第idx维的长度

异常处理

无

约束说明

无。

8.4.7.3 GetDims

函数原型

```
std::vector<int64_t> GetDims() const;
```

功能说明

获取Shape所有维度组成的向量。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	std::vector<int64_t>	Shape的所有维度组成的向量。

异常处理

无。

约束说明

无。

8.4.7.4 SetDim

函数原型

graphStatus SetDim(size_t idx, int64_t value);

功能说明

将Shape中第idx维度的值设置为value。

参数说明

参数名	输入/输出	类型	描述
idx	输入	size_t	Shape维度的索引，索引从0开始。
value	输入	int64_t	需设置的值。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.7.5 GetShapeSize

函数原型

int64_t GetShapeSize() const;

功能说明

获取Shape中所有dim的累乘结果。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	int64_t	返回所有dim的累乘结果。

异常处理

无。

约束说明

无。

8.4.8 Tensor 类接口

该类接口在tensor.h中定义。

8.4.8.1 GetTensorDesc

函数原型

TensorDesc GetTensorDesc() const;

功能说明

获取Tensor的描述符（TensorDesc）。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	TensorDesc	返回当前Tensor的描述符。

异常处理

无。

约束说明

无。

8.4.8.2 MutableTensorDesc

函数原型

TensorDesc& MutableTensorDesc();

功能说明

获取当前Tensor的描述符，可修改。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	TensorDesc&	当前Tensor的描述符，可修改。

异常处理

无。

约束说明

无。

8.4.8.3 SetTensorDesc

函数原型

graphStatus SetTensorDesc(const TensorDesc &tensorDesc);

功能说明

设置Tensor的描述符。

参数说明

参数名	输入/输出	类型	描述
tensorDesc	输入	const TensorDesc &	需设置的Tensor描述符。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.8.4 GetData

函数原型

const Buffer GetData() const;

功能说明

获取Tensor中的数据。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	const Buffer	Tensor中所存放的数据。

异常处理

无。

约束说明

无。

8.4.8.5 MutableData

函数原型

Buffer MutableData();

功能说明

获取Tensor中存放的数据。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Buffer	Tensor中存放的数据。

异常处理

无。

约束说明

无。

8.4.8.6 SetData

函数原型

```
graphStatus SetData(std::vector<uint8_t> &&data);
graphStatus SetData(const std::vector<uint8_t> &data);
graphStatus SetData(const Buffer &data);
graphStatus SetData(const uint8_t *data, size_t size);
```

功能说明

向Tensor中设置数据。

参数说明

参数名	输入/输出	类型	描述
data	输入	std::vector<uint8_t> && 或const std::vector<uint8_t> & 或const Buffer & 或const uint8_t *	需设置的数据。
size	输入	size_t	数据的长度，单位为字节。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	设置成功返回GRAPH_SUCCESS，否则，返回GRAPH_FAILED。

异常处理

无。

约束说明

无。

8.4.8.7 Clone

函数原型

Tensor Clone() const;

功能说明

拷贝Tensor。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Tensor	返回拷贝的Tensor对象。

异常处理

无。

约束说明

无。

8.4.9 TensorDesc 类接口

该类接口在tensor.h中定义。

8.4.9.1 Update

函数原型

void Update(Shape shape, Format format = FORMAT_NCHW, DataType dt = DT_FLOAT);

功能说明

更新TensorDesc对象的shape、format、datatype属性。

参数说明

参数名	输入/输出	类型	描述
shape	输入	Shape	需刷新的shape对象。
format	输入	Format	需刷新的format对象，默认取值FORMAT_NCHW。 关于Format数据类型的定义，请参见•**Format**。
dt	输入	DataType	需刷新的datatype对象，默认取值DT_FLOAT。 关于DataType数据类型的定义，请参见•**DataType**。

返回值

无。

异常处理

无。

约束说明

无。

数据类型说明

- Format

```
enum Format {
        FORMAT_NCHW = 0,          /**< NCHW */
        FORMAT_NHWC,              /**< NHWC */
        FORMAT_ND,                /**< Nd Tensor */
        FORMAT_NC1HWC0,           /**< NC1HWC0 */
        FORMAT_FRACTAL_Z,         /**< FRACTAL_Z */
        FORMAT_NC1C0HWPAD,
        FORMAT_NHWC1C0,
        FORMAT_FSR_NCHW,
        FORMAT_FRACTAL_DECONV,
        FORMAT_C1HWNC0,
        FORMAT_FRACTAL_DECONV_TRANSPOSE,
        FORMAT_FRACTAL_DECONV_SP_STRIDE_TRANS,
        FORMAT_NC1HWC0_C04,  /**< NC1HWC0, C0 =4*/
        FORMAT_FRACTAL_Z_C04,/**< FRACZ格式，C0 =4 */
        FORMAT_CHWN,
        FORMAT_FRACTAL_DECONV_SP_STRIDE8_TRANS,
        FORMAT_HWCN,
        FORMAT_NC1KHKWHWC0, /** < KH,KW kernel h& kernel w maxpooling max output format*/
        FORMAT_BN_WEIGHT,
        FORMAT_FILTER_HWCK,      /* filter input tensor format */
        FORMAT_HASHTABLE_LOOKUP_LOOKUPS=20,
        FORMAT_HASHTABLE_LOOKUP_KEYS,
        FORMAT_HASHTABLE_LOOKUP_VALUE,
        FORMAT_HASHTABLE_LOOKUP_OUTPUT,
        FORMAT_HASHTABLE_LOOKUP_HITS=24,
        FORMAT_RESERVED

    };
```

- DataType

```
enum DataType {
        DT_UNDEFINED = 16,   // Used to indicate a DataType field has not been set.
        DT_FLOAT = 0,           // float type
        DT_FLOAT16 = 1,          // fp16 type
        DT_INT8 = 2,            // int8 type
        DT_INT16 = 6,     // int16 type
        DT_UINT16 = 7,      // uint16 type
        DT_UINT8 = 4,           // uint8 type
        DT_INT32 = 3,           //
        DT_INT64 = 9,           // int64 type
        DT_UINT32 = 8,          // unsigned int32
        DT_UINT64 = 10,          // unsigned int64
        DT_BOOL = 12,           // bool type
        DT_DOUBLE = 11,          // double type
        DT_DUAL = 13,            /**< dual output type */
        DT_DUAL_SUB_INT8 = 14,    /**< dual output int8 type */
        DT_DUAL_SUB_UINT8 = 15,    /**< dual output uint8 type */
    };
```

8.4.9.2 GetShape

函数原型

Shape GetShape() const;

功能说明

获取TensorDesc描述的shape。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Shape	TensorDesc描述的shape。

异常处理

无。

约束说明

返回的shape为const，不可修改。

8.4.9.3 MutableShape

函数原型

Shape& MutableShape();

功能说明

获取TensorDesc中可修改的shape引用。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Shape &	返回TensorDesc中的shape引用，可修改。

异常处理

无。

约束说明

无。

8.4.9.4 SetShape

函数原型

void SetShape(Shape shape);

功能说明

设置TensorDesc的shape。

参数说明

参数名	输入/输出	类型	描述
shape	输入	Shape	需向TensorDesc设置的shape对象

返回值

无。

异常处理

无。

约束说明

无。

8.4.9.5 GetFormat

函数原型

Format GetFormat() const;

功能说明

获取TensorDesc所描述的Tensor的format信息。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	Format	TensorDesc所描述的Tensor的format信息。

异常处理

无。

约束说明

无。

8.4.9.6 SetFormat

函数原型

void SetFormat(Format format);

功能说明

设置TensorDesc所描述的Tensor的format信息。

参数说明

参数名	输入/输出	类型	描述
format	输入	Format	需设置的format信息。

返回值

无。

异常处理

无。

约束说明

无。

8.4.9.7 GetDataType

函数原型

DataType GetDataType() const;

功能说明

获取TensorDesc所描述的Tensor的数据类型。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	DataType	TensorDesc所描述的Tensor的数据类型。

异常处理

无。

约束说明

无。

8.4.9.8 SetDataType

函数原型

void SetDataType(DataType dt);

功能说明

设置TensorDesc所描述的Tensor的数据类型。

参数说明

参数名	输入/输出	类型	描述
dt	输入	DataType	需设置的dt信息。 关于DataType，请参见**•DataType**。

返回值

无。

异常处理

无。

约束说明

无。

8.4.9.9 Clone

函数原型

TensorDesc Clone() const;

功能说明

拷贝TensorDesc。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	TensorDesc	拷贝的TensorDesc对象。

异常处理

无。

约束说明

无。

8.4.9.10 IsValid

函数原型

graphStatus IsValid();

功能说明

判断tensor对象是否有效。

参数说明

无。

返回值

参数名	类型	描述（参数说明、取值范围等）
-	graphStatus	tensor对象是否有效，有效返回GRAPH_SUCCESS，否则返回GRAPH_FAILED

异常处理

无。

约束说明

无。

8.4.10 Operator 注册类

注册算子类型以REG_OP为起始，以“.”链接INPUT、OUTPUT、ATTR等接口注册算子的输入、输出和属性信息，最终以OP_END接口结束。注册算子类型成功后，自动生成以算子类型名称命名的类。

例如：

```
REG_OP(FullConnection)
.INPUT(x, TensorType::ALL())
.INPUT(w, TensorType::ALL())
.INPUT(b, TensorType::ALL())
.OUTPUT(y, TensorType::ALL())
.ATTR(num_output, AttrValue::INT{0})
.INFER_SHAPE_AND_TYPE(FullConnectionInfer)
.ATTR_ALL_VERIFY(FullConnectionVerify)
.OP_END()
```

Operator注册类接口在operator_reg.h中定义。已注册的算子及对应的头文件，请参见8.4.10.12 内置的算子类型列表。

8.4.10.1 REG_OP

函数原型

```
REG_OP(x)
```

功能说明

注册算子类型，同时自动生成算子类型的两个构造函数。

例如，注册算子的类型名称Conv2D，可调用REG_OP(Conv2D)接口，调用该接口后，定义了算子的类型名称Conv2D，同时产生Conv2D的两个构造函数，其中，**Conv2D(const string& name)**需指定算子名称，Conv2D()使用默认算子名称，例如“Conv2D*唯一编号*”。

```
class Conv2D : public Operator {
    typedef Conv2D _THIS_TYPE;
public:
    explicit Conv2D(const string& name);
    explicit Conv2D();
}
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，被注册算子的类型名称

返回值

无。

异常处理

无。

约束说明

注册的算子类型名称需保持唯一，不能重复。

8.4.10.2 ATTR

函数原型

ATTR(x, default_value)

功能说明

注册算子属性，必须指定默认值，用户不设置算子对象的属性值时使用默认值。

注册算子属性成功后，自动生成算子属性的3个对外接口，用于获取属性的名称、获取属性的值、设置属性的值。

现以注册类型为int64_t的属性、类型为int64_t列表两种场景为例，说明所生成的算子属性接口：

- 调用ATTR(mode, AttrValue::INT{1})接口，注册属性mode，属性类型为int64_t，默认值为1。

注册属性成功后，自动生成以下接口：

```
static const string name_attr_mode(); // 返回属性的名称，即“mode”
int64_t get_attr_mode() const; // 返回mode属性的值
_THIS_TYPE& set_attr_mode(int64_t v); // 设置mode属性的值，返回算子对象本身
```

- 调用ATTR(pad, AttrValue::LIST_INT{0, 0, 0, 0})接口，注册属性pad，属性类型为int64_t列表，默认值为{0,0,0,0}。

  注册属性成功后，自动生成以下接口：

```
static const string name_attr_pad(); // 返回属性的名称，即“pad”
vector<int64_t> get_attr_pad() const; ; // 返回属性pad的值
_THIS_TYPE& set_attr_pad(vector<int64_t> v); // 设置属性pad的值，返回算子对象本身
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子属性的名称。
default_value	输入	-	算子属性的值，根据不同类型指定默认值，支持的属性类型包括： ● AttrValue::INT，属性类型为int64_t ● AttrValue::FLOAT, 属性类型为float ● AttrValue::STR，属性类型为string ● AttrValue::BOOL，属性类型为bool ● AttrValue::TENSOR，属性类型为Tensor ● AttrValue::LIST_INT，属性类型为vector<int64_t>, int64_t列表 ● AttrValue::LIST_FLOAT, 属性类型为vector<float>, float列表 ● AttrValue::LIST_STR，属性类型为vector<string>，string列表 ● AttrValue::LIST_BOOL，属性类型为vector<bool>，bool列表 ● AttrValue::LIST_TENSOR，属性类型为vector<Tensor>，Tensor列表

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子属性名称需保持唯一，不能重复。

8.4.10.3 REQUIRED_ATTR

函数原型

REQUIRED_ATTR (x, type)

功能说明

注册算子属性，没有默认值，用户必须设置算子对象的属性值。

注册算子属性成功后，自动生成算子属性的3个对外接口，用于获取属性的名称、获取属性的值、设置属性的值。

例如，注册类型为int64_t的属性mode，可调用**REQUIRED_ATTR (mode, Int)**接口，注册算子属性成功后，会自动生成如下接口：

```
static const string name_attr_mode(); // 返回属性的名称，即“mode”
OpInt get_attr_mode() const; // 返回mode属性的值，OpInt即int64_t
_THIS_TYPE& set_attr_mode(const OpInt& v); // 设置mode属性的值，返回this对象
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子属性的名称。

参数名	输入/输出	类型	描述
type	输入	-	算子属性的类型，包括： ● AttrValue::INT，属性类型为int64_t ● AttrValue::FLOAT, 属性类型为float ● AttrValue::STR，属性类型为string ● AttrValue::BOOL，属性类型为bool ● AttrValue::TENSOR，属性类型为Tensor ● AttrValue::LIST_INT，属性类型为vector<int64_t>, int64_t列表 ● AttrValue::LIST_FLOAT, 属性类型为vector<float>, float列表 ● AttrValue::LIST_STR，属性类型为vector<string>，string列表 ● AttrValue::LIST_BOOL，属性类型为vector<bool>，bool列表 ● AttrValue::LIST_TENSOR，属性类型为vector<Tensor>，Tensor列表

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子属性名称需保持唯一，不能重复。

8.4.10.4 INPUT

函数原型

INPUT (x, t)

功能说明

注册算子输入信息。

注册算子输入信息成功后，自动生成算子输入的相关接口，用于获取算子输入的名称、设置算子输入的对应描述等。

例如，注册算子输入x，算子输入接收的数据类型为TensorType{DT_FLOAT}，可调用**INPUT(x, TensorType{DT_FLOAT})**接口，注册算子输入成功后，自动生成以下相关接口：

```
static const string name_in_x(); // 返回输入的名称，即“x”
_THIS_TYPE& set_input_x(Operator& v, const string& srcName);// 指定输入x与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE& set_input_x(Operator& v); // 指定输入x与算子对象v的索引0的输出存在连接关系，返回算子对象本身
TensorDesc get_input_desc_x(); // 返回输入x对应的描述
graphStatus update_input_desc_x(const TensorDesc& tensorDesc);// 设置输入x对应的描述，包括Shape、DataType、Format等信息，graphStatus即uint32_t类型，返回非0表示出错
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子输入的名称
t	输入	-	算子输入接收的数据类型，可以是TensorType定义的一个或多个，如果多个，通过“,”隔离，例如： TensorType{DT_FLOAT} TensorType({DT_FLOAT, DT_INT8} 关于TensorType类，请参见**TensorType类说明**。

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子输入名称需保持唯一，不能重复。

TensorType 类说明

TensorType类用以定义输入或者输出支持的数据类型，TensorType提供以下接口指定支持的数据类型：

- TensorType(DataType dt)：指定仅支持一个数据类型；
- TensorType(std::initializer_list<DataType> types)：指定支持多个数据类型；
- static TensorType ALL()：指定支持所有数据类型；

- static TensorType FLOAT()：指定支持DT_FLOAT和DT_FLOAT16数据类型。

8.4.10.5 OPTIONAL_INPUT

函数原型

OPTIONAL_INPUT(x, t)

功能说明

注册可选算子输入信息。

注册可选算子输入信息成功后，自动生成算子输入的相关接口，用于获取算子输入的名称、设置算子输入的对应描述等。

例如，注册算子输入b，算子输入接收的数据类型为TensorType{DT_FLOAT}，可调用**OPTIONAL_INPUT(b, TensorType{DT_FLOAT})**接口，注册算子输入成功后，自动生成以下相关接口：

```
static const string name_in_b(); // 返回输入的名称，即“b”
_THIS_TYPE& set_input_b(Operator& v, const string& srcName);// 指定输入b与算子对象v的输出srcName存在连接关系，返回算子对象本身
_THIS_TYPE& set_input_b(Operator& v); // 指定输入b与算子对象v的索引0的输出存在连接关系，返回算子对象本身
TensorDesc get_input_desc_b(); // 返回输入b对应的描述
graphStatus update_input_desc_b(const TensorDesc& tensorDesc);// 设置输入b对应的描述，包括Shape、DataType、Format等信息
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子输入的名称。
t	输入	-	算子输入接收的数据类型，可以是TensorType定义的一个或多个，如果多个，通过“,”隔离，例如： TensorType{DT_FLOAT} TensorType({DT_FLOAT, DT_INT8}

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子输入名称需保持唯一，不能重复。

8.4.10.6 DYNAMIC_INPUT

函数原型

DYNAMIC_INPUT (x, t)

功能说明

注册动态算子输入信息。

注册动态算子输入信息成功后，自动生成算子输入的相关接口，用于创建动态输入、设置算子输入的对应描述等。

例如，注册动态算子输入d，算子输入接收的数据类型为TensorType{DT_FLOAT}，可调用**DYNAMIC_INPUT(d, TensorType{DT_FLOAT})**接口，注册动态算子输入成功后，自动生成以下相关接口：

```
_THIS_TYPE& create_dynamic_input_d(unsigned int num); // 创建动态输入d，包括num个输入
TensorDesc get_dynamic_input_desc_d(unsigned int index);// 返回动态输入d第index个描述，包括Shape、DataType、Format等信息
graphStatus update_dynamic_input_desc_d(unsigned int index, const TensorDesc& tensorDesc);// 更新动态输入d的第index个描述
_THIS_TYPE& set_dynamic_input_d(unsigned int dstIndex, Operator &v); // 指定输入d的第dstIndex个输入与算子对象v的索引0的输出存在连接关系，返回算子对象本身
_THIS_TYPE& set_dynamic_input_d(unsigned int dstIndex, Operator &v, const string &srcName); / /指定动态输入d的第dstIndex个输入与算子对象v的输出srcName存在连接关系，返回算子对象本身
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子输入的名称
t	输入	-	算子输入接收的数据类型，可以是TensorType定义的一个或多个，如果多个，通过“,”隔离，例如： TensorType{DT_FLOAT} TensorType({DT_FLOAT, DT_INT8}

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子输入名称需保持唯一，不能重复。

8.4.10.7 OUTPUT

函数原型

OUTPUT (x, t)

功能说明

注册算子输出信息。

注册算子输出信息成功后，自动生成算子输出的相关接口，用户获取算子输出的名称、获取算子输出的描述、设置算子输出的描述。

例如，注册算子输出y，算子输出接收的数据类型为TensorType{DT_FLOAT}，可调用**OUTPUT(y, TensorType{DT_FLOAT})**接口，注册算子输出成功后，自动生成以下相关接口

```
static const string name_out_y();// 返回输出的名称，即“y”
TensorDesc get_output_desc_y();// 返回输出y对应的描述
graphStatus update_output_desc_y(const TensorDesc& tensorDesc); );// 设置输出y对应的描述，包括Shape、DataType、Format等信息
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子输出的名称。
t	输入	-	算子输出接收的数据类型，可以是TensorType定义的一个或多个，如果多个，通过“,”隔离，例如： TensorType{DT_FLOAT} TensorType({DT_FLOAT, DT_INT8}

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子输出名称需保持唯一，不能重复。

8.4.10.8 DYNAMIC_OUTPUT

函数原型

DYNAMIC_OUTPUT (x, t)

功能说明

注册动态算子输出信息。

注册动态算子输出信息成功后，自动生成动态算子输出的相关接口，包括用于创建动态输出、设置算子输出的对应描述等

例如，注册动态算子输出d，算子输出接收的数据类型为TensorType{DT_FLOAT}，可调用**DYNAMIC_OUTPUT (d, TensorType{DT_FLOAT})**接口，注册动态算子输出成功后，自动生成以下相关接口

```
_THIS_TYPE& create_dynamic_output_d(unsigned int num); // 创建动态输出d，包括num个输出
TensorDesc get_dynamic_output_desc_d(unsigned int index);// 返回动态输出d第index个描述，包括Shape、DataType、Format等信息
graphStatus update_dynamic_output_desc_d(unsigned int index, const TensorDesc& tensorDesc);// 更新动态输出d的第index个描述
```

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子输出的名称。
t	输入	-	算子输出接收的数据类型，可以是TensorType定义的一个或多个，如果多个，通过“,”隔离，例如： TensorType{DT_FLOAT} TensorType({DT_FLOAT, DT_INT8}

返回值

无。

异常处理

无。

约束说明

对于同一个算子，注册的算子输出名称需保持唯一，不能重复。

8.4.10.9 INFER_SHAPE_AND_TYPE

函数原型

INFER_SHAPE_AND_TYPE (x)

功能说明

注册用于推理算子的Shape和DataType的函数。

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子Shape和DataType推理函数。 例如，**INFER_SHAPE_AND_TYPE(FullConnectionInfer)**用于注册**FullConnectionInfer**函数，用于推理算子的Shape和DataType。 FullConnectionInfer通过DECLARE_INFERFUNC声明，通过IMPLEMT_INFERFUNC定义，详情请参见**DECLARE_INFERFUNC和IMPLEMT_INFERFUNC宏说明**。

返回值

无。

异常处理

无。

约束说明

无。

DECLARE_INFERFUNC 和 IMPLEMT_INFERFUNC 宏说明

在注册用于推理算子的Shape和DataType的函数前，需要先用DECLARE_INFERFUNC宏申明函数、用IMPLEMT_INFERFUNC宏定义函数。

- 声明函数
  ```
  DECLARE_INFERFUNC(FullConnection, FullConnectionInfer)
  ```
  DECLARE_INFERFUNC宏展开后的实现为：
  ```
  namespace op {
  class FullConnection;
  }
  static graphStatus FullConnectionInfer(op::FullConnection& op);
  ```
- 定义函数
  ```
  IMPLEMT_INFERFUNC(FullConnection, FullConnectionInfer) {
  // 实现细节
  }
  ```
  IMPLEMT_INFERFUNC宏展开后的实现为：
  ```
  static graphStatus FullConnectionInfer(op::FullConnection& op){
  // 实现细节
  }
  ```

8.4.10.10 ATTR_ALL_VERIFY

函数原型

ATTR_ALL_VERIFY (x)

功能说明

注册算子校验函数。

参数说明

参数名	输入/输出	类型	描述
x	输入	-	宏参数，算子校验函数。 例如，**ATTR_ALL_VERIFY(FullConnectionVerify)**用于注册算子校验函数为**FullConnectionVerify**。FullConnectionVerify通过DECLARE_VERIFIER声明，通过IMPLEMT_VERIFIER定义，详情请参见**DECLARE_VERIFIER和IMPLEMT_VERIFIER宏说明**。

返回值

无。

异常处理

无。

约束说明

无。

DECLARE_VERIFIER 和 IMPLEMT_VERIFIER 宏说明

在注册算子校验函数前，需要先用DECLARE_VERIFIER宏申明函数、用IMPLEMT_VERIFIER宏定义函数。

- 声明函数
  ```
  DECLARE_VERIFIER(FullConnection, FullConnectionVerify)
  ```
  DECLARE_VERIFIER宏展开后的实现为：
  ```
  namespace op {
  class FullConnection;
  }
  static graphStatus FullConnectionVerify(op::FullConnection op);
  ```
- 定义函数

```
IMPLEMT_VERIFIER(FullConnection, FullConnectionVerify) {
// 实现细节
}
```

IMPLEMT_VERIFIER宏展开后的实现为：

```
static graphStatus FullConnectionVerify(op::FullConnection op){
// 实现细节
}
```

8.4.10.11 OP_END

函数原型

OP_END ()

功能说明

结束算子注册。

参数说明

无。

返回值

无。

异常处理

无。

约束说明

无。

8.4.10.12 内置的算子类型列表

您可以在注册算子类型的头文件中查看内置的算子类型，注册各算子类型时设置的算子输入、输出、属性等信息。

表 8-29 算子列表

算子名称	所在头文件
Data	array_defs.h
Concat	
Flatten	
Reshape	
Split	
Const	const_defs.h

算子名称	所在头文件
Permute	detection_defs.h
Add	math_defs.h
Mul	
Activation	nn_defs.h
BatchNorm	
Convolution	
Eltwise	
LRN	
ConvolutionDepthwise	
FullConnection	
Pooling	
Scale	
ShuffleChannel	
Softmax	

8.4.11 模型构建类接口

模型构建时的接口调用顺序如下：

CreateModelBuff-->BuildIRModel-->ReleaseModelBuff

8.4.11.1 CreateModelBuff

函数原型

bool CreateModelBuff(ge::Model& irModel,ModelBufferData& output);

功能说明

创建模型Buffer。

参数说明

参数名	输入/输出	类型	描述
irModel	输入	ge::Model&	模型对象。

参数名	输入/输出	类型	描述

output	输出	ModelBufferData&	离线模型结构体对象。
			struct ModelBufferData
			{
			void* data;
			uint32_t length;
			};

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	● true，创建模型Buffer成功。 ● false，创建模型Buffer失败。

异常处理

无。

约束说明

无。

8.4.11.2 BuildIRModel

函数原型

bool BuildIRModel(ge::Model& irModel,ModelBufferData& output);

功能说明

离线模型构建接口，输入模型对象，输出离线模型。

参数说明

参数名	输入/输出	类型	描述
irModel	输入	ge::Model&	模型对象。

参数名	输入/输出	类型	描述

output	输出	ModelBufferData&	离线模型结构体对象。
			struct ModelBufferData
			{
			void* data;
			uint32_t length;
			};

返回值

参数名	类型	描述（参数说明、取值范围等）
-	bool	● true，构建模型成功。 ● false，构建模型失败。

异常处理

无。

约束说明

无。

8.4.11.3 ReleaseModelBuff

函数原型

void ReleaseModelBuff(ModelBufferData& output);

功能说明

释放模型Buffer。

参数说明

参数名	输入/输出	类型	描述

output	输出	ModelBufferData&	离线模型结构体对象。
			struct ModelBufferData
			{
			void* data;
			uint32_t length;
			};

返回值

无。

异常处理

无。

约束说明

无。

8.5 TE API 参考

8.5.1 TE 简介

TE（Tensor Engine）是基于TVM（Tensor Virtual Machine）的自定义算子开发框架。TVM是社区的开源项目，旨在将各算子的生成规则进一步抽象，将算子本身分成各个操作原语，在需要的时候加以组合。TVM会根据算子的计算过程的定义，使用Schedule技术和Codegen技术，生成对指定硬件的算子。

由于Schedule是描述在硬件上实现一个算子的计算过程，这需要较强的硬件知识。为了简化用户书写算子的难度，我们在TVM的基础上，简化了书写Schedule的难度，采用“Auto schedule”的概念，提供了一组TensorEngine API，来组合出算子的计算。用户通过使用API进行适当的组合定义一个算子的计算过程，把Schedule交给Auto schedule去完成。本文介绍在TVM基础上定义的微过程API，用户可通过这些API来开发自己的算子。

8.5.2 说明

为了改善用户自定义算子的易用性，提高开发效率，我们对部分Vector算子进行了模块化封装。针对Element wise操作的接口，用户需要利用TVM原语来定义自己的输入Tensor（张量），然后调用封装的接口，简要描述下自定义算子的计算过程，然后再调用提供的Auto schedule和Build接口把自定义算子编译生成二进制可执行文件。

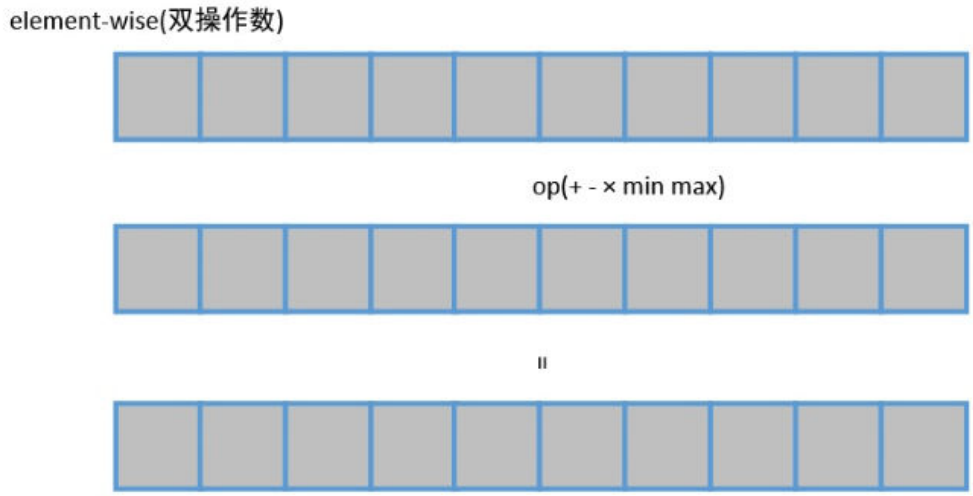
在Tensor Engine中提供了一组封装好的接口，目前主要涵盖向量运算，包括Element-wise类操作接口、Reduction 操作接口、Broadcast操作接口、Index操作接口、Concat操作接口、卷积接口、4D/5D互转接口、矩阵乘接口。

您可以在DDK包的安装目录下的“ddk/site-packages/te-0.4.0.egg/te/lang/cce/te_compute”、“ddk/site-packages/te-0.4.0.egg/te/lang/cce/te_schedule”和“ddk/site-packages/topi-0.4.0.egg/topi/generic”目录下查看接口的定义文件。如果通过引导安装的方式同时安装Mind Studio和DDK，您可以使用Mind Studio安装用户登录Mind Studio服务器，在“~/tools/che/ddk/ddk/site-packages”路径下的对应目录下查看接口定义文件。每个接口对应的定义文件请参见下文中具体的接口描述。关于如何使用这些接口进行自定义算子的开发以及代码的编译运行指导，请参见《TE自定义算子开发指导》。

Element-wise 类操作接口

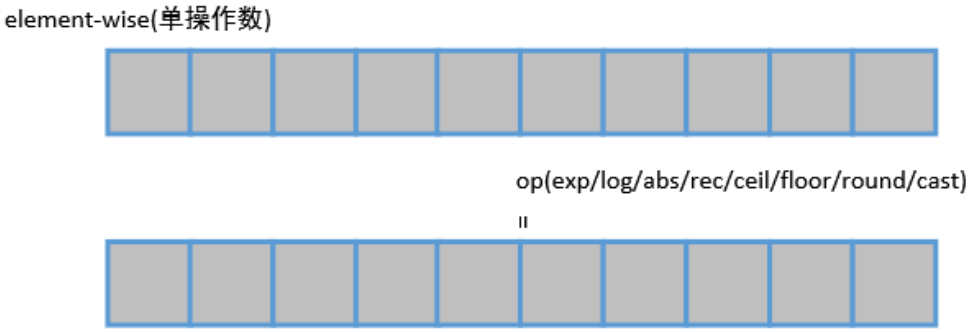
对输入数据进行逐个元素对应运算的操作，这种操作的输出往往具有与输入相同的形状。

- 双操作数操作。输入两个Tensor（张量）进行2个Tensor之间逐元素运算，得到一个结果的操作。
  - 两个Tensor对应元素相加，详细请参见**8.5.3.1 te.lang.cce.vadd(lhs, rhs)**。
  - 两个Tensor对应元素相减，详细请参见**8.5.3.2 te.lang.cce.vsub(lhs, rhs)**。
  - 两个Tensor对应元素相乘，详细请参见**8.5.3.3 te.lang.cce.vmul(lhs, rhs)**。
  - 两个Tensor对应元素比较取较小值，详细请参见**8.5.3.4 te.lang.cce.vmin(lhs, rhs)**。
  - 两个Tensor对应元素比较取较大值，详细请参见**8.5.3.5 te.lang.cce.vmax(lhs, rhs)**。
  - 两个Tensor对应元素按位求或，详细请参见**8.5.3.6 te.lang.cce.vor(lhs, rhs)**。
  - 两个Tensor对应元素按位求与，详细请参见**8.5.3.7 te.lang.cce.vand(lhs, rhs)**。

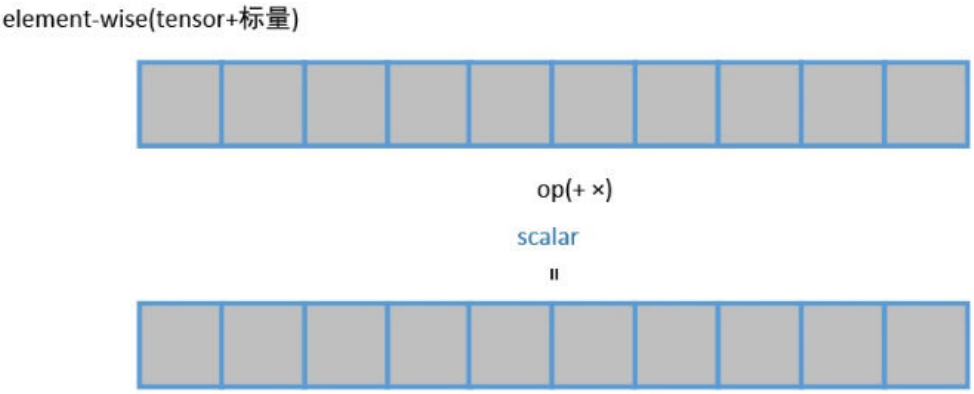


- 单操作数操作。对输入的一个张量进行逐元素运算。
  - 自然指数，详细请参见**8.5.3.11 te.lang.cce.vexp(raw_tensor)**。
  - 对数，详细请参见**8.5.3.10 te.lang.cce.vlog(raw_tensor)**。

- 绝对值，详细请参见**8.5.3.12 te.lang.cce.vabs(raw_tensor)**。
- 倒数，详细请参见**8.5.3.13 te.lang.cce.vrec(raw_tensor)**。
- 向上取整，详细请参见**8.5.3.21 te.lang.cce.ceil(raw_tensor)**。
- 向下取整，详细请参见**8.5.3.22 te.lang.cce.floor(raw_tensor)**。
- 四舍六入，详细请参见**8.5.3.23 te.lang.cce.round(raw_tensor)**。
- 转换数据类型，详细请参见**8.5.3.14 te.lang.cce.cast_to(data, dtype, f1628IntegerFlag=False)**。
- 实现relu计算，详细请参见**8.5.3.15 te.lang.cce.vrelu(raw_tensor)**。
- 按位取反，详细请参见**8.5.3.16 te.lang.cce.vnot(raw_tensor)**。



- Tensor与标量数值的运算操作。输入Tensor的每个元素与同一个数值进行运算。
  - Tensor加上标量，详细请参见**8.5.3.8 te.lang.cce.vadds(raw_tensor, scalar)**。
  - Tensor乘上标量，详细请参见**8.5.3.9 te.lang.cce.vmuls(raw_tensor, scalar)**。

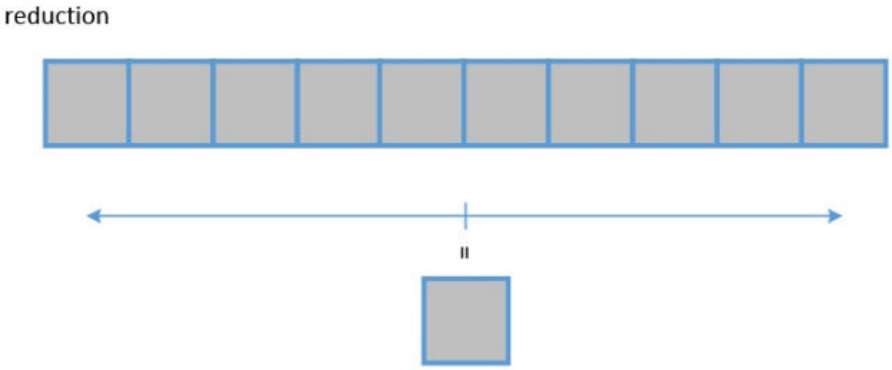


- 三操作数。输入为3个。tensor与标量按照上边的规则混合运算得到一个结果的操作。
  - 对一个Tensor进行缩放后和第二个Tensor相加，详细请参见**8.5.3.17 te.lang.cce.vaxpy(lhs, rhs, scalar)**。
  - 输入为三个Tensor，计算x * y + z，详细请参见**8.5.3.18 te.lang.cce.vmla(x, y, z)**。
  - 输入为三个Tensor，计算x * z + y，详细请参见**8.5.3.19 te.lang.cce.vmadd(x, y, z)**。
  - 输入为三个Tensor，计算relu (x * z + y)，详细请参见**8.5.3.20 te.lang.cce.vmaddrelu(x, y, z)**。

Reduction 操作接口

压缩某一维的数据，沿着指定方向将数据进行累加或累乘等操作，该操作输出比输入数据维度少一维的结果。

- 沿某个轴进行累加，详细请参见**8.5.3.24 te.lang.cce.sum(raw_tensor, axis, keepdims=False)**。
- 沿某个轴求最小值，详细请参见**8.5.3.25 te.lang.cce.reduce_min(raw_tensor, axis, keepdims=False)**。
- 沿某个轴求最大值，详细请参见**8.5.3.26 te.lang.cce.reduce_max(raw_tensor, axis, keepdims=False)**。



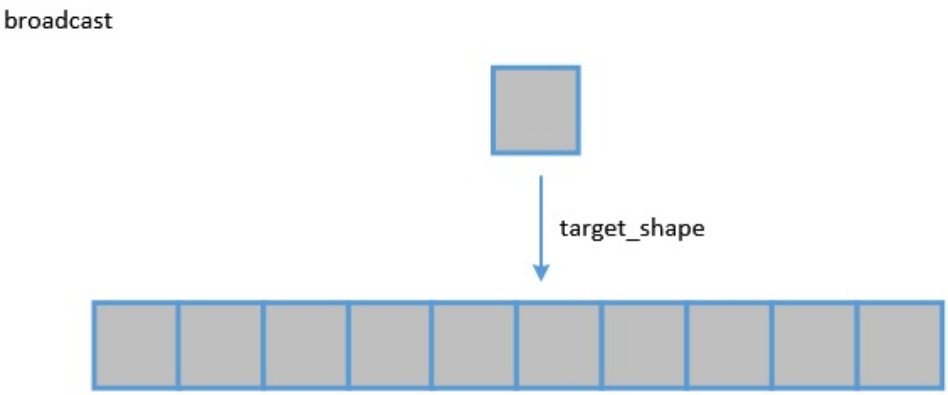
须知

使用限制：由于CCE计算平台的数据排布限制，reduction操作后的数据需要进行一次重排才能进行后续的操作。故当前在使用不同类型的接口时，暂时不能在reduction操作后进行任何向量运算操作。

Broadcast 操作接口

Broadcast操作主要是用于处理两个不同形状的tensor进行计算，将低维度的一个操作数按照高维度操作数的维度进行广播，使得两个操作数的维度相同后，再进行element-wise计算。

把较小的tensor广播为较大的tensor，详细请参见**8.5.3.28 te.lang.cce.broadcast(var, shape, output_dtype=None)**。

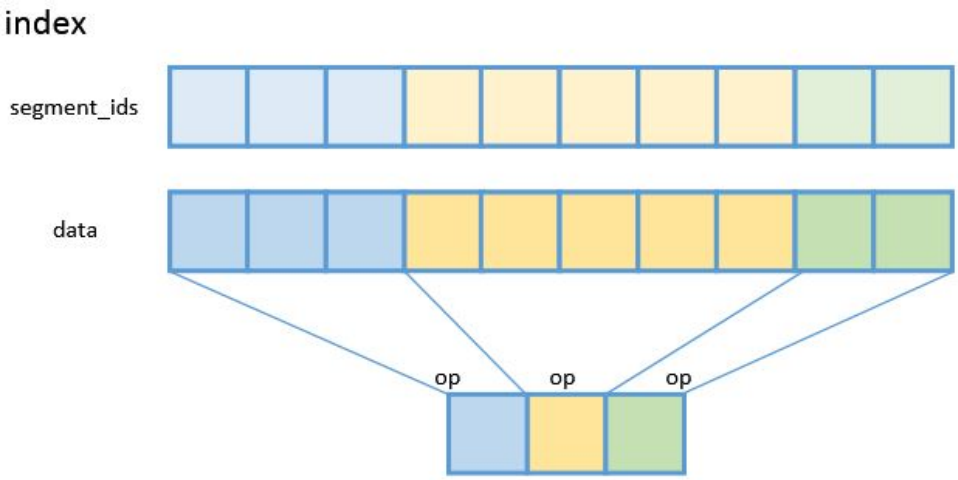


版权所有 © 华为技术有限公司

Index 操作接口

Index操作，对Tensor进行分段计算，比如求和、均值、内积、最大值、最小值等。

- 对Tensor进行分段求和，详细请参见**8.5.3.29 te.lang.cce.unsorted_segment_sum(tensor, segment_ids, num_segments, init_value=0)**。
- 对Tensor进行分段求均值，详细请参见**8.5.3.30 te.lang.cce.unsorted_segment_mean(tensor, segment_ids, num_segments, init_value=0)**。
- 对Tensor进行分段求内积，详细请参见**8.5.3.31 te.lang.cce.unsorted_segment_prod(tensor, segment_ids, num_segments, init_value=0)**。
- 对Tensor进行分段求最小值，详细请参见**8.5.3.32 te.lang.cce.unsorted_segment_min(tensor, segment_ids, num_segments, init_value=0)**。
- 对Tensor进行分段求最大值，详细请参见**8.5.3.33 te.lang.cce.unsorted_segment_max(tensor, segment_ids, num_segments, init_value=0)**。



Concat 操作接口

Concat操作是沿着某个轴对多个输入的Tensor进行连接。

沿一个轴连接Tensor，详细请参见**8.5.3.34 te.lang.cce.concat(raw_tensors, axis)**。

卷积接口

卷积接口是用来实现卷积算子的接口。

实现卷积，详细请参见**8.5.3.35 te.lang.cce.conv(*args)**。

4D/5D 互转接口

4D维度NCHW与5D维度NC1HWC0互转接口。

- 4D转5D，详细请参见**8.5.3.36 te.lang.cce.compute_four2five(input, raw_shape_4D)**。

- 5D转4D，详细请参见**8.5.3.37 te.lang.cce.compute_five2four(input, raw_shape_4D)**。

矩阵乘接口

矩阵乘接口是用来实现矩阵相乘的接口。

详细请参见**8.5.3.38 te.lang.cce.matmul(tensor_a, tensor_b, trans_a=False, trans_b=False, alpha_num=1.0, beta_num=0.0, tensor_c=None)**。

融合算子接口

TE算子融合功能需要对单算子的融合能力和入参等信息做预处理，因此在生成单算子时需要调用指定API生成单算子的*.so文件和*.json文件。

详细请参见**8.5.5.1 bool BuildTeCustomOp(std::string ddkVer, std::string opName, std::string opPath, std::string opFuncName, const char *format, ...)**。

8.5.3 compute 接口

8.5.3.1 te.lang.cce.vadd(lhs, rhs)

两个tensor按元素相加，元素的数据类型需要一致，支持的类型：float16，float32，int32。int8，uint8会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：表示lhs + rhs，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vadd(data1, data2)
```

8.5.3.2 te.lang.cce.vsub(lhs, rhs)

两个tensor按元素相减，元素的数据类型需要一致，支持的类型：float16，float32，int32。int8，uint8会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。

- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：表示 lhs - rhs，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vsub(data1, data2)
```

8.5.3.3 te.lang.cce.vmul(lhs, rhs)

两个tensor按元素相乘，元素的数据类型需要一致，支持的类型：float16，float32，int32。int8，uint8会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：表示lhs * rhs，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmul(data1, data2)
```

8.5.3.4 te.lang.cce.vmin(lhs, rhs)

两个tensor按元素比较并取较小值，元素的数据类型需要一致，支持的类型：float16，float32，int32。int8，uint8会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：结果tensor，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
```

```
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmin(data1, data2)
```

8.5.3.5 te.lang.cce.vmax(lhs, rhs)

两个tensor按元素比较并取较大值，元素的数据类型需要一致，支持的类型：float16，float32，int32。int8，uint8会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：结果tensor，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vmax(data1, data2)
```

8.5.3.6 te.lang.cce.vor(lhs, rhs)

两个tensor元素按位取或，元素的数据类型需要一致，支持的类型：int16，uint16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：表示lhs按位或rhs，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "int16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vor(data1, data2)
```

8.5.3.7 te.lang.cce.vand(lhs, rhs)

两个tensor元素按位取与，元素的数据类型需要一致，支持的类型：int16，uint16。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。

返回值

res_tensor：表示lhs按位与rhs，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "int16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
res = te.lang.cce.vand(data1, data2)
```

8.5.3.8 te.lang.cce.vadds(raw_tensor, scalar)

将raw_tensor中每个元素加上标量scalar，支持的类型：float16，float32。int8，uint8，int32会被转换为float16。如果scalar数据类型与raw_tensor数据类型不一致，计算中会转换为对应数据类型。

该接口在elewise_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型。
- scalar：raw_tensor中元素要加的系数，标量类型。

返回值

res_tensor：表示raw_tensor + scalar，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vadds(data, scalar)
```

8.5.3.9 te.lang.cce.vmuls(raw_tensor, scalar)

将raw_tensor中每个元素乘上标量scalar，支持的类型：float16，float32。int8，uint8，int32会被转换为float16。如果scalar数据类型与raw_tensor数据类型不一致，计算中会转换为对应数据类型。

该接口在elewise_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型。
- scalar：raw_tensor中元素要乘的系数，标量类型。

返回值

res_tensor：表示raw_tensor * scalar，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vmuls(data, scalar)
```

8.5.3.10 te.lang.cce.vlog(raw_tensor)

对raw_tensor中的每个元素做对数ln(x)运算，支持的类型：float16。int8，uint8，int32，float32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示ln(raw_tensor)，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vlog(data)
```

8.5.3.11 te.lang.cce.vexp(raw_tensor)

对tensor中的每个元素做自然指数运算e^x，支持的类型：float16。int8，uint8，int32，float32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示e^(raw_tensor)，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vexp(data)
```

8.5.3.12 te.lang.cce.vabs(raw_tensor)

对tensor中的每个元素做绝对值运算|x|，支持的类型：float16。int8，uint8，int32，float32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示|raw_tensor|，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vabs(data)
```

8.5.3.13 te.lang.cce.vrec(raw_tensor)

对tensor中的每个元素做倒数运算 1 / x，支持的类型：float16，float32。int8，uint8，int32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示1 / raw_tensor，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vrec(data)
```

8.5.3.14 te.lang.cce.cast_to(data, dtype, f1628IntegerFlag=False)

数据类型转换，把data中的数据转换为dtype类型。

该接口在common.py中定义。

支持如下的类型转换。

表 8-30 支持的类型转换

源数据类型	目的数据类型
float32	float16
float32	int8
float32	uint8
float16	float32
float16	int8
float16	uint8
float16	int32
int8	float16
int8	uint8
int32	float16
int32	int8
int32	uint8

参数说明

- data：输入tensor，tvm.tensor类型。
- dtype：目的数据类型，字符串类型。
- f1628IntegerFlag：默认值为False。如果转换前数据的小数部分为0，参数f1628IntegerFlag需要设为True；如果转换前数据的小数部分不为0，参数f1628IntegerFlag需要设为False。

返回值

res_tensor：转换后的数据，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.cast_to(data,"float32")
```

8.5.3.15 te.lang.cce.vrelu(raw_tensor)

对tensor中的每个元素做线性整流运算relu，支持的类型：float16。int8，uint8，int32，float32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示relu(raw_tensor)，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vrelu(data)
```

8.5.3.16 te.lang.cce.vnot(raw_tensor)

对tensor中的每个元素按位取反，支持的类型：int16，uint16。

该接口在elewise_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示raw_tensor按位取反，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "int16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.vnot(data)
```

8.5.3.17 te.lang.cce.vaxpy(lhs, rhs, scalar)

将lhs中每个元素乘上标量scalar，再加上rhs中的对应元素，lhs和rhs tensor的数据类型要求一致，支持的类型：float16、float32。int8、uint8、int32类型会被转换为float16。

scalar的数据类型如果跟tensor不一致，会被转换成tensor的数据类型。

该接口在elewise_compute.py中定义。

参数说明

- lhs：左tensor，tvm.tensor类型。
- rhs：右tensor，tvm.tensor类型。
- scalar：lhs中元素要乘的系数，标量类型。

返回值

res_tensor：表示lhs * scalar + rhs，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
```

```
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
scalar = tvm.const(2, dtype =input_dtype)
res = te.lang.cce.vaxpy(data1, data2, scalar)
```

8.5.3.18 te.lang.cce.vmla(x, y, z)

将x中每个元素乘上y中的对应元素，再加上z中的对应元素，三个tensor的数据类型要求一致，支持的类型：float16、float32。int8、uint8、int32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- x：tensor，tvm.tensor类型
- y：tensor，tvm.tensor类型
- z：tensor，tvm.tensor类型

返回值

res_tensor：表示x * y + z，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmla(data1, data2, data3)
```

8.5.3.19 te.lang.cce.vmadd(x, y, z)

将x中每个元素乘上z中的对应元素，再加上y中的对应元素，三个tensor的数据类型要求一致，支持的类型：float16、float32。int8、uint8、int32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- x：tensor，tvm.tensor类型。
- y：tensor，tvm.tensor类型。
- z：tensor，tvm.tensor类型。

返回值

res_tensor：表示x * z + y，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmadd(data1, data2, data3)
```

8.5.3.20 te.lang.cce.vmaddrelu(x, y, z)

将x中每个元素乘上z中的对应元素，再加上y中的对应元素，然后做线性整流，三个tensor的数据类型要求一致，支持的类型：float16、float32。int8、uint8、int32类型会被转换为float16。

该接口在elewise_compute.py中定义。

参数说明

- x：tensor，tvm.tensor类型。
- y：tensor，tvm.tensor类型。
- z：tensor，tvm.tensor类型。

返回值

res_tensor：表示relu(x * z + y)，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data1 = tvm.placeholder(shape, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape, name="data2", dtype=input_dtype)
data3 = tvm.placeholder(shape, name="data3", dtype=input_dtype)
res = te.lang.cce.vmaddrelu(data1, data2, data3)
```

8.5.3.21 te.lang.cce.ceil(raw_tensor)

对raw_tensor中的每个元素向上取整，支持的类型：float16。float32类型会被转换为float16。结果为int32。

该接口在cast_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型。

返回值

res_tensor：表示ceil(raw_tensor)，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.ceil(data)
```

8.5.3.22 te.lang.cce.floor(raw_tensor)

对raw_tensor中的每个元素向下取整，支持的类型：float16。float32类型会被转换为float16。结果为int32。

该接口在cast_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型

返回值

res_tensor：表示floor(raw_tensor)，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.floor(data)
```

8.5.3.23 te.lang.cce.round(raw_tensor)

对raw_tensor中的每个元素四舍六入，0.5取偶数，例如`1.5->2.0，2.5->2.0`，支持的类型：float16。float32类型会被转换为float16。结果为int32。

该接口在cast_compute.py中定义。

参数说明

raw_tensor：输入tensor，tvm.tensor类型

返回值

res_tensor：表示round(raw_tensor)，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.round(data)
```

8.5.3.24 te.lang.cce.sum(raw_tensor, axis, keepdims=False)

按某个轴求和，进行降维，支持的类型：float16、float32。int8、uint8、int32类型会被转换为float16。

该接口在reduction_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型
- axis：做reduce操作的轴，取值范围：[-d,d-1]，其中d是raw_tensor的维数，int或list类型
- keepdims：默认值是False，表示做reduce操作后，操作的轴长度为0，例如，原shape是(10,10,10)，keepdims=False时，reduce后shape是(10,10)。若将该参数值设置为True，表示做reduce操作后，操作的轴的长度设置为1，例如，原shape是(10,10,10)，keepdims=True时，reduce后shape是(10,10,1)。

返回值

res_tensor：求和后的tensor，tvm.tensor类型

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.sum(data, axis=1)
```

8.5.3.25 te.lang.cce.reduce_min(raw_tensor, axis, keepdims=False)

按某个轴求最小值，进行降维，支持的类型：float16。int8、uint8、int32、float32类型会被转换为float16。

该接口在reduction_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型。
- axis：做reduce操作的轴，取值范围：[-d,d-1]，其中d是raw_tensor的维数，int或list类型。
- keepdims：默认值是False，表示做reduce操作后，操作的轴长度为0，例如，原shape是(10,10,10)，keepdims=False时，reduce后shape是(10,10)。若将该参数值设置为True，表示做reduce操作后，操作的轴的长度设置为1，例如，原shape是(10,10,10)，keepdims=True时，reduce后shape是(10,10,1)。

返回值

res_tensor：取最小值后的tensor，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_min(data, axis=1)
```

8.5.3.26 te.lang.cce.reduce_max(raw_tensor, axis, keepdims=False)

按某个轴求最大值，进行降维，支持的类型：float16。int8、uint8、int32、float32类型会被转换为float16。

该接口在reduction_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型。
- axis：做reduce操作的轴，取值范围：[-d,d-1]，其中d是raw_tensor的维数，int或list类型。
- keepdims：默认值是False，表示做reduce操作后，操作的轴长度为0，例如，原shape是(10,10,10)，keepdims=False时，reduce后shape是(10,10)。若将该参数值设置为True，表示做reduce操作后，操作的轴的长度设置为1，例如，原shape是(10,10,10)，keepdims=True时，reduce后shape是(10,10,1)。

返回值

res_tensor：取最大值后的tensor，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_max(data, axis=1)
```

8.5.3.27 te.lang.cce.reduce_prod(raw_tensor, axis, keepdims=False)

按某个轴求乘积，进行降维，支持的类型：float16。int8、uint8、int32、float32类型会被转换为float16。

该接口在reduction_compute.py中定义。

参数说明

- raw_tensor：输入tensor，tvm.tensor类型。
- axis：做reduce操作的轴，取值范围：[-d,d-1]，其中d是raw_tensor的维数，int类型。
- keepdims：默认值是False，表示做reduce操作后，操作的轴长度为0，例如，原shape是(10,10,10)，keepdims=False时，reduce后shape是(10,10)。若将该参数值设置为True，表示做reduce操作后，操作的轴的长度设置为1，例如，原shape是(10,10,10)，keepdims=True时，reduce后shape是(10,10,1)。

返回值

res_tensor：按某个轴求乘积后的tensor，tvm.tensor类型。

调用示例

```
shape = (1024,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.reduce_prod(data, axis=1)
```

8.5.3.28 te.lang.cce.broadcast(var, shape, output_dtype=None)

把var broadcast为大小为shape的tensor，结果的数据类型由output_dtype指定，var可以是标量，或者是一个tensor，要求var的shape与第二个参数shape的长度一致，每个维度的大小要么与shape相等，要么为1，为1的维度会被broadcast到与shape一致。例如var的维度为（2，1，64），shape为（2，128，64），运算结果var的维度变为（2，128，64）。支持的类型：float16、float32、int32。

该接口在broadcast_compute.py中定义。

参数说明

- var：需要broadcast的数据，标量或者tensor类型。
- shape：目标shape，进行broadcast操作的目标shape。
- output_dtype：输出数据类型，默认值var.dtype。

返回值

res_tensor：由var扩展后得到的tensor，shape为参数指定的shape，数据类型为output_dtype。

调用示例

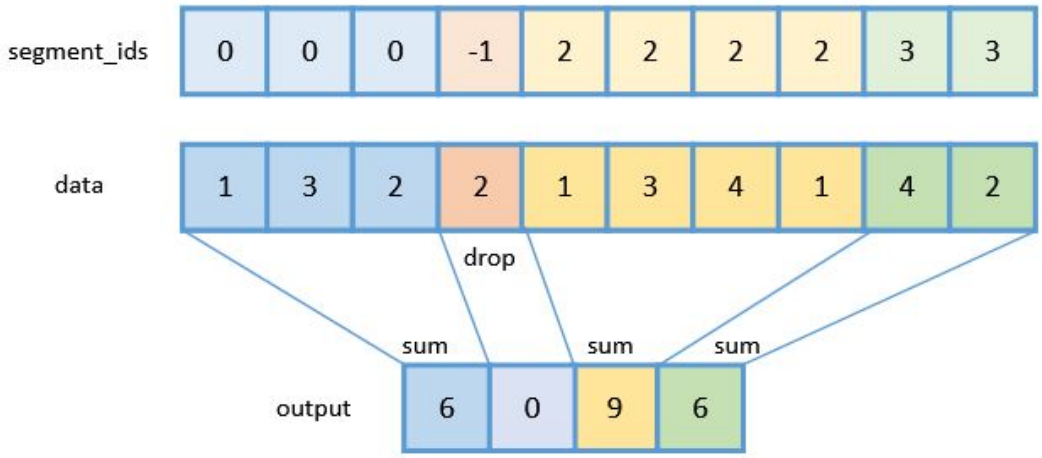
```
outshape = (1024,1024)
shape = (1024,1)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data", dtype=input_dtype)
res = te.lang.cce.broadcast(data, outshape)
```

8.5.3.29 te.lang.cce.unsorted_segment_sum(tensor, segment_ids, num_segments, init_value=0)

使用数组segment_ids对tensor进行分段求和。假设输入为data，输出为output，则output[i] = sum(data[j...])，其中“j...”是一个数组，“j...”中的元素j满足：segment_ids[j] == i。

如果某个下标i在segment_ids中没有出现，则output[i] = init_value。比如下图中，1在segment_ids中没有出现，则output[1] = 0。

如果segment_ids中某个值为负数，则对应位置的data中的值则丢弃。比如下图中，segment_ids[3] = -1，则data[3]的值被丢弃，不参与计算。



segment_ids其长度必须和data的第一维的长度相同。num_segments必须大于等于segment_ids的最大值加1。

支持的数据类型：float16、float32、int32。

该接口在segment_compute.py中定义。

参数说明

- tensor：输入tensor，必须是如下数据类型之一：float16、float32、int32。
- segment_ids：一维数组，对输入tensor进行分段的数组，其长度必须和输入tensor的第一维的长度相同。同时支持有序和无序。
- num_segments：输出tensor的第一维的长度。其值必须大于等于segment_ids的最大值加1。

- init_value：当segment_ids中某个下标不存在时，其输出的默认值。根据算子的实现来确定。默认值为0。

返回值

res_tensor：表示计算后的tensor。

调用示例

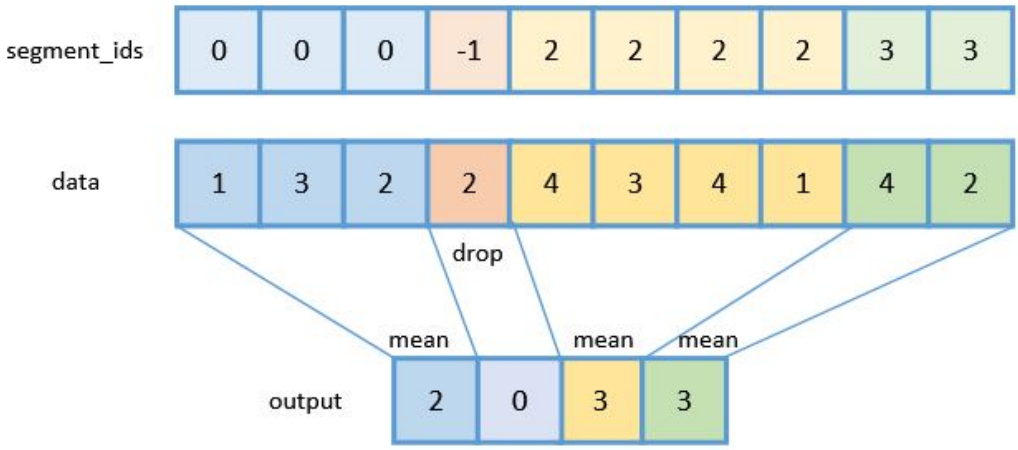
```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_sum(data, segment_ids, num_segments)
res.shape = (6,1024)
# res[0] = 0
# res[1] = data[0] + data[1]
# res[2] = 0
# res[3] = 0
# res[4] = data[2]
# res[5] = data[3] + data[4]
```

8.5.3.30 te.lang.cce.unsorted_segment_mean(tensor, segment_ids, num_segments, init_value=0)

使用数组segment_ids对tensor进行分段求均值。假设输入为data,输出为output，则output[i] = (1 / len(j...)) sum(data[j...])，其中“j...”是一个数组，“j...”中的元素j满足：segment_ids[j] == i。

如果某个下标i在segment_ids中没有出现,则output[i] = init_value。比如下图中,1在segment_ids中没有出现，则output[1] = 0。

如果segment_ids中某个值为负数，则对应位置的data中的值则丢弃。比如下图中，segment_ids[3] = -1，则data[3]的值被丢弃，不参与计算。



segment_ids其长度必须和data的第一维的长度相同。num_segments必须大于等于segment_ids的最大值加1。

支持的数据类型：float16、float32、int32。

该接口在segment_compute.py中定义。

参数说明

- tensor：输入tensor，必须是如下数据类型之一：float16、float32、int32。
- segment_ids：一维数组，对输入tensor进行分段的数组，其长度必须和输入tensor的第一维的长度相同。同时支持有序和无序。
- num_segments：输出tensor的第一维的长度。其值必须大于等于segment_ids的最大值加1。
- init_value：当segment_ids中某个下标不存在时，其输出的默认值。根据算子的实现来确定。默认值为0。

返回值:

res_tensor：表示计算后的tensor。

调用示例

```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,5,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_mean(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 0
# res[1] = (data[0] + data[1]) / 2
# res[2] = 0
# res[3] = 0
# res[4] = 0
# res[5] = (data[2] + data[3] + data[4]) / 3
```

8.5.3.31 te.lang.cce.unsorted_segment_prod(tensor, segment_ids, num_segments, init_value=0)

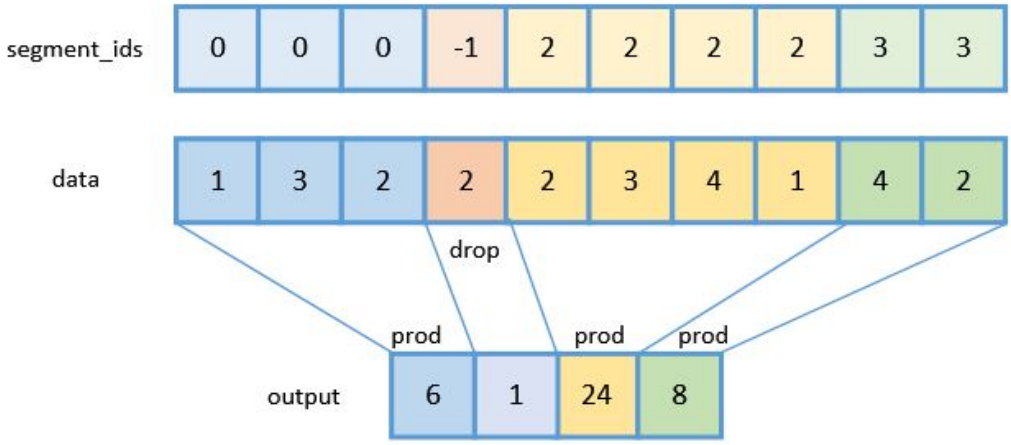
使用数组segment_ids对tensor进行分段求内积。假设输入为data，输出为output，则output[i] = product(data[j...])，其中“j...”是一个数组，“j...”中的元素j满足：segment_ids[j] == i。

说明

product表示求内积，即data[j...]中所有元素相乘。

如果某个下标i在segment_ids中没有出现，则output[i] = init_value。比如下图中，1在segment_ids中没有出现，则output[1] = 0。

如果segment_ids中某个值为负数，则对应位置的data中的值则丢弃。比如下图中，segment_ids[3] = -1，则data[3]的值被丢弃，不参与计算。



segment_ids其长度必须和data的第一维的长度相同。num_segments必须大于等于segment_ids的最大值加1。

支持的数据类型：float16、float32、int32。

该接口在segment_compute.py中定义。

参数说明

- tensor：输入tensor，必须是如下数据类型之一：float16、float32、int32。
- segment_ids：一维数组，对输入tensor进行分段的数组，其长度必须和输入tensor的第一维的长度相同。同时支持有序和无序。
- num_segments：输出tensor的第一维的长度。其值必须大于等于segment_ids的最大值加1。
- init_value：当segment_ids中某个下标不存在时，其输出的默认值。根据算子的实现来确定。默认值为0。

返回值

res_tensor：表示计算后的tensor。

调用示例

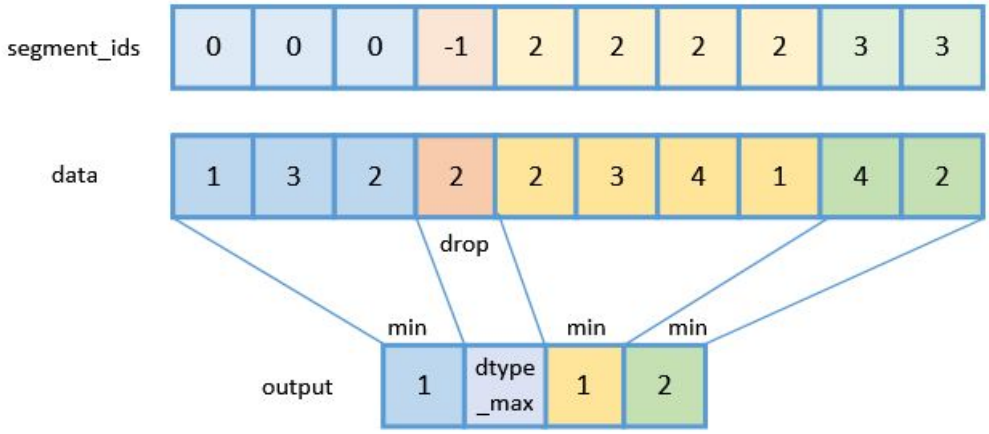
```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_prod(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 1
# res[1] = (data[0] * data[1])
# res[2] = 1
# res[3] = 1
# res[4] = data[2]
# res[5] = (data[3] * data[4])
```

8.5.3.32 te.lang.cce.unsorted_segment_min(tensor, segment_ids, num_segments, init_value=0)

使用数组segment_ids对tensor进行分段求最小值。假设输入为data，输出为output，则output[i] = min(data[j...])，其中“j...”是一个数组，“j...”中的元素j满足：segment_ids[j] == i。

如果某个下标i在segment_ids中没有出现，则output[i] = init_value。比如下图中，1在segment_ids中没有出现，则output[1] = 0。

如果segment_ids中某个值为负数,则对应位置的data中的值则丢弃。比如下图中，segment_ids[3] = -1，则data[3]的值被丢弃，不参与计算。



segment_ids其长度必须和data的第一维的长度相同。num_segments必须大于等于segment_ids的最大值加1。

支持的数据类型：float16、float32。

该接口在segment_compute.py中定义。

参数说明

- tensor：输入tensor，必须是如下数据类型之一：float16、float32、int32。
- segment_ids：一维数组，对输入tensor进行分段的数组，其长度必须和输入tensor的第一维的长度相同。同时支持有序和无序。
- num_segments：输出tensor的第一维的长度。其值必须大于等于segment_ids的最大值加1。
- init_value：当segment_ids中某个下标不存在时，其输出的默认值。根据算子的实现来确定。默认值为0。

返回值

res_tensor：表示计算后的tensor。

调用示例

```
import tvm
import te.lang.cce
shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1", dtype=input_dtype)
```

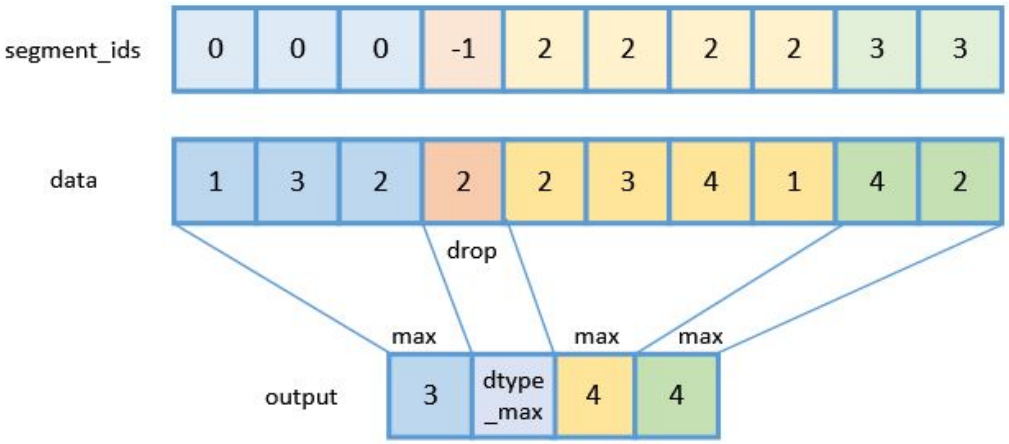
```
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_min(data, segment_ids, num_segments)
# res.shape = (6,1024)
# res[0] = 65504(float16的最大值)
# res[1] = min(data[0], data[1])
# res[2] = 65504
# res[3] = 65504
# res[4] = data[2]
# res[5] = min(data[3], data[4])
```

8.5.3.33 te.lang.cce.unsorted_segment_max(tensor, segment_ids, num_segments, init_value=0)

使用数组segment_ids对tensor进行分段求最大值。假设输入为data，输出为output，则output[i] = max(data[j...])，其中“j...”是一个数组，“j...”中的元素j满足：segment_ids[j] == i。

如果某个下标i在segment_ids中没有出现,则output[i] = init_value。比如下图中，1在segment_ids中没有出现，则output[1] = 0。

如果segment_ids中某个值为负数，则对应位置的data中的值则丢弃。比如下图中，segment_ids[3] = -1，则data[3]的值被丢弃，不参与计算。



segment_ids其长度必须和data的第一维的长度相同。num_segments必须大于等于segment_ids的最大值加1。

支持的数据类型：float16、float32、int32。

该接口在segment_compute.py中定义。

参数说明

- tensor：输入tensor，必须是如下数据类型之一：float16、float32、int32。
- segment_ids：一维数组，对输入tensor进行分段的数组，其长度必须和输入tensor的第一维的长度相同。同时支持有序和无序。
- num_segments：输出tensor的第一维的长度。其值必须大于等于segment_ids的最大值加1。
- init_value：当segment_ids中某个下标不存在时，其输出的默认值。根据算子的实现来确定。默认值为0。

返回值

res_tensor：表示计算后的tensor。

调用示例

```
import tvm
import te.lang.cce shape = (5,1024)
input_dtype = "float16"
data = tvm.placeholder(shape, name="data1",type=input_dtype)
segment_ids = [1,1,4,5,5]
num_segments = 6
res = te.lang.cce.unsorted_segment_max(data,segment_ids,num_segments)
# res.shape = (6,1024)
# res[0] = 65504(float16的最大值)
# res[1] = max(data[0], data[1])
# res[2] = 65504
# res[3] = 65504
# res[4] = data[2]
# res[5] = max(data[3], data[4])
```

8.5.3.34 te.lang.cce.concat(raw_tensors, axis)

在指定轴上对输入的多个Tensor进行重新连接。

输入raw_tensors为多个Tensor，数据类型相同。

如果raw_tensors[i].shape = [D0, D1, ... Daxis(i), ...Dn]，沿着轴axis连接后的结果的shape为：[D0, D1, ... Raxis, ...Dn]。

其中：Raxis = sum(Daxis(i))。

对输入tensor来说，除了轴axis以外，其他轴的维度要完全一致。

例如：

```
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
concat([t1, t2], 0)  # [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
concat([t1, t2], 1)  # [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]]

# tensor t1的shape为 [2, 3]
# tensor t2的shape为 [2, 3]
concat([t1, t2], 0).shape  # [4, 3]
concat([t1, t2], 1).shape  # [2, 6]
```

参数axis也可以为负数，表示从维度的最后开始计算，表示第axis + len(shape)跟轴。

例如：

```
t1 = [[[1, 2], [2, 3]], [[4, 4], [5, 3]]]
t2 = [[[7, 4], [8, 4]], [[2, 10], [15, 11]]]
concat([t1, t2], -1)
```

结果为：

```
[[[ 1,  2,  7,  4],
  [ 2,  3,  8,  4]],

 [[ 4,  4,  2, 10],
  [ 5,  3, 15, 11]]]
```

支持的数据类型：int8、uint8、int16、int32、float16、float32。

该接口在concat_compute.py中定义。

参数说明

- raw_tensors：tensor list，list类型，元素为tvm.tensor，且tensor shape的最后一维要32字节对齐。
- axis：做 concat 操作的轴，取值范围：[-d,d-1]，其中d是raw_tensor的维数。

返回值

res_tensor：重新连接后的tensor，tvm.tensor类型。

调用示例

```
import tvm
import te.lang.cce
shape1 = (64,128)
shape1 = (64,128)
input_dtype = "float16"
data1 = tvm.placeholder(shape1, name="data1", dtype=input_dtype)
data2 = tvm.placeholder(shape2, name="data1", dtype=input_dtype)
data = [data1, data2]
res = te.lang.cce.concat(data, 0)
# res.shape = (128,128)
```

8.5.3.35 te.lang.cce.conv(*args)

在给定4-D输入和filter的情况下计算2-D卷积。

要求输入Tensor和filter Tensor的格式均为NCHW。

接口可以支持bias，支持的数据类型float16。

该接口在conv_compute.py中定义。

参数说明

*args：是一个list，参数数量可变。

- bias场景参数列表为（其中hasBias需要为True）：
  A, W, B, res_dtype, padh, padw, strideh, stridew, hasBias
- 非bias场景参数列表为（其中hasBias需要为False）：
  A, W, res_dtype, padh, padw, strideh, stridew, hasBias
  其中：
  - A：输入Tensor，即卷积计算的feature map。
  - W：filter Tensor，即卷积计算的卷积核。
  - B：bias Tensor，即卷积计算的偏置。
  - res_dtype：输出Tensor的数据类型，即卷积计算结果的数据类型。
  - padh：padding的高，即卷积计算在feature map的H方向上的填充数。
  - padw：padding的宽，即卷积计算在feature map的W方向上的填充数。
  - strideh：H方向的步长，即卷积计算filter在feature map的H方向上移动的步长。
  - stridew：W方向的步长，即卷积计算filter在feature map的W方向上移动的步长。
  - hasBias：是否带bias。

- 约束：假设feature map的shape为（Fn, Fc, Fh, Fw），filter的shape为（Wn, Wc, Wh, Ww），卷积结果的输出shape为（On, Oc, Oh, Ow），padh记为Ph，padw记为Pw，strideh记为Sh，stridew记为Sw，上述各参数之间需满足如下关系：
  - Fc = Wc
  - On = Fn
  - Oc = Wn
  - Oh = ((Fh + 2Ph - Wh) / Sh) + 1
  - Ow = ((Fw + 2Pw - Ww) / Sw) + 1

返回值

res_tensor：表示卷积计算的tensor，即卷积计算的结果输出。

调用示例

```
import tvm
import te.lang.cce
shape_in = (64,64)
shape_w = (3,3)
in_dtype = "float16"
A = tvm.placeholder(shape_in, name='A', dtype=in_dtype)
W = tvm.placeholder(shape_w, name='W', dtype=in_dtype)
b_shape = (shape_w[0], )
B = tvm.placeholder(b_shape, name='B', dtype=res_dtype)
padh = 0
padw = 0
strideh = 1
stridew = 1
res = te.lang.cce.conv(A, W, B, in_dtype, padh, padw, strideh, stridew, True)
# res = A * W + B
```

8.5.3.36 te.lang.cce.compute_four2five(input, raw_shape_4D)

把给定4-D “NCHW”数据格式转换为5-D “NC1HWC0”数据格式。支持的数据类型：float16。

该接口在dim_conv.py中定义。

参数说明

- input：输入tensor，4-D格式(N, C, H, W)，tvm.tensor类型。
- raw_shape_4D：输入tensor的维度。

返回值:

res_tensor：转换为5-D格式(N, C1, H, W, C0)后的tensor，tvm.tensor类型

调用示例

```
import tvm
import te.lang.cce
raw_shape = (N,C,H,W)
in_dtype = "float16"
input = tvm.placeholder(raw_shape, name='input', dtype=in_dtype)
res = te.lang.cce.compute_four2five(input, raw_shape)
# res.shape = (N,(C+15)//16,H,W,16)
```

8.5.3.37 te.lang.cce.compute_five2four(input, raw_shape_4D)

把给定5-D “NC1HWC0”数据格式转换为4-D “NCHW”数据格式。支持的数据类型：float16。

该接口在dim_conv.py中定义。

参数说明

- input：输入tensor，5-D格式(N, C1, H, W, C0)，tvm.tensor类型。
- raw_shape_4D：转换后tensor的维度。

返回值

res_tensor：转换为4-D格式(N, C, H, W)后的tensor，tvm.tensor类型。

调用示例

```
import tvm
import te.lang.cce
raw_shape = (N,C,H,W)
input_shape = (N,(C+15)//16,H,W,16)
in_dtype = "float16"
input = tvm.placeholder(input_shape, name='input', dtype=in_dtype)
res = te.lang.cce.compute_five2four(input, raw_shape)
# res.shape = (N,C,H,W)
```

8.5.3.38 te.lang.cce.matmul(tensor_a, tensor_b, trans_a=False, trans_b=False, alpha_num=1.0, beta_num=0.0, tensor_c=None)

矩阵乘，计算：tensor_c=alpha_num * trans_a(tensor_a) * trans_b(tensor_b) + beta_num * tensor_c。

tensor_a与tensor_b的shape后两维(经过对应转置)需要满足矩阵乘(M, K) * (K, N) = (M, N)，且batch数只支持1。tensor_a数据排布要满足L0A的分形结构，tensor_b要满足L0B的分形结构，mini形态下，数据类型只支持float16。

该接口在mmad_compute.py中定义。

参数说明

- tensor_a：A矩阵，tvm.tensor类型。
- tensor_b：B矩阵，tvm.tensor类型
- trans_a：A矩阵是否转置，bool类型。
- trans_b：B矩阵是否转置，bool类型
- alpha_num : A*B矩阵系数，只支持1.0
- beta_num : C矩阵系数，只支持0.0
- tensor_c : C矩阵，tvm.tensor类型，由于beta_num只支持0.0，此参数为预留扩展接口

返回值

tensor_c：根据关系运算计算后得到的tensor，tvm.tensor类型。

调用示例

```
import tvm
import te.lang.cce
a_shape = (1024, 256)
b_shape = (256, 512)
a_fractal_shape = (a_shape[0] // 16, a_shape[1] // 16, 16, 16)
b_fractal_shape = (b_shape[0] // 16, b_shape[1] // 16, 16, 16)
in_dtype = "float16"
tensor_a = tvm.placeholder(a_fractal_shape, name='tensor_a', dtype=in_dtype)
tensor_b = tvm.placeholder(a_fractal_shape, name='tensor_b', dtype=in_dtype)
res = te.lang.cce.matmul(tensor_a, tensor_b, False, False)
```

8.5.4 build 接口

build接口的作用是把定义好的计算过程生成schedule对象，并进行build生成cce算子文件。

8.5.4.1 topi.generic.auto_schedule(outs)

生成dsl的schedule。该接口在cce.py中定义。

参数说明

outs：对算子的计算图描述，就是DSL。

返回值:

schedule：算子的计算schedule。

调用示例

```
import te.lang.cce
from te import tvm
import topi.generic

shape = (28,28)
dtype = "float16"
# 定义输入
data = tvm.placeholder(shape, name="data", dtype=dtype)
# 描述算子计算过程
res = te.lang.cce.vabs(data)
with tvm.target.cce():
    # 生成schedule对象
     sch = topi.generic.auto_schedule(res)
```

8.5.4.2 te.lang.cce.cce_build_code(sch, config_map = {})

对schedule打印lower code或者进行build。该接口在cce_schedule.py中定义。

参数说明

- sch：tvm.schedule，schedule to build or to print lower code。
- config_map：build的参数配置，是一个字典，默认是{}并使用默认配置，包含如下key。
  - print_ir：是否打印lower IR code，默认是True。
  - need_build：是否进行build，默认是True。
  - name：算子的名字，默认是`cce_op`。

– tensor_list：算子的输入和输出tensor列表，输入是placeholder接口返回的tensor对象，输出是经过计算后的tensor对象，必填值，否则会报错。而且这个列表决定了生成算子的kernel函数的参数的顺序，和此list中的输入和输出的顺序是一致的。

返回值

无。

调用示例

```
import te.lang.cce
from te import tvm
from topi import generic
# 定义输入占位符
data = tvm.placeholder(shape, name="data", dtype=dtype)
with tvm.target.cce():
    # 描述算子计算过程
    res = te.lang.cce.vabs(data)
    # 生成schedule对象
    sch = generic.auto_schedule(res)
# 定义build配置参数
config = {"print_ir" : True,
        "need_build" : True,
        "name" : "abs_28_28_float16",
        "tensor_list" : [data,res]
    }
# build算子
te.lang.cce.cce_build_code(sch, config)
```

8.5.5 融合接口

TE算子融合功能需要对单算子的融合能力和入参等信息做预处理，因此在生成单算子时需要调用指定API，如下对该API的使用方法和限制做详细介绍。

8.5.5.1 bool BuildTeCustomOp(std::string ddkVer, std::string opName, std::string opPath, std::string opFuncName, const char *format, ...)

调用该接口用于生成单算子的*.o文件和*.json文件。您可以在DDK包的安装目录下的“ddk/include/inc/custom/custom_op.h”文件中查看接口的定义。

参数说明

- ddkVer：指产品的版本号。

  版本号统一规则：x.y.z.patch.B***。

  x：对应VR版本；

  y：对应C版本；

  z：对应发布计数；

  path：补丁号，可选参数；

  B***：内部B版本号；

  具体版本号参考DDK包的安装目录下ddk_info文件中的配置。
- opName：指网络中单个算子节点名称，其值不能为空，且名称中只能包含字母、数字、下划线或中划线。
- opPath：指算子文件的路径，以“/”区分目录且文件名称不带扩展名，其值不能为空。文件名称中只能包含字母、数字、下划线或中划线。

- opFuncName：指算子文件中算子函数名称。函数名称中只能包含字母、数字、下划线或中划线。
- format：算子的入参，属于变长变量，需要与算子的参数顺序保持一致。

返回值:

- true：表示生成单算子成功
- false：表示生成单算子失败。

调用示例

```
std::string FilePath   = "topi/cce/caffe_reduction_layer";
std::string FuncName   = "caffe_reduction_layer_cce";
std::string KernelName = "cce_reductionLayer_5_10_5_5_float16__1_SUMSQ_1_0";

// i => int; s => string; f => dobule; O => bool, and bool value is Py_True or Py_False
te::BuildTeCustomOp(1.1.1.patch.B001, "reduction", FilePath, FuncName,"(i,i,i,i), s, i, s, f, s", 5, 5, 5, 5,
"float16", 1, "SUM", 1.0,KernelName.c_str());
```

8.5.6 编译依赖接口

如下接口只在算子编译过程中用到，用户在编写算子时不会直接调用。您可以在DDK包的安装目录下的“ddk/include/inc/tensor_engine/cce_aicpu_3rd_party.h”文件中查看接口的定义。

- __aicpu__ void *aicpu_malloc(unsigned int size)
  该接口用于实现内存申请，入参size表示申请内存的大小，返回申请内存的指针。
- __aicpu__ void aicpu_free(void * ptr)
  该接口用于实现内存释放，ptr为内存指针。
- __aicpu__ double log(double x)
  该接口用于实现对x取log值，double类型。
- __aicpu__ double exp(double x)
  该接口用于实现对x取exp值，double类型。
- __aicpu__ double round(double x)
  该接口用于实现对x四舍五入，double类型。
- __aicpu__ double floor(double x)
  该接口用于实现对x向下取整，double类型。
- __aicpu__ double ceil(double x);
  该接口用于实现对x向上取整，double类型。
- __aicpu__ double trunc(double x);
  该接口用于实现对x截断，double类型。
- __aicpu__ double sqrt(double x);
  该接口用于实现对x开平方，double类型。
- __aicpu__ float logf(float x)
  该接口用于实现对x取log值，double类型。
- __aicpu__ float expf(float x)
  该接口用于实现对x取exp值，float类型。

- __aicpu__ float roundf(float x)
  该接口用于实现对x四舍五入，float类型。
- __aicpu__ float floorf(float x)
  该接口用于实现对x向下取整，float类型。
- __aicpu__ float ceilf(float x);
  该接口用于实现对x向上取整，float类型。
- __aicpu__ float truncf(float x);
  该接口用于实现对x截断，float类型。
- __aicpu__ float sqrtf(float x);
  该接口用于实现对x开平方，float类型。

8.5.7 使用方式

8.5.7.1 使用示例

下面通过一个简单的示例，将上面的接口串起来。该示例是实现一个简单的支持float16数据类型的求绝对值的算子。

```
import te.lang.cce
from te import tvm
import topi
shape = (28,28)
dtype = "float16"
# 定义输入
data = tvm.placeholder(shape, name="data", dtype=dtype)

# 描述算子计算过程
res = te.lang.cce.vabs(data)

with tvm.target.cce():
    # 生成schedule对象
     sch = topi.generic.auto_schedule(res)

# 定义build配置参数
config = {"print_ir" : True,
          "need_build" : True,
          "name" : "abs_28_28_float16",
          "tensor_list" : [data,res]}
# build算子
te.lang.cce.cce_build_code(sch, config)
```

8.5.7.2 异常处理

接口如果执行异常，一般都是由于错误的入参引起的。下边例子给出了tensor_list不全给出的错误信息。

代码片段

```
data = tvm.placeholder(shape, name="data", dtype=inp_dtype)
with tvm.target.cce():
    res = te.lang.cce.vabs(data)
    sch = generic.auto_schedule(res)
    config = {"print_ir": need_print,
    "need_build": need_build,
    "name": kernel_name,
    "tensor_list": [res]}
te.lang.cce.cce_build_code(sch, config)
```

执行会发生如下错误：

```
Traceback (most recent call last):
  File "llt/tensor_engine/ut/testcase_python/tf_abs/test_tf_abs_cce.py", line 71, in test_cce_tf_abs_99991_fp16
    tf_abs_cce((99991,), dtype = "Float16", need_build = False, need_print = False, kernel_name =
"cce_tf_abs")
  File "/home1/repotvm/tensor_engine/topi/python/topi/cce/tf_abs.py", line 68, in tf_abs_cce
    te.lang.cce.cce_build_code(sch, config)
  File "/home1/repotvm/tensor_engine/python/te/lang/cce/te_schedule/cce_schedule.py", line 381, in
cce_build_code
    _build(sch, tensor_list, local_config_map["name"])
  File "/home1/repotvm/tensor_engine/python/te/lang/cce/te_schedule/cce_schedule.py", line 338, in _build
    mod = tvm.build(sch, tensor_list, device, name=name)
  File "/home1/repotvm/tensor_engine/python/te/tvm/build_module.py", line 432, in build
    binds=binds)
  File "/home1/repotvm/tensor_engine/python/te/tvm/build_module.py", line 353, in lower
    stmt = ir_pass.StorageFlatten(stmt, binds, 64)
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/function.py", line 280, in my_api_func
    return flocal(*args)
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/_ctypes/function.py", line 183, in __call__
    ctypes.byref(ret_val), ctypes.byref(ret_tcode)))
  File "/home1/repotvm/tensor_engine/python/te/tvm/_ffi/base.py", line 66, in check_call
    raise TVMError(py_str(_LIB.TVMGetLastError()))
TVMError: [17:12:02] /home1/repotvm/tensor_engine/src/pass/storage_flatten.cc:249: Check failed: it !=
buf_map_.end() Cannot find allocated buffer for placeholder(data, 0x27d7290)
```

参数更正为如下情况后，问题得到解决。

```
"tensor_list" : [data, res]
```

9 命令参考

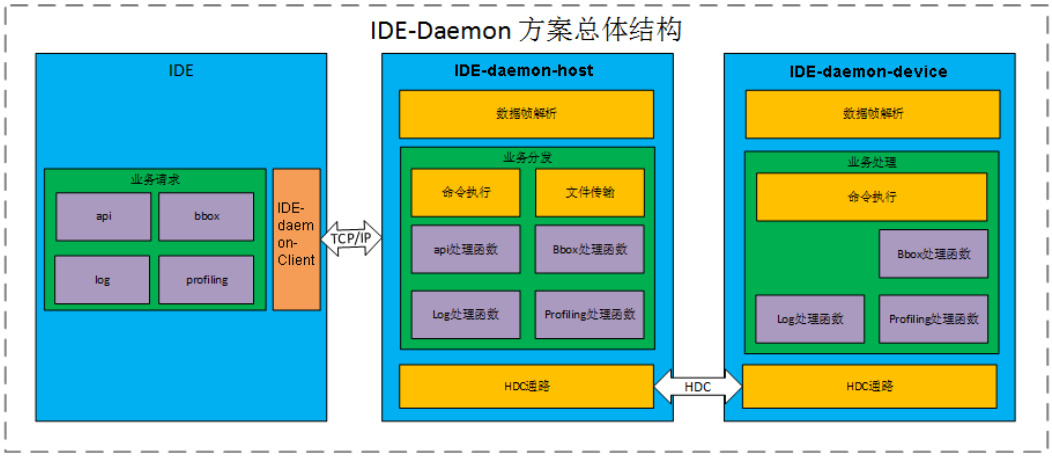
9.1 IDE-daemon-client命令参考

9.1 IDE-daemon-client 命令参考

9.1.1 简介

执行IDE-daemon-client命令后，将指定的命令发送Host侧，如果命令需要在Host侧执行，则直接执行；如果命令需要在Device侧执行，则再由Host侧连接Device，然后在Device侧执行命令，具体请参见《IDE-daemon-client命令参考》。

图 9-1 IDE-daemon 总体结构



目前IDE-daemon-client命令支持在Host侧服务器（PCIe形态）上执行，也可以在DDK所在的服务器上执行：

- 如果在Host侧服务器上，您可以HwHiAiUser用户登录Host侧服务器，直接执行IDE-daemon-client命令。下文以此为例。
- 如果安装了DDK，可以在DDK服务器上执行IDE-daemon-client命令。
  - DDK部署在单独的服务器上，不与Host侧服务器共用，您可以DDK安装用户登录服务器，先执行如下命令设置环境变量，再执行IDE-daemon-client命令。其中，*“DDK安装目录”*需根据实际情况替换。

版权所有 © 华为技术有限公司

```
export  LD_LIBRARY_PATH=DDK安装目录/ddk/uihost/lib
export  PATH=$PATH:DDK安装目录/ddk/uihost/bin
```

– DDK部署在Host侧服务器上且DDK安装用户为HwHiAiUser，您可以HwHiAiUser用户登录服务器，直接执行IDE-daemon-client命令，不用设置环境变量。

如果您直接复制下文中的命令，由于PDF文档格式的限制，超过单行的命令，会自动换行，因此您需要手动将多行命令合并成一行，参数之间以空格分割。

9.1.2 IDE Daemon 发送命令给 Device 侧执行

命令功能

IDE Daemon发送命令给Device侧执行，比如发送查看Device侧的日期信息的命令。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> [--device <deviceID>] --cmd <command>
```

参数说明

表 9-1 参数说明

参数名称	参数描述	是否必选	默认值
--host	● *host_ip*：指定Host侧的IP地址。 ● *port*：指定端口号。	是	*port*默认值为22118。
--device	如果Host连接了多个Device，则可以通过--device *deviceID*指定具体的Device。如果不指定Device，则默认指定*deviceID*为0的Device。	否	*deviceID*默认值为0。

参数名称	参数描述	是否必选	默认值
--cmd	指定需要Device侧执行的命令，命令列表请参见**command列表**。 除**date**命令外，其它command命令必须放在单引号或双引号中。如果command命令中已有双引号，则将command命令放在单引号中；如果command命令中已有单引号，则将command命令放在双引号中。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，查看Device侧的日期信息。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --cmd date
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

*command*列表

在半固化类的命令中，*xxxx*、*yy*等内容需要根据实际情况替换为目录名称或文件名称。

命令中的~表示启动IDE-daemon-host或IDE-daemon-device进程的用户的家目录。

类别	命令	备注
固化	pkill cce-gdbserver	-
	ide_cmd.sh --reboot	sudo提权，实际执行命令为 **sudo** *command*，实际执行为reboot mini命令。 用于重启开发者板。
	date	-
	mkdir -p /tmp/profiler-tools/	-

	rm /opt/mini/mini_developerKit*.rar	-
	chmod +x /opt/mini/minirc_install_phase1.sh	-
	chmod +w ~/ide_daemon/rollback_idedaemon_crt.sh	-
	chmod +w ~/ide_daemon/backup_idedaemon_crt.sh	-
	/opt/mini/minirc_install_phase1.sh	-
	tar -xvf ~/host/EVB.tar -C ~/host/	-
	rm -rf /var/dlog/*	-
	rm -rf ~/host	-
	mkdir -p ~/host	-
	rm -rf ~/HIAI_PROJECTS/dlog.tar	-
	tar -cPf ~/HIAI_PROJECTS/dlog.tar /var/dlog/	-
	grep -rn "firmware upgrade finish, start to reboot" /var/davinci/log	-
	grep -rn "fail" /var/davinci/log/firmware_upgrade_progress.log	-
	python -c \"import future;import sys;sys.exit(future.__version__ < 0.15)\"	检查Host侧的future库的版本号。 实际执行的命令为： python -c "import future;import sys;sys.exit(future.__version__ < 0.15)"
	python -c \"import numpy;import sys;sys.exit(numpy.__version__ < 1.14)\"	检查Host侧的numpy库的版本号。 实际执行的命令为： python -c "import numpy;import sys;sys.exit(numpy.__version__ < 1.14)"

	"python -c \"\n"\"import enum\n"\"import sys \n"\"try:\n"\" if enum.__version__ >= 0.4:\n"\" sys.exit(0)\n"\"except AttributeError:\n"\" if enum.version >= 1.0:\n"\" sys.exit(0)\n"\"\""	检查Host侧的enum库或enum34库的版本号。
		实际执行的命令为：
		python -c \"
		import enum
		import sys
		try:
		if enum.__version__ >= 0.4:
		sys.exit(0)
		except AttributeError:
		if enum.version >= 1.0:
		sys.exit(0)
		\"

	cat /etc/sys_version.conf	获取开发者板的Ascend 310软件版本号。
半固化	rm -rf ~/hdcd; rm -rf ~/HIAI_PROJECTS/*xxxx*; rm -rf ~/HIAI_DATANDMODELSET/*xxxx*; rm -rf ~/profiler-app/*xxxx*; rm -rf ~/ide_daemon/*xxxx*	-
	rm ~/HIAI_PROJECTS/*xxxx*; rm ~/HIAI_DATANDMODELSET/*xxxx*; rm ~/profiler-app/*xxxx*; rm ~/ide_daemon/*xxxx*	-
	wc -l ~/hdcd/*xxxx*; wc -l ~/HIAI_PROJECTS/*xxxx*; wc -l ~/HIAI_DATANDMODELSET/*xxxx*; wc -l ~/profiler-app/*xxxx*	-
	mkdir -p ~/hdcd/*xxxx*; mkdir -p ~/HIAI_PROJECTS/*xxxx*; mkdir -p ~/HIAI_DATANDMODELSET/*xxxx*; mkdir -p ~/profiler-app/*xxxx*; mkdir -p ~/ide_daemon/*xxxx*	执行可执行文件。
	mkdir ~/HIAI_PROJECTS/*xxxx*; mkdir ~/HIAI_DATANDMODELSET/*xxxx*; mkdir ~/profiler-app/*xxxx*; mkdir ~/ide_daemon/*xxxx*	-
	tar -cf ~/hdcd/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx xxxx*; tar -cf ~/HIAI_PROJECTS/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx xxxx*; tar -cf ~/HIAI_DATANDMODELSET/*xxxx*/*yy*.tar -C ~/HIAI_DATANDMODELSET/*xxxx xxxx*; tar -cf ~/profiler-app/*xxxx*/*yy*.tar -C ~/profiler-app/*xxxx xxxx*	将“-C参数后的目录及其下的文件”压缩成“-cf参数后的压缩包”。 将xxxx修改为实际的目录名称，将yy修改为实际的tar包名称。

	tar -xvf ~/hdcd/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx*; tar -xvf ~/HIAI_PROJECTS/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx*; tar -xvf ~/HIAI_DATANDMODELSET/*xxxx*/*yy*.tar -C ~/HIAI_DATANDMODELSET/*xxxx*; tar -xvf ~/profiler-app/*xxxx*/*yy*.tar -C ~/profiler-app/*xxxx*	-
	mv ~/hdcd/*xxxx* ~/HIAI_PROJECTS/*xxxx*; mv ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*; mv ~/profiler-app/*xxxx* ~/profiler-app/*xxxx*	-
	chmod +x ~/hdcd/*xxxx*; chmod +x ~/HIAI_PROJECTS/*xxxx*; chmod +x ~/profiler-app/*xxxx*; chmod +x ~/ide_daemon/*xxxx*	-
	chmod -w ~/hdcd/*xxxx*; chmod -w ~/ide_daemon/*xxxx*	-
	cp ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*; cp -af ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*	-
	sha512sum ~/hdcd/*xxxx*; sha512sum ~/HIAI_DATANDMODELSET/*xxxx*	-
	find ~/HIAI_DATANDMODELSET/*xxxx* -type f|xargs sha512sum|sort	-
	~/HIAI_PROJECTS/*xxxx*; ~/ide_daemon/*xxxx*	执行可执行文件。
	~/HIAI_PROJECTS/xxxx ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。
	cce-gdbserver *ip_address:port* --attach *xxx* 2>&1; cce-gdbserver *ip_address:port* ~/ide_daemon/debug/*xxxx*	*ip_address*和*port*需根据实际情况替换。 *xxx*需根据实际情况替换为进程ID。
	pgrep -x -f "cce-gdbserver *ip_address:port* --attach *xxx*"; pgrep -x -f "cce-gdbserver *ip _ddress:port* ~/ide_daemon/debug/*xxxx*"	*ip_address*和*port*需根据实际情况替换。 *xxx*需根据实际情况替换为进程ID。
	pidof xxxx	xxxx需根据实际情况替换为进程名
	ps ux | awk '{print}'| sed 1d; ps ux | awk '{print $x \t$y}'| sed 1d	-

	ps -ef | awk '{print}'| sed 1d; ps -ef | awk '{print $x \t$y}'| sed 1d	-
	ide_cmd.sh --eth0_up ip _address	sudo提权，实际执行命令为 **sudo** *command*，脚本中的命令为ifconfig eth0 ip_address 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	ide_cmd.sh --setup_eth0 *ip_address netmask_address gw_address*	sudo提权，实际执行命令为 **sudo** *command*，脚本中的命令为ifconfig eth0 ip_address netmask netmask_address && route add default gw gw_address 仅适用于开发者板。 *ip_address*/ netmask_address/ gw_address需根据实际情况替换。
	sed -i '/iface.eth0/{n;s/address.*/address *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	sed -i '/iface.eth0/{n;n;s/netmask.*/netmask *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	sed -i '/iface.eth0/{n;n;n;s/gateway.*/gateway *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。

	del /f/s/q *D:*\profiler-app; del /f/s/q *D:*\profiler-app\xxx	windows命令，删除文件。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	dir *D:*\profiler-app; dir *D:*\profiler-app\xxx	windows命令，列出文件夹内容。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	md *D:*\profiler-app; md *D:*\profiler-app\xxx	windows命令，创建文件夹。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	rd /s/q *D:*\profiler-app; rd /s/q *D:*\profiler-app\xxx	windows命令，删除文件夹。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	tar -xvf *D:*\profiler-app\xxx.tar -C *D:*\profiler-app\yyy	windows命令，创建压缩包。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	kill $(pidof ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill $(pidof -x ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill -9 $(pidof ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill -9 $(pidof -x ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill `pgrep -x -f "cce-gdbserver *ip_address:port* --attach xxx"`	*ip_address*和*port*需根据实际情况替换。 xxx需根据实际清楚替换为进程号。
	kill -9 `pgrep -x -f "cce-gdbserver *ip_address:port* --attach xxx"`	*ip_address*和*port*需根据实际情况替换。 xxx需根据实际清楚替换为进程号。

	kill `pgrep -x -f "cce-gdbserver *ip_address:port* ~/ide_daemon/debug/xxxx"`	*ip_address*和*port*需根据实际情况替换。
	kill -9 `pgrep -x -f "cce-gdbserver *ip_address:port* ~/ide_daemon/debug/xxxx"`	*ip_address*和*port*需根据实际情况替换。
	cd ~/HIAI_PROJECTS/*xxxx*/out;./*yy*	到目录下执行可执行文件，无参数。
	cd ~/HIAI_PROJECTS/*xxxx*/out;./*yy* ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	到目录下执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。
	cd ~/HIAI_PROJECTS/xxxx/out;~/HIAI_PROJECTS/xxxx/out/yyy ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	到目录下执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。

9.1.3 IDE Daemon 发送命令给 Host 侧执行

命令功能

IDE Daemon发送命令给Host侧执行，比如发送查看Host侧的日期信息的命令。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --hostcmd <command>
```

参数说明

表 9-2 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--hostcmd	指定需要Host侧执行的命令，命令列表请参见**command列表**。 除**date**命令外，其它command命令必须放在单引号或双引号中。如果command命令中已有双引号，则将command命令放在单引号中；如果command命令中已有单引号，则将command命令放在双引号中。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，查看Host侧的日期信息。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --hostcmd date
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

*command*列表

在半固化类的命令中，*xxxx*、*yy*等内容需要根据实际情况替换为目录名称或文件名称。

命令中的~表示启动IDE-daemon-host或IDE-daemon-device进程的用户的家目录。

类别	命令	备注
固化	pkill cce-gdbserver	-
	ide_cmd.sh --reboot	sudo提权，实际执行命令为 **sudo** *command*，实际执行为reboot mini命令。 用于重启开发者板。
	date	-
	mkdir -p /tmp/profiler-tools/	-

	rm /opt/mini/mini_developerKit*.rar	-
	chmod +x /opt/mini/minirc_install_phase1.sh	-
	chmod +w ~/ide_daemon/rollback_idedaemon_crt.sh	-
	chmod +w ~/ide_daemon/backup_idedaemon_crt.sh	-
	/opt/mini/minirc_install_phase1.sh	-
	tar -xvf ~/host/EVB.tar -C ~/host/	-
	rm -rf /var/dlog/*	-
	rm -rf ~/host	-
	mkdir -p ~/host	-
	rm -rf ~/HIAI_PROJECTS/dlog.tar	-
	tar -cPf ~/HIAI_PROJECTS/dlog.tar /var/dlog/	-
	grep -rn "firmware upgrade finish, start to reboot" /var/davinci/log	-
	grep -rn "fail" /var/davinci/log/firmware_upgrade_progress.log	-
	python -c \"import future;import sys;sys.exit(future.__version__ < 0.15)\"	检查Host侧的future库的版本号。 实际执行的命令为： python -c "import future;import sys;sys.exit(future.__version__ < 0.15)"
	python -c \"import numpy;import sys;sys.exit(numpy.__version__ < 1.14)\"	检查Host侧的numpy库的版本号。 实际执行的命令为： python -c "import numpy;import sys;sys.exit(numpy.__version__ < 1.14)"

	"python -c \"\n"\"import enum\n"\"import sys \n"\"try:\n"\" if enum.__version__ >= 0.4:\n"\" sys.exit(0)\n"\"except AttributeError:\n"\" if enum.version >= 1.0:\n"\" sys.exit(0)\n"\"\""	检查Host侧的enum库或enum34库的版本号。
		实际执行的命令为：
		python -c \"
		import enum
		import sys
		try:
		if enum.__version__ >= 0.4:
		sys.exit(0)
		except AttributeError:
		if enum.version >= 1.0:
		sys.exit(0)
		\"

	cat /etc/sys_version.conf	获取开发者板的Ascend 310软件版本号。
半固化	rm -rf ~/hdcd; rm -rf ~/HIAI_PROJECTS/*xxxx*; rm -rf ~/HIAI_DATANDMODELSET/*xxxx*; rm -rf ~/profiler-app/*xxxx*; rm -rf ~/ide_daemon/*xxxx*	-
	rm ~/HIAI_PROJECTS/*xxxx*; rm ~/HIAI_DATANDMODELSET/*xxxx*; rm ~/profiler-app/*xxxx*; rm ~/ide_daemon/*xxxx*	-
	wc -l ~/hdcd/*xxxx*; wc -l ~/HIAI_PROJECTS/*xxxx*; wc -l ~/HIAI_DATANDMODELSET/*xxxx*; wc -l ~/profiler-app/*xxxx*	-
	mkdir -p ~/hdcd/*xxxx*; mkdir -p ~/HIAI_PROJECTS/*xxxx*; mkdir -p ~/HIAI_DATANDMODELSET/*xxxx*; mkdir -p ~/profiler-app/*xxxx*; mkdir -p ~/ide_daemon/*xxxx*	执行可执行文件。
	mkdir ~/HIAI_PROJECTS/*xxxx*; mkdir ~/HIAI_DATANDMODELSET/*xxxx*; mkdir ~/profiler-app/*xxxx*; mkdir ~/ide_daemon/*xxxx*	-
	tar -cf ~/hdcd/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx xxxx*; tar -cf ~/HIAI_PROJECTS/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx xxxx*; tar -cf ~/HIAI_DATANDMODELSET/*xxxx*/*yy*.tar -C ~/HIAI_DATANDMODELSET/*xxxx xxxx*; tar -cf ~/profiler-app/*xxxx*/*yy*.tar -C ~/profiler-app/*xxxx xxxx*	将“-C参数后的目录及其下的文件”压缩成“-cf参数后的压缩包”。 将xxxx修改为实际的目录名称，将yy修改为实际的tar包名称。

	tar -xvf ~/hdcd/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx*; tar -xvf ~/HIAI_PROJECTS/*xxxx*/*yy*.tar -C ~/HIAI_PROJECTS/*xxxx*; tar -xvf ~/HIAI_DATANDMODELSET/*xxxx*/*yy*.tar -C ~/HIAI_DATANDMODELSET/*xxxx*; tar -xvf ~/profiler-app/*xxxx*/*yy*.tar -C ~/profiler-app/*xxxx*	-
	mv ~/hdcd/*xxxx* ~/HIAI_PROJECTS/*xxxx*; mv ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*; mv ~/profiler-app/*xxxx* ~/profiler-app/*xxxx*	-
	chmod +x ~/hdcd/*xxxx*; chmod +x ~/HIAI_PROJECTS/*xxxx*; chmod +x ~/profiler-app/*xxxx*; chmod +x ~/ide_daemon/*xxxx*	-
	chmod -w ~/hdcd/*xxxx*; chmod -w ~/ide_daemon/*xxxx*	-
	cp ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*; cp -af ~/HIAI_PROJECTS/*xxxx* ~/HIAI_PROJECTS/*xxxx*	-
	sha512sum ~/hdcd/*xxxx*; sha512sum ~/HIAI_DATANDMODELSET/*xxxx*	-
	find ~/HIAI_DATANDMODELSET/*xxxx* -type f|xargs sha512sum|sort	-
	~/HIAI_PROJECTS/*xxxx*; ~/ide_daemon/*xxxx*	执行可执行文件。
	~/HIAI_PROJECTS/xxxx ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。
	cce-gdbserver *ip_address:port* --attach *xxx* 2>&1; cce-gdbserver *ip_address:port* ~/ide_daemon/debug/*xxxx*	*ip_address*和*port*需根据实际情况替换。 *xxx*需根据实际情况替换为进程ID。
	pgrep -x -f "cce-gdbserver *ip_address:port* --attach *xxx*"; pgrep -x -f "cce-gdbserver *ip _ddress:port* ~/ide_daemon/debug/*xxxx*"	*ip_address*和*port*需根据实际情况替换。 *xxx*需根据实际情况替换为进程ID。
	pidof xxxx	xxxx需根据实际情况替换为进程名
	ps ux | awk '{print}'| sed 1d; ps ux | awk '{print $x \t$y}'| sed 1d	-

	ps -ef | awk '{print}'| sed 1d; ps -ef | awk '{print $x \t$y}'| sed 1d	-
	ide_cmd.sh --eth0_up ip _address	sudo提权，实际执行命令为 **sudo** *command*，脚本中的命令为ifconfig eth0 ip_address 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	ide_cmd.sh --setup_eth0 *ip_address netmask_address gw_address*	sudo提权，实际执行命令为 **sudo** *command*，脚本中的命令为ifconfig eth0 ip_address netmask netmask_address && route add default gw gw_address 仅适用于开发者板。 *ip_address*/ netmask_address/ gw_address需根据实际情况替换。
	sed -i '/iface.eth0/{n;s/address.*/address *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	sed -i '/iface.eth0/{n;n;s/netmask.*/netmask *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。
	sed -i '/iface.eth0/{n;n;n;s/gateway.*/gateway *ip_address*/;}' /etc/network/interfaces	sudo提权，实际执行命令为 **sudo** *command* 仅适用于开发者板。 *ip_address*需根据实际情况替换。

	del /f/s/q *D:*\profiler-app; del /f/s/q *D:*\profiler-app\xxx	windows命令，删除文件。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	dir *D:*\profiler-app; dir *D:*\profiler-app\xxx	windows命令，列出文件夹内容。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	md *D:*\profiler-app; md *D:*\profiler-app\xxx	windows命令，创建文件夹。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	rd /s/q *D:*\profiler-app; rd /s/q *D:*\profiler-app\xxx	windows命令，删除文件夹。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	tar -xvf *D:*\profiler-app\xxx.tar -C *D:*\profiler-app\yyy	windows命令，创建压缩包。 命令中的*D*需要根据实际情况替换为windows机器上的盘符。
	kill $(pidof ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill $(pidof -x ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill -9 $(pidof ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill -9 $(pidof -x ~/HIAI_PROJECTS/xxxx/out/yyy)	-
	kill `pgrep -x -f "cce-gdbserver *ip_address:port* --attach xxx"`	*ip_address*和*port*需根据实际情况替换。 xxx需根据实际清楚替换为进程号。
	kill -9 `pgrep -x -f "cce-gdbserver *ip_address:port* --attach xxx"`	*ip_address*和*port*需根据实际情况替换。 xxx需根据实际清楚替换为进程号。

	kill `pgrep -x -f "cce-gdbserver *ip_address:port* ~/ide_daemon/debug/xxxx"`	*ip_address*和*port*需根据实际情况替换。
	kill -9 `pgrep -x -f "cce-gdbserver *ip_address:port* ~/ide_daemon/debug/xxxx"`	*ip_address*和*port*需根据实际情况替换。
	cd ~/HIAI_PROJECTS/*xxxx*/out;./*yy*	到目录下执行可执行文件，无参数。
	cd ~/HIAI_PROJECTS/*xxxx*/out;./*yy* ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	到目录下执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。
	cd ~/HIAI_PROJECTS/xxxx/out;~/HIAI_PROJECTS/xxxx/out/yyy ~/HIAI_DATANDMODELSET/xxxx yyy ddd zzz	到目录下执行二进制文件，带固定参数。其中“~/HIAI_DATANDMODELSET/xxxx”、“yyy”、“ddd”、“zzz”表示参数，需根据实际情况替换。

9.1.4 IDE Daemon 传输文件给 Device 侧

命令功能

IDE Daemon传输文件给Device侧。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> [--device <deviceID>] --file <source_dir> <dest_dir>
```

参数说明

表 9-3 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--device	如果Host连接了多个Device，则可以通过--device *deviceID*指定具体的Device。如果不指定Device，则默认指定*deviceID*为0的Device。	否	*deviceID*默认值为0。
--file	● *source_dir*：指定存放源文件的路径，包含文件名。执行IDE-daemon-client命令的用户必须对该文件路径有读写权限。 ● *dest_dir*：指定Device侧用于存放目的文件的路径，包含文件名。必须指定存放文件的路径，否则命令会执行失败。此处文件存放的路径只能是“/home/HwHiAiUser/hdcd”下的目录，在命令中只需指定相对路径，例如目的文件的路径为“/home/HwHiAiUser/hdcd/ide_daemon/B.java”，在命令中只需将file参数指定为“ide_daemon/B.java”。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，将A.java文件传输到Device侧，并将文件保存为B.java。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --file /tmp/A.java "ide_daemon/B.java"
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.5 IDE Daemon 传输文件给 Host 侧

命令功能

IDE Daemon传输文件给Host侧。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --sync <source_dir> <dest_dir>
```

参数说明

表 9-4 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--sync	● *source_dir*：指定存放源文件的路径，包含文件名。执行IDE-daemon-client命令的用户必须对该文件路径有读写权限。 ● *dest_dir*：指定Host侧用于存放目的文件的路径（如“~/ide_daemon”或“~/ide_daemon/”，都表示将文件存放到“ide_daemon”目录下），不包含文件名。必须指定存放文件的路径，否则命令会执行失败。此处文件存放的路径只能是以下路径： – ~/ide_daemon – ~/HIAI_PROJECTS – ~/HIAI_DATANDMODELSET – /opt/mini – /var/log/hisi_logs – /var/dlog	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，将A.java文件传输到Host侧的指定路径下。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --sync /tmp/A.java "~/ide_daemon"
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.6 IDE Daemon 从 Host 侧获取文件

命令功能

IDE Daemon从Host侧获取文件。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --get <dest_dir> <source_dir>
```

参数说明

表 9-5 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--get	• *source_dir*：指定Host侧源文件的路径，包含文件名。此处文件存放的路径只能是以下路径： – ~/ide_daemon – ~/HIAI_PROJECTS – ~/HIAI_DATANDMODELSET – /opt/mini – /var/log/hisi_logs – /var/dlog • *dest_dir*：指定存放目的文件的路径，包含文件名。执行IDE-daemon-client命令的用户必须对该文件路径有读写权限。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，IDE Daemon从Host侧获取B.java，并将文件保存为A.java。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --get /tmp/A.java "~/ide_daemon/B.java"
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.7 检测 IDE Daemon 与 Host 之间的心跳

命令功能

检测IDE Daemon与Host之间的心跳，用于检测Host侧的服务器是否在运行。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --detect
```

参数说明

表 9-6 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。
--detect	检测心跳，如果有心跳，则返回“SUCCESS”；如果没有心跳，则不返回信息。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，检测IDE Daemon与Host之间的心跳。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --detect
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.8 设置日志级别

命令功能

设置Host侧和Device侧的日志级别。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> [--device <deviceID>] --log '<params>'
```

参数说明

表 9-7 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--device	如果Host连接了多个Device，则可以通过--device *deviceID*指定具体的Device。如果不指定Device，则默认指定*deviceID*为0的Device。	否	*deviceID*默认值为0。
--log	*params*包括以下项： ● SetLogLevel 设置全局日志级别，0表示全局级日志级别，1表示模块级日志级别，2表示event日志级别（在设置event日志级别时，“enable”表示开启event日志级别，“disable”表示不开启event日志级别）。 ● SetLogLevel[moduleName:level] 设置指定模块日志级别 – level：error, info, warning, debug, null（表示不打印日志） – moduleName: 模块名称	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，设置Host侧和Device侧的日志级别。

```
IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(0)[error]'
IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(1)[SLOG:error]'
IDE-daemon-client --host xx.xx.xx.xx:22118 --log 'SetLogLevel(2)[enable]'
```

说明

- *xx.xx.xx.xx*需要替换为实际的Host的IP地址。
- moduleName通过枚举形式定义，包括如下值：

```
enum {
DLOG = 0, // Dlog
SLOG, // Slog
IDEDD, // IDE daemon device
IDEDH, // IDE daemon host
LOGAGTH, // log agent host
HCCL, // HCCL
FMK, // Framework
HIAIENGINE, // Matrix
DVPP, // DVPP
RUNTIME, // Runtime
CCE, // CCE
#if (OS_TYPE == LINUX)
HDC, // HDC
#else
HDCL, // HDCL windows has a def with the same name HDC, so change HDC to HDCL
#endif
DRV, // Driver
MDCCONTROL, // Mdc control
MDCFUSION, // Mdc fusion
MDCLOCATION, // Mdc location
MDCPERCEPTION, // Mdc perception
MDCMOP,
MDCFSM,
MDCCOMMON,
MDCMONITOR,
MDCBSWP, // MDC basesoftware platform
MDCDEFAULT, // MDC UNDEFINE
MDCSC, // MDC spatial cognition
MDCBP,
MDCTF,
MLL,
DEVMM, // Dlog memory managent
KERNEL, // Kernel
MDCSMCMD, // sm_control_cmd
MDCSCREEN, // parking_spot_screen
LIBMEDIA, // Libmedia
CCECPU, // ai cpu
ASCENDDK, // AscendDK
ROS, // ROS
HCCP,
```

```
RoCE,
TEFUSION,
PROFILING, // Profiling
DP, // Data Preprocess
APP, // User Application call HIAI_ENGINE_LOG
INVLID_MOUDLE_ID
};
```

9.1.9 IDE Daemon 将 Profiling 采集项的信息发送给 Host

命令功能

IDE Daemon将Profiling采集项的信息发送至Host侧，并启动Profiling采集。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --profile "<params>"
```

参数说明

表 9-8 参数说明

参数名称	参数描述	是否必选	默认值
--host	● *host_ip*：指定Host侧的IP地址。 ● *port*：指定端口号。	是	*port*默认值为22118。
--profile	在*params*处指定Profiling采集项。	是	-

前提条件

已安装Mind Studio，且已提前将Mind Studio侧编译后的工程的相关文件复制到Host侧的对应目录下。

例如，Mind Studio侧的工程名是testProject，工程编译后的文件在“Mind Studio安装用户家目录/tools/projects/testProject/out”目录下，则需要将“Mind Studio安装用户家目录/tools/projects/testProject/out”目录下的文件上传到Host侧的“/home/HwHiAiUser/HIAI_PROJECTS/workspace_mind_studio/testProject/out”目录下，将可执行文件名称修改为workspace_mind_studio_*工程名称*（例如：workspace_mind_studio_testProject），同时，还需要执行**chmod +x workspace_mind_studio_testProject**命令给可执行文件加执行权限。

使用示例

1. 以Mind Studio安装用户登录Mind Studio服务器。
2. 执行如下命令设置环境变量。

```
export  LD_LIBRARY_PATH=~/tools/che/ddk/ddk/uihost/lib
export  PATH=$PATH:~/tools/che/ddk/ddk/uihost/bin
```

3. 执行IDE-daemon-client命令。

   此处是示例命令，需要根据实际情况修改。

   - *--host参数后的“xx.xx.xx.xx”*需要替换为实际的Host服务器的IP地址。
   - */home/ascend*表示Mind Studio安装用户家目录，需要根据实际情况替换。
   - app_dir后的“/xxx/xxx”需替换为Host侧app所在路径，例如：“/home/HwHiAiUser/HIAI_PROJECTS/workspace_mind_studio/testProject/out”。
   - app需要替换为app_dir指定目录下的应用程序名，例如：workspace_mind_studio_testProject。
   - result_dir后的路径需替换为Host服务器上已存在的路径。执行命令后，您可以在result_dir指定的目录下查看采集结果。

```
IDE-daemon-client --host xx.xx.xx.xx:22118 --profile "{\"ddk_dir\": \"/home/ascend/tools/che/ddk/ddk\",\"app\": \"workspace_mind_studio_testProject\",\"app_dir\": \"/home/HwHiAiUser/HIAI_PROJECTS/workspace_mind_studio/testProject/out\", \"umode\": \"MIND\",\"result_dir\": \"/home/ascend/tools/projects/test_ycm\",\"peripheral_profiling\": \"\",\"ts_cpu_profiling\": \"off\",\"ai_cpu_profiling\": \"off\",\"rts_profiling\": \"on\",\"ai_core_profiling_mode\": \"\",\"ai_core_profiling\": \"off\", \"hiai_engine_profiling\": \"off\",\"framework_profiling\": \"off\",\"ctrl_cpu_profiling\": \"off\", \"profiling_mode\": \"online\",\"llc_profiling\": \"off\",\"ddr_profiling\": \"on\",\"deviceid\": \"0\", \"ddr_profiling_events\": \"read,write\", \"hiai_engine_profiling\": \"on\", \"analysis_target\": \"Launch Application\"}"
```

说明

关于各采集项的含义请参见《Ascend 310 Mind Studio开发辅助工具》中的“Profiling”章节。

9.1.10 从 Host 侧采集黑匣子的 dump 异常信息

命令功能

IDE Daemon从Host侧采集黑匣子的dump异常信息。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --bbox <params>
```

参数说明

表 9-9 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--bbox	*params*包括以下项： ● --all：获取所有设备上发生异常的文件列表与路径 ● --status：获取所有设备上发生的异常的数量。若只有一个设备，则返回示例如“{(0,3)}”，表示在设备0上有3个异常；若有多个设备，则返回示例如“{(1,3);(0,3)}”，表示在设备0上有3个异常，在设备1上有3个异常，多个设备间的信息以“;”隔开。 ● --latest：获取最新产生的有异常的文件列表与路径。若返回“FAILED”，则表示没有异常。 ● '--filter --device *deviceID* --except-num *n*'：获取所有指定设备的前*n*个有异常的文件列表和路径，如果需要指定多个设备，则多个设备编号之前以“,”分隔。**--filter --device *deviceID* --except-num *n*** 参数必须放在单引号或双引号中。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，IDE Daemon从Host侧采集所有设备的有异常的文件列表和路径。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --bbox --all
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.11 获取与 Host 连接的 Device 信息

命令功能

IDE Daemon获取与Host连接的Device信息。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --api <params>
```

参数说明

表 9-10 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--api	*params*包括以下项： ● device_info: 与Host连接的Device的列表 ● device_status: 仅支持查询开发者板的摄像头状态(0表示摄像头不存在，1摄像头存在)，如返回100，则其中1表示3559芯片，后两个0表示两个摄像头不存在。 ● board_id: 仅支持查询开发者板的ID。 ● sys_version: 仅支持查询开发者板的产品版本。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，IDE Daemon获取与Host连接的所有Device的列表。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --api device_info
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.12 生成字符串加密文件

命令功能

在IDE Daemon的服务器上生成字符串加密文件，包括*.secu文件和*.store文件，在证书替换时需要使用这两个文件。

命令格式

```
IDE-daemon-client --key
```

参数说明

表 9-11 参数说明

参数名称	参数描述	是否必选	默认值
--key	用于生成字符串加密文件。	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，在当前目录下生成字符串加密文件。

   根据系统提示输入密码。且需确保执行命令的目录，HwHiAiUser用户有写权限。

```
IDE-daemon-client --key
```

9.1.13 同步 IDE Daemon 与 Host 侧的时间

命令功能

以IDE Daemon服务器的时间为准，将Host侧服务器的时间与IDE Daemon服务器的时间同步。

命令格式

```
IDE-daemon-client --host <host_ip>:<port> --time
```

参数说明

表 9-12 参数说明

参数名称	参数描述	是否必选	默认值
--host	• *host_ip*：指定Host侧的IP地址。 • *port*：指定端口号。	是	*port*默认值为22118。

参数名称	参数描述	是否必选	默认值
--time	当“--host”与“--time”两个参数同时使用时，表示以IDE Daemon服务器的时间为准，将Host侧服务器的时间与IDE Daemon服务器的时间同步。 在带有“--host”参数的其它IDE-daemon-client命令中，都可以带“--time”参数（可选参数），例如： IDE-daemon-client --host <*host_ip*>:<*port*> --detect --time	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，检查心跳后，同步时间。
```
IDE-daemon-client --host xx.xx.xx.xx:22118 --detect --time
```

说明

*xx.xx.xx.xx*需要替换为实际的Host的IP地址。

9.1.14 查看帮助信息

命令功能

查看IDE-daemon-client命令的帮助信息。

命令格式

```
IDE-daemon-client --help
```

参数说明

表 9-13 参数说明

参数名称	参数描述	是否必选	默认值
--help	查看帮助信息	是	-

使用示例

1. 以HwHiAiUser用户登录Host侧服务器。
2. 执行命令，查看帮助信息。
```
IDE-daemon-client --help
```

10 附录

10.1 如何获取帮助
10.2 免责声明

10.1 如何获取帮助

10.1.1 联系华为前的准备

为了更好的解决故障，建议在寻求华为技术支持前做好必要的准备工作，包括收集必要的故障信息和做好必要的调试准备。

10.1.2 联系华为技术支持

故障处理过程中遇到难以确定或解决的问题时，请联系华为技术有限公司客户服务中心（电话：4008229999、网址：**http://enterprise.huawei.com**）。同时，您在向华为工程师反馈问题时，请注意收集以下信息：

- 用户名称、地址。
- 联系人姓名、电话号码。
- 故障发生的具体时间。
- 故障现象的详细描述。
- 设备类型、硬件型号及软件版本。
- 故障后已采取的措施和结果。
- 问题的级别及希望解决的时间。

说明

对于以上介绍的可能在本产品上出现的故障现象，按参考处理建议操作后，如果故障仍无法得到解决，请及时与就近的华为办事处或客户服务中心联系，以便能够快速获取华为公司的技术支持。

10.1.3 做好必要的调试准备

在寻求华为技术支持时，华为技术支持工程师可能会协助您做一些操作，以进一步收集故障信息或者直接排除故障。

在寻求技术支持前请准备好单板和端口模块的备件、螺丝刀、螺丝、串口线、网线等可能使用到的物品。

10.1.4 如何使用文档

华为技术有限公司提供全面的随设备发货的指导文档。指导文档能解决您在日常维护或故障处理过程中遇到的常见问题。

为了更好的解决故障，在寻求华为技术支持前，建议充分使用指导文档。

10.1.5 如何从网站获取帮助

华为技术有限公司通过办事处、公司二级技术支持体系、电话技术指导、远程支持及现场技术支持等方式向用户提供及时有效的技术支持。

技术支持网址

查阅技术支持网站上的技术资料：

- 企业网网址：**http://e.huawei.com**
- 运营商网址：**http://carrier.huawei.com**

获取华为技术支持

如果在设备维护或故障处理过程中，遇到难以确定或难以解决的问题，通过文档的指导仍然不能解决，请通过如下方式获取技术支持：

- 联系华为技术有限公司客户服务中心。

  中国区企业用户请通过以下方式联系华为：

  - 客户服务电话：400-822-9999
  - 客户服务邮箱：**ChinaEnterprise_TAC@huawei.com**

    企业网全球各地区客户服务热线可以通过以下网站查找：**企业用户全球服务热线**

    中国区运营商用户请通过以下方式联系我们：
  - 客户服务电话：400-830-2118
  - 客户服务邮箱：**support@huawei.com**

    运营商全球各地区客户服务热线可以通过以下网站查找：**运营商用户全球服务热线**
- 联系华为技术有限公司驻当地办事处的技术支持人员。

案例库与自助平台

如果您想进一步学习和交流：

- 访问**华为服务器信息自助服务平台**，获取相关服务器产品资料。
- 访问**华为服务器自助问答系统**，快速查询产品问题。
- 访问**华为企业互动论坛（服务器）**，进行学习交流。
- 参阅已有案例进行学习：**华为服务器案例库**。

10.1.6 联系华为的方法

华为技术有限公司为客户提供全方位的技术支持，用户可与就近的办事处联系，也可直接与公司总部联系。

华为技术有限公司

地址：深圳市龙岗区坂田华为总部办公楼

邮编：518129

网址：**http://enterprise.huawei.com/**

10.2 免责声明

- 本文档可能包含第三方信息、产品、服务、软件、组件、数据或内容（统称“第三方内容”）。华为不控制且不对第三方内容承担任何责任，包括但不限于准确性、兼容性、可靠性、可用性、合法性、适当性、性能、不侵权、更新状态等，除非本文档另有明确说明。在本文档中提及或引用任何第三方内容不代表华为对第三方内容的认可或保证。
- 用户若需要第三方许可，须通过合法途径获取第三方许可，除非本文档另有明确说明。