Atlas 200I DK A2 开发者套件
23.0.RC3

MindX SDK 应用开发指南

文档版本 01
发布日期 2023-11-16

华为技术有限公司

版权所有 © 华为技术有限公司 2023。 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暗示的声明或保证。

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

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

安全声明

漏洞声明

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该政策可参考华为公司官方网站的网址：https://www.huawei.com/cn/psirt/vul-response-process。
如企业客户须获取漏洞信息，请访问：https://securitybulletin.huawei.com/enterprise/cn/security-advisory。

目 录

1 学习向导.....1

2 使用 API 接口方式开发（C++）.....4

2.1 开发流程.....4
2.2 初始化与去初始化.....5
2.3 媒体数据处理.....5
2.3.1 通过接口进行图像处理（Image）.....5
2.3.1.1 图片解码.....5
2.3.1.2 图片编码.....7
2.3.1.3 抠图.....9
2.3.1.4 缩放.....11
2.3.1.5 补边.....13
2.3.1.6 抠图缩放.....15
2.3.1.7 抠图贴图.....17
2.3.2 通过张量方法进行图像处理（Tensor）.....20
2.3.3 视频编码.....20
2.3.4 视频解码.....23
2.4 模型推理.....25
2.5 异步调用.....27
2.6 编译运行.....30

3 使用 API 接口方式开发（Python）.....32

3.1 开发流程.....33
3.2 初始化.....34
3.3 媒体数据处理.....34
3.3.1 图片解码.....34
3.3.2 图片编码.....35
3.3.3 抠图.....37
3.3.4 缩放.....38
3.3.5 补边.....40
3.3.6 抠图缩放.....41
3.3.7 抠图贴图.....43
3.3.8 视频编码.....44
3.3.9 视频解码.....46

3.4 模型推理 48
3.5 运行 49
4 模型支持参考 51
4.1 模型支持列表 51
4.2 模型后处理配置参数 55
4.3 模型结构 67
5 API 参考（C++） 77
5.1 使用须知 77
5.2 多进程编写规范 78
5.3 头文件列表说明 82
5.4 数据结构及枚举类型 87
5.4.1 通用数据结构 87
5.4.1.1 ModelLoadOptV2 88
5.4.1.2 VideoDecodeConfig 90
5.4.1.3 VideoDecodeCallBack 93
5.4.1.4 VideoEncodeConfig 93
5.4.1.5 VideoEncodeCallBack 95
5.4.1.6 Rect 96
5.4.1.7 Size 96
5.4.1.8 Point 97
5.4.1.9 Color 97
5.4.1.10 Dim 98
5.4.1.11 TensorDType 99
5.4.1.12 VisionDataFormat 100
5.4.1.13 StreamFormat 100
5.4.1.14 Interpolation 101
5.4.1.15 BorderType 101
5.4.1.16 ImageFormat 102
5.4.1.17 RotateAngle 103
5.4.1.18 CropRoiConfig 104
5.4.1.19 ResizeConfig 104
5.4.1.20 MakeBorderConfig 105
5.4.1.21 BaseTensor 106
5.4.1.22 TensorDesc 107
5.4.1.23 ModelDesc 107
5.4.1.24 ModelDataset 108
5.4.1.25 DvppDataInfo 108
5.4.1.26 DvppImageInfo 109
5.4.1.27 DvppImageOutput 110
5.4.1.28 MemoryData 111
5.4.1.29 DetectBox 112
5.4.1.30 VdecConfig 113

5.4.1.31 PostImageInfo.... 116
5.4.1.32 KeyPointInfo.... 117
5.4.1.33 HungarianHandle.... 117
5.4.1.34 IOUMethod.... 118
5.4.1.35 EncodeH26xInfo.... 118
5.4.1.36 VencConfig.... 119
5.4.1.37 PostProcessorImageInfo.... 120
5.4.1.38 AspectRatioPostImageInfo.... 120
5.4.1.39 ObjectInfo.... 121
5.4.1.40 ClassInfo.... 122
5.4.1.41 SemanticSegInfo.... 122
5.4.1.42 TextsInfo.... 123
5.4.1.43 TextObjectInfo.... 123
5.4.1.44 AttributeInfo.... 124
5.4.1.45 ResizedImageInfo.... 125
5.4.1.46 CropRoiBox.... 125
5.4.1.47 DataFormat.... 126
5.4.1.48 DynamicType.... 126
5.4.1.49 ImagePreProcessInfo.... 127
5.4.1.50 CoorDim.... 128
5.4.1.51 TensorArrangementType.... 128
5.4.1.52 MxbasePixelFormat.... 129
5.4.1.53 MxbaseStreamFormat.... 129
5.4.1.54 DecodeH26xInfo.... 130
5.4.1.55 CropResizePasteConfig.... 130
5.4.1.56 ImageConstrainInfo.... 131
5.4.1.57 ImageSize.... 132
5.4.1.58 DynamicInfo.... 133
5.4.1.59 RoiBox.... 133
5.4.1.60 TextObjDetectInfo.... 134
5.4.1.61 TensorDataType.... 135
5.4.1.62 TrackFlag.... 135
5.4.1.63 MxbaseDvppChannelMode.... 135
5.4.1.64 DeviceContext.... 136
5.4.1.65 ResizeType.... 136
5.4.1.66 KeyPointDetectionInfo.... 137
5.4.1.67 AppGlobalCfg.... 138
5.4.1.68 ReduceDim.... 138
5.4.1.69 ReduceType.... 138
5.4.1.70 CmpOp.... 139
5.4.1.71 WarpAffineMode.... 139
5.4.1.72 WarpPerspectiveMode.... 140

5.4.1.73 PaddingMode....140
5.5 基础组件层....141
5.5.1 MemoryHelper....141
5.5.1.1 类说明....141
5.5.1.2 Malloc....141
5.5.1.3 Free....142
5.5.1.4 Memset....142
5.5.1.5 Memcpy....143
5.5.1.6 MxbsMalloc....144
5.5.1.7 MxbsFree....145
5.5.1.8 MxbsMemset....145
5.5.1.9 MxbsMemcpy....146
5.5.1.10 MxbsMallocAndCopy....147
5.5.1.11 SetMaxDataSize....147
5.5.1.12 CheckDataSize....148
5.5.1.13 CheckDataSizeAllowZero....148
5.5.1.14 MakeShared....149
5.5.2 DvppWrapper....149
5.5.2.1 类说明....150
5.5.2.2 DvppWrapper....150
5.5.2.3 Init....150
5.5.2.4 DeInit....151
5.5.2.5 InitVdec....151
5.5.2.6 DeInitVdec....152
5.5.2.7 DvppVdec....152
5.5.2.8 DvppVdecFlush....153
5.5.2.9 DvppPngDecode....153
5.5.2.10 DvppJpegEncode....154
5.5.2.11 DvppJpegDecode....155
5.5.2.12 DvppJpegPredictDecSize....156
5.5.2.13 VpcCrop....156
5.5.2.14 VpcBatchCrop....157
5.5.2.15 VpcBatchCropMN....158
5.5.2.16 VpcBatchCropResize....159
5.5.2.17 VpcCropAndPaste....161
5.5.2.18 VpcResize....162
5.5.2.19 VpcPadding....163
5.5.2.20 VpcPictureConstrainInfoCheck....164
5.5.2.21 InitVenc....164
5.5.2.22 DeInitVenc....165
5.5.2.23 DvppVenc....165
5.5.2.24 GetPictureDec....166

5.5.2.25 InitJpegEncodeChannel.....167
5.5.2.26 InitJpegDecodeChannel.....167
5.5.2.27 InitVpcChannel.....168
5.5.2.28 InitPngDecodeChannel.....168
5.5.3 ModelInferenceProcessor.....169
5.5.3.1 类说明.....169
5.5.3.2 ModelInferenceProcessor.....169
5.5.3.3 ~ModelInferenceProcessor.....169
5.5.3.4 Init.....170
5.5.3.5 DeInit.....170
5.5.3.6 ModelInference.....171
5.5.3.7 GetModelDesc.....171
5.5.3.8 GetDataFormat.....172
5.5.3.9 GetDynamicType.....172
5.5.3.10 GetInputDataType.....172
5.5.3.11 GetOutputDataType.....173
5.5.3.12 GetInputFormat.....173
5.5.3.13 GetOutputFormat.....173
5.5.3.14 GetDynamicImageSizes.....173
5.5.3.15 GetInputShape.....174
5.5.3.16 GetOutputShape.....174
5.5.3.17 GetDynamicBatch.....174
5.5.3.18 GetDynamicGearInfo.....174
5.5.4 Nms.....175
5.5.4.1 NmsSortByArea.....175
5.5.4.2 NmsSort.....176
5.5.4.3 CalcIou.....176
5.5.4.4 FilterByIou.....177
5.5.5 SimilarityTransform.....178
5.5.5.1 SimilarityTransform.....178
5.5.5.2 ~SimilarityTransform.....178
5.5.5.3 Transform.....178
5.5.6 WarpAffine.....179
5.5.6.1 WarpAffine.....179
5.5.6.2 ~WarpAffine.....179
5.5.6.3 Process.....179
5.5.7 Hungarian.....180
5.5.7.1 HungarianHandleInit.....180
5.5.7.2 HungarianSolve.....181
5.5.8 KalmanTracker.....181
5.5.8.1 KalmanTracker.....181
5.5.8.2 ~KalmanTracker.....182

5.5.8.3 Predict.....182
5.5.8.4 Update.....182
5.5.8.5 CvKalmanInit.....183
5.5.9 FastMath.....183
5.5.9.1 类说明.....183
5.5.9.2 FastMath.....183
5.5.9.3 ~FastMath.....183
5.5.9.4 exp.....184
5.5.9.5 sigmoid.....184
5.5.9.6 softmax.....185
5.5.9.7 sign.....185
5.5.9.8 FExp.....185
5.5.9.9 Sigmoid.....186
5.5.9.10 Softmax.....186
5.5.9.11 sign.....187
5.5.10 ConfigData.....187
5.5.10.1 类说明.....187
5.5.10.2 ConfigData.....187
5.5.10.3 ~ConfigData.....188
5.5.10.4 operator=.....188
5.5.10.5 SetJsonValue.....188
5.5.10.6 LoadLabels.....189
5.5.10.7 GetClassName.....189
5.5.10.8 GetFileValue.....190
5.5.10.9 GetFileValueWarn.....190
5.5.10.10 SetFileValue.....191
5.5.10.11 InitFile.....192
5.5.10.12 InitJson.....192
5.5.10.13 InitContent.....193
5.5.10.14 GetCfgJson.....193
5.5.11 ConfigUtil.....193
5.5.11.1 类说明.....193
5.5.11.2 LoadConfiguration.....194
5.5.12 DeviceManager.....194
5.5.12.1 类说明.....194
5.5.12.2 ~DeviceManager.....194
5.5.12.3 GetInstance.....195
5.5.12.4 InitDevices.....195
5.5.12.5 GetDevicesCount.....195
5.5.12.6 GetCurrentDevice.....196
5.5.12.7 SetDevice.....196
5.5.12.8 DestroyDevices.....197

5.5.12.9 SetDeviceSimple ... 197
5.5.12.10 IsInitDevices ... 198
5.5.12.11 CheckDeviceId ... 198
5.5.12.12 IsAscend310B ... 199
5.5.12.13 GetSocName ... 199
5.5.13 LineRegressionFit ... 200
5.5.13.1 类说明 ... 200
5.5.13.2 LineRegressionFit ... 200
5.5.13.3 ~LineRegressionFit ... 200
5.5.13.4 LRFunction ... 200
5.5.13.5 SetAlphaAndBeta ... 201
5.5.14 NpySort ... 201
5.5.14.1 类说明 ... 201
5.5.14.2 NpySort ... 201
5.5.14.3 ~NpySort ... 202
5.5.14.4 NpyArgQuickSort ... 202
5.5.14.5 NpyArgHeapSort ... 202
5.5.14.6 GetSortIdx ... 202
5.5.15 TensorBase ... 203
5.5.15.1 类说明 ... 203
5.5.15.2 CreateTensorBase ... 203
5.5.15.3 operator= ... 204
5.5.15.4 GetIndices ... 204
5.5.15.5 GetTensorDataTypeDesc ... 205
5.5.15.6 TensorBase ... 205
5.5.15.7 ~TensorBase ... 206
5.5.15.8 TensorBaseMalloc ... 206
5.5.15.9 TensorBaseCopy ... 207
5.5.15.10 GetTensorType ... 208
5.5.15.11 GetSize ... 208
5.5.15.12 GetByteSize ... 208
5.5.15.13 SetShape ... 209
5.5.15.14 GetShape ... 209
5.5.15.15 GetStrides ... 210
5.5.15.16 GetDeviceId ... 210
5.5.15.17 GetDataType ... 210
5.5.15.18 GetDataTypeSize ... 211
5.5.15.19 IsHost ... 211
5.5.15.20 IsDevice ... 211
5.5.15.21 GetBuffer ... 212
5.5.15.22 GetValue ... 213
5.5.15.23 ToDevice ... 213

5.5.15.24 ToDvpp 214
5.5.15.25 ToHost 214
5.5.15.26 BatchConcat 215
5.5.15.27 BatchStack 215
5.5.15.28 BatchVector 216
5.5.15.29 GetDesc 217
5.5.15.30 CheckTensorValid 217
5.5.15.31 SetValidRoi 217
5.5.15.32 GetValidRoi 218
5.5.16 TensorBuffer 218
5.5.16.1 类说明 218
5.5.16.2 TensorBuffer 218
5.5.16.3 ~TensorBuffer 219
5.5.16.4 IsDevice 219
5.5.16.5 IsHost 219
5.5.16.6 SetContext 220
5.5.16.7 TensorBufferMalloc 220
5.5.16.8 TensorBufferCopy 221
5.5.16.9 GetBufferCopyType 221
5.5.16.10 CheckCopyValid 222
5.5.16.11 CopyBetweenHost 223
5.5.16.12 CopyBetweenHostDevice 223
5.5.16.13 CopyBetweenSameDevice 224
5.5.16.14 CopyBetweenDiffDevice 224
5.5.17 TensorContext 225
5.5.17.1 类说明 225
5.5.17.2 TensorContext 225
5.5.17.3 ~TensorContext 225
5.5.17.4 GetInstance 226
5.5.17.5 SetContext 226
5.5.18 TensorShape 226
5.5.18.1 类说明 226
5.5.18.2 TensorShape 227
5.5.18.3 ~TensorShape 227
5.5.18.4 GetDims 227
5.5.18.5 SetShape 228
5.5.18.6 GetShape 228
5.5.18.7 GetSize 228
5.5.18.8 operator[] 229
5.5.18.9 SetValidRoi 229
5.5.18.10 GetValidRoi 230
5.5.19 Version 230

5.5.19.1 功能说明 230
5.5.19.2 GetCurrentSDKVersion 231
5.5.19.3 GetCurrentMajorVersion 231
5.5.19.4 GetCurrentMinorVersion 231
5.5.19.5 GetCurrentMicroVersion 232
5.5.19.6 GetSDKVersion 232
5.5.20 Log 232
5.5.20.1 类说明 233
5.5.20.2 getLogger 233
5.5.20.3 Init 234
5.5.20.4 InitWithoutCfg 234
5.5.20.5 Deinit 235
5.5.20.6 Debug 235
5.5.20.7 Info 235
5.5.20.8 Warn 236
5.5.20.9 Error 236
5.5.20.10 Fatal 237
5.5.20.11 Flush 237
5.5.20.12 SetLogParameters 237
5.5.20.13 LogRotateByTime 238
5.5.20.14 LogRotateByNumbers 238
5.5.20.15 UpdateFileMode 238
5.5.21 ErrorCode 239
5.5.21.1 类说明 239
5.5.21.2 GetAppErrCodeInfo 239
5.5.21.3 GetError 239
5.5.21.4 ConvertReturnCodeToLocal 240
5.5.21.5 GetErrorInfo 240
5.5.21.6 APP_ERROR 说明 241
5.6 初始化和去初始化 250
5.6.1 MxInit 250
5.6.2 MxDeInit 251
5.7 媒体数据处理 251
5.7.1 Image 251
5.7.1.1 类说明 252
5.7.1.2 Image 252
5.7.1.3 ~Image 253
5.7.1.4 operator = 253
5.7.1.5 GetDeviceId 254
5.7.1.6 GetOriginalData 254
5.7.1.7 SetImageOriginalSize 255
5.7.1.8 SetImageAlignedSize 255

5.7.1.9 GetData....256
5.7.1.10 GetDataSize....256
5.7.1.11 GetSize....257
5.7.1.12 GetOriginalSize....257
5.7.1.13 GetFormat....257
5.7.1.14 ToHost....258
5.7.1.15 ToDevice....258
5.7.1.16 ConvertToTensor....258
5.7.1.17 Serialize....259
5.7.1.18 Unserialize....260
5.7.1.19 DumpBuffer....261
5.7.1.20 TensorToImage....261
5.7.2 ImageProcessor....262
5.7.2.1 类说明....262
5.7.2.2 ImageProcessor....263
5.7.2.3 ~ImageProcessor....264
5.7.2.4 InitJpegEncodeChannel....264
5.7.2.5 InitJpegDecodeChannel....265
5.7.2.6 InitVpcChannel....265
5.7.2.7 InitPngDecodeChannel....266
5.7.2.8 Decode....266
5.7.2.9 Encode....268
5.7.2.10 Resize....269
5.7.2.11 Padding....271
5.7.2.12 Crop....273
5.7.2.13 CropResize....274
5.7.2.14 CropAndPaste....276
5.7.3 Tensor....278
5.7.3.1 类说明....278
5.7.3.2 Tensor....278
5.7.3.3 ~Tensor....280
5.7.3.4 operator =....280
5.7.3.5 operator ==....280
5.7.3.6 GetData....281
5.7.3.7 SetShape....281
5.7.3.8 GetShape....282
5.7.3.9 GetDataType....282
5.7.3.10 GetMemoryType....282
5.7.3.11 GetByteSize....283
5.7.3.12 GetDeviceId....283
5.7.3.13 ToDevice....283
5.7.3.14 ToHost....284

5.7.3.15 ToDvpp....284
5.7.3.16 IsEmpty....285
5.7.3.17 TensorFree....285
5.7.3.18 IsWithMargin....286
5.7.3.19 SetValidRoi....286
5.7.3.20 GetValidRoi....287
5.7.3.21 TensorMalloc....287
5.7.3.22 BatchConcat....288
5.7.3.23 Transpose....288
5.7.3.24 Clone....289
5.7.3.25 SetTensorValue....290
5.7.3.26 SetReferRect....291
5.7.3.27 GetReferRect....292
5.7.4 TensorDvpp....292
5.7.4.1 CvtColor....292
5.7.5 TensorFeatures....294
5.7.5.1 Sift....294
5.7.5.1.1 Sift....294
5.7.5.1.2 Init....295
5.7.5.1.3 DetectAndCompute....296
5.7.6 VideoEncoder....297
5.7.6.1 类说明....297
5.7.6.2 VideoEncoder....297
5.7.6.3 ~VideoEncoder....298
5.7.6.4 Encode....298
5.7.7 VideoDecoder....298
5.7.7.1 类说明....298
5.7.7.2 VideoDecoder....299
5.7.7.3 ~VideoDecoder....299
5.7.7.4 Decode....299
5.7.7.5 Flush....300
5.8 模型推理....301
5.8.1 Model....301
5.8.1.1 类说明....301
5.8.1.2 Model....301
5.8.1.3 ~Model....302
5.8.1.4 Infer....302
5.8.1.5 GetInputTensorNum....303
5.8.1.6 GetOutputTensorNum....303
5.8.1.7 GetInputTensorShape....304
5.8.1.8 GetOutputTensorShape....304
5.8.1.9 GetInputTensorDataType....305

5.8.1.10 GetOutputTensorDataType 305
5.8.1.11 GetInputFormat 306
5.8.1.12 GetDynamicGearInfo 306
5.9 异步调用 307
5.9.1 AscendStream 307
5.9.1.1 公共参数 307
5.9.1.2 AscendStream 307
5.9.1.3 ~AscendStream 307
5.9.1.4 CreateAscendStream 308
5.9.1.5 DestroyAscendStream 308
5.9.1.6 CreateChannel 308
5.9.1.7 GetChannel 309
5.9.1.8 Synchronize 309
5.9.1.9 DefaultStream 310
5.9.1.10 GetDeviceId 310
5.9.1.11 SetErrorCode 310
5.9.1.12 GetErrorCode 311
5.9.1.13 LaunchCallback 311
5.9.1.14 AddTensorRefPtr 312
5.10 模型后处理 312
5.10.1 模型后处理类参考（tensorinfer 框架） 312
5.10.1.1 ImagePreProcessInfo 312
5.10.1.1.1 ImagePreProcessInfo 312
5.10.1.2 PostProcessBase 313
5.10.1.2.1 PostProcessBase 313
5.10.1.2.2 ~PostProcessBase 313
5.10.1.2.3 operator= 313
5.10.1.2.4 Init 314
5.10.1.2.5 DeInit 314
5.10.1.2.6 GetCurrentVersion 315
5.10.1.2.7 LoadConfigDataAndLabelMap 315
5.10.1.2.8 LoadConfigData 316
5.10.1.2.9 CheckAndMoveTensors 316
5.10.1.2.10 GetBuffer 317
5.10.1.2.11 JudgeResizeType 317
5.10.1.3 ImagePostProcessBase 318
5.10.1.3.1 类说明 318
5.10.1.3.2 SetCropRoiBoxes 318
5.10.1.4 ObjectPostProcessBase 318
5.10.1.4.1 类功能 318
5.10.1.4.2 ObjectPostProcessBase 318
5.10.1.4.3 ~ObjectPostProcessBase 319

5.10.1.4.4 operator=.....319
5.10.1.4.5 Init.....319
5.10.1.4.6 DeInit.....320
5.10.1.4.7 Process.....320
5.10.1.4.8 CoordinatesReduction.....321
5.10.1.4.9 ResizeReduction.....322
5.10.1.4.10 LogObjectInfos.....323
5.10.1.4.11 GetSeparateScoreThresh.....323
5.10.1.4.12 GetObjectConfigData.....323
5.10.1.5 ClassPostProcessBase.....324
5.10.1.5.1 ClassPostProcessBase.....324
5.10.1.5.2 ~ClassPostProcessBase.....324
5.10.1.5.3 operator=.....324
5.10.1.5.4 Init.....325
5.10.1.5.5 DeInit.....325
5.10.1.5.6 Process.....326
5.10.1.6 SemanticSegPostProcessBase.....326
5.10.1.6.1 类功能.....326
5.10.1.6.2 SemanticSegPostProcessBase.....326
5.10.1.6.3 ~SemanticSegPostProcessBase.....327
5.10.1.6.4 operator=.....327
5.10.1.6.5 Init.....327
5.10.1.6.6 DeInit.....328
5.10.1.6.7 Process.....328
5.10.1.6.8 CoordinatesReduction.....329
5.10.1.6.9 GetSemanticSegConfigData.....329
5.10.1.7 TextGenerationPostProcessBase.....329
5.10.1.7.1 TextGenerationPostProcessBase.....330
5.10.1.7.2 ~TextGenerationPostProcessBase.....330
5.10.1.7.3 operator=.....330
5.10.1.7.4 Init.....330
5.10.1.7.5 DeInit.....331
5.10.1.7.6 Process.....331
5.10.1.8 TextObjectPostProcessBase.....332
5.10.1.8.1 类功能.....332
5.10.1.8.2 TextObjectPostProcessBase.....332
5.10.1.8.3 ~TextObjectPostProcessBase.....332
5.10.1.8.4 operator =.....333
5.10.1.8.5 Init.....333
5.10.1.8.6 DeInit.....333
5.10.1.8.7 Process.....334
5.10.1.8.8 ResizeReduction.....334

5.10.1.8.9 FixCoords 335
5.10.1.9 KeypointPostProcessBase 335
5.10.1.9.1 KeypointPostProcessBase 335
5.10.1.9.2 ~KeypointPostProcessBase 336
5.10.1.9.3 operator= 336
5.10.1.9.4 Init 336
5.10.1.9.5 DeInit 337
5.10.1.9.6 Process 337
5.10.1.9.7 LogKeyPointInfos 338
5.10.1.9.8 GetSeparateScoreThresh 338
5.10.1.10 postprocess 338
5.10.1.10.1 Yolov3PostProcess 338
5.10.1.10.2 SsdMobilenetv1FpnPostProcess 339
5.10.1.10.3 Ssdvgg16PostProcess 339
5.10.1.10.4 FasterRcnnPostProcess 339
5.10.1.10.5 MaskRcnnMindsporePost 339
5.10.1.10.6 SsdMobilenetFpnMindsporePost 340
5.10.1.10.7 Resnet50PostProcess 340
5.10.1.10.8 Deeplabv3Post 340
5.10.1.10.9 UNetMindSporePostProcess 340
5.10.1.10.10 TransformerPostProcess 341
5.10.1.10.11 CrnnPostProcess 341
5.10.1.10.12 CtpnPostProcess 341
5.10.1.10.13 PSENetPostProcess 341
5.10.1.10.14 OpenPosePostProcess 341
5.10.1.10.15 HigherHRnetPostProcess 342
5.10.1.10.16 RetinaNetPostProcess 342
5.10.2 模型后处理类参考（modelinfer 框架） 342
5.10.2.1 Resnet50PostProcessor 342
5.10.2.1.1 类说明 342
5.10.2.1.2 Init 342
5.10.2.1.3 DeInit 343
5.10.2.1.4 Process 343
5.10.2.2 ModelPostProcessorBase 344
5.10.2.2.1 类说明 344
5.10.2.2.2 Init 344
5.10.2.2.3 DeInit 345
5.10.2.2.4 Process 345
5.10.2.2.5 MemoryDataToHost 346
5.10.2.2.6 GetLabelName 346
5.10.2.2.7 LoadConfigDataAndLabelMap 347
5.10.2.2.8 GetModelTensorsShape 347

5.10.2.2.9 CheckModelCompatibility....348
5.10.2.3 ObjectPostProcessorBase....348
5.10.2.3.1 类说明....348
5.10.2.3.2 Init....348
5.10.2.3.3 DeInit....349
5.10.2.3.4 Process....349
5.10.2.3.5 SetAspectRatioImageInfo....350
5.10.2.3.6 ObjectDetectionOutput....350
5.10.2.3.7 CoordinatesReduction....351
5.10.2.3.8 GetSeparateScoreThresh....351
5.10.2.3.9 NmsSort....352
5.10.2.4 FasterRcnnPostProcessor....352
5.10.2.4.1 类说明....352
5.10.2.4.2 Init....352
5.10.2.4.3 DeInit....353
5.10.2.4.4 Process....353
5.10.2.5 Yolov3PostProcessor....354
5.10.2.5.1 类说明....354
5.10.2.5.2 Init....354
5.10.2.5.3 DeInit....355
5.10.2.5.4 Process....355
5.10.2.5.5 SetPoolSize....356
5.10.2.6 CtpnPostProcessor....356
5.10.2.6.1 类说明....356
5.10.2.6.2 CtpnPostProcessor....356
5.10.2.6.3 ~CtpnPostProcessor....357
5.10.2.6.4 Init....357
5.10.2.6.5 DeInit....357
5.10.2.6.6 Process....358

6 API 参考（Python）....359

6.1 使用须知....359
6.2 依赖列表说明....360
6.3 Python 枚举类型及数据类....360
6.3.1 通用枚举类型/数据类....360
6.3.1.1 基本数据枚举类型....360
6.3.1.2 ModelLoadType 模型加载方式枚举类型....361
6.3.1.3 ModelType 模型枚举类型....361
6.3.1.4 VisionDataFormat 图像数据格式排布枚举类型....362
6.3.1.5 image_format 图片格式枚举类型....362
6.3.1.6 Resize 缩放枚举类型....363
6.3.1.7 Interpolation 缩放方式格式枚举类型....363
6.3.1.8 BorderType 补边方式枚举类型....363

6.3.1.9 StreamFormat 视频流数据格式枚举类型 364
6.3.1.10 ModelLoadOptV2 类 364
6.3.1.11 AttributeInfo 类 366
6.3.1.12 ClassInfo 类 366
6.3.1.13 ObjectInfo 类 366
6.3.1.14 ResizedImageInfo 类 367
6.3.1.15 SemanticSegInfo 类 367
6.3.1.16 TextObjectInfo 类 367
6.3.1.17 TextsInfo 类 368
6.3.1.18 KeyPointDetectionInfo 类 368
6.3.1.19 ResizeType 类 368
6.3.1.20 dtype 类 369
6.3.1.21 image.format 类 369
6.3.1.22 type 类 369
6.3.1.23 BufferInput 类 370
6.3.1.24 MetadataInput 类 370
6.3.1.25 DataOutput 类 370
6.3.1.26 BufferOutput 类 371
6.3.1.27 MetadataOutput 类 371
6.3.1.28 MxDataInput 类 371
6.3.1.29 MxDataOutput 类 371
6.3.1.30 Color 类 372
6.3.1.31 Dim 类 372
6.3.1.32 Rect 类 372
6.3.1.33 Size 类 372
6.3.1.34 Point 类 372
6.3.1.35 VideoDecodeConfig 类 373
6.3.1.36 VideoEncodeConfig 类 375
6.3.2 流程编排数据类 376
6.3.2.1 RoiBox 376
6.3.2.2 RoiBoxVector 377
6.3.2.3 MxDataInput 377
6.3.2.4 MxDataOutput 378
6.3.2.5 MxProtobufIn 378
6.3.2.6 MxProtobufOut 379
6.3.2.7 MxDataInputExt 379
6.3.2.8 MxMetadataInput 380
6.3.2.9 MxMetadataOutput 380
6.3.2.10 MxBufferAndMetadataOutput 381
6.3.2.11 StringVector 381
6.3.2.12 InProtobufVector 382
6.3.2.13 OutProtobufVector 382

6.3.2.14 MetadataInputVector ... 382
6.3.2.15 MetadataOutputVector ... 383
6.3.2.16 MxBufferInput ... 383
6.3.2.17 MxBufferOutput ... 384
6.4 初始化 ... 384
6.4.1 mx_init() ... 384
6.5 媒体数据处理 ... 384
6.5.1 Tensor 类 ... 384
6.5.1.1 Tensor ... 384
6.5.1.2 to_device ... 385
6.5.1.3 to_host ... 386
6.5.1.4 Tensor 类的属性列表 ... 386
6.5.2 Image 类 ... 386
6.5.2.1 Image ... 386
6.5.2.2 to_tensor ... 388
6.5.2.3 get_tensor ... 388
6.5.2.4 get_original_tensor ... 388
6.5.2.5 to_device ... 389
6.5.2.6 to_host ... 389
6.5.2.7 serialize ... 389
6.5.2.8 unserialize ... 390
6.5.2.9 dump_buffer ... 390
6.5.2.10 Image 类的属性列表 ... 391
6.5.3 ImageProcessor 类 ... 391
6.5.3.1 类说明 ... 391
6.5.3.2 ImageProcessor(deviceId=0) ... 391
6.5.3.3 decode(inputPath, decodeFormat) ... 391
6.5.3.4 encode(inputImage, savePath, encodeLevel) ... 392
6.5.3.5 resize(inputImage, resize, interpolation) ... 392
6.5.3.6 padding(inputImage, padDim, color, borderType) ... 393
6.5.3.7 crop(inputImage, cropRectVec) ... 394
6.5.3.8 crop(inputImageVec, cropRectVec) ... 394
6.5.3.9 crop_resize(inputImage, cropResizeVec) ... 395
6.5.3.10 crop_paste(inputImage, cropPasteRect, pastedImage) ... 395
6.5.4 VdecCallBacker 类 ... 396
6.5.4.1 类说明 ... 396
6.5.4.2 回调函数格式 ... 396
6.5.4.3 VdecCallBacker() ... 396
6.5.4.4 registerVdecCallBack(callback_func) ... 397
6.5.5 VideoDecoder 类 ... 397
6.5.5.1 类说明 ... 397
6.5.5.2 VideoDecoder(pyVdecConfig, pyVdecCallBacker, deviceId, channelId) ... 397

6.5.5.3 decode(inputData, frameId) 398
6.5.6 VencCallBacker 类 398
6.5.6.1 类说明 398
6.5.6.2 回调函数格式 399
6.5.6.3 VencCallBacker() 399
6.5.6.4 registerVencCallBack(callback_func) 400
6.5.7 VideoEncoder 类 400
6.5.7.1 类说明 400
6.5.7.2 VideoEncoder(pyVencConfig, pyVencCallBacker, deviceId, channelId) 400
6.5.7.3 encode(inputImage, frameId) 401
6.5.8 DeviceMemory 类 401
6.5.8.1 类说明 401
6.5.8.2 DeviceMemory(int size) 401
6.5.8.3 get_data() 402
6.5.9 dvpp 模块 402
6.5.9.1 模块说明 402
6.5.9.2 read_image(inputPath, deviceId, decodeFormat) 402
6.5.9.3 resize(inputImage, resize, interpolation) 403
6.5.10 batch_concat(inputs) 403
6.5.11 image(inputPath, deviceId, decodeFormat) 404
6.5.12 resize_info(image, resize_height, resize_width, resize_type) 404
6.5.13 bytes_to_ptr(data) 405
6.5.14 transpose(input_tensor, axes) 405
6.6 模型推理 406
6.6.1 Model 类 406
6.6.1.1 Model(modelPath, deviceId=0) 406
6.6.1.2 Model(option, deviceId) 406
6.6.1.3 infer(tensorList) 407
6.6.1.4 infer(tensor, args) 407
6.6.1.5 input_dtype(index) 407
6.6.1.6 output_dtype(index) 408
6.6.1.7 input_shape(index) 408
6.6.1.8 output_shape(index) 408
6.6.1.9 model_gear() 409
6.6.1.10 Model 类的属性列表 409
6.6.2 model(modelPath, deviceId) 409
6.7 模型后处理 410
6.7.1 分类后处理 410
6.7.2 目标检测后处理 411
6.7.3 分割后处理 412
6.7.4 文本生成后处理 413
6.7.5 文本检测后处理 415

6.7.6 关键点检测后处理 416
6.8 日志输出 417
6.8.1 log 类函数集 417
7 常用操作 418
7.1 模型后处理开发 418
7.1.1 模型后处理（tensorinfer 框架） 418
7.1.1.1 模型后处理介绍 418
7.1.1.2 后处理类开发步骤 419
7.1.1.3 编译 421
7.1.2 模型后处理（modelinfer 框架） 422
7.1.2.1 模型后处理介绍 422
7.1.2.2 后处理类开发步骤 423
7.1.2.3 输入数据获取 424
7.1.2.4 输出结果发送 424
7.1.2.5 编译 424
7.2 辅助工具 425
7.2.1 一键式运维数据收集工具 425
7.2.1.1 工具介绍 425
7.2.1.2 工具使用 426
7.2.2 单插件测试工具 428
7.2.2.1 工具介绍 428
7.2.2.2 工具使用 428
7.2.2.3 单插件测试样例 428
7.2.3 插件信息搜集工具 430
7.2.3.1 工具介绍 430
7.2.3.2 工具使用 430
7.2.4 插件性能统计工具 431
7.2.4.1 工具介绍 431
7.2.4.2 配置项介绍 431
7.2.4.3 使用步骤 433
7.2.4.4 观测结果 434
8 FAQ 438
8.1 程序执行出现 libascendcl.so 报错 438
8.2 使用 yum 命令出现 No module named 'dnf'报错 439
8.3 使用 yum 命令出现 No module named '_conf'错误 439
8.4 执行命令查看版本号时，出现 Permission denied 报错 439
8.5 openEuler 操作系统系统命令 yum、cmake 不可用 440
8.6 源码方式安装完 Python 3.9.2 后，pip3 工具无法使用 440
8.7 在服务端接收请求或执行证书导入脚本时，KMC 初始化随机数阻塞 441
8.8 使用 mxVision 时，出现 can not find the element factory : mxpi_xxxpostprocessor 报错 441
8.9 编译时出现 libgfortran.so.4 报错 442
8.10 在 Kunpeng 服务器中运行多路视频分析，出现 CPU 资源被占满，导致业务堆积、断流的情况 442

8.11 调用 API 参考（Python）视频解码/编码接口结束时出现 core dumped 现象 443
8.12 在 Atlas 500 A2 智能小站默认配置下，无法安装 mxVision 443
8.13 执行 run.sh 脚本时报错 cannot find factory: mxpi_classpostprocessor 444
8.14 尝试打印 Device 侧的 Tensor、Image 内存数据时，发生 coredump 444
8.15 使用 mxVersion 安装时，出现 libpython3.9.so.1.0 报错 445
8.16 mxpi_rtspsrc 拉流插件运行偶现 BUS ERROR：XXX streamming stopped, reason error (-5) 错误 445

A 安装 7.3.0 版本 gcc 447

B Python 依赖安装参考 449

C 日志配置文件 450

D Metadata proto 文件 452

E 公网地址说明 459

F 修订记录 460

1 学习向导

使用导引

mxVision用户指南用于引导开发者基于现有模型，通过mxVision API接口开发方式或流程编排方式实现目标识别、图像分类等功能。

您可以了解到以下内容：

- mxVision软件架构、基本概念以及各开发方式对应的使用流程。
- 如何使用mxVision API接口进行应用开发，如何通过流程编排插件实现应用。

拥有C/C++、Python语言开发能力，对推理应用开发有一定了解的开发者，可更好的理解该产品。

初次使用，建议先了解基本概念，选择适合实际业务，理解习惯的开发方式，再通过对应的章节具体学习开发实现mxVision应用。

基本概念

表 1-1 概念介绍

概念	描述
同步/异步	本文中提及的同步、异步是站在调用者和执行者的角度，在当前场景下，若在调用接口后不等待Device执行完成再返回，则表示调度是异步的；若在调用接口后需等待Device执行完成再返回，则表示调度是同步的。
Host	Host指与Device相连接的x86_64服务器、Arm服务器，会利用Device提供的NN（Neural-Network）计算能力，完成业务。
Device	Device指安装了昇腾AI处理器的硬件设备，利用PCIe接口与Host侧连接，为Host提供NN计算能力。若存在多个Device，多个Device之间的内存资源不能共享。
Stream	Stream用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在Device上执行。

概念	描述
AIPP	AIPP（Artificial Intelligence Pre-Processing）用于在AI Core上完成图像预处理，包括色域转换（转换图像格式）和抠图（指定抠图起始点，抠出神经网络需要大小的图片）等。 AIPP区分为静态AIPP和动态AIPP。您只能选择静态AIPP或动态AIPP方式来处理图片，不能同时配置静态AIPP和动态AIPP两种方式。 ● 静态AIPP：模型转换时设置AIPP模式为静态，同时设置AIPP参数，模型生成后，AIPP参数值被保存在离线模型（*.om）中，每次模型推理过程采用固定的AIPP预处理参数（无法修改）。 如果使用静态AIPP方式，多Batch情况下共用同一份AIPP参数。 ● 动态AIPP：模型转换时设置AIPP模式为动态，每次模型推理前，根据需求，在执行模型前设置动态AIPP参数值，然后在模型执行时可使用不同的AIPP参数。 如果使用动态AIPP方式，多Batch可使用不同的AIPP参数。
动态Batch/动态分辨率	在某些场景下，模型每次输入的batch size或分辨率是不固定的，如检测出目标后再执行目标识别网络，由于目标个数不固定导致目标识别网络输入BatchSize不固定。 ● 动态Batch：用户执行推理时，其batch size是动态可变的。 ● 动态分辨率: 用户执行推理时，每张图片的分辨率H * W是动态可变的。
动态维度（ND格式）	为了支持Transformer等网络在输入格式的维度不确定的场景，需要支持ND格式下任意维度的动态设置。 ND表示支持任意格式，当前N ≤ 4。
通道	在RGB色彩模式下，图像通道就是指单独的红色R、绿色G、蓝色B部分。也就是说，一幅完整的图像，是由红色绿色蓝色三个通道组成的，它们共同作用产生了完整的图像。同样在HSV色系中指的是色调H，饱和度S，亮度V三个通道。
DVPP	DVPP（Digital Video Pre-Processing）是昇腾AI处理器内置的图像处理单元，通过媒体数据处理接口提供强大的媒体处理硬加速能力，主要功能包括缩放、抠图、格式转换、图片编解码、视频编解码等。
VPC	VPC（Vision Preprocessing Core）负责图像处理功能，支持对图片做抠图、缩放、格式转换等操作。
EULA	End User License Agreement，华为企业业务最终用户许可协议。

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

2 使用 API 接口方式开发（C++）

2.1 开发流程
2.2 初始化与去初始化
2.3 媒体数据处理
2.4 模型推理
2.5 异步调用
2.6 编译运行

2.1 开发流程

关键步骤说明

1. 使用mxVision接口进行应用开发，需首先进行初始化才能进行后续处理，在整体应用流程完成后还需进行去初始化，否则可能会导致后续系统内部资源初始化出错，进而导致其它业务异常，具体请参见**2.2 初始化与去初始化**。
2. 媒体数据处理/模型推理。
   a. 媒体数据处理功能，可实现图像编解码、抠图、缩放、补边、色域转换等操作，具体请参见**2.3 媒体数据处理**。
   b. 模型推理，可实现如目标识别，图像分类等应用，环节可参考以下几个步骤，具体请参见**2.4 模型推理**。
      i. 模型推理前，需准备合适的om模型。
      ii. 模型加载，将准备好的模型通过文件或内存方式加载到系统中，具体请参见**2.4 模型推理**。
      iii. （可选）媒体数据处理，模型推理前，可对输入的媒体数据进行处理，如图片解码、抠图、缩放、补边等操作。
      iv. 模型执行，使用模型实现图片分类、目标识别等功能。
      v. （可选）模型后处理，对模型推理输出的数据进行后处理操作，此处根据用户的实际需求来处理推理结果，如整理为可理解阅读的具体结果。模型后处理支持通过三种方式进行：
         - （推荐）自行选择模型后处理方式。

- 通过调用mxVision接口进行，具体操作请参见**5.10 模型后处理**。
- 在已有模型的基础上进行二次开发，具体请参见**7.1 模型后处理开发**。

3. 调用MxDeInit()接口进行去初始化，具体请参见**2.2 初始化与去初始化**。

2.2 初始化与去初始化

功能介绍

- mxVision应用开发初始化接口，在代码调用相关接口之前，需调用全局初始化函数**MxInit()**申请设备资源和日志资源。
- 如需配置全局变量（如调整VPC通道资源池数量），可调用**MxInit(const AppGlobalCfg &globalCfg)**接口，通过传入配置参数进行调整。

mxVision接口全部执行完毕后，用户需调用**MxDeInit()**接口对初始化的全局资源进行去初始化操作。

示例代码

以下为各初始化方法的代码示例，不可以直接拷贝编译运行，仅供参考。

- 全局初始化，加载设备资源和日志资源。

```
APP_ERROR ret = MxInit();
// 执行全局初始化后即可正常调用mxVision接口
...
// mxVision接口全部执行完毕后，调用去初始化接口释放全局资源
ret = MxDeInit();
```

- 全局初始化，加载全局变量。

```
MxBase::AppGlobalCfg globalCfg;
globalCfg.vpcChnNum = 48;
APP_ERROR ret = MxInit(globalCfg);
...
// mxVision接口全部执行完毕后，调用去初始化接口释放全局资源
ret = MxDeInit();
```

2.3 媒体数据处理

2.3.1 通过接口进行图像处理（Image）

2.3.1.1 图片解码

功能介绍

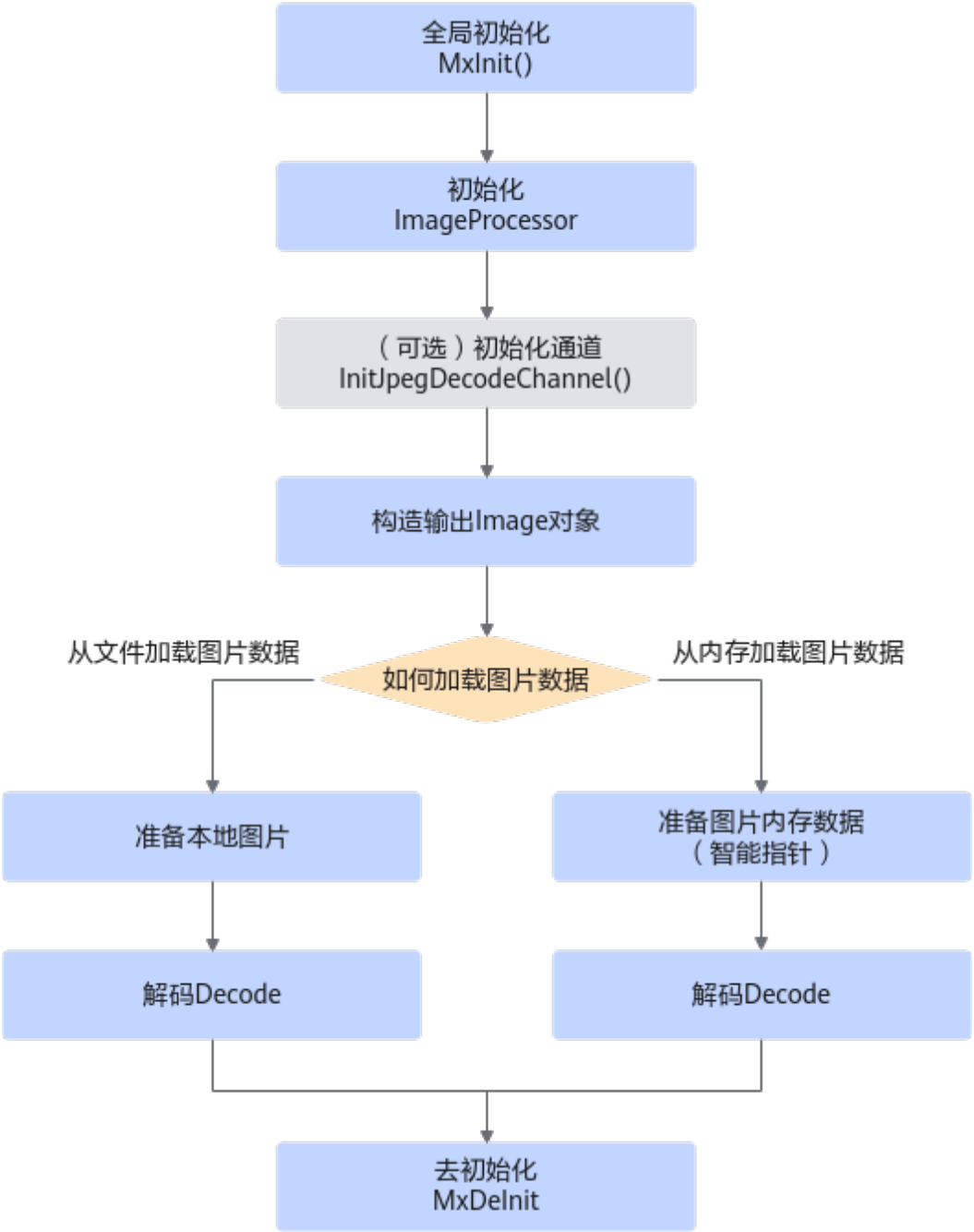
通过对输入图片数据进行解码处理，将本地图片或图片数据转换为Image类，用于后续前处理和推理业务，类型目前支持JPEG和PNG格式。

接口调用流程

用户需准备好本地待解码的图片文件或待解码的图片数据，初始化ImageProcessor类，构造输出的Image对象，通过调用ImageProcessor类的**Decode**接口解码结果。

图片解码调用流程参考如下：

图 2-1 图片解码接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor构造及初始化。

   用户构造ImageProcessor对象，在构造完成后可以调用**InitJpegDecodeChannel()**初始化通道，若不选择调用该接口，ImageProcessor将会在执行解码前自动进行通道初始化。

3. 构造输出的Image对象。
   使用构造函数**Image()**存放待解码输出的图片对象。
4. 准备数据。
   根据业务需求选择**从文件加载图片**或**从内存加载图片数据**。
5. 调用**Decode()**接口对图片进行解码。
   用户需根据输如的不同数据类型，调用对应的Decode接口。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);
    //（可选）初始化解码通道
    imageProcessor.InitJpegDecodeChannel();
    //图像解码
    //解码后的图像类
    Image decodedImage;
    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage, ImageFormat::YUV_SP_420);
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.2 图片编码

功能介绍

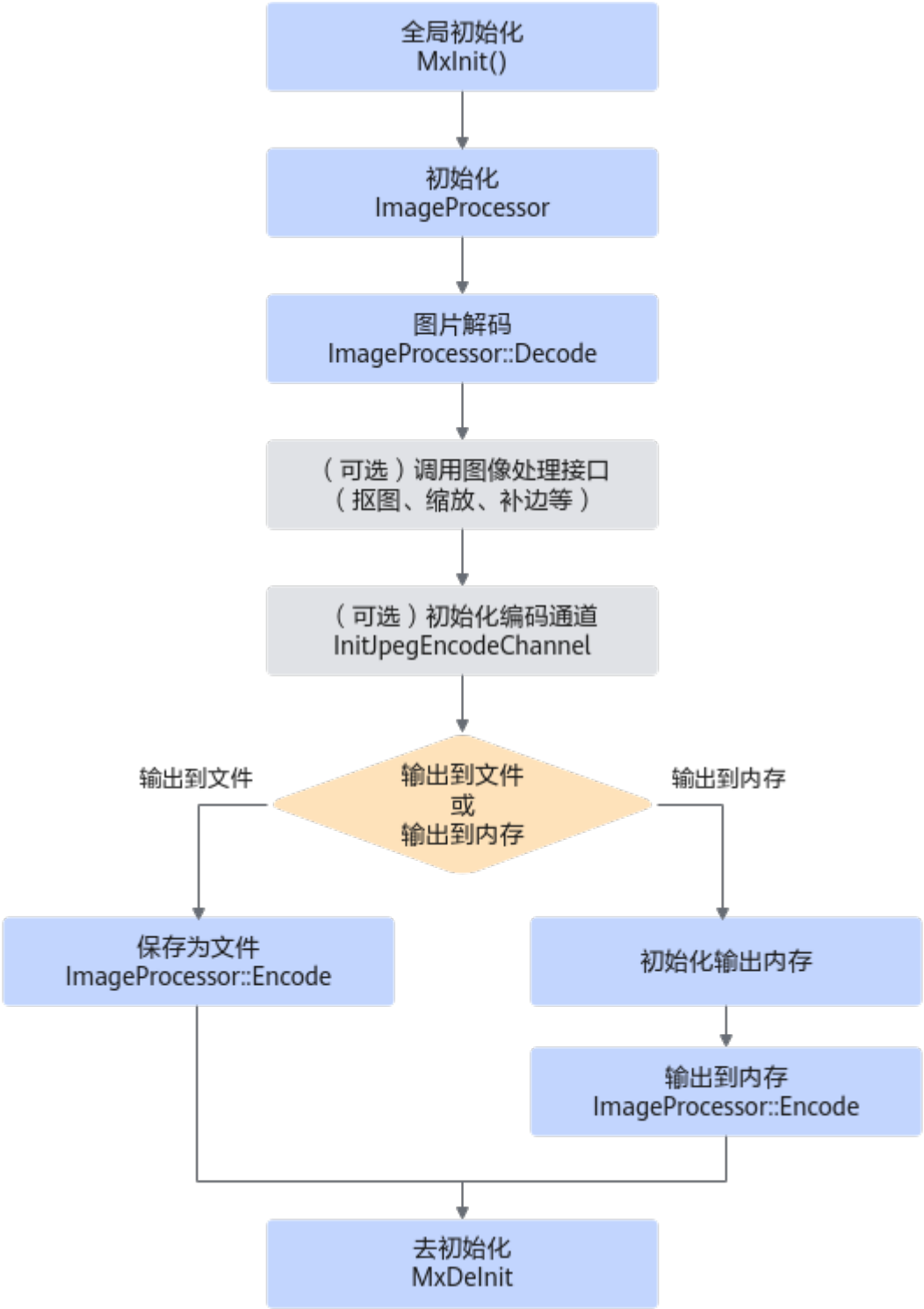
将接口输出的Image对象编码为JPG格式的图片内存或保存到指定的图片路径。

接口调用流程

用户需提前准备好待编码的Image对象，可以通过图片解码以及图像处理接口（抠图、缩放、补边等操作对应接口）输出，输出到本地图片或者内存中。

图片编码调用流程参考如下：

图 2-2 图片编码接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor构造及初始化。

   用户构造ImageProcessor对象，之后可以调用**InitJpegEncodeChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行编码前自动进行通道初始化。

3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可编码的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要编码的Image对象。

4. 调用**Encode()**接口对输入图片进行编码。

   用户需根据实际业务情况输出图片数据，选择**输出到文件**或**输出到内存**。

5. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //图像解码/图像处理操作生成Image
    //解码后的图像类
    Image decodedImage;
    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }
    //图像处理操作后图像类
    Image resizedImage;
    //图像处理操作（缩放）
    ret = imageProcessor.Resize(decodedImage, Size(416, 416), resizedImage,
Interpolation::HUAWEI_HIGH_ORDER_FILTER);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //（可选）初始化编码通道JpegEncodeChnConfig jpegEncodeChnConfig;
    JpegEncodeChnConfig.maxPicWidth = 4096;
    JpegEncodeChnConfig.maxPicHeight = 4096;
    imageProcessor.InitJpegEncodeChannel(JpegEncodeChnConfig);

    //图像编码
    ret = imageProcessor.Encode(resizedImage,"encode.jpg");
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.3 抠图

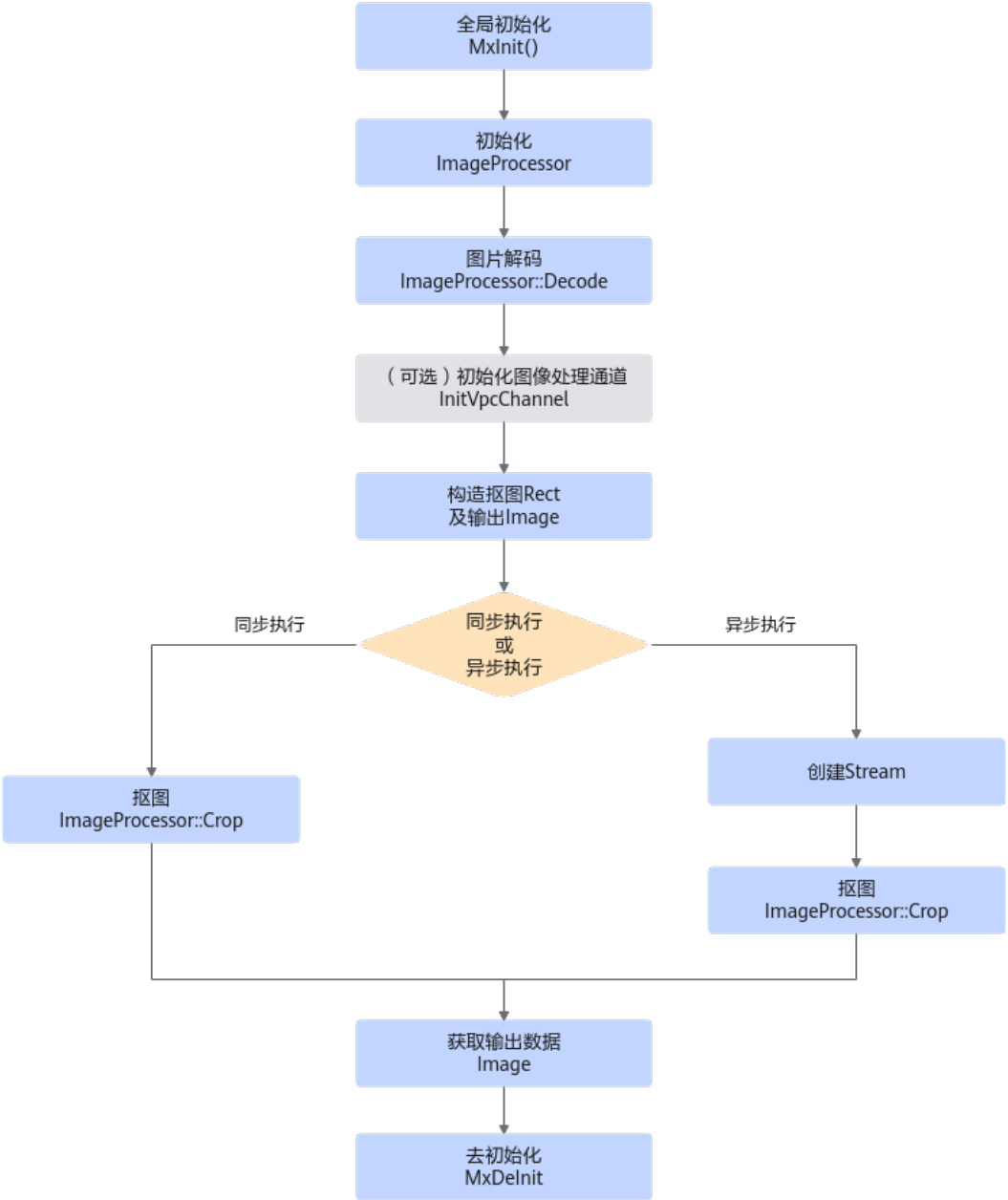
功能介绍

对输入的图像进行抠图操作，输出到Image对象中。

接口调用流程

使用抠图接口前，用户需准备好被抠图的图片对象。

图 2-3 图像处理（抠图）接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，之后可以调用**InitVpcChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行抠图前自动进行通道初始化。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要抠图的Image对象。
4. 构造抠图Rect以及输出Image。

   根据业务要求，选择一对多、一对一或多对多的抠图方式，构造对应的输入和输出Rect和Image。

5. 选择执行方式进行抠图，请根据实际业务选择同步抠图方式或异步抠图方式。
   a. 同步执行。
      i. 不创建Stream，将输入图片及其他参数传入**Crop()**接口，获取抠图结果。
   b. 异步执行。
      i. 创建Stream，具体请参见**2.5 异步调用**。
      ii. 将输入图片、已创建的Stream及其他参数传入Crop()接口，获取抠图结果。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //（可选）初始化图像处理通道
    imageProcessor.InitVpcChannel();

    //图像解码操作生成Image
    //解码后的图像类
    Image decodedImage;
    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //执行Crop
    //抠图图像类
    Image cropImage;
    //抠图坐标信息
    Rect cropRect {0, 0, 640, 512};
    //抠图操作
    ret = imageProcessor.Crop(decodeImage, cropRect, cropImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.4 缩放

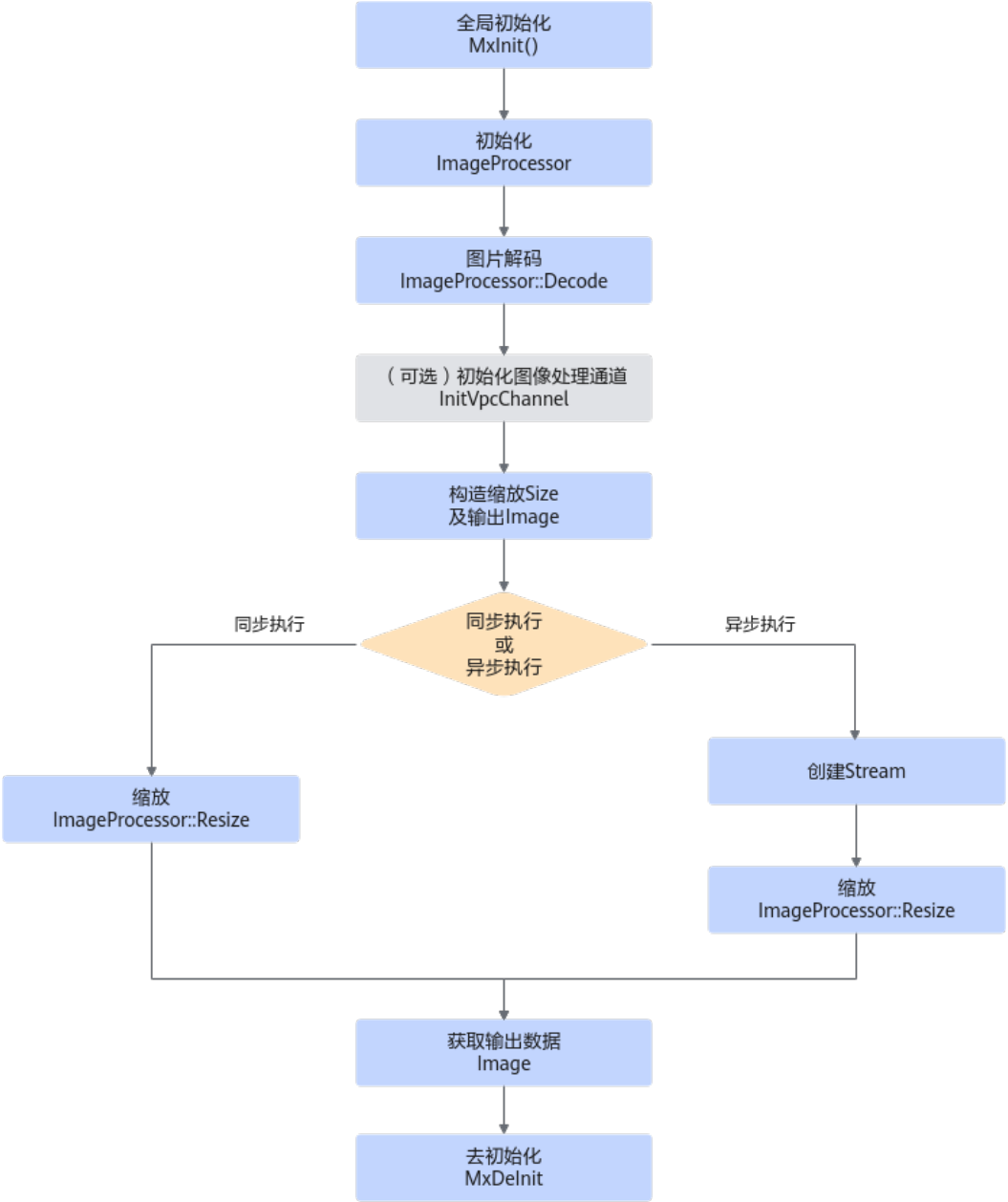
功能介绍

对输入的图像进行缩放操作，输出到Image对象中。

接口调用流程

使用缩放接口前，用户需准备好被缩放的图片对象。

图 2-4 图像处理（缩放）接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，之后可以调用**InitVpcChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行缩放前自动进行通道初始化。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可缩放的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要缩放的Image对象。
4. 构造缩放参数和输出Image。

   根据业务要求，构造对应的输入和输出Size和Image。

5. 选择执行方式进行抠图。
   a. 同步执行。
      i. 不创建Stream，将输入图片及其他参数传入**Resize()**接口，获取缩放结果。
   b. 异步执行。
      i. 创建Stream，具体请参见**2.5 异步调用**。
      ii. 将输入图片、已创建的Stream及其他参数传入Resize()接口，获取缩放结果。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //图像解码生成Image
    //解码后的图像类
    Image decodedImage;

    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //（可选）初始化图像处理通道
    imageProcessor.InitVpcChannel();

    //执行缩放
    //缩放操作后图像类
    Image resizedImage;

    //缩放尺寸
    Size size(416, 416);

    //缩放操作
    ret = imageProcessor.Resize(decodedImage, size, resizedImage,
Interpolation::HUAWEI_HIGH_ORDER_FILTER);
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.5 补边

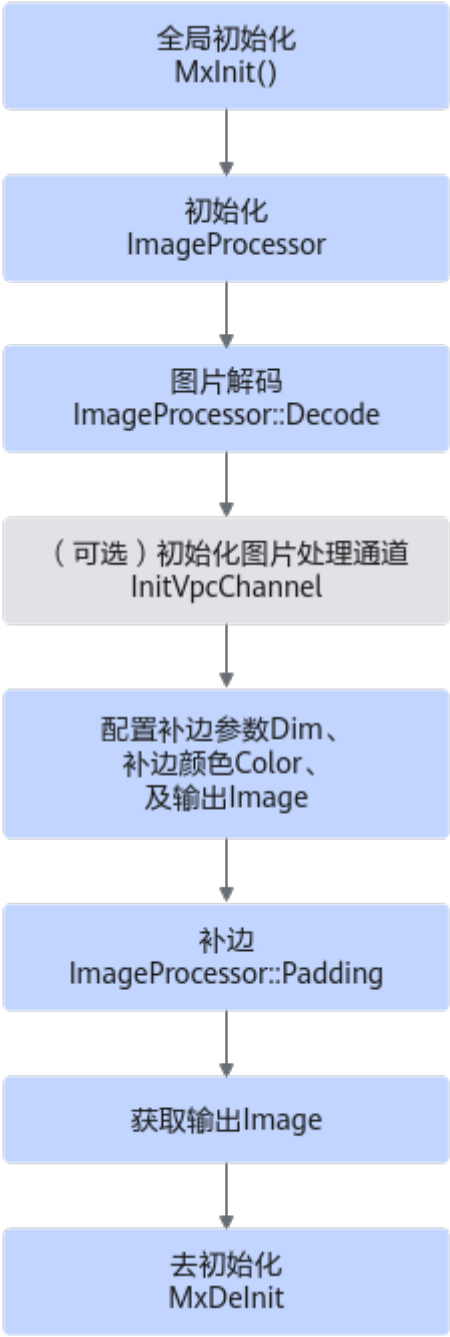
功能介绍

对输入的图像进行补边操作，输出到Image对象中。

接口调用流程

使用补边接口前，用户需准备好被补边的图片对象。

图 2-5 图像处理（补边）接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，之后可以调用**InitVpcChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行补边前自动进行通道初始化。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可补边的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要补边的Image对象。

4. 构造补边参数。

   根据业务要求，构造对应的输入的“padDim”（表示补边上下左右的像素值）、“Color”（补边颜色）和输出的Image。

5. 调用**Padding()**接口对输入图像进行补边。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //图像解码生成Image
    //解码后的图像类
    Image decodedImage;

    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //（可选）初始化图像处理通道
    imageProcessor.InitVpcChannel();

    //执行补边
    //补边操作后图像类
    Image paddingImage;

    //构造补边参数
    Dim padDim(0, 0, 240, 240);
    Color color(0, 0, 0);

    // Padding操作
    ret = imageProcessor.Padding(decodedImage, padDim, color, BorderType::BORDER_CONSTANT,
paddingImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.6 抠图缩放

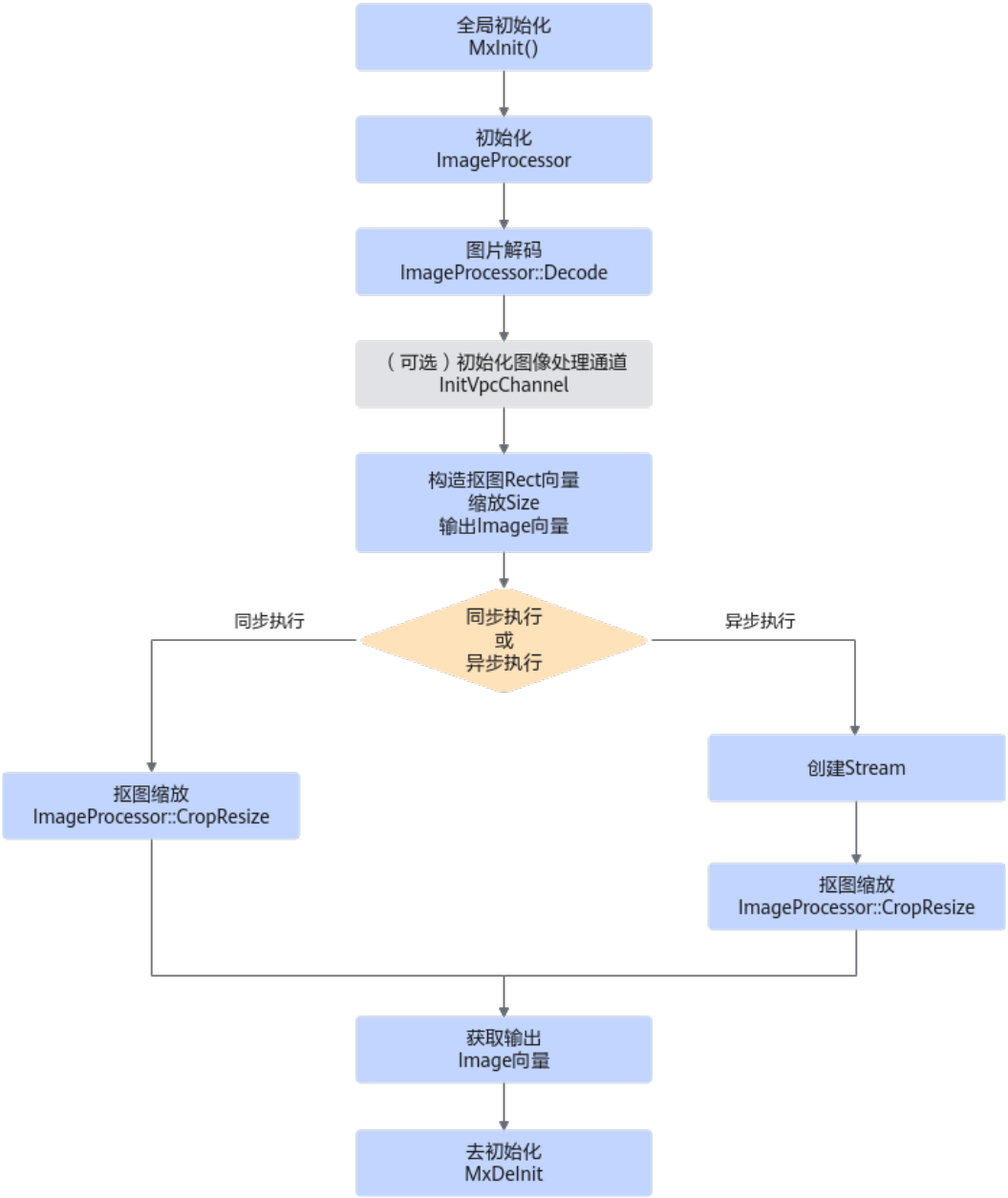
功能介绍

对输入的图像进行抠图并缩放操作，输出到Image对象中。

接口调用流程

使用抠图缩放接口前，用户需准备好被抠图缩放的图片对象。

图 2-6 图像处理（抠图缩放）接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，之后可以调用**InitVpcChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行抠图缩放前自动进行通道初始化。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图缩放的Image对象。后续可以通过图片处理接口对图片进行处理，生成最终需要抠图缩放的Image对象。
4. 构造抠图缩放参数和输出Image。

   根据业务要求，选择单输入多抠图单缩放、单输入多抠图多缩放或多输入多抠图多缩放的方式，构造对应的输入的Rect、Size和输出的Image。

5. 调用**CropResize()**接口对输入图片进行抠图缩放。
   a. 同步执行。
      i. 不创建Stream，将输入图片及其他参数传入接口，获取抠图缩放结果。
   b. 异步执行。
      i. 创建Stream，具体请参见**2.5 异步调用**。
      ii. 将输入图片、已创建的Stream及其他参数传入接口，获取抠图缩放结果。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //图像解码生成Image
    //解码后的图像类
    Image decodedImage;

    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //（可选）初始化图像处理通道
    imageProcessor.InitVpcChannel();

    //执行抠图缩放
    //抠图缩放操作后图像类
    std::vector<Image> cropResizedImageVec(1);

    //抠图尺寸
    Rect rect(0, 0, 240, 240);
    std::vector<Rect> cropConfigVec = {rect};

    //缩放尺寸
    Size size(416, 416);

    //抠图缩放操作
    ret = imageProcessor.CropResize(decodedImage, cropConfigVec, size, cropResizedImageVec);
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.1.7 抠图贴图

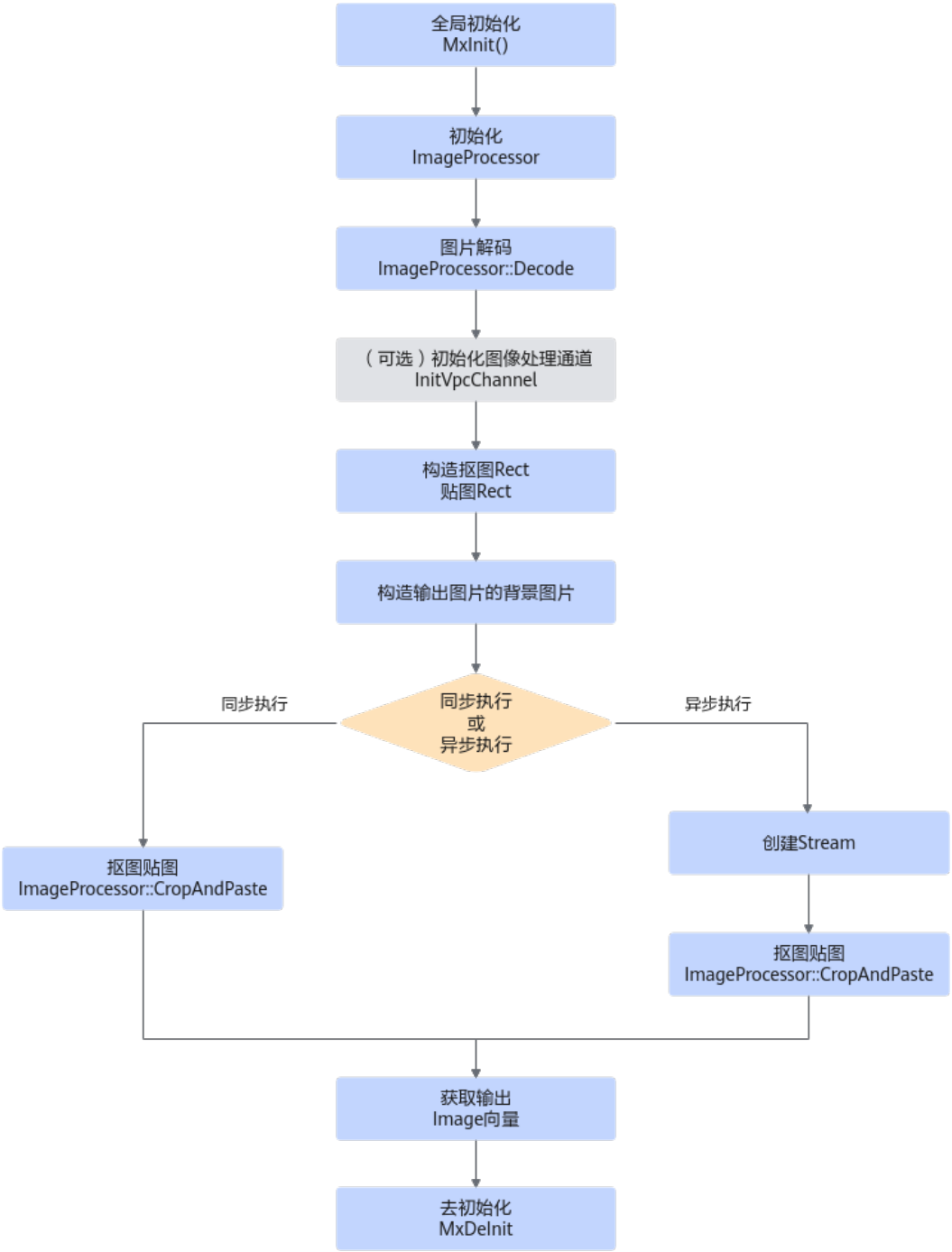
功能介绍

对输入的图像进行抠图并粘贴到背景图片中的操作，输出到Image对象中。

接口调用流程

使用抠图贴图接口前，用户需准备好被抠图以及被粘贴的图片对象。

图 2-7 图像处理（抠图贴图）接口调用流程



关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. ImageProcessor初始化。

用户构造ImageProcessor对象，之后可以调用**InitVpcChannel()**初始化通道，若不调用该接口，ImageProcessor会在执行抠图并贴图前自动进行通道初始化。

3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图贴图的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要抠图贴图的Image对象。

4. 构造抠图贴图参数和输出Image。
   a. 根据业务要求，设置被抠图片的矩阵并设置粘贴图片位置的矩阵，二者大小不一样时将自动进行缩放。
   b. 构造输出图片的背景图片，使用预申请内存的**Image()**构造函数构造被粘贴图片或其他非空图片作为输出。
5. 调用**CropAndPaste()**对输入图片进行抠图并将图片粘贴至指定位置。
   a. 同步执行。
      i. 不创建Stream，将输入图片及其他参数传入接口，获取抠图贴图结果。
   b. 异步执行。
      i. 创建Stream，具体请参见**2.5 异步调用**。
      ii. 将输入图片、已创建的Stream及其他参数传入接口，获取抠图贴图结果。
   c. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
//初始化
MxInit();
{
    //构造图像处理类
    ImageProcessor imageProcessor(deviceId);

    //图像解码生成Image
    //解码后的图像类
    Image decodedImage;

    //根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    if (ret != APP_ERR_OK) {
        return 0;
    }

    //（可选）初始化图像处理通道
    imageProcessor.InitVpcChannel();

    //构造背景Image（也可以选择其他已有数据的Image对象）
    Size imageSize(640, 640);
    size_t dataSize = 640 * 640 * 3 / 2;
    MemoryData imgData(dataSize, MemoryData::MemoryType::MEMORY_DVPP, deviceId);
    if (MemoryHelper::Malloc(imgData) != APP_ERR_OK) {
        return 0;
    }
    std::shared_ptr<uint8_t> pastedData((uint8_t*)imgData.ptrData, imgData.free);

    / 抠图贴图操作后图像类
    Image pastedImage(pastedData, dataSize, deviceId, imageSize);

    //执行抠图贴图
    //抠图和贴图尺寸
    Rect rectFrom(0, 0, 240, 240);
    Rect rectTo(0, 0, 480, 480);
```

```
    std::pair<Rect, Rect> cropPasteRect = {rectFrom, rectTo};

    //抠图贴图操作
    ret = imageProcessor.CropAndPaste(resizeImage, cropPasteRect, pastedImage) ;
    if (ret != APP_ERR_OK) {
        return 0;
    }
}
//去初始化
MxDeInit();
```

2.3.2 通过张量方法进行图像处理（Tensor）

2.3.3 视频编码

功能介绍

通过构造**VideoEncoder**类实例可实现视频编码功能，编码功能配置项及各项约束与支持情况请参考**5.4.1.4 VideoEncodeConfig**数据结构说明。

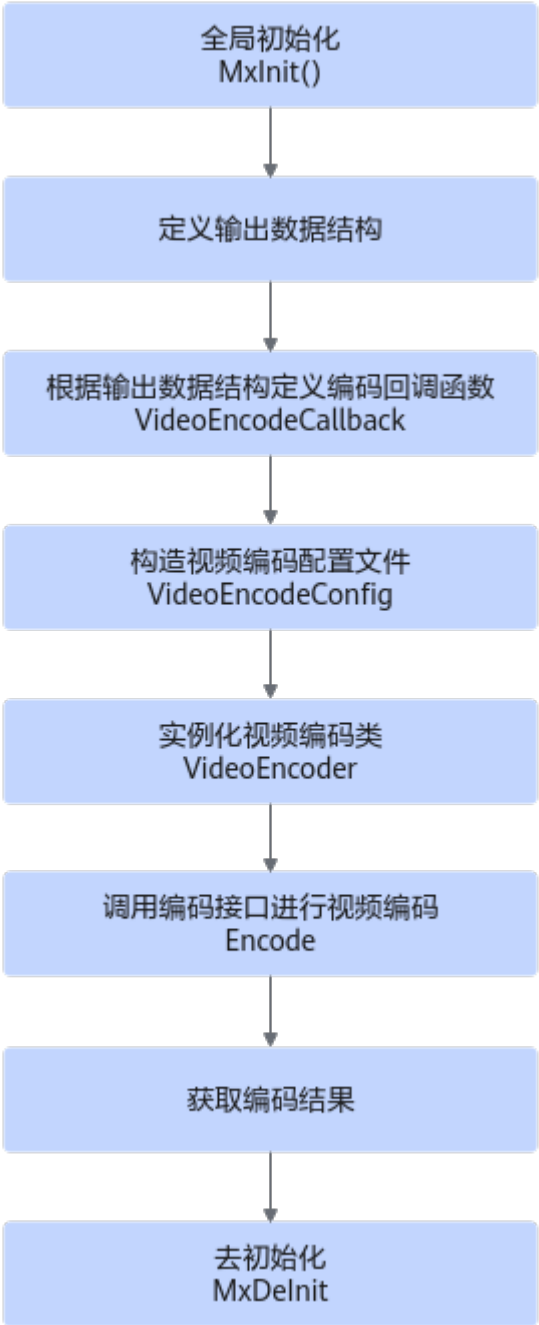
视频编码支持自定义输出数据格式，通过自定义回调函数传入编码功能配置项，方便用户使用编码后的数据，详情可参考**5.4.1.5 VideoEncodeCallBack**。

接口调用流程

首先根据需求定义需要的输出数据组合方式，根据组合方式参考VideoEncodeCallBack定义回调函数传入编码功能配置项，然后实例化VideoEncoder类，最后调用**Encode**成员函数完成解码，获取数据。

视频编码接口调用流程参考如下：

图 2-8 视频编码接口调用流程



mxVision提供VideoEncoder类进行视频编码，关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. 定义输出数据组合形式。
   - 输出数据包含视频帧编码后得到的的Image类数据、当前编码帧的“frameId”和通道“channelId”。
   - 根据需要选择获取以上哪些数据。
3. 定义输出回调函数。
   - 根据要获取的数据定义回调函数，在函数内组装自定义数据。

- 回调函数输入参数固定为VideoEncodeCallBack形式，函数内可选择输出。
  请勿在回调函数内实现过于复杂的操作，建议使用自定义“userData”来接收视频编码回调结果，否则回调线程会发生卡住的现象，导致视频编码速度变慢。

4. 构造视频编码配置项。
   配置项及各项约束与支持情况请参考VideoEncodeConfig数据结构说明。
5. 实例化视频编码类。
   将配置好的VideoEncodeConfig传入构造函数接口，实例化视频编码类。
6. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

如下提供关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
// 初始化
MxBase::MxInit();
{
    // 定义要接收的解码数据
    struct FrameImage {
        Image image;                // Video Image Class
        uint32_t frameId = 0;       // Video Frame Index
        uint32_t channelId = 0;     // Video Channel Index
    };

    // 定义获取解码数据到自定义结构的回调函数
    APP_ERROR CallBackVenc(std::shared_ptr<uint8_t>& outDataPtr, uint32_t& outDataSize, uint32_t&
channelId,
                           uint32_t& frameId, void* userData)
    {
        Image image(outDataPtr, outDataSize, -1, Size(1920, 1080));
        FrameImage* encodedVec = static_cast<FrameImage*>(userData);
        FrameImage frameImage;
        encodedVec.image = image;
        encodedVec.channelId = channelId;
        encodedVec.frameId = frameId;
        return APP_ERR_OK;
    }
    // 构造解码配置项
    VideoEncodeConfig vEncodeConfig;
    VideoEncodeCallBack cPtr2 = CallBackVenc;
    vEncodeConfig.callbackFunc = cPtr2;
    vEncodeConfig.width = 1920;
    vEncodeConfig.height = 1080;
    vEncodeConfig.keyFrameInterval = 1;
    vEncodeConfig.srcRate = 30;
    vEncodeConfig.rcMode = 1;
    vEncodeConfig.maxBitRate = 30000;
    vEncodeConfig.ipProp = 70;
    // 实例化解码类
    VideoEncoder videoEncoder(vEncodeConfig, deviceId);
    // 执行解码操作
    FrameImage encodedFrame;

    for (size_t i = 0; i < inputFrameList.size(); i++) {
        ret = videoEncoder.Encode(image, frameId, &encodedFrame);
    }
}
// 去初始化
MxBase::MxDeInit();
```

2.3.4 视频解码

功能介绍

通过构造**VideoDecoder**类实例可实现视频解码功能，解码功能配置项及各项约束与支持情况请参考**5.4.1.2 VideoDecodeConfig**数据结构说明。

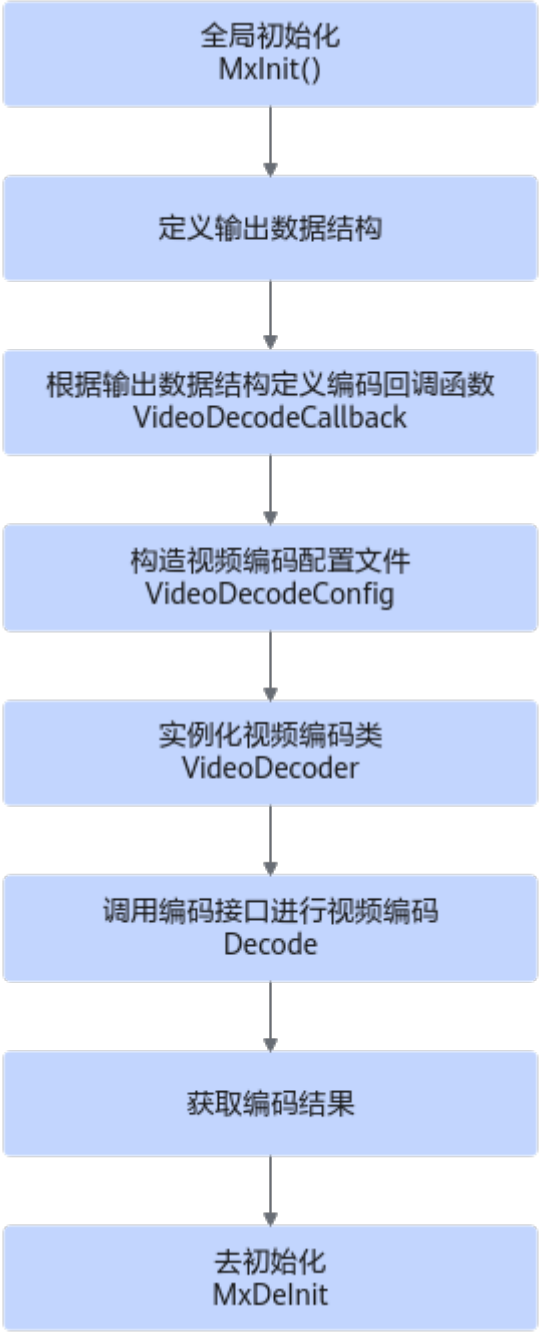
视频解码支持自定义输出数据格式，通过自定义回调函数传入解码功能配置项，方便用户解码后的数据，详情可参考**5.4.1.3 VideoDecodeCallBack**数据结构说明。

接口调用流程

首先根据需求定义需要的输出数据组合方式，根据组合方式参考VideoDecodeCallBack定义回调函数传入解码功能配置项，然后实例化VideoDecoder类，最后调用**Decode**成员函数完成解码，获取数据。

视频解码接口调用流程参考如下：

图 2-9 视频解码接口调用流程



mxVision提供VideoDecoder类进行视频解码，关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. 定义输出数据组合形式。
   - 输出数据包含视频帧解码后得到的的Image类数据、当前解码帧的“frameId”和通道“channelId”。
   - 可根据需要选择获取以上哪些数据。
3. 定义输出回调函数。
   - 根据要获取的数据定义回调函数，在函数内组装自定义数据。

- 回调函数输入参数固定为VideoDecodeCallBack形式，函数内可选择输出。
- 请勿在回调函数内实现过于复杂的操作，建议只进行自定义获取解码数据操作。

4. 构造视频解码配置项。
   配置项及各项约束与支持情况请参考VideoDecodeConfig数据结构说明。
5. 实例化视频解码类。
   将配置好的VideoDecodeConfig传入构造函数接口，实例化视频解码类。
6. 调用解码接口。
   - 一个实例化的解码器在首次调用Decode接口时，会判断是否为预申请场景。
   - 使用预申请场景时，需要在后续调用Decode接口进行预申请输出内存，防止接口调用失败。
7. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

如下提供关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
// 初始化
MxBase::MxInit();
{
    // 定义要接收的解码数据
    struct FrameImage {
        Image image;                // Video Image Class
        uint32_t frameId = 0;        // Video Frame Index
        uint32_t channelId = 0;      // Video Channel Index
    };

    //定义获取解码数据到自定义结构的回调函数
    APP_ERROR CallBackVdec(Image& decodedImage, uint32_t channelId, uint32_t frameId, void* userData)
    {
      FrameImage* decodedVec = static_cast<FrameImage*>(userData);
      decodedVec->image = decodedImage;
      decodedVec->channelId = channelId;
      decodedVec->frameId = frameId;
    }
    // 构造解码配置项
    VideoDecodeConfig config;
    VideoDecodeCallBack cPtr = CallBackVdec;
    config.width = 1920;
    config.height = 1080;
    config.callbackFunc = cPtr;
    config.skipInterval = 0;
    config.inputVideoFormat = StreamFormat::H264_MAIN_LEVEL;
    config.outputImageFormat = ImageFormat::YUV_SP_420;
    // 实例化解码类
    VideoDecoder videoDecoder(config, deviceId, channelId);
    // 执行解码操作
    ret = videoDecoder.Decode(dataPtr, dataSize, frameId, &decodedFrame);
}
// 去初始化
MxBase::MxDeInit();
```

2.4 模型推理

功能介绍

使用mxVision模型推理功能通过给定输入和指定模型，进行推理获得输出结果，支持om格式的模型推理，可使用ATC工具构建的动态Batch、动态分辨率和分档动态维度模型进行推理。

接口调用流程

使用模型推理前，用户需准备好输入数据以及需要加载的模型，根据模型路径或者内存方式初始化**Model**类，通过调用Model类的**Infer**接口获取模型推理结果。输入数据必须和模型输入数据类型以及格式一致，如果用户自行申请输出数据内存，则输出数据类型和格式需和模型输出保持一致，模型输入输出信息可以通过Model类的相关接口进行查询。

模型推理调用流程参考如下：

关键接口说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. 模型初始化。

   用户需根据实际业务情况确认模型加载方式，选择**从文件加载模型**或**从内存加载模型**。如果从内存加载，需要先将模型文件读取到内存，可通过以下两种方式传入。

   - 从文件加载模型，可选择直接向Model接口传入模型路径，进行初始化。
   - 通过结构体ModelLoadOptV2中的“loadType”字段指定加载方式后，再传入Model接口，其中加载方式在模型方面区分从文件加载模型，还是从内存加载模型，在内存方面区分是由系统内部管理，还是由用户管理，具体可参见**5.4.1.1 ModelLoadOptV2**。
3. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
// 初始化
MxBase::MxInit();
{
    // 输入二进制数据， 需用户自行准备
    std::string filePath = "./test.bin";
    // 读取输入数据到内存
    void* dataPtr = ReadTensor(filePath);
    // 输入数据类型，和模型输入数据类型一致
    auto dataType = MxBase::TensorDType::INT32;
    // 构造输入shape，和模型输入shape一致
    std::vector<uint32_t> shape = {1, 128};
    // 构造tensor
    MxBase::Tensor tensor(dataPtr, shape, dataType, 0);
    // 构造模型输入
    std::vector<MxBase::Tensor> inputs{tensor};
    // 模型路径，需用户自行指定
    std::string modelPath = "./test.om";
    // 根据模型路径加载模型
    MxBase::Model model(modelPath);
    // 执行模型推理, outputs即为推理结果
    std::vector<MxBase::Tensor> outputs = model.Infer(inputs);
}
//去初始化
MxBase::MxDeInit();
```

根据结构体ModelLoadOptV2方式初始化示例参考如下：

```
MxBase::ModelLoadOptV2 mdlLoadOpt;
mdlLoadOpt.loadType = ModelLoadOptV2::LOAD_MODEL_FROM_FILE;  // 指定模型加载方式
mdlLoadOpt.modelPath = modelPath;
MxBase::Model model(mdlLoadOpt);
```

2.5 异步调用

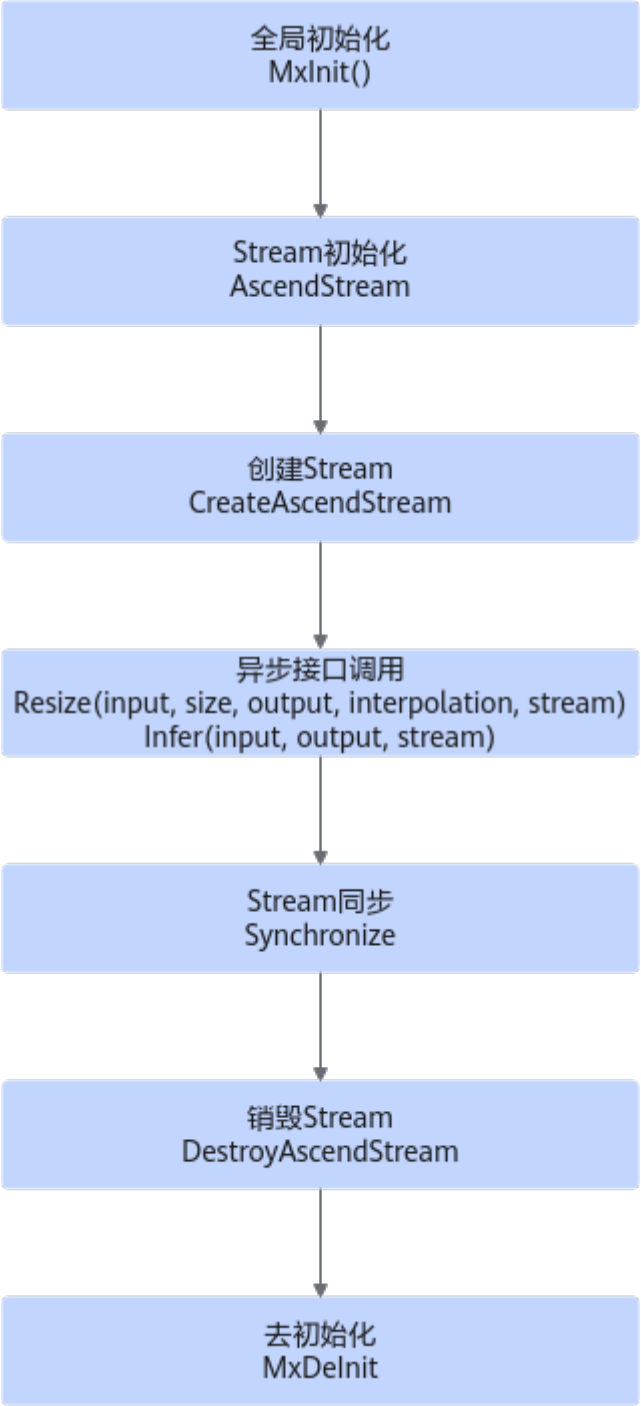
功能介绍

mxVision默认采用同步执行模式，部分接口已支持用户通过申请AscendStream异步执行，具体接口异步支持情况请参见**5 API参考（C++）**。

接口调用流程

- 用户通过自定义**AscendStream**类构造需要的Stream实例传入异步接口，通过接口传入并在指定Stream上执行，同一Stream串行指定接口任务后可通过调用**Synchronize()**接口阻塞应用程序或线程运行，直到该Stream中的所有任务全部完成。
- 支持多Stream异步执行，每个Stream内接口顺序执行，如果用户使用多Stream异步执行或异步执行结果需传入不支持异步接口情况，需要用户可选择执行Synchronize()接口，在适当位置执行同步Stream操作，保证结果已正确返回供后续使用。
- 对于异步调用进行媒体数据处理时，需要调用**Synchronize()**接口使异步任务完成才能及时归还使用的通道，避免资源池被耗尽。

图 2-10 Stream 异步模式接口调用流程（以 Resize 操作为例）



mxVision提供AscendStream类进行Stream管理，关键步骤说明如下：

1. 调用**MxInit()**接口进行全局初始化。
2. 初始化Stream。

   用户通过向构造函数传入“deviceId”指定要创建Stream的设备，支持的“deviceId”根据用户环境存在差异。
3. 创建Stream。

- 与Stream初始化绑定使用，在使用Stream前需要调用**CreateAscendStream()**成员函数完成Stream创建。
- 同一设备（deviceId），Stream最大支持创建1021个，具体请参见《CANN 应用软件开发指南 (C&C++)CANN 应用软件开发指南 (C&C++)》。

4. 调用异步接口。
   - 支持异步的接口传入指定Stream，同一Stream中接口按顺序串行执行。
   - 指定不同Stream的接口之间并行执行。
5. 同步Stream。

   如需保证异步执行结果在作为下一个接口入参时已执行完成，可调用**Synchronize()**成员函数显式同步Stream。
6. 销毁Stream。

   用户在业务流程完成或Stream结束时，需调用**DestroyAscendStream()**成员函数销毁Stream，否则可能会造成Stream耗尽情况。
7. 调用**MxDeInit()**接口对初始化的全局资源进行去初始化。

示例代码

以图像异步缩放和yoloV3模型异步推理为例，提供关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考。

```
// 初始化
MxBase::MxInit();
{
    // 用户创建目标deviceId的stream
    AscendStream stream(deviceId)
    stream.CreatAscendStream();
    // 创建图像解码类实例
    ImageProcessor imageProcessor(deviceId);
    // 解码后的Image类实例
    Image decodedImage;
    // 根据图像路径进行解码
    APP_ERROR ret = imageProcessor.Decode(imagePath, decodedImage);
    // 缩放后的图像类实例
    Image resizedImage;
    // 异步调用缩放接口
    ret = imageProcessor.Resize(decodedImage, Size(416, 416), resizedImage,
Interpolation::HUAWEI_HIGH_ORDER_FILTER, stream);
    // 调用同步接口，可以显式保证调用后续接口时缩放已完成
    stream.Synchronize();
    // 将 Image 类转换为 Tensor 类
    Tensor tensorImg = resizedImage.ConvertToTensor();
    // yoloV3模型推理
    string yoloPath = "./model/yolov3_tf_bs1_fp16.om";
    Model yoloV3(yoloPath, deviceId);
    // 构造推理输入输出 (单batch)
    vector<Tensor> yoloV3Inputs = {tensorImg};
    vector<Tensor> yoloV3Outputs = {};
    // 开始异步推理
    ret = yoloV3.Infer(yoloV3Inputs, yoloV3Outputs，stream);
    // 调用同步接口，可以显式保证调用后续接口时缩放已完成
    // 用户也可选择同步前在该stream串行其他异步接口
    stream.Synchronize();
    // 调用DestroyAscendStream销毁stream
    stream.DestroyAscendStream();
}
//去初始化
MxBase::MxDeInit();
```

2.6 编译运行

运行样例前，需设置mxVison环境变量。

```
source {mxVison安装目录}/mxVision/set_env.sh
```

编译

1. 文件准备，用户准备好自己的源码文件“main.cpp”，以及“CMakeLists.txt”文件，“CMakeLists.txt”编写可参考如下内容：

```
# CMake lowest version requirement
cmake_minimum_required(VERSION 3.5.2)
# project information
project(MindX_SDK_Sample)
set(MX_SDK_HOME $ENV{MX_SDK_HOME})
if (NOT DEFINED ENV{MX_SDK_HOME})
string(REGEX REPLACE "(.*)/(.*)/(.*)/(.*)" "\\1" MX_SDK_HOME  ${CMAKE_CURRENT_SOURCE_DIR})
message(STATUS "set default MX_SDK_HOME: ${MX_SDK_HOME}")
else ()
message(STATUS "env MX_SDK_HOME: ${MX_SDK_HOME}")
endif()
# Compile options
add_definitions(-D_GLIBCXX_USE_CXX11_ABI=0)
add_definitions(-Dgoogle=mindxsdk_private)
add_compile_options(-std=c++11 -fPIC -fstack-protector-all -Wall)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})
set(CMAKE_CXX_FLAGS_DEBUG "-g")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,-z,relro,-z,now,-z,noexecstack -s -
pie")
set(CMAKE_SKIP_RPATH TRUE)

# add mxbase header path
include_directories(
${MX_SDK_HOME}/include/
${MX_SDK_HOME}/opensource/include/
)

# add mxbase lib path
link_directories(
${MX_SDK_HOME}/lib/
${MX_SDK_HOME}/lib/modelpostprocessors/
${MX_SDK_HOME}/opensource/lib/
${MX_SDK_HOME}/opensource/lib64/
)
add_executable(main main.cpp)
target_link_libraries(main glog mxbase opencv_world cpprest pthread mindxsdk_protobuf)
install(TARGETS main DESTINATION ${CMAKE_RUNTIME_OUTPUT_DIRECTORY})
```

2. CMake编译。
   a. 在“CMakeLists.txt”文件所在目录新建“build”目录。
      ```
      mkdir build
      ```
   b. 进入“build”目录。
      ```
      cd build
      ```
   c. 执行**cmake**命令。
      ```
      cmake ..
      ```
   d. 执行**make**命令生成可执行文件“main”。
      ```
      make -j
      ```

运行

执行编译生成的可执行文件即可运行程序。

```
./main
```

说明

- 若程序执行出现libascendcl.so报错，请参见**8.1 程序执行出现libascendcl.so报错**解决。
- 若尝试打印Device侧的Tensor、Image内存数据时出现coredump提示，请参见**8.14 尝试打印Device侧的Tensor、Image内存数据时，发生coredump**解决。

3 使用 API 接口方式开发（Python）

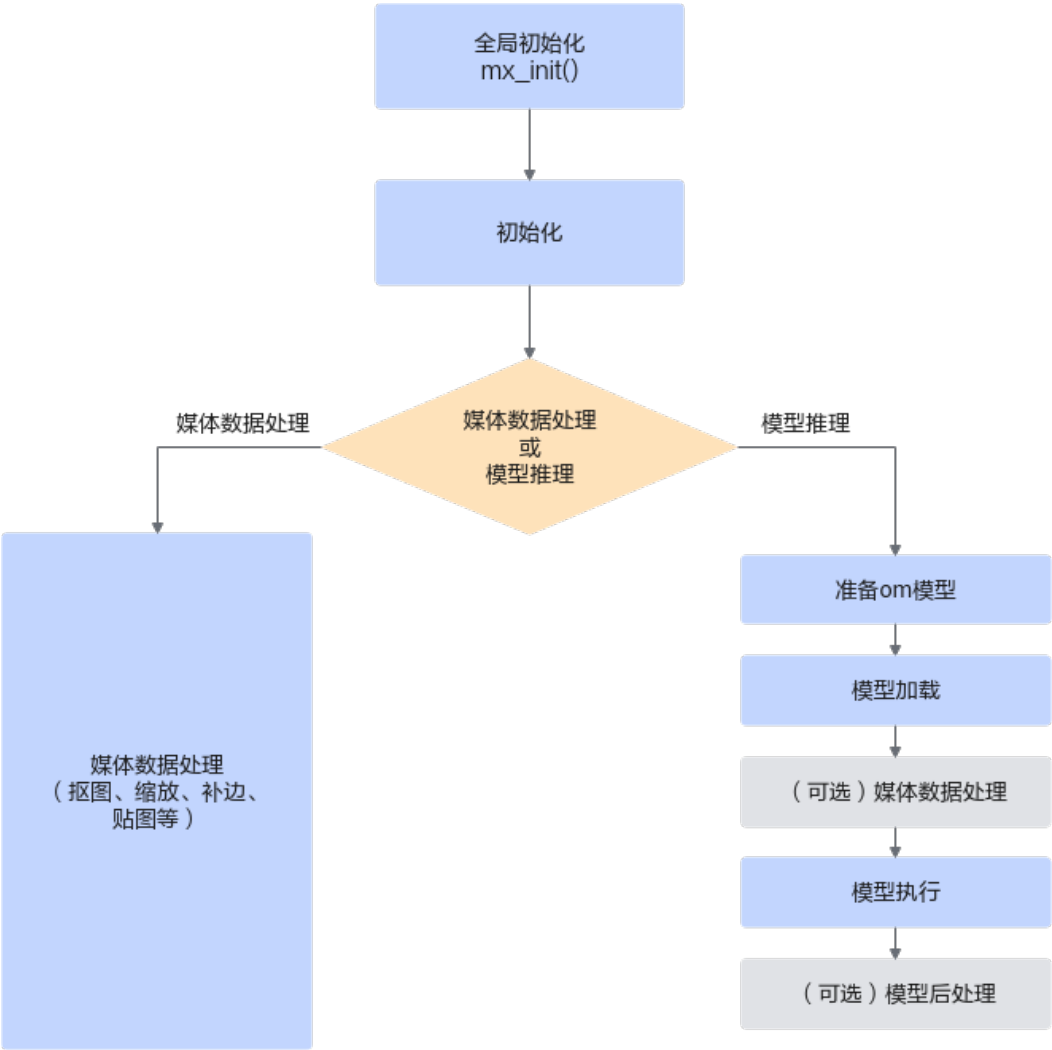
3.1 开发流程
3.2 初始化
3.3 媒体数据处理
3.4 模型推理
3.5 运行

版权所有 © 华为技术有限公司

3.1 开发流程

流程介绍

图 3-1 API 开发流程



关键步骤说明

1. 使用mxVision接口进行应用开发，需首先进行初始化才能进行后续处理，具体请参见**3.2 初始化**。
2. 媒体数据处理/模型推理。
   a. 媒体数据处理功能，可实现图像编解码、抠图、缩放、补边等操作，具体请参见**3.3 媒体数据处理**。
   b. 模型推理，可实现如目标识别，图像分类等应用，环节可参考以下几个步骤，具体请参见**3.4 模型推理**。
      i. 模型推理前，需准备合适的om模型。

ii. 模型加载，将准备好的模型通过文件或内存方式加载到系统中，具体请参见**3.4 模型推理**。
iii. （可选）媒体数据处理，模型推理前，可对输入的媒体数据进行处理，如图片解码、抠图、缩放、补边等操作。
iv. 模型执行，使用模型实现图片分类、目标识别等功能。
v. （可选）模型后处理，对模型推理输出的数据进行后处理操作，此处根据用户的实际需求来处理推理结果，如整理为可理解阅读的具体结果。模型后处理支持通过两种方式进行：
   - （推荐）自行选择模型后处理方式。
   - 通过调用mxVision接口进行，具体操作请参见**6.7 模型后处理**。

3.2 初始化

功能介绍

mxVision应用开发初始化接口，在代码调用相关接口之前，需调用全局初始化函数mx_init()申请设备资源和日志资源。

示例代码

以下为初始化方法的代码示例，不可以直接拷贝运行，仅供参考。

全局初始化，加载设备资源和日志资源。

```
from mindx.sdk import base
base.mx_init()
# 执行全局初始化后即可正常调用mxVision接口
```

3.3 媒体数据处理

3.3.1 图片解码

功能介绍

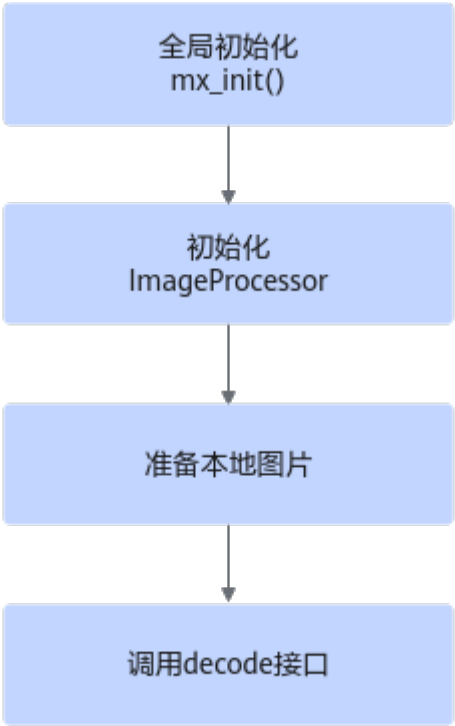
通过对输入图片数据进行解码处理，将本地图片转换为Image类，用于后续前处理和推理业务，类型目前支持JPEG和PNG格式。

接口调用流程

用户需准备好本地待解码的图片文件，初始化ImageProcessor类，通过调用ImageProcessor类的**decode**接口解码返回输出的Image对象。

图片解码调用流程参考如下：

图 3-2 图片解码接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。
   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 调用**decode**接口对图片进行解码。
   用户需根据解码的不同需要，传入对应格式参数到decode接口。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)
```

3.3.2 图片编码

功能介绍

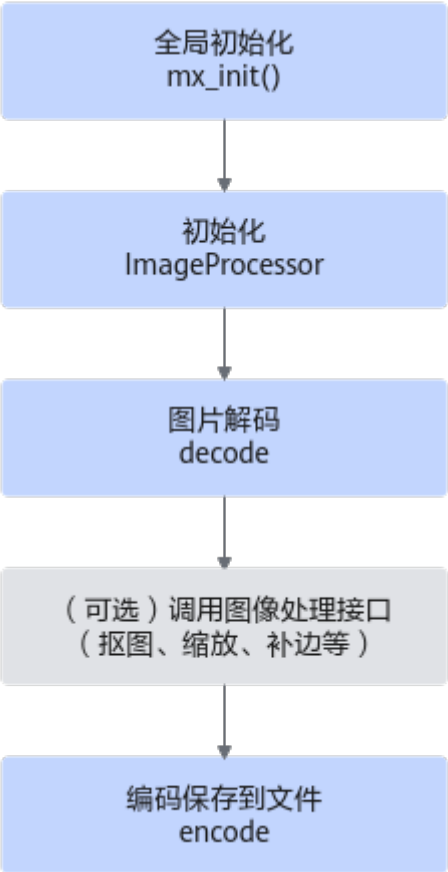
将接口输出的Image对象编码为JPG格式的图片保存到指定的图片路径。

接口调用流程

用户需提前准备好待编码的Image对象，可以通过图片解码以及图像处理接口（抠图、缩放、补边等操作对应接口）输出，输出到本地图片。

图片编码调用流程参考如下：

图 3-3 图片编码接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可编码的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要编码的Image对象。
4. 调用**encode**接口对输入图片进行编码。

   用户需根据实际业务情况输出图片数据，指定待输出的Image和输出路径。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Rect, Image
```

```
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "image_data/test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像处理操作（抠图）
crop_para = [Rect(300, 100, 550, 350)]
croped_images = imageProcessor.crop(decoded_image, crop_para)

image_save_path = "croped_image.jpg"
imageProcessor.encode(croped_images[0], image_save_path)
```

3.3.3 抠图

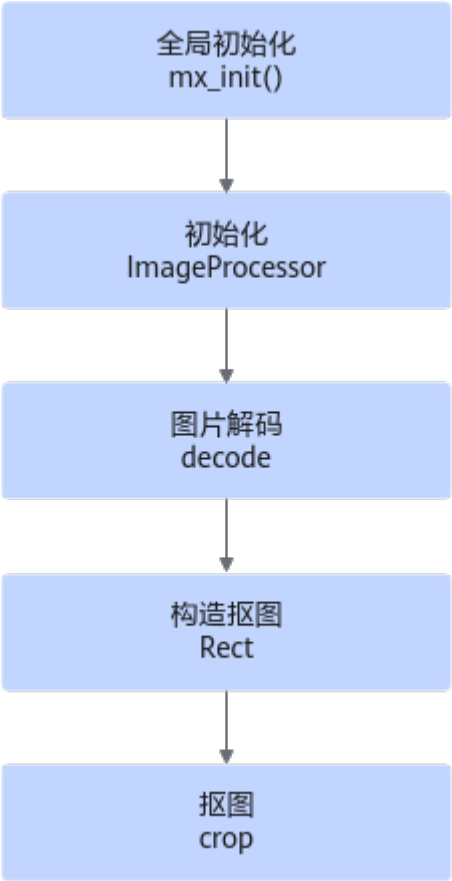
功能介绍

对输入的图像进行抠图操作，输出到Image对象中。

接口调用流程

使用抠图接口前，用户需准备好被抠图的图片对象。

图 3-4 图像处理（抠图）接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要抠图的Image对象。
4. 构造抠图Rect以及输出Image。

   根据业务要求，选择一对多、一对一或多对多的抠图方式，构造对应的输入和输出Rect和Image。
5. 将输入图片及其他参数传入**crop**接口，获取抠图结果。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Rect, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "image_data/test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像处理操作（抠图）
crop_para = [Rect(300, 100, 550, 350)]
croped_images = imageProcessor.crop(decoded_image, crop_para)
```

3.3.4 缩放

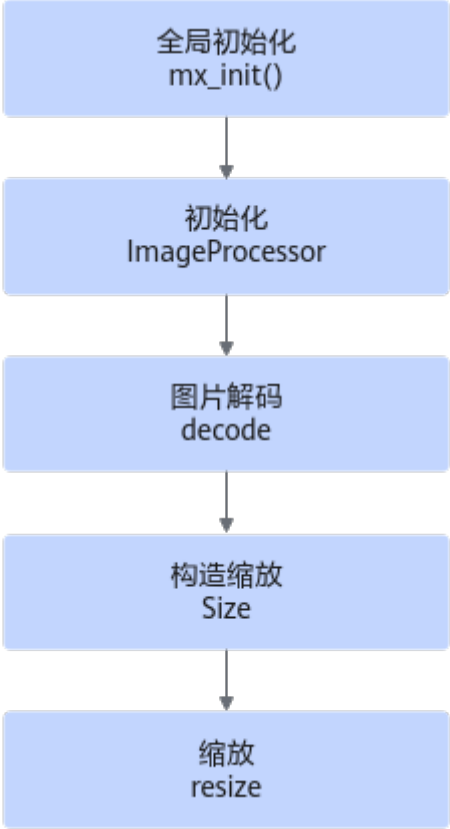
功能介绍

对输入的图像进行缩放操作，输出到Image对象中。

接口调用流程

使用缩放接口前，用户需准备好被缩放的图片对象。

图 3-5 图像处理（缩放）接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可缩放的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要缩放的Image对象。
4. 构造缩放参数和输出Image。

   根据业务要求，构造对应的输入和输出Size和Image。
5. 将输入图片及其他参数传入**resize**接口，获取缩放结果。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Size, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "image_data/test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像缩放
# 缩放尺寸
```

```
size_para = Size(224, 224)
# 读取将解码后的Image类按尺寸进行缩放，缩放方式为华为自研的高阶滤波算法
（huaweiu_high_order_filter）
resized_image = imageProcessor.resize(decoded_image, size_para, base.huaweiu_high_order_filter)
```

3.3.5 补边

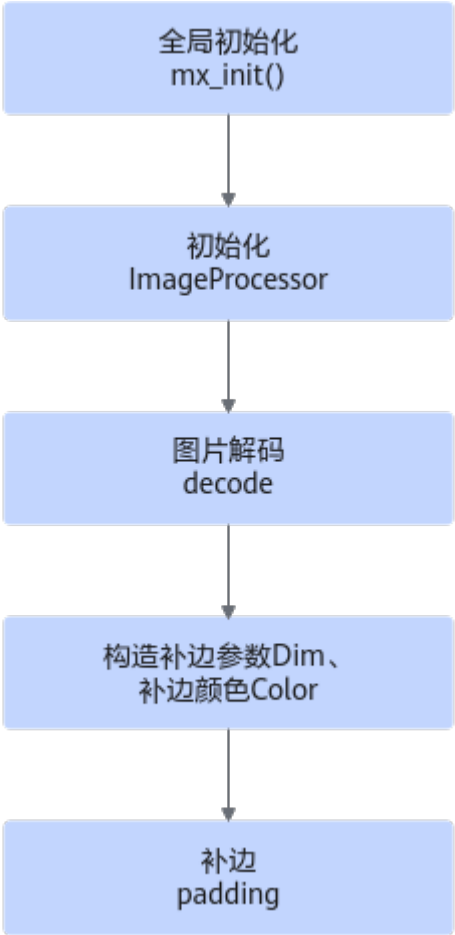
功能介绍

对输入的图像进行补边操作，输出到Image对象中。

接口调用流程

使用补边接口前，用户需准备好被补边的图片对象。

图 3-6 图像处理（补边）接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。
   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 使用图片解码接口对输入图片进行解码。

根据业务对图片进行解码，生成可补边的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要补边的Image对象。

4. 构造补边参数。

   根据业务要求，构造对应的输入的“padDim”（表示补边上下左右的像素值）、“color”（补边颜色）。

5. 调用**padding**接口对输入图像进行补边。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Dim, Color, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "image_data/test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像补边
# 补边尺寸
dim_para = Dim(100, 100, 100, 100)
# 读取将解码后的Image类按Dim进行补边，补边方式为重复最后一个元素
padded_image = imageProcessor.padding(decoded_image, dim_para, Color(0, 0, 0), base.border_replicate)
```

3.3.6 抠图缩放

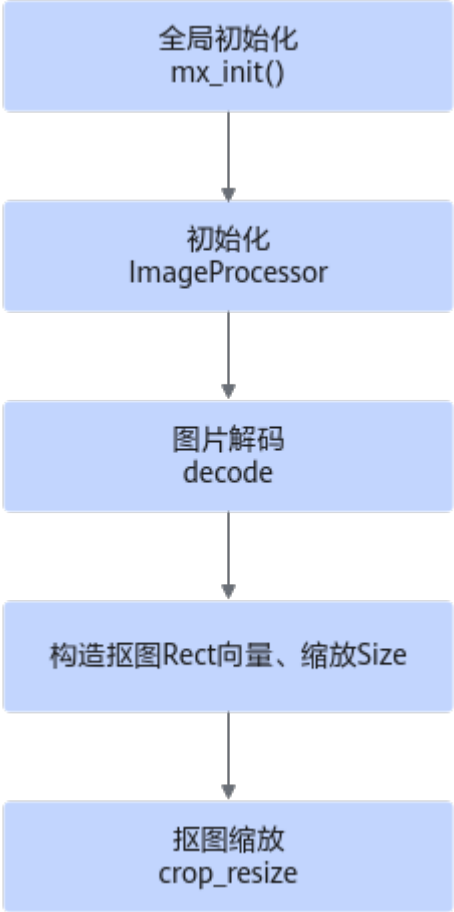
功能介绍

对输入的图像进行抠图并缩放操作，输出到Image对象中。

接口调用流程

使用抠图缩放接口前，用户需准备好被抠图缩放的图片对象。

图 3-7 图像处理（抠图缩放）接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。

   用户构造ImageProcessor对象，构造时需用户指定设备ID。
3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图缩放的Image对象。后续可以通过图片处理接口对图片进行处理，生成最终需要抠图缩放的Image对象。
4. 构造抠图缩放参数和输出Image。

   根据业务要求，选择单输入单抠图单缩放或单输入多抠图多缩放的方式，构造对应的输入的Rect、Size。
5. 调用**crop_resize**接口对输入图片进行抠图缩放。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Rect, Size, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "test_image.jpg"
```

```
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像抠图并缩放
crop_resize_para = [(Rect(300, 100, 550, 350), Size(100, 100))]
crop_resize_image = imageProcessor.crop_resize(decoded_image, crop_resize_para)
```

3.3.7 抠图贴图

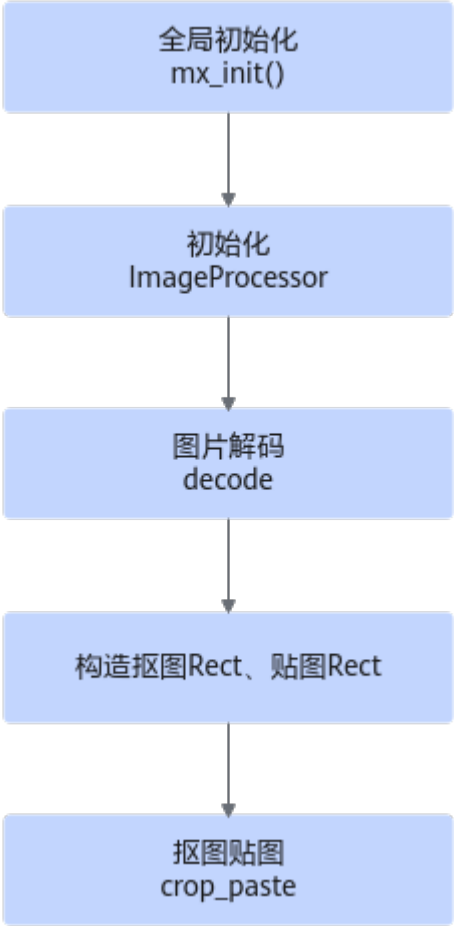
功能介绍

对输入的图像进行抠图并粘贴到背景图片中的操作，输出到Image对象中。

接口调用流程

使用抠图贴图接口前，用户需准备好被抠图以及被粘贴的图片对象。

图 3-8 图像处理（抠图贴图）接口调用流程



关键步骤说明如下：

1. 调用**mx_init()**接口进行全局初始化。
2. ImageProcessor初始化。
   用户构造ImageProcessor对象，构造时需用户指定设备ID。

3. 使用图片解码接口对输入图片进行解码。

   根据业务对图片进行解码，生成可抠图贴图的Image对象。后续可以通过图像处理接口对图片进行处理，生成最终需要抠图贴图的Image对象。

4. 构造抠图贴图参数和输出Image。
   a. 根据业务要求，设置被抠图片的矩阵并设置粘贴图片位置的矩阵，二者大小不一样时将自动进行缩放。
   b. 构造输出图片的背景图片，使用Image构造函数构造被粘贴图片或其他非空图片作为输出。
5. 调用**crop_paste**对输入图片进行抠图并将图片粘贴至指定位置。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
from mindx.sdk import base
from mindx.sdk.base import ImageProcessor, Rect, Image
# 图像解码
# 初始化ImageProcessor对象
imageProcessor = ImageProcessor(device_id)
image_path = "image_data/test_image.jpg"
# 读取图片路径进行解码，解码格式为nv12（YUV_SP_420）
decoded_image = imageProcessor.decode(image_path, base.nv12)

# 图像抠图并贴图
crop_paste_para = (Rect(300, 100, 550, 350), Rect(100, 100, 1500, 1500))
imageProcessor.crop_paste(decoded_image , crop_paste_para, paste_image)
```

3.3.8 视频编码

功能介绍

通过构造**VideoEncoder**类实例可实现视频编码功能，编码功能配置项及各项约束与支持情况请参考**6.3.1.36 VideoEncodeConfig类**数据结构说明。

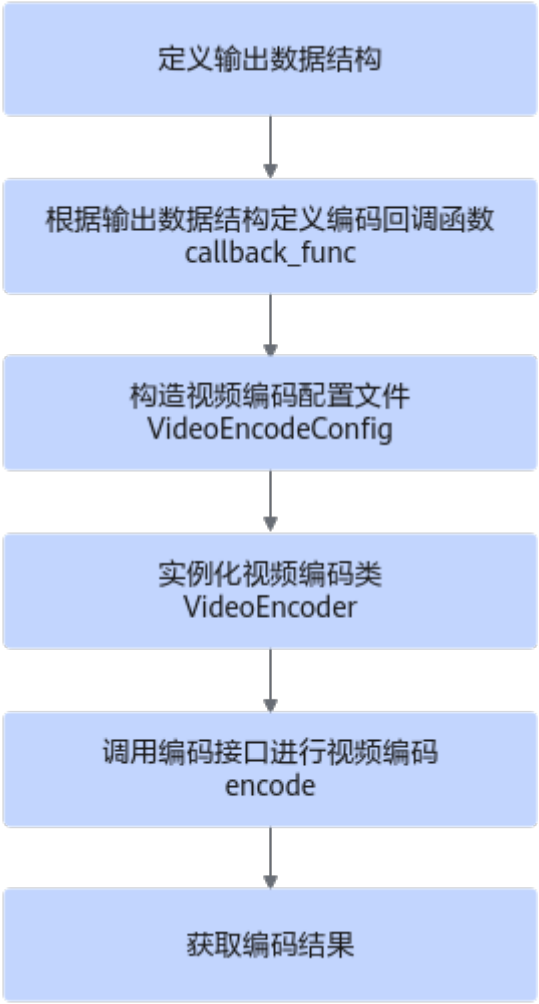
视频编码支持自定义输出数据格式，通过自定义回调函数传入编码功能配置项，方便用户使用编码后的数据，详情可参考**6.5.6 VencCallBacker类**。

接口调用流程

首先根据需求定义需要的输出数据组合方式，根据组合方式参考VencCallBacker定义回调函数传入编码功能配置项，然后实例化VideoEncoder类，最后调用**encode**成员函数完成解码，获取数据。

视频编码接口调用流程参考如下：

图 3-9 视频编码接口调用流程



mxVision提供VideoEncoder类进行视频编码，关键步骤说明如下：

1. 定义输出数据组合形式。
   - 输出数据包含视频帧编码后得到的的Image类数据、当前编码帧的“frameId”和通道“channelId”。
   - 根据需要选择获取以上哪些数据。
2. 定义输出回调函数。
   - 根据要获取的数据定义回调函数，在函数内组装自定义数据。
   - 回调函数输入参数固定为callback_func形式，函数内可选择输出。

     请勿在回调函数内实现过于复杂的操作，建议使用自定义“userData”来接收视频编码回调结果，否则回调线程会发生卡住的现象，导致视频编码速度变慢。
3. 构造视频编码配置项。

   配置项及各项约束与支持情况请参考**VideoEncodeConfig**数据结构说明。
4. 实例化视频编码类。

   将配置好的VideoEncodeConfig传入构造函数接口，实例化视频编码类。
5. 调用**encode**接口对视频进行编码。

示例代码

如下提供关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import os
import numpy as np
import time
from mindx.sdk import base
from mindx.sdk.base import Image, ImageProcessor
from mindx.sdk.base import VideoEncoder, VideoEncodeConfig, VencCallBacker
# 视频编码回调函数
def venc_callback(pyChar, outDataSize, channelId, frameId):
    with open('video_save_data/output.h264', 'ab') as file:
        file.write(pyChar)

# 初始化VencCallBacker类并注册回调函数
vencCallBacker = VencCallBacker()
vencCallBacker.registerVencCallBack(venc_callback)
# 初始化VideoEncodeConfig
venc_conf = VideoEncodeConfig()
venc_conf.keyFrameInterval = 50
venc_conf.srcRate = 30
venc_conf.maxBitRate = 6000
venc_conf.ipProp = 30
# 初始化VideoEncoder
videoEncoder = VideoEncoder(venc_conf, vencCallBacker, device_id)
# 将编码后数据保存为本地视频，若视频文件已存在则删除
venc_save_path = os.path.join(save_path, 'output.h264')
video_encode_exists = os.path.exists(venc_save_path)
if video_encode_exists:
     os.remove(venc_save_path)
# 从decoded_data_list中循环取Image类进行编码
for i, img in enumerate(decoded_data_list):
    videoEncoder.encode(img, i)
```

3.3.9 视频解码

功能介绍

通过构造VideoDecoder类实例可实现视频解码功能，解码功能配置项及各项约束与支持情况请参考**6.3.1.35 VideoDecodeConfig类**数据结构说明。

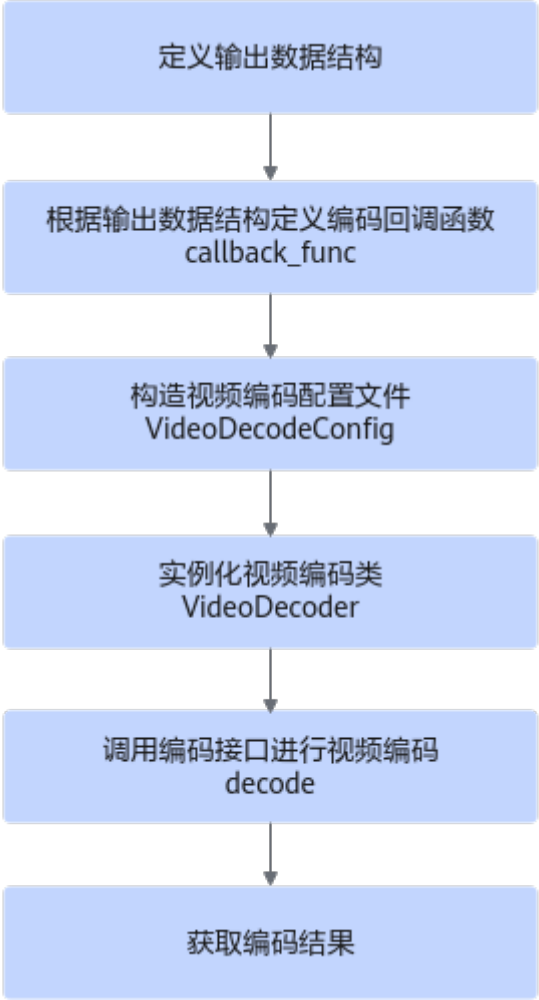
视频解码支持自定义输出数据格式，通过自定义回调函数传入解码功能配置项，方便用户解码后的数据，详情可参考**6.5.4 VdecCallBacker类**数据结构说明。

接口调用流程

首先根据需求定义需要的输出数据组合方式，根据组合方式参考VdecCallBacker定义回调函数传入解码功能配置项，然后实例化VideoDecoder类，最后调用**decode**成员函数完成解码，获取数据。

视频解码接口调用流程参考如下：

图 3-10 视频解码接口调用流程



mxVision提供VideoDecoder类进行视频解码，关键步骤说明如下：

1. 定义输出数据组合形式。
   - 输出数据包含视频帧解码后得到的的Image类数据、当前解码帧的“frameId”和通道“channelId”。
   - 可根据需要选择获取以上哪些数据。
2. 定义输出回调函数。
   - 根据要获取的数据定义回调函数，在函数内组装自定义数据。
   - 回调函数输入参数固定为VideoDecodeCallBack形式，函数内可选择输出。
   - 请勿在回调函数内实现过于复杂的操作，建议只进行自定义获取解码数据操作。
3. 构造视频解码配置项。

   配置项及各项约束与支持情况请参考**VideoDecodeConfig**数据结构说明。
4. 实例化视频解码类。

   将配置好的VideoDecodeConfig传入构造函数接口，实例化视频解码类。
5. 调用**decode**接口对视频进行解码。

示例代码

如下提供关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import os
import numpy as np
import time
from mindx.sdk import base
from mindx.sdk.base import Image, ImageProcessor
from mindx.sdk.base import VideoDecoder, VideoDecodeConfig, VdecCallBacker
decoded_data_list = []
# 视频解码回调函数
def vdec_callback(decodedImage, channelId, frameId):
    # 解码完成的Image类存入列表中
    decoded_data_list.append(decodedImage)

# 初始化VdecCallBacker类并注册回调函数
vdecCallBacker = VdecCallBacker()
vdecCallBacker.registerVdecCallBack(vdec_callback)
# 初始化VideoDecodeConfig类并设置参数
vdecConfig = VideoDecodeConfig()
vdecConfig.skipInterval = 0
vdecConfig.inputVideoFormat = base.h264_main_level
vdecConfig.outputImageFormat = base.nv12
vdecConfig.width = 1920
vdecConfig.height = 1080
# 初始化VideoDecoder
videoDecoder = VideoDecoder(vdecConfig, vdecCallBacker, device_id, channel_id)
# 获取需解码视频帧文件名
srcDataList = ["frame-{}.data".format(i) for i in range(100)]
# 循环取帧解码
for i, fileName in enumerate(srcDataList):
# 读取视频帧数据存入file
file = np.fromfile(fileName, dtype='uint8')
# 视频帧数据解码
videoDecoder.decode(file, i)
```

3.4 模型推理

功能介绍

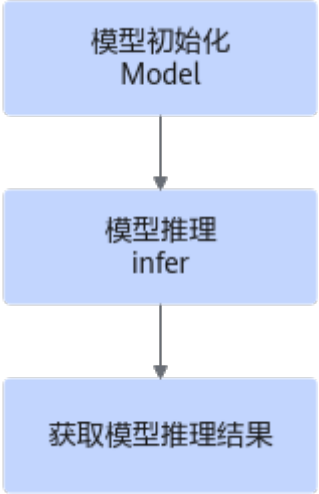
使用mxVision模型推理功能通过给定输入和指定模型，进行推理获得输出结果，支持om格式的模型推理，可使用ATC工具构建的动态Batch、动态分辨率和分档动态维度模型进行推理。模型推理输入为张量Tensor类型，由用户使用mxvision提供的接口构造使用。当前mxvision提供的python接口只支持同步推理。

接口调用流程

使用模型推理前，用户需准备好输入数据以及需要加载的模型，根据模型路径或者内存方式初始化**Model**类，通过调用Model类的**infer**接口获取模型推理结果。

模型推理调用流程参考如下：

图 3-11 模型推理接口调用流程



关键接口说明如下：

1. 模型初始化。

   用户需根据实际业务情况确认模型加载方式，选择**从文件加载模型**或**从内存加载模型**。如果从内存加载，需要先将模型文件读取到内存，可通过以下两种方式传入。

   - 从文件加载模型，可选择直接向Model接口传入模型路径，进行初始化。
   - 通过结构体ModelLoadOptV2中的“loadType”字段指定加载方式后，再传入Model接口，其中加载方式在模型方面区分从文件加载模型，还是从内存加载模型，在内存方面区分是由系统内部管理，还是由用户管理，具体可参见**ModelLoadOptV2**。

2. 调用**infer**接口获取模型推理结果。

示例代码

以下为功能特性关键步骤的代码示例，不可以直接拷贝运行，仅供参考。

```
import numpy as np
from mindx.sdk import base
from mindx.sdk.base import Tensor, Model
# 模型推理
# 构造输入Tensor（以二进制输入为例）
# 读取前处理好的numpy array二进制数据
input_array = np.load("preprocess_array.npy")
# 构造输入Tensor类并转移至device侧
input_tensor = Tensor(input_array)
input_tensor.to_device(device_id)
# 构造输入Tensor列表
input_tensors = [input_tensor]
# 模型路径
model_path = "resnet50_batchsize_1.om"
# 初始化Model类
model = Model(modelPath=model_path, deviceId=device_id)
# 执行推理
outputs = model.infer(input_tensors)
```

3.5 运行

运行样例前，需设置mxVison环境变量。

```
source {mxVison安装目录}/mxVision/set_env.sh
```

运行

用户准备好自己的源码文件“main.py”，以及Python运行环境。

使用**python3**执行文件，命令参考如下。

```
python3 main.py
```

说明

- 若程序执行出现libascendcl.so报错，请参见**8.1 程序执行出现libascendcl.so报错**解决。
- 若调用API参考（Python）视频解码/编码接口结束时出现core dumped现象，请参见**8.11 调用API参考（Python）视频解码/编码接口结束时出现core dumped现象**解决。

4 模型支持参考

4.1 模型支持列表
4.2 模型后处理配置参数
4.3 模型结构

4.1 模型支持列表

模型种类	模型框架	使用后处理动态库	获取途径
YOLOv3	TensorFlow	（tensorinfer框架）modelpostprocessors/libyolov3postprocess.so	模型代码获取地址 模型文件获取地址
ResNet-50	TensorFlow	（tensorinfer框架）modelpostprocessors/libresnet50postprocess.so	模型代码获取地址 模型文件获取地址
Faster Rcnn	TensorFlow	（tensorinfer框架）modelpostprocessors/libfasterrcnnpostprocess.so	模型文件获取地址

模型种类	模型框架	使用后处理动态库	获取途径
	MindSpore	（tensorinfer框架）modelpostprocessors/libfasterrcnnpostprocess.so	模型代码获取地址 模型文件获取地址
YOLOv4	Pytorch	（tensorinfer框架）modelpostprocessors/libyolov3postprocess.so	模型文件获取地址
SSD-VGG16	Caffe	（tensorinfer框架）modelpostprocessors/libssdvggpostprocess.so	无
SSD MobileNet v1 FPN	TensorFlow	（tensorinfer框架）modelpostprocessors/libssdmobilenetfpnpostprocess.so	
CRNN	TensorFlow	（tensorinfer框架）modelpostprocessors/libcrnnpostprocess.so	
YOLOv5	Pytorch	（tensorinfer框架）modelpostprocessors/libyolov3postprocess.so	
FasterRCNN-FPN/CascadeRCNN-FPN	Pytorch	（tensorinfer框架）modelpostprocessors/libfasterrcnnpostprocess.so	

模型种类	模型框架	使用后处理动态库	获取途径
ResNet-18	TensorFlow	（tensorinfer框架）modelpostprocessors/libresnet50postprocess.so	
CTPN	TensorFlow	（tensorinfer框架）modelpostprocessors/libctpnpostprocess.so	模型文件获取地址
CTPN	MindSpore	（tensorinfer框架）modelpostprocessors/libctpnpostprocess.so	模型代码获取地址 模型文件获取地址
DeepLabv3	MindSpore	（tensorinfer框架）modelpostprocessors/libdeeplabv3post.so	模型代码获取地址 模型文件获取地址
BERT-Base (Uncased)	TensorFlow	（tensorinfer框架）modelpostprocessors/libresnet50postprocess.so	模型文件获取地址
DeepLabv3+	Pytorch	（tensorinfer框架）modelpostprocessors/libdeeplabv3post.so	模型文件获取地址
U-Net	MindSpore	（tensorinfer框架）modelpostprocessors/libunetmindsporepostprocess.so	模型文件获取地址

模型种类	模型框架	使用后处理动态库	获取途径
Mask R-CNN	Pytorch	（tensorinfer框架）modelpostprocessors/libmaskrcnnmindsporepost.so	模型文件获取地址
FaceNet	TensorFlow	无需后处理。	模型文件获取地址
SSD MobileNet v1 FPN	MindSpore	（tensorinfer框架）modelpostprocessors/libSsdMobilenetFpn_MindsporePost.so	模型代码获取地址 模型文件获取地址
OpenPose	TensorFlow	（tensorinfer框架）modelpostprocessors/libopenposepostprocess.so	无
UNet++	MindSpore	（tensorinfer框架）modelpostprocessors/libunetmindsporepostprocess.so	模型代码获取地址 模型文件获取地址
RetinaNet	TensorFlow	（tensorinfer框架）modelpostprocessors/retinanetpostprocess.so	模型文件获取地址
HigherHRnet	Pytorch	（tensorinfer框架）modelpostprocessors/libhigherhrnetpostprocess.so	模型文件获取地址
YoloV7Detection	Pytorch	无	无
PPYOLOEPlus Detection	Paddle	无	无

4.2 模型后处理配置参数

各个模型所需要的配置参数见下表。

表 4-1 YOLOv3 模型后处理配置参数（yolov3_tf_bs1_fp16.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	80	无
BIASES_NUM	anchor宽高的数量（18表示9个anchor，每个对应一对宽高值）。	18	[0, 100]
BIASE	每两个数组成一个anchor的宽高值，例如10、13表示第一个anchor的宽、高值。	10，13，16，30，33，23，30，61，62，45，59，119，116，90，156，198，373，326	无
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.3	[0.0, 1.0]
OBJECTNESS_THRESH	是否为目标的阈值，大于阈值即认为是目标。	0.3	[0.0, 1.0]
IOU_THRESH	两个框的IOU阈值，超过阈值即认为同一个框。	0.45	[0.0, 1.0]
YOLO_TYPE	表示输出Tensor的个数，3表示有三个feature map输出。	3	[0, 16]
ANCHOR_DIM	每个feature map对应的anchor框数量。	3	[0, 16]
MODEL_TYPE	数据排布格式，0表示NHWC，1表示NCHW，2表示NCHWC。	0	无
FRAMEWORK	String类型，可选值有MindSpore，PyTorch，TensorFlow 和Caffe。	TensorFlow	无
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）	无

表 4-2 ResNet-50 模型后处理配置参数（resnet50_aipp_tf.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	1001	[0, 2000]
SOFTMAX	布尔型，是否需要在后处理中做softmax计算。	false	无
TOP_K	前K个可能性最大的类。	1	[0, 16]

表 4-3 FasterRcnn 模型后处理配置参数（faster_rcnn_uncut.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	91	[0, 1000]
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.5	[0.0, 1.0]
IOU_THRESH	两个框的IOU阈值，超过阈值即认为同一个框。	0.45	[0.0, 1.0]
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）	无
MODEL_TYPE	可选以下三个值： 0：original 1：nms_cut（模型未做非极大值抑制） 2：FPN	0	无
FRAMEWORK	可选以下三个值： ● TensorFlow ● MindSpore ● Pytorch	TensorFlow	无

参数名	描述	默认值	取值区间
NMS_FINISHED	modelinfer框架特有的属性值，为布尔值： ● false：模型本身不存在NMS算子，需后处理进行NMS操作。 ● true：模型本身存在NMS算子，不需后处理进行NMS操作。	true	无
注：“MODEL_TYPE”和“FRAMEWORK”参数配套说明如下： ● orginal配套TensorFlow框架。 ● nms_cut配套TensorFlow或MindSpore框架。 ● FPN配套Pytorch框架。			

表 4-4 ssd_vgg 模型后处理配置参数（ssd_vgg16_caffe_release.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	5
SCORE_THRESH	目标阈值。	0.4
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）

表 4-5 Ssd-Mobilenet-v1-Fpn 模型后处理配置参数（ssd_mobilenetv1_fpn.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	3
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.5

参数名	描述	默认值
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）

表 4-6 CRNN 模型后处理配置参数（crnn_ssh_2.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	0	[0, 10000]
OBJECT_NUM	可检测的字符数上限。	0	[0, 1000]
BLANK_INDEX	空白符的索引值。	0	[0, 10000]
WITH_ARGMAX	模型backbone是否已经做了argmax。	false	无

表 4-7 modelinfer 框架 ResNet 特征模型后处理配置参数（resnet_feature_caffe_release.cfg）

参数名	描述	默认值
ACTIVATION_FUNCTION	用来激活模型输出数据的激活函数。	None

表 4-8 modelinfer 框架 ResNet 多分类属性模型后处理配置参数（resnet_attribute_caffe_release.cfg）

参数名	描述	默认值
ATTRIBUTE_NUM	模型输出属性的数量。	5
ACTIVATION_FUNCTION	激活函数的类型，目前仅支持sigmoid函数。	无

参数名	描述	默认值
ATTRIBUTE_INDEX	模型输出属性的索引。请确保索引的数量与ATTRIBUTE_NUM值相等。	无

表 4-9 modelinfer 框架 ResNet 二分类属性模型后处理配置参数（resnet_attribute_caffe_release.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	5

表 4-10 modelinfer 框架 YOLOv4 模型后处理配置参数（yolov4_pt_bs1_fp16.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	80
BIASES_NUM	anchor宽高的数量（18表示9个anchor，每个对应一对宽高值）	18
BIASES	每两个数组成一个anchor的宽高值，例如10、13表示第一个anchor的宽、高值。	10，13，16，30，33，23，30，61，62，45，59，119，116，90，156，198，373，326
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.3
OBJECTNESS_THRESH	是否为目标的阈值，大于阈值即认为是目标。	0.3
IOU_THRESH	两个框的IOU阈值，超过阈值即认为同一个框。	0.45
YOLO_TYPE	表示输出Tensor的个数，3表示有三个feature map输出。	3
ANCHOR_DIM	每个feature map对应的anchor框数量。	3
MODEL_TYPE	数据排布格式，0表示NHWC，1表示NCHW。	0
FRAMEWORK_TYPE	模型框架，0表示Pytorch，1表示MindSpore。	0

参数名	描述	默认值
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）

表 4-11 YOLOv4 模型后处理配置参数（yolov4_pt_bs1_fp16.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	80
BIASES_NUM	anchor宽高的数量（18表示9个anchor，每个对应一对宽高值）。	18
BIASES	每两个数组成一个anchor的宽高值，例如10、13表示第一个anchor的宽、高值。	10，13，16，30，33，23，30，61，62，45，59，119，116，90，156，198，373，326
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.3
OBJECTNESS_THRESH	是否为目标的阈值，大于阈值即认为是目标。	0.3
IOU_THRESH	两个框的IOU阈值，超过阈值即认为同一个框。	0.45
YOLO_TYPE	表示输出Tensor的个数，3表示有三个feature map输出。	3
ANCHOR_DIM	每个feature map对应的anchor框数量。	3
MODEL_TYPE	数据排布格式，0表示NHWC，1表示NCHW，2表示NCHWC。	0
FRAMEWORK	String类型，可选值有MindSpore，PyTorch，TensorFlow 和Caffe。	MindSpore

参数名	描述	默认值
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）
YOLO_VERSION	使用的yolo模型版本。	（必选）YOLO_VERSION=4

表 4-12 YOLOv5 模型后处理配置参数（yolov5_pt_bs1_fp32.cfg）

参数名	描述	默认值	取值空间
CLASS_NUM	类别数量。	80	[0, 1000]
BIASES_NUM	anchor宽高的数量（18表示9个anchor，每个对应一对宽高值）。	18	[0, 1000]
BIASE	每两个数组成一个anchor的宽高值，例如10、13表示第一个anchor的宽、高值。	10，13，16，30，33，23，30，61，62，45，59，119，116，90，156，198，373，326	无
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.3	[0.0, 1.0]
OBJECTNESS_THRESH	是否为目标的阈值，大于阈值即认为是目标。	0.3	[0.0, 1.0]
IOU_THRESH	两个框的IOU阈值，超过阈值即认为同一个框。	0.45	[0.0, 1.0]
YOLO_TYPE	表示输出Tensor的个数，3表示有三个feature map输出。	3	[0, 1000]
ANCHOR_DIM	每个feature map对应的anchor框数量。	3	[0, 1000]
MODEL_TYPE	数据排布格式，0表示NHWC，1表示NCHW，2表示NCHWC（当前仅支持的PyTorch框架模型）。	2	[0, 1000]

参数名	描述	默认值	取值空间
FRAMEWORK	String类型，可选值有MindSpore，PyTorch，TensorFlow 和Caffe。	PyTorch	无
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）	无
YOLO_VERSION	使用的yolo模型版本。	（必选）YOLO_VERSION=5	无

表 4-13 modelinfer 框架 FasterRCNN-Fpn/CascadeRCNN-Fpn 模型后处理配置参数（fasterrcnn.cfg 或 cascadercnn.cfg）

参数名	描述	默认值
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标，实验使用0.3。	0.5
FPN_SWITCH	FPN开关，这两个模型均要设为true。	false

表 4-14 DeepLabV3+（TensorFlow）模型后处理配置参数（deeplabv3.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	21
FRAMEWORK_TYPE	深度学习框架类型选择。	TensorFlow框架选择0

表 4-15 CTPN 模型后处理配置参数（ctpn_tf.cfg）

参数名	描述	默认值	取值区间
IS_ORIENTED	是否进行检测框倾斜。	false	无
BOX_IOU_THRESH	检测小框的iou阈值。	0.7	[0.0, 1.0]
TEXT_IOU_THRESH	最终文本框的iou阈值。	0.2	[0.0, 1.0]

参数名	描述	默认值	取值区间
TEXT_PROPOSALS_MIN_SCORE	检测小框的最小分数过滤。	0.7	[0.0, 1.0]
LINE_MIN_SCORE	最终文本框的最小分数过滤。	0.9	[0.0, 1.0]
IS_MINDSPORE	是否为MindSpore框架。	false	无

表 4-16 CTPN 模型后处理配置参数（ctpn_mindspore.cfg）

参数名	描述	默认值	取值区间
IS_ORIENTED	是否进行检测框倾斜。	false	无
BOX_IOU_THRESH	检测小框的iou阈值。	0.7	[0.0, 1.0]
TEXT_IOU_THRESH	最终文本框的iou阈值。	0.2	[0.0, 1.0]
TEXT_PROPOSALS_MIN_SCORE	检测小框的最小分数过滤。	0.7	[0.0, 1.0]
LINE_MIN_SCORE	最终文本框的最小分数过滤。	0.9	[0.0, 1.0]
IS_MINDSPORE	是否为MindSpore框架。	true	无

表 4-17 ResNet-18 模型后处理配置参数（resnet18_aipp_tf.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	2

表 4-18 DeepLabv3（MindSpore）模型后处理配置参数（deeplabv3.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	21
MODEL_TYPE	模型推理输出数据的排布格式，0表示NHWC，1表示NCHW。	1

参数名	描述	默认值
FRAMEWORK_TYPE	深度学习框架类型选择。	MindSpore框架选择2

表 4-19 BERT-Base (Uncased) 模型后处理配置参数（bert.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	2
CHECK_MODEL	检查模型兼容性。	false

表 4-20 DeepLabV3+（Pytorch）模型后处理配置参数（deeplabv3.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	21
CHECK_MODEL	检查模型兼容性。	true
MODEL_TYPE	模型推理输出数据的排布格式，0表示NHWC，1表示NCHW。	1
FRAMEWORK_TYPE	深度学习框架类型选择。	Pytorch框架选择1

表 4-21 U-Net 模型后处理配置参数（unet_simple.cfg）

参数名	描述	默认值
CLASS_NUM	类别数量。	2
POST_TYPE	模型后处理方式，0表示对模型logits输出（NHWC型）做argmax，1表示对模型argmax输出结果（NHW型）透传。	1
RESIZE_TYPE	对像素图做插值还原的方式，暂只支持两种方式： 0：不做插值还原。 1：最邻近插值还原。	1

表 4-22 Mask R-CNN 模型后处理配置参数（mask_rcnn_2017.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	推理类别总数，背景不计算在内。	80	[0, 100]
SCORE_THRESH	置信度得分阈值，可根据业务场景调整。	0.7	[0.0, 1.0]
IOU_THRESH	IOU阈值，可根据业务场景调整。	0.5	[0.0, 1.0]
RPN_MAX_NUM	Region Proposal Network最大个数。	1000	[0, 1000]
MAX_PER_IMG	按置信度排序，每张图预测框的最大值。	128	[0, 150]
MASK_THREAD_BINARY	输入RCNN的掩码阈值。	0.5	[0.0, 1.0]
MASK_SHAPE_SIZE	mask_rcnn中掩码的形状，只支持单参数表示正方形。	28	[0, 100]
MODEL_TYPE	可选以下两个值。 0：MindSpore 1：Pytorch	0	无
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）	无

表 4-23 Ssd_Mobilenet_v1_Fpn_for_MindSpore 模型后处理配置参数（ssd_mobilenetv1_fpn.cfg））

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	81	[0, 100]
SCORE_THRESH	目标是否为某种类别物体的阈值，大于阈值即认为是该目标。	0.5	[0.0, 1.0]
IOU_THRESH	目标重合程度的阈值，大于阈值即认为两个目标框对应同一个目标。	0.6	[0.0, 1.0]

参数名	描述	默认值	取值区间
SEPARATE_SCORE_THRESH	各个类别对应的阈值。	CLASS_NUM（数量）个SCORE_THRESH（阈值），以逗号为分隔符。（阈值数量等于CLASS_NUM。）	无

表 4-24 OpenPose 模型后处理配置参数（openpose.cfg）

参数名	描述	默认值	取值区间
KEYPOINT_NUM	关键点个数，加上背景（背景算一个）。	19	[0, 100]
FILTER_SIZE	高斯滤波核的长（或宽）。	25	[0, 100]
SIGMA	高斯滤波核的方差。	3	[0, 10]

表 4-25 HigherHRnet 模型后处理配置参数（higherhrnet.cfg）

参数名	描述	默认值	取值区间
KEYPOINT_NUM	关键点个数。	17	[0, 20]
SCORE_THRESH	关键点阈值。	0.1	[0.0, 1.0]

表 4-26 Unet++模型后处理配置参数（unet_nested.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	类别数量。	3	无
POST_TYPE	模型后处理方式，0表示对模型logits输出（NHWC型）做argmax，1表示对模型argmax输出结果（NHW型）透传。	1	[0, 16]

参数名	描述	默认值	取值区间
RESIZE_TYPE	对像素图做插值还原的方式，暂只支持两种方式： 0：不做插值还原。 1：最邻近插值还原。	1	[0, 16]

表 4-27 RetinaNet 模型后处理配置参数（retinanet_tf.cfg）

参数名	描述	默认值	取值区间
CLASS_NUM	代表数据集类别数量（默认COCO数据集是80类）。	80	[0, 100]
MODEL_TYPE	代表模型类别，目前仅支持TensorFlow模型。	0	[0, 100]
SCORE_THRESH	代表分数阈值。	0.5	[0.0, 1.0]

4.3 模型结构

YOLOv3

- 可接受的模型与YOLOv3具有类似的输出张量。一般有三个输出张量（YOLOv3-Tiny只有两个，配置参数“YOLO_TYPE”需设为“2”），分别是8倍，16倍与32倍降采样后的特征层。
- 每个输出张量的第一维与模型所支持的最大batchsize相同，按照NHWC或NCHW，输出张量形状略有差异，**W**与**H**分别等于模型输入宽高除以8，16，32，**C**等于先验框个数**anchorDim=3 * (边框坐标4 + 边框置信度1 + 类别数80)** 。
- 配置参数**MODEL_TYPE = 0**时，采取NHWC，**MODEL_TYPE = 1**时，采取NCHW。

图 4-1 NHWC 型

```
Shape of output pool 0 (name: yolov3/yolov3_head/feature_map_1:0) of model is as follow:
    shape 0: 1
    shape 1: 13
    shape 2: 13
    shape 3: 255
Shape of output pool 1 (name: yolov3/yolov3_head/feature_map_2:0) of model is as follow:
    shape 0: 1
    shape 1: 26
    shape 2: 26
    shape 3: 255
Shape of output pool 2 (name: yolov3/yolov3_head/feature_map_3:0) of model is as follow:
    shape 0: 1
    shape 1: 52
    shape 2: 52
    shape 3: 255
Shape of input pool 0 of model is as follow:
    shape 0: 1
    shape 1: 416
    shape 2: 416
    shape 3: 3
```

图 4-2 NCHW 型

```
Shape of output pool 0 (name: 289:0) of model is as follow:
    shape 0: 1
    shape 1: 21
    shape 2: 19
    shape 3: 19
Shape of output pool 1 (name: 293:0) of model is as follow:
    shape 0: 1
    shape 1: 21
    shape 2: 38
    shape 3: 38
Shape of output pool 2 (name: 297:0) of model is as follow:
    shape 0: 1
    shape 1: 21
    shape 2: 76
    shape 3: 76
Shape of input pool 0 of model is as follow:
    shape 0: 1
    shape 1: 608
    shape 2: 608
    shape 3: 3
```

FasterRCNN

- 支持两种结构的模型，原生FasterRCNN模型，以及对模型中NMS（非最大值抑制）进行裁剪后的模型。
- 配置参数**NMS_FINISHED = 0**时，采取后者，**NMS_FINISHED = 1**时采取前者。
- 原生模型：

  有四个输出张量，分别为目标数，置信度，坐标框和类别ID。

  图 4-3 FasterRCNN 原生模型

```
Shape of output pool 0 (name: num_detections:0) of model is as follow:
    shape 0: 1
Shape of output pool 1 (name: detection_scores:0) of model is as follow:
    shape 0: 1
    shape 1: 100
Shape of output pool 2 (name: detection_boxes:0) of model is as follow:
    shape 0: 1
    shape 1: 100
    shape 2: 4
Shape of output pool 3 (name: detection_classes:0) of model is as follow:
    shape 0: 1
    shape 1: 100
Shape of input pool 0 of model is as follow:
    shape 0: 1
    shape 1: 608
    shape 2: 608
    shape 3: 3
```

- 裁剪NMS后：

  有三个输出张量，分别为目标数，每种类别目标可能的坐标框，每种类别目标框的置信度。

图 4-4 FasterRCNN 裁剪 NMS 后

```
Shape of output pool 0 (name: num_boxes:0) of model is as follow:
    shape 0: 1
Shape of output pool 1 (name: SecondStagePostprocessor/Reshape_4:0) of model is as follow:
    shape 0: 1
    shape 1: 100
    shape 2: 90
    shape 3: 4
Shape of output pool 2 (name: SecondStagePostprocessor/Reshape_6:0) of model is as follow:
    shape 0: 1
    shape 1: 100
    shape 2: 90
Shape of output pool 3 (name: SecondStagePostprocessor/ToFloat:0) of model is as follow:
    shape 0: 1
    shape 1: 4
Shape of input pool 0 of model is as follow:
    shape 0: 1
    shape 1: 608
    shape 2: 608
    shape 3: 3
```

SSD MobileNet v1 FPN

SSD MobileNet v1 FPN与FasterRCNN原生模型类似，有四个输出张量，分别为目标数，置信度，坐标框和类别ID。

图 4-5 SSD MobileNet v1 FPN

```
Shape of output pool 0 (name: num_detections:0) of model is as follow:
    shape 0: 1
Shape of output pool 1 (name: detection_scores:0) of model is as follow:
    shape 0: 1
    shape 1: 300
Shape of output pool 2 (name: detection_boxes:0) of model is as follow:
    shape 0: 1
    shape 1: 300
    shape 2: 4
Shape of output pool 3 (name: detection_classes:0) of model is as follow:
    shape 0: 1
    shape 1: 300
Shape of input pool 0 (name: Preprocessor/sub) of model is as follow:
    shape 0: 1
    shape 1: 1536
    shape 2: 2048
    shape 3: 3
```

SSD-VGG16

SSD-VGG16有两个输出张量，第一个输出张量是目标数，第二个输出张量的为目标框的信息[batch, keep_top_k, 8]，其中的“8”表示batchID、label（classID）、score（类别概率）、xmin、ymin、xmax、ymax、null。

图 4-6 SSD-VGG16

```
Shape of output pool 0 (name: detection_out:0:out_boxnum) of model is as follow:
    shape 0: 1
    shape 1: 8
Shape of output pool 1 (name: detection_out:1:y) of model is as follow:
    shape 0: 1
    shape 1: 200
    shape 2: 8
Shape of input pool 0 (name: data) of model is as follow:
    shape 0: 1
    shape 1: 800
    shape 2: 1200
    shape 3: 3
```

CRNN

CRNN的输出张量仅一个，第一维是batchsize，第二维即其所能检测的目标数上限，代表其识别到的每个目标的类别ID（包含占位符）。

图 4-7 CRNN

```
Shape of output pool 0 (name: scores:0) of model is as follow:
    shape 0: 1
    shape 1: 32
Shape of input pool 0 (name: input) of model is as follow:
    shape 0: 1
    shape 1: 32
    shape 2: 128
    shape 3: 3
```

ResNet-50

ResNet-50仅需一个输出张量，第一维是batchsize，第二维与类别数一致，为模型特征层softmax之后的结果。第二个输出张量为概率最大的类别对应的类别ID。

图 4-8 ResNet-50

```
Shape of output pool 0 (name: softmax_tensor:0) of model is as follow:
    shape 0: 1
    shape 1: 1001
Shape of output pool 1 (name: ArgMax:0) of model is as follow:
    shape 0: 1
Shape of input pool 0 (name: input_tensor) of model is as follow:
    shape 0: 1
    shape 1: 224
    shape 2: 224
    shape 3: 3
```

YOLOv4

YOLOv4与YOLOv3模型类似，有三个输出张量，分别是8倍，16倍与32倍降采样后的特征层。

图 4-9 YOLOv4

```
Shape of output tensor: 0 (name: output_0_trans_Cast_351_0) of model is as follow:
   dim 0: 1
   dim 1: 46
   dim 2: 46
   dim 3: 3
   dim 4: 11
Shape of output tensor: 1 (name: output_1_trans_Cast_352_0) of model is as follow:
   dim 0: 1
   dim 1: 92
   dim 2: 92
   dim 3: 3
   dim 4: 11
Shape of output tensor: 2 (name: output_2_trans_Cast_353_0) of model is as follow:
   dim 0: 1
   dim 1: 184
   dim 2: 184
   dim 3: 3
   dim 4: 11
Shape of input tensor: 0 (name: x) of model is as follow:
   dim 0: 1
   dim 1: 1472
   dim 2: 1472
   dim 3: 3
```

YOLOv5

- YOLOv5有三个输出张量，分别是8倍，16倍与32倍降采样后的特征层。
- 输出张量按照N(C0)HW(C1)的形式排布，**W**与**H**分别等于模型输入宽高除以8、16、32，**C**等于先验框个数**anchorDim=3 * (边框坐标4 + 边框置信度1 + 类别数80)** 。

图 4-10 YOLOv5

```
Shape of output tensor: 0 (name: Transpose_271:0) of model is as follow:
   dim 0: 1
   dim 1: 3
   dim 2: 80
   dim 3: 80
   dim 4: 85
Shape of output tensor: 1 (name: Transpose_287:0) of model is as follow:
   dim 0: 1
   dim 1: 3
   dim 2: 40
   dim 3: 40
   dim 4: 85
Shape of output tensor: 2 (name: Transpose_303:0) of model is as follow:
   dim 0: 1
   dim 1: 3
   dim 2: 20
   dim 3: 20
   dim 4: 85
```

FasterRCNN-Fpn/CascadeRCNN-Fpn

模型有两个输出张量，分别是5 * 100的预测框和置信度（x0, y0, x1, y1,confidence）(其中坐标为左上和右下的检测框的坐标)，1 * 100是每个类别的分值。输入为固定size的RGB图片：3 * 1216 * 1216 。

图 4-11 FasterRCNN-Fpn/CascadeRCNN-Fpn

```
Shape of output tensor: 0 (name: Add_836:0) of model is as follow:
   dim 0: 100
   dim 1: 5
Shape of output tensor: 1 (name: Reshape_833:0) of model is as follow:
   dim 0: 100
 Shape of input tensor: 0 (name: input) of model is as follow:
    dim 0: 1
    dim 1: 1216
    dim 2: 1216
    dim 3: 3
```

CTPN（TensorFlow）

CTPN（TensorFlow）模型有两个输出张量，分别是38 * 67 * 40的预测小框，相当于38 * 67 * 4的每个像素点生成10个小框，38 * 67 * 20的预测分数，相当于38 * 67 * 2的每个像素点生成10个预测分数。输入为固定size的RGB图片：3 * 608 * 1072 。

图 4-12 CTPN（TensorFlow)

```
I0112 12:22:30.351997 38861 ModelPostProcessorBase.cpp:79] Shape of output tensor: 0 (name: bbox_pred/Reshape_1:0) of m
odel is as follow:
I0112 12:22:30.352008 38861 ModelPostProcessorBase.cpp:83]    shape 0: 1
I0112 12:22:30.352020 38861 ModelPostProcessorBase.cpp:83]    shape 1: 38
I0112 12:22:30.352027 38861 ModelPostProcessorBase.cpp:83]    shape 2: 67
I0112 12:22:30.352042 38861 ModelPostProcessorBase.cpp:83]    shape 3: 40
I0112 12:22:30.352053 38861 ModelPostProcessorBase.cpp:79] Shape of output tensor: 1 (name: cls_pred/Reshape_1:0) of mo
del is as follow:
I0112 12:22:30.352061 38861 ModelPostProcessorBase.cpp:83]    shape 0: 1
I0112 12:22:30.352067 38861 ModelPostProcessorBase.cpp:83]    shape 1: 38
I0112 12:22:30.352074 38861 ModelPostProcessorBase.cpp:83]    shape 2: 67
I0112 12:22:30.352080 38861 ModelPostProcessorBase.cpp:83]    shape 3: 20
I0112 12:22:30.352089 38861 ModelPostProcessorBase.cpp:90] Shape of input tensor: 0 (name: input_image) of model is as
follow:
I0112 12:22:30.352095 38861 ModelPostProcessorBase.cpp:94]     shape 0: 1
I0112 12:22:30.352102 38861 ModelPostProcessorBase.cpp:94]     shape 1: 608
I0112 12:22:30.352109 38861 ModelPostProcessorBase.cpp:94]     shape 2: 1072
I0112 12:22:30.352116 38861 ModelPostProcessorBase.cpp:94]     shape 3: 3
I0112 12:22:30.352128 38861 ObjectPostProcessorBase.cpp:40] End to initialize ObjectPostProcessor.
```

CTPN（MindSpore）

CTPN（MindSpore）模型有两个输出张量，分别是1000个预测小框，5个维度分别是四个坐标和分数， 另一个1000是每个小框的类别，分别为1、0，表示前景还是后景。输入则是固定size的RGB图片：3 * 576 * 960 。

图 4-13 CTPN（MindSpore）

```
Begin to initialize MxpiModelTensorInfer(mxpi_tensorinfer0).
dataSource_(mxpi_imageresize0).
 Shape of output tensor: 0 (name: output_0_Default/network-CTPN/proposal_generator_test-Proposal/GatherNd-op134_0) of model is
    dim 0: 1000
    dim 1: 5
 Shape of output tensor: 1 (name: output_1_Default/network-CTPN/proposal_generator_test-Proposal/GatherNd-op135_0) of model is
    dim 0: 1000
 Shape of input tensor: 0 (name: img_data) of model is as follow:
    dim 0: 1
    dim 1: 576
    dim 2: 960
    dim 3: 3
 Shape of input tensor: 1 (name: img_metas) of model is as follow:
    dim 0: 1
    dim 1: 4
End to initialize MxpiModelTensorInfer(mxpi_tensorinfer0).
```

ResNet-18+

ResNet-18+模型输入是1 * 408 * 64 * 3大小的张量。输出是1 * 2的张量，表示每个样本的分类概率。

图 4-14 ResNet-18+

```
Shape of output tensor: 0 (name: probs:0) of model is as follow:
    dim 0: 1
    dim 1: 2
 Shape of input tensor: 0 (name: input_data) of model is as follow:
    dim 0: 1
    dim 1: 408
    dim 2: 64
    dim 3: 3
```

BERT-Base（Uncased）

BERT-Base（Uncased）模型输入有三个张量，shape都是1 * 128。1表示batchsize，128表示句长。

模型输出有一个张量，1 * 2，表示每个分类类别的概率。

图 4-15 BERT-Base（Uncased）

```
dataSources (appsrc0,appsrc1,appsrc2are read from config file in plugin(mxpi_tensorinfer0).
Shape of output tensor: 0 (name: loss/Softmax:0, dtype: TENSOR_DTYPE_FLOAT32) of model is as follow:
   dim 0: 1
   dim 1: 2
Shape of input tensor: 0 (name: input_ids_1, dtype: TENSOR_DTYPE_INT32) of model is as follow:
   dim 0: 1
   dim 1: 128
Shape of input tensor: 1 (name: input_mask_1, dtype: TENSOR_DTYPE_INT32) of model is as follow:
   dim 0: 1
   dim 1: 128
Shape of input tensor: 2 (name: segment_ids_1, dtype: TENSOR_DTYPE_INT32) of model is as follow:
   dim 0: 1
   dim 1: 128
```

DeeplabV3+（TensorFlow）

DeeplabV3+（TensorFlow）模型输出有一个张量，是1 * 513 * 513 * 21的NHWC排布方式，物理含义相当于每个像素点的分类概率。原始输入图片是动态shape的RGB图片。模型输入是1 * 513 * 513 * 3的张量。

图 4-16 DeeplabV3+（TensorFlow）

```
Shape of output tensor: 0 (name: ResizeBilinear_3:0) of model is as follow:
   dim 0: 1
   dim 1: 513
   dim 2: 513
   dim 3: 21
] Shape of input tensor: 0 (name: sub_7) of model is as follow:
]    dim 0: 1
]    dim 1: 513
]    dim 2: 513
]    dim 3: 3
```

DeepLabV3（MindSpore）

DeepLabV3（MindSpore）模型输入是NHWC排布的，输出以NCHW排布。

图 4-17 DeepLabV3（MindSpore）

```
Shape of output tensor: 0 (name: output_0_trans_Cast_1547_0) of model is as follow:
   dim 0: 1
   dim 1: 21
   dim 2: 513
   dim 3: 513
Shape of input tensor: 0 (name: input_data) of model is as follow:
   dim 0: 1
   dim 1: 513
   dim 2: 513
   dim 3: 3
```

DeepLabV3（Pytorch）

DeepLabV3（Pytorch）模型输入以NHWC排布，输出为NCHW排布。

图 4-18 DeepLabV3（Pytorch）

```
I0506 04:33:03.531086 61645 MpModelInfer.cpp:102] Shape of output tensor: 0 (name: Resize_429:0) of model is as follow:
I0506 04:33:03.531167 61645 MpModelInfer.cpp:106]    dim 0: 1
I0506 04:33:03.531186 61645 MpModelInfer.cpp:106]    dim 1: 21
I0506 04:33:03.531198 61645 MpModelInfer.cpp:106]    dim 2: 513
I0506 04:33:03.531208 61645 MpModelInfer.cpp:106]    dim 3: 513
I0506 04:33:03.531216 61645 MpModelInfer.cpp:112] Shape of input tensor: 0 (name: actual_input_1) of model is as follow:
I0506 04:33:03.531224 61645 MpModelInfer.cpp:116]    dim 0: 1
I0506 04:33:03.531234 61645 MpModelInfer.cpp:116]    dim 1: 513
I0506 04:33:03.531242 61645 MpModelInfer.cpp:116]    dim 2: 513
I0506 04:33:03.531251 61645 MpModelInfer.cpp:116]    dim 3: 3
```

Unet（MindSpore）

Unet（MindSpore）模型输出Tensor为NCHW，其中N为1，C为2，作为mxVision后处理输入。

1. 在C通道上进行argmax，得到最大概率值的索引值，生成数值为0和1的二维数组。
2. 判断模型输出Tensor的HW是否与输入原图尺寸一致，若一致则直接输出argmax后的二维数组，否则进行最近邻插值到输入图片尺寸。

图 4-19 Unet（MindSpore）

```
Shape of output tensor: 0 (name: output_0_trans_Cast_57_0, dtype: TENSOR_DTYPE_FLOAT32) of model is as follow:
   dim 0: 1
   dim 1: 2
   dim 2: 96
   dim 3: 96
Shape of input tensor: 0 (name: inputs, dtype: TENSOR_DTYPE_UINT8) of model is as follow:
   dim 0: 1
   dim 1: 96
   dim 2: 96
   dim 3: 3
```

Mask R-CNN（TensorFlow）

Mask R-CNN（TensorFlow）模型输入张量为4维NHWC（1 * 480 * 640 * 3）排布格式。

- N为批数量。
- H为输入图像的高度（480）。
- W为输入图像的宽度（640）。

Mask R-CNN（TensorFlow）模型输出张量有5个（tensor[0]~tensor[4]）。

- tensor[0]的维度为1维（1），第一维长度为1，代表模型检测出的目标的个数。
- tensor[1]的维度为2维（1 * 100），第一维长度为1，代表批数量。第二维长度为100，代表前top100个目标的置信度分数。
- tensor[2]的维度为3维为（1 * 100 * 4），第一维长度为1，代表批数量。第二维长度为100，代表前top100个目标框。第三维长度为4，代表目标框的四个顶点坐标(x0, y0, x1, y1)。
- tensor[3]的维度为4维（1 * 100 * 33 * 33），第一维长度为1，代表批数量。第二维长度为100,代表前top100个目标框。第三维和第四维代表一张33 * 33大小的掩码图。
- tensor[4]的维度为2维（1 * 100），第一维长度为1，代表批数量。第二维长度为100，代表前top100目标的分类类别。

图 4-20 Mask R-CNN（TensorFlow）

```
I0618 16:06:58.208107 123662 MxpiTensorInfer.cpp:330] Shape of output tensor: 0 (name: num_detections:0, dtype: TENSOR_DTYPE_FLOAT16) of model is as follow:
I0618 16:06:58.208148 123662 MxpiTensorInfer.cpp:333]    dim 0: 1
I0618 16:06:58.208163 123662 MxpiTensorInfer.cpp:330] Shape of output tensor: 1 (name: detection_scores:0, dtype: TENSOR_DTYPE_FLOAT16) of model is as follow:
I0618 16:06:58.208179 123662 MxpiTensorInfer.cpp:333]    dim 0: 1
I0618 16:06:58.208190 123662 MxpiTensorInfer.cpp:333]    dim 1: 100
I0618 16:06:58.208202 123662 MxpiTensorInfer.cpp:330] Shape of output tensor: 2 (name: detection_boxes:0, dtype: TENSOR_DTYPE_FLOAT16) of model is as follow:
I0618 16:06:58.208215 123662 MxpiTensorInfer.cpp:333]    dim 0: 1
I0618 16:06:58.208226 123662 MxpiTensorInfer.cpp:333]    dim 1: 100
I0618 16:06:58.208237 123662 MxpiTensorInfer.cpp:333]    dim 2: 4
I0618 16:06:58.208248 123662 MxpiTensorInfer.cpp:330] Shape of output tensor: 3 (name: detection_masks:0, dtype: TENSOR_DTYPE_FLOAT16) of model is as follow:
I0618 16:06:58.208261 123662 MxpiTensorInfer.cpp:333]    dim 0: 1
I0618 16:06:58.208271 123662 MxpiTensorInfer.cpp:333]    dim 1: 100
I0618 16:06:58.208282 123662 MxpiTensorInfer.cpp:333]    dim 2: 33
I0618 16:06:58.208292 123662 MxpiTensorInfer.cpp:333]    dim 3: 33
I0618 16:06:58.208302 123662 MxpiTensorInfer.cpp:330] Shape of output tensor: 4 (name: detection_classes:0, dtype: TENSOR_DTYPE_FLOAT16) of model is as follow:
I0618 16:06:58.208317 123662 MxpiTensorInfer.cpp:333]    dim 0: 1
I0618 16:06:58.208326 123662 MxpiTensorInfer.cpp:333]    dim 1: 100
I0618 16:06:58.208338 123662 MxpiTensorInfer.cpp:338] Shape of input tensor: 0 (name: image_tensor, dtype: TENSOR_DTYPE_UINT8) of model is as follow:
I0618 16:06:58.208351 123662 MxpiTensorInfer.cpp:341]    dim 0: 1
I0618 16:06:58.208362 123662 MxpiTensorInfer.cpp:341]    dim 1: 480
I0618 16:06:58.208372 123662 MxpiTensorInfer.cpp:341]    dim 2: 640
I0618 16:06:58.208382 123662 MxpiTensorInfer.cpp:341]    dim 3: 3
I0618 16:06:58.208393 123662 MxpiTensorInfer.cpp:348] Input size : 1
```

FaceNet（TensorFlow）

FaceNet（TensorFlow）输入张量的形状为 NHWC （1*160*160*3），数据类型为UINT8。

- N为批数量。
- H代表输入图像的高度160。
- W代表输入图像的宽度160。
- C代表图像的通道数。

输出张量为目标图像对应的特征向量，形状为1 * 512，第一维代表批数量，第二维为特征向量的长度512，数据类型为FLOAT32。

图 4-21 FaceNet（TensorFlow）

```
Shape of output tensor: 0 (name: embeddings:0, dtype: TENSOR_DTYPE_FLOAT32) of model is as follow:
    dim 0: 1
    dim 1: 512
Shape of input tensor: 0 (name: input, dtype: TENSOR_DTYPE_UINT8) of model is as follow:
    dim 0: 1
    dim 1: 160
    dim 2: 160
    dim 3: 3
Input size : 1
```

SSD MobileNet v1 FPN（MindSpore）

SSD MobileNet v1 FPN（MindSpore）有2个输出张量，分别为坐标框和置信度。

图 4-22 SSD MobileNet v1 FPN（MindSpore）

```
Shape of output tensor: 0 (name: output_0_Default/Maximum-op62Default/Minimum-op61_0, dtype: TENSOR_DTYPE_FLOAT32) of mod

    dim 0: 1
    dim 1: 392832
    dim 2: 4
Shape of output tensor: 1 (name: output_1_Default/network-SsdMobilenetV1Fpn/Sigmoid-op295_0, dtype: TENSOR_DTYPE_FLOAT32)

    dim 0: 1
    dim 1: 392832
    dim 2: 5
Shape of input tensor: 0 (name: x, dtype: TENSOR_DTYPE_UINT8) of model is as follow:
    dim 0: 1
    dim 1: 1536
    dim 2: 2048
    dim 3: 3
```

OpenPose

OpenPose输出张量为[batch, outputHeight, outputWidth, channel]，“outputHeight”表示输出图像的高，“outputWidth”表示输出图像的宽，“channel”由两部分组成，前1/3为heat mat，后2/3为paf mat。此模型输出为[1, 54, 46, 57]。

图 4-23 OpenPose

```
Shape of output tensor: 0 (name: Openpose/concat_stage7:0) of model is as follow:
   dim 0: 1
   dim 1: 54
   dim 2: 46
   dim 3: 57
 Shape of input tensor: 0 (name: image) of model is as follow:
    dim 0: 1
    dim 1: 432
    dim 2: 368
    dim 3: 3
```

Unet++（MindSpore）

Unet++（MindSpore）模型后处理有一个NCHW的输入Tensor，为模型进行argmax操作之后经过aipp处理后输入给后处理模块；有一个NHW的输出Tensor，由于模型已经做了argmax，C通道已经计算在Tensor HW对应的值。

图 4-24 Unet++（MindSpore）

```
Shape of output tensor: 0 (name: output_0_Default/Argmax-op16_0, dtype: TENSOR_DTYPE_INT32) of model is as follow:
   dim 0: 1
   dim 1: 192
   dim 2: 256
Shape of input tensor: 0 (name: x, dtype: TENSOR_DTYPE_UINT8) of model is as follow:
   dim 0: 1
   dim 1: 192
   dim 2: 256
   dim 3: 3
```

5 API 参考（C++）

5.1 使用须知
5.2 多进程编写规范
5.3 头文件列表说明
5.4 数据结构及枚举类型
5.5 基础组件层
5.6 初始化和去初始化
5.7 媒体数据处理
5.8 模型推理
5.9 异步调用
5.10 模型后处理

5.1 使用须知

须知

本章节开放的类，请遵循同一个类实例在单一线程中使用的原则，请勿将同一个类实例在不同的线程中使用。

API 划分

mxVision C++ API目前分为三大部分：面向昇腾硬件底层接口的基础组件层、面向插件开发和流程编排的编程框架层。

接口管理规范

- 标记有宏 SDK_AVAILABLE_FOR_OUT 的接口，表示对用户开放。
- 标记有宏 SDK_AVAILABLE_FOR_IN 的接口，表示内部使用，请用户不要使用。

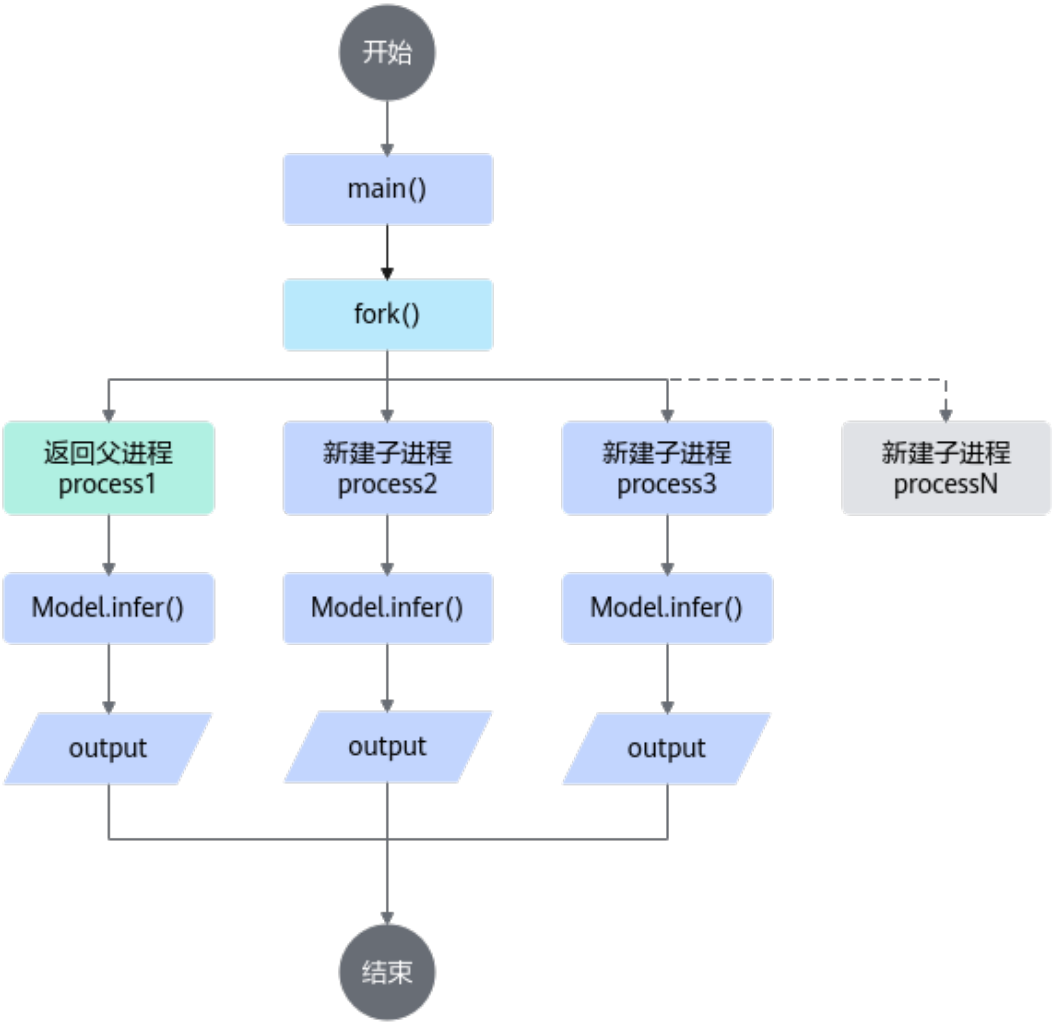
- 标记有宏 SDK_UNAVAILABLE_FOR_OTHER 的接口，表示为隐藏，用户无法使用。
- 标记有宏 SDK_DEPRECATED_FOR 的接口，表示在未来版本会弃用，建议使用替代接口。

5.2 多进程编写规范

避免父子进程嵌套

对父子进程事先进行职责区分，父进程负责创建子进程，子进程负责调度推理，避免父子进程同时调度推理，造成资源抢占问题。在实际使用中，请在fork()执行后再在返回的父子进程中执行模型推理。程序逻辑示意图如图5-1所示。

图 5-1 避免父子进程嵌套



采用不同 Device 执行推理任务

若业务上需要父子进程嵌套，则规范对于Device的指定，避免父子进程同时在同一Device上执行推理，而是将二者的推理指定到不同Device上。程序逻辑如图5-2、图5-3所示。

图 5-2 父子进程推理指定到不同 Device 上

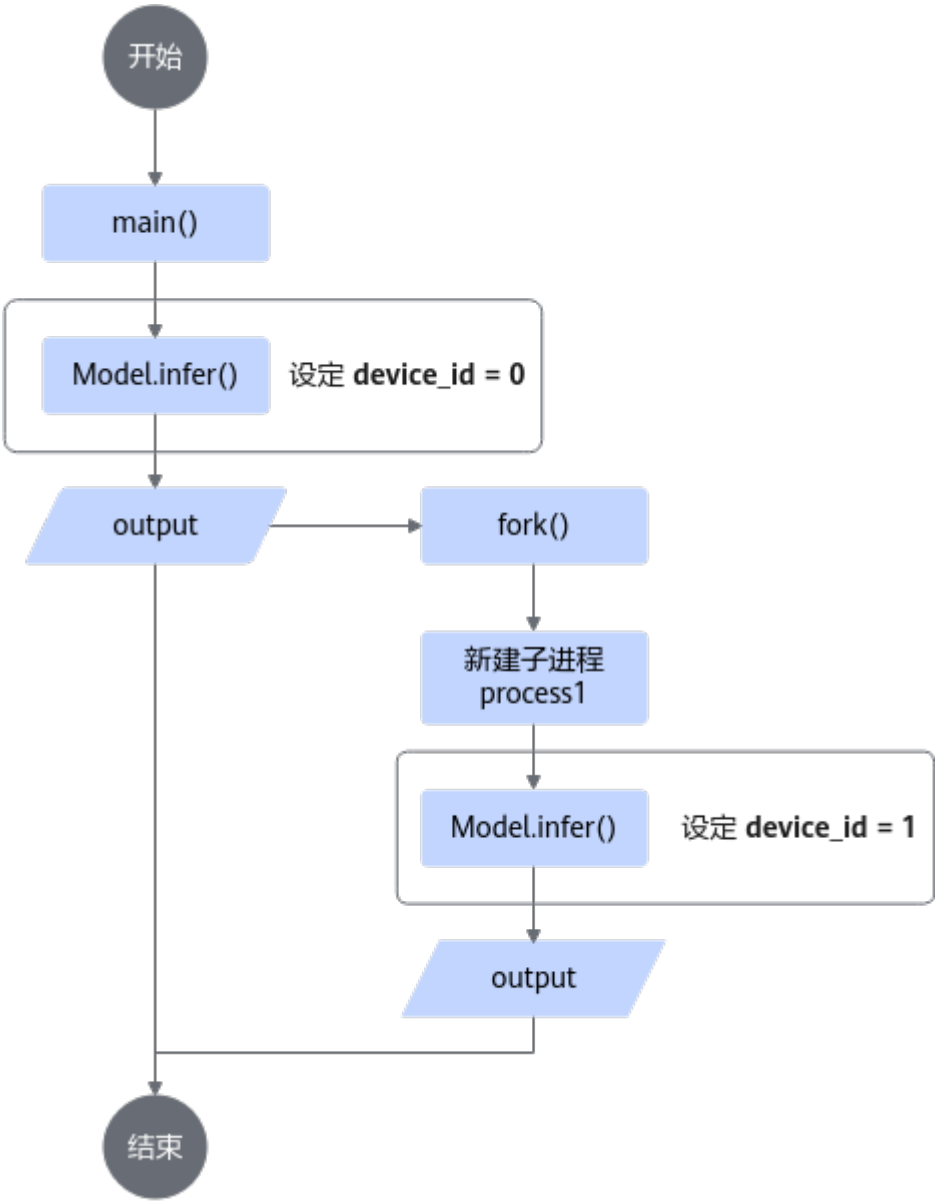
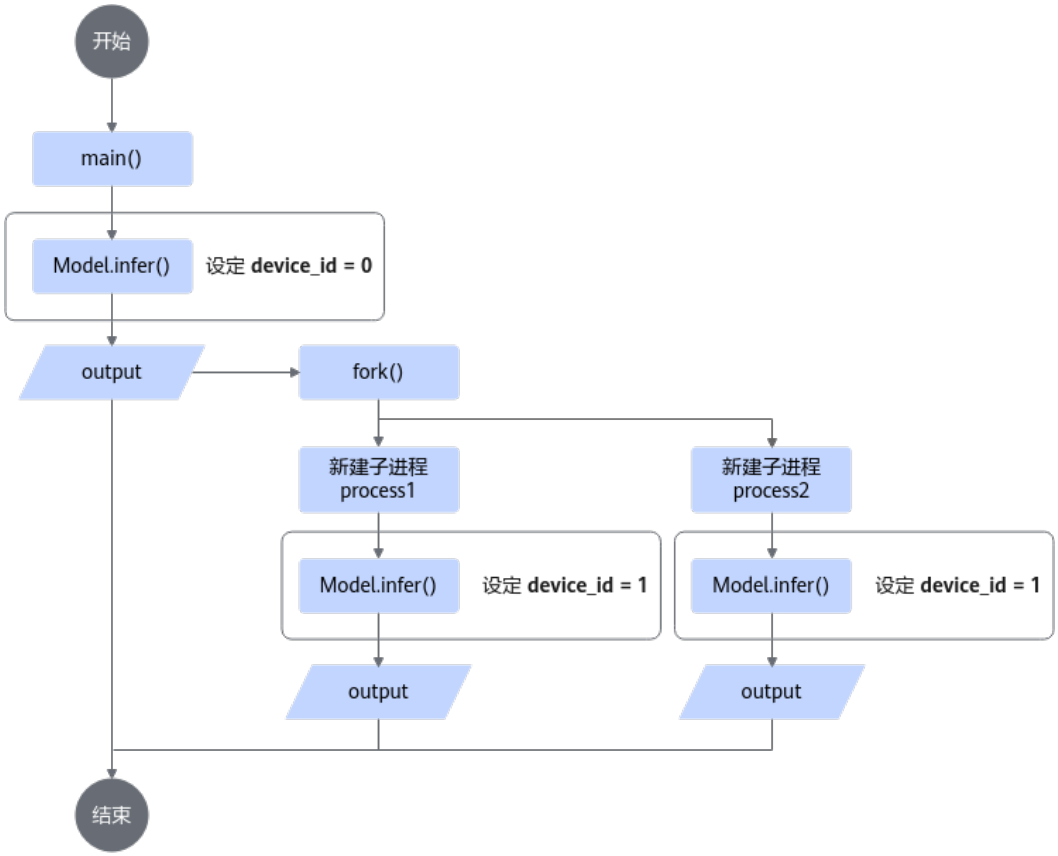


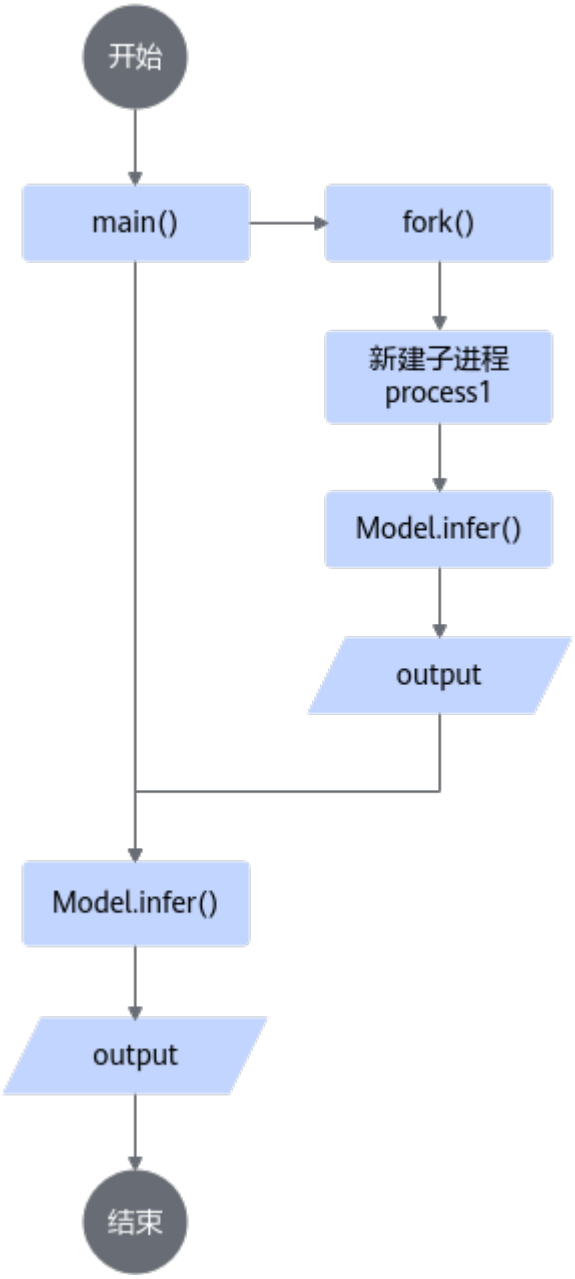
图 5-3 新建的子进程可设定同一个 device_id



子进程推理在前，父进程推理在后

若业务需要父子进程嵌套，建议先让子进程进行推理， 再调用父进程推理，但此解决方案在调用逻辑上来讲比较难以理解，因此不推荐使用、程序调用逻辑如图5-4所示。

图 5-4 先进行子进程推理，再调用父进程推理



示例代码

```
int main(){
  // 下文的func()函数代表一个完整的推理任务
  // 以下为推荐的多进程调用方式，请在fork()后于返回的父子进程中调用推理
  pid_t pid;
  int i;
  for (i = 0; i < 5; ++i)
  {
     pid = fork(); //创建子进程
     if (pid == -1)
     {
        perror("fork失败!");
        exit(1);
     }
     if (pid==0)
```

```
    {
        break;
    }
  }
  //返回大于0的进程就是父进程
  if(pid>0)  //父进程
  {
    printf("父进程: pid= %d , ppid=%d,子进程: %d \n", getpid(),getppid(),pid);
    func(); //在fork()后进行调用推理
    sleep(1); //这里延迟父进程程序，等子进程先执行完。
  }
  else if(pid == 0)  //子进程
  {
    func(); //在fork()后进行调用推理
    printf("i的变量是: %d 子进程: pid= %d , ppid=%d \n", i,getpid(),getppid());
  }
  return 0;
}
```

5.3 头文件列表说明

表 5-1 头文件列表

头文件名称	所在目录	用途
AscendStream.h	MxBase/Asynchron	定义AscendStream类，用于异步流程控制。
Constants.h	MxBase/Common	声明常量EPSILON，EPSILON_DBL。
HiddenAttr.h	MxBase/Common	声明__attribute__相关的宏，用于标记接口属性，当前主要用于标记废弃接口、使得在编译阶段出现接口已废弃的告警提示。
Version.h	MxBase/Common	声明返回mxVision版本信息的接口，mxVision版本信息主要包括当前mxVision版本号、主版本号、从版本号、小版本号。
ConfigUtil.h	MxBase/ConfigUtil	定义ConfigData类和ConfigUtil类用于后处理模块和日志模块。具体来说，ConfigUtil类用于将配置文件读取到ConfigData对象当中，ConfigData对象定义了如添加JASON文件键值对、加载Label文件到std::vector对象、根据索引返回标签名等接口，供后处理模块和日志模块调用。
DataType.h	MxBase/CV/Core	定义目标检测相关数据结构，具体包含：DetectBox结构体、RoiBox结构体、IOUMethod枚举类、TrackFlag枚举类。
Huangarian.h	MxBase/CV/Core/MultipleObjectTracking	定义匈牙利算法相关数据结构和接口，用于二分图匹配。

头文件名称	所在目录	用途
KalmanTracker.h	MxBase/CV/Core/MultipleObjectTracking	定义卡尔曼滤波相关数据结构和接口，用于跟踪检测目标。
Nms.h	MxBase/CV/ObjectDetection/Nms	定义非极大值抑制算法相关接口，用于去除冗余目标框。
SimilarityTransform.h	MxBase/CV/WarpAffine	定义SimilarityTransform类，用于计算相似变换。
WrapAffine.h	MxBase/CV/WarpAffine	定义WarpAffine类，用于计算仿射变换。
DeviceManager.h	MxBase/DeviceManager	定义DeviceManager类，其中包括设备初始化、返回设备数量、返回当前设备、设置运行设备等相关接口。
DvppWrapper.h	MxBase/DvppWrapper	定义了DvppWrapper类，该类会根据芯片的不同调用不同的接口进行实例化，DvppWrapper类封装了图像变换相关的操作。
DvppWrapperDataType.h	MxBase/DvppWrapper	主要定义了数字图片相关的数据结构以及用于图片参数校验的常量。
DvppWrapperBase.h	MxBase/DvppWrapper	主要定义了DvppWrapperBase类，该类作为中间层，被其他类继承，同时作为DvppWrapper类的私有成员被使用。
Color.h	MxBase/E2eInfer/Color	定义了Color结构体。
Dim.h	MxBase/E2eInfer/Dim	定义了Dim结构体。
GlobalInit.h	MxBase/E2eInfer/GlobalInit	声明了初始化与去初始化接口。
Image.h	MxBase/E2eInfer/Image	定义了Image类，该类作为图片的数据结构，包含获取图片属性、将图片转为Tensor对象等接口。
ImageProcessor.h	MxBase/E2eInfer/ImageProcessor	定义了ImageProcessor类，提供编码、解码、切片、缩放、抠图、贴图等常用接口，ImageProcessor类底层调用了DvppWrapper类中的接口。
Model.h	MxBase/E2eInfer/Model	定义了Model类，提供高性能推理，返回输入输出Tensor格式等接口。
Point.h	MxBase/E2eInfer/Point	定义了Point结构体用于表示点。

头文件名称	所在目录	用途
Rect.h	MxBase/E2eInfer/Rect	定义了Rect结构体用于表示矩形。
Size.h	MxBase/E2eInfer/Size	定义了Size结构体用于表示尺寸。
CallBack.h	MxBase/E2eInfer/Tensor	定义了Tensor类操作回调所需的结构体以及相关接口。
Tensor.h	MxBase/E2eInfer/Tensor	定义了Tensor类，提供返回Tensor数据类型、返回字节大小、返回Tensor当前设备Id等接口。
TensorDvpp.h	MxBase/E2eInfer/Tensor	定义了Tensor图像处理方法，提供抠图、缩放、色域转换等图像处理功能。
TensorOperations.h	MxBase/E2eInfer/Tensor	定义了Tensor运算方法。
TensorFeatures.h	MxBase/E2eInfer/Tensor	定义了Tensor特征提取类算法。
TensorFusion.h	MxBase/E2eInfer/TensorOperation	定义了Tensor融合方法，提供背景替换、动效透明度贴图和字幕透明度贴图等功能。
TensorWarping.h	MxBase/E2eInfer/TensorOperation	定义了Tensor图像变形处理方法，提供旋转、仿射变换等功能。
PerElementOperations.h	MxBase/E2eInfer/TensorOperation/MatricesOperation	定义了Tensor按元素处理方法，提供计算Tensor加、减、乘、除等功能。
MatrixReductions.h	MxBase/E2eInfer/TensorOperation/MatricesOperation	定义了Tensor归约方法，提供Tensor归约、求和等功能。
CoreOperationsOnTensors.h	MxBase/E2eInfer/TensorOperation/MatricesOperation	定义了Tensor核心处理方法，提供Tensor裁剪、扩展、水平堆叠和垂直堆叠等功能。
VideoDecoder.h	MxBase/E2eInfer/VideoDecoder	定义了VideoDecoder类，提供回调式、非阻塞视频解码接口，VideoDecoder类底层调用了DvppWrapper类中的接口。
VideoEncoder.h	MxBase/E2eInfer/VideoEncoder	定义了VideoEncoder类，提供回调式、非阻塞视频编码接口，VideoEncoder类底层调用了DvppWrapper类中的接口。
DataType.h	MxBase/E2eInfer	定义图像处理相关枚举类。

头文件名称	所在目录	用途
ErrorCode.h	MxBase/ErrorCode	定义错误码相关的字符串数组。
ErrorCodes.h	MxBase/ErrorCode	定义错误码相关的枚举类。
ErrorCodeThirdParty.h	MxBase/ErrorCode	定义第三方错误码相关的枚举类。
Log.h	MxBase/Log	定义Log类，主要提供了Debug、Info、Warn、Error等接口用于记录不同级别的日志信息。
FastMath.h	MxBase/Maths	定义了FastMath类，主要提供了Sigmoid、Softmax函数计算的接口。
MathFunction.h	MxBase/Maths	定义了LineRegressionFit类，主要提供了线性回归函数计算的接口。
NpySort.h	MxBase/Maths	定义了NpySort，主要提供快排算法接口。
MemoryHelper.h	MxBase/MemoryHelper	定义了MemoryData类和MemoryHelper类，主要提供内存申请、是否、拷贝等接口。
ModelDataType.h	MxBase/ModelInfer	定义了模型相关的数据结构。
ModelInferenceProcessor.h	MxBase/ModelInfer	定义了ModelInferenceProcessor类，主要提供模型推理相关接口，其功能与MxBase/E2eInfer/Model/Model.h相似。
CrnnPostProcessor.h	MxBase/ModelPostProcessors/AttributePostProcessor	定义了CrnnPostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
Resnet50PostProcessor.h	MxBase/ModelPostProcessors/ClassPostProcessor	定义了Resnet50PostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
ModelPostProcessorBase.h	MxBase/ModelPostProcessors/ModelPostProcessorBase	定义了ModelPostProcessorBase类，该类为MxBase/ModelPostProcessors目录下的基类。
ObjectPostDataType.h	MxBase/ModelPostProcessors/ModelPostProcessorBase/	定义了目标后处理相关的数据结构。

头文件名称	所在目录	用途
ObjectPostProcessorBase.h	MxBase/ModelPostProcessors/ModelPostProcessorBase	定义了ObjectPostProcessorBase类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
FasterRcnnPostProcessor.h	MxBase/ModelPostProcessors/ObjectPostProcessors	定义了FasterRcnnPostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
SsdmobilenetfpnPostProcessor.h	MxBase/ModelPostProcessors/ObjectPostProcessors	定义了SsdmobilenetfpnPostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
SsdvggPostProcessor.h	MxBase/ModelPostProcessors/ObjectPostProcessors	定义了SsdvggPostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
Yolov3PostProcessor.h	MxBase/ModelPostProcessors/ObjectPostProcessors	定义了Yolov3PostProcessor类，该类继承ModelPostProcessorBase类，该类已标识为弃用。
CtpnPostProcessor.h	MxBase/ModelPostProcessors/OcrPostProcessors	定义了CtpnPostProcessor类，该类ModelPostProcessorBase类，该类已标识为弃用。
ClassPostProcessBase.h	MxBase/PostProcessBases	定义了ClassPostProcessBase类，该类继承PostProcessBase类，主要提供Process接口。
ImagePostProcessBase.h	MxBase/PostProcessBases	定义了ImagePostProcessBase类，该类继承PostProcessBase类，主要提供设置裁剪感兴趣区域的接口。
KeypointPostProcessBase.h	MxBase/PostProcessBases	定义了KeypointPostProcessBase类，该类继承PostProcessBase类，主要提供关键点检测任务相关的后处理接口。
ObjectPostProcessBase.h	MxBase/PostProcessBases	定义了ObjectPostProcessBase类，该类继承PostProcessBase类，主要提供目标检测任务相关的后处理接口。

头文件名称	所在目录	用途
PostProcessBase.h	MxBase/PostProcessBases	定义PostProcessBase类，该类为其余后处理类的基类。
PostProcessDataType.h	MxBase/PostProcessBases	定义了目标检测、图片缩放、图片裁剪等任务相关的数据结构。
SemanticSegPostProcessBase.h	MxBase/PostProcessBases	定义了SemanticSegPostProcessBase类，该类继承PostProcessBase类，主要提供语义分割任务相关的后处理接口。
TextGenerationPostProcessBase.h	MxBase/PostProcessBases	定义了TextGenerationPostProcessBase类，该类继承PostProcessBase类，主要提供文本生成任务相关的后处理接口。
TextObjectPostProcessBase.h	MxBase/PostProcessBases	定义了TextObjectPostProcessBase类，该类继承PostProcessBase类，主要提供文本对象任务相关的后处理接口。
OperatorDesc.h	MxBase/SingleOp	定义了OperatorDesc类，主要用于加载单算子的描述。
OpRunner.h	MxBase/SingleOp	定义了OpRunner类，主要用于执行单算子。
TensorBase.h	MxBase/Tensor/TensorBase	定义TensorBase类。
TensorDataType.h	MxBase/Tensor/TensorBase	定义TensorDataType枚举类，该类对象为TensorBase类的成员变量。
TensorBuffer.h	MxBase/Tensor/TensorBuffer	定义TensorBuffer类。
TensorContext.h	MxBase/Tensor/TensorContext	定义TensorContext类。
TensorShape.h	MxBase/Tensor/TensorShape	定义TensorShape类。

5.4 数据结构及枚举类型

5.4.1 通用数据结构

5.4.1.1 ModelLoadOptV2

功能

支持多种推理模型输入方式，用户可通过该数据结构，选择其中一种方式输入推理模型。

须知

请根据实际情况选择对应配置，如配置与实际输入存在差异，会在**5.8.1.2 Model**处抛出异常，如未对该异常进行**catch**操作，则程序会发生**core dumped**。

结构定义

```
struct ModelLoadOptV2 {
    enum ModelLoadType {
        LOAD_MODEL_FROM_FILE = 1,
        LOAD_MODEL_FROM_FILE_WITH_MEM,
        LOAD_MODEL_FROM_MEM,
        LOAD_MODEL_FROM_MEM_WITH_MEM
    };
    enum ModelType {
        MODEL_TYPE_OM = 0,
        MODEL_TYPE_MINDIR
    };
    ModelType modelType = MODEL_TYPE_OM;
    ModelLoadType loadType = LOAD_MODEL_FROM_FILE;
    std::string modelPath = "";
    void* modelPtr = nullptr;
    void* modelWorkPtr = nullptr;
    void* modelWeightPtr = nullptr;
    size_t modelSize = 0;
    size_t workSize = 0;
    size_t weightSize = 0;
};
```

参数说明

参数名	输入/输出	说明
modelType	输入	推理模型类型，支持情况参见如下。（默认为“MODEL_TYPE_OM”。） ● MODEL_TYPE_OM ● MODEL_TYPE_MINDIR（对于MindIR模型，仅支持静态Shape和动态Batch。）

参数名	输入/输出	说明
loadType	输入	推理模型输入类型，可通过以下参数选定输入方式。（默认为“LOAD_MODEL_FROM_FILE”。） ● LOAD_MODEL_FROM_FILE：从文件加载离线模型数据，由系统内部管理内存。 ● LOAD_MODEL_FROM_FILE_WITH_MEM：从文件加载离线模型数据，由用户自行管理模型运行的内存（包括工作内存和权值内存，工作内存用于模型执行过程中的临时数据，权值内存用于存放权值数据）。 ● LOAD_MODEL_FROM_MEM：从内存加载离线模型数据，由系统内部管理内存。 ● LOAD_MODEL_FROM_MEM_WITH_MEM：从内存加载离线模型数据，由用户自行管理模型运行的内存（包括工作内存和权值内存）。 ● 当“modelType”为“MODEL_TYPE_MINDIR”时，仅支持“LOAD_MODEL_FROM_FILE”和“LOAD_MODEL_FROM_MEM”两种输入类型。
modelPath	输入	推理模型文件路径，仅在以下模式生效。（默认为""，最大只支持至4GB大小的模型且建议模型属主为当前用户，模型文件的权限小于或等于“640”。） ● LOAD_MODEL_FROM_FILE ● LOAD_MODEL_FROM_FILE_WITH_MEM
modelPtr	输入	推理模型所在内存地址指针，仅在以下模式生效。（默认为nullptr，用户需根据实际情况输入内存地址。） ● LOAD_MODEL_FROM_MEM ● LOAD_MODEL_FROM_MEM_WITH_MEM
modelWorkPtr	输入	推理模型所在工作内存地址指针，仅在以下模式生效。（默认为nullptr，表示由系统管理内存。） ● LOAD_MODEL_FROM_FILE_WITH_MEM ● LOAD_MODEL_FROM_MEM_WITH_MEM
modelWeightPtr	输入	推理模型权值内存地址指针，仅在以下模式生效。（默认为nullptr，表示由系统管理内存。） ● LOAD_MODEL_FROM_FILE_WITH_MEM ● LOAD_MODEL_FROM_MEM_WITH_MEM
modelSize	输入	推理模型数据长度，单位Byte，仅在以下模式生效。（默认为0，最大只支持至4GB大小的模型） ● LOAD_MODEL_FROM_MEM ● LOAD_MODEL_FROM_MEM_WITH_MEM

参数名	输入/输出	说明
workSize	输入	推理模型所在工作内存大小，单位Byte。（默认为0，当“modelWorkPtr”为nullptr时无效。）
weightSize	输入	推理模型权值内存大小，单位Byte。（默认为0，当“modelWeightPtr”为nullptr时无效。）

5.4.1.2 VideoDecodeConfig

功能

用于保存视频解码的参数。

结构定义

```
struct VideoDecodeConfig {
    uint32_t width = 1920;
    uint32_t height = 1080;
    StreamFormat inputVideoFormat = StreamFormat::H264_MAIN_LEVEL;
    ImageFormat outputImageFormat = ImageFormat::YUV_SP_420;
    VideoDecodeCallBack callbackFunc = nullptr;
    uint32_t skipInterval = 0;
    uint32_t cscMatrix = 0;
};
```

参数说明

参数名	说明
width	视频解码缓冲区的宽，取值范围：[128, 4096]，默认值：1920，必须为偶数并自动对齐到16的倍数。 请用户根据实际的视频帧数据进行适当设置，需大于或等于实际的视频帧数据宽，否则会无解码输出，设置过大将会产生多余的内存资源。
height	视频解码缓冲区的高，取值范围：[128, 4096]，默认值：1080，必须为偶数。 请用户根据实际的视频帧数据进行设置，需大于或等于实际的视频帧数据高，否则会无解码输出，设置过大将会产生多余的内存资源。
inputVideoFormat	输入的视频格式。 ● H.264格式视频支持h264_baseline_level、h264_main_level、h264_high_level三种格式。 ● H.265格式视频只支持h265_main_level格式，默认值：StreamFormat::H264_MAIN_LEVEL。

参数名	说明
outputImageFormat	解码后输出的图像格式，默认值：ImageFormat::YUV_SP_420。 ● Atlas 200/500 A2 推理产品支持 YUV_SP_420、YVU_SP_420 格式。
callbackFunc	解码完成后的回调函数，不能为空指针。
skipInterval	视频解码的跳帧参数，取值范围：[0, 250]，默认值：0。

参数名	说明
cscMatrix	色域转换选项，默认值为“0”，（即HI_CSC_MATRIX_BT601_WIDE）。 ● HI_CSC_MATRIX_BT601_WIDE = 0，基于BT601 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.000 0.000 1.402 | | Y - 0 | # | G | = | 1.000 -0.344 -0.714 | * | U - 128 | # | B | | 1.000 1.772 0.000 | | V - 128 | # RGB转YUV： # | Y | |-0.5 | | 0.299 0.587 0.114 | | R | # | U | = |127.5 | + | -0.168 -0.331 0.500 | * | G | # | V | |127.5 | | 0.500 -0.419 -0.081 | | B | ● HI_CSC_MATRIX_BT601_NARROW =1，基于BT601 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.59602 | | Y - 16 | # | G | = | 1.16438 -0.39176 -0.81297 | * | U - 128 | # | B | | 1.16438 2.01723 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.25679 0.51564 0.10014 | | R | # | U | = |128| + | -0.14491 -0.29099 0.43922 | * | G | # | V | |128| | 0.42941 -0.36779 -0.07143 | | B | ● HI_CSC_MATRIX_BT709_WIDE =2，基于BT709 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.57480 | | Y - 0 | # | G | = | 1.00000 -0.18732 -0.46812 | * | U - 128 | # | B | | 1.00000 1.85560 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.21260 0.71520 0.07220 | | R | # | U | = |128| + | -0.11457 -0.38543 0.50000 | * | G | # | V | |128| | 0.50000 -0.45415 -0.04585 | | B | ● HI_CSC_MATRIX_BT709_NARROW =3，基于BT709 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.79274 | | Y - 16 | # | G | = | 1.16438 -0.21325 -0.53291 | * | U - 128 | # | B | | 1.16438 2.11240 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.18259 0.62825 0.06342 | | R | # | U | = |128| + | -0.09840 -0.33857 0.43922 | * | G | # | V | |128| | 0.42941 -0.39894 -0.04027 | | B | ● HI_CSC_MATRIX_BT2020_WIDE =4，基于BT2020 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.47460 | | Y - 0 | # | G | = | 1.00000 -0.16455 -0.57135 | * | U - 128 | # | B | | 1.00000 1.88140 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.26270 0.67800 0.05930 | | R | # | U | = |128| + | -0.13963 -0.36037 0.50000 | * | G | # | V | |128| | 0.50000 -0.45979 -0.04021 | | B | ● HI_CSC_MATRIX_BT2020_NARROW =5，基于BT2020 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.67868 | | Y - 16 | # | G | = | 1.16438 -0.18733 -0.65042 | * | U - 128 | # | B | | 1.16438 2.14177 0.00000 | | V - 128 | # RGB转YUV：

参数名	说明
	# | Y | |16 | | 0.22564 0.59558 0.05209 | | R | # | U | = |128| + | -0.11992 -0.31656 0.43922 | * | G | # | V | |128| | 0.42941 -0.40389 -0.03533 | | B |

5.4.1.3 VideoDecodeCallBack

功能

用于定义视频解码输入的回调函数，请勿在回调函数内实现过于复杂的操作，而是单纯用用户自定义“userData”来接收视频解码回调结果，否则回调线程会发生卡住的现象，导致视频解码速度变慢。

结构定义

```
typedef APP_ERROR (*VideoDecodeCallBack)(Image& decodedImage, uint32_t channelId,
                uint32_t frameId, void* userData);
```

参数说明

参数名	说明
decodedImage	解码后输出的图像类。
channelId	视频流索引，由VideoDecoder类的构造函数来设置。
frameId	视频帧索引，由VideoDecoder类的Decode函数来设置。
userData	用户自定义回调输入的数据类型（主要用来获取解码数据）。

5.4.1.4 VideoEncodeConfig

功能

用于保存视频编码的相关参数。

结构定义

```
struct VideoEncodeConfig {
    uint32_t maxPicWidth = 4096;
    uint32_t maxPicHeight = 4096;
    uint32_t width = 1920;
    uint32_t height = 1080;
    // stream format renference acldvppStreamFormat
    StreamFormat outputVideoFormat = StreamFormat::H264_MAIN_LEVEL;
    // output format renference acldvppPixelFormat
    ImageFormat inputImageFormat = ImageFormat::YUV_SP_420;
    VideoEncodeCallBack callbackFunc = nullptr;
    uint32_t keyFrameInterval = 30;
    uint32_t srcRate = 30;
    uint32_t rcMode = 0;
    uint32_t maxBitRate = 300;
```

```
    uint32_t ipProp = 70;
};
```

参数说明

参数名	说明
width	编码后视频帧数据的宽，默认值：1920，必须与待编码图片的宽（originalSize）一致。 ● Atlas 200/500 A2 推理产品，取值范围：[128, 1920]，输入的视频宽需要对齐16。
height	编码后视频帧数据的高，默认值：1080，必须与待编码图片的高（originalSize）一致。 ● Atlas 200/500 A2 推理产品，取值范围：[128, 1920]，输入的视频高需要对齐2。
outputVideoFormat	编码后输出的视频格式， 默认值：StreamFormat::H264_MAIN_LEVEL。
inputImageFormat	输入的图像格式，默认值：ImageFormat::YUV_SP_420。 ● Atlas 200/500 A2 推理产品支持 YUV_SP_420、YVU_SP_420 格式。
callbackFunc	编码完成后的回调函数，不能为空指针。
keyFrameInterval	视频I帧间隔大小，默认为30，取值范围[1, 2000]。
srcRate	输入码流帧率，单位fps，默认为30。 ● Atlas 200/500 A2 推理产品，取值范围：[1, 120]。
rcMode	指定码率控制模式，默认值为 0，取值如下： ● 0：默认模式。Atlas 200/500 A2 推理产品默认为CBR模式。。 ● 1：变码率VBR模式。 ● 2：定码率CBR模式。
maxBitRate	输出码率，单位kbps，默认值为300。 ● Atlas 200/500 A2 推理产品，取值范围：[10, 30000]。
ipProp	一个GOP内单个I帧bit数和单个P帧bit数的比例，默认值为70，取值范围[1, 100]。

推荐配置

以下为视频采集场景下推荐的参数配置，可以保证较好的视频编码质量，其他场景下请根据实际应用对应调整。

画质/分辨率	srcRate	keyFrameInterval	maxBitRate
4K 3840 * 2160/4096 * 2160	25或30	50或60	● H.264： 8000 ~ 12000 ● H.265： 8000 ~ 12000
2K 2560 * 1440	25或30	50或60	● H.264： 6000 ~ 10000 ● H.265： 6000 ~ 10000
1080P（蓝光） 1920 * 1080	25或30	50或60	● H.264： 2000~6000 ● H.265： 1000~4000
720P（高清） 1280 * 720	25或30	50或60	● H.264： 1000~3000 ● H.265： 800~2000
480P/D1_N（标清） 854 * 480/720 * 480	25或30	50或60	● H.264： 600~1400 ● H.265： 300~700
576P/D1 （标清） 720 * 576	25或30	50或60	● H.264： 600~1400 ● H.265： 300~700
270P（流畅） 480 * 270	25或30	50或60	-
CIF P/N 352 * 288/320 * 240	25或30	50或60	● H.264： 250 ● H.265： 250

5.4.1.5 VideoEncodeCallBack

功能

用于定义视频编码输入的回调函数。

请勿在回调函数内实现过于复杂的操作，而是单纯用用户自定义“userData”来接收视频编码回调结果，否则回调线程会发生卡住的现象，导致视频编码速度变慢。

结构定义

```
typedef APP_ERROR (*VideoEncodeCallBack)(std::shared_ptr<uint8_t>& outDataPtr, uint32_t& outDataSize,
                                         uint32_t& channelId, uint32_t& frameId, void* userData);
```

参数说明

参数名	说明
outDataPtr	编码后输出视频帧数据的内存地址。
outDataSize	编码后输出视频帧数据的内存大小。
channelId	视频流索引，由VideoEncoder类的构造函数来设置。

参数名	说明
frameId	视频帧索引，由VideoEncoder类的Encode函数来设置。
userData	用户自定义回调输入的数据类型（主要用来获取编码数据）。

5.4.1.6 Rect

功能

矩形框结构体（抠图贴图），用来保存一个矩形框的左上角坐标和右下角坐标（图像的左上角坐标和右下角坐标）。

结构定义

```
struct Rect {
    Rect()
        : x0(0), y0(0), x1(0), y1(0) {};
    Rect(const uint32_t leftTopX, const uint32_t leftTopY,
         const uint32_t rightBottomX, const uint32_t rightBottomY)
        : x0(leftTopX), y0(leftTopY), x1(rightBottomX), y1(rightBottomY) {};
    Rect(const Point leftTop, const Point rightBottom)
        : x0(leftTop.x), y0(leftTop.y), x1(rightBottom.x), y1(rightBottom.y) {};

    uint32_t x0 = 0;
    uint32_t y0 = 0;
    uint32_t x1 = 0;
    uint32_t y1 = 0;
};
```

参数说明

参数名	说明
x0，leftTopX	矩形框左上角坐标的横坐标（以图像左上角为原点）。
y0，leftTopY	矩形框左上角坐标的纵坐标（以图像左上角为原点）。
x1，rightBottomX	矩形框右下角坐标的横坐标（以图像左上角为原点）。
y1，rightBottomY	矩形框右下角坐标的纵坐标（以图像左上角为原点）。
leftTop	矩形框左上角坐标点（Point 结构体）。
rightBottom	矩形框右下角坐标点（Point 结构体）。

5.4.1.7 Size

功能

图像大小结构体（缩放），用来保存一个图像的高和宽。

结构定义

```
struct Size {
    Size()
        : width(0), height(0) {};
    Size(const uint32_t inputWidth, const uint32_t inputHeight)
        : width(inputWidth), height(inputHeight) {};

    uint32_t width = 0;
    uint32_t height = 0;
};
```

参数说明

参数名	说明
width，inputWidth	图像宽。
height，inputHeight	图像高。

5.4.1.8 Point

功能

坐标点，用于保存图像像素点位置的结构体。

结构定义

```
struct Point {
    Point()
        : x(0), y(0) {};
    Point(const uint32_t inputX, const uint32_t inputY)
        : x(inputX), y(inputY) {};

    uint32_t x = 0;
    uint32_t y = 0;
};
```

参数说明

参数名	说明
x，inputX	横坐标（以图像左上角为原点）。
y，inputY	纵坐标（以图像左上角为原点）。

5.4.1.9 Color

功能

色彩值，用于图像处理补边功能，描述三通道色彩的结构体。

结构定义

```
struct Color {
    Color()
```

```
        : channel_zero(0), channel_one(0), channel_two(0) {};
    Color(const uint32_t inputRed, const uint32_t inputGreen, const uint32_t inputBlue)
        : channel_zero(inputRed), channel_one(inputGreen), channel_two(inputBlue) {};

    uint32_t channel_zero;
    uint32_t channel_one;
    uint32_t channel_two;
};
```

参数说明

参数名	说明
channel_zero，inputRed	0号通道取值，范围[0, 255]。 例如：对于RGB_888格式图像，该通道为R；对于YUV格式图像，该通道为Y。
channel_one，inputGreen	1号通道取值，范围[0, 255]。 例如：对于RGB_888格式图像，该通道为G；对于YUV格式图像，该通道为U。
channel_two，inputBlue	2号通道取值，范围[0, 255]。 例如：对于RGB_888格式图像，该通道为B；对于YUV格式图像，该通道为V。

5.4.1.10 Dim

功能

补边值，用于图像处理“ImageProcessor”的补边功能，描述左、右、上、下四个方向补边的像素个数。

结构定义

```
struct Dim {
    Dim()
        : left(0), right(0), top(0), bottom(0) {};
    Dim(const uint32_t inputDim)
        : left(inputDim), right(inputDim), top(inputDim), bottom(inputDim) {};
    Dim(const uint32_t inputLeft, const uint32_t inputRight, const uint32_t inputTop, const uint32_t
inputBottom)
        : left(inputLeft), right(inputRight), top(inputTop), bottom(inputBottom) {};
    uint32_t left;
    uint32_t right;
    uint32_t top;
    uint32_t bottom;
};
```

参数说明

参数名	说明
left，inputLeft	左侧补边像素数量。
right，inputRight	右侧补边像素数量。

参数名	说明
top，inputTop	上方补边像素数量。
bottom，inputBottom	下方补边像素数量。
inputDim	左、右、上、下补边像素数量。 仅在使用**Dim(const uint32_t inputDim)**构造函数时，将左、右、上、下补边像素数量设为与inputDim相同的值。

5.4.1.11 TensorDType

功能

用于描述Tensor类的数据类型。

结构定义

```
enum class TensorDType {
    UNDEFINED = -1,
    FLOAT32 = 0,
    FLOAT16 = 1,
    INT8 = 2,
    INT32 = 3,
    UINT8 = 4,
    INT16 = 6,
    UINT16 = 7,
    UINT32 = 8,
    INT64 = 9,
    UINT64 = 10,
    DOUBLE64 = 11,
    BOOL = 12
};
```

参数说明

参数名	说明
UNDEFINED	未定义类型。
FLOAT32	32位浮点型。
FLOAT16	16位浮点型。
INT8	8为整型。
INT32	32位整型。
UINT8	8位无符号整型。
INT16	16位整型。
UINT16	16位无符号整型。
UINT32	32位无符号整型。

参数名	说明
INT64	64位整型。
DOUBLE64	64位双精度浮点型。
BOOL	布尔型。

5.4.1.12 VisionDataFormat

功能

图像数据格式排布形式。

结构定义

```
enum class VisionDataFormat {
    NCHW = 0,
    NHWC = 1
};
```

参数说明

参数名	说明
NCHW	图像数据按NCHW格式排布。
NHWC	图像数据按NHWC格式排布。

5.4.1.13 StreamFormat

功能

视频流数据格式，用于视频解码和视频编码。

结构定义

```
enum class StreamFormat {
    H265_MAIN_LEVEL = 0,
    H264_BASELINE_LEVEL = 1,
    H264_MAIN_LEVEL = 2,
    H264_HIGH_LEVEL = 3,
};
```

参数说明

参数名	说明
H265_MAIN_LEVEL	**H.265** 格式视频流，**主流画质**。
H264_BASELINE_LEVEL	**H.264** 格式视频流，**基本画质**。

参数名	说明
H264_MAIN_LEVEL	**H.264** 格式视频流，**主流画质**。
H264_HIGH_LEVEL	**H.264** 格式视频流，**高级画质**。

5.4.1.14 Interpolation

功能

缩放方式，用于**ImageProcessor**类的缩放接口。

结构定义

```
enum class Interpolation {
    HUAWEI_HIGH_ORDER_FILTER = 0,
    BILINEAR_SIMILAR_OPENCV = 1,
    NEAREST_NEIGHBOR_OPENCV = 2,
    BILINEAR_SIMILAR_TENSORFLOW = 3,
    NEAREST_NEIGHBOR_TENSORFLOW = 4,
};
```

参数说明

参数名	说明
HUAWEI_HIGH_ORDER_FILTER	华为自研的高阶滤波算法。
BILINEAR_SIMILAR_OPENCV	业界通用的 Bilinear 算法（与 OpenCV 算法的计算过程类似）。
NEAREST_NEIGHBOR_OPENCV	业界通用的 Nearest neighbor 算法（与 OpenCV 算法的计算过程类似）。
BILINEAR_SIMILAR_TENSORFLOW	业界通用的 Bilinear 算法（与 TensorFlow 算法的计算过程类似）。
NEAREST_NEIGHBOR_TENSORFLOW	业界通用的 Nearest neighbor 算法（与 TensorFlow 算法的计算过程类似，）。

5.4.1.15 BorderType

功能

补边方式，用于图像处理“ImageProcessor”类的补边功能。

结构定义

```
enum class BorderType {
    BORDER_CONSTANT = 0,
    BORDER_REPLICATE,
    BORDER_REFLECT,
    BORDER_REFLECT_101,
};
```

参数说明

参数名	说明
BORDER_CONSTANT	添加有颜色的常数值边界。
BORDER_REPLICATE	重复最后一个元素。 例如：aaaaaa|a*****h|hhhhhhh（其中*表示任意图像元素）。
BORDER_REFLECT	边界元素的镜像，镜像包括边界元素。 例如：ba|abc*******fgh|hg（其中*表示任意图像元素）。
BORDER_REFLECT_101	边界元素的镜像，镜像不包括边界元素。 例如：cb|abc****fgh|gf（其中*表示任意图像元素）。

5.4.1.16 ImageFormat

功能

图像格式，用于描述解码后Image类的数据格式。

结构定义

```
enum class ImageFormat {
    YUV_400 = 0,
    YUV_SP_420 = 1,
    YVU_SP_420 = 2,
    YUV_SP_422 = 3,
    YVU_SP_422 = 4,
    YUV_SP_444 = 5,
    YVU_SP_444 = 6,
    YUYV_PACKED_422 = 7,
    UYVY_PACKED_422 = 8,
    YVYU_PACKED_422 = 9,
    VYUY_PACKED_422 = 10,
    YUV_PACKED_444 = 11,
    RGB_888 = 12,
    BGR_888 = 13,
    ARGB_8888 = 14,
    ABGR_8888 = 15,
    RGBA_8888 = 16,
    BGRA_8888 = 17,
};
```

参数说明

参数名	说明
YUV_400	YUV_400 的图像格式。
YUV_SP_420	YUV_SP_420 的图像格式。

参数名	说明
YVU_SP_420	YVU_SP_420 的图像格式。
YUV_SP_422	YUV_SP_422 的图像格式。
YVU_SP_422	YVU_SP_422 的图像格式。
YUV_SP_444	YUV_SP_444 的图像格式。
YVU_SP_444	YVU_SP_444 的图像格式。
YUYV_PACKED_422	YUYV_PACKED_422 的图像格式。
UYVY_PACKED_422	UYVY_PACKED_422 的图像格式。
YVYU_PACKED_422	YVYU_PACKED_422 的图像格式。
VYUY_PACKED_422	VYUY_PACKED_422 的图像格式。
YUV_PACKED_444	YUV_PACKED_444 的图像格式。
RGB_888	RGB_888 的图像格式。
BGR_888	BGR_888 的图像格式。
ARGB_8888	ARGB_8888 的图像格式。
ABGR_8888	ABGR_8888 的图像格式。
RGBA_8888	RGBA_8888 的图像格式。
BGRA_8888	BGRA_8888 的图像格式。

5.4.1.17 RotateAngle

功能

用于描述旋转角度的枚举类型。

结构定义

```
enum class RotateAngle {
    ROTATE_90 = 90,
    ROTATE_180 = 180,
    ROTATE_270 = 270
};
```

参数说明

参数名	说明
ROTATE_90	用于描述旋转90度。
ROTATE_180	用于描述旋转180度。
ROTATE_270	用于描述旋转270度。

5.4.1.18 CropRoiConfig

功能

定义抠图的范围。

结构定义

```
struct CropRoiConfig {
    uint32_t x0;
    uint32_t x1;
    uint32_t y1;
    uint32_t y0;
};
```

参数说明

参数名	输入/输出	说明
x0	输入	左上角横坐标。
x1	输入	右下角横坐标。
y1	输入	右下角纵坐标。
y0	输入	左上角纵坐标。

5.4.1.19 ResizeConfig

功能

缩放的配置定义。

结构定义

```
struct ResizeConfig {
     uint32_t height = 0;
     uint32_t width = 0;
     float scale_x = 0.f;
     float scale_y = 0.f;
     uint32_t interpolation = 0;
 }
```

参数说明

参数名	输入/输出	说明
height	输入	缩放后的高度。
width	输入	缩放后的宽度。
scale_x	输入	横向缩放比例。

参数名	输入/输出	说明
scale_y	输入	纵向缩放比例。
interpolation	输入	指定缩放算子，取值范围： ●“0”：默认值，华为自研的最近邻插值算法。 ●“1”：业界通用的Bilinear算法，当前不支持。 ●“2”：业界通用的Nearest neighbor算法，当前不支持。

5.4.1.20 MakeBorderConfig

功能

用于在图像处理补边功能，定义具体的补边配置，可设置图像补边的左、右、上、下像素个数、补边颜色常数和补边类型。

结构定义

```
struct MakeBorderConfig {
    enum BorderType {
        BORDER_CONSTANT = 0,
        BORDER_REPLICATE,
        BORDER_REFLECT,
        BORDER_REFLECT_101
    };
    uint32_t left;
    uint32_t right;
    uint32_t top;
    uint32_t bottom;
    uint32_t channel_zero;
    uint32_t channel_one;
    uint32_t channel_two;
    BorderType borderType;
};
```

参数说明

参数名	输入/输出	说明
left	输入	左侧补边像素数量。
right	输入	右侧补边像素数量。
top	输入	上方补边像素数量。
bottom	输入	下方补边像素数量。
channel_zero	输入	0号通道取值，范围[0, 255]。
channel_one	输入	1号通道取值，范围[0, 255]。

参数名	输入/输出	说明
channel_two	输入	2号通道取值，范围[0, 255]。
borderType	输入	补边类型。 ● **BORDER_CONSTANT**：添加有颜色的常数值边界。 ● **BORDER_REPLICATE**：重复最后一个元素。 举例：aaaaaa|a*****h|hhhhhhh（其中*表示任意图像元素）。 ● **BORDER_REFLECT**：边界元素的镜像，镜像包括边界元素。 举例：ba|abc*******fgh|hg（其中*表示任意图像元素）。 ● **BORDER_REFLECT_101**：边界元素的镜像，镜像不包括边界元素。 举例：cb|abc****fgh|gf（其中*表示任意图像元素）。

说明

“channel_zero”、“channel_one”、“channel_two”，依次对应各图像通道，例如：使用**RGB**时，“channel_zero”对应**R**，“channel_one”对应**G**通道，“channel_two”对应**B**通道。

5.4.1.21 BaseTensor

功能

定义一个张量数据的结构。

结构定义

```
struct BaseTensor {
    void* buf;
    std::vector<int> shape;
    size_t size;
}
```

参数说明

参数名	输入/输出	说明
buf	输入	张量的数据。
shape	输入	张量的形状。
size	输入	张量数据对应内存大小，单位为Byte。 size大小应与实际内存大小一致，否则可能会导致程序出现coredump情况。

5.4.1.22 TensorDesc

功能

张量描述信息定义。

结构定义

```
struct TensorDesc {
    size_t tensorSize;
    std::string tensorName;
    std::vector<int64_t> tensorDims;
};
```

参数说明

参数名	输入/输出	说明
tensorSize	输出	张量的大小，单位为Byte。
tensorName	输出	张量名称。
tensorDims	输出	张量维度。

5.4.1.23 ModelDesc

功能

模型描述信息定义。

结构定义

```
struct ModelDesc {
    std::vector<TensorDesc> inputTensors;
    std::vector<TensorDesc> outputTensors;
    std::vector<size_t> batchSizes;
    bool dynamicBatch;
};
```

参数说明

参数名	输入/输出	说明
inputTensors	输出	输入张量描述。
outputTensors	输出	输出张量描述。
batchSizes	输出	batch大小。
dynamicBatch	输出	是否为动态Batch。

5.4.1.24 ModelDataset

功能

模型数据集定义。

结构定义

```
struct ModelDataset {
    void* mdlDataPtr;
    size_t dynamicBatchSize;
};
```

参数说明

参数名	输入/输出	说明
mdlDataPtr	输入	模型数据集数据的内存地址。
dynamicBatchSize	输入	模型推理时的批量大小。

5.4.1.25 DvppDataInfo

功能

DVPP实体数据定义。

结构定义

```
struct DvppDataInfo {
    uint32_t width = 0;
    uint32_t height = 0;
    uint32_t widthStride = 0;
    uint32_t heightStride = 0;
    MxbasePixelFormat format = MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
    uint32_t frameId = 0;
    uint32_t channelId = 0;
    uint32_t dataSize = 0;
    uint32_t outDataSize = 0;
    uint32_t dataType = 0;
    uint8_t* data = nullptr;
    uint8_t* outData = nullptr;
    uint32_t resizeWidth = 0;
    uint32_t resizeHeight = 0;
    std::string device = "host:0";
    void (*destory)(void *) = nullptr;
};
```

参数说明

参数名	输入/输出	说明
width	输入	原始图像宽。
height	输入	原始图像高。
widthStride	输入/输出	原始像片对齐后的宽。

参数名	输入/输出	说明
heightStride	输入/输出	原始像片对齐后的高。
format	输入/输出	图像格式，默认值为“MXBASE_PIXEL_FORMAT_YUV_SEMI PLANAR_420”。 DVPP图像格式定义具体请参见**5.4.1.52 MxbasePixelFormat**。
frameId	输入	图像帧编号。
channelId	输入	图像通道数编号。
dataSize	输入/输出	图像数据大小，单位为Byte。 dataSize大小需要和图像实际大小一致，否则可能会导致程序出现coredump情况。
outDataSize	输入/输出	输出图像数据大小，单位为Byte。 outDataSize大小需要和输出图像大小一致，否则可能会导致程序出现coredump情况。
dataType	输入	图像的数据类型。
data	输入/输出	图像数据。
outData	输入/输出	预申请内存地址，主要用于存放视频解码后的图像数据。
device	输入	设备号。
destory	输入/输出	回调函数，用于释放该DVPP数据。

5.4.1.26 DvppImageInfo

功能

DVPP图像信息描述。

结构定义

```
struct DvppImageInfo {
    enum PictureType {
        PIXEL_FORMAT_ANY = 0,
        PIXEL_FORMAT_JPEG = 1,
        PIXEL_FORMAT_PNG = 2
    };
    const void* data;
    uint32_t size;
    PictureType pictureType;
};
```

参数说明

参数名	输入/输出	说明
data	输入	图片数据的内存地址。
size	输入	图片大小，单位为Byte。 size大小需要和实际图片大小一致，否则可能会导致程序出现coredump情况。
pictureType	输入	图片类型，为下列三者之一： ● PIXEL_FORMAT_ANY = 0 ● PIXEL_FORMAT_JPEG = 1 ● PIXEL_FORMAT_PNG = 2

5.4.1.27 DvppImageOutput

功能

DVPP图像输出定义。

结构定义

```
struct DvppImageOutput {
    uint32_t width;
    uint32_t height;
    int32_t components;
    uint32_t widthStride;
    uint32_t heightStride;
    uint32_t outImgDatasize;
 }
```

参数说明

参数名	输入/输出	说明
width	输出	输出图像的宽度。
height	输出	输出图像的高度。
components	输出	输出图像的通道个数。
widthStride	输出	输出对齐后的图像宽。
heightStride	输出	输出对齐后的图像高。
outImgDatasize	输出	输出图像的内存大小。

5.4.1.28 MemoryData

功能

内存管理结构体。

结构定义

```
struct MemoryData {
    enum MemoryType {
        MEMORY_HOST = 0,
        MEMORY_DEVICE,
        MEMORY_DVPP,
        MEMORY_HOST_MALLOC,
        MEMORY_HOST_NEW,
    };
    MemoryData() = default;
    MemoryData(size_t size, MemoryType type = MEMORY_HOST, int32_t deviceId = 0)
        : size(size), deviceId(deviceId), type(type) {}
    MemoryData(void* ptrData, size_t size, MemoryType type = MEMORY_HOST, int32_t deviceId = 0)
        : ptrData(ptrData), size(size), deviceId(deviceId), type(type) {}
    void* ptrData;
    size_t size;
    int32_t deviceId;
    MemoryType type;
    APP_ERROR (*free)(void*) = nullptr;
};
```

参数说明

参数名	输入/输出	说明
ptrData	输出	存放数据的内存地址。
size	输入	内存大小，单位为Byte。 size值应与实际内存大小一致，否则可能导致出现coredump情况。
deviceId	输入	设备编号。
type	输入	申请的内存类型： ● “MEMORY_HOST”对应Host侧。 ● “MEMORY_DEVICE”对应Device侧。 ● “MEMORY_DVPP”对应DVPP侧。 ● “MEMORY_HOST_MALLOC”对应malloc申请内存。 ● “MEMORY_HOST_NEW”对应new申请内存。
free	输出	ptrData指针释放函数。

参数名	输入/输出	说明
MemoryData(size_t size, MemoryType type = MEMORY_HOST, size_t deviceId = 0)	-	定义结构体的形式一。
MemoryData(void* ptrData, size_t size, MemoryType type = MEMORY_HOST, size_t deviceId = 0)	-	定义结构体的形式二。

5.4.1.29 DetectBox

功能

定义获取推理结果使用的结构体。

结构定义

```
struct DetectBox {
    float prob;
    int classID;
    float x;
    float y;
    float width;
    float height;
    std::string className;
    void *maskPtr;
};
```

参数说明

参数名	输入/输出	说明
prob	输出	类别的置信度。
classID	输出	类别的编号。
x	输出	目标的横坐标。
y	输出	目标的纵坐标。
width	输出	目标的宽度，从横坐标**x**处开始计算。
height	输出	目标的高度，从纵坐标**y**处开始计算。
className	输出	类别名。
maskPtr	输出	实例分割需要使用的掩码。

5.4.1.30 VdecConfig

功能

定义视频流解码使用的结构体。

结构定义

```
struct VdecConfig {
    uint32_t width = 0;
    uint32_t height = 0;
    MxbaseStreamFormat inputVideoFormat = MXBASE_STREAM_FORMAT_H264_MAIN_LEVEL;
    MxbasePixelFormat outputImageFormat = MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
    uint32_t channelId = 0;
    uint32_t deviceId  = 0;
    pthread_t threadId  = 0;
    DecodeCallBackFunction callbackFunc = nullptr;
    uint32_t outMode = 0;
    uint32_t videoChannel = 0;
    uint32_t skipInterval = 0;
    uint32_t cscMatrix = 0;
    void* userData = nullptr;
};
```

参数说明

参数名	输入/输出	说明
width	输入	图像的宽度。
height	输入	图像的高度。
inputVideoFormat	输入	输入图像的格式。
outputImageFormat	输入	输出图像的格式。
channelId	输入	解码的通道编号。
deviceId	输入	设备编号。
threadId	输入	回调线程编号。
callbackFunc	输入	回调函数。 结构为： APP_ERROR (*DecodeCallBackFunction)(std::shared_ptr<void> **buffer**, DvppDataInfo& **dvppDataInfo**, void* **userData**) ● buffer：解码后输出的数据。 ● dvppDataInfo：解码后输出的数据信息。 ● userData：用户定义获取数据指针。

参数名	输入/输出	说明
outMode	输入	选择出帧模式。默认值为0，可取值为0或1。 ● “0”：由于解码过程中存在缓存帧，无法实时输出，因此VDEC需要在收到码流中的多帧数据后，才开始输出解码结果。 ● “1”：快速出帧模式，VDEC获取码流中的一帧数据后，就开始实时输出解码结果，只支持简单参考关系的H.264/H.265标准码流（无长期参考帧，无B帧）。
videoChannel	输入	预留参数。
skipInterval	输入	跳帧参数。

参数名	输入/输出	说明
cscMatrix	输入	色域转换选项。 ● HI_CSC_MATRIX_BT601_WIDE = 0，基于BT601 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.000 0.000 1.402 | | Y - 0 | # | G | = | 1.000 -0.344 -0.714 | * | U - 128 | # | B | | 1.000 1.772 0.000 | | V - 128 | # RGB转YUV： # | Y | |-0.5 | | 0.299 0.587 0.114 | | R | # | U | = |127.5 | + | -0.168 -0.331 0.500 | * | G | # | V | |127.5 | | 0.500 -0.419 -0.081 | | B | ● HI_CSC_MATRIX_BT601_NARROW =1，基于BT601 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.59602 | | Y - 16 | # | G | = | 1.16438 -0.39176 -0.81297 | * | U - 128 | # | B | | 1.16438 2.01723 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.25679 0.51564 0.10014 | | R | # | U | = |128| + | -0.14491 -0.29099 0.43922 | * | G | # | V | |128| | 0.42941 -0.36779 -0.07143 | | B | ● HI_CSC_MATRIX_BT709_WIDE =2，基于BT709 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.57480 | | Y - 0 | # | G | = | 1.00000 -0.18732 -0.46812 | * | U - 128 | # | B | | 1.00000 1.85560 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.21260 0.71520 0.07220 | | R | # | U | = |128| + | -0.11457 -0.38543 0.50000 | * | G | # | V | |128| | 0.50000 -0.45415 -0.04585 | | B | ● HI_CSC_MATRIX_BT709_NARROW =3，基于BT709 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.79274 | | Y - 16 | # | G | = | 1.16438 -0.21325 -0.53291 | * | U - 128 | # | B | | 1.16438 2.11240 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.18259 0.62825 0.06342 | | R | # | U | = |128| + | -0.09840 -0.33857 0.43922 | * | G | # | V | |128| | 0.42941 -0.39894 -0.04027 | | B | ● HI_CSC_MATRIX_BT2020_WIDE =4，基于BT2020 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.47460 | | Y - 0 | # | G | = | 1.00000 -0.16455 -0.57135 | * | U - 128 | # | B | | 1.00000 1.88140 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.26270 0.67800 0.05930 | | R | # | U | = |128| + | -0.13963 -0.36037 0.50000 | * | G | # | V | |128| | 0.50000 -0.45979 -0.04021 | | B | ● HI_CSC_MATRIX_BT2020_NARROW =5，基于BT2020 narrow标准的色域转换矩阵，各参数值参考如下。

参数名	输入/输出	说明
		# YUV转RGB： # | R | | 1.16438 0.00000 1.67868 | | Y - 16 | # | G | = | 1.16438 -0.18733 -0.65042 | * | U - 128 | # | B | | 1.16438 2.14177 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.22564 0.59558 0.05209 | | R | # | U | = |128| + | -0.11992 -0.31656 0.43922 | * | G | # | V | |128| | 0.42941 -0.40389 -0.03533 | | B |
userData	输入	用户自定义数据。

5.4.1.31 PostImageInfo

功能

定义后处理图片信息使用的结构体。

结构定义

```
struct PostImageInfo {
    uint32_t widthOriginal = 0;
    uint32_t heightOriginal = 0;
    uint32_t widthResize = 0;
    uint32_t heightResize = 0;
    float x0 = 0;
    float y0 = 0;
    float x1 = 0;
    float y1 = 0;
};
```

参数说明

参数名	输入/输出	说明
widthOriginal	输入	原始图片宽度。
heightOriginal	输入	原始图片高度。
widthResize	输入	缩放后图片宽度。
heightResize	输入	缩放后图片高度。
x0	输入	左上角横坐标。
y0	输入	左上角纵坐标。
x1	输入	右下角横坐标。
y1	输入	右下角纵坐标。

5.4.1.32 KeyPointInfo

功能

面部关键点（左/右眼、鼻尖、左/右嘴角）坐标信息。

结构定义

```
struct KeyPointInfo {
    float kPBefore[LANDMARK_LEN];
};
```

参数说明

参数名	输入/输出	说明
kPBefore	输入	关键点坐标数组。

5.4.1.33 HungarianHandle

功能

匈牙利算法，一种在多项式时间内求解任务分配问题的组合优化算法。

结构定义

```
struct HungarianHandle {
    int rows;
    int cols;
    int max;
    int* resX;
    int* resY;
    bool transpose;
    std::shared_ptr<int> adjMat;
    std::shared_ptr<int> xMatch;
    std::shared_ptr<int> yMatch;
    std::shared_ptr<int> xValue;
    std::shared_ptr<int> yValue;
    std::shared_ptr<int> slack;
    std::shared_ptr<int> xVisit;
    std::shared_ptr<int> yVisit;
};
```

参数说明

参数名	输入/输出	说明
rows	输入	匹配矩阵行。
cols	输入	匹配矩阵列。
max	输出	匹配矩阵行与列最大值。
resX	输出	点集**x**匹配结果。
resY	输出	点集**y**匹配结果。

参数名	输入/输出	说明
transpose	输出	矩阵转置标识符。
adjMat	输入	权值矩阵。
xMatch	输出	点集x匹配值。
yMatch	输出	点集y匹配值。
xValue	输出	点集x顶标值。
yValue	输出	点集y值，默认为0。
slack	输出	slack数组。
xVisit	输出	点集x匹配标识符。
yVisit	输出	点集y匹配标识符。

5.4.1.34 IOUMethod

功能

IOU计算方式。

结构定义

```
enum IOUMethod {
    MAX = 0,
    MIN = 1,
    UNION = 2,
    DIOU = 3
};
```

参数说明

参数名	输入/输出	说明
MAX	输入	重叠区域除以两者中面积大的。
MIN	输入	重叠区域除以两者中面积小的。
UNION	输入	重叠区域除以两者面积并集。
DIOU	输入	重叠区域除以两者面积并集减去距离的交并比。

5.4.1.35 EncodeH26xInfo

功能

视频编码回调传入数据。

结构定义

```
struct EncodeH26xInfo {
    std::function<void(std::shared_ptr<uint8_t>, uint32_t)> func = {};
};
```

参数说明

参数名	输入/输出	说明
func	输入	用于视频编码回调处理的函数对象。 函数对象的第一个参数“std::shared_ptr<uint8_t>”为输出的视频流数据（Host侧的数据），第二个参数“uint32_t”为视频流数据大小。

5.4.1.36 VencConfig

功能

视频编码结构体。

结构定义

```
struct VencConfig {
    uint32_t width = 0;
    uint32_t height = 0;
    MxbaseStreamFormat outputVideoFormat = MXBASE_STREAM_FORMAT_H264_MAIN_LEVEL;
    MxbasePixelFormat inputImageFormat = MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_420;
    uint32_t keyFrameInterval = 30;
    uint32_t deviceId  = 0;
    pthread_t encoderThreadId = 0;
    bool stopEncoderThread = false;
    uint32_t srcRate = 0;
    uint32_t rcMode = 0;
    uint32_t maxBitRate = 0;
    uint32_t ipProp = 0;
    std::function<void(std::shared_ptr<uint8_t>, uint32_t)>* userData;
};
```

参数说明

参数名	输入/输出	说明
width	输入	输入宽度。
height	输入	输入高度。
outputVideoFormat	输入	输出码流格式。
inputImageFormat	输入	输入图片格式。
keyFrameInterval	输入	I帧间隔。
deviceId	输入	设备编号。
encoderThreadId	输入	回调线程编号。

参数名	输入/输出	说明
stopEncoderThread	输入	回调线程是否在运行的标记位。
srcRate	输入	输入码流帧率，单位fps。
rcMode	输入	指定码率控制模式。
maxBitRate	输入	输出码率，单位kbps。
ipProp	输入	一个GOP内单个I帧bit数和单个P帧bit数的比例。
userData	输入	用户自定义数据。

5.4.1.37 PostProcessorImageInfo

功能

视频编码回调传入数据。

结构定义

```
struct PostProcessorImageInfo {
    std::vector<MxBase::PostImageInfo> postImageInfoVec;
    bool useMpPictureCrop = false;
};
```

参数说明

参数名	输入/输出	说明
postImageInfoVec	输入	图片信息（包括原图和缩放宽高，目标框坐标）。
useMpPictureCrop	输入	是否还原坐标到目标框坐标上。

5.4.1.38 AspectRatioPostImageInfo

功能

继承了PostImageInfo类，并增加了图片缩放相关属性。

结构定义

```
struct AspectRatioPostImageInfo : PostImageInfo {
    enum ResizeType {
        RESIZER_STRETCHING = 0,
        RESIZER_TF_KEEP_ASPECT_RATIO,
        RESIZER_MS_KEEP_ASPECT_RATIO,
        RESIZER_ONLY_PADDING,
        RESIZER_KEEP_ASPECT_RATIO_LONG,
        RESIZER_KEEP_ASPECT_RATIO_SHORT,
        RESIZER_RESCALE,
```

```
        RESIZER_RESCALE_DOUBLE,
        RESIZER_MS_YOLOV4
    };
    float keepAspectRatioScaling = 0;
    ResizeType resizeType;
};
```

参数说明

参数名	输入/输出	说明
keepAspectRatioScaling	输入	图片缩放比例。
resizeType	输入	图片缩放方式。 ● RESIZER_STRETCHING ：拉伸缩放。 ● RESIZER_TF_KEEP_ASPECT_RATIO ：对应FastRCNN缩放方式。 ● RESIZER_MS_KEEP_ASPECT_RATIO ：等比缩放。 ● RESIZER_ONLY_PADDING：按原始长宽进行填充。 ● RESIZER_KEEP_ASPECT_RATIO_LONG：按长边比例缩放。 ● RESIZER_KEEP_ASPECT_RATIO_SHORT：按短边比例缩放。 ● RESIZER_RESCALE：按长宽缩放的较小比例拉伸缩放。 ● RESIZER_RESCALE_DOUBLE：按长宽缩放的较小比例拉伸缩放两次。 ● RESIZER_MS_YOLOV4：对应YOLOv4缩放方式。

5.4.1.39 ObjectInfo

功能

目标检测类任务的目标框信息。

结构定义

```
class ObjectInfo {
public:
    float x0 = 0;
    float y0 = 0;
    float x1 = 0;
    float y1 = 0;
    float confidence = 0;
    float classId = 0;
    std::string className;
    std::vector<std::vector<uint8_t>> mask;
};
```

参数说明

参数名	说明
x0	目标左上角横坐标。
y0	目标左上角纵坐标。
x1	目标右下角横坐标。
y1	目标右下角纵坐标。
confidence	类别的置信度。
classId	类别的编号。
className	类别名称。
mask	实例分割任务使用，目标框内的像素图。

5.4.1.40 ClassInfo

功能

分类任务的分类信息。

结构定义

```
class ClassInfo {
public:
    int classId;
    float confidence;
    std::string className;
};
```

参数说明

参数名	说明
classId	类别的编号。
confidence	类别的置信度。
className	类别名称。

5.4.1.41 SemanticSegInfo

功能

语义分割的信息。

结构定义

```
class SemanticSegInfo {
public:
```

```
    std::vector<std::vector<int>> pixels;
    std::vector<std::string> labelMap;
};
```

参数说明

参数名	说明
pixels	图片中每个像素所属的类别ID。
labelMap	类别ID与类别名的映射关系。

5.4.1.42 TextsInfo

功能

文本信息。

结构定义

```
class TextsInfo {
public:
    std::vector<std::string> text;
};
```

参数说明

参数名	说明
text	文本信息。

5.4.1.43 TextObjectInfo

功能

文本框目标信息。

结构定义

```
class TextObjectInfo {
public:
    float x0;
    float y0;
    float x1;
    float y1;
    float x2;
    float y2;
    float x3;
    float y3;
    float confidence;
    std::string result;
};
```

参数说明

参数名	说明
x0	文本目标框（四边形）第一个点的横坐标。
y0	文本目标框（四边形）第一个点的纵坐标。
x1	文本目标框（四边形）第二个点的横坐标。
y1	文本目标框（四边形）第二个点的纵坐标。
x2	文本目标框（四边形）第三个点的横坐标。
y2	文本目标框（四边形）第三个点的纵坐标。
x3	文本目标框（四边形）第四个点的横坐标。
y3	文本目标框（四边形）第四个点的纵坐标。
confidence	文本目标框的置信度。
result	文本目标框的文字识别结果。

5.4.1.44 AttributeInfo

功能

分类任务的分类信息。

结构定义

```
class AttributeInfo {
public:
    int attrId;
    std::string attrName;
    std::string attrValue;
    float confidence;
};
```

参数说明

参数名	说明
attrId	属性ID。
attrName	属性名。
attrValue	属性值。
confidence	属性置信度。

5.4.1.45 ResizedImageInfo

功能

用于记录图像类任务中模型前处理中的缩放方式，提供给模型后处理的坐标还原使用。

结构定义

```
class ResizedImageInfo {
public:

ResizedImageInfo() {}
    ResizedImageInfo(uint32_t wResize, uint32_t hResize, uint32_t wOriginal, uint32_t hOriginal, ResizeType
rType,
        float kARScaling) : widthResize(wResize), heightResize(hResize), widthOriginal(wOriginal),
        heightOriginal(hOriginal), resizeType(rType), keepAspectRatioScaling(kARScaling) {}
    uint32_t widthResize = 0;
    uint32_t heightResize = 0;
    uint32_t widthOriginal = 0;
    uint32_t heightOriginal = 0;
    ResizeType resizeType = RESIZER_STRETCHING;
    float keepAspectRatioScaling = 0;
};
```

参数说明

参数名	说明
widthResize	缩放后图像宽，即模型的输入宽。
heightResize	缩放后图像高，即模型的输入高。
widthOriginal	缩放前图像宽。
heightOriginal	缩放前图像高。
resizeType	枚举类型ResizeType，代表图像的缩放方式。 ● “RESIZER_STRETCHING”：拉伸缩放，默认缩放方式。 ● “RESIZER_TF_KEEP_ASPECT_RATIO”：等比缩放，和TensorFlow框架FastRCNN模型缩放方式对应。 ● “RESIZER_MS_KEEP_ASPECT_RATIO”：等比缩放，使图片等比缩放至在指定宽高的区域内面积最大化。
keepAspectRatioScaling	等比例缩放的缩放比例，等比缩放缩放方式下生效。

5.4.1.46 CropRoiBox

功能

用于记录图像类任务中模型前处理中的感兴趣（region of interest，简称ROI）区域，提供给模型后处理的坐标还原使用。

结构定义

```
class CropRoiBox {
public:
    float x0;
    float y0;
    float x1;
    float y1;
};
```

参数说明

参数名	说明
x0	ROI框左上角横坐标。
y0	ROI框左上角纵坐标。
x1	ROI框右下角横坐标。
y1	ROI框右下角纵坐标。

5.4.1.47 DataFormat

功能

数据排布格式描述。

结构定义

```
enum DataFormat {
    NCHW = 0;
    NHWC = 1;
}
```

参数说明

参数名	说明
NCHW	数据按NCHW排布。
NHWC	数据按NHWC排布。

5.4.1.48 DynamicType

功能

动态类型描述。

结构定义

```
enum DynamicType {
    STATIC_BATCH = 0,
    DYNAMIC_BATCH = 1,
```

```
    DYNAMIC_HW = 2,
    DYNAMIC_DIMS = 3,
    DYNAMIC_SHAPE = 4
};
```

5.4.1.49 ImagePreProcessInfo

功能

用于记录图像类任务中模型前处理中的感兴趣（region of interest）区域，提供给模型后处理的坐标还原使用。

结构定义

```
class ImagePreProcessInfo {
public:
    // image
    uint32_t imageWidth = 0;          // memoryWidth
    uint32_t imageHeight = 0;         // memoryHeight
    uint32_t originalWidth = 0;         // originalWidth
    uint32_t originalHeight = 0;        // originalHeight

    // mapping parameters
    float xRatio = 1.0;
    float xBias = 0.0;
    float yRatio = 1.0;
    float yBias = 0.0;

    // valid region
    float x0Valid = 0.0;
    float y0Valid = 0.0;
    float x1Valid = 0.0;
    float y1Valid = 0.0;
};
```

参数说明

参数名	说明
imageWidth	图像宽。
imageHeight	图像高。
originalWidth	原图宽。
originalHeight	原图高。
xRatio	x方向伸缩比。
xBias	x方向偏置。
yRatio	y方向伸缩比。
yBias	y方向偏置。
x0Valid	图像中有效区域的左上角横坐标。
y0Valid	图像中有效区域的左上角纵坐标。
x1Valid	图像中有效区域的右下角横坐标。
y1Valid	图像中有效区域的右下角纵坐标。

5.4.1.50 CoorDim

功能

用于坐标的信息描述。

结构定义

```
struct TextObjectPostProcessBase::CoorDim {
    int i;
    int j;
    int k;
    int index;
    CoorDim(int i, int j, int k, int idx):i(i), j(j), k(k), index(idx){}
};

struct OcrPostProcessors::CoorDim {
    int i;
    int j;
    CoorDim(int i, int j):i(i), j(j){}
};
```

参数说明

参数名	说明
i , j , k	坐标分量值。
index	坐标索引。

5.4.1.51 TensorArrangementType

功能

描述张量的组织形式。

结构定义

```
enum TensorArrangementType {
    TYPE_NHWC = 0,
    TYPE_NCHW = 1,
    TYPE_NHW = 2,
    TYPE_NWH = 3
};
```

参数说明

参数名	说明
TYPE_NHWC	以NHWC形式组织张量。
TYPE_NCHW	以NCHW形式组织张量。

参数名	说明
TYPE_NHW	以NHW形式组织张量。
TYPE_NWH	以NWH形式组织张量。

5.4.1.52 MxbasePixelFormat

功能

描述图片格式。

结构定义

```
enum MxbasePixelFormat {
    MXBASE_PIXEL_FORMAT_YUV_400 = 0, // 0
    MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_420 = 1, // 1
    MXBASE_PIXEL_FORMAT_YVU_SEMIPLANAR_420 = 2, // 2
    MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_422 = 3, // 3
    MXBASE_PIXEL_FORMAT_YVU_SEMIPLANAR_422 = 4, // 4
    MXBASE_PIXEL_FORMAT_YUV_SEMIPLANAR_444 = 5, // 5
    MXBASE_PIXEL_FORMAT_YVU_SEMIPLANAR_444 = 6, // 6
    MXBASE_PIXEL_FORMAT_YUYV_PACKED_422 = 7, // 7
    MXBASE_PIXEL_FORMAT_UYVY_PACKED_422 = 8, // 8
    MXBASE_PIXEL_FORMAT_YVYU_PACKED_422 = 9, // 9
    MXBASE_PIXEL_FORMAT_VYUY_PACKED_422 = 10, // 10
    MXBASE_PIXEL_FORMAT_YUV_PACKED_444 = 11, // 11
    MXBASE_PIXEL_FORMAT_RGB_888 = 12, // 12
    MXBASE_PIXEL_FORMAT_BGR_888 = 13, // 13
    MXBASE_PIXEL_FORMAT_ARGB_8888 = 14, // 14
    MXBASE_PIXEL_FORMAT_ABGR_8888 = 15, // 15
    MXBASE_PIXEL_FORMAT_RGBA_8888 = 16, // 16
    MXBASE_PIXEL_FORMAT_BGRA_8888 = 17, // 17
    MXBASE_PIXEL_FORMAT_ANY = 100,
    MXBASE_PIXEL_FORMAT_JPEG = 101,
    MXBASE_PIXEL_FORMAT_PNG = 102,
    MXBASE_PIXEL_FORMAT_BOTTOM = 103,
};
```

5.4.1.53 MxbaseStreamFormat

功能

获取视频编码处理通道的描述信息：视频编码协议。

结构定义

```
enum MxbaseStreamFormat {
    MXBASE_STREAM_FORMAT_H265_MAIN_LEVEL = 0,
    MXBASE_STREAM_FORMAT_H264_BASELINE_LEVEL = 1,
    MXBASE_STREAM_FORMAT_H264_MAIN_LEVEL = 2,
    MXBASE_STREAM_FORMAT_H264_HIGH_LEVEL = 3,
};
```

参数说明

参数名	说明
MXBASE_STREAM_FORMAT_H265_MAIN_LEVEL	H.265主流画质。
MXBASE_STREAM_FORMAT_H264_BASELINE_LEVEL	H.264基本画质。
MXBASE_STREAM_FORMAT_H264_MAIN_LEVEL	H.264主流画质。
MXBASE_STREAM_FORMAT_H264_HIGH_LEVEL	H.264高级画质。

5.4.1.54 DecodeH26xInfo

功能

解码H.264或H.265的相关信息结构体。

结构定义

```
struct DecodeH26xInfo {
    DecodeH26xInfo(uint32_t i, uint32_t i1, DecodeCallBackFunction pFunction, void *pVoid): channelId(i),
frameId(i1), callbackFunc(pFunction), userData(pVoid) {}
    uint32_t channelId = 0;
    uint32_t frameId = 0;
    DecodeCallBackFunction callbackFunc = nullptr;
    void* userData;
    bool userMalloc = false;
};
```

参数说明

参数名	说明
channelId	视频路ID。
frameId	帧ID。
callbackFunc	CallBack函数。
userData	用户自定义数据。
userMalloc	数据内存是否由用户申请。

5.4.1.55 CropResizePasteConfig

功能

抠图坐标及贴图坐标。

结构定义

```
struct CropResizePasteConfig {
    // CROP CONFIG
    uint32_t cropLeft;
    uint32_t cropRight;
    uint32_t cropTop;
    uint32_t cropBottom;
    // PASTE CONFIG
    uint32_t pasteLeft;
    uint32_t pasteRight;
    uint32_t pasteTop;
    uint32_t pasteBottom;
    // RESIZE CONFIG
    uint32_t interpolation;
};
```

参数说明

参数名	说明
cropLeft	抠图坐标左边。
cropRight	抠图坐标右边。
cropTop	抠图坐标上边。
cropBottom	抠图坐标下边。
pasteLeft	贴图坐标左边。
pasteRight	贴图坐标右边。
pasteTop	贴图坐标上边。
pasteBottom	贴图坐标下边。
interpolation	指定缩放算子，取值范围： ● “0”：默认值，华为自研的最近邻插值算法。 ● “1”：业界通用的Bilinear算法，当前不支持。 ● “2”：业界通用的Nearest neighbor算法，当前不支持。

5.4.1.56 ImageConstrainInfo

功能

图片对齐相关参数。

结构定义

```
struct ImageConstrainInfo {
    uint32_t minWidthStride;
    uint32_t maxWidthStride;
    uint32_t minHeightStride;
    uint32_t maxHeightStride;
    uint32_t widthStrideAlign;
    uint32_t heightStrideAlign;
```

```
    uint32_t widthAlign;
    uint32_t heightAlign;
    float ratio;
    uint32_t pixelBit;
};
```

参数说明

参数名	说明
minWidthStride	宽对齐最小值。
maxWidthStride	宽对齐最大值。
minHeightStride	高对齐最小值。
maxHeightStride	高对齐最大值。
widthStrideAlign	宽对齐的对齐方式。
heightStrideAlign	高对齐的对齐方式。
widthAlign	宽对齐方式。
heightAlign	高对齐方式。
ratio	比例。
pixelBit	像数点数据。

5.4.1.57 ImageSize

功能

图片尺寸信息。

结构定义

```
namespace MxBase{
struct ImageSize {
    size_t height;
    size_t width;
    ImageSize() = default;
    ImageSize(size_t height, size_t width)
    {
        this->width = width;
        this->height = height;
    }
}
namespace MxTools{
struct ImageSize {
    size_t height;
    size_t width;
    size_t area;
    ImageSize(int height, int width)
    {
        this->width = static_cast<size_t>(width);
        this->height = static_cast<size_t>(height);
        this->area = static_cast<size_t>(height * width);
    }
}
```

参数说明

参数名	说明
height	图片的高。
width	图片的宽。

5.4.1.58 DynamicInfo

功能

模型支持的类型。

结构定义

```
struct DynamicInfo {
    DynamicType dynamicType = DYNAMIC_BATCH;
    size_t batchSize;
    ImageSize imageSize = {};
    std::vector<std::vector<uint32_t>> shape = {};
};
```

参数说明

参数名	说明
dynamicType	动态类型描述。
batchSize	batch大小。
imageSize	图像大小。
shape	动态Batch的形状。

5.4.1.59 RoiBox

功能

定义裁剪框。

结构定义

```
struct RoiBox {
    float x0;
    float y0;
    float x1;
    float y1;
};
```

参数说明

参数名	说明
x0	左上角横坐标。
y0	左上角纵坐标。
x1	右下角横坐标。
y1	右下角纵坐标。

5.4.1.60 TextObjDetectInfo

功能

定义检测框的四个坐标点和置信度

结构定义

```
struct TextObjDetectInfo {
    float x0;
    float y0;
    float x1;
    float y1;
    float x2;
    float y2;
    float x3;
    float y3;
    float confidence;
};
```

参数说明

参数名	说明
x0	检测框其中一个横坐标。
y0	检测框其中一个纵坐标。
x1	检测框其中一个横坐标。
y1	检测框其中一个纵坐标。
x2	检测框其中一个横坐标。
y2	检测框其中一个纵坐标。
x3	检测框其中一个横坐标。
y4	检测框其中一个纵坐标。
confidence	置信度。

5.4.1.61 TensorDataType

功能

TensorDataType枚举类。

结构定义

```
enum TensorDataType {
    TENSOR_DTYPE_UNDEFINED = -1,
    TENSOR_DTYPE_FLOAT32 = 0,
    TENSOR_DTYPE_FLOAT16 = 1,
    TENSOR_DTYPE_INT8 = 2,
    TENSOR_DTYPE_INT32 = 3,
    TENSOR_DTYPE_UINT8 = 4,
    TENSOR_DTYPE_INT16 = 6,
    TENSOR_DTYPE_UINT16 = 7,
    TENSOR_DTYPE_UINT32 = 8,
    TENSOR_DTYPE_INT64 = 9,
    TENSOR_DTYPE_UINT64 = 10,
    TENSOR_DTYPE_DOUBLE64 = 11,
    TENSOR_DTYPE_BOOL = 12
};
```

5.4.1.62 TrackFlag

功能

目标检测跟踪的状态。

结构定义

```
enum TrackFlag {
    NEW_OBJECT = 0,
    TRACKED_OBJECT = 1,
    LOST_OBJECT = 2
};
```

参数说明

参数名	说明
NEW_OBJECT	新目标。
TRACKED_OBJECT	正在跟踪。
LOST_OBJECT	遗失目标。

5.4.1.63 MxbaseDvppChannelMode

功能

DvppWrapper初始化配置。

结构定义

```
enum MxbaseDvppChannelMode {
    MXBASE_DVPP_CHNMODE_DEFAULT = 0,  // default mode, contain VPC, JPEGD and JPEGE mode
```

```
    MXBASE_DVPP_CHNMODE_VPC = 1,
    MXBASE_DVPP_CHNMODE_JPEGD = 2,
    MXBASE_DVPP_CHNMODE_JPEGE = 3,
    MXBASE_DVPP_CHNMODE_PNGD = 4,
};
```

参数说明

参数名	说明
MXBASE_DVPP_CHNMODE_DEFAULT	默认配置。
MXBASE_DVPP_CHNMODE_VPC	VPC图像处理。
MXBASE_DVPP_CHNMODE_JPEGD	JPEG图像解码。
MXBASE_DVPP_CHNMODE_JPEGE	JPEG图像编码。
MXBASE_DVPP_CHNMODE_PNGD	PNG解码。

5.4.1.64 DeviceContext

功能

设备内容设置。

结构定义

```
struct DeviceContext {
    enum DeviceStatus {
        IDLE = 0,  // idle status
        USING      // running status
    } devStatus = IDLE;
    int32_t devId = 0;
};
```

参数说明

参数名	说明
devStatus	设备状态，默认为**idle**闲置。
devId	设备ID。

5.4.1.65 ResizeType

功能

枚举类型ResizeType，代表图像的缩放方式。

结构定义

```
enum ResizeType {
    RESIZER_STRETCHING = 0,
```

```
    RESIZER_TF_KEEP_ASPECT_RATIO,
    RESIZER_MS_KEEP_ASPECT_RATIO,
    RESIZER_ONLY_PADDING,
    RESIZER_KEEP_ASPECT_RATIO_LONG,
    RESIZER_KEEP_ASPECT_RATIO_SHORT,
    RESIZER_RESCALE,
    RESIZER_RESCALE_DOUBLE,
    RESIZER_MS_YOLOV4,
};
```

参数说明

参数名	说明
RESIZER_STRETCHING	拉伸缩放。
RESIZER_TF_KEEP_ASPECT_RATIO	对应TensorFlow框架FastRCNN模型缩放方式。
RESIZER_MS_KEEP_ASPECT_RATIO	等比缩放。
RESIZER_ONLY_PADDING	按原始长宽进行填充。
RESIZER_KEEP_ASPECT_RATIO_LONG	按长边比例缩放。
RESIZER_KEEP_ASPECT_RATIO_SHORT	按短边比例缩放。
RESIZER_RESCALE	按长宽缩放的较小比例拉伸缩放。
RESIZER_RESCALE_DOUBLE	按长宽缩放的较小比例拉伸缩放两次。
RESIZER_MS_YOLOV4	对应YOLOv4模型缩放方式。

5.4.1.66 KeyPointDetectionInfo

功能

关键点检测信息。

结构定义

```
class KeyPointDetectionInfo {
public:
    std::map<int, std::vector<float>> keyPointMap;
    std::map<int, float> scoreMap;
    float score;
};
```

参数说明

参数名	说明
keyPointMap	每个关键点的坐标等信息。
scoreMap	各每个关键点对应的置信度。
score	整体置信度。

5.4.1.67 AppGlobalCfg

功能

全局应用配置。

结构定义

```
struct AppGlobalCfg {
    uint32_t vpcChnNum = DEFAULT_VPC_CHN_NUM;
};
```

参数说明

参数名	说明
vpcChnNum	VPC通道资源池大小。默认值为“DEFAULT_VPC_CHN_NUM = 48”。取值范围为[1, 128]。

5.4.1.68 ReduceDim

功能

用于描述规约轴的枚举类型。

结构定义

```
enum class ReduceDim{
    REDUCE_HEIGHT = 0,
    REDUCE_WIDTH = 1
};
```

参数说明

参数名	说明
REDUCE_HEIGHT	用于描述规约高维度。
REDUCE_WIDTH	用于描述规约宽维度。

5.4.1.69 ReduceType

功能

用于描述规约操作的枚举类型。

结构定义

```
enum class ReduceType{
    REDUCE_SUM = 0,
    REDUCE_MEAN = 1,
```

```
    REDUCE_MAX = 2,
    REDUCE_MIN = 3
};
```

参数说明

参数名	说明
REDUCE_SUM	用于描述求和规约操作。
REDUCE_MEAN	用于描述求平均规约操作。
REDUCE_MAX	用于描述求最大值规约操作。
REDUCE_MIN	用于描述求最小值规约操作。

5.4.1.70 CmpOp

功能

张量比较类型。

结构定义

```
enum class CmpOp {
    CMP_EQ = 0,
    CMP_NE,
    CMP_LT,
    CMP_GT,
    CMP_LE,
    CMP_GE
};
```

参数说明

参数名	说明
CMP_EQ	相等操作，代表“==”。
CMP_NE	不相等操作，代表“!=”。
CMP_LT	小于操作，代表“<”。
CMP_GT	大于操作，代表“>”。
CMP_LE	小于等于操作，代表“<=”。
CMP_GE	大于等于操作，代表“>=”。

5.4.1.71 WarpAffineMode

功能

仿射变化的插值方式，当前仅支持线性插值。

结构定义

```
enum class WarpAffineMode {
    INTER_LINEAR = 0
};
```

参数说明

参数名	说明
INTER_LINEAR	线性插值方式

5.4.1.72 WarpPerspectiveMode

功能

透射变化的插值方式，当前仅支持线性插值。

结构定义

```
enum class WarpPerspectiveMode {
    INTER_LINEAR = 0
};
```

参数说明

参数名	说明
INTER_LINEAR	线性插值方式

5.4.1.73 PaddingMode

功能

仿射变化和透射变化的补边方式，当前仅支持常量补边。

结构定义

```
enum class PaddingMode {
    PADDING_CONST = 0
};
```

参数说明

参数名	说明
PADDING_CONST	常量补边方式

5.5 基础组件层

5.5.1 MemoryHelper

5.5.1.1 类说明

该类负责Host侧和Device侧内存的管理：分配、初始化、移动和释放。

5.5.1.2 Malloc

函数功能

内存分配函数，通过给MemoryData结构体type赋值，确定内存是Host、Device或者DVPP。通过MemoryData中size，指定内存大小。通过MemoryData中deviceId，指定Device内存ID。并且申请的内存，在销毁时，需要用户自行搭配函数**5.5.1.3 Free**进行销毁。

函数原型

```
static APP_ERROR Malloc(MemoryData& data);
```

参数说明

参数名	输入/输出	说明
data	输入	待分配的内存数据。其输入是内存的类型和大小，分别赋值到data.type，data.size，输出是分配的内存指针，赋值到data.ptrData。 “data.size”取值范围为[1, 4294967296]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.3 Free

须知

当采用该API进行内存销毁时，需要确保待销毁的内存未被手动执行**free**等销毁函数，否则可能出现double free的情况。

函数功能

释放指定内存。

函数原型

```
static APP_ERROR Free(MemoryData& data);
```

参数说明

参数名	输入/输出	说明
data	输入	待释放的内存数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.4 Memset

函数功能

内存设置函数。

函数原型

```
static APP_ERROR Memset(MemoryData& data, int32_t value, size_t count);
static APP_ERROR Memset(MemoryData& data, int32_t value, size_t count, AscendStream &stream);
```

参数说明

参数名	输入/输出	说明
data	输入	待设置的内存数据。“data.size”取值范围为[1, 4294967296]。
value	输入	设置的值。

参数名	输入/输出	说明
count	输入	设置的数据长度。长度不可大于内存实际长度，否则可能会导致出现coredump情况。
stream	输入	**AscendStream**类型，当参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.5 Memcpy

函数功能

内存复制函数，根据MemoryData中指定的内存位置在Host侧和Device侧之间进行复制。

函数原型

```
static APP_ERROR Memcpy(MemoryData& dest, const MemoryData& src, size_t count);
```

参数说明

参数名	输入/输出	说明
dest	输入	待复制的目标内存。 “dest.ptrData”不能为空指针。
src	输入	待复制的源内存。 “src.ptrData”不能为空指针。
count	输入	复制的数据长度。长度的设置应参考目标内存及源内存长度，错误的设置可能会导致出现coredump情况。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.6 MxbsMalloc

函数功能

内存分配函数，通过给MemoryData结构体type赋值，确定内存是Host、Device或者DVPP。通过MemoryData中size，指定内存大小。通过MemoryData中deviceId，指定Device内存ID。需要销毁申请的内存时，用户自行搭配**5.5.1.7 MxbsFree**函数进行销毁。

说明

由于Malloc和系统函数重名，建议优先使用**MxbsMalloc**函数。

函数原型

```
static APP_ERROR MxbsMalloc(MemoryData& data);
```

参数说明

参数名	输入/输出	说明
data	输入	待分配的内存数据。其输入是内存的类型和大小，分别赋值到data.type，data.size，输出是分配的内存指针，赋值到data.ptrData。 “data.size”取值范围为[1, 4294967296]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.7 MxbsFree

须知

当采用该API进行内存销毁时，需要确保待销毁的内存未被手动执行free等销毁函数，否则可能出现double free的情况。

函数功能

释放指定内存。

说明

由于Free和系统函数重名，建议优先使用MxbsFree函数。

函数原型

```
static APP_ERROR MxbsFree(MemoryData& data);
```

参数说明

参数名	输入/输出	说明
data	输入	待释放的内存数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.8 MxbsMemset

函数功能

内存设置函数。

说明

由于Memset和系统函数重名，建议优先使用MxbsMemset函数。

函数原型

```
static APP_ERROR MxbsMemset(MemoryData& data, int32_t value, size_t count);
static APP_ERROR MxbsMemset(MemoryData& data, int32_t value, size_t count, AscendStream &stream);
```

参数说明

参数名	输入/输出	说明
data	输入	待设置的内存数据。“data.size”取值范围为[1, 4294967296]。
value	输入	设置的值。
count	输入	设置的数据长度。长度不可大于内存实际长度，否则可能会导致出现coredump情况。
stream	输入	**AscendStream**类型，当参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.9 MxbsMemcpy

函数功能

内存复制函数，根据MemoryData中指定的内存位置在Host侧和Device侧之间进行复制。

说明

由于Memcpy和系统函数重名，建议优先使用MxbsMemcpy函数。

函数原型

```
static APP_ERROR MxbsMemcpy (MemoryData& dest, MemoryData& src, size_t count);
```

参数说明

参数名	输入/输出	说明
dest	输入	待复制的目标内存。“dest.ptrData”不能为空指针。
src	输入	待复制的源内存。 “src.ptrData”不能为空指针。

参数名	输入/输出	说明
count	输入	复制的数据长度。长度的设置应参考目标内存及源内存长度，错误的设置可能会导致出现coredump情况。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.10 MxbsMallocAndCopy

函数功能

内存复制函数，根据MemoryData中指定的内存位置在Host侧和Device侧之间进行dest内存申请及复制。

函数原型

```
static APP_ERROR MxbsMallocAndCopy(MemoryData& dest, const MemoryData& src);
```

参数说明

参数名	输入/输出	说明
dest	输入	待复制的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.11 SetMaxDataSize

函数功能

设置MemoryData对象所允许的最大内存大小。

函数原型

```
static APP_ERROR SetMaxDataSize(long size);
```

参数说明

参数名	输入/输出	说明
size	输入	最大内存大小，单位：字节。（限制范围：1 ≤ size ≤ 4294967296）

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.12 CheckDataSize

函数功能

检测MemoryData对象内存大小是否符合要求。内存大小不允许为0字节。

函数原型

```
static APP_ERROR CheckDataSize(long size);
```

参数说明

参数名	输入/输出	说明
size	输入	MemoryData对象内存大小，检查范围为[1, 4294967296]，单位：字节。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.13 CheckDataSizeAllowZero

函数功能

检测MemoryData对象内存大小是否符合要求。内存大小允许为0字节。

函数原型

```
static APP_ERROR CheckDataSizeAllowZero(long size);
```

参数说明

参数名	输入/输出	说明
size	输入	MemoryData对象内存大小，检查范围为[0, 4294967296]，单位：字节。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.1.14 MakeShared

函数功能

使用std::make_shared创建对象，创建失败时返回空指针。

函数原型

```
template<typename T, typename... Args>
static std::shared_ptr<T> MakeShared(Args && ... args);
```

参数说明

参数名	输入/输出	说明
T	输入	对象类名。
args	输入	创建的T对象构造函数参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2 DvppWrapper

5.5.2.1 类说明

该类封装DVPP基本的编码、解码、抠图和缩放功能。

5.5.2.2 DvppWrapper

须知

如出现**因内存耗尽，导致构造函数执行失败抛出**的异常时，请勿继续调用后续的成员函数。

函数功能

类构造函数，创建DvppWrapper对象（数字视觉预处理）。

函数原型

```
DvppWrapper();
```

5.5.2.3 Init

函数功能

DVPP初始化函数，必须与**5.5.2.4 DeInit**接口搭配使用。

适用于除了视频解码、视频编码以外的其他功能（视频解码的初始化须使用**5.5.2.5 InitVdec**，视频编码的初始化必须使用**5.5.2.21 InitVenc**）。初始化后必须调用对应功能的去初始化函数来释放资源。

函数原型

```
APP_ERROR DvppWrapper::Init(void);  //Atlas 200/500 A2 推理产品环境下使用
```

```
APP_ERROR DvppWrapper::Init(MxbaseDvppChannelMode dvppChannelMode);  // Atlas 200/500 A2 推理产品环境下通用
```

参数说明

参数名	输入/输出	说明
dvppChannelMode	输入	通道模式。 ● Atlas 200/500 A2 推理产品环境下默认0。 `MXBASE_DVPP_CHNMODE_DEFAULT = 0  // 包含 VPC， JPEGD 、 JPEGE和PNG`

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.4 DeInit

函数功能

DVPP去初始化函数，用于完成相关资源的释放，必须与5.5.2.3 Init接口搭配使用。

适用于除了视频解码、视频编码以外的其他功能（视频解码的去初始化须使用5.5.2.6 DeInitVdec，视频编码的去初始化必须使用5.5.2.22 DeInitVenc）。

函数原型

```
APP_ERROR DeInit(void);
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.5 InitVdec

函数功能

DVPP视频解码初始化函数，用于完成相关资源的申请，必须与5.5.2.6 DeInitVdec接口搭配使用。

函数原型

```
APP_ERROR InitVdec(VdecConfig& vdecConfig);
```

参数说明

参数名	输入/输出	说明
vdecConfig	输入	视频解码的配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.6 DeInitVdec

函数功能

DVPP视频解码去初始化函数，用于完成相关资源的释放，必须与5.5.2.5 InitVdec接口搭配使用。

函数原型

```
APP_ERROR DeInitVdec();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.7 DvppVdec

函数功能

用于实现DVPP对H.264/H.265格式视频的解码，调用该接口前必须调用5.5.2.5 InitVdec接口来实现视频解码功能的初始化。

函数原型

```
APP_ERROR DvppVdec(DvppDataInfo& inputDataInfo, void* userData);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	被解码的数据。 ● “inputDataInfo.width”取值范围为[128, 4096]。 ● “inputDataInfo.height”取值范围为[128, 4096]。 ● “inputDataInfo.data”不为空且“inputDataInfo.dataSize”不为0。

参数名	输入/输出	说明
userData	输入	用户自定义数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.8 DvppVdecFlush

函数功能

发送完所有码流后，获取缓存数据并清空缓存队列，接口调用完成后将结束解码过程。

函数原型

```
APP_ERROR DvppVdecFlush();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.9 DvppPngDecode

函数功能

用于实现DVPP对PNG格式的解码，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR DvppPngDecode(
    DvppDataInfo& inputDataInfo,
    DvppDataInfo& outputDataInfo
);

APP_ERROR DvppPngDecode(
    const std::string& inputPicPath,
    DvppDataInfo& outputDataInfo
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	被解码的数据。
inputPicPath	输入	被解码的图片路径。
outputDataInfo	输出	解码后的输出数据。 “outputDataInfo.dataSize”取值范围为[1, 4294967296]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.10 DvppJpegEncode

函数功能

用于实现DVPP对JPEG格式的编码，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR DvppJpegEncode (
    DvppDataInfo& inputDataInfo ,
    DvppDataInfo& outputDataInfo ,
    uint32_t encodeLevel
);

APP_ERROR DvppJpegEncode (
    DvppDataInfo& inputDataInfo ,
    std::string outputPicPath ,
    std::string outputPicName ,
    uint32_t encodeLevel
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	输入的待编码数据。
outputDataInfo	输出	编码后的输出。
outputPicPath	输入	生成的图片路径。

参数名	输入/输出	说明
outputPicName	输入	生成的图片名称。 最终生成的图片会自动生成.jpg后缀，若目标路径中已存在同名的.jpg格式的文件，生成的图片将覆盖原文件。
encodeLevel	输入	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。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.11 DvppJpegDecode

函数功能

用于实现DVPP对JPEG格式的解码，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR DvppJpegDecode(
    DvppDataInfo& inputDataInfo,
    DvppDataInfo& outputDataInfo
);

APP_ERROR DvppJpegDecode(
    const std::string& inputPicPath,
    DvppDataInfo& outputDataInfo
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	被解码的数据。
inputPicPath	输入	被解码的图片路径。
outputDataInfo	输出	解码后的输出数据。 “outputDataInfo.dataSize”取值范围为[1, 4294967296]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.12 DvppJpegPredictDecSize

函数功能

根据存放JPEG图片数据的内存预估JPEG图片解码后所需的输出内存的大小。

函数原型

```
APP_ERROR DvppJpegPredictDecSize(const void *imageData, uint32_t dataSize, MxbasePixelFormat
outputPixelFormat, uint32_t &decSize);
```

参数说明

参数名	输入/输出	说明
imageData	输入	Host上存放JPEG图片数据的内存地址，不能为Device侧内存地址。
dataSize	输入	内存大小，单位为Byte。
outputPixelFormat	输入	解码后的输出图片的格式。
decSize	输出	预估JPEG图片解码后所需的输出内存的大小，单位Byte。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.13 VpcCrop

函数功能

用于实现抠图功能，调用该接口前必须调用5.5.2.3 Init接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcCrop (
    DvppDataInfo& inputDataInfo ,
    DvppDataInfo& outputDataInfo ,
```

```
    CropRoiConfig& cropConfig,
    AscendStream& stream
);
APP_ERROR VpcCrop (
    DvppDataInfo& inputDataInfo ,
    DvppDataInfo& outputDataInfo ,
    CropRoiConfig& cropConfig
 );
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	待抠图的数据。
outputDataInfo	输出	抠图后的数据。
cropConfig	输入	抠图配置。 抠图配置约束： ● “cropConfig.x1”要大于“cropConfig.x0”，且不能超过图片宽度。抠图宽度最小为10，最大为图片宽度。 ● “cropConfig.y1”要大于“cropConfig.y0”，且不能超过图片高度。抠图高度最小为6，最大为图片高度。
stream	输入	**AscendStream**类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.14 VpcBatchCrop

函数功能

用于实现批量抠图功能，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcBatchCrop (
    DvppDataInfo& inputDataInfo ,
```

```
    std::vector<DvppDataInfo>& outputDataInfoVec ,
    std::vector<CropRoiConfig>& cropConfigVec,
    AscendStream& stream
);
APP_ERROR VpcBatchCrop (
    DvppDataInfo& inputDataInfo ,
    std::vector<DvppDataInfo>& outputDataInfoVec ,
    std::vector<CropRoiConfig>& cropConfigVec
);
APP_ERROR VpcBatchCrop (
    std::vector<DvppDataInfo>& inputDataInfoVec ,
    std::vector<DvppDataInfo>& outputDataInfoVec ,
    std::vector<CropRoiConfig>& cropConfigVec,
    AscendStream& stream
);
APP_ERROR VpcBatchCrop (
    std::vector<DvppDataInfo>& inputDataInfoVec ,
    std::vector<DvppDataInfo>& outputDataInfoVec ,
    std::vector<CropRoiConfig>& cropConfigVec
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	待抠图的数据。
inputDataInfoVec	输入	待抠图的一组数据。
outputDataInfoVec	输出	抠图后输出的一组数据。长度不为0。
cropConfigVec	输入	抠图配置组。长度要求与outputDataInfoVec一致。数组中单个配置要求和VpcCrop方法的cropConfig一致。具体请参见**5.4.1.18 CropRoiConfig**。
stream	输入	**AscendStream**类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.15 VpcBatchCropMN

函数功能

用于实现图像处理批量抠图功能，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcBatchCropMN (
std::vector<DvppDataInfo>& inputDataInfoVec,
std::vector<DvppDataInfo>& outputDataInfoVec,
std::vector<CropRoiConfig>& cropConfigVec,
AscendStream& stream
);
APP_ERROR VpcBatchCropMN (
std::vector<DvppDataInfo>& inputDataInfoVec,
std::vector<DvppDataInfo>& outputDataInfoVec,
std::vector<CropRoiConfig>& cropConfigVec
);
```

参数说明

参数名	输入/输出	说明
inputDataInfoVec	输入	待抠图的一组数据。数组长度不超过12。
outputDataInfoVec	输出	抠图后输出的一组数据。 长度不为0，长度为inputDataInfoVec与cropConfigVec的乘积。数组长度不超过256。
cropConfigVec	输入	抠图配置组。数组长度不超过256。具体请参见**5.4.1.18 CropRoiConfig**。
stream	输入	**AscendStream**类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.16 VpcBatchCropResize

函数功能

用于完成批量抠图和大小缩放，调用该接口前必须调用**5.5.2.3 Init**接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcBatchCropResize(
    DvppDataInfo& inputDataInfo,
    std::vector<DvppDataInfo>& outputDataInfoVec,
```

```
    std::vector<CropRoiConfig>& cropConfigVec,
    ResizeConfig& resizeConfig,
    AscendStream& stream
);
APP_ERROR VpcBatchCropResize(
    DvppDataInfo& inputDataInfo,
    std::vector<DvppDataInfo>& outputDataInfoVec,
    std::vector<CropRoiConfig>& cropConfigVec,
    ResizeConfig& resizeConfig
);
APP_ERROR VpcBatchCropResize(
      DvppDataInfo& inputDataInfo,
      std::vector<DvppDataInfo>& outputDataInfoVec,
      std::vector<CropRoiConfig>& cropConfigVec,
      std::vector<ResizeConfig>& resizeConfigVec,
      AscendStream& stream
);
APP_ERROR VpcBatchCropResize(
      DvppDataInfo& inputDataInfo,
      std::vector<DvppDataInfo>& outputDataInfoVec,
      std::vector<CropRoiConfig>& cropConfigVec,
      std::vector<ResizeConfig>& resizeConfigVec
);
APP_ERROR VpcBatchCropResize(
      std::vector<DvppDataInfo>& inputDataInfoVec,
      std::vector<DvppDataInfo>& outputDataInfoVec,
      std::vector<CropRoiConfig>& cropConfigVec,
      std::vector<ResizeConfig>& resizeConfigVec,
      AscendStream& stream
);
APP_ERROR VpcBatchCropResize(
      std::vector<DvppDataInfo>& inputDataInfoVec,
      std::vector<DvppDataInfo>& outputDataInfoVec,
      std::vector<CropRoiConfig>& cropConfigVec,
      std::vector<ResizeConfig>& resizeConfigVec
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	待抠图缩放的数据。
inputDataInfoVec	输入	待抠图缩放的一组数据。长度要求与cropConfigVec一致。
cropConfigVec	输入	抠图配置组。长度不能为0。具体请参见**5.4.1.18 CropRoiConfig**。
resizeConfigVec	输入	缩放配置组。长度要求与cropConfigVec一致。
outputDataInfoVec	输出	抠图后的输出数据组。长度要求与cropConfigVec一致。数组中单个配置要求和VpcCrop方法的cropConfig一致。
stream	输入	**AscendStream**类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.17 VpcCropAndPaste

函数功能

校验图片约束信息，调用该接口前必须调用5.5.2.3 Init接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcCropAndPaste(const DvppDataInfo& inputDataInfo, DvppDataInfo& outputDataInfo,
        CropRoiConfig& pasteRoi, CropRoiConfig& cropRoi, AscendStream& stream);
APP_ERROR VpcCropAndPaste(const DvppDataInfo& inputDataInfo, DvppDataInfo& outputDataInfo,
        CropRoiConfig& pasteRoi, CropRoiConfig& cropRoi);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	原始图片相关信息。
outputDataInfo	输出	返回贴图相关信息。
pasteRoi	输入	贴图区域。
cropRoi	输入	原始图片抠图区域。
stream	输入	AscendStream类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.18 VpcResize

函数功能

用于完成图片的缩放，调用该接口前必须调用5.5.2.3 Init接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcResize (
    DvppDataInfo& inputDataInfo ,
    DvppDataInfo& outputDataInfo ,
    ResizeConfig& resizeConfig,
    AscendStream& stream
 );
APP_ERROR VpcResize (
    DvppDataInfo& inputDataInfo ,
    DvppDataInfo& outputDataInfo ,
    ResizeConfig& resizeConfig
 );
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	缩放数据。 ● inputDataInfo.width取值范围[10, 8192]。 ● inputDataInfo.height取值范围[6, 8192]。 ● inputDataInfo.heightStride取值范围[6, 8192]。 ● inputDataInfo.widthStride取值范围根据图像格式定义。
outputDataInfo	输出	缩放后的数据。
resizeConfig	输入	缩放配置。 ● resizeConfig.width范围[32, 8192]。 ● resizeConfig.height范围[6, 8192]。
stream	输入	AscendStream类型，当不输入AscendStream& stream或者参数值为“AscendStream::DefaultStream()”时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.19 VpcPadding

函数功能

用于实现图像处理补边功能，调用该接口前必须调用5.5.2.3 Init接口来实现该功能的初始化。

函数原型

```
APP_ERROR VpcPadding(
DvppDataInfo& inputDataInfo,
DvppDataInfo& outputDataInfo,
MakeBorderConfig& makeBorderConfig
);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	待补边的数据。 ● inputDataInfo.width取值范围[10, 4096]。 ● inputDataInfo.height取值范围[6, 4096]。 ● inputDataInfo.heightStride取值范围[6, 4096]。 ● inputDataInfo.widthStride取值范围根据图像格式定义。
outputDataInfo	输出	补边后的数据。
makeBorderConfig	输入	补边配置。YUV_SP_420和YVU_SP_420补边尺寸建议为偶数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.2.20 VpcPictureConstrainInfoCheck

函数功能

校验图片约束信息。

函数原型

```
static APP_ERROR VpcPictureConstrainInfoCheck(const DvppDataInfo& inputDataInfo,
AscendStream& stream);
static APP_ERROR VpcPictureConstrainInfoCheck(const DvppDataInfo& inputDataInfo);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	待校验的数据。
stream	输入	**AscendStream**类型，用于记录任务中产生的错误码。 当输入AscendStream& stream时，错误码记录到指定的stream中；当不输入AscendStream& stream时，错误码记录到defaultstream中。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.21 InitVenc

函数功能

DVPP视频编码初始化函数，用于完成相关资源的申请，必须与**5.5.2.22 DeInitVenc**接口搭配使用。

函数原型

```
APP_ERROR InitVenc(VencConfig vencConfig);
```

参数说明

参数名	输入/输出	说明
vencConfig	输入	视频编码的配置参数。 vencConfig.keyFrameInterval参数不能为0。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.22 DeInitVenc

函数功能

DVPP视频编码去初始化函数，用于完成相关资源的释放，必须与**5.5.2.22 DeInitVenc**接口搭配使用。

函数原型

```
APP_ERROR DeInitVenc();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.23 DvppVenc

函数功能

用于实现DVPP对H.264/H.265格式视频的编码，调用该接口前必须调用**5.5.2.21 InitVenc**接口来实现视频编码功能的初始化。

函数原型

```
APP_ERROR DvppVenc(DvppDataInfo& inputDataInfo, std::function<void(std::shared_ptr<uint8_t>,
uint32_t)>* handleFunc);
```

参数说明

参数名	输入/输出	说明
inputDataInfo	输入	被编码的数据。
handleFunc	输入	回调函数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.24 GetPictureDec

函数功能

获取输入图像的宽、高和颜色通道个数。

函数原型

```
APP_ERROR GetPictureDec(DvppImageInfo& imageInfo, DvppImageOutput& imageOutput);
```

参数说明

参数名	输入/输出	说明
imageInfo	输入	输入图片的格式、存放JPEG图片数据的内存的地址和内存大小（单位为Byte）。 JPEG图片数据的内存地址要求为Host侧地址，不能为DVPP侧内存。
imageOutput	输出	图像的宽、高以及颜色通道个数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.25 InitJpegEncodeChannel

函数功能

初始化JPEGE图像通道，用于JPEG编码。

函数原型

```
APP_ERROR InitJpegEncodeChannel(const JpegEncodeChnConfig& config);
```

参数说明

参数名	输入/输出	说明

config	输入	通道配置参数，取值范围为[32, 8192]，（即图像最大宽高为8192 * 8192）。该配置无效。
		当前仅支持配置图片编码的通道宽高（maxPicWidth、maxPicHeight），内部自动16对齐，当高小于宽时，高自动向上对齐到宽的长度，请根据实际编码场景图片预留适当的宽高。
		对应数据结构参见如下。
		struct JpegEncodeChnConfig {
		uint32_t maxPicWidth = MAX_HIMPI_VENC_PIC_WIDTH;
		uint32_t maxPicHeight = MAX_HIMPI_VENC_PIC_HEIGHT;
		};

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.26 InitJpegDecodeChannel

函数功能

初始化JPEGD图像通道，用于JPEG解码。

函数原型

```
APP_ERROR InitJpegDecodeChannel(const JpegDecodeChnConfig& config);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数。 对应数据结构参见如下（当前预留）。 struct JpegDecodeChnConfig {};

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.27 InitVpcChannel

函数功能

初始化VPC图像通道，用于图像处理功能（抠图、缩放、补边、抠图缩放、抠图贴图、色域转换）。

函数原型

```
APP_ERROR InitVpcChannel(const VpcChnConfig& config);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数。 对应数据结构参见如下（当前预留）。 struct VpcChnConfig {};

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.2.28 InitPngDecodeChannel

函数功能

初始化PNGD图像通道，用于PNG图片解码。

函数原型

```
APP_ERROR InitPngDecodeChannel(const PngDecodeChnConfig& config);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数。 对应数据结构参见如下（当前预留）。 struct PngDecodeChnConfig {};

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.3 ModelInferenceProcessor

5.5.3.1 类说明

该类完成模型推理功能的处理。

5.5.3.2 ModelInferenceProcessor

须知

如果使用堆申请内存方式创建ModelinferenceProcessor对象，在使用结束后，请务必释放ModelinferenceProcessor对象，防止出现不可预知的错误。

如出现**因内存耗尽，导致构造函数执行失败抛出**的异常时，请勿继续调用后续的成员函数。

函数功能

类构造函数，创建模型推理对象。

函数原型

```
ModelInferenceProcessor();
```

5.5.3.3 ~ModelInferenceProcessor

函数功能

ModelInferenceProcessor类的默认析构函数。

函数原型

```
~ModelInferenceProcessor() = default;
```

5.5.3.4 Init

函数功能

用于完成模型推理初始化。

函数原型

```
APP_ERROR Init (std::string modelPath, ModelDesc& modelDesc);
APP_ERROR Init (const std::string& modelPath);
```

参数说明

参数名	输入/输出	说明
modelPath	输入	模型文件路径。 路径上的模型要求属主为当前用户，权限不大于640。 最大只支持至4GB大小的模型。
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.3.5 DeInit

函数功能

用于模型推理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit (void);
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.3.6 ModelInference

函数功能

用于执行模型推理。需要先执行5.5.3.4 Init方法初始化模型。

函数原型

```
APP_ERROR ModelInference (
    std::vector<BaseTensor>& inputTensors,
    std::vector<BaseTensor>& outputTensors,
    size_t dynamicBatchSize = 0
);
APP_ERROR ModelInference (
    std::vector<BaseTensor>& inputTensors,
    std::vector<BaseTensor>& outputTensors,
    DynamicInfo dynamicInfo
);
APP_ERROR ModelInference (
    const std::vector<TensorBase>& inputTensors,
    std::vector<TensorBase>& outputTensors,
    DynamicInfo dynamicInfo
);
```

参数说明

参数名	输入/输出	说明
inputTensors	输入	输入张量，模型推理的输入数据。
outputTensors	输出	输出张量，模型推理的输出数据。
dynamicBatchSize	输入	指定模型推理时的批量大小Batch。
dynamicInfo	输入	动态Batch的信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.3.7 GetModelDesc

函数功能

用于获得模型描述信息。

函数原型

```
const ModelDesc& GetModelDesc() const;
```

返回参数说明

数据结构	说明
ModelDesc	具体请参考**5.4.1.23 ModelDesc**。

5.5.3.8 GetDataFormat

函数功能

用于获得数据格式。

函数原型

```
DataFormat GetDataFormat() const;
```

返回参数说明

数据结构	说明
DataFormat	具体请参考**5.4.1.47 DataFormat**。

5.5.3.9 GetDynamicType

函数功能

用于获得数据动态类型，包括静态BatchSize，动态BatchSize以及ImageSize（分辨率）。

函数原型

```
DynamicType GetDynamicType() const;
```

返回参数说明

数据结构	说明
DynamicType	具体请参考**5.4.1.48 DynamicType**。

5.5.3.10 GetInputDataType

函数功能

用于获得输入数据的类型。

函数原型

```
std::vector<TensorDataType> GetInputDataType() const;
```

返回参数说明

数据结构	说明
std::vector<TensorDataType>	具体请参考**5.4.1.61 TensorDataType**。

5.5.3.11 GetOutputDataType

函数功能

用于获得输出数据的类型。

函数原型

```
std::vector<TensorDataType> GetOutputDataType() const;
```

返回参数说明

数据结构	说明
std::vector<TensorDataType>	具体请参考**5.4.1.61 TensorDataType**。

5.5.3.12 GetInputFormat

函数功能

用于获得输入数据的格式。

函数原型

```
std::vector<size_t> GetinputFormat() const;
```

5.5.3.13 GetOutputFormat

函数功能

用于获得输出数据的格式。

函数原型

```
std::vector<size_t> GetOutputFormat() const;
```

5.5.3.14 GetDynamicImageSizes

函数功能

获取动态分辨率。

函数原型

```
const std::vector<ImageSize>& GetDynamicImageSizes() const;
```

返回参数说明

数据结构	说明
ImageSize	具体请参考5.4.1.57 ImageSize。

5.5.3.15 GetInputShape

函数功能

获得输入张量的形状。

函数原型

```
std::vector<std::vector<int64_t>> GetInputShape() const;
```

5.5.3.16 GetOutputShape

函数功能

获得输出张量的形状。

函数原型

```
std::vector<std::vector<int64_t>> GetOutputShape() const;
```

5.5.3.17 GetDynamicBatch

函数功能

获取动态Batch。

函数原型

```
std::vector<int64_t> GetDynamicBatch() const;
```

5.5.3.18 GetDynamicGearInfo

函数功能

获得模型支持的动态输入档位信息，支持动态Batch、动态分辨率和分档动态维度模型。

函数原型

```
std::vector<std::vector<uint64_t>> GetDynamicGearInfo();
```

返回参数说明

数据结构	说明
std::vector<std::vector<uint64_t>>	模型支持的输入Tensor档位信息。 ● **std::vector<std::vector<uint64_t>>**为所有档位信息。 ● **std::vector<uint64_t>**为某一档的具体值。 返回数据结构为空，表示获取分档信息失败或者模型不属于函数功能中描述的三种模型之一，具体的原因可查看报错信息确认。 例如，模型为动态分辨率模型，**std::vector<uint64_t>**为某一档的高、宽值。

5.5.4 Nms

5.5.4.1 NmsSortByArea

函数功能

根据面积大小过滤重复目标。

函数原型

```
void NmsSortByArea(std::vector<DetectBox>& detBoxes, const float iouThresh, const IOUMethod method = UNION);
```

```
void NmsSortByArea(std::vector<ObjectInfo>& detBoxes, const float iouThresh, const IOUMethod method = UNION);
```

参数说明

参数名	输入/输出	说明
detBoxes	输入/输出	待过滤DetectBox（中心点）/ ObjectInfo（边角点）。输出过滤后的目标信息。
iouThresh	输入	过滤阀值。
method	输入	IOU计算方式： ● MAX 重叠区域除以两者中面积大的。 ● MIN 重叠区域除以两者中面积小的。 ● UNION 重叠区域除以两者面积并集。 ● DIOU 重叠区域除以两者面积并集减去距离的交并比。

5.5.4.2 NmsSort

函数功能

根据置信度大小过滤重复目标。

函数原型

```
void NmsSort(std::vector<DetectBox>& detBoxes, float iouThresh, IOUMethod method = UNION);
```

```
void NmsSort(std::vector<ObjectInfo>& detBoxes, float iouThresh, IOUMethod method = UNION);
```

参数说明

参数名	输入/输出	说明
detBoxes	输出	待过滤DetectBox（中心点）/ ObjectInfo（边角点）。输出过滤后的目标信息。
iouThresh	输入	过滤阀值。
method	输入	IOU计算方式： ● MAX 重叠区域除以两者中面积大的。 ● MIN 重叠区域除以两者中面积小的。 ● UNION 重叠区域除以两者面积并集。 ● DIOU 重叠区域除以两者面积并集减去距离的交并比。

5.5.4.3 CalcIou

函数功能

计算IOU值。

函数原型

```
float CalcIou(DetectBox a, DetectBox b, IOUMethod method = UNION);
```

```
float CalcIou(ObjectInfo a, ObjectInfo b, IOUMethod method = UNION);
```

参数说明

参数名	输入/输出	说明
a	输入	DetectBox（中心点）/ ObjectInfo（边角点）。
b	输入	DetectBox（中心点）/ ObjectInfo（边角点）。

参数名	输入/输出	说明
method	输入	IOU计算方式： ● MAX 重叠区域除以两者中面积大的。 ● MIN 重叠区域除以两者中面积小的。 ● UNION 重叠区域除以两者面积并集。 ● DIOU 重叠区域除以两者面积并集减去距离的交并比。

返回参数说明

数据结构	说明
float	IOU值。

5.5.4.4 FilterByIou

函数功能

根据 IOU 值过滤目标。

函数原型

```
void FilterByIou(std::vector<DetectBox> dets, std::vector<DetectBox>& sortBoxes, float iouThresh,
IOUMethod method = UNION);
```

```
void FilterByIou(std::vector<ObjectInfo> dets, std::vector<ObjectInfo>& sortBoxes, float iouThresh,
IOUMethod method = UNION);
```

参数说明

参数名	输入/输出	说明
dets	输入	待过滤 DetectBox（中心点）/ ObjectInfo（边角点）。
sortBoxes	输出	排序过的数组。
iouThresh	输入	过滤阀值。
method	输入	IOU计算方式： ● MAX 重叠区域除以两者中面积大的。 ● MIN 重叠区域除以两者中面积小的。 ● UNION 重叠区域除以两者面积并集。 ● DIOU 重叠区域除以两者面积并集减去距离的交并比。

5.5.5 SimilarityTransform

5.5.5.1 SimilarityTransform

函数功能

类构造函数，创建相似变换类对象（图像变换的一种）。

函数原型

```
SimilarityTransform();
```

5.5.5.2 ~SimilarityTransform

函数功能

SimilarityTransform类的默认析构函数。

函数原型

```
~SimilarityTransform();
```

5.5.5.3 Transform

函数功能

计算仿射变换矩阵。

函数原型

```
cv::Mat Transform(const std::vector<cv::Point2f> &srcPoint, const std::vector<cv::Point2f> &dstPoint) const;
```

参数说明

参数名	输入/输出	说明
srcPoint	输入	转换前数组。向量中point个数不超过10000，坐标范围为[0，8192]。
dstPoint	输入	转换后数组。 长度需要与srcPoint一致。向量中point个数不超过10000，坐标范围为[0，8192]。

返回参数说明

数据结构	说明
cv::Mat	返回仿射变换矩阵。

5.5.6 WarpAffine

5.5.6.1 WarpAffine

函数功能

类构造函数，创建仿射变换类对象（图像变换的一种）。

函数原型

```
WarpAffine();
```

5.5.6.2 ~WarpAffine

函数功能

WarpAffine类的默认析构函数。

函数原型

```
~WarpAffine();
```

5.5.6.3 Process

函数功能

进行仿射变换。

函数原型

```
APP_ERROR Process(std::vector<MxBase::DvppDataInfo> &warpAffineDataInfoInputVec,
                  std::vector<MxBase::DvppDataInfo> &warpAffineDataInfoOutputVec,
                  std::vector<KeyPointInfo> &keyPointInfoVec, int picHeight, int picWidth);
```

参数说明

参数名	输入/输出	说明
warpAffineDataInfoInputVec	输入	变换前图片信息。
warpAffineDataInfoOutputVec	输出	变换后图片信息。长度需要与warpAffineDataInfoInputVec一致。

参数名	输入/输出	说明
keyPointInfoVec	输入	关键点参数。长度需要与warpAffineDataInfoInputVec一致。
picHeight	输入	图片高度。取值范围[32, 8192]。
picWidth	输入	图片宽度。取值范围[32, 8192]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.7 Hungarian

5.5.7.1 HungarianHandleInit

函数功能

初始化匈牙利算法匹配矩阵图。

函数原型

```
APP_ERROR HungarianHandleInit(HungarianHandle &handle, int row, int cols);
```

参数说明

参数名	输入/输出	说明
handle	输入	匈牙利算法专用结构体。
row	输入	匈牙利算法匹配矩阵行数。 限定范围(0, 8192]。
cols	输入	匈牙利算法匹配矩阵列数。 限定范围(0, 8192]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.7.2 HungarianSolve

须知

在调用HungarianSolve函数时，用户需保证已经使用HungarianHandleInit函数对相关参数进行了初始化，否则HungarianSolve函数无法正确执行。

函数功能

执行匈牙利算法，解决二部图最小权值匹配问题，需要先执行5.5.7.1 HungarianHandleInit方法。

函数原型

```
int HungarianSolve(HungarianHandle &handle, const std::vector<std::vector<int>> &cost, const int rows, const int cols);
```

参数说明

参数名	输入/输出	说明
handle	输入	匈牙利算法专用结构体。
cost	输入	权值矩阵。
rows	输入	匈牙利算法匹配矩阵行数。
cols	输入	匈牙利算法匹配矩阵列数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.8 KalmanTracker

5.5.8.1 KalmanTracker

函数功能

类构造函数，创建卡尔曼滤波器对象。

函数原型

```
KalmanTracker();
```

5.5.8.2 ~KalmanTracker

函数功能

KalmanTracker类的默认析构函数。

函数原型

```
~KalmanTracker();
```

5.5.8.3 Predict

须知

在调用Predict函数时，用户需保证已经使用CvKalmanInit函数对相关参数进行了初始化，否则Predict函数无法正确执行。

函数功能

使用卡尔曼滤波器对运动目标坐标框进行预测。需要先执行**5.5.8.5 CvKalmanInit**方法。

函数原型

```
MxBase::DetectBox Predict();
```

返回参数说明

数据结构	说明
MxBase::DetectBox	坐标框。

5.5.8.4 Update

须知

在调用Update函数时，用户需保证已经使用CvKalmanInit函数对相关参数进行了初始化，否则Update函数无法正确执行。

函数功能

更新卡尔曼滤波器。需要先执行**5.5.8.5 CvKalmanInit**方法。

函数原型

```
void Update(const MxBase::DetectBox &stateMat);
```

参数说明

参数名	输入/输出	说明
stateMat	输入	坐标框。

5.5.8.5 CvKalmanInit

函数功能

初始化卡尔曼滤波器。

函数原型

```
void CvKalmanInit(const MxBase::DetectBox &initRect);
```

参数说明

参数名	输入/输出	说明
initRect	输入	坐标框。

5.5.9 FastMath

5.5.9.1 类说明

FastMath类在FastMath.h定义。

5.5.9.2 FastMath

函数功能

类构造函数，创建数学函数类对象。

函数原型

```
FastMath();
```

5.5.9.3 ~FastMath

函数功能

FastMath类的默认析构函数。

函数原型

```
~FastMath();
```

5.5.9.4 exp

函数功能

快速幂运算。

函数原型

```
inline float exp(const float x);
```

参数说明

参数名	输入/输出	说明
x	输入	指数。

返回参数说明

数据结构	说明
float	运算结果。

5.5.9.5 sigmoid

函数功能

快速sigmoid函数运算。

函数原型

```
inline float sigmoid(float x);
```

参数说明

参数名	输入/输出	说明
x	输入	指数。

返回参数说明

数据结构	说明
float	运算结果。

5.5.9.6 softmax

函数功能

快速softmax函数运算。

函数原型

```
inline void softmax(std::vector<float>& digits);
```

参数说明

参数名	输入/输出	说明
digits	输入/输出	float型的vector，原位进行softmax。

5.5.9.7 sign

函数功能

符号函数，根据输入数字的符号返回1或-1。

函数原型

```
inline flot sign(float x);
```

参数说明

参数名	输入/输出	说明
x	输入	待判断符号的数字。

5.5.9.8 FExp

函数功能

快速幂运算，单例模式。

函数原型

```
inline float FastMath::FExp(const flaot x);
```

参数说明

参数名	输入/输出	说明
x	输入	指数。

返回参数说明

数据结构	说明
float	运算结果。

5.5.9.9 Sigmoid

函数功能

快速sigmoid函数运算，单例模式。

函数原型

```
inline float FastMath::Sigmod(float x);
```

参数说明

参数名	输入/输出	说明
x	输入	指数。

返回参数说明

数据结构	说明
float	运算结果。

5.5.9.10 Softmax

函数功能

快速归一化指数函数运算，单例模式。

函数原型

```
void FastMath::SoftMax(std::vector<float> &digits);
```

参数说明

参数名	输入/输出	说明
digits	输入/输出	float类型的vector，原位进行softmax。

5.5.9.11 sign

函数功能

快速符号函数运算，单例模式。

函数原型

```
inline float FastMath::sign(float x);
```

参数说明

参数名	输入/输出	说明
x	输入	float型。

返回参数说明

数据结构	运算说明
float	运算结果。

5.5.10 ConfigData

5.5.10.1 类说明

ConfigData接口在ConfigUtil.h中定义，当前为内部使用类，请用户不要使用。

5.5.10.2 ConfigData

函数功能

用于存储配置文件数据。

函数原型

```
ConfigData::ConfigData();
ConfigData::ConfigData(const ConfigData &other);
```

参数说明

参数名	输入/输出	说明
other	输入	复制传入的 ConfigData（指针）。

5.5.10.3 ~ConfigData

函数功能

ConfigData类的默认析构函数。

函数原型

```
~ConfigData();
```

5.5.10.4 operator=

函数功能

复制数据。

函数原型

```
ConfigData &ConfigData::operator = (const ConfigData &other);
```

参数说明

参数名	输入/输出	说明
other	输入	深度复制传入的 ConfigData（非指针）。

5.5.10.5 SetJsonValue

函数功能

根据传入的key赋值对应的value（保存数据）。

函数原型

```
APP_ERROR ConfigData::SetJsonValue(const std::string &key, const std::string &value, int pos = -1);
```

参数说明

参数名	输入/输出	说明
key	输入	关键字。
value	输入	关键字对应的值。
pos	输入	int型。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.6 LoadLabels

函数功能

根据路径读取Labels数据。

函数原型

```
APP_ERROR ConfigData::LoadLabels(const std::string &labelPath);
```

参数说明

参数名	输入/输出	说明
labelPath	输入	Label数据文件的路径。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.7 GetClassName

函数功能

根据类别 ID 获取类别名。

函数原型

```
std::string ConfigData::GetClassName(const size_t classId);
```

参数说明

参数名	输入/输出	说明
classId	输入	类别ID。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.8 GetFileValue

函数功能

根据输入的key获取对应的value。

函数原型

```
template<typename T> APP_ERROR ConfigData::GetFileValue(const std::string &key, T &value);
template<typename T> APP_ERROR ConfigData::GetFileValue(const std::string &key, T &value, const T
&min, const T &max);
```

参数说明

参数名	输入/输出	说明
key	输入	关键字。
value	输出	关键字对应的值。
min	输入	限定的最小值（低于最小值取最小值）。
max	输入	限定的最大值（高于最大值取最大值）。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.9 GetFileValueWarn

函数功能

函数在GetFileValue基础上不返回错误码。

函数原型

```
template<typename T> void ConfigData::GetFileValueWarn(const std::string &key, T &value);
```

```
template<typename T> void ConfigData::GetFileValueWarn(const std::string &key, T &value, const T &min, const T &max);
```

参数说明

参数名	输入/输出	说明
key	输入	关键字。
value	输出	关键字对应的值。
min	输入	限定的最小值（低于最小值取最小值）。
max	输入	限定的最大值（高于最大值取最大值）。

返回参数说明

数据结构	运算说明
void	无返回值。

5.5.10.10 SetFileValue

函数功能

根据传入的key赋值对应的value（保存数据至 File）。

函数原型

```
template<typename T> APP_ERROR ConfigData::SetFileValue(const std::string &key, const T &value);
```

参数说明

参数名	输入/输出	说明
key	输入	关键字。
value	输入	关键字对应的值。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.11 InitFile

函数功能

初始化File。

函数原型

```
APP_ERROR ConfigData::InitFile(std::ifstream &inFile);
```

参数说明

参数名	输入/输出	说明
inFile	输入	读取的文件。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.12 InitJson

函数功能

初始化JSON。

函数原型

```
APP_ERROR ConfigData::InitJson(std::ifstream &inFile);
```

参数说明

参数名	输入/输出	说明
inFile	输入	读取的文件。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.13 InitContent

函数功能

初始化Content。

函数原型

```
APP_ERROR ConfigData::InitContent(const std::string &content);
```

参数说明

参数名	输入/输出	说明
content	输入	string类型，配置信息。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.10.14 GetCfgJson

函数功能

返回JSON数据。

函数原型

```
const web::json::value &ConfigData::GetCfgJson();
```

返回参数说明

数据结构	运算说明
web::json::value	返回JSON数据。

5.5.11 ConfigUtil

5.5.11.1 类说明

ConfigUtil接口在ConfigUtil.h中定义，**当前为内部使用类，请用户不要使用。**

5.5.11.2 LoadConfiguration

函数功能

读取配置文件数据。

函数原型

```
APP_ERROR ConfigUtil::LoadConfiguration(const std::string &config, ConfigData &data, ConfigMode mode = CONFIGJSON);
```

参数说明

参数名	输入/输出	说明
config	输入	配置文件路径。
data	输出	从配置文件中获取参数数据。
mode	输入	文件类型： ● “CONFIGJSON”对应 JSON 文件。 ● “CONFIGFILE”对应 Normal 文件。 ● “CONFIGPM”对应 Pm 文件。 ● “CONFIGCONTENT”对应 Json content。

返回参数说明

数据结构	运算说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12 DeviceManager

5.5.12.1 类说明

DeviceManager接口在DeviceManager.h中定义，当前为内部使用类，请用户不要使用。

5.5.12.2 ~DeviceManager

函数功能

DeviceManager类的默认析构函数。

函数原型

```
~DeviceManager();
```

5.5.12.3 GetInstance

函数功能

单例模式获取设备管理器。

函数原型

```
static DeviceManager *DeviceManager::GetInstance();
```

返回参数说明

数据结构	说明
DeviceManager	设备管理器。

5.5.12.4 InitDevices

函数功能

初始化所有设备。

函数原型

```
APP_ERROR DeviceManager::InitDevices(std::string configFilePath = "");
```

参数说明

参数名	输入/输出	说明
configFilePath	输入	配置文件路径。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.5 GetDevicesCount

函数功能

获取设备数目。

函数原型

```
APP_ERROR DeviceManager::GetDevicesCount(uint32_t& deviceCount);
```

参数说明

参数名	输入/输出	说明
deviceCount	输出	设备数目。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.6 GetCurrentDevice

函数功能

获取最近使用的设备。

函数原型

```
APP_ERROR DeviceManager::GetCurrentDevice(DeviceContext& device);
```

参数说明

参数名	输入/输出	说明
device	输出	获取最近运行的设备。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.7 SetDevice

函数功能

指定待使用的设备。

函数原型

```
APP_ERROR DeviceManager::SetDevice(DeviceContext device);
```

参数说明

参数名	输入/输出	说明
device	输入	待使用的设备。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.8 DestroyDevices

函数功能

释放所有设备资源。

函数原型

```
APP_ERROR DeviceManager::DestroyDevices();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.9 SetDeviceSimple

函数功能

功能等同于SetDevice函数。

函数原型

```
APP_ERROR DeviceManager::SetDeviceSimple(DeviceContext device);
```

参数说明

参数名	输入/输出	说明
device	输入	要使用的设备。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.10 IsInitDevices

函数功能

所有设备是否初始化。

函数原型

```
bool DeviceManager::IsInitDevices() const;
```

参数说明

参数名	输入/输出	说明
device	输入	要使用的设备。

返回参数说明

数据结构	说明
bool	所有设备是否初始化。

5.5.12.11 CheckDeviceId

函数功能

检查设备ID是否合法。

函数原型

```
APP_ERROR DeviceManager::CheckDeviceId(int32_t deviceId);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	设备ID。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.12.12 IsAscend310B

函数功能

判断是否使用Atlas 200/500 A2 推理产品。

函数原型

```
static bool DeviceManager::IsAscend310B();
```

参数说明

参数名	输入/输出	说明
device	输入	要使用的设备。

返回参数说明

数据结构	说明
bool	判断是否使用Atlas 200/500 A2 推理产品。

5.5.12.13 GetSocName

函数功能

查询当前运行环境的芯片版本。

函数原型

```
static std::string GetSocName();
```

返回参数说明

数据结构	说明
string	返回芯片版本信息。 ● 对于Atlas 200/500 A2 推理产品，返回“310B”。

5.5.13 LineRegressionFit

5.5.13.1 类说明

线性拟合类。

5.5.13.2 LineRegressionFit

函数功能

类构造函数，创建线性拟合类对象。

函数原型

```
LineRegressionFit();
```

5.5.13.3 ~LineRegressionFit

函数功能

LineRegressionFit类的默认析构函数。

函数原型

```
~LineRegressionFit();
```

5.5.13.4 LRFunction

函数功能

计算x横坐标值在拟合线性后斜率和截距条件下的纵坐标值。

函数原型

```
double LRFunction(const float &x) const;
```

参数说明

参数名	输入/输出	说明
x	输入	横坐标值。

5.5.13.5 SetAlphaAndBeta

函数功能

计算斜率和截距。

函数原型

```
void SetAlphaAndBeta(const std::vector<float> &xVec, const std::vector<float> &yVec);
```

参数说明

参数名	输入/输出	说明
xVec	输入	横坐标数组。
yVec	输入	纵坐标数组。

5.5.14 NpySort

5.5.14.1 类说明

numpy排序算法C++实现类。

5.5.14.2 NpySort

函数功能

numpy排序算法构造函数。

函数原型

```
explicit NpySort() = default;
NpySort(std::vector<float> preSortVec, std::vector<int> sortIdx);
```

参数说明

参数名	输入/输出	说明
preSortVec	输入	待排序数组，长度大于0。
sortIdx	输入	待排序索引数组。 单个索引最小为0， 最大为该数组长度-1。长度与待排序数组相同。

5.5.14.3 ~NpySort

函数功能

NpySort类的默认析构函数。

函数原型

```
~NpySort();
```

5.5.14.4 NpyArgQuickSort

函数功能

numpy快速排序。

函数原型

```
void NpyArgQuickSort(bool reverse);
```

参数说明

参数名	输入/输出	说明
reverse	输入	是否倒序。

5.5.14.5 NpyArgHeapSort

函数功能

numpy堆排序。

函数原型

```
void NpyArgHeapSort(int tosort, int n);
```

参数说明

参数名	输入/输出	说明
tosort	输入	待排序数组起始索引。
n	输入	待排序的元素个数。

5.5.14.6 GetSortIdx

函数功能

排序后的索引数组。

函数原型

```
std::vector<int> GetSortIdx();
```

参数说明

参数名	输入/输出	说明
tosort	输入	待排序数组起始索引。
n	输入	待排序的元素个数。

5.5.15 TensorBase

5.5.15.1 类说明

该类主要有TensorBase数据结构的构造方法和相关的功能接口。

须知

如出现**因内存耗尽，导致构造函数执行失败抛出**的异常时，请勿继续调用后续的成员函数。

5.5.15.2 CreateTensorBase

函数功能

用来创建TensorBase对象，并根据传入参数不同，调用不同的构造函数。

函数原型

```
template<typename... Param>
static APP_ERROR CreateTensorBase(TensorBase &tensor, Param... params);
```

参数说明

参数名	输入/输出	说明
tensor	输出	TensorBase对象，用于接收构造的TensorBase对象。
params	输入	用于构造TensorBase的参数，具体参数可参考构造函数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.3 operator=

函数功能

等号运算符重载，用于TensorBase对象之间的赋值。

函数原型

```
TensorBase& operator = (const TensorBase& other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	TensorBase对象。

5.5.15.4 GetIndices

函数功能

将一个新的元素加到向量的最后面。

函数原型

```
void GetIndices (std::vector<T>& indices, U value);
void GetIndices (std::vector<T>& indices, U value, Ix ... idxs);
void GetIndices (std::vector<T>& indices, std::vector<U > values);
```

参数说明

参数名	输入/输出	说明
indices	输入/输出	输入的向量。
value	输入	被添加到向量的值。
idxs	输入	被添加到向量的值。

5.5.15.5 GetTensorDataTypeDesc

函数功能

获取张量的数据类型。

函数原型

```
std::string GetTensorDataTypeDesc (TensorDataType type);
```

参数说明

参数名	输入/输出	说明
type	输入	枚举值，范围在[-1, 12]，参考**5.4.1.61 TensorDataType**。

返回参数说明

数据结构	说明
std::string	返回的张量的数据。

5.5.15.6 TensorBase

函数功能

TensorBase的构造函数，用来创建TensorBase对象，根据传入的不同参数可选用不同的构造函数。

须知

如出现**因内存耗尽，导致构造函数执行失败抛出**的异常时，请勿继续调用后续的成员函数。

函数原型

```
TensorBase();
TensorBase(const MemoryData &memoryData, const bool &isBorrowed, const std::vector<uint32_t> &shape,
const TensorDataType &type);
TensorBase(const std::vector<uint32_t> &shape, const TensorDataType &type, const
MemoryData::MemoryType &bufferType, const int32_t &deviceId);
TensorBase(const std::vector<uint32_t> &shape, const TensorDataType &type, const int32_t &deviceId);
TensorBase(const std::vector<uint32_t> &shape, const TensorDataType &type);
TensorBase(const std::vector<uint32_t> &shape);
TensorBase(const TensorBase& tensor) = default;
```

参数说明

说明

- 函数原型2、3、4、5只是预先设置了shape，但不会申请内存空间，需要调用5.5.15.8 TensorBaseMalloc才能申请对应内存空间。
- 函数原型1使用场景：用户已在外部申请内存空间，构造Tensor对象的时候直接引用已申请的内存，需确保内存空间大小与Tensor对象的shape一致，同时可以选择外部申请的空间是Tensor对象内部自行释放还是用户外部释放。

参数名	输入/输出	说明
memoryData	输入	用于构造TensorBase对象的参数，内存管理结构体，具体请参见5.4.1.28 MemoryData。
isBorrowed	输入	表示传入的MemoryData数据是否要Tensor主动释放。 ● 若为“true”，表示不需要主动释放，用户自行释放。 ● 若为“false”，用户不需要手动释放，Tensor析构时自动释放。
shape	输入	用于构造TensorBase对象的参数，张量的形状。
type	输入	用于构造TensorBase对象的参数，TensorDataType类型数据，具体请参见5.4.1.61 TensorDataType枚举说明。
bufferType	输入	用于构造TensorBase对象的参数，张量数据的内存类型。
deviceId	输入	用于构造TensorBase对象的参数，int类型数据，设备编号。
tensor	输入	TensorBase对象，用于构造TensorBase对象的参数。

5.5.15.7 ~TensorBase

函数功能

TensorBase类的默认析构函数。

函数原型

```
virtual ~TensorBase() = default;
```

5.5.15.8 TensorBaseMalloc

函数功能

用于获取TensorBase对象的内存。

Tensor对象在析构时会自动释放内存，因此无需通过函数释放内存。

函数原型

```
static APP_ERROR TensorBaseMalloc(TensorBase &tensor);
```

参数说明

参数名	输入/输出	说明
tensor	输入	用来获取对应内存的Tensor数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.9 TensorBaseCopy

函数功能

内存复制函数，根据MemoryData中指定的内存位置在Host侧和Device侧之间进行复制。

函数原型

```
static APP_ERROR TensorBaseCopy(TensorBase &dst, const TensorBase &src);
```

参数说明

参数名	输入/输出	说明
dst	输出	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.10 GetTensorType

函数功能

用于获取Tensor的内存类型。

函数原型

```
MemoryData::MemoryType GetTensorType() const;
```

返回参数说明

数据结构	说明
MemoryType	申请的内存类型： ● MEMORY_HOST：对应Host侧。 ● MEMORY_DEVICE：对应Device侧。 ● MEMORY_DVPP：对应DVPP侧。 ● MEMORY_HOST_MALLOC：对应malloc申请内存。 ● MEMORY_HOST_NEW：对应new申请内存。

5.5.15.11 GetSize

函数功能

用于获取张量数据对应内存大小。

size大小应与实际内存大小一致，否则可能会导致程序出现coredump情况。

函数原型

```
size_t GetSize() const;
```

返回参数说明

数据结构	说明
size_t	返回具体数值。

5.5.15.12 GetByteSize

函数功能

用于获取buffer字节数据量。

函数原型

```
size_t GetByteSize() const;
```

返回参数说明

数据结构	说明
size_t	返回具体数值。

5.5.15.13 SetShape

函数功能

设置TensorBase的形状。

函数原型

```
APP_ERROR SetShape(std::vector<uint32_t> shape);
```

参数说明

参数名	输入/输出	说明
shape	输入	shape向量中各维度要求为正整数且单个或各项乘积需小于536,870,912（512 * 1024 * 1024），否则函数将抛出异常。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.14 GetShape

函数功能

用于获取Tensor的形状。

函数原型

```
std::vector<uint32_t> GetShape() const;
```

返回参数说明

数据结构	说明
std::vector<uint32_t>	shape数组。

5.5.15.15 GetStrides

函数功能

用于获取Tensor的步长。

函数原型

```
std::vector<uint32_t> GetStrides() const;
```

返回参数说明

参数名	说明
std::vector<uint32_t>	Tensor的步长数组。

5.5.15.16 GetDeviceId

函数功能

用于获取Tensor的设备号。

函数原型

```
int32_t GetDeviceId() const;
```

返回参数说明

数据结构	说明
tensor	返回设备号，是一个int类型。

5.5.15.17 GetDataType

函数功能

用于获取内存中张量的数据类型。

函数原型

```
TensorDataType GetDataType() const;
```

返回参数说明

数据结构	说明
TensorDataType	请参考**5.4.1.61 TensorDataType**枚举说明。

5.5.15.18 GetDataTypeSize

函数功能

用于获取Tensor的数据类型。根据TensorDataType的具体类型返回该类型字节长度。

函数原型

```
uint32_t GetDataTypeSize() const;
```

返回参数说明

数据结构	说明
uint32_t	Tensor的数据类型对应的字节长度。

5.5.15.19 IsHost

函数功能

用于判断Tensor对象是否在Host侧。

函数原型

```
bool IsHost() const;
```

返回参数说明

数据结构	说明
bool	● true：在Host侧。 ● false：不在Host侧。

5.5.15.20 IsDevice

函数功能

用于判断Tensor对象是否在Device侧。

函数原型

```
bool IsDevice() const;
```

返回参数说明

数据结构	说明
bool	• true：在Device侧。 • false：不在Device侧。

5.5.15.21 GetBuffer

函数功能

用于获取Tensor指针。

函数原型 1

```
void* GetBuffer() const;
```

返回参数说明 1

数据结构	说明
void*	指针的首地址，即buffer指针。

函数原型 2

```
APP_ERROR GetBuffer(void *&ptr, const std::vector<uint32_t> &indices) const;
```

参数说明 2

参数名	输入/输出	说明
ptr	输出	获取到的buffer指针。
indices	输入	传入的索引列表，待复制的源内存。

返回参数说明 2

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

函数原型 3

```
template<typename T, typename... Ix>
APP_ERROR GetBuffer(T* &value, Ix... index) const;
```

参数说明 3

参数名	输入/输出	说明
value	输出	获取到的buffer。
index	输入	传入的索引值。

返回参数说明 3

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.22 GetValue

函数功能

获取Host侧的Tensor的buffer，首先判断Tensor是否在Host侧，然后在其中调用GetBuffer函数实现获取buffer的功能。

函数原型

```
template<typename T, typename... Ix>
APP_ERROR GetValue(T &value, Ix... index) const;
```

参数说明

参数名	输入/输出	说明
value	输出	获取到对应Tensor的buffer。
index	输入	待获取Tensor的索引。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.23 ToDevice

函数功能

将Tensor对象部署到Device侧内存。

函数原型

```
APP_ERROR ToDevice(int32_t deviceId);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	设备号。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.24 ToDvpp

函数功能

将Tensor对象部署到DVPP内存中。

函数原型

```
APP_ERROR ToDvpp(int32_t deviceId);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	设备号。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.25 ToHost

函数功能

将Tensor对象部署到Host侧内存。

函数原型

```
APP_ERROR ToHost();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.26 BatchConcat

函数功能

将多batch的Tensor组合成一个Tensor。

函数原型

```
static APP_ERROR BatchConcat(const std::vector<TensorBase> &inputs, TensorBase &output);
```

参数说明

参数名	输入/输出	说明
inputs	输入	Tensor列表，列表大小不超过1024。
output	输出	组合后的Tensor对象。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.27 BatchStack

函数功能

将多batch的Tensor组合成一个Tensor，相比于BatchConcat拓展了Tensor的维度。

函数原型

```
static APP_ERROR BatchStack(const std::vector<TensorBase> &inputs, TensorBase &output);
```

参数说明

参数名	输入/输出	说明
inputs	输入	Tensor列表，列表大小不超过1024。
output	输出	组合后的Tensor对象。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.28 BatchVector

函数功能

合并batch组合成一个Tensor。该函数实际调用BatchConcat和BatchStack，实现组batch操作，通过keepDims参数控制具体调用函数。

函数原型

```
static APP_ERROR BatchVector(const std::vector<TensorBase> &inputs, TensorBase &output, const bool &keepDims = false);
```

参数说明

参数名	输入/输出	说明
inputs	输入	多batch模型，将TensorBase对象列表作为输出。列表大小不超过1024。
output	输出	组batch后输出一个TensorBase对象。
keepDims	输入	保持其多维特性的参数。 ● “true”即**保持**其维度特性函数内部调用函数BatchConcat实现。 ● “false”即**不保持**维度特性函数内部调用函数BatchStack实现。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.29 GetDesc

函数功能

用于获取详细信息。

函数原型

```
std::string GetDesc();
```

返回参数说明

数据结构	说明
std::string	返回详情信息。

5.5.15.30 CheckTensorValid

函数功能

用于查看Tensor对象是否合法。

函数原型

```
APP_ERROR CheckTensorValid() const;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.15.31 SetValidRoi

函数功能

支持设置TensorBase的有效区域，仅支持NHWC（N=1）、HWC与HW维度的TensorBase对象设置, 有效区域的宽高不能超过原始图片。

函数原型

```
APP_ERROR SetValidRoi(Rect rect);
```

参数说明

参数名	输入/输出	说明
rect	输入	表示TensorBase对象的数据有效区域，数据类型请参见5.4.1.6 Rect。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.5.15.32 GetValidRoi

函数功能

支持查询TensorBase的有效数据区域。

函数原型

```
Rect GetValidRoi() const;
```

返回参数说明

数据结构	说明
Rect	返回TensorBase对象的有效数据区域，数据类型请参见5.4.1.6 Rect。

5.5.16 TensorBuffer

5.5.16.1 类说明

该类主要有TensorBuffer数据结构的构造方法和相关的功能接口。

5.5.16.2 TensorBuffer

函数功能

TensorBuffer的构造函数，用来创建TensorBuffer对象，根据传入的不同参数可选用不同的构造函数。

函数原型

```
TensorBuffer();
```

```
TensorBuffer(uint32_t size, MemoryData::MemoryType type, int32_t deviceId);
TensorBuffer(uint32_t size, int32_t deviceId);
TensorBuffer(uint32_t size);
TensorBuffer(void *ptr, uint32_t size, MemoryData::MemoryType type =
MemoryData::MemoryType::MEMORY_HOST_NEW, int32_t deviceId = -1);
```

参数说明

参数名	输入/输出	说明
ptr	输入	内存指针。
size	输入	内存大小。
type	输入	用于构造TensorBuffer对象的内存类型。
deviceId	输入	用于构造TensorBuffer对象的参数，int类型数据，设备编号。

5.5.16.3 ~TensorBuffer

函数功能

TensorBuffer类的默认析构函数。

函数原型

```
~TensorBuffer() {};
```

5.5.16.4 IsDevice

函数功能

用于判断TensorBuffer对象是否在Device侧。

函数原型

```
bool IsDevice() const;
```

返回参数说明

数据结构	说明
bool	● true：在Device侧。 ● false：不在Device侧。

5.5.16.5 IsHost

函数功能

用于判断TensorBuffer对象是否在Host侧。

函数原型

```
bool IsHost() const;
```

返回参数说明

数据结构	说明
bool	● true：在Host侧。 ● false：不在Host侧。

5.5.16.6 SetContext

函数功能

根据TensorBuffer对象中的deviceId，设置线程的Context。

函数原型

```
APP_ERROR SetContext() const;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.7 TensorBufferMalloc

函数功能

用于获取TensorBuffer对象的内存。

Tensor对象在析构时会自动释放内存，因此无需通过函数释放内存。

函数原型

```
static APP_ERROR TensorBufferMalloc(TensorBuffer &buffer);
```

参数说明

参数名	输入/输出	说明
buffer	输入	用来获取对应内存的TensorBuffer数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.8 TensorBufferCopy

函数功能

内存复制函数，根据src中指定的内存位置在Host侧和Device侧之间进行复制。

函数原型

```
static APP_ERROR TensorBufferCopy(TensorBuffer &dst, const TensorBuffer &src);
```

参数说明

参数名	输入/输出	说明
dst	输出	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.9 GetBufferCopyType

函数功能

获取两块TensorBuffer内存的数据类型。

函数原型

```
static TensorBufferCopyType GetBufferCopyType(const TensorBuffer &buffer1, const TensorBuffer &buffer2);
```

参数说明

参数名	输入/输出	说明
buffer1	输入	TensorBuffer对象。

参数名	输入/输出	说明
buffer2	输入	TensorBuffer对象。

返回参数说明

数据结构	说明
TensorBufferCopyType	两块内存拷贝类型。 ● HOST_AND_HOST：两块都为Host侧内存。 ● HOST_AND_DEVICE：一块为Host侧内存，另一块为Device侧。 ● DEVICE_AND_SAME_DEVICE：两块都为Device侧内存，且deviceId相同。 ● DEVICE_AND_DIFF_DEVICE：两块都为Device侧内存，且deviceId不同。

5.5.16.10 CheckCopyValid

函数功能

检测两块TensorBuffer对象的内存长度是否相同，符合相互拷贝条件。

函数原型

```
static APP_ERROR CheckCopyValid(const TensorBuffer &buffer1, const TensorBuffer &buffer2);
```

参数说明

参数名	输入/输出	说明
buffer1	输入	TensorBuffer对象。要求保证实际分配的内存大小与size一致。
buffer2	输入	TensorBuffer对象。要求保证实际分配的内存大小与size一致。

返回参数说明

数据结构	说明
TensorBufferCopyType	两块TensorBuffer内存的数据类型。 ● HOST_AND_HOST：两块都为Host侧内存。 ● HOST_AND_DEVICE：一块为Host侧内存，另一块为Device侧。 ● DEVICE_AND_SAME_DEVICE：两块都为Device侧内存，且deviceId相同。 ● DEVICE_AND_DIFF_DEVICE：两块都为Device侧内存，且deviceId不同。

5.5.16.11 CopyBetweenHost

函数功能

将Host侧的src对象拷贝至Host侧的dst对象。

函数原型

```
static APP_ERROR CopyBetweenHost(TensorBuffer &dst, const TensorBuffer &src);
```

参数说明

参数名	输入/输出	说明
dst	输入	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.12 CopyBetweenHostDevice

函数功能

将src对象拷贝至dst对象。两块TensorBuffer内存分别处于Host侧和Device侧。

函数原型

```
static APP_ERROR CopyBetweenHostDevice(TensorBuffer &dst, const TensorBuffer &src);
```

参数说明

参数名	输入/输出	说明
dst	输入	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.13 CopyBetweenSameDevice

函数功能

将Device侧的src对象拷贝至Device侧的dst对象。两块TensorBuffer内存的deviceId相同。

函数原型

```
static APP_ERROR CopyBetweenSameDevice(TensorBuffer &dst, const TensorBuffer &src);
```

参数说明

参数名	输入/输出	说明
dst	输入	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.16.14 CopyBetweenDiffDevice

函数功能

将Device侧的src对象拷贝至Device侧的dst对象。两块TensorBuffer内存的deviceId不同。

函数原型

```
static APP_ERROR CopyBetweenDiffDevice(TensorBuffer &dst, const TensorBuffer &src);
```

参数说明

参数名	输入/输出	说明
dst	输入	复制后的目标内存。
src	输入	待复制的源内存。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.17 TensorContext

5.5.17.1 类说明

该类主要有TensorContext数据结构的构造方法和相关的功能接口。

5.5.17.2 TensorContext

函数功能

TensorContext构造函数，创建TensorContext()对象并初始化AscendCL。

函数原型

```
TensorContext();
```

5.5.17.3 ~TensorContext

函数功能

TensorContext类的默认析构函数。

函数原型

```
~TensorContext();
```

5.5.17.4 GetInstance

函数功能

获取TensorContext单例对象。

函数原型

```
static std::shared_ptr<TensorContext> GetInstance();
```

返回参数说明

数据结构	说明
std::shared_ptr<TensorContext>	TensorContext单例对象。

5.5.17.5 SetContext

函数功能

设置线程的Context。

函数原型

```
APP_ERROR SetContext(const uint32_t &deviceId);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	线程当前的deviceId。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.18 TensorShape

5.5.18.1 类说明

该类主要有TensorShape数据结构的构造方法和相关的功能接口。

5.5.18.2 TensorShape

函数功能

TensorShape构造函数，创建TensorShape()对象。

函数原型

```
TensorShape() = default;
TensorShape (std::vector<T> shape);
```

参数说明

参数名	输入/输出	说明
shape	输入	Tensor的形状。 shape向量中各维度要求为正整数且单个或各项乘积需要小于52,428,8000（500MB），否则函数将抛出异常。

5.5.18.3 ~TensorShape

函数功能

TensorShape类的默认析构函数。

函数原型

```
~TensorShape() = default;
```

5.5.18.4 GetDims

函数功能

获取Tensor形状的维度。

函数原型

```
uint32_t GetDims() const;
```

返回参数说明

数据结构	说明
uint32_t	Tensor形状的维度值。

5.5.18.5 SetShape

函数功能

设置Tensor的形状。

函数原型

```
template<typename T>
void SetShape(std::vector<T> shape);
```

参数说明

参数名	输入/输出	说明
shape	输入	Tensor的形状。 shape向量中各维度要求为正整数且单个或各项乘积需小于536,870,912（512 * 1024 * 1024），否则函数将抛出异常。

5.5.18.6 GetShape

函数功能

获取Tensor的具体形状。

函数原型

```
std::vector<uint32_t> GetShape() const;
```

返回参数说明

数据结构	说明
std::vector<uint32_t>	Tensor的形状。

5.5.18.7 GetSize

函数功能

获取Tensor的大小。

函数原型

```
uint32_t GetSize() const;
```

返回参数说明

数据结构	说明
uint32_t	Tensor的大小。

5.5.18.8 operator[]

函数功能

获取Tensor指定索引维度的大小。

函数原型

```
size_t operator[] (uint32_t idx) const;
size_t &operator[] (uint32_t idx);
```

参数说明

参数名	输入/输出	说明
uint32_t	输入	Tensor维度索引值。

返回参数说明

数据结构	说明
size_t	Tensor指定维度的大小。

5.5.18.9 SetValidRoi

函数功能

支持设置TensorShape的validRoi_成员变量，仅支持NHWC（N=1）、HWC与HW维度的TensorShape对象设置validRoi_成员变量, 有效区域的宽高不能超过原始图片。

函数原型

```
APP_ERROR SetValidRoi(Rect rect);
```

参数说明

参数名	输入/输出	说明
rect	输入	输入图像的坐标框，数据类型请参见**5.4.1.6 Rect**。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.18.10 GetValidRoi

函数功能

返回TensorShape的成员变量validRoi_。

函数原型

```
Rect GetValidRoi() const;
```

返回参数说明

数据结构	说明
Rect	返回TensorShape的成员变量validRoi_，数据类型请参见**5.4.1.6 Rect**。

5.5.19 Version

5.5.19.1 功能说明

功能

获取mxVision的版本号。

参数说明

参数	说明
MINDX_SDK_VERSION	获取mxVision的版本号（uint64型）。 调用后处理插件的**5.10.1.2.6 GetCurrentVersion**接口获得该插件的版本号。
MINDX_SDK_MAJOR_VERSION	获取mxVision的大版本号（uint32型）。
MINDX_SDK_MINOR_VERSION	获取mxVision的中版本号（uint32型）。
MINDX_SDK_MICRO_VERSION	获取mxVision的小版本号（uint32型）。

5.5.19.2 GetCurrentSDKVersion

说明

此接口将在后续版本删除，请用户不要使用。

函数功能

获取mxVision的版本号（uint64型）。

函数原型

```
uint64_t GetCurrentSDKVersion();
```

返回参数说明

数据结构	说明
uint64_t	返回mxVision的版本号（uint64型）。

5.5.19.3 GetCurrentMajorVersion

说明

此接口将在后续版本删除，请用户不要使用。

函数功能

获取mxVision的大版本号（uint32型）。

函数原型

```
const uint32_t GetCurrentMajorVersion();
```

返回参数说明

数据结构	说明
uint32_t	返回mxVision的大版本号（uint32型）。

5.5.19.4 GetCurrentMinorVersion

说明

此接口将在后续版本删除，请用户不要使用。

函数功能

获取mxVision的中版本号（uint32型）。

函数原型

```
const uint32_t GetCurrentMinorVersion();
```

返回参数说明

数据结构	说明
uint32_t	返回mxVision的中版本号（uint32型）。

5.5.19.5 GetCurrentMicroVersion

说明

此接口将在后续版本删除，请用户不要使用。

函数功能

获取mxVision的小版本号（uint32型）。

函数原型

```
const uint32_t GetCurrentMicroVersion();
```

返回参数说明

数据结构	说明
uint32_t	返回mxVision的小版本号（uint32型）。

5.5.19.6 GetSDKVersion

函数功能

获取mxVision的版本号。

函数原型

```
std::string GetSDKVersion();
```

返回参数说明

数据结构	说明
std::string	返回mxVision的版本号，字符串类型保存。

5.5.20 Log

5.5.20.1 类说明

Log接口在“Log.h”中定义，**当前为内部使用接口，请用户不要使用**。

说明

在多进程场景中，单个进程的日志转储个数默认为50，总的转储个数限制为1000个。在进程数超过1000时，转储的日志会有丢失。

日志级别枚举

```
enum LogLevels {
    LOG_LEVEL_DEBUG = -1,
    LOG_LEVEL_INFO = 0,
    LOG_LEVEL_WARN = 1,
    LOG_LEVEL_ERROR = 2,
    LOG_LEVEL_FATAL = 3,
    LOG_LEVEL_NONE
};
```

以下宏定义可以用于打印错误日志：

表 5-2 宏定义表格

宏定义	说明
#define LogDebug VLOG(MxBase::LOG_LEVEL_DEBUG)	LogDebug打印调试信息。
#define LogInfo LOG(INFO)	LogInfo打印提示信息，打印关键步骤（例如初始化）的信息，与数据无关。
#define LogWarn LOG(WARNING)	LogWarn打印警告信息，打印处理数据时的告警信息。
#define LogError LOG(ERROR)	LogError打印错误信息，打印处理数据时的错误信息。
#define LogFatal LOG(FATAL)	LogFatal打印致命信息，打印关键步骤（例如初始化）的错误信息。

参考样例

```
LogDebug << "Begin to process MpDataSerialize(" << elementName_ << ").";
LogInfo << "Begin to initialize MpDataSerialize(" << elementName_ << ").";
LogWarn << "Input data is invalid, element (" << elementName_ <<") will not run normally.";
LogError << GetError(ret, elementName_) << "Fail to initialize dvppWrapper_ object.";
LogFatal << GetError(ret, elementName_) << "Invalid transfer mode.";
```

5.5.20.2 getLogger

函数功能

根据LoggerName获取Log，若不存在，则创建。

函数原型

```
static Log & Log::getLogger(const std::string loggerName = DEFAULT_LOGGER);
```

参数说明

参数名	输入/输出	说明
loggerName	输入	Log的名字。

返回参数说明

数据结构	说明
Log	返回Log。

5.5.20.3 Init

函数功能

初始化Log。

函数原型

```
static APP_ERROR Log::Init();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.20.4 InitWithoutCfg

函数功能

初始化Log时，不读取配置文件。

函数原型

```
static APP_ERROR Log::InitWithoutCfg();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.20.5 Deinit

函数功能

对已初始化的Log进行去初始化处理。

函数原型

```
static APP_ERROR Log::Deinit();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.5.20.6 Debug

函数功能

输出调试类信息。

函数原型

```
void Log::Debug(const std::string& file, const std::string& function, const int& line, const std::string& msg);
```

参数说明

参数名	输入/输出	说明
file	输入	信息对应的文件。
function	输入	信息对应的函数。
line	输入	信息所在的行数。
msg	输入	信息内容。

5.5.20.7 Info

函数功能

输出消息类信息。

函数原型

```
void Log::Info (const std::string& file, const std::string& function, const int& line, const std::string& msg);
```

参数说明

参数名	输入/输出	说明
file	输入	信息对应的文件。
function	输入	信息对应的函数。
line	输入	信息所在的行数。
msg	输入	信息内容。

5.5.20.8 Warn

函数功能

输出警告类信息。

函数原型

```
void Log::Warn (const std::string& file, const std::string& function, const int& line, const std::string& msg);
```

参数说明

参数名	输入/输出	说明
file	输入	信息对应的文件。
function	输入	信息对应的函数。
line	输入	信息所在的行数。
msg	输入	信息内容。

5.5.20.9 Error

函数功能

输出错误类信息。

函数原型

```
void Log::Error (const std::string& file, const std::string& function, const int& line, const std::string& msg);
```

参数说明

参数名	输入/输出	说明
file	输入	信息对应的文件。
function	输入	信息对应的函数。

参数名	输入/输出	说明
line	输入	信息所在的行数。
msg	输入	信息内容。

5.5.20.10 Fatal

须知

在输出Fatal级别的日志后，会终止程序的运行，请谨慎使用。

函数功能

输出致命类信息。

函数原型

```
void Log::Fatal(const std::string& file, const std::string& function, const int& line, const std::string& msg);
```

参数说明

参数名	输入/输出	说明
file	输入	信息对应的文件。
function	输入	信息对应的函数。
line	输入	信息所在的行数。
msg	输入	信息内容。

5.5.20.11 Flush

函数功能

清空显示的Log信息。

函数原型

```
void Log::Flush();
```

5.5.20.12 SetLogParameters

函数功能

设置日志配置数据。

函数原型

```
static void Log::SetLogParameters(const ConfigData& configData);
```

参数说明

参数名	输入/输出	说明
configData	输出	配置数据。

5.5.20.13 LogRotateByTime

函数功能

记录一定时间内的数据。

函数原型

```
static void Log::LogRotateByTime(int rotateDay);
```

参数说明

参数名	输入/输出	说明
rotateDay	输入	设置记录文件时长。

5.5.20.14 LogRotateByNumbers

函数功能

记录一定个数的数据。

函数原型

```
static void Log::LogRotateByNumbers(int fileNumbers);
```

参数说明

参数名	输入/输出	说明
fileNumbers	输入	设置记录文件数量，取值范围为大于0的整数。

5.5.20.15 UpdateFileMode

函数功能

更新文件的模式。

函数原型

```
static void Log::UpdateFileMode();
```

5.5.21 ErrorCode

5.5.21.1 类说明

根据错误码获取错误信息。

5.5.21.2 GetAppErrCodeInfo

函数功能

根据错误码获取对应错误信息。

函数原型

```
std::string GetAppErrCodeInfo(APP_ERROR err);
```

参数说明

参数名	输入/输出	说明
err	输入	错误码。

返回参数说明

数据结构	说明
std::string	错误码对应错误信息。

5.5.21.3 GetError

函数功能

根据错误码获取对应错误信息，支持用户增加模块名称。

该接口后续将废弃，请使用**GetErrorInfo**接口获取对应错误信息。

函数原型

```
std::string GetError(APP_ERROR err, std::string moduleName = "");
```

参数说明

参数名	输入/输出	说明
err	输入	错误码。

参数名	输入/输出	说明
moduleName	输入	错误所在模块名字。

返回参数说明

数据结构	说明
std::string	错误码对应错误信息。

5.5.21.4 ConvertReturnCodeToLocal

函数功能

函数内部使用，将三方库返回错误码转换到本地错误码。

函数原型

```
APP_ERROR ConvertReturnCodeToLocal(ReturnCodeType type, int errCode)
```

参数说明

参数名	输入/输出	说明

type	输入	返回码类型，类型参见如下。
		enumReturnCodeType {
		GST_FLOW_TYPE = 0,
		};

errCode	输入	错误码。

返回参数说明

数据结构	说明
APP_ERROR	本地错误码。

5.5.21.5 GetErrorInfo

函数功能

根据错误码获取对应错误信息。该接口用于替代**GetError**接口，可以提供更准确、具体的错误信息，为推荐使用的错误信息获取接口。

函数原型

```
std::string GetErrorInfo(APP_ERROR err, std::string callingFuncName = "")
```

参数说明

参数名	输入/输出	说明
err	输入	APP_ERROR类型，错误码。
callingFuncName	输入	std::string类型，指定产生错误码的函数名。仅当错误码为CANN错误码时，才需要传入该参数，用于记录产生该错误码的CANN函数名，从而方便后续进行错误定位。

返回参数说明

数据结构	说明
std::string	错误码对应的错误相关信息，便于用户进行错误定位。错误码的说明请参见5.5.21.6 APP_ERROR说明。

5.5.21.6 APP_ERROR 说明

```
using APP_ERROR = int;
```

APP_ERROR类用于表示程序执行的返回码，其包括mxVision返回码及部分CANN返回码；CANN返回码用于表示CANN接口执行的情况，不推荐用户使用，其返回码含义详见《CANN 应用软件开发指南 (C&C++)CANN 应用软件开发指南 (C&C++)》;mxVision返回码用于表示mxVision接口执行的情况，常与GetErrorInfo接口配合使用、用于记录日志。全量APP_ERROR返回码请参考“MxBase/ErrorCode/ErrorCode.h”文件。为了方便用户查阅mxVision返回码，下表展示了mxVision返回码列表。其中，返回信息用于解释返回码的具体含义。

表 5-3 mxVision 返回码列表

返回码	返回码值	返回信息
APP_ERR_OK	0	"Success"
APP_ERR_ACL_FAILURE	-1	"ACL: general failure"
APP_ERR_ACL_ERR_BASE	0	"Success"
APP_ERR_ACL_INVALID_PARAM	1	"ACL: invalid parameter"
APP_ERR_ACL_BAD_ALLOC	2	"ACL: memory allocation fail"
APP_ERR_ACL_RT_FAILURE	3	"ACL: runtime failure"

返回码	返回码值	返回信息
APP_ERR_ACL_GE_FAILURE	4	"ACL: Graph Engine failure"
APP_ERR_ACL_OP_NOT_FOUND	5	"ACL: operator not found"
APP_ERR_ACL_OP_LOAD_FAILED	6	"ACL: fail to load operator"
APP_ERR_ACL_READ_MODEL_FAILURE	7	"ACL: fail to read model"
APP_ERR_ACL_PARSE_MODEL	8	"ACL: parse model failure"
APP_ERR_ACL_MODEL_MISSING_ATTR	9	"ACL: model missing attribute"
APP_ERR_ACL_DESERIALIZE_MODEL	10	"ACL: deserialize model failure"
APP_ERR_ACL_EVENT_NOT_READY	12	"ACL: event not ready"
APP_ERR_ACL_EVENT_COMPLETE	13	"ACL: event complete"
APP_ERR_ACL_UNSUPPORTED_DATA_TYPE	14	"ACL: unsupported data type"
APP_ERR_ACL_REPEAT_INITIALIZE	15	"ACL: repeat initialize"
APP_ERR_ACL_COMPILER_NOT_REGISTERED	16	"ACL: compiler not registered"
APP_ERR_ACL_IO	17	"ACL: IO failed"
APP_ERR_ACL_INVALID_FILE	18	"ACL: invalid file"
APP_ERR_ACL_INVALID_DUMP_CONFIG	19	"ACL: invalid dump config"
APP_ERR_ACL_INVALID_PROFILING_CONFIG	20	"ACL: invalid profiling config"
APP_ERR_ACL_OP_TYPE_NOT_MATCH	21	"ACL: operator type not match"
APP_ERR_ACL_OP_INPUT_NOT_MATCH	22	"ACL: operator input not match"
APP_ERR_ACL_OP_OUTPUT_NOT_MATCH	23	"ACL: operator output not match"

返回码	返回码值	返回信息
APP_ERR_ACL_OP_ATTR_NOT_MATCH	24	"ACL: operator attribute not match"
APP_ERR_ACL_API_NOT_SUPPORT	25	"ACL: API not supported"
APP_ERR_ACL_BAD_COPY	26	"ACL: memory copy fail"
APP_ERR_ACL_BAD_FREE	27	"ACL: memory free fail"
APP_ERR_COMM_BASE	1000	"Success"
APP_ERR_COMM_FAILURE	1001	"General Failed"
APP_ERR_COMM_INNER	1002	"Internal error"
APP_ERR_COMM_INVALID_POINTER	1003	"Invalid Pointer"
APP_ERR_COMM_INVALID_PARAM	1004	"Invalid parameter"
APP_ERR_COMM_UNREALIZED	1005	"Not implemented"
APP_ERR_COMM_OUT_OF_MEM	1006	"Out of memory"
APP_ERR_COMM_ALLOC_MEM	1007	"memory allocation error"
APP_ERR_COMM_FREE_MEM	1008	"free memory error"
APP_ERR_COMM_OUT_OF_RANGE	1009	"out of range"
APP_ERR_COMM_NO_PERMISSION	1010	"NO Permission "
APP_ERR_COMM_TIMEOUT	1011	"Timed out"
APP_ERR_COMM_NOT_INIT	1012	"Not initialized"
APP_ERR_COMM_INIT_FAIL	1013	"initialize failed"
APP_ERR_COMM_INPROGRESS	1014	"Operation now in progress "
APP_ERR_COMM_EXIST	1015	"Object, file or other resource already exist"

返回码	返回码值	返回信息
APP_ERR_COMM_NO_EXIST	1016	"Object, file or other resource doesn't exist"
APP_ERR_COMM_BUSY	1017	"Object, file or other resource is in use"
APP_ERR_COMM_FULL	1018	"No available Device or resource"
APP_ERR_COMM_OPEN_FAIL	1019	"Device, file or resource open failed"
APP_ERR_COMM_READ_FAIL	1020	"Device, file or resource read failed"
APP_ERR_COMM_WRITE_FAIL	1021	"Device, file or resource write failed"
APP_ERR_COMM_DESTORY_FAIL	1022	"Device, file or resource destroy failed"
APP_ERR_COMM_EXIT	1023	"Fail to exit"
APP_ERR_COMM_CONNECTION_CLOSE	1024	"Out of connection, Communication shutdown"
APP_ERR_COMM_CONNECTION_FAILURE	1025	"connection fail"
APP_ERR_COMM_STREAM_INVALID	1026	"ACL stream is null pointer"
APP_ERR_COMM_LOGGING_CONFIG_OPEN_FAIL	1027	"Logging config loading failed"
APP_ERR_COMM_SDK_HOME_NOSET	1028	"SDK_HOME not set"
APP_ERR_COMM_INVALID_PATH	1029	"Invalid path"
APP_ERR_COMM_REPEAT_INITIALIZE	1030	"Repeat initialize"
APP_ERR_DVPP_BASE	2000	"Success"
APP_ERR_DVPP_CROP_FAIL	2001	"DVPP: crop fail"
APP_ERR_DVPP_RESIZE_FAIL	2002	"DVPP: resize fail"
APP_ERR_DVPP_CROP_RESIZE_FAIL	2003	"DVPP: crop and resize fail"
APP_ERR_DVPP_CONVERT_FROMAT_FAIL	2004	"DVPP: convert image format fail"

返回码	返回码值	返回信息
APP_ERR_DVPP_VPC_FAIL	2005	"DVPP: VPC(crop, resize, convert format) fail"
APP_ERR_DVPP_JPEG_DECODE_FAIL	2006	"DVPP: decode jpeg or jpg fail"
APP_ERR_DVPP_JPEG_ENCODE_FAIL	2007	"DVPP: encode jpeg or jpg fail"
APP_ERR_DVPP_PNG_DECODE_FAIL	2008	"DVPP: encode png fail"
APP_ERR_DVPP_H26X_DECODE_FAIL	2009	"DVPP: decode H264 or H265 fail"
APP_ERR_DVPP_H26X_ENCODE_FAIL	2010	"DVPP: encode H264 or H265 fail"
APP_ERR_DVPP_HANDLE_NULL	2011	"DVPP: acldvppChannelDesc is nullptr"
APP_ERR_DVPP_PICDESC_FAIL	2012	"DVPP: fail to create or set acldvppCreatePicDesc"
APP_ERR_DVPP_CONFIG_FAIL	2013	"DVPP: fail to set dvpp configuration"
APP_ERR_DVPP_OBJ_FUNC_MISMATCH	2014	"DVPP: DvppCommon object mismatch the function"
APP_ERR_DEVICE_ID_MISMATCH	2015	"DVPP: DvppCommon object mismatch the function"
APP_ERR_MEMEROY_TYPE_MISMATCH	2016	"DVPP: mismatch the memory type"
APP_ERR_METADATA_IS_NULL	2017	"DVPP: metadata is null"
APP_ERR_PROTOBUF_NAME_MISMATCH	2018	"DVPP: VpcReSize mismatch the protobuf name"
APP_ERR_DVPP_INVALID_FORMAT	2019	"DVPP: mismatch the image format"
APP_ERR_DVPP_INVALID_IMAGE_WIDTH	2020	"DVPP: image width out of range"
APP_ERR_DVPP_INVALID_IMAGE_HEIGHT	2021	"DVPP: image height out of range"
APP_ERR_VDEC_CHANNEL_NUM_OUT_OF_RANGE	2022	"DVPP: vdec channel num out of range"

返回码	返回码值	返回信息
APP_ERR_DVPP_CROP_RESIZE_PASTE_FAIL	2023	"DVPP: resize and paste fail"
APP_ERR_INFER_BASE	3000	"Success"
APP_ERR_INFER_SET_INPUT_FAIL	3001	"Infer: set input fail"
APP_ERR_INFER_SET_OUTPUT_FAIL	3002	"Infer: set output fail"
APP_ERR_INFER_CREATE_OUTPUT_FAIL	3003	"Infer: create output fail"
APP_ERR_INFER_OP_SET_ATTR_FAIL	3004	"Infer: set op attribute fail"
APP_ERR_INFER_GET_OUTPUT_FAIL	3005	"Infer: get model output fail"
APP_ERR_INFER_FIND_MODEL_ID_FAIL	3006	"Infer: find model id fail"
APP_ERR_INFER_FIND_MODEL_DESC_FAIL	3007	"Infer: find model description fail"
APP_ERR_INFER_FIND_MODEL_MEM_FAIL	3008	"Infer: find model memory fail"
APP_ERR_INFER_FIND_MODEL_WEIGHT_FAIL	3009	"Infer: find model weight fail"
APP_ERR_INFER_DYNAMIC_IMAGE_SIZE_FAIL	3010	"Infer: In DYNAMIC_HW mode, only batchSize=1 is supported."
APP_ERR_COMMANDER_BASE	4000	"Success"
APP_ERR_COMMANDER_SPLIT_PARA_ERROR	4001	"Commander Splitter parameter error"
APP_ERR_COMMANDER_SPLIT_CONVERT_ERROR	4002	"Commander Splitter coordinate conversion error"
APP_ERR_COMMANDER_NO_AVAIL_SERVER_ERROR	4003	"Commander Splitter server is not available"
APP_ERR_COMMANDER_INFER_RESULT_ERROR	4004	"Commander subtask inference result error."
APP_ERR_QUEUE_BASE	5000	"Success"
APP_ERR_QUEUE_EMPTY	5001	"empty queue"

返回码	返回码值	返回信息
APP_ERR_QUEUE_STOPED	5002	"queue stopped"
APP_ERR_QUEUE_FULL	5003	"full queue"
APP_ERR_STREAM_BASE	6000	"Success"
APP_ERR_STREAM_EXIST	6001	"stream is exist"
APP_ERR_STREAM_NOT_EXIST	6002	"stream is not exist"
APP_ERR_STREAM_CHANGE_STATE_FAILED	6003	"stream change state fail"
APP_ERR_STREAM_CREATE_FAILED	6004	"stream create fail"
APP_ERR_STREAM_INVALID_CONFIG	6005	"stream invalid config"
APP_ERR_STREAM_INVALID_LINK	6006	"stream invalid link"
APP_ERR_STREAM_LINK_FAILED	6007	"stream link fail"
APP_ERR_STREAM_TRANS_MODE_NOT_MATCHED	6008	"stream trans mode not match"
APP_ERR_STREAM_TRANS_MODE_INVALID	6009	"stream trans mode invalid"
APP_ERR_STREAM_TIMEOUT	6010	"stream timeout"
APP_ERR_STREAM_ELEMENT_INVALID	6011	"stream element invalid"
APP_ERR_STREAM_ELEMENT_EXIST	6012	"stream element exist"
APP_ERR_STREAM_ELEMENT_NOT_EXIST	6013	"stream element not exist"
APP_ERR_ELEMENT_INVALID_FACTORY	6014	"element invalid factory"
APP_ERR_ELEMENT_INVALID_PROPERTIES	6015	"element invalid properties"
APP_ERR_ELEMENT_PAD_UNLINKED	6016	"element pad unlinked"

返回码	返回码值	返回信息
APP_ERR_PIPELINE_PROPERTY_CONFIG_ERROR	6017	"pipeline property config error"
APP_ERR_PLUGIN_TOOLKIT_BASE	7000	"Success"
APP_ERR_PLUGIN_TOOLKIT_CREATE_NODE_FAILED	7001	"plugin toolkit create node fail"
APP_ERR_PLUGIN_TOOLKIT_NODE_ALREADY_EXIST	7002	"plugin toolkit node already exist"
APP_ERR_PLUGIN_TOOLKIT_MESSAGE_NOT_MATCH	7003	"plugin toolkit message not match"
APP_ERR_PLUGIN_TOOLKIT_PARENT_NOT_MATCH	7004	"plugin toolkit parent not match"
APP_ERR_PLUGIN_TOOLKIT_NOT_INITIALIZED	7005	"plugin toolkit not initialized"
APP_ERR_PLUGIN_TOOLKIT_NODELIST_NOT_EXIST	7006	"plugin toolkit nodelist not exist"
APP_ERR_PLUGIN_TOOLKIT_NODE_NOT_EXIST	7007	"plugin toolkit node not exist"
APP_ERR_PLUGIN_TOOLKIT_INVALID_MEMBERID	7008	"plugin toolkit invalid memberid"
APP_ERR_PLUGIN_TOOLKIT_METADATA_BUFFER_IS_NULL	7009	"plugin toolkit metadata buffer is null"
APP_ERR_PLUGIN_TOOLKIT_METADATA_KEY_ALREADY_EXIST	7010	"plugin toolkit metadata key already exist"
APP_ERR_PLUGIN_TOOLKIT_METADATA_KEY_NOEXIST	7011	"plugin toolkit metadata key not exist"
APP_ERR_PLUGIN_TOOLKIT_METADATA_KEY_ERASE_FAIL	7012	"plugin toolkit metadata key erase fail"
APP_ERR_PLUGIN_TOOLKIT_METADATA_IS_NULL	7013	"plugin toolkit metadata is null"

返回码	返回码值	返回信息
APP_ERR_PLUGIN_TOOLKIT_METADATA_ADD_ERROR_INFO_FAIL	7014	"plugin toolkit metadata add error info fail"
APP_ERR_PLUGIN_TOOLKIT_METADATA_PLUGIN_NAME_KEY_ALREADY_EXIST	7015	"plugin toolkit metadata plugin name key is exist"
APP_ERR_PLUGIN_TOOLKIT_METADATA_ERROR_INFO_MAP_IS_NULL	7016	"plugin toolkit metadata error info map is null"
APP_ERR_PLUGIN_TOOLKIT_MESSAGE_TO_STRING_FAILED	7017	"plugin toolkit metadata failed to convert message to string"
APP_ERR_PLUGIN_TOOLKIT_FLOW_NOT_LINKED	7018	"plugin toolkit pad is not linked"
APP_ERR_PLUGIN_TOOLKIT_FLOW_FLUSHING	7019	"plugin toolkit pad is flushing"
APP_ERR_PLUGIN_TOOLKIT_FLOW_EOS	7020	"plugin toolkit pad is EOS"
APP_ERR_PLUGIN_TOOLKIT_FLOW_NOT_NEGOTIATED	7021	"plugin toolkit pad is not negotiated"
APP_ERR_PLUGIN_TOOLKIT_FLOW_ERROR	7022	"plugin toolkit some error occurred.Element generating this error should post an error message"
APP_ERR_PLUGIN_TOOLKIT_FLOW_NOT_SUPPORTED	7023	"plugin toolkit operation is not supported"
APP_ERR_PLUGIN_TOOLKIT_METADATA_KEY_EMPTY	7024	"the key cannot be empty"
APP_ERR_PLUGIN_TOOLKIT_METADATA_PLUGIN NAME_EMPTY	7025	"the plugin name cannot be empty"
APP_ERR_DEVICE_MANAGER_BASE	9000	"Success"
APP_ERR_DEVICE_MANAGER_QUERY_DEVICE_ERROR	9001	"DeviceManager:query device count error"

返回码	返回码值	返回信息
APP_ERR_DEVICE_MANAGER_DESTROY_DEVICE_CHECK_ERROR	9002	"DeviceManager:all devices have been released,init or release again fail"
APP_ERR_OP_BASE	10000	"Success"
APP_ERR_OP_CREATE_TENSOR_FAIL	10001	"Fail to create tensor in operator"
APP_ERR_OP_BAD_FREE	10002	"Bad free in operator"
APP_ERR_OP_EXECUTE_FAIL	10003	"Fail to execute operator"
APP_ERR_OP_SYNACHRONIZE_STREAM_FAIL	10004	"Fail to synchronize stream in operator"
APP_ERR_SIZE_NOT_MATCH	10005	"Mismatched size in operator"
APP_ERR_MXPLUGINS_BASE	11000	"Success"
APP_ERR_MXPLUGINS_DEVICE_ID_MISMATCH	11001	"MxPlugins: mismatch the device id"
APP_ERR_MXPLUGINS_MEMEROY_TYPE_MISMATCH	11002	"MxPlugins: mismatch the memory type"
APP_ERR_MXPLUGINS_METADATA_IS_NULL	11003	"MxPlugins: metadata is null"
APP_ERR_MXPLUGINS_PROTOBUF_NAME_MISMATCH	11004	"MxPlugins: protobuf descriptor name mismatch"

5.6 初始化和去初始化

5.6.1 MxInit

函数功能

全局初始化，申请Device资源与日志资源，mxVision接口全部执行完毕后，用户需调用**MxDeInit()**接口对初始化的全局资源进行去初始化操作。

- 对于Atlas 200/500 A2 推理产品：使用MxInit()。

函数原型

```
APP_ERROR MxInit()
```

参数说明

参数名	输入/输出	说明
globalCfg	输入	**AppGlobalCfg**类型，应用全局配置。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.6.2 MxDeInit

函数功能

全局去初始化，释放Device等相关资源，需与全局初始化接口（**MxInit**）配套使用。

说明

涉及申请Device侧资源的类（**Tensor**、**Model**、**Image**、**ImageProcessor**、**VideoEncoder**和**VideoDecoder**）与MxDeInit的作用域冲突，因此，这些类的作用域不能大于或等于MxDeInit的作用域。

函数原型

```
APP_ERROR MxDeInit();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7 媒体数据处理

5.7.1 Image

5.7.1.1 类说明

Image数据类，作为图像处理（包含图片编解码）的输入与输出的数据结构。默认用户构造输入数据在Host侧，数据在设备间的转移由ImageProcessor类来进行管理，用户无需进行内存数据的转移操作。图像类的校验会在ImageProcessor类进行。

说明

Image类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

5.7.1.2 Image

函数功能

Image类的构造函数，可支持以下方式进行创建。若因内存不足，或无法识别芯片等构造失败的场景，则会抛出异常。

- 创建空Image对象。
- 创建自定义内存数据的图片，图片格式类型默认为ImageFormat::YUV_SP_420，设备ID默认为Host侧（deviceId = -1）。
- 创建带对齐高、宽和有效高、宽，自定义内存数据的图片，使用时需满足以下条件。
  - “deviceId”（设备ID）需为有效值，取值范围为[-1, *识别到的Device数*-1]，否则接口会调用失败。
  - “format”支持以下图片格式类型。
    ```
    YUV_400 = 0,
    RGB_888 = 12,
    BGR_888 = 13,
    ARGB_8888 = 14,
    ABGR_8888 = 15,
    RGBA_8888 = 16,
    BGRA_8888 = 17,
    ```
  - “dataSize”需要与**有效宽高**或**对齐后宽高**一致，计算公式可参考：**dataSize = 宽 * 高 * 通道数**。

说明

用户申请Host内存后，如需在Device侧构造Image对象，请参考如下操作进行。

1. 在Host侧构造Image对象，“deviceId”为“-1”（与“imageData”内存位置一致，设置为Host侧）。
2. 构造后的Image对象使用**ToDevice(deviceId)**方法，将内存迁移至Device侧。

函数原型

```
Image();
Image(const std::shared_ptr<uint8_t> imageData, const uint32_t dataSize, const int32_t deviceId = -1, const Size imageSize = DEFAULT_IMAGE_SIZE, const ImageFormat format = ImageFormat::YUV_SP_420);
Image(const std::shared_ptr<uint8_t> imageData, const uint32_t dataSize, const int32_t deviceId, const std::pair<Size, Size> imageSizeInfo, const ImageFormat format);
```

参数说明

参数名	输入/输出	说明
imageData	输入	用户构造的输入内存，该内存由用户管理申请和释放，不能为空指针（nullptr）。
dataSize	输入	用户输入内存的大小，需要与实际内存数据大小一致。
imageSize	输入	图像的宽高，默认为(0, 0)。 若用户自申请Device侧的内存，请设置实际内存数据对应的图像高宽数据。
imageSizeInfo	输入	图像**有效宽高**与**对齐后宽高**的组合，有效宽、高应不超过对齐后的宽、高，输入方式参见如下。 std::pair<Size, Size> imageSizeInfo(*有效宽高*，*对齐后的宽高*) ● 有效宽、高取值范围为[6, 8192]。 ● 对齐后宽、高取值范围为[16, 8192]，其中宽为16的倍数、高为2的倍数。
format	输入	图片格式类型。
deviceId	输入	用户输入内存的设备ID。若用户自申请Device侧的内存，请输入对应的deviceId。 取值范围：[-1, *识别到的Device数*-1]。 该值需要与“imageData”为同一侧内存（Host侧为“-1”，Device侧为具体设备ID），否则后续业务存在风险和异常。

5.7.1.3 ~Image

函数功能

Image类的默认析构函数。

函数原型

```
~Image()
```

5.7.1.4 operator =

函数功能

Image类重载等号运算符，对成员变量进行深拷贝，对内存数据进行浅拷贝，引用计数加一。

函数原型

```
Image &Image::operator = (const Image &img)
```

参数说明

参数名	输入/输出	说明
img	输入	输入的Image类。

5.7.1.5 GetDeviceId

函数功能

获取Image对象的deviceId。

函数原型

```
int32_t GetDeviceId() const;
```

返回参数说明

数据结构	说明
int32_t	Image的设备号。

5.7.1.6 GetOriginalData

函数功能

用于获取有效图片内存数据。

当前支持以下图像类型。

```
YUV_400 = 0,
RGB_888 = 12,
BGR_888 = 13,
ARGB_8888 = 14,
ABGR_8888 = 15,
RGBA_8888 = 16,
BGRA_8888 = 17,
```

函数原型

```
std::shared_ptr<uint8_t> GetOriginalData() const;
```

返回参数说明

数据结构	说明
std::shared_ptr<uint8_t>	返回内存地址（智能指针）。

5.7.1.7 SetImageOriginalSize

函数功能

设置有效图片数据的宽高。

需根据5.7.1 Image实际输入的图像数据进行设置，图像数据不可为空。

函数原型

```
APP_ERROR SetImageOriginalSize(const Size whSize);
```

参数说明

参数名	输入/输出	说明
whSize	输入	有效宽高，单位为像素，有效宽、高取值范围为[6, 8192]且不超过对齐后的宽、高。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.1.8 SetImageAlignedSize

函数功能

设置对齐后的图片宽高。

- 需根据5.7.1 Image实际输入的图像数据进行设置，图像数据不可为空。
- 当前支持以下图像类型。

```
YUV_400 = 0,
RGB_888 = 12,
BGR_888 = 13,
ARGB_8888 = 14,
ABGR_8888 = 15,
RGBA_8888 = 16,
BGRA_8888 = 17,
```

函数原型

```
APP_ERROR SetImageAlignedSize(const Size whSize);
```

参数说明

参数名	输入/输出	说明
whSize	输入	对齐后的图片宽高，单位为像素，对齐后的宽、高取值范围为[16, 8192]且大于或等于有效宽、高。 其中宽需为16的倍数，高需为2的倍数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.9 GetData

函数功能

获取Image对象的内存数据指针。

函数原型

```
std::shared_ptr<uint8_t> GetData() const;
```

返回参数说明

数据结构	说明
std::shared_ptr<uint8_t>	返回内存数据的智能指针。

5.7.1.10 GetDataSize

函数功能

获取Image对象的内存数据大小。

函数原型

```
uint32_t GetDataSize() const;
```

返回参数说明

数据结构	说明
uint32_t	返回内存数据的大小。

5.7.1.11 GetSize

函数功能

获取Image对象的图片对齐后的宽高（实际内存大小的宽高）。

函数原型

```
Size GetSize() const;
```

返回参数说明

数据结构	说明
Size	返回图片对齐后的宽高，实际内存大小的宽高。

5.7.1.12 GetOriginalSize

函数功能

获取Image对象的原始图片宽高。

函数原型

```
Size GetOriginalSize() const;
```

返回参数说明

数据结构	说明
Size	返回图片的原始宽高。

5.7.1.13 GetFormat

函数功能

获取Image对象的图片类型。

函数原型

```
ImageFormat GetFormat() const;
```

返回参数说明

数据结构	说明
ImageFormat	返回图片的类型，具体请参见**5.4.1.16 ImageFormat**。

5.7.1.14 ToHost

函数功能

将Image类内保存的图像内存数据转移到Host侧。

函数原型

```
APP_ERROR ToHost();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.15 ToDevice

函数功能

将Image类内保存的图像内存数据转移到Device侧。

函数原型

```
APP_ERROR ToDevice(const int32_t devId);
```

参数说明

参数名	输入/输出	说明
devId	输入	要转移到的设备ID。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.16 ConvertToTensor

函数功能

将DVPP侧Image类转换为Device侧Tensor类，或将Host侧Image转换为Host侧Tensor，转换后的Tensor类对象的数据类型为uint8。

- 若转换失败，则会返回空的Tensor实例。
- 若因内存不足或无法识别芯片等异常场景，导致Tensor类构造失败的话，则会抛出异常。

说明

- “withStride”参数为“true”时，Tensor对象会保留Image对象的补边信息。为减少内存拷贝，提高运行效率，此时返回的Tensor对象与Image对象共享数据内存，同时在该场景下，Image对象的数据内存会与Tensor对象的数据内存相互影响，例如：在Image对象释放后，对应的Tensor对象中的数据将变为无效内存数据。
- ConvertToTensor()为无参数接口时，返回的Tensor对象会保留Image对象的补边信息并存在batch维度（即NHWC，其中batch维度**N = 1**）。

函数原型

```
Tensor ConvertToTensor(bool withStride, bool formatNHWC);
Tensor ConvertToTensor();
```

参数说明

参数名	输入/输出	说明
withStride	输入	bool类型，指定是否保留补边信息。 ● 若为“true”，则表示保留补边信息，与Image对象共享数据内存。 ● 若为“false”，表示不保留补边信息，不与Image对象共享数据内存。
formatNHWC	输入	bool类型，指定转换得到的Tensor是否有batch维度。 ● 若为“true”，则指定返回Tensor中存在batch维度。 ● 若为“false”，则指定返回Tensor中没有batch维度。

返回参数说明

数据结构	说明
Tensor	Tensor类，具体请参见**5.7.3 Tensor**。

5.7.1.17 Serialize

函数功能

将图像内存数据及元数据序列化后落盘保存为文件。

函数原型

```
APP_ERROR Serialize(const std::string& filePath, bool forceOverwrite = false);
```

参数说明

参数名	输入/输出	说明
filePath	输入	序列化后的数据文件保存路径（包括文件名称），不支持软连接。
forceOverwrite	输入	保存时是否强制覆盖已有文件，默认为“false”，不覆盖。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.18 Unserialize

函数功能

将**5.7.1.17 Serialize**中保存的落盘数据文件加载到内存中，需指定文件名称及具体路径。

函数原型

```
APP_ERROR Unserialize(const std::string& filePath);
```

参数说明

参数名	输入/输出	说明
filePath	输入	落盘数据文件保存路径，输入文件大小支持范围为(0, 4GB]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.19 DumpBuffer

函数功能

将图像内存数据落盘到二进制文件，需要指定文件名称及具体路径。

函数原型

```
APP_ERROR DumpBuffer(const std::string& filePath, bool forceOverwrite = false);
```

参数说明

参数名	输入/输出	说明
filePath	输入	落盘数据文件路径（包含文件名称），不支持软连接。
forceOverwrite	输入	保存时是否强制覆盖已有文件，默认为“false”，不覆盖。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.1.20 TensorToImage

函数功能

将Host侧Tensor类转换为Host侧Image类，或将Device侧Tensor类转换为DVPP侧Image类。

在转换过程中，会对Image的宽进行16向上对齐、对Image的高进行2向上对齐，转换后的Image类对象将存在补边区域。

转换后的Image类可调用成员函数**GetSize()**返回Size类对象查看补边后的宽、高，调用成员函数**GetOriginalSize()**返回Size类对象查看图片原始宽、高。

例如：

- Tensor类对象对应的图片宽、高为500、499时，转换得到的Image类对象在补边后，通过GetOriginalSize()可获得宽、高分别为500、499，通过GetSize()可获得宽、高分别为512、500。
- Tensor类对象对应的图片宽、高为512、500时，转换得到的Image类对象不需要补边，即转换后的Image类对象宽、高仍为512、500。

函数原型

```
APP_ERROR TensorToImage(const Tensor& inputTensor, Image& outImg, const ImageFormat& imageFormat)
```

参数说明

参数名	输入/输出	说明
inputTensor	输入	Tensor类，输入张量。输入需满足以下要求。 ● 元素类型需为Uint8类型。 ● Tensor的维度需为2（YUV400格式时）、3、4。 ● Tensor的宽、高、通道数需与“imageFormat”相匹配。
outImg	输出	Image类，输出图片，其内存分配在DVPP侧。
imageFormat	输入	**ImageFormat**类，指定图片的格式，需与“inputTensor”数据所对应的图片格式相匹配。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2 ImageProcessor

5.7.2.1 类说明

ImageProcessor类，作为图像处理类，主要开放图像编解码、缩放和抠图等接口。

ImageProcessor对象不支持在多线程中并发使用，如需多线程使用同一个ImageProcessor对象，用户需自行保证加锁互斥。

说明

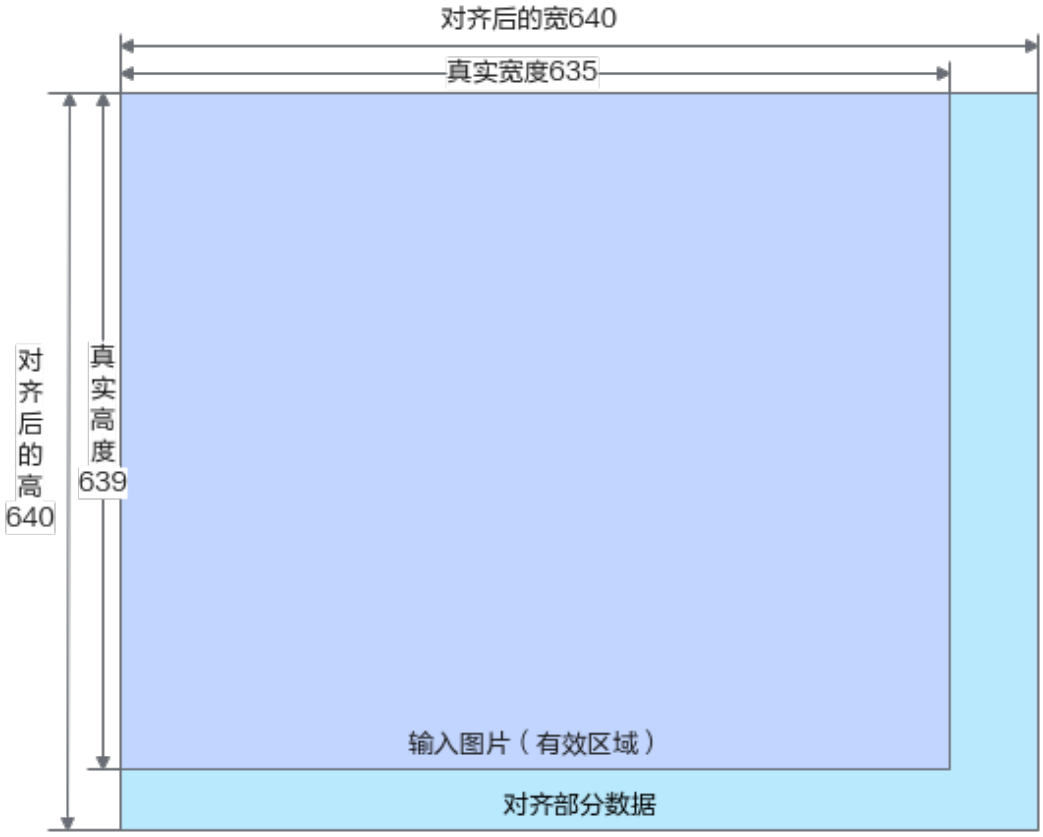
ImageProcessor类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

关于真实图片宽高与对齐后的图片宽高说明

如**图5-5**所示，由于硬件限制，ImageProcessor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（采用向右、向下填充无效数据的方式，对齐到指定大小）。

在对图片进行缩放等操作时，ImageProcessor将会以图片原始的高宽进行处理。

图 5-5 对齐宽高与真实宽高



5.7.2.2 ImageProcessor

函数功能

ImageProcessor类的构造函数，支持异步执行。

若因内存不足，或无法识别芯片等构造失败的场景会抛出std::runtime_error异常。

图像处理接口（解码除外）的输入Image类包含的内存数据均需要在Device侧，用户使用ImageProcessor类内的接口获取的Image对象内存数据已经在Device侧，并且无需用户管理释放。

函数原型

```
ImageProcessor(const int32_t deviceId = 0);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	图像处理类部署的芯片，默认为0号芯片。 取值范围：[0, *识别到的芯片个数* - 1]。

5.7.2.3 ~ImageProcessor

函数功能

ImageProcessor类的默认析构函数。

函数原型

```
~ImageProcessor()
```

5.7.2.4 InitJpegEncodeChannel

函数功能

初始化JPEGE图像通道，用于JPEG编码。

函数原型

```
APP_ERROR InitJpegEncodeChannel(const JpegEncodeChnConfig& config = JPEG_ENCODE_CHN_CONFIG);
```

参数说明

参数名	输入/输出	说明

config	输入	通道配置参数，取值范围为[32, 8192]，默认值为“JPEG_ENCODE_CHN_CONFIG”，（即图像最大宽高为8192 * 8192）。该配置无效。
		当前仅支持配置图片编码的通道宽高（maxPicWidth、maxPicHeight），内部自动16对齐，当高小于宽时，高自动向上对齐到宽的长度，请根据实际编码场景图片预留适当的宽高。
		对应数据结构参见如下。
		struct JpegEncodeChnConfig {
		uint32_t maxPicWidth = MAX_HIMPI_VENC_PIC_WIDTH;
		uint32_t maxPicHeight = MAX_HIMPI_VENC_PIC_HEIGHT;
		};

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.5 InitJpegDecodeChannel

函数功能

初始化JPEGD图像通道，用于JPEG解码。

函数原型

```
APP_ERROR InitJpegDecodeChannel(const JpegDecodeChnConfig& config = JPEG_DECODE_CHN_CONFIG);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数，默认值为“JPEG_DECODE_CHN_CONFIG”。 对应数据结构参见如下（当前预留）。 `struct JpegDecodeChnConfig {};`

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.6 InitVpcChannel

函数功能

初始化VPC图像通道，用于图像处理功能（抠图、缩放、补边、抠图缩放、抠图贴图、色域转换）。

函数原型

```
APP_ERROR InitVpcChannel(const VpcChnConfig& config = VPC_CHN_CONFIG);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数，默认值为“VPC_CHN_CONFIG”。 对应数据结构参见如下（当前预留）。 `struct VpcChnConfig {};`

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.7 InitPngDecodeChannel

函数功能

初始化PNGD图像通道，用于PNG图片解码。

函数原型

```
APP_ERROR InitPngDecodeChannel(const PngDecodeChnConfig& config = PNG_DECODE_CHN_CONFIG);
```

参数说明

参数名	输入/输出	说明
config	输入	通道配置参数，默认值为“PNG_DECODE_CHN_CONFIG”。 对应数据结构参见如下（当前预留）。 `struct PngDecodeChnConfig {};`

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.8 Decode

函数功能

ImageProcessor类的图片解码接口，使用该接口申请的Image内存无需用户管理，由内部管理释放。仅支持以Host侧申请的内存作为解码接口的输入。输入图片内存的数据类型目前支持**JPEG和PNG两种**格式。

- JPG/JPEG格式：
  - JPG/JPEG输入图片的最大分辨率：8192 * 8192，其中 RGB_888, BGR_888 格式只支持至 4096*4096。
  - JPG/JPEG输入图片的最小分辨率：32 * 32。
  - 输出解码后的图片及“outputImage”的数据类型目前仅支持**YUV_SP_420**，**YVU_SP_420**两种图像格式。

- 输出图片的宽。
  - Atlas 200/500 A2 推理产品对齐到128（即宽度为128的倍数），在对齐操作前，接口会向下2对齐操作，例如：图片原图宽为1023，在进行Decode接口解码处理后，通过**GetSize**()获得的值为1024，通过**GetOriginalSize**()获得的值为1022。
- 输出图片的高：对齐到16（即高度为16的倍数），解码接口自动对齐。

  对于Atlas 200/500 A2 推理产品，在对齐操作前，接口会向下2对齐操作，例如：图片原图高为683，在进行Decode接口解码处理后，通过GetSize()获得的值为688，通过GetOriginalSize()获得的值为682。

  说明

  JPG/JPEG输入图片格式约束：
  - 只支持Huffman编码，码流的subsample为444/422/420/400/440。
  - 不支持算术编码。
  - 不支持渐进JPEG格式。
  - 不支持JPEG2000格式。

- PNG格式：
  - PNG输入图片的最大分辨率：4096 * 4096。
  - PNG输入图片的最小分辨率：32 * 32。
  - 输出图片的宽：对齐到128（即宽度为128的倍数），解码接口自动对齐。
  - 输出图片的高：对齐到16（即高度为16的倍数），解码接口自动对齐。

函数原型

```
APP_ERROR Decode(const std::shared_ptr<uint8_t> dataPtr, const uint32_t dataSize,
        Image& outputImage, const ImageFormat decodeFormat = ImageFormat::YUV_SP_420);
APP_ERROR Decode(const std::string inputPath, Image& outputImage,
        const ImageFormat decodeFormat = ImageFormat::YUV_SP_420);
```

参数说明

参数名	输入/输出	说明
dataPtr	输入	输入待解码图片数据的内存地址。解码前图像数据内存地址需要用户进行管理。
dataSize	输入	输入待解码图片数据的内存大小。需要与“dataPtr”的实际内存大小相符。 ● 用户提供的“dataSize”无法完整读取图片文件头信息，则会返回异常。 ● 用户提供的“dataSize”大于文件头信息并且小于实际内存大小，则会根据“dataSize”进行部分解码。 ● 用户提供的“dataSize”大于实际内存大小，则以图片文件结束符为准。

参数名	输入/输出	说明
decodeFormat	输入	输入解码后图片的格式。 ● JPG/JPEG图片“decodeFormat”默认参数为“YUV_SP_420”，可自行设置该参数。 ● 对于PNG图片，“decodeFormat”仅在输入图片通道是RGB/GRAY格式时，配置BGR_888会生效；其余情况设置该参数无效，将按PNG源格式进行解码，例如： – PNG图片通道为RGB/GRAY格式，则解码输出图片格式为RGB_888。 – PNG图片通道为RGBA/AGRAY格式，则解码输出图片格式为RGBA_8888。
inputPath	输入	输入待解码的图片路径。
outputImage	输出	输出解码后的Image类。 图片宽高和对齐后的宽高会自动合入进“outputImage”内。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.9 Encode

函数功能

ImageProcessor类的图片编码接口，使用该接口申请的Image内存无需用户管理，由内部管理释放。

- 输入图像的最大分辨率：8192 * 8192。
- 输入图像的最小分辨率：32 * 32。
- 输入图像的真实宽、高均为偶数，若为奇数，则自动向上取偶数，编码出的图片自动补边**1**像素。
- 输入图像的宽：对于YUV420SP或RGB数据，对齐到16。
- 输入图像的高：与输入图片的高度相同的数值，或为输入图片的高度向上对齐到16的数值（最小为32）。
- 输入图像格式：

  - Atlas 200/500 A2 推理产品支持YUV_SP_420、YVU_SP_420（nv12、nv21）。
- 输出图片格式：jpeg压缩格式的图片文件，例如*.jpg。

函数原型

```
APP_ERROR Encode(const Image& inputImage, const std::string savePath, const uint32_t encodeLevel = 100);
APP_ERROR Encode(const Image& inputImage, std::shared_ptr<uint8_t>& outDataPtr, uint32_t& outDataSize, const uint32_t encodeLevel = 100);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入编码前的Image类。Decode接口和其他VPC接口获取的Image类可以直接作为输入。 若是用户自定义构造的Image类，则需要设置图像宽高。
encodeLevel	输入	默认为“100”，输入编码的级别[0, 100]。
savePath	输入	输入编码后保存的图片路径，文件后缀名限制为**jpg**。
outDataPtr	输出	输出编码后的图片内存数据地址。
outDataSize	输出	输出编码后的图片内存数据大小。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.10 Resize

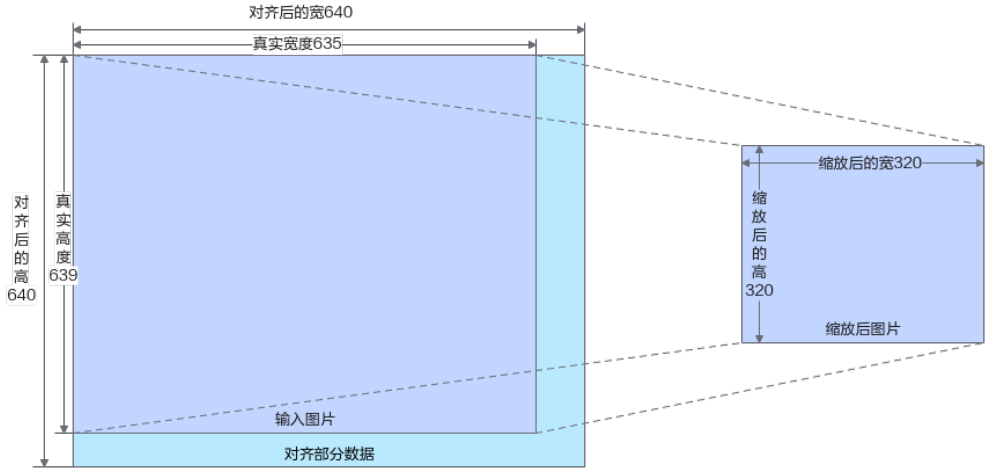
函数功能

ImageProcessor类的图像缩放接口，使用该接口申请的Image内存无需用户管理，由内部管理释放，缩放效果示意图请参见**图5-6**。

1. 输入输出Image类支持的图像格式参考如下。
   - Atlas 200/500 A2 推理产品支持YUV_SP_420、YVU_SP_420（nv12、nv21）。
2. “inputImage”的真实图片分辨率范围：18 * 6 ~ 4096 * 4096，其中YUV_SP_420 和 YVU_SP_420格式的宽高可达到 8192 * 8192。

3. 参数“resize”的最大分辨率：4096 * 4096，最小分辨率：32 * 6。
4. “outputImage”宽自动与16对齐，高与2对齐，因此宽高范围为：32 * 6 ~ 4096 * 4096。
5. 缩放后图片的宽高不能超出真实图片的[1/32 ,16]倍数区间。

图 5-6 缩放



函数原型

```
APP_ERROR Resize(const Image& inputImage, const Size& resize, Image& outputImage, const Interpolation interpolation = Interpolation::HUAWEI_HIGH_ORDER_FILTER);
APP_ERROR Resize(const Image& inputImage, const Size& resize, Image& outputImage, const Interpolation interpolation, AscendStream& stream);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入缩放前的Image类。Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高和图像对齐后的宽高。
resize	输入	输入图像缩放的宽高。

参数名	输入/输出	说明
interpolation	输入	输入图像的缩放方式，可选参数参见如下。 HUAWEI_HIGH_ORDER_FILTER = 0 BILINEAR_SIMILAR_OPENCV = 1 NEAREST_NEIGHBOR_OPENCV = 2 BILINEAR_SIMILAR_TENSORFLOW = 3 NEAREST_NEIGHBOR_TENSORFLOW = 4 Atlas 200/500 A2 推理产品支持以下算法（默认为0）。 ● 0：华为自研的高滤波算法。 ● 1：业界通用的Bilinear算法（与OpenCV算法的计算精度接近）。 ● 2：业界通用的Nearest neighbor算法（与OpenCV算法的计算精度接近）。 ● 3：业界通用的Bilinear算法（与Tensorflow框架的计算精度接近）。 ● 4：业界通用的Nearest neighbor算法（与Tensorflow框架的计算精度接近）。
outputImage	输出	输出缩放后的Image类。
stream	输入	输入用于异步执行的Stream，具体请参见**5.9.1 AscendStream**。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.11 Padding

须知

Atlas 200/300/500 推理产品暂不支持。

函数功能

ImageProcessor类的图像补边接口，使用该接口申请的Image内存无需用户管理，由内部管理释放。

1. 输入输出Image类支持的图像格式为YUV_SP_420、YVU_SP_420、RGB_888、BGR_888（nv12、nv21、rgb、bgr），其中RGB（BGR）图像格式分辨率不超过（4096 * 4096）。
2. “inputImage”的真实图片分辨率范围：18 * 6 ~ 4096 * 4096。
3. 当前接口仅支持“BORDER_CONSTANT”补边方式，其余补边方式预留接口。

   YUV_SP_420和YVU_SP_420格式，补边尺寸建议为偶数，当补边尺寸为奇数时，会自动进行向上对齐。

   例如，用户输入补边尺寸为(1, 1, 1, 1)，将自动对齐到(2, 2, 2, 2)，上下左右各补2个像素点。

   当输入图片分辨率为4095 * 4095，补边尺寸为(1, 0, 1, 0)时，由于自动对齐后的补边尺寸为4097 * 4097，超出范围，因此会补边失败。
4. “outputImage”分辨率为18 * 6 ~ 4096 * 4096，宽自动与16对齐，高与2对齐，范围为32 * 6 ~ 4096 * 4096。

函数原型

```
APP_ERROR Padding(const Image& inputImage, Dim &padDim, const Color& color, const BorderType borderType, Image& outputImage);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入补边前的Image类。Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高和图像对齐后的宽高。
padDim	输入	输入图像补边的尺寸。
color	输入	输入补边三通道颜色值，仅在“borderType”设置为“BORDER_CONSTANT”时有效。
borderType	输入	输入补边方式，具体实现请参见**5.4.1.15 BorderType**。
outputImage	输出	输出缩放后的Image类。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.12 Crop

函数功能

ImageProcessor类的图像抠图接口，使用该接口申请的Image内存无需用户管理，由内部管理释放，抠图效果示意图请参见图5-7。

输入输出Image类支持的图像格式参考如下。

- Atlas 200/500 A2 推理产品支持YUV_SP_420、YVU_SP_420（nv12、nv21）。

1. “inputImage”的真实图片分辨率范围：18 * 6 ~ 4096 * 4096，其中YUV_SP_420 和 YVU_SP_420 格式的分辨率为18 * 6 ~ 8192 * 8192。
2. 抠图区域不超出输入图片区域，输入抠图坐标框“cropRect”的四个值推荐均为偶数。

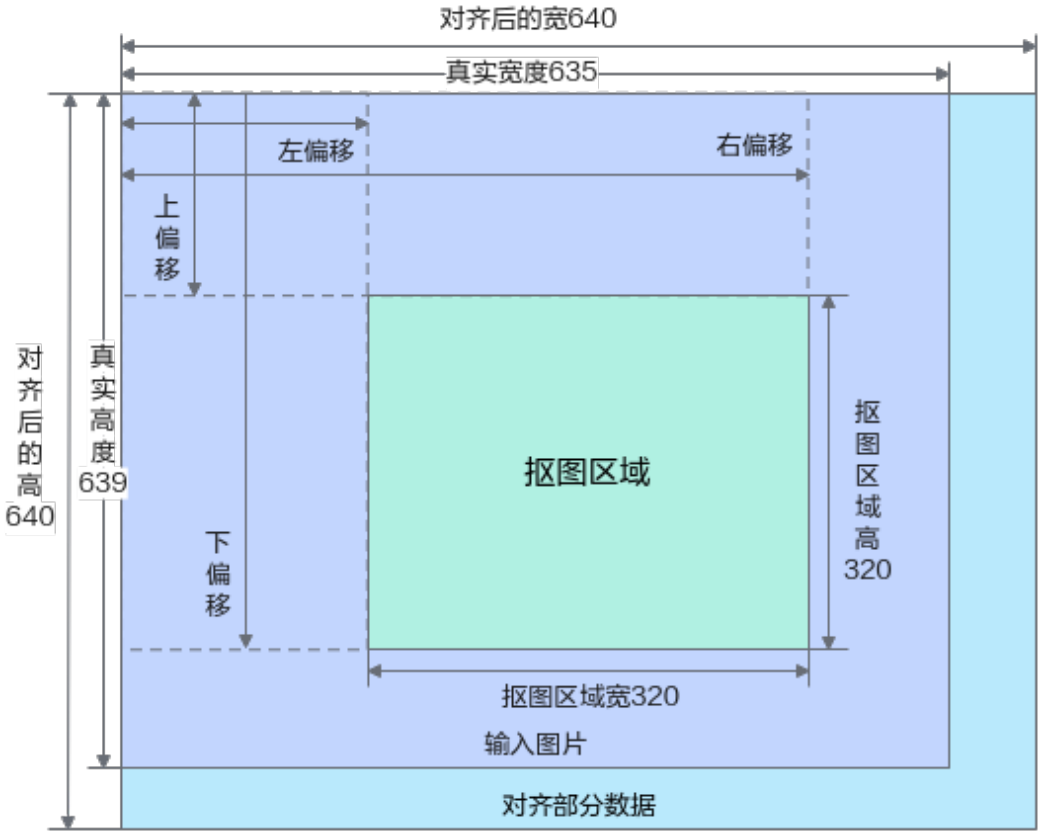
   除RGB、BGR以外，若包含奇数，则左上角坐标自动向下取偶数，右下角坐标自动向上取偶数。

   抠图区域的最大分辨率为4096 * 4096，最小分辨率为18 * 6。

   例如：cropRect{1, 1, 1287, 1287}，实际抠图宽高为：((1287 + 1) - (1 - 1))= 1288，对应的分辨率为1288 * 1288。
3. 输出“outputImageVec”中，各项图片的宽自动与16对齐， 高与2对齐，范围为：32 * 6 ~ 4096 * 4096。

批量输入图片的抠图场景，输入图片数量不超过12张，抠图配置参数“cropRectVec”长度不超过256，输出图片张数不超过256且应满足**输出图片张数 = 输入图片张数 * 抠图配置参数“cropRectVec”长度**。

图 5-7 抠图



函数原型

```
APP_ERROR Crop(const Image& inputImage, const Rect& cropRect, Image& outputImage);
APP_ERROR Crop(const Image& inputImage, const Rect& cropRect, Image& outputImage, AscendStream& stream);
APP_ERROR Crop(const Image& inputImage, const std::vector<Rect>& cropRectVec, std::vector<Image>& outputImageVec);
APP_ERROR Crop(const Image& inputImage, const std::vector<Rect>& cropRectVec, std::vector<Image>& outputImageVec, AscendStream& stream);
APP_ERROR Crop(const std::vector<Image>& inputImageVec, const std::vector<Rect>& cropRectVec, std::vector<Image>& outputImageVec);
APP_ERROR Crop(const std::vector<Image>& inputImageVec, const std::vector<Rect>& cropRectVec, std::vector<Image>& outputImageVec, AscendStream& stream);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入抠图前的Image类。 Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高。
inputImageVec	输入	输入抠图前的Image类列表（针对批量抠图场景）。
cropRect	输入	输入图像的抠图坐标框。
cropRectVec	输入	输入图像的抠图坐标框列表（针对批量抠图场景）。
outputImage	输出	输出抠图后的Image类。
outputImageVec	输出	输出抠图后的Image类列表（针对批量抠图场景）。
stream	输入	输入用于异步执行的Stream，具体请参见**5.9.1 AscendStream**。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.2.13 CropResize

函数功能

ImageProcessor类的图像抠图并缩放接口，使用该接口申请的Image内存无需用户管理，由内部管理释放，抠图缩放效果示意图请参见**图5-8**。

输入输出Image类支持的图像格式参考如下。

- Atlas 200/500 A2 推理产品支持YUV_SP_420、YVU_SP_420（nv12、nv21）。

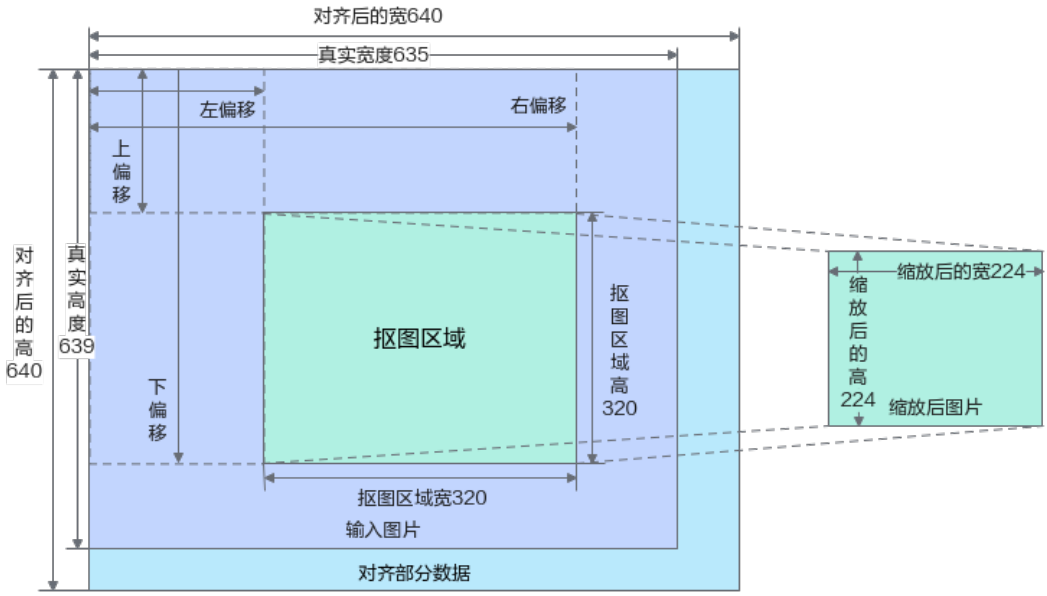
1. “inputImage”的真实图片宽高大小范围：18 * 6 ~ 4096 * 4096，其中YUV_SP_420和YVU_SP_420格式的宽高可达到8192 * 8192。
2. 抠图区域的最小为10 * 6，抠图区域不能超出输入图片的真实图片宽高，输入“cropRect”的四个值推荐均为偶数。

   除RGB、BGR以外，若包含奇数，则左上角坐标自动向下取偶数，右下角坐标自动向上取偶数。

   输出Image类的最大分辨率为4096 * 4096，最小分辨率为18 * 6。

   例如：cropRect{1, 1, 1287, 1287}，实际抠图宽高为：((1287 + 1) - (1 - 1))= 1288，对应的分辨率为1288 * 1288。
3. 缩放的范围为18 * 6 ~ 4096 * 4096，不能超出抠图区域的[1/32, 16]倍数区间。
4. 输出“outputImageVec”中，各项图片的宽自动与16对齐， 高与2对齐，范围为：32 * 6 ~ 4096 * 4096。

图 5-8 抠图缩放



函数原型

```
APP_ERROR CropResize(const Image& inputImage, const std::vector<Rect>& cropRectVec, const Size& resize, std::vector<Image>& outputImageVec);
APP_ERROR CropResize(const Image& inputImage, const std::vector<Rect>& cropRectVec, const Size& resize, std::vector<Image>& outputImageVec, AscendStream& stream);
APP_ERROR CropResize(const Image& inputImage, const std::vector<std::pair<Rect, Size>>& cropResizeVec, std::vector<Image>& outputImageVec);
APP_ERROR CropResize(const Image& inputImage, const std::vector<std::pair<Rect, Size>>& cropResizeVec, std::vector<Image>& outputImageVec, AscendStream& stream);
APP_ERROR CropResize(const std::vector<Image>& inputImageVec, const std::vector<std::pair<Rect, Size>>& cropResizeVec, std::vector<Image>& outputImageVec);
APP_ERROR CropResize(const std::vector<Image>& inputImageVec, const std::vector<std::pair<Rect, Size>>& cropResizeVec, std::vector<Image>& outputImageVec, AscendStream& stream);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入抠图缩放前的Image类。 Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高和图像对齐后的宽高。
inputImageVec	输入	输入抠图缩放前的Image类列表（针对批量抠图场景）。 Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高和图像对齐后的宽高。
cropRectVec	输入	输入抠图参数列表，需要与输出图像列表的元素个数一致。
resize	输入	输入统一缩放宽高。
cropResizeVec	输入	输入图像的抠图缩放参数列表。Rect为抠图坐标框，Size为缩放宽高（针对批量抠图缩放场景）。
outputImageVec	输出	输出抠图后的Image类列表（针对批量抠图场景）。
stream	输入	输入用于异步执行的Stream，具体请参见5.9.1 AscendStream。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.2.14 CropAndPaste

函数功能

ImageProcessor类的图像抠图并贴图接口，抠图贴图效果示意图请参见图5-9。

1. 输入输出Image类支持的图像格式参考如下。
   - Atlas 200/500 A2 推理产品支持YUV_SP_420、YVU_SP_420（nv12、nv21）。
2. 从“inputImage”中抠取一块图像。

a. 输入Image类的真实图片宽高大小范围：18 * 6 ~ 4096 * 4096，其中YUV_SP_420 和 YVU_SP_420 格式的宽高可达到 8192 * 8192。

b. 抠图宽高范围不能超过“inputImage”的真实图片宽高，抠图区域的范围最小为：10 * 6。

输入**抠图参数**的四个值推荐均为偶数。除RGB、BGR以外，若包含奇数，则左上角坐标自动向下取偶数，右下角坐标自动向上取偶数。

例如：cropRect{1, 1, 1287, 1287}，实际抠图宽高为：((1287 + 1) - (1 - 1))= 1288。

3. 缩放至指定贴图区域的大小。
4. 将缩放后的图片贴到“pastedImage”的指定贴图区域。

a. 贴图宽高范围不能超过“pastedImage”的真实图片宽高，贴图区域的范围最小为： 10 * 6 ，最大为：4096 * 4096。

b. 输入**贴图参数**的四个值推荐均为偶数。除RGB、BGR以外，若包含奇数，则左上角坐标自动向下取偶数，右下角坐标自动向上取偶数。

c. **贴图参数**左上角坐标的x会自动对齐到16的倍数。例如：

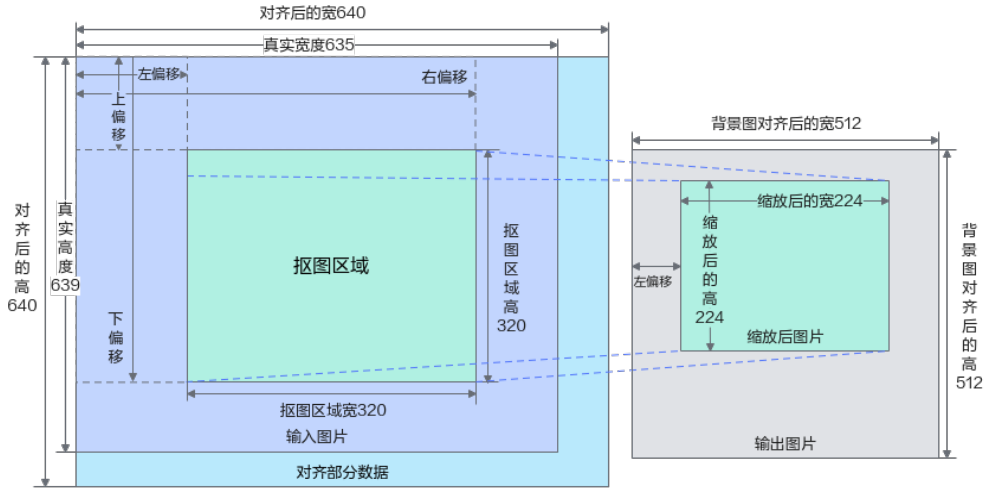
- pasteRect{17, 17, 1287, 1287}，实际贴图宽高为：((1287 + 1) - (17 - 1))= 1272，对应的分辨率为1272 * 1272。
- pasteRect{18, 18, 1287, 1287}，实际贴图宽为：((1287 + 1) - 32) = 1256，高为：((1287 + 1) - 18) = 1270，对应的分辨率为1256 * 1270。

d. 贴图宽高不能超过抠图宽高的[1/32, 16]倍数区间。

e. 在Atlas 200/500 A2 推理产品环境下，贴图“rect”的右下角的“x”值推荐与“16”对齐。

5. 输出的“pastedImage”宽自动与16对齐，高与2对齐，因此宽高范围为：32 * 6 ~ 4096 * 4096。

图 5-9 抠图缩放贴图



函数原型

```
APP_ERROR CropAndPaste(const Image& inputImage, const std::pair<Rect, Rect>& cropPasteRect, Image& pastedImage);
APP_ERROR CropAndPaste(const Image& inputImage, const std::pair<Rect, Rect>& cropPasteRect, Image& pastedImage, AscendStream& stream);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入抠图缩放前的Image类。 Decode接口和其他VPC接口获取的Image类可以直接作为输入。若是用户自定义构造的Image类，则需要设置图像宽高和图像对齐后的宽高。
cropPasteRect	输入	输入图像的抠图缩放参数。第一个Rect对应**抠图参数**，第二个Rect对应**贴图参数**。
pastedImage	输入/输出	输出抠图后的Image类。
stream	输入	输入用于异步执行的Stream，具体请参见**5.9.1 AscendStream**。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3 Tensor

5.7.3.1 类说明

Tensor数据类，作为模型推理的输入与输出的数据结构。

说明

Tensor类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

5.7.3.2 Tensor

函数功能

Tensor类的构造函数。

函数原型

```
Tensor()  //默认构造函数。 构造失败时，会抛出std::runtime_error异常
Tensor(const Tensor &tensor); // 支持拷贝构造
Tensor(const std::vector<uint32_t> &shape, const MxBase::TensorDType &dataType, const int32_t &deviceId
= -1)
//不传入内存，可搭配TensorMalloc接口进行内存申请，申请的内存无需用户管理释放。 构造失败时，会抛出
std::runtime_error异常
```

```
Tensor(void* usrData,const std::vector<uint32_t> &shape, const MxBase::TensorDType &dataType, const
int32_t &deviceId = -1)
//传入用户自己构造的数据内存，需要用户自身对此内存进行管理（保证内存数据的生命周期）。 构造失败时，
会抛出std::runtime_error异常
 Tensor(const std::vector<uint32_t> &shape, const MxBase::TensorDType &dataType, const int32_t &deviceId,
        bool isDvpp);
Tensor(void *usrData, const std::vector<uint32_t> &shape, const MxBase::TensorDType &dataType,
     const int32_t &deviceId, const bool isDvpp, const bool isBorrowed);
Tensor(const Tensor &tensor, const Rect &rect); // 带引用区域的拷贝构造
```

参数说明

参数名	输入/输出	说明
tensor	输入	已初始化的其它Tensor。
usrData	输入	用户构造的输入内存，该内存由用户管理申请和释放。
shape	输入	Tensor的shape属性。
dataType	输入	Tensor的数据类型，具体请参见**5.4.1.11 TensorDType**。
deviceId	输入	Tensor所在的设备ID，默认为-1，在Host侧。 ● 当Tensor构造函数中使用“isDVPP”参数，此时deviceID无默认值，需根据实际情况输入。 ● 如果传入用户指针“usrData”，该值需要与deviceId为同一侧内存（Host侧为“-1”，Device侧为具体设备ID），否则后续业务存在风险和异常。
isDvpp	输入	设置是否申请DVPP的内存，“deviceId”如果为“-1”，则申请Host侧内存，该参数设置无效。 如果传入用户指针“usrData”并且“isDvpp”设置为“true”，则需保证“usrData”指向的内存在Device侧，否则后续业务存在风险和异常。
isBorrowed	输入	设置是否将usrData指向的内存交由Tensor来释放，“isBorrowed”的值设为“false”，则由Tensor来释放usrData指向的内存，用户无需释放；“isBorrowed”的值设为“true”，则用户需要自行管理usrData指向的内存。 **须知** 委托Tensor管理该内存，仅支持需要手动释放的内存，否则可能导致内存重复释放。

参数名	输入/输出	说明
rect	输入	代表图片的引用区域坐标（x0, y0, x1, y1）左闭右开。 **须知** 若采用Tensor(const Tensor &tensor, const Rect &rect)的方式进行构造，要求： ● tensor不能为空，仅支持NHWC、HWC和HW的tensor，通道数为1/3/4，batch维度为1。 ● rect区域x0,y0需要分别小于x1,y1；x0, y0, x1,y1需要在tensor的宽高范围内。

5.7.3.3 ~Tensor

函数功能

Tensor类的默认析构函数。

函数原型

```
~Tensor()
```

5.7.3.4 operator =

函数功能

Tensor类重载等号运算符，对成员变量进行深拷贝，对内存数据进行浅拷贝，引用计数加一。

函数原型

```
Tensor &operator=(const Tensor &other)
```

参数说明

参数名	输入/输出	说明
other	输入	输入的Tensor类。

5.7.3.5 operator ==

函数功能

Tensor类重载相等运算符，检查两个Tensor的内容是否相等。

函数原型

```
bool operator==(const Tensor &other);
```

参数说明

参数名	输入/输出	说明
other	输入	输入的Tensor类。

5.7.3.6 GetData

函数功能

获得Tensor的内存数据。

函数原型

```
void* GetData() const
```

返回参数说明

数据结构	说明
void*	Tensor数据裸指针。

5.7.3.7 SetShape

函数功能

设置Tensor的形状（Shape）。

函数原型

```
APP_ERROR SetShape(std::vector<uint32_t> shape);
```

参数说明

参数名	输入/输出	说明
shape	输入	Tensor的形状。 shape所代表的元素个数需要与Tensor的原shape所代表的元素个数相同。shape向量中各维度要求为正整数且单个或各项乘积需小于536,870,912（512 * 1024 * 1024），否则函数将抛出异常。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.8 GetShape

函数功能

获得Tensor的shape数据。

函数原型

```
std::vector<uint32_t> GetShape() const
```

返回参数说明

数据结构	说明
std::vector<uint32_t>	Tensor的shape数据。

5.7.3.9 GetDataType

函数功能

获得Tensor的数据类型。

函数原型

```
MxBase::TensorDType GetDataType() const
```

返回参数说明

数据结构	说明
MxBase::TensorDType	Tensor的数据类型，请参见**5.4.1.11 TensorDType**。

5.7.3.10 GetMemoryType

函数功能

获得Tensor的内存类型。

函数原型

```
MemoryData::MemoryType GetMemoryType() const
```

返回参数说明

数据结构	说明
MemoryData::MemoryType	MemoryType的数据类型，请参见**5.4.1.28 MemoryData**。

5.7.3.11 GetByteSize

函数功能

获得Tensor数据内存占的字节量。

函数原型

```
size_t GetByteSize() const
```

返回参数说明

数据结构	说明
size_t	Tensor数据内存占的字节量。

5.7.3.12 GetDeviceId

函数功能

获得Tensor数据所在的芯片的编号。

函数原型

```
int32_t GetDeviceId() const
```

返回参数说明

数据结构	说明
int32_t	Tensor数据所在的芯片的编号（-1代表在Host侧）。

5.7.3.13 ToDevice

函数功能

将Tensor数据转移到Device侧。

- 如果原始的内存为用户通过构造函数传入，则该原始内存由用户自行管理释放。
- 若是通过**5.7.3.21 TensorMalloc**申请的内存，则是将数据转移到Device侧，原始内存被自动释放。

函数原型

```
APP_ERROR ToDevice(int32_t deviceId)
```

参数说明

参数名	输入/输出	说明
deviceId	输入	将Tensor转移到deviceId对应的设备。“deviceId”需为有效的设备ID。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.3.14 ToHost

函数功能

将Tensor数据转移到Host侧。

- 如果原始的内存为用户通过构造函数传入，则该内存由用户自行管理释放。
- 若是通过5.7.3.21 TensorMalloc申请的内存，则是将数据转移到Host侧，原始内存被自动释放，无需用户管理。

函数原型

```
APP_ERROR ToHost()
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.3.15 ToDvpp

函数功能

将Tensor数据转移到Dvpp侧。

- 如果原始的内存为用户通过构造函数传入，则该内存由用户自行管理释放。
- 若是通过5.7.3.21 TensorMalloc申请的内存，则原始内存被自动释放，无需用户管理。

- 使用媒体数据处理功能，申请DVPP侧内存时存在一定约束，具体请参见《CANN 应用软件开发指南 (C&C++)CANN 应用软件开发指南 (C&C++)》。

函数原型

```
APP_ERROR ToDvpp(int32_t deviceId)
```

参数说明

参数名	输入/输出	说明
deviceId	输入	将Tensor转移到deviceId对应的设备。 “deviceId”需为有效的设备ID。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.16 IsEmpty

函数功能

判断Tensor是否为空。

函数原型

```
bool IsEmpty()
```

返回参数说明

数据结构	说明
bool	判断Tensor是否为空的结果，布尔类型。

5.7.3.17 TensorFree

函数功能

释放Tensor数据。

函数原型

```
static APP_ERROR TensorFree(Tensor &tensor);
```

参数说明

参数名	输入/输出	说明
tensor	输入	待释放的Tensor类数据。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.18 IsWithMargin

函数功能

查询Tensor是否存在补边。

函数原型

```
bool IsWithMargin() const;
```

返回参数说明

数据结构	说明
bool	判断Tensor是否存在补边的结果，布尔类型。

5.7.3.19 SetValidRoi

函数功能

支持设置Tensor有效区域，支持NHWC（N=1）、HWC与HW维度, 有效区域的宽高不能超过原始图片。

函数原型

```
APP_ERROR SetValidRoi(Rect rect);
```

参数说明

参数名	输入/输出	说明
rect	输入	输入图像的坐标框，数据类型请参见**5.4.1.6 Rect**。 起始坐标(x0, y0)只能为(0, 0)。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.3.20 GetValidRoi

函数功能

支持查询Tensor有效区域。

函数原型

```
Rect GetValidRoi() const;
```

参数说明

无

返回参数说明

数据结构	说明
MxBase::Rect	Rect的数据类型，请参见5.4.1.6 Rect。

5.7.3.21 TensorMalloc

函数功能

Tensor的内存申请接口，使用该接口申请的Tensor内存无需用户管理，由内部管理释放。

函数原型

```
static APP_ERROR TensorMalloc(Tensor &tensor)
```

参数说明

参数名	输入/输出	说明
tensor	输出	待申请内存的Tensor，使用不传入内存的构造函数进行构造。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.3.22 BatchConcat

函数功能

将多个Tensor进行组batch，按照batch维组装，默认输入的每个Tensor第一维为batch维，内存连续。

函数原型

```
friend APP_ERROR BatchConcat(const std::vector<Tensor> &inputs, Tensor &output)
```

参数说明

参数名	输入/输出	说明
inputs	输入	待组batch的Tensor列表。
output	输出	组装好batch的Tensor。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.7.3.23 Transpose

函数功能

可通过指定的一组轴维度（axes），对输入的Tensor数据进行转置处理，如未指定具体轴维度，则默认对Tensor数据进行反序转置。

该功能支持对输出数据进行内存复用，用户可通过预先申请的内存（内存大小需与输入一致）传入输出数据。

功能仅支持在Host侧执行操作，如需处理Device侧Tensor数据，请先通过5.7.3.14 ToHost接口，将Device侧数据转移到Host侧，再进行转置。

函数原型

```
friend APP_ERROR Transpose(const Tensor &input, Tensor &output, std::vector<uint32_t> axes = {})
```

参数说明

参数名	输入/输出	说明
input	输入	待转置Tensor类。 维度支持2维、3维、4维，数据类型支持float32、float16、uint8。
output	输出	转置后的Tensor类。
axes	输入	转置选项，默认值为空。 如果未指定具体axes，则默认生成反序“axes”对“input”中的数据进行反序转置。 例如：三维张量默认生成反序“axes”为{2, 1, 0}。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.24 Clone

函数功能

原型1：

将Tensor进行深拷贝，返回拷贝后得到的Tensor。

- 如果Tensor在Host侧，拷贝过程均为同步操作。
- 如果Tensor在DVPP或Device侧，则需根据“stream”参数传入决定同步或异步操作, Tensor所在的Device需与stream所在的Device一致。

原型2：

将Tensor进行指定区域深拷贝，将src的区域内容拷贝到被赋值张量的区域，要求：

- src和被赋值Tensor均不能为空，被赋值Tensor的长宽尺寸为[64, 4096]，src的最大高度不超过“1048576”，src总尺寸（N*H*W*C）不超过“6710886”。
- src和被赋值Tensor包含相同高宽的引用区域（ReferRect），且均不为0，ReferRect宽度不超过“1920”。
- src和被赋值Tensor数据类型支持uint8、float16，且类型需一致。
- src和被赋值Tensor仅支持NHWC、HWC和HW形状的Tensor（通道数为1或3，N为1），两者维度和通道数均需相等。
- src和被赋值Tensor需要在DVPP或Device侧。

- src和被赋值所在的Device需与stream所在的Device一致。

函数原型

```
// 原型1
Tensor Clone(AscendStream stream=AscendStream::DefaultStream()) const;
// 原型2
APP_ERROR Clone(const Tensor &src, AscendStream &stream = AscendStream::DefaultStream());
```

参数说明

参数名	输入/输出	说明
src	输入	将src的引用指定区域赋值给执行该方法的张量的引用区域。
stream	输入	**AscendStream**类型，默认值为AscendStream::DefaultStream()。当参数值为默认值时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

原型1：

数据结构	说明
Tensor	Tensor类，请参见**5.7.3 Tensor**。

原型2：

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.25 SetTensorValue

函数功能

设置Tensor的值，支持int32_t、uint8_t、float16、float32类型。

Tensor对象需在Device侧且数据类型与调用的SetTensorValue方法匹配。

Tensor所在的Device需与stream所在的Device一致。

函数原型

```
APP_ERROR SetTensorValue(uint8_t value, AscendStream& stream = AscendStream::DefaultStream());
APP_ERROR SetTensorValue(float value, bool IsFloat16 = false, AscendStream& stream =
AscendStream::DefaultStream());
```

```
APP_ERROR SetTensorValue(int32_t value, AscendStream& stream = AscendStream::DefaultStream());
```

参数说明

参数名	输入/输出	说明
value	输入	int32_t、uint8_t、float类型，指定Tensor待设置的值。
isFloat16	输入	bool类型，默认为“false”。 ● 为“true”时表示设置Tensor元素为floa16类型。 ● 为“false”时表示设置Tensor元素为floa32类型。
stream	输入	**AscendStream**类型，默认值为AscendStream::DefaultStream()。当参数值为默认值时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.26 SetReferRect

函数功能

支持设置Tensor引用区域，支持NHWC（N=1）、HWC与HW维度，通道为1、3或4，引用区域的宽高不能超过原始图片。

函数原型

```
APP_ERROR SetReferRect(Rect rect);
```

参数说明

参数名	输入/输出	说明
rect	输入	输入图像的坐标框，x0 < x1，y0 < y1。数据类型请参见**5.4.1.6 Rect**。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.3.27 GetReferRect

函数功能

支持查询Tensor的引用区域。

函数原型

```
Rect GetReferRect() const;
```

参数说明

无

返回参数说明

数据结构	说明
MxBase::Rect	Rect的数据类型，请参见**5.4.1.6 Rect**。

5.7.4 TensorDvpp

5.7.4.1 CvtColor

函数功能

将Tensor的图像色域类型转换为指定的色域类型，支持以下类型间的转换。

GRAY转RGB和RGBA转mRGBA场景可通过预加载Split和Merge提高接口执行效率。该场景接口执行时，会在接口执行目录生成记录融合规则的“fusion_result.json”文件，该文件字段详细说明及融合规则开关配置可参考《CANN ATC工具使用指南CANN ATC工具使用指南》的“参数说明 > 高级功能参数 > 模型调优选项 > - > -fusion_switch_file”章节。

“outputTensor”若设置“keepMargin”为“true”，输出宽自动与16对齐。默认值为“false”，不保留Tensor中无效的边界区域。

当前Atlas 200/500 A2 推理产品支持YUVSP420（nv12）转YUV400（GRAY）。

支持：

函数原型

```
APP_ERROR CvtColor(const Tensor &inputTensor, Tensor &outputTensor, const CvtColorMode &mode, bool keepMargin = false, AscendStream &stream = AscendStream::DefaultStream());
```

参数说明

参数名	输入/输出	说明
inputTensor	输入	Tensor类，输入张量，不可为空，需在Device/DVPP侧分配内存，数据类型为UINT8。 ● 形状为 {*高*， *宽*， *通道数*} 的张量，通道数需要与输入格式一致。 **说明** 若“inputTensor”色域类型为YUVSP420或YVUSP420，设置输入张量形状时，高为图片显示高度的1.5倍。张量的高需要为3的倍数，宽为16的倍数，通道为1。 “inputTensor”使用**SetValidRoi**接口设置有效区域时，其**y1**（对应张量的有效高度）需要为3的倍数，对应实际图片的有效高度为**y1**/3 * 2。 ● 输入张量宽度范围为[10, 4096]，高度范围为[6, 4096]，若输入或输出张量格式包含YUV400、YUVSP420、YVUSP420、GRAY时，宽度范围为[18, 4096]。 ● 如果是灰度图，支持形状为{高*宽}的张量。
outputTensor	输出	Tensor类，输出张量。 若不为空，需在Device/DVPP侧分配内存且数据类型为UINT8。

mode	输入	枚举类值，对应色域转换的原始类型和目标类型。
		`enum class CvtColorMode {`
		`  COLOR_YUVSP4202GRAY = 0`
		`};`
		若“mode”色域类型为“COLOR_BGR2YUVSP420”、“COLOR_RGB2YUVSP420”、“COLOR_RGB2YVUSP420”或COLOR_BGR2YVUSP420：
		● 建议输入的RGBTensor宽高为偶数，否则输出图片的边缘可能存在异常数据。
		● 输出张量的高为输入张量高的1.5倍，例如，输入高为4096，输出的高则为6144，超过部分DVPP接口的限制，使用时需要注意是否满足后续业务需求。

参数名	输入/输出	说明
keepMargin	输入	输出的outputTensor中是否保留Tensor中无效的边界区域。 当mode=COLOR_GRAY2RGB或mode=COLOR_RGBA2mRGBA时，该参数无效。
stream	输入	**AscendStream**类型，默认值为“AscendStream::DefaultStream()”。当参数值为默认值时，接口为同步操作，其他情况下，接口为异步操作。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.5 TensorFeatures

5.7.5.1 Sift

5.7.5.1.1 Sift

函数功能

Sift类构造函数，作为图像尺度不变特征转换的实现类，主要开放特征点提取及计算接口。目前仅支持Atlas 200I A2 加速模块（20 TOPS，12GB）。

函数原型

```
Sift(int nFeatures = 0, int nOctaveLayers = 3, double contrastThreshold = 0.04, double edgeThreshold = 10, double sigma = 1.6, int descriptorType = CV_32F);// 构造失败时，会抛出std::runtime_error异常
```

参数说明

参数名	输入/输出	说明
nFeatures	输入	提取特征点的排位，即对提取特征点排序，返回最佳的前“nFeatures”位（包含并列情况）。默认值为0，表示提取全部特征点。特征点识别数量需以实际返回结果为准。

参数名	输入/输出	说明
nOctaveLayers	输入	尺度空间每组图像中的中间层数。默认值为3，目前仅支持默认值，如配置其他参数构造将会失败并抛出std::runtime_error异常。 ● **nOctaveLayers + 3**为高斯金字塔每组图像中所包含的层数。 ● **nOctaveLayers + 2**为差分金字塔每组图像中所包含的层数。
contrastThreshold	输入	特征点过滤阈值。默认值为0.04，取值范围为[0.0, 20.0]。
edgeThreshold	输入	边缘效应过滤阈值。默认值为10， 取值范围为[0.0, 1000.0]。
sigma	输入	初始模糊尺度，高斯金字塔第0层图像的高斯滤波系数。默认值为1.6，目前仅支持默认值，如配置其他参数构造将会失败并抛出std::runtime_error异常。
descriptorType	输入	特征描述符的数据类型。默认值为CV_32F，目前支持以下数据类型。 ● CV_8U ● CV_32F

5.7.5.1.2 Init

函数功能

实现尺度空间构造的模型处理资源初始化。

函数原型

```
APP_ERROR Init(int32_t deviceId = 0);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	模型部署的芯片ID，默认为0。（目前仅支持0号芯片。）

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.5.1.3 DetectAndCompute

函数功能

实现图像特征点的提取及计算。

函数原型

```
APP_ERROR DetectAndCompute(Tensor _image, Rect _mask, std::vector<cv::KeyPoint> &keyPoints,
cv::OutputArray descriptors, bool useProvidedKeyPoints);
```

参数说明

参数名	输入/输出	说明
_image	输入	Tensor类，特征提取的输入图片。 ● Tensor维度支持HWC（三维）。 ● 不可为空，第三维**C**（通道数）为1，仅支持单通道。 ● 支持1280 * 720输入图片尺寸。
_mask	输入	**Rect**类，输入图片掩码，用于限制需要计算特征的区域，针对该区域内的图像进行特征提取。 “_mask”所表示的掩码矩形框需放置在图片内，Rect类对应的左上角坐标及右下角坐标需在图片有效范围内。
keyPoints	输入/输出	提取的特征点列表。
descriptors	输出	生成的描述子列表。
useProvidedKeyPoints	输入	是否提供特征点列表，即该函数仅进行描述子生成。 ● 参数为“true”时，接口将会根据keyPoints传入的特征点列表生成描述子列表。 ● 参数为“false”时，接口将会根据Sift算法提取特征点列表并根据列表生成描述子列表。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.6 VideoEncoder

5.7.6.1 类说明

VideoEncoder类，作为视频编码类，主要开放视频编码接口。

说明

VideoEncoder类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

5.7.6.2 VideoEncoder

函数功能

VideoEncoder类的构造函数。若因内存不足，或无法识别芯片等构造失败的场景会抛出 std::runtime_error 异常。

函数原型

```
VideoEncoder(const VideoEncodeConfig& vEncodeConfig, const int32_t deviceId = 0,
             const uint32_t channelId = 0);
```

参数说明

参数名	输入/输出	说明
vEncodeConfig	输入	视频编码的参数（以结构体的形式传入）。 视频编码的最大分辨率（宽 * 高）不能超过 4096 * 2304。
deviceId	输入	视频编码器部署的芯片，默认为 0 号芯片。 取值范围：[0, *识别到的芯片个数* - 1]。
channelId	输入	视频编码器的视频流索引。 ● Atlas 200/500 A2 推理产品：只能构造一个VideoEncoder，因此用户无需设置 channelId，重复构造会导致失败。

5.7.6.3 ~VideoEncoder

函数功能

VideoEncoder类的默认析构函数。

函数原型

```
~VideoEncoder()
```

5.7.6.4 Encode

函数功能

VideoEncoder的视频编码接口。

函数原型

```
APP_ERROR Encode(const Image &inputImage, const uint32_t frameId, void* userData);
```

参数说明

参数名	输入/输出	说明
inputImage	输入	输入编码前的FrameImage结构体。
frameId	输入	输入视频帧索引。
userData	输入/输出	输入用户自定义数据（用于传入用户自定义回调函数内，主要用来获取编码结果）。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.7 VideoDecoder

5.7.7.1 类说明

VideoDecoder类，作为视频解码类，主要开放视频解码接口。

在用户传入预申请解码内存场景下，需要先析构 VideoDecoder 类，再对预申请的内存进行释放（建议采用智能指针管理 VideoDecoder，智能指针的reset方法便可以提前进行析构）。

说明

VideoDecoder类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

5.7.7.2 VideoDecoder

函数功能

VideoDecoder类的构造函数。若因内存不足，或无法识别芯片等构造失败的场景会抛出std::runtime_error异常。

函数原型

```
VideoDecoder(const VideoDecodeConfig& vDecodeConfig, const int32_t deviceId = 0,
        const uint32_t channelId = 0);
```

参数说明

参数名	输入/输出	说明
vDecodeConfig	输入	视频解码的参数（以结构体的形式传入）。
deviceId	输入	视频解码器部署的芯片，默认为 0 号芯片。 取值范围：[0, *识别到的芯片个数*-1]。
channelId	输入	视频解码器的视频流索引。 ● Atlas 200/500 A2 推理产品：取值范围：[0, 31]。 若存在重复构造相同 channelId的场景，则会构造失败。

5.7.7.3 ~VideoDecoder

函数功能

VideoDecoder类的默认析构函数。

函数原型

```
~VideoDecoder()
```

5.7.7.4 Decode

函数功能

VideoDecoder的视频解码接口。

一个实例化的解码器在首次调用Decode接口时，会判断是否为**预申请场景**。使用预申请场景时，用户需要在后续调用Decode接口进行预申请输出内存，防止接口调用失败。

函数原型

```
APP_ERROR Decode(const std::shared_ptr<uint8_t> data, const uint32_t dataSize,
                 const uint32_t frameId, void* userData);
APP_ERROR Decode(const std::shared_ptr<uint8_t> data, const uint32_t dataSize, const uint32_t frameId,
                 Image& preMallocData, void* userData);
```

参数说明

参数名	输入/输出	说明
data	输入	输入解码前的数据内存地址。
dataSize	输入	输入解码前的数据内存大小。
frameId	输入	输入解码前视频帧的帧ID，输出的帧顺序跟发帧顺序不一定一致。
userData	输入/输出	用户自定义数据（用于传入用户自定义回调函数内，主要用来获取解码结果）。
preMallocData	输入	用户需要使用预申请内存场景时需要通过申请好的内存地址，内存大小，图像宽高，设备ID及图像格式 来构造Image类。

说明

“frameId”采用显示序进行输出与解码序（输出的帧顺序与发帧顺序一致）不同，其输出的帧顺序与发帧的顺序不一定一致，但与实际画面的显示一致。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.7.7.5 Flush

函数功能

发送完所有码流后，获取缓存数据并清空缓存队列，接口调用完成后将结束解码过程。

由于视频解码内部通过异步实现，请使用该接口后保留一定等待时间，用于回调函数获取并处理缓存数据。

函数原型

```
APP_ERROR Flush();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.8 模型推理

5.8.1 Model

5.8.1.1 类说明

Model类，作为模型的抽象，持有模型推理的资源，并主要开放推理接口。支持使用ATC工具构建的动态Batch、动态分辨率和分档动态维度模型的推理。

说明

Model类涉及申请Device侧资源，与MxDeInit的作用域冲突，因此，其作用域不能大于或等于MxDeInit的作用域。

5.8.1.2 Model

函数功能

Model类的构造函数。

函数原型

```
Model(std::string &modelPath, const int32_t deviceId = 0) // 构造失败时，会抛出std::runtime_error异常
Model(const Model&) = delete   // 请勿拷贝构造
Model &operate=(const Model&) = delete  // 请勿赋值构造
Model(ModelLoadOptV2 &mdlLoadOpt, const int32_t deviceId = 0);// 构造失败时，会抛出std::runtime_error
异常
```

参数说明

参数名	输入/输出	说明
modelPath	输入	模型的路径。最大仅支持至**4G**的模型且建议模型属主为当前用户，模型文件的权限小于或等于“640”。（对于MindIR模型，仅支持静态Shape和动态Batch。）

参数名	输入/输出	说明
deviceId	输入	模型部署的芯片，默认为0号芯片。（-1表示模型部署在Host侧，为保留字段，请勿使用。）
mdlLoadOpt	输入	选择以特定模型输入方式，输入推理模型。具体请参见**5.4.1.1 ModelLoadOptV2**。

5.8.1.3 ~Model

函数功能

Model类的默认析构函数。

函数原型

```
~Model()
```

5.8.1.4 Infer

函数功能

Model的推理接口。

Model推理提供对使用ATC工具构建的动态Batch、动态分辨率、动态维度模型的支持。

使用动态shape模型推理时，输入Tensor必须满足模型构建时的设置，若输入Tensor的“shape”不满足模型构建时的档位，会提示**设置动态Batch/分辨率/维度信息失败**。具体错误码请参见返回参数“APP_ERROR”说明。

例如：动态Batch模型构建时设置Batch档位为“2,4,8”，输入“Batch”为“1”的Tensor，推理报错“Dynamic batch set failed, modelId = 1, index = 1, dynamicBatchSize = 1”。

说明

对于只加载一次的模型，内部相关联资源唯一，因此不能在多线程中并发推理。多线程使用时需要每个线程加载一次模型，再调用推理业务。

函数原型

```
APP_ERROR Infer(std::vector<Tensor>& inputTensors, std::vector<Tensor>& outputTensors, AscendStream
&stream = AscendStream::DefaultStream())
// 由用户构造outputTensors，自行使用TensorMalloc()申请输出的Tensor内存,自由度更高
```

```
std::vector<Tensor> Infer(std::vector<Tensor>& inputTensors)
// 内部申请输出内存，并将推理的输出Tensor返回给用户
```

参数说明

参数名	输入/输出	说明
inputTensors	输入	模型需要的Tensor输入。
outputTensors	输出	模型的Tensor输出。
stream	输入	输入用于异步执行的Stream，默认为“AscendStream::DefaultStream()”，表示创建一个默认流（即同步执行）。

返回参数说明

数据结构	说明
std::vector<Tensor>	模型的Tensor输出。
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.8.1.5 GetInputTensorNum

函数功能

获得模型的输入个数。

函数原型

```
uint32_t GetInputTensorNum()
```

返回参数说明

数据结构	说明
uint32_t	模型的输入个数。

5.8.1.6 GetOutputTensorNum

函数功能

获得模型的输出个数。

函数原型

```
uint32_t GetOutputTensorNum()
```

返回参数说明

数据结构	说明
uint32_t	模型的输出个数。

5.8.1.7 GetInputTensorShape

函数功能

获得模型输入的对应Tensor的数据shape信息。

动态模型场景下对应的动态维度查询结果为-1。如果要使用查询的结果直接传入Tensor构造函数构造Tensor，需要将int64_t数据转换为uint32_t数据。

函数原型

```
std::vector<int64_t> GetInputTensorShape(uint32_t index = 0)
```

参数说明

参数名	输入/输出	说明
index	输入	模型输入的索引（第index个输入）。

返回参数说明

数据结构	说明
std::vector<int64_t>	对应输入的Tensor的shape信息。

5.8.1.8 GetOutputTensorShape

函数功能

获得模型输出的对应Tensor的数据shape信息。查询的结果可直接传入Tensor构造函数用来构造Tensor。

函数原型

```
std::vector<uint32_t> GetOutputTensorShape(uint32_t index = 0)
```

参数说明

参数名	输入/输出	说明
index	输入	模型输出的索引（第index个输出）。

返回参数说明

数据结构	说明
std::vector<uint32_t>	对应输出的Tensor的shape信息。

5.8.1.9 GetInputTensorDataType

函数功能

获得模型输入的对应Tensor的数据类型信息。

函数原型

```
MxBase::TensorDType GetInputTensorDataType(uint32_t index = 0)
```

参数说明

参数名	输入/输出	说明
index	输入	模型输入的索引（第index个输入）。

返回参数说明

数据结构	说明
MxBase::TensorDType	Tensor的数据类型，具体请参见**5.4.1.11 TensorDType**。

5.8.1.10 GetOutputTensorDataType

函数功能

获得模型输出的对应Tensor的数据类型信息。

函数原型

```
MxBase::TensorDType GetOutputTensorDataType(uint32_t index = 0)
```

参数说明

参数名	输入/输出	说明
index	输入	模型输出的索引（第index个输出）。

返回参数说明

数据结构	说明
MxBase::TensorDType	Tensor的数据类型，具体请参见**5.4.1.11 TensorDType**。

5.8.1.11 GetInputFormat

函数功能

获得模型输入的数据组织形式(NHWC 或者 NCHW)。

函数原型

```
MxBase::VisionDataFormat GetInputFormat()
```

返回参数说明

数据结构	说明
MxBase::VisionDataFormat	模型的输入数据组织形式，具体请参见**5.4.1.12 VisionDataFormat**。

5.8.1.12 GetDynamicGearInfo

函数功能

获得模型支持的动态输入档位信息，支持动态Batch、动态分辨率和分档动态维度模型。

函数原型

```
std::vector<std::vector<uint64_t>> GetDynamicGearInfo()
```

返回参数说明

数据结构	说明
std::vector<std::vector<uint64_t>>	模型支持的输入Tensor档位信息。 ● **std::vector<std::vector<uint64_t>>**为所有档位信息。 ● **std::vector<uint64_t>**为某一档的具体值。 返回数据结构为空，表示获取分档信息失败或者模型不属于函数功能中描述的三种模型之一，具体的原因可查看报错信息确认。 例如，模型为动态分辨率模型，**std::vector<uint64_t>**为某一档的高、宽值。

5.9 异步调用

5.9.1 AscendStream

须知

通过AscendStream创建的Stream，不能直接通过AsecndCL接口进行操作，否则会导致异常。

5.9.1.1 公共参数

参数名	数据类型	参数说明
stream	void*	Stream对应的指针，默认值为“nullptr”。
isDefault	bool	声明当前创建的Stream是否为默认Stream，默认值为“false”（即非默认）。

5.9.1.2 AscendStream

函数功能

AscendStream类的构造函数。

函数原型

```
explicit AscendStream(int32_t deviceId = 0);
```

参数说明

参数名	输入/输出	说明
deviceId	输入	指定Stream对应的设备ID。

5.9.1.3 ~AscendStream

函数功能

AscendStream类的默认析构函数。

函数原型

```
~AscendStream();
```

5.9.1.4 CreateAscendStream

函数功能

在当前进程或线程中创建一个Stream，同步接口。

- Atlas 200/500 A2 推理产品暂不支持。

函数原型

```
APP_ERROR CreateAscendStream();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.9.1.5 DestroyAscendStream

函数功能

销毁一个Stream。

函数原型

```
APP_ERROR DestroyAscendStream();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.9.1.6 CreateChannel

函数功能

创建一个Stream的通道，最多仅支持创建255个，如超出该限制将会出现异常。

函数原型

```
APP_ERROR CreateChannel();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.9.1.7 GetChannel

函数功能

获取指定Stream的通道号。

函数原型

```
APP_ERROR GetChannel(int* channelId);
```

参数说明

参数名	输入/输出	说明
channelId	输出	Stream对应的通道号。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.9.1.8 Synchronize

函数功能

阻塞应用程序运行，直到指定Stream中的所有任务都完成，同步接口。

函数原型

```
APP_ERROR Synchronize() const;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.9.1.9 DefaultStream

函数功能

在当前进程创建一个默认Stream，静态对象，不支持异步调用。使用前需初始化设备并调用5.5.12.7 SetDevice指定具体设备ID。

函数原型

```
static AscendStream &DefaultStream();
```

返回参数说明

数据结构	说明
AscendStream	返回AscendStream类。

5.9.1.10 GetDeviceId

函数功能

获取Stream所在设备ID（deviceId）。

函数原型

```
int32_t GetDeviceId() const;
```

返回参数说明

数据结构	说明
int32_t	Stream所在的设备ID。

5.9.1.11 SetErrorCode

函数功能

记录异步任务中产生的错误码。

接口将会记录第一次以及最后一次产生的错误码，可通过5.9.1.12 GetErrorCode接口获取。

函数原型

```
void SetErrorCode(APP_ERROR errCode);
```

参数说明

参数名	输入/输出	说明
errCode	输入	错误码，记录单次产生的错误码。

5.9.1.12 GetErrorCode

函数功能

获取并返回已记录的错误码，以pair（对）形式返回。

与5.9.1.11 SetErrorCode配合使用，需在调用5.9.1.8 Synchronize之后使用。

函数原型

```
std::pair<APP_ERROR, APP_ERROR> GetErrorCode();
```

返回参数说明

数据结构	说明
std::pair<APP_ERROR, APP_ERROR>	返回已记录的错误码对，其中第一个字段为第一次记录的错误码，第二个字段为最后记录的错误码。

5.9.1.13 LaunchCallback

函数功能

提供异步回调功能。

需在调用5.9.1.4 CreateAscendStream之后使用。

函数原型

```
APP_ERROR LaunchCallBack(aclrtCallback fn, void* userData)
```

参数说明

参数名	输入/输出	说明
fn	输入	指定要增加的回调函数。回调函数的函数原型为参见如下。 `typedef void (*aclrtCallback)(void *userData)`
userData	输入	回调函数所需的参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.9.1.14 AddTensorRefPtr

函数功能

提供异步回调功能。

需在调用5.9.1.4 CreateAscendStream之后使用。

将输入Tensor挂载到AscendStream对象上，延长输入Tensor的生命周期。当调用5.9.1.8 Synchronize或者5.9.1.5 DestroyAscendStream时，AscendStream对象会释放被挂载的Tensor。

函数原型

```
void AddTensorRefPtr(const Tensor& inputTensor);
```

参数说明

参数名	输入/输出	说明
inputTensor	输入	Tensor类，需要挂载到AscendStream对象上的张量。

5.10 模型后处理

5.10.1 模型后处理类参考（tensorinfer 框架）

5.10.1.1 ImagePreProcessInfo

5.10.1.1.1 ImagePreProcessInfo

函数功能

类构造函数。

函数原型

```
ImagePreProcessInfo();
ImagePreProcessInfo(uint32_t width, uint32_t height);
ImagePreProcessInfo(uint32_t widthResize, uint32_t heightResize, uint32_t widthOriginal, uint32_t
heightOriginal);
```

参数说明

参数名	输入/输出	说明
width	输入	图像宽。
height	输入	图像高。
widthResize	输入	缩放宽。
heightResize	输入	缩放高。
widthOriginal	输入	原图宽。
heightOriginal	输入	原图高。

5.10.1.2 PostProcessBase

5.10.1.2.1 PostProcessBase

函数功能

类构造函数。

函数原型

```
PostProcessBase() = default;
PostProcessBase(const PostProcessBase& other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	PostProcessBase对象。

5.10.1.2.2 ~PostProcessBase

函数功能

PostProcessBase类的默认析构函数。

函数原型

```
virtual ~PostProcessBase() = default;
```

5.10.1.2.3 operator=

函数功能

等号运算符重载，用于PostProcessBase对象之间的赋值。

函数原型

```
PostProcessBase& operator = (const PostProcessBase& other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	PostProcessBase对象。

5.10.1.2.4 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() virtual APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>>
&postConfig);  //已废弃
virtual APP_ERROR Init(const std::map<std::string, std::string> &postConfig);
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.2.5 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
virtual APP_ERROR DeInit();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.2.6 GetCurrentVersion

函数功能

返回当前使用的后处理so的整型版本号。（5.10.1.10 postprocess中所有后处理子类都具备此接口。）

函数原型

```
virtual uint64_t GetCurrentVersion();
```

返回参数说明

数据结构	说明
uint64_t	当前使用的后处理so的整型版本号。

5.10.1.2.7 LoadConfigDataAndLabelMap

须知

该函数已废弃，请勿使用。

函数功能

保护成员函数，用于初始化后处理的配置文件和标签文件。

函数原型

```
APP_ERROR LoadConfigDataAndLabelMap(const std::map<std::string, std::shared_ptr<void>> &postConfig);
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.2.8 LoadConfigData

函数功能

保护成员函数，用于初始化后处理的配置文件。

函数原型

```
APP_ERROR LoadConfigData(const std::map<std::string, std::string> &postConfig);
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.2.9 CheckAndMoveTensors

函数功能

保护成员函数，调用IsValidTensors校验张量无误后将张量内存搬运至Host侧。

函数原型

```
APP_ERROR CheckAndMoveTensors(std::vector<TensorBase> &tensors);
```

参数说明

参数名	输入/输出	说明
tensors	输入	输入张量。

返回参数说明

数据结构	说明
APP_ERROR	校验结果。

5.10.1.2.10 GetBuffer

函数功能

保护成员函数，获取输入张量中的指定索引的指针。

函数原型

```
void* GetBuffer(const TensorBase& tensor, uint32_t index) const;
```

参数说明

参数名	输入/输出	说明
tensors	输入	输入张量。
index	输入	待获取张量索引。

返回参数说明

数据结构	说明
void*	输入张量中的指定索引的指针。

5.10.1.2.11 JudgeResizeType

函数功能

用于判断图片信息中resize类型是否合法。

函数原型

bool JudgeResizeType(const ResizedImageInfo& resizedImageInfo);

参数说明

参数名	输入/输出	说明
resizedImageInfo	输入	输入的图片信息。

返回参数说明

数据结构	说明
bool	图片信息中包含的resize类型是否合法。

5.10.1.3 ImagePostProcessBase

5.10.1.3.1 类说明

图像类任务后处理的基类，继承自**5.10.1.2 PostProcessBase**。

5.10.1.3.2 SetCropRoiBoxes

函数功能

用于设置当前后处理的输入图像在原图中的位置。

函数原型

```
void SetCropRoiBoxes(std::vector<MxBase::CropRoiBox> cropRoiBoxes)
```

参数说明

参数名	输入/输出	说明
cropRoiBoxes	输入	当前后处理的输入图像（vector）在原图中的位置。

5.10.1.4 ObjectPostProcessBase

5.10.1.4.1 类功能

目标检测类任务后处理的基类，继承自**5.10.1.3 ImagePostProcessBase**。

5.10.1.4.2 ObjectPostProcessBase

函数功能

类构造函数。

函数原型

```
ObjectPostProcessBase() = default;
ObjectPostProcessBase(const ObjectPostProcessBase & other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	ObjectPostProcessBase对象。

5.10.1.4.3 ~ObjectPostProcessBase

函数功能

ObjectPostProcessBase类的默认析构函数。

函数原型

```
virtual ~ObjectPostProcessBase() = default;
```

5.10.1.4.4 operator=

函数功能

等号运算符重载，用于ObjectPostProcessBase对象之间的赋值。

函数原型

```
ObjectPostProcessBase& operator = (const ObjectPostProcessBase & other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	ObjectPostProcessBase对象。

5.10.1.4.5 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() virtual APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>>
&postConfig) override;  //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.4.6 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.4.7 Process

函数功能

接受模型推理输出张量，输出目标检测结果。后续执行时，需要先执行5.10.1.4.5 Init方法。

函数原型 1

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
                          std::vector<std::vector<ObjectInfo>>& objectInfos,
                          const std::vector<ResizedImageInfo>& resizedImageInfos = {},
                          const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明 1

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
objectInfos	输出	输出目标检测的坐标、类型、置信度等信息。
resizedImageInfos	输入	图片信息（包括原图和缩放后宽高）。
configParamMap	输入	其他配置参数，可选。

返回参数说明 1

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

函数原型 2

```
virtual SDK_DEPRECATED_FOR() APP_ERROR Process(const std::vector<TensorBase>& tensors,
                std::vector<std::vector<ObjectInfo>>& objectInfos,
                const std::vector<ImagePreProcessInfo>& imagePreProcessInfos);//已废弃
```

参数说明 2

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
objectInfos	输出	输出目标检测的坐标、类型、置信度等信息。
imagePreProcessInfos	输入	图片前处理信息。

返回参数说明 2

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.4.8 CoordinatesReduction

函数功能

保护成员函数，用于对坐标数据进行缩放。（该方法为内部调用方法，请用户不要使用。）

函数原型 1

```
void CoordinatesReduction(const uint32_t index, const ResizedImageInfo& resizedImageInfo,
                std::vector<ObjectInfo>& objInfos, bool normalizedFlag = true);
```

参数说明 1

参数名	输入/输出	说明
index	输入	指定切分向量的索引。
resizedImageInfo	输入	图片信息（包括原图和缩放后宽高）。

参数名	输入/输出	说明
objInfos	输出	输出目标检测的坐标、类型、置信度等信息。
normalizedFlag	输入	是否进行标准化，可选。

函数原型 2

```
void CoordinatesReduction(std::vector<std::vector<ObjectInfo>> &objInfos,
        const std::vector<MxBase::ImagePreProcessInfo> &imagePreProcessInfos,
        bool normalizedFlag = true);
```

参数说明 2

参数名	输入/输出	说明
objectInfos	输出	输出目标检测的坐标、类型、置信度等信息。
imagePreProcessInfos	输入	图片前处理信息。
normalizedFlag	输入	是否进行标准化，可选。

5.10.1.4.9 ResizeReduction

函数功能

保护成员函数，用于对目标检测数据进行缩放。（该方法为内部调用方法，请用户不要使用。）

函数原型

```
void ResizeReduction(const ResizedImageInfo& resizedImageInfo, const int imgWidth, const int imgHeight,
                     std::vector<ObjectInfo>& objInfos);
```

参数说明

参数名	输入/输出	说明
resizedImageInfo	输入	图片信息（包括原图和缩放后的宽高）。
imgWidth	输入	图片宽度。
imgHeight	输入	图片高度。
objInfos	输出	输出目标检测的坐标、类型、置信度等信息。

5.10.1.4.10 LogObjectInfos

函数功能

保护成员函数，仅用于debug，打印目标检测的坐标、置信度等信息。（该方法为内部调用方法，请用户不要使用。）

函数原型

```
void LogObjectInfos(const std::vector<std::vector<ObjectInfo>>& objectInfos);
```

参数说明

参数名	输入/输出	说明
objInfos	输入	目标检测的坐标、类型、置信度等信息。

5.10.1.4.11 GetSeparateScoreThresh

函数原型

```
APP_ERROR GetSeparateScoreThresh(std::string& strSeparateScoreThresh);
```

参数说明

参数名	输入/输出	说明
strSeparateScoreThresh	输入	未分割的原始字符串。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.4.12 GetObjectConfigData

函数功能

保护成员函数，将从私有成员中获取原始的结果并调用GetSeparateScoreThresh()进行分割。（该方法为内部调用方法，请用户不要使用。）

函数原型

```
APP_ERROR GetObjectConfigData();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.5 ClassPostProcessBase

5.10.1.5.1 ClassPostProcessBase

函数功能

类构造函数。

函数原型

```
ClassPostProcessBase() = default;
ClassPostProcessBase(const ClassPostProcessBase &other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	ClassPostProcessBase对象。

5.10.1.5.2 ~ClassPostProcessBase

函数功能

ClassPostProcessBase类的默认析构函数。

函数原型

```
virtual ~ClassPostProcessBase() = default;
```

5.10.1.5.3 operator=

函数功能

等号运算符重载，用于ClassPostProcessBase对象之间的赋值。

函数原型

```
ClassPostProcessBase& operator=(const ClassPostProcessBase &other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	ClassPostProcessBase对象。

5.10.1.5.4 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>> &postConfig)
override;    //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.5.5 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.5.6 Process

函数功能

接受模型推理输出张量，输出分类结果。

函数原型

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
    std::vector<std::vector<ClassInfo>> &classInfos,
                        const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
classInfos	输出	输出分类结果。
configParamMap	输入	其他配置参数，可选。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.6 SemanticSegPostProcessBase

5.10.1.6.1 类功能

语义分割类任务后处理的基类，继承自**5.10.1.3 ImagePostProcessBase**。

5.10.1.6.2 SemanticSegPostProcessBase

函数功能

类构造函数。

函数原型

```
SemanticSegPostProcessBase() = default;
SemanticSegPostProcessBase(const SemanticSegPostProcessBase& other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	SemanticSegPostProcessBase对象。

5.10.1.6.3 ~SemanticSegPostProcessBase

函数功能

SemanticSegPostProcessBase类的默认析构函数。

函数原型

```
virtual ~SemanticSegPostProcessBase() = default;
```

5.10.1.6.4 operator=

函数功能

等号运算符重载，用于SemanticSegPostProcessBase对象之间的赋值。

函数原型

```
SemanticSegPostProcessBase& operator=(const SemanticSegPostProcessBase& other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	SemanticSegPostProcessBase对象。

5.10.1.6.5 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>> &postConfig)
override;    //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.6.6 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.6.7 Process

函数功能

接受模型推理输出张量，输出语义分割结果。后续执行时，需要先执行5.10.1.6.5 Init方法。

函数原型

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
    std::vector<SemanticSegInfo>& semanticSegInfos,
                        const std::vector<ResizedImageInfo>& resizedImageInfos = {},
                        const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
semanticSegInfos	输出	输出语义分割像素值，类别映射表等信息。
resizedImageInfos	输入	图片信息（包括原图和缩放后宽高）。
configParamMap	输入	其他配置参数，可选。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.6.8 CoordinatesReduction

函数功能

保护成员函数，根据图片信息，输出降维后的语义分割信息。（该方法为内部调用方法，请用户不要使用。）

函数原型

```
void CoordinatesReduction(const ResizedImageInfo& resizedImageInfo, SemanticSegInfo&
semanticSegInfos);
```

参数说明

参数名	输入/输出	说明
resizedImageInfo	输入	图片信息。图片原始宽高和缩放宽高不能大于8192，缩放比例范围[1/32, 16]。
semanticSegInfos	输出	语义分割像素值，类别映射表等信息。

5.10.1.6.9 GetSemanticSegConfigData

函数功能

保护成员函数，将配置信息写入类别映射表中。（该方法为内部调用方法，请用户不要使用。）

函数原型

```
APP_ERROR GetSemanticSegConfigData();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.7 TextGenerationPostProcessBase

5.10.1.7.1 TextGenerationPostProcessBase

函数功能

类构造函数。

函数原型

```
TextGenerationPostProcessBase() = default;
TextGenerationPostProcessBase(const TextGenerationPostProcessBase& other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	TextGenerationPostProcessBase对象。

5.10.1.7.2 ~TextGenerationPostProcessBase

函数功能

TextGenerationPostProcessBase类的默认析构函数。

函数原型

```
virtual ~TextGenerationPostProcessBase() = default;
```

5.10.1.7.3 operator=

函数功能

等号运算符重载，用于TextGenerationPostProcessBase对象之间的赋值。

函数原型

```
TextGenerationPostProcessBase& operator = (const TextGenerationPostProcessBase& other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	TextGenerationPostProcessBase对象。

5.10.1.7.4 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>> &postConfig)
override;    //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.7.5 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.7.6 Process

函数功能

接受模型推理输出张量，输出文本生成结果。后续执行时，需要先执行Init()方法。

函数原型

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
    std::vector<TextsInfo>& textsInfos,
                    const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
textsInfos	输出	输出文本生成结果。
configParamMap	输入	其他配置参数，可选。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.8 TextObjectPostProcessBase

5.10.1.8.1 类功能

文本目标检测类任务后处理的基类，继承自**5.10.1.3 ImagePostProcessBase**。

5.10.1.8.2 TextObjectPostProcessBase

函数功能

类构造函数。

函数原型

```
TextObjectPostProcessBase() = default;
TextObjectPostProcessBase(const TextObjectPostProcessBase& other) = default;
```

参数说明

参数名	输入/输出	说明
other	输入/输出	TextObjectPostProcessBase对象。

5.10.1.8.3 ~TextObjectPostProcessBase

函数功能

TextObjectPostProcessBase类的默认析构函数。

函数原型

```
virtual ~TextObjectPostProcessBase() = default;
```

5.10.1.8.4 operator =

函数功能

等号运算符重载，用于TextObjectPostProcessBase对象之间的赋值。

函数原型

```
TextObjectPostProcessBase& operator= (const TextObjectPostProcessBase& other);
```

参数说明

参数名	输入/输出	说明
other	输入/输出	TextObjectPostProcessBase对象。

5.10.1.8.5 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>> &postConfig)
override;    //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.8.6 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.8.7 Process

函数功能

接受模型推理输出张量，输出目标检测结果。后续执行时，需要先执行Init()方法。

函数原型

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
    std::vector<std::vector<TextObjectInfo>> &textObjectInfos,
                        const std::vector<ResizedImageInfo>& resizedImageInfos = {},
                        const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
textObjectInfos	输出	输出文本目标框的坐标、置信度等信息。
resizedImageInfos	输入	图片信息（包括原图和缩放后宽高）。
configParamMap	输入	其他配置参数，可选。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.8.8 ResizeReduction

函数功能

保护成员函数，用于对输入的图片信息进行缩放。该方法为内部调用方法，请用户不要使用。

函数原型

```
void ResizeReduction(const ResizedImageInfo &resizedImageInfo, TextObjectInfo &textObjInfo);
```

参数说明

参数名	输入/输出	说明
textObjectInfo	输出	输出文本目标框的坐标等信息。
resizedImageInfo	输入	图片信息。

5.10.1.8.9 FixCoords

函数功能

保护成员函数，用于对坐标数据进行修复。该方法为内部调用方法，请用户不要使用。

函数原型

```
void FixCoords(uint32_t scrData, float &desData);
```

参数说明

参数名	输入/输出	说明
scrData	输入	原始坐标数据。
desData	输出	被修复的坐标数据。

5.10.1.9 KeypointPostProcessBase

5.10.1.9.1 KeypointPostProcessBase

函数功能

类构造函数。

函数原型

```
KeypointPostProcessBase() = default;
KeypointPostProcessBase(const KeypointPostProcessBase &other) = default;
```

参数说明

参数	输入/输出	说明
other	输入/输出	KeypointPostProcessBase对象。

5.10.1.9.2 ~KeypointPostProcessBase

函数功能

KeypointPostProcessBase类的默认析构函数。

函数原型

```
virtual ~KeypointPostProcessBase() = default;
```

5.10.1.9.3 operator=

函数功能

等号运算符重载，用于KeypointPostProcessBase对象间的赋值。

函数原型

```
KeypointPostProcessBase& operator = (const KeypointPostProcessBase &other);
```

参数说明

参数	输入/输出	说明
other	输入/输出	KeypointPostProcessBase对象。

5.10.1.9.4 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
SDK_DEPRECATED_FOR() APP_ERROR Init(const std::map<std::string, std::shared_ptr<void>> &postConfig)
override;  //已废弃
APP_ERROR Init(const std::map<std::string, std::string> &postConfig) override;
```

参数说明

参数名	输入/输出	说明
postConfig	输入	配置参数。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.9.5 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.9.6 Process

函数功能

接受模型推理输出张量，输出关键点结果。后续执行时，需要先执行5.10.1.9.4 Init方法。

函数原型

```
virtual APP_ERROR Process(const std::vector<TensorBase>& tensors,
                         std::vector<std::vector<KeyPointDetectionInfo>>& keyPointInfos,
                         const std::vector<ResizedImageInfo>& resizedImageInfos = {},
                         const std::map<std::string, std::shared_ptr<void>> &configParamMap = {});
```

参数说明

参数名	输入/输出	说明
tensors	输入	模型后处理的输入张量。
keyPointInfos	输出	输出关键点结果。
resizedImageInfos	输入	图片信息（包括原图和缩放后宽高）。
configParamMap	输入	其他配置参数，可选。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.1.9.7 LogKeyPointInfos

函数功能

保护成员函数，仅用于debug，打印图片关键点。该方法为内部调用方法，请用户不要使用。

函数原型

```
void LogKeyPointInfos(const std::vector<std::vector<KeyPointDetectionInfo>>& keyPointInfos);
```

参数说明

参数名	输入/输出	说明
keyPointInfos	输入	图片关键点向量。

5.10.1.9.8 GetSeparateScoreThresh

函数功能

保护成员函数，将原始结果分割为float向量类型。该方法为内部调用方法，请用户不要使用。

函数原型

```
APP_ERROR GetSeparateScoreThresh(std::string& strSeparateScoreThresh);
```

参数说明

参数名	输入/输出	说明
strSeparateScoreThresh	输入	未分割的原始字符串。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.1.10 postprocess

5.10.1.10.1 Yolov3PostProcess

YOLOv3模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase中的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::Yolov3PostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.2 SsdMobilenetv1FpnPostProcess

SsdMobilenetv1FpnPostProcess模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::SsdMobilenetv1FpnPostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.3 Ssdvgg16PostProcess

Ssdvgg16PostProcess模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::Ssdvgg16PostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.4 FasterRcnnPostProcess

FasterRcnnPostProcess模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::FasterRcnnPostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.5 MaskRcnnMindsporePost

MaskRcnnMindsporePost模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::MaskRcnnMindsporePost> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.6 SsdMobilenetFpnMindsporePost

SsdMobilenetFpnMindsporePost模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号。重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::SsdMobilenetv1FpnPostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.7 Resnet50PostProcess

Resnet50PostProcess模型后处理类，继承自分类任务后处理基类**5.10.1.5 ClassPostProcessBase**。

重载了ClassPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.5 ClassPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::Resnet50PostProcess> GetClassInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.8 Deeplabv3Post

Deeplabv3Post模型后处理类，继承自语义分割后处理基类**5.10.1.6 SemanticSegPostProcessBase**。

重载了SemanticSegPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.6 SemanticSegPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::Deeplabv3Post> GetSemanticSegInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.9 UNetMindSporePostProcess

UNetMindSporePostProcess模型后处理类，继承自语义分割后处理基类**5.10.1.6 SemanticSegPostProcessBase**。

重载了SemanticSegPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.6 SemanticSegPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::UNetMindSporePostProcess> GetSemanticSegInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.10 TransformerPostProcess

TransformerPostProcess模型后处理类，继承自文本生成后处理基类**5.10.1.7 TextGenerationPostProcessBase**。

重载了TextGenerationPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.7 TextGenerationPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::TransformerPostProcess> GetTextGenerationInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.11 CrnnPostProcess

CrnnPostProcess模型后处理类，继承自文本生成后处理基类**5.10.1.7 TextGenerationPostProcessBase**。

重载了TextGenerationPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.7 TextGenerationPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::CrnnPostProcess> GetTextGenerationInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.12 CtpnPostProcess

CtpnPostProcess模型后处理类，继承自文本生成后处理基类**5.10.1.8 TextObjectPostProcessBase**。

重载了TextObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.8 TextObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::CtpnPostProcess> GetTextObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.13 PSENetPostProcess

PSENetPostProcess模型后处理类，继承自文本生成后处理基类**5.10.1.8 TextObjectPostProcessBase**。

重载了TextObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.8 TextObjectPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::PSENetPostProcess> GetTextObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.14 OpenPosePostProcess

OpenPosePostProcess模型后处理类，继承自关键点后处理基类**5.10.1.9 KeypointPostProcessBase**。

重载了KeypointPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.9 KeypointPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::OpenPosePostProcess> GetKeypointInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.15 HigherHRnetPostProcess

HigherHRnetPostProcess模型后处理类，继承自关键点后处理基类**5.10.1.9 KeypointPostProcessBase**。

重载了KeypointPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.9 KeypointPostProcessBase**。

重载了PostProcessBase的**5.10.1.2.6 GetCurrentVersion**接口，用以获取当前后处理so的版本号，重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::HigherHRnetPostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.1.10.16 RetinaNetPostProcess

RetinaNetPostProcess模型后处理类，继承自目标检测后处理基类**5.10.1.4 ObjectPostProcessBase**。

重载了ObjectPostProcessBase的Init()、DeInit()和Process()接口，具体描述请参见**5.10.1.4 ObjectPostProcessBase**。重载了赋值运算符“=”。

实现了**std::shared_ptr<MxBase::RetinaNetPostProcess> GetObjectInstance()**方法用于获取该类的智能指针实例。

5.10.2 模型后处理类参考（modelinfer 框架）

5.10.2.1 Resnet50PostProcessor

5.10.2.1.1 类说明

该类继承了ModelPostProcessorBase用于ResNet-50模型后处理，用户可以根据需要继承该类并选择性实现Init、DeInit、Process等虚函数。

5.10.2.1.2 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
APP_ERROR Init(const std::string& configPath, const std::string& labelPath, MxBase::ModelDesc modelDesc)
override;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.1.3 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.1.4 Process

函数功能

用于推理后处理，并获取类型相关信息。

函数原型

```
APP_ERROR Process(std::vector<std::shared_ptr<void>>& featLayerData, int& argmaxIndex, float&
confidence);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	特征数据数组。
argmaxIndex	输出	最可能类的索引。
confidence	输出	最可能类的置信度。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2 ModelPostProcessorBase

5.10.2.2.1 类说明

该类用于实现推理后处理，用户需要继承该类并实现Init、DeInit、Process等虚函数。

5.10.2.2.2 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
virtual APP_ERROR Init(const std::string& configPath, const std::string& labelPath, MxBase::ModelDesc modelDesc) = 0;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.3 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
virtual APP_ERROR DeInit() = 0;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.4 Process

函数功能

用于推理后处理，获取目标、类型等信息。

函数原型

```
virtual APP_ERROR Process(std::vector<std::shared_ptr<void>>& featLayerData);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	特征数据数组。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.5 MemoryDataToHost

函数功能

把数据从Device侧拷贝到Host侧用于后处理。

函数原型

```
APP_ERROR MemoryDataToHost(const int index, const std::vector<std::vector<MxBase::BaseTensor>>&
tensors,std::vector<std::shared_ptr<void>>& featLayerData);
```

参数说明

参数名	输入/输出	说明
index	输入	输出池索引。
tensors	输入	输出Tensor数组。
featLayerData	输出	输出特征数据数组。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.6 GetLabelName

函数功能

根据索引获取类名称。

函数原型

```
std::string GetLabelName(int index);
```

参数说明

参数名	输入/输出	说明
index	输入	类索引。

返回参数说明

数据结构	说明
std::string	类名称。

5.10.2.2.7 LoadConfigDataAndLabelMap

函数功能

保护成员函数，该接口由Init()调用，用于实现配置参数的获取。

函数原型

```
APP_ERROR LoadConfigDataAndLabelMap(const std::string& configPath, const std::string& labelPath);
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.8 GetModelTensorsShape

函数功能

保护成员函数，该接口由Init()调用，用于获取模型的输入输出张量形状。

函数原型

```
APP_ERROR GetModelTensorsShape(MxBase::ModelDesc modelDesc);
```

参数说明

参数名	输入/输出	说明
modelDesc	输入	模型基本信息结构体。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.2.9 CheckModelCompatibility

函数功能

保护成员函数，该接口由Init()调用，用于对所读取模型的输出形状进行校验。

函数原型

```
APP_ERROR CheckModelCompatibility();
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.3 ObjectPostProcessorBase

5.10.2.3.1 类说明

该类继承了**ModelPostProcessorBase**用于目标检测推理后处理，用户可以根据需要继承该类并选择性实现Init、DeInit、Process等虚函数。

5.10.2.3.2 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
APP_ERROR Init(const std::string& configPath, const std::string& labelPath, MxBase::ModelDesc modelDesc)
override;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。

参数名	输入/输出	说明
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.3.3 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.3.4 Process

函数功能

从输出数据中获取检测目标的信息。

函数原型

```
virtual APP_ERROR Process(std::vector<std::shared_ptr<void>>& featLayerData, std::vector<ObjDetectInfo>&
objInfos,const bool useMpPictureCrop, MxBase::PostImageInfo postImageInfo);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	输出特征数据数组。

参数名	输入/输出	说明
objInfos	输出	输出对象坐标、类型、置信度等信息。 输出单个对象的坐标范围为[0, 8192]，类型不能小于0。
useMpPictureCrop	输入	是否还原坐标到目标框坐标上。
postImageInfo	输入	图片信息（包括原图和缩放宽高，目标框坐标）。原图和缩放宽高均不能超过8192，目标框坐标范围为[0, 8192]。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.3.5 SetAspectRatioImageInfo

函数功能

设置后处理图像信息。

函数原型

```
void SetAspectRatioImageInfo(const MxBase::AspectRatioPostImageInfo& postProcessorImageInfo);
```

参数说明

参数名	输入/输出	说明
postProcessorImageInfo	输入	后处理图像信息。

5.10.2.3.6 ObjectDetectionOutput

函数功能

保护成员函数，由Process()调用，用于对模型输出进行目标处理，如去重、排序、筛选等。该方法为内部调用方法，请用户不要使用。

函数原型

```
virtual void ObjectDetectionOutput(std::vector<std::shared_ptr<void>>& featLayerData,
                                   std::vector<ObjDetectInfo>& objInfos, ImageInfo& imgInfo);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	输出特征数据数组。
objInfos	输出	输出对象坐标、类型、置信度等信息。
imgInfo	输入	图片信息。

5.10.2.3.7 CoordinatesReduction

函数功能

保护成员函数，用于对坐标数据进行缩放。该方法为内部调用方法，请用户不要使用。

函数原型

```
void CoordinatesReduction(ImageInfo& imgInfo, std::vector<ObjDetectInfo>& objInfos);
```

参数说明

参数名	输入/输出	说明
objInfos	输出	输出目标检测的坐标、类型、置信度等信息。
imageInfo	输入	图片信息（包括原图和缩放后的宽高）。

5.10.2.3.8 GetSeparateScoreThresh

函数功能

保护成员函数，将原始结果分割为float向量类型。该方法为内部调用方法，请用户不要使用。

函数原型

```
APP_ERROR GetSeparateScoreThresh(std::string& strSeparateScoreThresh);
```

参数说明

参数名	输入/输出	说明
strSeparateScoreThresh	输入	未分割的原始字符串。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.2.3.9 NmsSort

函数功能

保护成员函数，根据置信度大小过滤重复目标。该方法为内部调用方法，请用户不要使用。

函数原型

```
void NmsSort(std::vector<ObjDetectInfo>& objInfos, float iouThresh, IOUMethod method = UNION);
```

参数说明

参数名	输入/输出	说明
objInfos	输出	待过滤的目标边角点。
iouThresh	输入	过滤阀值。
method	输入	IOU计算方式： ● MAX 重叠区域除以两者中面积大的。 ● MIN 重叠区域除以两者中面积小的。 ● UNION 重叠区域除以两者面积并集。 ● DIOU 重叠区域除以两者面积并集减去距离的交并比。

5.10.2.4 FasterRcnnPostProcessor

5.10.2.4.1 类说明

该类继承了**ModelPostProcessorBase**用于FasterRcnn模型后处理，用户可以根据需要继承该类并选择性实现Init、DeInit、Process等虚函数。

5.10.2.4.2 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
APP_ERROR Init(const std::string &configPath, const std::string &labelPath, MxBase::ModelDesc modelDesc)
override;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.4.3 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.4.4 Process

函数功能

从输出数据中获取检测目标的信息。

函数原型

```
APP_ERROR Process(std::vector<std::shared_ptr<void>> &featLayerData, std::vector<ObjDetectInfo>
&objInfos,const bool useMpPictureCrop, MxBase::PostImageInfo postImageInfo);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	输出特征数据数组。
objInfos	输出	输出对象坐标、类型、置信度等信息。
useMpPictureCrop	输入	是否还原坐标到目标框坐标上。
postImageInfo	输入	图片信息（包括原图和缩放宽高，目标框坐标）。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.5 Yolov3PostProcessor

5.10.2.5.1 类说明

该类继承了**ModelPostProcessorBase**用于YOLOv3模型后处理，用户可以根据需要继承该类并选择性实现Init、DeInit、Process等虚函数。

5.10.2.5.2 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
APP_ERROR Init(const std::string &configPath, const std::string &labelPath, MxBase::ModelDesc modelDesc)
override;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。
modelDesc	输出	模型描述信息。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.2.5.3 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考5.5.21.6 APP_ERROR说明。

5.10.2.5.4 Process

函数功能

从输出数据中获取检测目标的信息。

函数原型

```
APP_ERROR Process(std::vector<std::shared_ptr<void>>& featLayerData, std::vector<ObjDetectInfo>&
objInfos,const bool useMpPictureCrop, MxBase::PostImageInfo postImageInfo);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	输出特征数据数组。
objInfos	输出	输出对象坐标、类型、置信度等信息。
useMpPictureCrop	输入	是否还原坐标到目标框坐标上。
postImageInfo	输入	图片信息（包括原图和缩放宽高，目标框坐标）。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.5.5 SetPoolSize

函数功能

设置后处理输入张量的形状为模型输出张量的形状。当前为内部使用类，请用户不要使用。

函数原型

```
APP_ERROR SetPoolSize(const std::vector<std::vector<int>>& outputTensorShapes);
```

参数说明

参数名	输入/输出	说明
outputTensorShapes	输入	模型输出张量的形状。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.6 CtpnPostProcessor

5.10.2.6.1 类说明

该类继承了**ModelPostProcessorBase**用于Ctpn模型后处理, 用户可以根据需要继承该类并选择性实现Init、DeInit、Process等虚函数。

5.10.2.6.2 CtpnPostProcessor

函数功能

类构造函数，初始化类成员变量。

函数原型

```
CtpnPostProcess();
```

5.10.2.6.3 ~CtpnPostProcessor

函数功能

CtpnPostProcessor类的默认析构函数。

函数原型

```
virtual ~CtpnPostProcessor();
```

5.10.2.6.4 Init

函数功能

用于完成模型后处理初始化。

函数原型

```
APP_ERROR Init(const std::string& configPath, const std::string& labelPath, MxBase::ModelDesc modelDesc)
override;
```

参数说明

参数名	输入/输出	说明
configPath	输入	配置文件路径。
labelPath	输入	标签文件路径。
modelDesc	输出	模型描述信息。

5.10.2.6.5 DeInit

函数功能

用于模型后处理的去初始化，完成资源释放。

函数原型

```
APP_ERROR DeInit() override;
```

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

5.10.2.6.6 Process

函数功能

从输出数据中获取检测目标的信息。

函数原型

```
APP_ERROR Process(std::vector<std::shared_ptr<void>> &featLayerData, std::vector<TextObjDetectInfo>
&textObjInfos, const bool useMpPictureCrop, MxBase::PostImageInfo postImageInfo);
```

参数说明

参数名	输入/输出	说明
featLayerData	输入	输出特征数据数组。
textObjInfos	输出	输出对象坐标、置信度等信息。
useMpPictureCrop	输入	是否还原坐标到目标框坐标上。
postImageInfo	输入	图片信息（包括原图和缩放宽高，目标框坐标）。 原始和缩放宽高不能超过8192，目标框坐标不能小于0且不能超过8192。

返回参数说明

数据结构	说明
APP_ERROR	程序执行返回的错误码，请参考**5.5.21.6 APP_ERROR说明**。

6 API 参考（Python）

6.1 使用须知
6.2 依赖列表说明
6.3 Python枚举类型及数据类
6.4 初始化
6.5 媒体数据处理
6.6 模型推理
6.7 模型后处理
6.8 日志输出

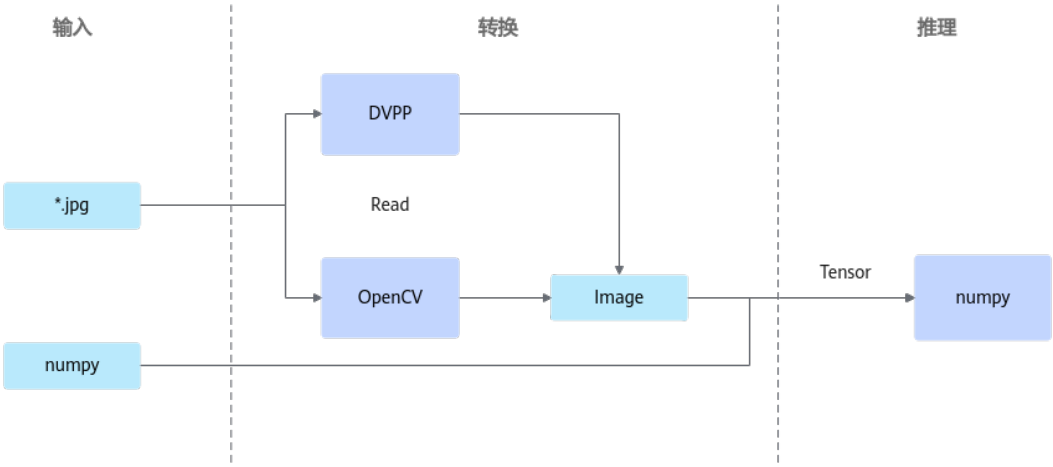
6.1 使用须知

须知

本章节开放的类，请遵循同一个类实例在单一线程中使用的原则，请勿将同一个类实例在不同的线程中使用。

Python 推理数据流程图

图 6-1 推理数据流图



6.2 依赖列表说明

表 6-1 依赖列表

依赖名称	用途
Tensor	与Numpy数据换转，Host/Device换转，Python模型推理的输入输出统一结构。
Image	是DVPP输入输出统一结构，能转换为Tensor。
Model	用于模型加载推理
ImageProcessor	图像处理类，主要开放图像编解码、缩放和抠贴图等接口。
VideoDecoder	视频解码类，主要开放视频解码接口。
dvpp	含DVPP图像解码读取和缩放功能，即将废弃，建议使用ImageProcessor类。
log	用于不同级别的日志打印。
post	包含多种模型后处理类（ResNet-50、YoloV3等）。

依赖使用方法如下：

```
from mindx.sdk import base
from mindx.sdk.base import Image, Model, ImageProcessor, Size, post
```

6.3 Python 枚举类型及数据类

6.3.1 通用枚举类型/数据类

6.3.1.1 基本数据枚举类型

表 6-2 数据格式枚举类型

数据类型	说明
base.undefined	未定义类型。
base.int8	int8类型。
base.uint8	uint8类型。
base.int16	int16类型。
base.uint16	uint16类型。

数据类型	说明
base.int32	int32类型。
base.uint32	uint32类型。
base.int64	int64类型。
base.uint64	uint64类型。
base.float16	float16类型。
base.float32	float32类型。
base.double	double类型。
base.bool	bool类型。

6.3.1.2 ModelLoadType 模型加载方式枚举类型

表 6-3 模型加载方式

参数名	说明
base.load_model_from_file	从文件加载离线模型数据，由系统内部管理内存。
base.load_model_from_mem	从内存加载离线模型数据，由系统内部管理内存。
base.load_model_from_file_with_mem	从文件加载离线模型数据，由用户自行管理模型运行的内存。
base.load_model_from_mem_with_mem	从内存加载离线模型数据，由用户自行管理模型运行的内存。

6.3.1.3 ModelType 模型枚举类型

表 6-4 模型类型

参数名	说明
base.model_type_om	OM模型。
base.model_type_mindir	MindIR模型（对于MindIR模型，仅支持静态Shape和动态Batch。）。

6.3.1.4 VisionDataFormat 图像数据格式排布枚举类型

表 6-5 图像数据格式排布形式

排布形式	说明
NCHW	图像数据按NCHW格式排布。
NHWC	图像数据按NHWC格式排布。

6.3.1.5 image_format 图片格式枚举类型

表 6-6 图像格式类型

内存类型	说明
base.yuv_400	YUV_400 的图像格式。
base.nv12	YUV_SP_420 的图像格式。
base.nv21	YVU_SP_420 的图像格式。
base.yuv_sp_422	YUV_SP_422 的图像格式。
base.yvu_sp_422	YVU_SP_422 的图像格式。
base.yuv_sp_444	YUV_SP_444 的图像格式。
base.yvu_sp_444	YVU_SP_444 的图像格式。
base.yuyv_packed_422	YUYV_PACKED_422 的图像格式。
base.uyvy_packed_422	UYVY_PACKED_422 的图像格式。
base.yvyu_packed_422	YVYU_PACKED_422 的图像格式。
base.vyuy_packed_422	VYUY_PACKED_422 的图像格式。
base.yuv_packed_444	YUV_PACKED_444 的图像格式。
base.bgr	BGR_888的图像格式。
base.rgb	RGB_888的图像格式。
base.argb	ARGB_8888 的图像格式。
base.abgr	ABGR_8888 的图像格式。
base.rgba	RGBA_8888 的图像格式。
base.bgra	BGRA_8888 的图像格式。

6.3.1.6 Resize 缩放枚举类型

表 6-7 Resize 参数类型说明

参数类型	说明
base.resize_ms_keep_ratio	ResizeType.RESIZER_MS_KEEP_ASPECT_RATIO。
base.resize_stretching	ResizeType.RESIZER_STRETCHING。
base.resize_tf_keep_ratio	ResizeType.RESIZER_TF_KEEP_ASPECT_RATIO。

6.3.1.7 Interpolation 缩放方式格式枚举类型

表 6-8 缩放方式格式类型

参数名	说明
base.huaweiu_high_order_filter	华为自研的高阶滤波算法。
base.bilinear_similar_opencv	业界通用的 Bilinear 算法（与 OpenCV 算法的计算过程类似）。
base.nearest_neighbor_opencv	业界通用的 Nearest neighbor 算法（与 OpenCV 算法的计算过程类似）。
base.bilinear_similar_tensorflow	业界通用的 Bilinear 算法（与 TensorFlow 算法的计算过程类似）。
base.nearestneighbor_tensorflow	业界通用的 Nearest neighbor 算法（与 TensorFlow 算法的计算过程类似）。

6.3.1.8 BorderType 补边方式枚举类型

表 6-9 补边方式格式类型

参数名	说明
base.border_constant	添加有颜色的常数值边界。
base.border_replicate	重复最后一个元素。 例如：aaaaaa|a*****h|hhhhhhh（其中*表示任意图像元素）。
base.border_reflect	边界元素的镜像，镜像包括边界元素。 例如：ba|abc*******fgh|hg（其中*表示任意图像元素）。

参数名	说明
base.border_reflect_101	边界元素的镜像，镜像不包括边界元素。 例如：cb|abc****fgh|gf（其中*表示任意图像元素）。

6.3.1.9 StreamFormat 视频流数据格式枚举类型

表 6-10 视频流数据格式类型

参数名	说明
base.h265_main_level	**H.265** 格式视频流，**主流画质**。
base.h264_baseline_level	**H.264** 格式视频流，**基本画质**。
base.h264_main_level	**H.264** 格式视频流，**主流画质**。
base.h264_high_level	**H.264** 格式视频流，**高级画质**。

6.3.1.10 ModelLoadOptV2 类

须知

请根据实际情况选择对应配置，如配置与实际输入存在差异，会在**6.6.1.2 Model(option, deviceId)**处抛出**RuntimeError**。

表 6-11 模型加载选项

属性名	类型	说明
modelType	ModelType 枚举类型	推理模型类型，支持情况参见如下。（默认为“base.model_type_om”。） ● base.model_type_om ● base.model_type_mindir（对于MindIR模型，仅支持静态Shape和动态Batch。）

属性名	类型	说明
loadType	ModelLoadType枚举类型	推理模型输入类型，可通过以下参数选定输入方式。（默认为“base.load_model_from_file”。） ● base.load_model_from_file：从文件加载离线模型数据，由系统内部管理内存。 ● base.load_model_from_file_with_mem：从文件加载离线模型数据，由用户自行管理模型运行的内存（包括工作内存和权值内存，工作内存用于模型执行过程中的临时数据，权值内存用于存放权值数据）。 ● base.load_model_from_mem：从内存加载离线模型数据，由系统内部管理内存。 ● base.load_model_from_mem_with_mem：从内存加载离线模型数据，由用户自行管理模型运行的内存（包括工作内存和权值内存）。 ● 当“modelType”为“base.model_type_mindir”时，仅支持“base.load_model_from_file”和“base.load_model_from_mem”两种输入类型。
modelPath	str	推理模型文件路径，仅在以下模式生效。（默认为""，最大只支持至4GB大小的模型且要求模型属主为当前用户，模型文件的权限应小于或等于“640”。） ● base.load_model_from_file ● base.load_model_from_file_with_mem
modelPtr	int	推理模型所在内存地址，仅在以下模式生效。（默认为0，用户需根据实际情况输入内存地址。） ● base.load_model_from_mem ● base.load_model_from_mem_with_mem
modelSize	int	推理模型数据长度，单位Byte，仅在以下模式生效。（默认为0，最大只支持至4GB大小的模型。） ● base.load_model_from_mem ● base.load_model_from_mem_with_mem
modelWorkPtr	int	推理模型所在工作内存地址，仅在以下模式生效。（默认为0，表示由系统管理内存。） ● base.load_model_from_file_with_mem ● base.load_model_from_mem_with_mem
modelWeightPtr	int	推理模型权值内存地址，仅在以下模式生效。（默认为0，表示由系统管理内存。） ● base.load_model_from_file_with_mem ● base.load_model_from_mem_with_mem
workSize	int	推理模型所在工作内存大小，单位Byte。（默认为0，当“modelWorkPtr”为0时无效。）

属性名	类型	说明
weightSize	int	推理模型权值内存大小，单位Byte。（默认为0，当“modelWeightPtr”为0时无效。）

6.3.1.11 AttributeInfo 类

表 6-12 AttributeInfo 类属性说明

属性名	类型	说明
attrId	int	属性ID。
attrName	str	属性名。
attrValue	str	属性值。
confidence	float	置信度。

6.3.1.12 ClassInfo 类

表 6-13 ClassInfo 类属性说明

属性名	类型	说明
classId	int	类ID。
className	str	类名。
confidence	float	置信度。

6.3.1.13 ObjectInfo 类

表 6-14 ObjectInfo 类属性说明

属性名	类型	说明
classId	float	类ID。
className	str	类名。
confidence	float	置信度。
mask	list	mask列表。
x0	float	左上角x坐标。
x1	float	右下角x坐标。

属性名	类型	说明
y0	float	左上角y坐标。
y1	float	右下角y坐标。

6.3.1.14 ResizedImageInfo 类

表 6-15 ResizedImageInfo 类属性说明

属性名	类型	说明
heightOriginal	int	原图的高度。
heightResize	int	resize后的高度。
resizeType	ResizeType	resize类型。
widthOriginal	int	原图的宽度。
widthResize	int	resize后的宽度。

6.3.1.15 SemanticSegInfo 类

表 6-16 SemanticSegInfo 类属性说明

属性名	类型	说明
pixels	list	像素集。
labelMap	list	labelMap。

6.3.1.16 TextObjectInfo 类

表 6-17 TextObjectInfo 类属性说明

属性名	类型	说明
confidence	float	置信度。
result	str	结果。
x0	float	第一检测点的x坐标。
x1	float	第二检测点的x坐标。
x2	float	第三检测点的x坐标。
x3	float	第四检测点的x坐标。

属性名	类型	说明
y0	float	第一检测点的y坐标。
y1	float	第二检测点的y坐标。
y2	float	第三检测点的y坐标。
y3	float	第四检测点的y坐标。

6.3.1.17 TextsInfo 类

表 6-18 TextsInfo 类属性说明

属性名	类型	说明
text	数组	text内容。

6.3.1.18 KeyPointDetectionInfo 类

表 6-19 KeyPointDetectionInfo 类属性说明

属性名	类型	说明
keyPointMap	dict	每个关键点的坐标等信息。
scoreMap	dict	各每个关键点对应的置信度。
score	float	整体置信度。

6.3.1.19 ResizeType 类

表 6-20 ResizeType 类属性说明

属性名	类型	说明
RESIZER_MS_KEEP_ASPECT_RATIO	ResizeType 枚举类型。	基于MindSpore的等比率缩放。
RESIZER_STRETCHING	ResizeType 枚举类型。	拉伸缩放。
RESIZER_TF_KEEP_ASPECT_RATIO	ResizeType 枚举类型。	基于TensorFlow的等比率缩放。

6.3.1.20 dtype 类

属性名	对应类型
dtype.int8	int8类型。
dtype.uint8	uint8类型。
dtype.int16	int16类型。
dtype.uint16	uint16类型。
dtype.int32	int32类型。
dtype.uint32	uint32类型。
dtype.int64	int64类型。
dtype.uint64	uint64类型。
dtype.float16	float16类型。
dtype.float32	float32类型。
dtype.double	double类型。
dtype.bool	bool类型。
dtype.undefined	未定义类型。

6.3.1.21 image.format 类

表 6-21 image.format 类属性说明

属性名	类型
image.format.nv12	NV12格式类型。
image.format.nv21	NV12格式类型。
image.format.bgr	BGR888格式类型。
image.format.rgb	RGB888格式类型。

6.3.1.22 type 类

表 6-22 type 类属性说明

属性名	类型
type.memory_host	Host侧内存类型。
type.memory_device	Device侧内存类型。

属性名	类型
type.memory_dvpp	DVPP内存类型。

6.3.1.23 BufferInput 类

表 6-23 BufferInput 类属性说明

属性名	类型	说明
data	str	图片数据。
frame_info	str	帧信息。
vision_info	str	图片信息。

6.3.1.24 MetadataInput 类

属性名	类型	说明
data_source	str	源字段。
data_type	str	protobuf数据类型。
serialized_metadata	str	protobuf数据的序列化表示。

6.3.1.25 DataOutput 类

属性名	类型	说明
error_code	int	错误码。
error_msg	str	错误信息。
buffer_output	BufferOutput	一般类型的输出结果。
metadata_list	List[MetadataOutput]	protobuf类型的输出结果。
set_error_info(int, str)	函数，无返回值	设置错误信息。

6.3.1.26 BufferOutput 类

属性名	类型	说明
error_code	int	错误码。
error_msg	str	错误信息。
data	str	输出结果。
get_byte_data()	函数，返回byte类型数据	该函数用来返回二进制数据，也可以返回正常的推理结果。

6.3.1.27 MetadataOutput 类

属性名	类型	说明
error_code	int	错误码。
error_msg	str	错误信息。
data_type	str	protobuf数据类型。
serialized_metadata	str	protobuf数据的序列化表示。
set_error_info(int, str)	函数，无返回值	设置错误信息。

6.3.1.28 MxDataInput 类

属性名	类型	说明
data	str	图片数据。
fragmentId	int	分块ID。
customParam	str	自定义的参数。
roiBoxs	List[float]	裁剪框坐标数组，[x0,y0,x1,x1]。

6.3.1.29 MxDataOutput 类

属性名	类型	说明
errorCode	int	返回的错误码，0为成功。
dataSize	int	数据大小。
data	str	推理结果。

6.3.1.30 Color 类

参数	类型	说明
Channel_zero	int	0通道取值，范围[0, 255]。
Channel_one	int	1通道取值，范围[0, 255]。
Channel_two	int	2通道取值，范围[0, 255]。

6.3.1.31 Dim 类

参数	类型	说明
left	int	左侧补边像素数量。
right	int	右侧补边像素数量。
top	int	上方补边像素数量。
bottom	int	下方补边像素数量。

6.3.1.32 Rect 类

参数	类型	说明
x0	int	矩形框左上角坐标的**横坐标**。
y0	int	矩形框左上角坐标的**纵坐标**。
x1	int	矩形框右下角坐标的**横坐标**。
y1	int	矩形框右下角坐标的**纵坐标**。

6.3.1.33 Size 类

参数	类型	说明
width	int	图像宽。
height	int	图像高。

6.3.1.34 Point 类

参数	类型	说明
x	int	横坐标（以图像左上角为原点）。
y	int	纵坐标（以图像左上角为原点）。

6.3.1.35 VideoDecodeConfig 类

表 6-24 保存视频解码的类

参数名	类型	说明
width	int	视频解码缓冲区的宽，取值范围：[128, 4096]，默认值：1920，必须为偶数。并且会自动对齐到16（向上对齐）。 请用户根据实际的视频帧数据进行适当设置，需大于或等于实际的视频帧数据宽，否则会无解码输出，设置过大将会产生多余的内存资源。
height	int	视频解码缓冲区的高，取值范围：[128, 4096]，默认值：1080，必须输入偶数。 请用户根据实际的视频帧数据进行适当设置，需大于或等于实际的视频帧数据高，否则会无解码输出，设置过大将会产生多余的内存资源。
inputVideoFormat	StreamFormat枚举类	输入的视频格式，H.264格式视频支持base.h264_baseline_level、base.h264_main_level、base.h264_high_level三种格式，H.265格式视频只支持base.h265_main_level格式。默认值为：base.h264_main_level。
outputImageFormat	image_format枚举类	解码后输出的图像格式，默认值为：base.nv12。 ● Atlas 200/500 A2 推理产品环境只支持base.nv12、base.nv21两种格式。
skipInterval	int	视频解码的跳帧参数，取值范围：[0, 250]，默认值：0。

参数名	类型	说明
cscMatrix	int	色域转换选项，默认值为“0”（即HI_CSC_MATRIX_BT601_WIDE）。 ● HI_CSC_MATRIX_BT601_WIDE = 0，基于BT601 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.000 0.000 1.402 | | Y - 0 | # | G | = | 1.000 -0.344 -0.714 | * | U - 128 | # | B | | 1.000 1.772 0.000 | | V - 128 | # RGB转YUV： # | Y | |-0.5 | | 0.299 0.587 0.114 | | R | # | U | = |127.5 | + | -0.168 -0.331 0.500 | * | G | # | V | |127.5 | | 0.500 -0.419 -0.081 | | B | ● HI_CSC_MATRIX_BT601_NARROW =1，基于BT601 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.59602 | | Y - 16 | # | G | = | 1.16438 -0.39176 -0.81297 | * | U - 128 | # | B | | 1.16438 2.01723 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.25679 0.51564 0.10014 | | R | # | U | = |128| + | -0.14491 -0.29099 0.43922 | * | G | # | V | |128| | 0.42941 -0.36779 -0.07143 | | B | ● HI_CSC_MATRIX_BT709_WIDE =2，基于BT709 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.57480 | | Y - 0 | # | G | = | 1.00000 -0.18732 -0.46812 | * | U - 128 | # | B | | 1.00000 1.85560 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.21260 0.71520 0.07220 | | R | # | U | = |128| + | -0.11457 -0.38543 0.50000 | * | G | # | V | |128| | 0.50000 -0.45415 -0.04585 | | B | ● HI_CSC_MATRIX_BT709_NARROW =3，基于BT709 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.79274 | | Y - 16 | # | G | = | 1.16438 -0.21325 -0.53291 | * | U - 128 | # | B | | 1.16438 2.11240 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.18259 0.62825 0.06342 | | R | # | U | = |128| + | -0.09840 -0.33857 0.43922 | * | G | # | V | |128| | 0.42941 -0.39894 -0.04027 | | B | ● HI_CSC_MATRIX_BT2020_WIDE =4，基于BT2020 wide标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.00000 0.00000 1.47460 | | Y - 0 | # | G | = | 1.00000 -0.16455 -0.57135 | * | U - 128 | # | B | | 1.00000 1.88140 0.00000 | | V - 128 | # RGB转YUV： # | Y | |0 | | 0.26270 0.67800 0.05930 | | R | # | U | = |128| + | -0.13963 -0.36037 0.50000 | * | G | # | V | |128| | 0.50000 -0.45979 -0.04021 | | B |

参数名	类型	说明
		● HI_CSC_MATRIX_BT2020_NARROW =5，基于BT2020 narrow标准的色域转换矩阵，各参数值参考如下。 # YUV转RGB： # | R | | 1.16438 0.00000 1.67868 | | Y - 16 | # | G | = | 1.16438 -0.18733 -0.65042 | * | U - 128 | # | B | | 1.16438 2.14177 0.00000 | | V - 128 | # RGB转YUV： # | Y | |16 | | 0.22564 0.59558 0.05209 | | R | # | U | = |128| + | -0.11992 -0.31656 0.43922 | * | G | # | V | |128| | 0.42941 -0.40389 -0.03533 | | B |

6.3.1.36 VideoEncodeConfig 类

参数名	类型	说明
width	int	编码后视频帧数据的宽，默认值：1920，必须与待编码图片的宽（original_width）一致。 ● Atlas 200/500 A2 推理产品，取值范围：[128, 1920]，输入的视频宽需要对齐16。
height	int	编码后视频帧数据的高，默认值：1080，必须与待编码图片的高（original_height）一致。 ● Atlas 200/500 A2 推理产品，取值范围：[128, 1920]，输入的视频高需要对齐2。
outputVideoFormat	StreamFormat枚举类	编码后输出的视频格式，默认值为：base.h264_main_level。
inputImageFormat	image_format枚举类	输入的图像格式，只支持base.nv12、base.nv21两种格式，默认值为：base.nv12。
keyFrameInterval	int	视频I帧间隔大小，默认为30，取值范围[1, 2000]。
srcRate	int	输入码流帧率，单位fps，默认为30。 ● Atlas 200/500 A2 推理产品，取值范围：[1, 120]。
rcMode	int	指定码率控制模式，默认值为0，取值如下： ● 0：默认模式，Atlas 200/500 A2 推理产品默认为CBR模式。 ● 1：变码率VBR模式。 ● 2：定码率CBR模式。
maxBitRate	int	输出码率，单位kbps，默认值为300。 ● Atlas 200/500 A2 推理产品，取值范围：[10, 30000]。

参数名	类型	说明
ipProp	int	一个GOP内单个I帧bit数和单个P帧bit数的比例，默认值为70，取值范围[1, 100]。

推荐配置

以下为视频采集场景下推荐的参数配置，可以保证较好的视频编码质量，其他场景下请根据实际应用对应调整。

画质/分辨率	srcRate	keyFrameInterval	maxBitRate
4K 3840 * 2160/4096 * 2160	25或30	50或60	● H.264： 8000 ~ 12000 ● H.265： 8000 ~ 12000
2K 2560 * 1440	25或30	50或60	● H.264： 6000 ~ 10000 ● H.265： 6000 ~ 10000
1080P（蓝光） 1920 * 1080	25或30	50或60	● H.264： 2000~6000 ● H.265： 1000~4000
720P（高清） 1280 * 720	25或30	50或60	● H.264： 1000~3000 ● H.265： 800~2000
480P/D1_N（标清） 854 * 480/720 * 480	25或30	50或60	● H.264： 600~1400 ● H.265： 300~700
576P/D1 （标清） 720 * 576	25或30	50或60	● H.264： 600~1400 ● H.265： 300~700
270P（流畅） 480 * 270	25或30	50或60	-
CIF P/N 352 * 288/320 * 240	25或30	50或60	● H.264： 250 ● H.265： 250

6.3.2 流程编排数据类

6.3.2.1 RoiBox

功能

裁剪框的定义。

结构定义

```
class RoiBox:
    def __init__(self):
        self.x0 = None
        self.y0 = None
        self.x1 = None
        self.y1 = None
```

参数说明

参数名	类型	说明
x0	float	左上角横坐标。
y0	float	左上角纵坐标。
x1	float	右下角横坐标。
y1	float	右下角纵坐标。

6.3.2.2 RoiBoxVector

功能

RoiBox的Vector类，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class RoiBoxVector:
    def push_back(self, x):
        pass
    def pop_back(self):
        pass
    def pop(self):
        pass
```

6.3.2.3 MxDataInput

功能

Stream接收的数据结构定义。

结构定义

```
class MxDataInput :
    def __init__(self):
        self.data = None
        self.fragmentId = None
        self.customParam = None
        self.roiBoxs = list()   # list
```

参数说明

参数名	类型	说明
data	bytes	图像数据。
fragmentId	int	分块ID，保留参数。
customParam	bytes	自定义的参数，保留参数。
roiBoxs	RoiBoxVector	裁剪框坐标数组。

6.3.2.4 MxDataOutput

功能

推理业务输出数据定义。

结构定义

```
class MxDataOutput:
    def __init__(self):
        self.errorCode = None
        self.data = None
```

参数说明

参数名	类型	说明
errorCode	int	输出错误码。
data	bytes	输出结果数据。

6.3.2.5 MxProtobufIn

功能

推理业务输入protobuf数据定义。

结构定义

```
class MxProtobufIn:
    def __init__(self):
        self.key= None
        self.type= None
        self.protobuf= None
```

参数说明

参数名	类型	说明
key	bytes	输入protobuf的key。

参数名	类型	说明
type	bytes	输入protobuf数据的类型。
protobuf	bytes	输入protobuf的value数据，需要反序列化为与类型对应的数据。

6.3.2.6 MxProtobufOut

功能

推理业务输出数据定义。

结构定义

```
class MxProtobufOut:
    def __init__(self):
        self.errorCode = None
        self.messageName= None
        self.messageBuf= None
```

参数说明

参数名	类型	说明
errorCode	int	输出错误码。
messageName	bytes	输出protobuf数据的key。
messageBuf	bytes	输出protobuf数据的value，需要反序列化为与类型对应的数据。

6.3.2.7 MxDataInputExt

功能

推理业务输出数据定义。

结构定义

```
class MxDataInputExt :
    def __init__(self):
        self.data = None
        self.fragmentId = None
        self.customParam = None
        self.roiBoxs = list()  # list
        self.mxpiFrameInfo = None
        self.mxpiVisionInfo = None
```

参数说明

参数名	类型	说明
data	bytes	图像数据。
fragmentId	int	分块ID，保留参数。
customParam	bytes	自定义的参数，保留参数。
roiBoxs	RoiBoxVector	裁剪框坐标数组。
mxpiFrameInfo	bytes	图片的frame信息序列化成字符串。
mxpiVisionInfo	bytes	图片信息序列化成字符串。

6.3.2.8 MxMetadataInput

功能

Stream接收的元数据的数据定义。

结构定义

```
class MxMetadataInput:
    def __init__(self):
        self.datasource = None
        self.dataType = None
        self.serializedMetadata = None
```

参数说明

参数名	类型	说明
datasource	bytes	挂载元数据时使用的dataSource，用于下游插件获取元数据。
dataType	bytes	元数据的数据类型。
serializedMetadata	bytes	序列化的元数据。

6.3.2.9 MxMetadataOutput

功能

Stream输出的元数据的数据定义。

结构定义

```
class MxMetadataOutput:
    def __init__(self):
        self.errorCode = None
        self.errorMsg = None
```

```
        self.dataType= None
        self.serializedMetadata = None
```

参数说明

参数名	类型	说明
errorCode	int	输出错误码。
errorMsg	bytes	输出错误信息。
dataType	bytes	元数据的数据类型。
serializedMetadata	bytes	序列化的元数据。

6.3.2.10 MxBufferAndMetadataOutput

功能

推理业务输出数据及元数据。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class StringVector:
    def __init__(self):
        self.errorCode = None
        self.errorMsg = None
        self.bufferOutput = None
        self.metadataVec = list()
    def SetErrorInfo(self, errorCodeIn, errorMsgIn):
        pass
```

参数说明

参数名	类型	说明
errorCode	int	输出错误码。
errorMsg	bytes	输出错误信息。
bufferOutput	bytes	输出结果数据。
metadataVec	vector	输出元数据信息。

6.3.2.11 StringVector

功能

用于存放String的向量类型，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class StringVector:
    def pop(self):
        pass
    def append(self, x):
        pass
    def size(self):
        pass
    ...
```

6.3.2.12 InProtobufVector

功能

用于存放MxProtobufIn的向量类型，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class InProtobufVector:
    def pop(self):
        pass
    def append(self, x):
        pass
    def size(self):
        pass
    ...
```

6.3.2.13 OutProtobufVector

功能

用于存放MxProtobufOut的向量类型，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class OutProtobufVector:
    def pop(self):
        pass
    def append(self, x):
        pass
    def size(self):
        pass
    ...
```

6.3.2.14 MetadataInputVector

功能

用于存放MxMetadataInput的向量类型，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class MetadataInputVector:
    def pop(self):
        pass
    def append(self, x):
        pass
    def size(self):
        pass
    ...
```

6.3.2.15 MetadataOutputVector

功能

用于存放MxMetadataOutput的向量类型，实现了Vector的通用操作。

结构定义

由swig生成，仅展示部分业务相关接口，具体请参考“StreamManagerApi.py”。

```
class MetadataOutputVector:
    def pop(self):
        pass
    def append(self, x):
        pass
    def size(self):
        pass
    ...
```

6.3.2.16 MxBufferInput

功能

Stream接收的数据定义。

结构定义

```
class MxBufferInput:
    def __init__(self):
        self.data = None
        self.mxpiFrameInfo = None
        self.mxpiVisionInfo = None
```

参数说明

参数名	类型	说明
data	bytes	输入的数据。
mxpiFrameInfo	bytes	输入Frame数据。
mxpiVisionInfo	bytes	输入图片数据。

6.3.2.17 MxBufferOutput

功能

Stream输出的数据定义。

结构定义

```
class MxBufferOutput:
    def __init__(self):
        self.errorCode = None
        self.errorMsg= None
        self.data = None
```

参数说明

参数名	类型	说明
errorCode	int	输出的错误码。
errorMsg	bytes	输出的错误信息。
data	bytes	输出的数据。

6.4 初始化

6.4.1 mx_init()

函数功能

全局初始化，申请设备资源与日志资源。

6.5 媒体数据处理

6.5.1 Tensor 类

6.5.1.1 Tensor

须知

多个Tensor对象按batch维进行concat操作时请使用base.batch_concat()函数，具体使用方式请参见**batch_concat(inputs)**。

函数功能

numpy数组转换为Tensor。

Tensor数据结构接收了作为参数（buffer）传进来的一段缓冲区内存地址，因此在**使用numpy数组进行数据预处理并转为Tensor**时，要注意预处理的过程中是否对内存进行修改。

例如：transpose()函数作用是调换数组的索引值，但是**该函数实际未对数据内存地址进行重新排列**，仅在获取numpy数组时返回转换后的值。如将该numpy数组转为Tensor，实际获得的是未经重新排列的Tensor。（即numpy数组为transpose()转置后数组，Tensor为transpose()转置前数组。）

如需使用例如transpose()等不改变内存排列的函数，可以在transpose()处理后使用如**numpy.ascontiguousarray()**等接口来重新排列内存数据，使得转换后的Tensor对象数据和期望得到的转置后数组一致。

函数原型

```
Tensor(buffer)
```

输入参数说明

参数名	类型	说明
buffer	numpy数组	待转换为Tensor的numpy数组。

输出参数说明

Tensor对象。

6.5.1.2 to_device

函数功能

移动Tensor到Device内存中。

函数原型

```
to_device(deviceId)
```

输入参数说明

参数名	类型	说明
deviceId	int	Device设备的ID号。

抛异常接口

分配Tensor内存失败及拷贝到Device侧失败，抛出runtime异常。

6.5.1.3 to_host

函数功能

移动Tensor到Host内存中。

函数原型

```
to_host()
```

抛异常接口

分配Tensor内存失败及拷贝到Host侧失败，抛出runtime异常。

6.5.1.4 Tensor 类的属性列表

属性名	说明	备注
device	Tensor所在Device的ID	-1表示运行在Host侧，1表示设备号1，2表示设备号2，以此类推。
dtype	Tensor数据类型	base.dtype数据类型。
shape	Tensor的维度信息	返回的是list数据类型。

6.5.2 Image 类

6.5.2.1 Image

函数原型

```
Image()
Image(ndarray b, image_format format, tuple(Size, Size) imageSizeInfo = DEFAULT_IMAGE_SIZE_INFO)
```

函数功能

- 构造一个空的Image对象。
- 从Host侧的numpy的ndarray转为Image对象。

输入参数说明

参数名	类型	说明
b	ndarray	构造Image的numpy数组，各元素类型为np.uint8且形状为3维，依次表示图片的HWC（高、宽、通道数）。

参数名	类型	说明
format	image_format	图像色域格式，支持以下色域类型。 ● base.yuv_400 ● base.rgb ● base.bgr ● base.rgba ● base.bgra ● base.argb ● base.abgr “b”（输入ndarray）形状中的通道数（C）需要与色域类型一致，各色域类型对应通道数参见如下。 ● yuv_400为1。 ● bgr、rgb为3。 ● rgba、bgra、argb、abgr为4。
imageSizeInfo	tuple	图像**有效宽高**与**对齐后宽高**的组合，有效宽、高应不超过对齐后的宽、高，输入方式参见如下。 tuple(*有效宽高*，*对齐后的宽高*) ● 有效宽、高取值范围为[6, 8192]。 ● 对齐后宽、高取值范围为[16, 8192]，其中宽为16的倍数、高为2的倍数。 ● “b”（输入ndarray）形状中的高宽需要与有效宽高和对齐宽高其中之一保持一致。 ● 默认值为“DEFAULT_IMAGE_SIZE_INFO”，其中**有效宽高**与**对齐后宽高**均为(0, 0)，若该参数为默认值，则自动根据“b”的形状获取有效宽高和对齐宽高。

说明

1. 从opencv读取图片转换的ndarray，若形状为WHC（宽，高，通道数），需要进行转置为HWC（高，宽，通道数）之后进行构造。
2. 从get_tensor/to_tensor/get_orginal_tensor接口获取的tensor，转换为ndarray后，形状为NHWC（个数，高，宽，通道数）或NHW（个数，高，宽），需要根据实际情况截取或扩展到HWC之后进行构造。

输出参数说明

Image对象。

6.5.2.2 to_tensor

函数功能

将Image对象转换为Tensor类对象。

函数原型

```
to_tensor()
```

输出参数说明

返回Tensor。

6.5.2.3 get_tensor

函数功能

获取Tensor类对象。

函数原型

```
get_tensor()
```

输出参数说明

返回Tensor。

6.5.2.4 get_original_tensor

函数功能

获取图片有效范围的Tensor类对象。

支持以下图片色域类型。

- base.yuv_400
- base.rgb
- base.bgr
- base.rgba
- base.bgra
- base.argb
- base.abgr

如需将图片转换为ndarray时，请将图片在Host侧通过该接口转为Tensor，再转为ndarray。

函数原型

```
get_original_tensor()
```

输出参数说明

返回Tensor。

6.5.2.5 to_device

函数功能

搬移image到Device内存中。

函数原型

```
to_device(deviceId)
```

输入参数说明

参数名	类型	说明
deviceId	int	Device设备的ID号。

抛异常接口

调用Tensor的to_device函数失败，抛出runtime异常。

6.5.2.6 to_host

函数功能

搬移image到Host内存中。

函数原型

```
to_host()
```

抛异常接口

调用Tensor的to_host函数失败，抛出runtime异常。

6.5.2.7 serialize

函数功能

将图像内存数据及元数据序列化后落盘保存为文件。

函数原型

```
serialize(filePath, forceOverwrite = False);
```

参数说明

参数名	输入/输出	说明
filePath	输入	序列化后的数据文件保存路径（包括文件名称），不支持软连接。
forceOverwrite	输入	保存时是否强制覆盖已有文件，默认为“False”，不覆盖。

6.5.2.8 unserialize

函数功能

将**serialize**中保存的落盘数据文件加载到内存中，需指定文件名称及具体路径。

函数原型

```
unserialize(filePath);
```

参数说明

参数名	输入/输出	说明
filePath	输入	落盘数据文件保存路径，输入文件大小支持范围为(0, 4GB]。

6.5.2.9 dump_buffer

函数功能

将图像内存数据落盘到二进制文件，需要指定文件名称及具体路径。

函数原型

```
dump_buffer (filePath, forceOverwrite = False);
```

参数说明

参数名	输入/输出	说明
filePath	输入	落盘数据文件路径（包含文件名称），不支持软连接。
forceOverwrite	输入	保存时是否强制覆盖已有文件，默认为“False”，不覆盖。

6.5.2.10 Image 类的属性列表

属性名	说明
device	Image所在Device的ID。
height	返回Image类对齐后的高。
width	返回Image类对齐后的宽。
original_height	返回Image类的原始高。
original_width	返回Image类的原始宽。
format	返回图像的格式。

6.5.3 ImageProcessor 类

6.5.3.1 类说明

作为图像处理类，主要开放图像编解码、缩放和抠图等接口。

6.5.3.2 ImageProcessor(deviceId=0)

函数功能

ImageProcessor类的构造函数，若因内存不足，或无法识别芯片等构造失败的场景会抛出runtime异常。

输入参数说明

参数名	类型	说明
deviceId	int	图像处理类部署的芯片，默认为0号芯片。 取值范围：[0, *识别到的芯片个数* - 1]

输出参数说明

ImageProcessor对象。

6.5.3.3 decode(inputPath, decodeFormat)

函数功能

ImageProcessor类的图片解码接口，输入输出格式、分辨率范围、对齐方式请参见**Decode**。

输入参数说明

参数名	类型	说明
inputPath	str	输入待解码的图片路径。
decodeFormat	image_format枚举类	输入解码后图片的格式。（PNG格式解码无需设置。）

返回参数说明

返回解码后的Image类。

抛异常接口

图片路径错误、解码格式错误、解码失败，抛出runtime异常。

6.5.3.4 encode(inputImage, savePath, encodeLevel)

函数功能

ImageProcessor类的图片编码接口，输入输出格式、分辨率范围、对齐方式请参见**Encode**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入编码前的Image类。
savePath	str	输入编码后保存的图片路径，文件后缀名限制为**jpg**，编码后图片保存至该路径下。
encodeLevel	int	默认为“100”，输入编码的级别[0, 100]。

返回参数说明

无

抛异常接口

图片路径错误、编码级别错误、编码失败，抛出runtime异常。

6.5.3.5 resize(inputImage, resize, interpolation)

函数功能

ImageProcessor类的图像缩放接口，输入输出格式、分辨率范围、对齐方式请参见**Resize**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入缩放前的Image类。
resize	Size类	输入图像缩放的宽高。
interpolation	Interpolation参数枚举类	输入图像的缩放方式，默认为“HUAWEI_HIGH_ORDER_FILTER”。

返回参数说明

返回缩放后的Image类。

抛异常接口

缩放失败，抛出runtime异常。

6.5.3.6 padding(inputImage, padDim, color, borderType)

函数功能

ImageProcessor类的图像处理补边接口，输入输出格式、分辨率范围请参见**Padding**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入补边前的Image类。
padDim	Dim类	输入图像补边的尺寸。
color	Color类	输入补边三通道颜色，仅在“borderType”设置为“BORDER_CONSTANT”时有效。
borderType	BorderType参数枚举类	输入补边方式。具体请参见**BorderType补边方式枚举类型**。

返回参数说明

返回补边后的Image类。

抛异常接口

补边失败，抛出runtime异常。

6.5.3.7 crop(inputImage, cropRectVec)

函数功能

ImageProcessor类的图像抠图接口，输入输出格式、分辨率范围、对齐方式请参见**Crop**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入抠图前的Image类。
cropRectVec	List[Rect]	输入图像的抠图坐标框列表。

返回参数说明

返回抠图后的Image类列表List[Image]。

抛异常接口

抠图失败，抛出runtime异常。

6.5.3.8 crop(inputImageVec, cropRectVec)

函数功能

ImageProcessor类的图像抠图接口，输入输出格式、分辨率范围、对齐方式请参见**Crop**。

输入参数说明

参数名	类型	说明
inputImageVec	List[Image]	输入抠图前的Image类列表。
cropRectVec	List[Rect]	输入图像的抠图坐标框列表。

返回参数说明

返回抠图后的Image类列表List[Image]。

抛异常接口

抠图失败，抛出runtime异常。

6.5.3.9 crop_resize(inputImage, cropResizeVec)

函数功能

ImageProcessor类的图像抠图并缩放接口，输入输出格式、分辨率范围、对齐方式请参见**CropResize**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入抠图缩放前的Image类。
cropResizeVec	List[tuple(Rect, Size)]	输入图像的抠图缩放参数列表。Rect为抠图坐标框，Size为缩放宽高。

返回参数说明

返回抠图缩放后的Image类列表List[Image]。

抛异常接口

抠图缩放失败，抛出runtime异常。

6.5.3.10 crop_paste(inputImage, cropPasteRect, pastedImage)

函数功能

ImageProcessor类的图像抠图并贴图接口。

1. 从“inputImage”中抠取一块图像，缩放至指定贴图区域的大小。
2. 将抠图后的图片贴到“pastedImage”的指定贴图区域。输入输出格式、分辨率范围、对齐方式请参见**CropAndPaste**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入抠图前的Image类。
cropPasteRect	tuple(Rect, Rect)	输入图像的抠图参数。 第一个Rect对应抠图参数，第二个Rect对应贴图参数。
pastedImage	Image类	被贴图的Image类。

返回参数说明

无

抛异常接口

抠图并贴图失败，抛出runtime异常。

6.5.4 VdecCallBacker 类

6.5.4.1 类说明

视频解码绑定回调函数类，用于绑定用户自定义的回调函数。

6.5.4.2 回调函数格式

功能说明

回调函数用于接收VideoDecoder解码后的数据（如Image数据、frameId等）并在回调函数内实现自定义处理功能。

用户如需实现自定义处理功能，建议进行简单的操作，如数据保存、列表写入（append），降低函数处理耗时，防止解码线程阻塞，导致视频解码速度变慢。

函数原型

```
def callback_func(decodedImage: Image, channelId: int, frameId: int) -> None
```

参数说明

callback_func为函数名称，用户可自己定义，与**registerVdecCallBack**注册的函数名一致即可。

参数名	类型	说明
decodedImage	Image类	解码后输出的图像类。
channelId	int	视频流索引，在VideoDecoder类初始化设置。
frameId	int	视频帧索引，由VideoDecoder类的decode函数来设置。

抛异常接口

未按指定格式定义的函数参数及输出，抛出“pybind11::error_already_set”异常。

6.5.4.3 VdecCallBacker()

函数功能

VdecCallBacker的构造函数。

输出参数说明

VdecCallBacker对象。

6.5.4.4 registerVdecCallBack(callback_func)

函数功能

将用户自定义的回调函数注册，以便解码后在回调函数中实现自定义操作。

输入参数说明

参数名	类型	说明
callback_func	Python定义的函数名	需传入按指定格式定义的函数，否则将抛出异常。

返回参数说明

无。

抛异常接口

未按指定格式传入定义的回调函数，将抛出TypeError异常或runtime异常。

6.5.5 VideoDecoder 类

6.5.5.1 类说明

视频解码类，用于提供视频解码接口。

6.5.5.2 VideoDecoder(pyVdecConfig, pyVdecCallBacker, deviceId, channelId)

函数功能

VideoDecoder类的构造函数。

若因内存不足，pyVdecCallBacker未注册回调函数，或无法识别芯片等构造失败的场景会抛出“std::runtime_error”异常。

输入参数说明

参数名	类型	说明
pyVdecConfig	VideoDecodeConfig类	视频解码的参数。
pyVdecCallBacker	VdecCallBacker类	绑定回调函数的VdecCallBacker类。（须先注册回调函数，否则构造VideoDecoder会抛出异常。）

参数名	类型	说明
deviceId	int	视频解码器部署的芯片，默认为 0 号芯片。 取值范围：[0, *识别到的芯片个数* - 1]。
channelId	int	视频解码器的视频流索引。 ● Atlas 200/500 A2 推理产品：取值范围：[0, 31]。 若存在重复构造相同 channelId的场景，则会构造失败。

返回参数说明

VideoDecoder对象。

6.5.5.3 decode(inputData, frameId)

函数功能

VideoDecoder的视频解码接口。

输入参数说明

参数名	类型	说明
inputData	py::buffer二进制数据类型	传入视频帧二进制数据。
frameId	int	输入解码前视频帧的帧ID。

返回参数说明

无。（默认解码方式为非实时出帧，解码过程中存在缓存帧，Vdec需要在收到码流中的多帧数据后，才开始输出解码结果。）

抛异常接口

解码失败，抛出runtime异常。

6.5.6 VencCallBacker 类

6.5.6.1 类说明

视频编码绑定回调函数类，用于绑定用户自定义的回调函数。

6.5.6.2 回调函数格式

说明

回调函数用于接收VideoEncoder编码后的数据（如视频帧数据、ByteSize、frameId等）并在回调函数内实现自定义处理功能。

用户如需实现自定义处理功能，建议进行简单的操作，如数据保存、列表写入（append），降低函数处理耗时，防止编码线程阻塞，导致视频编码速度变慢。

函数原型

```
def callback_func(pyBytes: bytes, outDataSize: int, channelId: int, frameId: int) -> None
```

参数说明

callback_func为函数名称，用户可自己定义，与**registerVencCallBack**注册的函数名一致即可。

参数名	类型	说明
pyBytes	bytes	编码后输出视频帧字节数据。
outDataSize	int	编码后输出视频帧数据的内存大小。
channelId	int	视频流索引，在 VideoEncoder 类初始化设置。
frameId	int	视频帧索引，由 VideoEncoder 类的 encode 函数来设置。

返回参数说明

无。

抛异常接口

未按指定格式定义的函数参数及输出，抛出“pybind11::error_already_set”异常。

6.5.6.3 VencCallBacker()

函数功能

VencCallBacker的构造函数。

输出参数说明

VencCallBacker对象。

6.5.6.4 registerVencCallBack(callback_func)

函数功能

将用户自定义的回调函数注册，以便编码后在回调函数中实现自定义操作。

输入参数说明

参数名	类型	说明
callback_func	Python定义的函数名	需传入按指定格式定义的函数，否则将抛出异常。

返回参数说明

无。

抛异常接口

未按指定格式传入定义的回调函数，将抛出TypeError异常或runtime异常。

6.5.7 VideoEncoder 类

6.5.7.1 类说明

VideoEncoder类，作为视频编码类，主要开放视频编码接口。

6.5.7.2 VideoEncoder(pyVencConfig, pyVencCallBacker, deviceId, channelId)

函数功能

VideoEncoder类的构造函数。

若因内存不足，pyVencCallBacker未注册回调函数，或无法识别芯片等构造失败的场景会抛出“std::runtime_error”异常。

输入参数说明

参数名	类型	说明
pyVencConfig	VencConfig类	视频编码的参数。视频编码的最大分辨率（宽 * 高）不能超过 4096 * 2304。
pyVencCallBacker	VencCallBacker类	绑定回调函数的VencCallBacker类。（须先注册回调函数，否则构造VideoEncoder会抛出异常。）
deviceId	int	视频编码器部署的芯片，默认为 0 号芯片。 取值范围：[0, *识别到的芯片个数* - 1]。

参数名	类型	说明
channelId	int	视频编码器的视频流索引。 ● Atlas 200/500 A2 推理产品：只能实例化一个VideoEncoder，因此用户无需设置channelId。

返回参数说明

VideoEncoder对象。

6.5.7.3 encode(inputImage, frameId)

函数功能

VideoEncoder的视频编码接口。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入编码前的Image类。
frameId	int	输入视频帧索引。

返回参数说明

无。

抛异常接口

编码失败，抛出runtime异常。

6.5.8 DeviceMemory 类

6.5.8.1 类说明

从Device侧申请内存，获取内存指针地址的值。

6.5.8.2 DeviceMemory(int size)

函数功能

从Device侧申请内存。

输入参数说明

参数名	类型	说明
size	int	申请内存的大小，单位Byte。

返回参数说明

DeviceMemory对象。

6.5.8.3 get_data()

函数功能

获取对象中内存指针的值。

返回参数说明

返回值	类型	说明
Output	int	返回对象内存指针中的值。

6.5.9 dvpp 模块

6.5.9.1 模块说明

dvpp模块预计于2024年退出，建议使用**ImageProcessor类**。

6.5.9.2 read_image(inputPath, deviceId, decodeFormat)

函数功能

解码读取图片。

输入输出格式及对齐方式请参见**decode(inputPath, decodeFormat)**。

输入参数说明

参数名	类型	说明
inputPath	string	图片路径。
deviceId	int	Device设备的ID号。
decodeFormat	image_format枚举类	输入图片的解码格式。

返回参数说明

Image对象。

抛异常接口

创建ImageProcessor失败、图片解码失败，抛出runtime异常。

6.5.9.3 resize(inputImage, resize, interpolation)

函数功能

图像缩放。输入输出范围及对齐方式请参见**resize(inputImage, resize, interpolation)**。

输入参数说明

参数名	类型	说明
inputImage	Image类	输入缩放前的Image类。
resize	Size类	输入图像缩放的宽高。
interpolation	Interpolation参数枚举类	输入图像的缩放方式，默认为“HUAWEI_HIGH_ORDER_FILTER”。

返回参数说明

Image对象。

抛异常接口

创建ImageProcessor失败、图片缩放失败，抛出runtime异常。

6.5.10 batch_concat(inputs)

函数功能

将多个Tensor进行组batch，按照第0维组装，内存连续。

输入参数说明

参数名	类型	说明
inputs	List[base.Tensor]	Tensor列表。

输出参数说明

Tensor对象。

抛异常接口

传入Tensor异常、组Tensor后输出的Tensor异常，抛出runtime异常。

6.5.11 image(inputPath, deviceId, decodeFormat)

函数功能

解码读取图片。

输入输出格式及对齐方式请参见**decode(inputPath, decodeFormat)**。

输入参数说明

参数名	类型	说明
inputPath	string	图片路径。
deviceId	int	Device设备的ID号。
decodeFormat	image_format枚举类	输入图片的解码格式。

输出参数说明

Image对象。

抛异常接口

创建ImageProcessor失败、图片解码失败，抛出runtime异常。

6.5.12 resize_info(image, resize_height, resize_width, resize_type)

函数功能

设置缩放信息。

输入参数说明

参数名	类型	说明
image	Image对象	图像信息。
resize_height	Int	缩放高度。
resize_weight	Int	缩放宽度。
resize_type	ResizeType	缩放类型。

输出参数说明

ResizedImageInfo对象。

6.5.13 bytes_to_ptr(data)

函数功能

将二进制数据转换成指针地址值。

输入参数说明

参数名	类型	说明
data	py::buffer二进制数据类型	传二进制数据。

返回参数说明

返回指针地址值。

6.5.14 transpose(input_tensor, axes)

函数功能

可通过指定的一组轴维度（axes），对输入的Tensor数据进行转置处理，如未指定具体轴维度，则默认对Tensor数据进行反序转置。

功能仅支持在Host侧执行操作，如需处理Device侧Tensor数据，请先通过**to_host**接口，将Device侧数据转移到Host侧，再进行转置。

输入参数说明

参数名	类型	说明
input_tensor	Tensor类	待转置Tensor类。 维度支持2维、3维、4维，数据类型支持base.dtype.float32、base.dtype.float16、base.dtype.uint8，具体请参见**Tensor类的属性列表**。
axes	List[int]	转置选项，默认值为空。 如果未指定具体axes，则默认生成反序“axes”对“input”中的数据进行反序转置。 例如：三维张量默认生成反序“axes”为{2, 1, 0}。

返回参数说明

返回转置后的Tensor类数据。

6.6 模型推理

6.6.1 Model 类

6.6.1.1 Model(modelPath, deviceId=0)

函数功能

初始化模型。

输入参数说明

参数名	类型	说明
modelPath	string	模型路径。
deviceId	int	模型部署的芯片ID，默认为0号芯片。（-1表示模型部署在Host侧，为保留字段，请勿使用。）

返回参数说明

模型对象

6.6.1.2 Model(option, deviceId)

函数功能

用户可选择特定加载方式，加载推理模型。

输入参数说明

参数名	类型	说明
option	ModelLoadOptV2数据类型	用户可通过参数选择模型输入方式，具体请参见**ModelLoadOptV2类**。
deviceId	int	模型部署的芯片ID，默认为0号芯片。（-1表示模型部署在Host侧，为保留字段，请勿使用。）

返回参数说明

返回模型对象。

6.6.1.3 infer(tensorList)

函数功能

通过输入Tensor列表进行模型推理。

输入参数说明

参数名	类型	说明
tensorList	List[base.Tensor]	Tensor列表。

返回参数说明

返回模型推理输出的List[base.Tensor]。

抛异常接口

分配Tensor内存失败及推理失败，抛出runtime异常。

6.6.1.4 infer(tensor, args)

函数功能

通过输入多个Tensor对象进行模型推理。

输入参数说明

参数名	类型	说明
tensor	Tensor对象	模型推理所需Tensor。 如需输入多个Tensor，请使用”,”以分隔形式传入，最大传入Tensor数量为1024。
args	可变参数	

返回参数说明

返回模型推理输出的List[base.Tensor]。

抛异常接口

参数校验失败，分配Tensor内存失败及推理失败，抛出runtime异常。

6.6.1.5 input_dtype(index)

函数功能

获得模型输入的对应Tensor的数据类型信息。

输入参数说明

参数名	类型	说明
index	int	模型输入的索引（第index个输入）。

返回参数说明

返回输入Tensor的数据类型，base.dtype类型。

6.6.1.6 output_dtype(index)

函数功能

获得模型输出的对应Tensor的数据类型信息。

输入参数说明

参数名	类型	说明
index	int	模型输出的索引（第index个输出）。

返回参数说明

返回输出Tensor的数据类型，base.dtype类型。

6.6.1.7 input_shape(index)

函数功能

获得模型输入的对应Tensor的数据shape信息。

输入参数说明

参数名	类型	说明
index	int	模型输入的索引（第index个输入）。

返回参数说明

返回对应输入的Tensor的shape信息，List[int]类型。

6.6.1.8 output_shape(index)

函数功能

获得模型输出的对应Tensor的数据shape信息。

输入参数说明

参数名	类型	说明
index	int	模型输出的索引（第index个输出）。

返回参数说明

返回对应输出的Tensor的shape信息，List[int]类型。

6.6.1.9 model_gear()

函数功能

获得模型支持的动态输入档位信息，支持动态Batch、动态分辨率和分档动态维度模型。

返回参数说明

返回模型支持的输入Tensor档位信息，List[List[int]]类型。

返回数据结构为空，表示获取分档信息失败或者模型不属于函数功能中描述的三种模型之一，具体的原因可查看报错信息确认。

例如，模型为动态分辨率模型，List[int]为某一档的高、宽值。

6.6.1.10 Model 类的属性列表

属性名	说明	备注
input_format	获得模型输入的数据组织形式。	VisionDataFormat类型，NHWC或者NCHW。
input_num	获得模型的输入个数。	int类型。
output_num	获得模型的输出个数。	int类型。

6.6.2 model(modelPath, deviceId)

函数功能

加载模型

输入参数说明

参数名	类型	说明
modelPath	str	模型路径。

参数名	类型	说明
deviceId	Int	加载的deviceId。

输出参数说明

Model对象。

6.7 模型后处理

6.7.1 分类后处理

基类ClassPostProcessBase。

支持的模型后处理

模型后处理	说明
Resnet50PostProcess	ResNet-50后处理。

构造函数

```
Resnet50PostProcess(config_data, label_path)
Resnet50PostProcess(config_path, label_path)
```

Resnet50PostProcess 输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

Resnet50PostProcess 输出参数说明

Resnet50PostProcess对象。

统一的后处理函数

```
process(tensor)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。

后处理函数输出参数说明

list[list[ClassInfo]]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.7.2 目标检测后处理

基类ObjectPostProcessBase。

支持的模型后处理

模型后处理	说明
Yolov3PostProcess	YOLOv3后处理。
SsdMobilenetv1FpnPostProcess	SsdMobilenetv1Fpn后处理。
SsdMobilenetFpn_MindsporePost	SsdMobilenetFpn MindSpore后处理。
Ssdvgg16PostProcess	Ssdvgg16后处理。
FasterRcnnPostProcess	FasterRcnn后处理。
MaskRcnnMindsporePost	MaskRcnnMindspore后处理。

统一的构造函数

```
postProcessName(config_data, label_path)
postProcessName(config_path, label_path)
```

例如：Yolov3PostProcess(config_data, label_path)，Yolov3PostProcess(config_path, label_path)。

构造函数输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

构造函数输出参数说明

对应后处理对象。

统一的后处理函数

```
process(tensor, params)
process(tensor, param)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。
params	List[base.ResizedImageInfo]	ResizedImageInfo列表。
param	base.ResizedImageInfo	ResizedImageInfo对象。

后处理函数输出参数说明

list[list[ObjectInfo]]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.7.3 分割后处理

基类SemanticSegPostProcessBase。

支持的模型后处理

模型后处理	说明
Deeplabv3PostProcess	Deeplabv3后处理。
Deeplabv3PytorchPost	Deeplabv3 pytorch后处理。
Deeplabv3TensorflowPost	Deeplabv3 TensorFlow后处理。
Deeplabv3MindsporePost	Deeplabv3 MindSpore后处理。
UNetMindSporePostProcess	UNet MindSpore后处理。

统一的构造函数

```
postProcessName(config_data, label_path)
postProcessName(config_path, label_path)
```

例如：Deeplabv3PostProcess (config_data, label_path)，Deeplabv3PostProcess(config_path, label_path)。

构造函数输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

构造函数输出参数说明

对应的后处理对象。

统一的后处理函数

```
process(tensor, params)
process(tensor, param)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。
params	List[base.ResizedImageInfo]	ResizedImageInfo列表。
param	base.ResizedImageInfo	ResizedImageInfo对象。

后处理函数输出参数说明

list[SemanticSegInfo]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.7.4 文本生成后处理

基类TextGenerationPostProcessBase。

支持的模型后处理

模型后处理	说明
CrnnPostProcess	Crnn后处理。
TransformerPostProcess	Transformer后处理。

统一的构造函数

```
postProcessName(config_data, label_path)
postProcessName(config_path, label_path)
```

例如：CrnnPostProcess(config_data, label_path)，CrnnPostProcess(config_path, label_path)。

构造函数输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

构造函数输出参数说明

对应的后处理对象。

统一的后处理函数

```
process(tensor, params)
process(tensor, param)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。
params	List[base.ResizedImageInfo]	ResizedImageInfo列表。
param	base.ResizedImageInfo	ResizedImageInfo对象。

后处理函数输出参数说明

list[TextsInfo]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.7.5 文本检测后处理

支持的模型后处理

模型后处理	说明
CtpnPostProcess	Ctpn后处理。
PSENetPostProcess	PSEnet后处理。

统一的构造函数

```
postProcessName(config_data, label_path)
postProcessName(config_path, label_path)
```

例如：CtpnPostProcess (config_data, label_path)，CtpnPostProcess (config_path, label_path)。

构造函数输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

构造函数输出参数说明

对应的后处理对象。

统一的后处理函数

```
process(tensor, params)
process(tensor, param)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。
params	List[base.ResizedImageInfo]	ResizedImageInfo列表。
param	base.ResizedImageInfo	ResizedImageInfo对象。

后处理函数输出参数说明

list[list[TextObjectInfo]]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.7.6 关键点检测后处理

基类KeypointPostProcessBase。

支持的模型后处理

模型后处理	说明
openPosePostProcess	openPost后处理。
HigherHRnetPostProcess	HigherHRnet后处理。

构造函数

```
postProcessName(config_data, label_path)
postProcessName(config_path, label_path)
```

构造函数输入参数说明

参数名	类型	说明
config_data	dict	配置数据。
label_path	str	标签路径。
config_path	str	配置路径。

构造函数输出参数说明

对应的后处理对象。

统一的后处理函数

```
process(tensor, params)
process(tensor, param)
```

后处理函数输入参数说明

参数名	类型	说明
tensor	List[base.Tensor]	Tensor列表。
params	List[base.ResizedImageInfo]	ResizedImageInfo列表。

参数名	类型	说明
param	base.ResizedImageInfo	ResizedImageInfo对象。

后处理函数输出参数说明

list[list[KeyPointDetectionInfo]]

抛异常接口

传入Tensor失败，拷贝至Host失败，抛出runtime异常。

6.8 日志输出

6.8.1 log 类函数集

函数名	参数类型	说明
debug(str)	String	打印Debug级日志。
info(str)	String	打印Info级日志。
warning(str)	String	打印Warning级日志。
error(str)	String	打印Error级日志。
fatal(str)	String	打印Fatal级日志。
init()	-	日志初始化。

7 常用操作

7.1 模型后处理开发
7.2 辅助工具

7.1 模型后处理开发

7.1.1 模型后处理（tensorinfer 框架）

7.1.1.1 模型后处理介绍

模型后处理为与模型一一对应的配套操作，在mxVision中其主要工作是用于对模型推理插件传入的推理结果张量进行处理，如在目标检测任务中，需要对目标框进行去重，排序，筛选等，最后将处理结果写入目标信息的类对象中，传递回给目标检测后处理插件进行写入元数据以传递给下游插件。目前mxVision所支持的模型均已开发相应的后处理动态库，具体请参见**表7-1**。请用户根据实际情况选择现有后处理插件或自行开发适配所需模型，具体操作步骤请参见**7.1.1.2 后处理类开发步骤**。

表 7-1 已支持模型及对应后处理动态库

已支持模型名称	后处理类名称	推理插件使用的后处理动态库
YOLOv3	Yolov3PostProcess	modelpostprocessors/libyolov3postprocess.so
YOLOv3-tiny		
ResNet-50	Resnet50PostProcess	modelpostprocessors/libresnet50postprocess.so
Faster Rcnn	FasterRcnnPostProcess	modelpostprocessors/libfasterrcnnpostprocess.so
SSD-VGG16	Ssdvgg16PostProcess	modelpostprocessors/libssdvgg16postprocess.so

已支持模型名称	后处理类名称	推理插件使用的后处理动态库
SSD MobileNet v1 FPN	SsdMobilenetv1FpnPostProcess	modelpostprocessors/libssdmobilenetv1fpnpostprocess.so
CRNN	CrnnPostProcessor	modelpostprocessors/libcrnnpostprocess.so

后处理需要一个配置文件和一个标签文件。

- 标签文件需将所有类别的名称按类别ID顺序逐行输入（#开头的行不会被读取），填写示例参考如下：

```
# This is modified from https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
unknown type
tench, Tinca tinca
goldfish, Carassius auratus
great white shark, white shark, man-eater, man-eating shark, Carcharodon carchariasB
tiger shark, Galeocerdo cuvieri
……
```

- 每个模型所需要的配置参数一般不相同，用户可自行定义增加。后处理基类中已经读取“CHECK_MODEL”参数，当值为“true”时会对模型的输出张量形状进行校验，并拦截后处理不兼容的模型，当值为“false”时会跳过模型校验。当前mxVision已支持的模型输出张量形状与配置文件参数请见**4.2 模型后处理配置参数**。

7.1.1.2 后处理类开发步骤

1. 根据任务类型，选择mxVision已经支持的后处理基类去派生一个新的子类，后处理基类分别为目标检测、分类任务、语义分割、文本生成，这些基类都继承自相同的父类PostProcessBase。

   在Init函数中，需要调取父类的Init()接口读取配置参数，然后再调用父类对象的configData_.GetFileValue()接口读取子类所需要的配置参数。

   在Process函数中，需要以模型的输出张量Tensors作为输入，以对应的数据结构类对象作为输出。

2. 调用父类的CheckAndMoveTensors()接口，对Tensors的形状进行校验并将内存移动至Host侧，然后再进行相应操作获得结果。
3. 后处理开发完成后，增加一个对外的接口如GetObjectInstance()，以便于让业务流中的后处理插件动态加载此后处理so。

   如果当前后处理基类所采用的数据结构无法满足需求，可以新增后处理基类继承PostProcessBase，并写入新的数据结构。

```
// 1.目标检测数据结构：
class ObjectInfo {
public:
    float x0;
    float y0;
    float x1;
    float y1;
    float confidence;
    float classId;
    std::string className;
    std::vector<std::vector <int>> mask; // 用于实例分割
};
```

```
// 2.分类任务数据结构：
class ClassInfo {
public:
    int classId;
    float confidence;
    std::string className;
};

// 3.语义分割任务数据结构：
class SemanticSegInfo {
public:
    std::vector<std::vector<int>> pixels;
    std::vector<std::string> labelMap;
};

// 4.文本生成（机器翻译，文字识别，语音识别等）接口：
class TextsInfo {
public:
    std::vector<std::string> text;
};

// 5.文本框检测接口：
class TextObjectInfo {
public:
    float x0;
    float y0;
    float x1;
    float y1;
    float x2;
    float y2;
    float x3;
    float y3;
    float confidence;
    std::string result;
};
```

图 7-1 已支持模型的后处理 UML 类图

7.1.1.3 编译

CMakeLists.txt 文件配置

主要用于设置后处理动态库的目标文件以及链接相关的第三方库。每一个后处理都有一个CMakeLists.txt文件与之对应。对于新增后处理的CMakeLists.txt文件，用户只需修改生成的后处理名和生成动态库的目标文件即可，沿用原有的其他配置。以开发一个ResNet-50后处理为例，示例操作如下：

其中，“samplepostprocess”和“SamplePostProcess.cpp”分别代表生成的后处理动态库名和生成后处理的目标文件，其他的配置沿用原有的后处理。

命令行编译

命令行编译主要根据“CMakeLists.txt”文件的配置，进行插件的编译和生成。以下步骤中*{sdk_install_path}*为mxVision软件包的安装路径。

步骤1 新建并进入“SamplePostProcess”目录。

```
cd {sdk_install_path}/samples/
mkdir SamplePostProcess
cd SamplePostProcess
```

步骤2 将后处理动态库头文件、源文件和“CMakeLists.txt”文件存放至“SamplePostProcess”目录。

步骤3 新建并进入“build”目录。

```
mkdir build
cd build
```

步骤4 创建Makefile文件，避免污染项目的CMakeLists。

```
cmake ..
```

步骤5 根据生成的Makefile文件编译项目。

```
make -j
```

步骤6 将生成后处理的.so文件拷贝至“*{sdk_install_path}*/lib/modelpostprocessors”目录下。

```
make install
```

----结束

7.1.2 模型后处理（modelinfer 框架）

7.1.2.1 模型后处理介绍

说明

从当前版本不再演进，预计2024年退出，建议使用**7.1.1 模型后处理（tensorinfer框架）**。

模型后处理为与模型一一对应的配套操作，在mxVision中其主要工作是用于对模型推理插件传入的推理结果张量进行处理，如在目标检测任务中，需要对目标框进行去重，排序，筛选等。最后将处理结果写入推理插件的元数据中。目前mxVision所支持的模型均已开发相应的后处理参见。如果现有后处理无法兼容用户模型，用户应自行开发后处理，步骤见本章后续介绍。

表 7-2 已支持的模型及对应的后处理动态库

已支持模型名称	后处理类名称	推理插件使用的后处理动态库
YOLOv3	Yolov3PostProcessor	libMpYOLOv3PostProcessor.so
YOLOv3-tiny		
ResNet-50	Resnet50PostProcessor	libresnet50postprocessor.so
Faster Rcnn	FasterRcnnPostProcessor	libfasterrcnnpostprocessor.so
SSD-VGG16	SsdvggPostProcessor	libssdvggpostprocessor.so
SSD MobileNet v1 FPN	SsdmobilenetfpnPostProcessor	libssdmobilenetfpnpostprocessor.so
CRNN	CrnnPostProcessor	libcrnnpostprocessor.so

后处理需要一个配置文件和一个标签文件。

- 标签文件只需将所有类别的名称按类别ID顺序逐行输入即可（#开头的行不会被读取），如下所示。

```
# This is modified from https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
unknown type
tench, Tinca tinca
```

```
goldfish, Carassius auratus
great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias
tiger shark, Galeocerdo cuvieri
……
```

- 每个模型所需要的配置参数一般不相同，用户可自行定义增加。后处理基类中已经读取“CHECK_MODEL”的参数。
  - 当其为“true”时，会对模型的输出张量形状进行校验并拦截后处理不兼容的模型。
  - 当其为“false”时，会跳过模型校验。

  当前mxVision已支持的模型输出张量形状与配置文件参数请见**4 模型支持参考**。

7.1.2.2 后处理类开发步骤

开发步骤如下：

1. 继承“ModelPostProcessorBase.h”中的ModelPostProcessorBase基类，并自定义后处理类名。
2. 用户重写基类中的Init()、DeInit()、Process()三个成员函数，参见**表7-3**。
3. 用户可选择重写CheckModelCompatibility，用于对所读取模型的输出形状进行校验。
4. 在后处理类中实现“std::shared_ptr<ModelPostProcessorBase> GetInstance() ”接口函数功能。

表 7-3 成员函数说明

成员函数	说明
Init()	该接口仅调用一次，用于实现配置参数获取、业务逻辑初始化。
DeInit()	该接口仅调用一次，用于实现去初始化任务（例如内存释放）。
Process()	该接口由数据驱动，用于接收上游插件传入的数据，用于对模型输出进行处理，并将处理结果写入元数据。
MemoryDataToHost	可选，该接口可用于将上游推理插件传入的推理结果张量搬运到Host侧。
LoadConfigDataAndLabelMap	可选，该接口由Init()调用，用于实现配置参数获取。
GetModelTensorsShape	可选，该接口由Init()调用，用于获取模型的输入输出张量形状。
CheckModelCompatibility	可选，该接口由Init()调用，用于对所读取模型的输出形状进行校验。

对于目标检测类的模型，也可以让后处理依赖于已经开发好的ObjectPostProcessorBase基类。

表 7-4 成员函数说明

成员函数	说明
Init()	该接口仅调用一次，用于实现配置参数获取。
DeInit()	该接口仅调用一次，用于实现去初始化任务（例如内存释放）。
Process()	该接口由数据驱动，用于对模型输出进行目标处理，如去重，排序，筛选等。并将处理结果写入元数据。
ObjectDetectionOutput	该接口由Process()调用，用于对模型输出进行目标处理，如去重，排序，筛选等。

说明

后处理动态库中尽量避免使用静态变量，须保证多个后处理动态库实例间不会相互干扰。

7.1.2.3 输入数据获取

后处理的数据输入是通过“Process()”函数，将数据以入参的形式传入，具体参数参考**5.10.2 模型后处理类参考（modelinfer框架）**中的参数说明。

7.1.2.4 输出结果发送

metaDataPtr ： 向下游插件传输数据指针。

具体操作参考如下：

```
std::shared_ptr<MxTools::MxpiObjectList> objectList =
std::static_pointer_cast<MxTools::MxpiObjectList>(metaDataPtr);
MxTools::MxpiObject* objectData = objectList->add_objectvec();
objectData->set_x0(objInfo.x0 + xOffset);
objectData->set_y0(objInfo.y0 + yOffset);
objectData->set_x1(objInfo.x1 + xOffset);
objectData->set_y1(objInfo.y1 + yOffset);
```

其中，“objInfo”为后处理计算得出的框坐标，“xOffset”和“yOffset”为目标框坐标在原图上的偏移值。

7.1.2.5 编译

CMakeLists 文件配置

CMakeLists用于设置后处理动态库的目标文件以及链接相关的第三方库，每一个后处理都有一个“CMakeLists.txt”文件与之对应。对于后处理的CMakeLists.txt文件，用户需要修改其中**生成的后处理名**和**生成动态库的目标文件**，延用原有的其他配置。以ResNet-50后处理为例，示例操作参见如下。

其中，“resnet50postprocessor”和“ResNet50PostProcessor.cpp”分别代表生成的后处理动态库名和生成后处理的目标文件，其他的配置延用原有的后处理。

命令行编译

命令行编译主要根据“CMakeLists.txt”文件的配置，进行插件的编译和生成。

步骤1 新建“ResNet50PostProcessor”目录。

```
cd {sdk_install_path}/samples/
mkdir ResNet50PostProcessor
```

说明

*{sdk_install_path}*为mxVision软件包的安装路径。

步骤2 跳转到“ResNet50PostProcessor”目录下。

```
cd ResNet50PostProcessor
```

步骤3 将后处理动态库头文件、源文件和“CMakeLists.txt”文件存放至“ResNet50PostProcessor”目录。

步骤4 新建一个“build”文件夹，并跳转到“build”文件夹中。

```
mkdir build
cd build
```

步骤5 创建Makefile文件，避免污染项目的CMakeLists。

```
cmake ..
```

步骤6 根据生成的Makefile文件编译项目。

```
make -j
```

步骤7 将生成后处理的.so文件拷贝至“lib”目录下。

```
make install
```

----结束

7.2 辅助工具

7.2.1 一键式运维数据收集工具

7.2.1.1 工具介绍

一键式信息收集工具用于快速收集MindX SDK问题，定位所依赖的相关信息，避免用户反复收集确认日志。主要包括芯片信息（版本和日志）、操作系统版本信息、环境变量、网络信息以及MindX SDK信息（版本、配置文件、日志、第三方库版本）等。

该工具由“sdk_info_collector.sh”和“sdk_info_collector.py”两个文件组成，其中在“sdk_info_collector.sh”文件中调用了“sdk_info_collector.py”文件。此外，用户可以根据自己的需求在已有的信息收集工具上扩展新功能。

一键式信息收集工具为芯片日志和MindX SDK日志的信息收集提供了可配置的参数，请参见表7-5，其他的信息都默认收集。

表 7-5 参数介绍

参数	说明
--help|-h|-H	查看帮助信息。
--version|-v|-V	查看版本信息。

参数	说明
-p	必须设置的MindX SDK的日志存放目录，该目录下的日志文件数量不超过1500个。 -p *logs_path*：*logs_path*必须是绝对路径的形式。
-x	可配置的芯片日志收集参数。 ● 默认不收集芯片信息。 ● -x *n*：收集指定*n*号芯片的日志信息，*n*必须是整数。 ● -x *n1*,*n2*,*n3*：收集指定*n1*，*n2*，*n3*号芯片的日志信息，*n1*，*n2*，*n3*必须是整数。 ● -x -1：收集所有芯片的日志信息。
-r	可配置的MindX SDK日志天数收集参数。 ● 默认收集所有日志信息。 ● -r *x*d：收集*x*天内的日志信息，*x*必须是整数。 ● -r full：收集每天的日志信息。
-t	可配置的MindX SDK日志级别收集参数。 ● 默认收集所有日志信息。 ● -t d：收集调试（debug）日志及以上级别的日志信息。 ● -t i：收集正常（info）日志及以上级别的日志信息。 ● -t w：收集警告（warn）日志及以上级别的日志信息。 ● -t e：收集错误（error）日志及以上级别的日志信息。 ● -t f：收集致命（fatal）日志信息。 ● -t full：收集所有级别的日志信息。

7.2.1.2 工具使用

步骤1 跳转到信息收集工具所在的目录。

```
cd ${MX_SDK_HOME}/bin
```

*MX_SDK_HOME*为MindX SDK软件包安装目录，请确保**MX_SDK_HOME**为有效的环境变量。

步骤2 执行“sdk_info_collector.sh”。

以下为三种一键式信息采集的方式，用户可以根据自己的实际情况来使用。

- 采用默认参数，默认收集MindX SDK所有日志信息，不收集芯片日志。

```
bash sdk_info_collector.sh -p logs_path
```

表 7-6 命令参数说明（采用默认参数）

参数	说明
-p logs_path	收集指定目录“logs_path”中存放的MindX SDK日志信息。

- 指定参数收集，具体参数说明请参见表7-7。
```
bash sdk_info_collector.sh -p logs_path -r 3d -t w -x 6
```

表 7-7 命令参数说明（使用指定参数收集）

参数	说明
-p logs_path	收集指定目录“logs_path”中存放的MindX SDK日志信息。
-r 3d	收集最近3天内的MindX SDK日志信息。
-t w	收集警告日志及以上级别的MindX SDK日志信息。
-r 3d -t w	收集最近3天内的警告日志及以上级别的MindX SDK日志信息。
-x 6	收集编号为“6”的芯片日志信息。
-x 0,2,5	收集编号为“0”，“2”，“5”的芯片日志信息。

命令示例：
```
bash sdk_info_collector.sh -p logs_path -r 3d -t w -x 0,2,5
```

- 全量收集。
```
bash sdk_info_collector.sh -p logs_path -r full -t full -x -1
```

表 7-8 命令参数说明

参数	说明
-p logs_path	收集指定目录“logs_path”中存放的MindX SDK日志信息。
-r full	收集每天的MindX SDK日志信息。
-t full	收集所有级别的MindX SDK日志信息。
-r full -t full	收集所有的MindX SDK日志信息。
-x -1	收集所有的芯片日志信息。

步骤3 获取结果。

运行一键式信息收集工具的结果存放在“${MX_SDK_HOME}/bin”目录下，以“mindx_sdk_info_*年月日_时分秒*.tar.gz”格式命名的文件中。

----结束

7.2.2 单插件测试工具

7.2.2.1 工具介绍

单插件测试工具可以根据待测试插件的配置，自动添加Load与Dump数据，生成待测插件的pipeline文件，并对待测插件进行串流测试。

7.2.2.2 工具使用

步骤1 在“${MX_SDK_HOME}/toolkit/TestPlugin/config/config.py”文件中，配置待测插件属性，config文件大小限制为100M。

```
    "factory": "mxpi_pluginname",       # 插件类型
    "plugin_name": "mxpi_pluginname0",  # 插件名称
    "props": {                          # 插件属性
        "dataSource": "mxpi_parentName0",
        "resizeHeight": "416",
        "resizeWidth": "416"
    }
```

步骤2 在“${MX_SDK_HOME}/toolkit/TestPlugin/config/config.py”文件“load”参数中，配置插件的数据输入文件。若插件为多输入，可以配置多个数据输入文件。

数据输入文件中的内容必须是待测插件的输入数据。

```
    "load": [                           # load插件的输入文件名，一个输入文件对应一个load插件
        "input/pluginname0.json"
    ]
```

步骤3 在“${MX_SDK_HOME}/toolkit/TestPlugin/config/config.py”文件“dump”参数中，若插件为多输出，可以配置多个数据输出文件。

```
    "dump": [                           # dump插件的输出文件名，一个输出文件对应一个dump插件
        "pluginname0-output.json"
    ]
```

步骤4 运行单插件测试工具，生成待测插件的pipeline文件，并对待测插件进行串流测试。

执行**run.sh**脚本，会在pipeline目录下生成待测插件的pipeline文件，并对待测插件进行串流测试，生成“dump”参数中配置的数据输出文件（其最终执行文件为main.py）。

```
chmod +x run.sh && ./run.sh
```

----结束

7.2.2.3 单插件测试样例

步骤1 在“${MX_SDK_HOME}/toolkit/TestPlugin/config/config.py”配置文件中配置待测插件mxpi_imageresize的配置信息。配置信息如下：

- 如果待测插件是多输入插件，“load”配置项可以配置多个输入文件。
- 如果待测插件是多输出插件，“dump”配置项可以配置多个输出文件。

```
PipelineConfig = {
    "stream_config": {
        "deviceId": "0"
    },
    "factory": "mxpi_imageresize",      # 插件名称
    "props": {                          # 插件属性
        "dataSource": "mxpi_imagedecoder0",
```

```
        "resizeHeight": "416",
        "resizeWidth": "416"
    },
    "load": [                        # load插件的输入文件名，一个输入文件对应一个load插件
        "input/imageresize0.json"
    ],
    "dump": [                        # dump插件的输出文件名，一个输出文件对应一个dump插件
        "imageresize0-output.json"
    ]
}
```

步骤2 创建待测插件的数据输入文件“input/imageresize0.json”。

```
{
    "buffer": {
    },
    "metaData": [{
        "key": "ExternalObjects",
        "protoDataType": "MxpiObjectList"
    },
    {
        "key": "ReservedFrameInfo",
        "protoDataType": "MxpiFrameInfo"
    },
    {
        "key": "ReservedVisionList",
        "content": "mxpi_imagedecoder0"
    },
    {
        "key": "mxpi_imagedecoder0",
        "content": "{\"visionVec\":[{\"visionInfo\":{\"format\":1,\"width\":512,\"height
\":512,\"widthAligned\":512,\"heightAligned\":512},\"visionData\":{\"dataPtr
\":\"17695267880960\",\"dataSize\":393216,\"memType\":\"MXPI_MEMORY_DVPP\"}}]}",
        "visionData": [{
            "dataPtr": "17695267880960",  // dataPtr对应MxpiVisionData的dataPtr的地址
            "content": "vLu6ury9vr68vLy8" // 存在Device侧的图像数据，经过base64编码后的值
        }],
        "protoDataType": "MxpiVisionList"
    }]
}
```

步骤3 运行单插件测试工具，生成待测插件的pipeline文件，并对待测插件进行串流测试。命令如下：

```
chmod +x run.sh && ./run.sh
```

步骤4 检测单插件测试工具生成了pipeline文件。执行命令如下：

```
cat pipeline/mxpi_imageresize0.pipeline
{
    "TestPlugin": {
        "stream_config": {
            "deviceId": "0"
        },
        "mxpi_imageresize0": {
            "factory": "mxpi_imageresize",
            "next": [
                "mxpi_dumpdata0"
            ],
            "props": {
                "dataSource": "mxpi_imagedecoder0",
                "resizeHeight": "416",
                "resizeWidth": "416"
            }
        },
```

```
        "filesrc0": {
            "factory": "filesrc",
            "next": "mxpi_loaddata0",
            "props": {
                "location": "input/imageresize0.json",
                "blocksize": "40960000"
            }
        },
        "mxpi_loaddata0": {
            "factory": "mxpi_loaddata",
            "next": "mxpi_imageresize0:0"
        },
        "mxpi_dumpdata0": {
            "factory": "mxpi_dumpdata",
            "next": "fakesink0",
            "props": {
                "location": "imageresize0-output.json"
            }
        },
        "fakesink0": {
            "factory": "fakesink"
        }
    }
}
```

步骤5 检测单插件测试工具生成了待测插件的输出文件“imageresize0-output.json”。

步骤6 检测串流结果，串流成功后，会显示如图7-2所示的成功日志。

图 7-2 成功日志

```
I1127 19:37:14.000095 41097 MxsmStream.cpp:328] Links all elements successfully.
I1127 19:37:14.501157 41097 MpDumpData.cpp:27] initialize MpDumpData(mxpi_dumpdata0).
I1127 19:37:14.501327 41097 VpcResize.cpp:39] Begin to initialize VpcResize(mxpi_imageresize0).
I1127 19:37:14.538569 41097 VpcResize.cpp:63] End to initialize VpcResize(mxpi_imageresize0).
I1127 19:37:14.539198 41097 MxStreamManager.cpp:108] Creates stream(TestPlugin) successfully.
I1127 19:37:14.539247 41097 MxStreamManager.cpp:150] Creates streams successfully.
I1127 19:37:14.565289 41103 MxPluginBase.cpp:184] memory data is in device. memType=2, deviceId=0(mxpi_dumpdata0)
I1127 19:37:14.770403 41103 MxPluginBase.cpp:184] memory data is in device. memType=2, deviceId=0(mxpi_dumpdata0)
I1127 19:37:15.040222 41097 MxsmStream.cpp:492] Flushes the Stream data successfully.
I1127 19:37:15.040475 41097 MpDumpData.cpp:41] deInitialize MpDumpData(mxpi_dumpdata0).
I1127 19:37:15.040697 41097 VpcResize.cpp:69] Begin to deinitialize VpcResize(mxpi_imageresize0).
I1127 19:37:15.042662 41097 VpcResize.cpp:76] End to deinitialize VpcResize(mxpi_imageresize0).
I1127 19:37:15.043570 41097 MxsmStream.cpp:516] Destroys the stream(TestPlugin) successfully.
I1127 19:37:15.043660 41097 MxStreamManager.cpp:188] Destroy streams successfully.
```

----结束

7.2.3 插件信息搜集工具

7.2.3.1 工具介绍

插件信息收集工具用于快速搜集所有插件信息。

7.2.3.2 工具使用

步骤1 设置程序所需环境变量。

```
. ${MX_SDK_HOME}/set_env.sh
```

步骤2 执行程序。

进入到执行程序所在“bin”目录，执行如下命令。

```
./InspectAllPlugins
```

步骤3 获取结果。

程序输出结果保存在“AllPluginsInfo.json”文件。

----结束

7.2.4 插件性能统计工具

7.2.4.1 工具介绍

插件性能统计工具，可以快速测试分析MindX SDK的推理性能（所有性能统计数据涉及的时间单位都是微秒）。性能统计指标请见表7-9：

图 7-3 MindX SDK 推理简易流程图

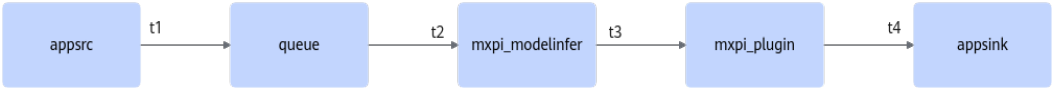


表 7-9 性能统计指标

性能	介绍
端到端耗时	从appsrc输入插件的srcpad输出数据（t1时刻）开始计时统计，到appsink输出插件的sinkpad接收数据（t4时刻）结束计时统计。 端到端耗时 = 结束时间（t4） - 开始时间（t1）
单插件耗时	统计单个插件（只涉及SDK开发的插件，即插件名以mxpi_开头，不包含Gstreamer插件）的耗时。 如mxpi_modelinfer插件的耗时，从这个插件sinkpad（t2时刻）开始计时，到这个插件srcpad（t3时刻）结束计时。 单插件耗时 = 结束时间（t3） - 开始时间（t2）
模型推理耗时	模型推理只在模型插件中存在，因此在模型推理插件内部会统计模型推理耗时。
后处理耗时	后处理只在modelinfer框架模型推理插件中存在，因此在modelinfer框架模型推理插件内部会统计后处理耗时。
吞吐率	appsink输出插件每次从sinkpad接收一次数据，做一次统计。
队列长度	每隔“ps_queue_size_interval_time”时间，获取一次队列的当前长度，获取“ps_queue_size_times”次数后，做一次队列长度的统计。

7.2.4.2 配置项介绍

插件性能统计功能新增了MindX SDK的配置文件“${MX_SDK_HOME}/config/sdk.conf”。如图7-4所示，配置项注释中如果注明了“[dynamic config]”，说明这

个配置项是动态可配置。（如果配置项的值小于最小值，则会设置成最小值。如果配置项的值大于最大值，则会设置成最大值。）配置项说明请参见表7-10。

须知

队列长度统计的时间间隔 * 队列长度统计次数需小于推理总时间，如未生成.queue日志，可通过降低时间间隔（ps_queue_size_interval_time）或统计次数（ps_queue_size_times）的方式进行调试。

图 7-4 SDK 配置文件示例

```
# MindX SDK configuration file

# whether to enable performance statistics, default is false
enable_ps=false

# performance statistics log output directory, create logs directory before use
ps_log_dir=logs

# performance statistics log filename
ps_log_filename=performance-statistics.log

# max log size, unit is MB, 1 <= value <= 1024, default is 100 [dynamic config]
ps_max_log_size=100

# value is percent, 0 <= value <= 100, default is 80 [dynamic config]
ps_queue_size_warn_percent=80

# unit is second, 1 <= value <= 24 * 3600, default is 60 [dynamic config]
ps_interval_time=60

# unit is millisecond, 10 <= value <= 1000. default is 50 [dynamic config]
ps_queue_size_interval_time=50

# 1 <= value <= 1000, default is 100 [dynamic config]
ps_queue_size_times=100

# set max malloc data size byte, k/K 1024, m/M 1024k, g/G 1024m, max support 4G
malloc_max_data_size=1G
```

表 7-10 配置项说明

配置项	说明
enable_ps	性能统计开关，默认为“false”。
ps_log_dir	性能统计日志目录，请在使用工具前配置到具体目录。
ps_log_filename	性能统计日志文件名。端到端日志文件会添加后缀.e2e。单插件、模型推理、后处理的日志文件会添加后缀.plugin。吞吐率日志文件会添加后缀.tpr。队列长度日志文件会添加后缀.queue。
ps_max_log_size	性能统计日志文件最大长度，日志文件大小超过这个值，将切换新的日志文件，单位MB，默认为100，取值范围为[1, 1024]。

配置项	说明
ps_queue_size_warn_percent	队列长度告警值百分比。当队列当前长度达到了队列总长度的告警值，将会打印告警日志，默认为80，取值范围为[1, 100]。
ps_interval_time	性能统计时间间隔，端到端、单插件、模型推理、后处理的性能统计时间间隔，默认为60，取值范围为[1, 24*3600]。
ps_queue_size_interval_time	队列长度统计的时间间隔，默认为50，取值范围为[10, 1000]。
ps_queue_size_times	队列长度统计次数，默认为100，取值范围为[1, 1000]。
malloc_max_data_size	设置申请内存的上限字节数，默认为1GByte，最大可支持至4GByte。

7.2.4.3 使用步骤

步骤1 开启性能统计开关，在“${MX_SDK_HOME}/config/sdk.conf”配置文件中，设置“enable_ps”为“true”，开启性能统计开关。

步骤2 调整性能统计时间间隔，默认值为60秒，统计时间偏长，调整“ps_interval_time”为“6”，每隔6秒，进行一次性能统计。

步骤3 Pipeline文件中，加入队列插件。如果pipeline中未使用队列插件，将不会统计队列长度。

图 7-5 Pipeline 文件添加队列插件

```
},
"classification+detection": {
    "stream_config": {
        "deviceId": "0"
    },
    "mxpi_imagedecoder0": {
        "factory": "mxpi_imagedecoder",
        "next": "queue0"
    },
    "queue0": {
            "factory": "queue",
            "next": "mxpi_imageresize0"
    },
    "mxpi_imageresize0": {
        "props": {
            "dataSource": "mxpi_imagedecoder0",
            "resizeHeight": "416",
            "resizeWidth": "416"
        },
        "factory": "mxpi_imageresize",
        "next": "mxpi_modelinfer0"
    },
    "mxpi_modelinfer0": {
        "props": {
            "dataSource": "mxpi_imageresize0",
            "modelPath": "../models/yolov3/yolov3_tf_bs1_fp16.om",
            "postProcessConfigPath": "../models/yolov3/yolov3_tf_bs1_fp16.cfg",
            "labelPath": "../models/yolov3/coco.names",
            "postProcessLibPath": "../../../lib/libMpYOLOv3PostProcessor.so"
        },
        "factory": "mxpi_modelinfer",
        "next": "mxpi_imagecrop0"
    },
```

步骤4 启动MindX SDK服务。

----结束

7.2.4.4 观测结果

步骤1 观察性能统计日志文件的生成。

查看“ps_log_dir”配置项设置的目录下，是否生成插件性能统计日志。

当配置项**ps_interval_time=6**时，程序会在启动**6**秒后，生成端到端日志文件、单插件日志文件、吞吐率日志文件。

当配置项**ps_queue_size_interval_time=50**、**ps_queue_size_times=100**时，程序会在启动**50 × 100**毫秒（即**5**秒）后，生成队列长度日志文件。如下图：

```
-r--r-----. 1 root root    270 Jun 24 17:46 performance-statistics.log.tpr.2022-06-24 17:46:55
-r--r-----. 1 root root    268 Jun 24 17:58 performance-statistics.log.tpr.2022-06-24 17:58:13
-r--r-----. 1 root root    270 Jun 24 19:25 performance-statistics.log.tpr.2022-06-24 19:25:55
-r--r-----. 1 root root    270 Jun 25 09:20 performance-statistics.log.tpr.2022-06-25 09:20:43
-r--r-----. 1 root root    294 Jun 25 15:08 performance-statistics.log.tpr.2022-06-25 15:08:51
-r--r-----. 1 root root    294 Jun 25 15:22 performance-statistics.log.tpr.2022-06-25 15:22:58
-r--r-----. 1 root root    294 Jun 25 15:55 performance-statistics.log.tpr.2022-06-25 15:55:30
-r--r-----. 1 root root    294 Jun 25 16:00 performance-statistics.log.tpr.2022-06-25 16:00:23
-r--r-----. 1 root root    294 Jun 27 10:02 performance-statistics.log.tpr.2022-06-27 10:02:23
-r--r-----. 1 root root    294 Jun 27 11:15 performance-statistics.log.tpr.2022-06-27 11:15:03
-r--r-----. 1 root root    270 Jun 27 11:15 performance-statistics.log.tpr.2022-06-27 11:15:49
-r--r-----. 1 root root    294 Jun 27 15:58 performance-statistics.log.tpr.2022-06-27 15:58:27
-r--r-----. 1 root root    270 Jun 27 15:59 performance-statistics.log.tpr.2022-06-27 15:59:08
-r--r-----. 1 root root    294 Jun 27 17:27 performance-statistics.log.tpr.2022-06-27 17:27:44
-r--r-----. 1 root root    294 Jun 28 02:07 performance-statistics.log.tpr.2022-06-28 02:07:19
-r--r-----. 1 root root    268 Jun 28 02:08 performance-statistics.log.tpr.2022-06-28 02:08:01
-r--r-----. 1 root root    270 Jun 28 09:29 performance-statistics.log.tpr.2022-06-28 09:29:43
-r--r-----. 1 root root    296 Jun 28 10:42 performance-statistics.log.tpr.2022-06-28 10:42:16
-r--r-----. 1 root root    270 Jun 28 10:42 performance-statistics.log.tpr.2022-06-28 10:42:58
-r--r-----. 1 root root    294 Jun 28 11:46 performance-statistics.log.tpr.2022-06-28 11:46:43
-r--r-----. 1 root root    270 Jun 28 16:45 performance-statistics.log.tpr.2022-06-28 16:45:12
-r--r-----. 1 root root    294 Jun 29 02:06 performance-statistics.log.tpr.2022-06-29 02:06:00
-r--r-----. 1 root root    270 Jun 29 02:06 performance-statistics.log.tpr.2022-06-29 02:06:43
-r--r-----. 1 root root    294 Jun 29 09:57 performance-statistics.log.tpr.2022-06-29 09:57:56
-r--r-----. 1 root root    294 Jun 29 10:02 performance-statistics.log.tpr.2022-06-29 10:02:08
-r--r-----. 1 root root    294 Jun 29 10:27 performance-statistics.log.tpr.2022-06-29 10:27:10
-r--r-----. 1 root root    294 Jun 29 14:32 performance-statistics.log.tpr.2022-06-29 14:32:33
-r--r-----. 1 root root    270 Jun 29 14:36 performance-statistics.log.tpr.2022-06-29 14:36:11
```

表 7-11 文件类型解释

文件名	文件类型
performance—statistics.log.e2e.20210329-183958.877032	端到端日志文件。
performance—statistics.log.plugin.20210329-183958.877905	单插件日志文件。
performance—statistics.log.queue.20210329-183957.893092	队列长度日志文件。
performance—statistics.log.tpr.20210329-183958.883263	吞吐率日志文件。

步骤2 观察端到端日志文件。

以下为端到端日志文件的示例内容。

{"average":"45138","frequency":"3","maxTime":"73058","minTime":"30511","streamName":"classification+detection","totalTime":"135416","type":"e2e","updateTime":"2021-03-29 20:13:59.717927"}

表 7-12 日志参数解释

参数	介绍
average	端到端平均耗时。单位：微秒。
frequency	端到端统计次数。
maxTime	最大的一次端到端耗时。单位：微秒。
minTime	最小的一次端到端耗时。单位：微秒。
streamName	pipeline中的“streamName”。
totalTime	端到端总耗时。单位：微秒。
type	统计类型，端到端类型为e2e。
updateTime	本次统计的当前时间。

步骤3 观察单插件日志文件。

以下为单插件日志文件的示例内容。

{"average":"18546","elementName":"mxpi_modelinfer0","factory":"mxpi_modelinfer","frequency":"3","maxTime":"18608","minTime":"18493","streamName":"classification+detection","totalTime":"55638","type":"plugin","updateTime":"2021-03-29 20:13:59.720010"}

表 7-13 日志参数解释

参数	介绍
average	统计的单个插件平均耗时。单位：微秒。
elementName	插件元件名称。
factory	插件类型。
frequency	插件处理的推理次数。
maxTime	最大的一次耗时。单位：微秒。
minTime	最小的一次耗时。单位：微秒。
streamName	pipeline中的“streamName”。
totalTime	单插件总耗时。单位：微秒。
type	统计类型，单插件统计类型为“plugin”，模型推理统计类型为“modelInference”，后处理统计类型为“postProcessor”。
updateTime	本次统计的当前时间。

步骤4 观察吞吐率日志文件。

以下为吞吐率日志文件的示例内容。


```
{"streamName":"classification
+detection","throughput":3,"throughputRate":0.5,"type":"throughputRate","updateTime":"2021-03-29
20:13:59.721819"}
```


表 7-14 日志参数解释

参数	介绍
streamName	pipeline中的“streamName”。
throughput	吞吐量。
throughputRate	吞吐率。
type	统计类型，吞吐率统计类型为“throughputRate”。
updateTime	本次统计的当前时间。

步骤5 观察队列长度日志文件。

以下为队列长度日志文件的示例内容。

队列长度的数据一直显示为0，即使在推理过程中，统计到的队列长度依然为0。说明在推理过程中，数据留在队列中的时间非常短，很快就被下游插件取走，导致统计队列长度时，获取到的队列长度为0。只有当推理过程中，出现阻塞现象，数据留在队列才会出现队列长度不为0的情况。


```
{"curSize":0,"elementName":"queue0","intervalTime":50000,"maxSize":0,"maxSizeBuffers":200,"recentSize":
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],"streamName":"classification
+detection","type":"queueSize","updateTime":"2021-03-29 20:14:45.953312"}
```


表 7-15 日志参数解释

参数	介绍
curSize	当前的队列长度，即“recentSize”列表中的最后一个值。
elementName	插件元件名称。
intervalTime	统计间隔时间，即与配置项“ps_queue_size_interval_time”含义相同。
maxSize	统计的这段时间内，最大的队列长度，即“recentSize”列表中的最大值。
maxSizeBuffers	队列总长度。
recentSize	每隔“ps_queue_size_interval_time”毫秒，统计“ps_queue_size_times”次数的队列长度数据列表。
streamName	pipeline中的“streamName”。

参数	介绍
type	统计类型，队列长度统计类型为“queueSize”。
updateTime	本次统计的当前时间。

----结束

8 FAQ

8.1 程序执行出现libascendcl.so报错

8.2 使用yum命令出现No module named 'dnf'报错

8.3 使用yum命令出现No module named '_conf'错误

8.4 执行命令查看版本号时，出现Permission denied报错

8.5 openEuler操作系统系统命令yum、cmake不可用

8.6 源码方式安装完Python 3.9.2后，pip3工具无法使用

8.7 在服务端接收请求或执行证书导入脚本时，KMC初始化随机数阻塞

8.8 使用mxVision时，出现can not find the element factory : mxpi_xxxpostprocessor报错

8.9 编译时出现libgfortran.so.4报错

8.10 在Kunpeng服务器中运行多路视频分析，出现CPU资源被占满，导致业务堆积、断流的情况

8.11 调用API参考（Python）视频解码/编码接口结束时出现core dumped现象

8.12 在Atlas 500 A2 智能小站默认配置下，无法安装mxVision

8.13 执行run.sh脚本时报错cannot find factory: mxpi_classpostprocessor

8.14 尝试打印Device侧的Tensor、Image内存数据时，发生coredump

8.15 使用mxVersion安装时，出现libpython3.9.so.1.0报错

8.16 mxpi_rtspsrc拉流插件运行偶现 BUS ERROR：XXX streamming stopped, reason error (-5) 错误

8.1 程序执行出现 libascendcl.so 报错

问题现象

程序在执行过程中出现如下报错：

```
error while loading shared libraries: libascendcl.so: cannot open shared object file: No such file or directory
```

原因分析

“LD_LIBRARY_PATH”环境变量中CANN环境变量路径未正确配置。

解决方案

请参考《CANN 软件安装指南CANN 软件安装指南》配置CANN环境变量。

8.2 使用 yum 命令出现 No module named 'dnf'报错

问题现象

在系统为CentOS或EulerOS时，使用**yum**命令出现以下错误：

```
Traceback (most recent call last):
  File "/usr/bin/yum", line 57, in <module>
    from dnf.cli import main
ModuleNotFoundError: No module named 'dnf'
```

解决方案

请删除“LD_LIBRARY_PATH”环境变量中的Python路径，使用系统的Python路径。

8.3 使用 yum 命令出现 No module named '_conf'错误

问题现象

在系统为CentOS或EulerOS时，使用**yum**命令出现以下错误：

```
ImportError: /lib64/libcurl.so.4: symbol SSLv3_client_method version OPENSSL_1_1_0 not defined in file
libssl.so.1.1 with link time reference
ModuleNotFoundError: No module named '_conf'
```

解决方案

请删除“LD_LIBRARY_PATH”环境变量中的“*${MX_SDK_HOME}*/opensource/lib”，使用系统的“libssl.so”。

8.4 执行命令查看版本号时，出现 Permission denied 报错

问题现象

使用以下命令查看版本号时：

```
./Ascend-mindxsdk-mxvision_{version}_linux-{arch}.run --version
```

出现类似如下的错误提示信息：

```
rm: cannot remove "xxx": Permission denied
```

原因分析

安装过其他未正确安装的软件包，该软件包在“/tmp”路径下留有缓存，影响到当前版本。

解决方案

可执行以下命令解决：

```
sudo rm -rf /tmp/mxVision
```

如果无sudo权限可执行：

```
chmod u+w -R /tmp/mxVision && rm -rf /tmp/mxVision
```

命令中的“/tmp/mxVision”请以实际情况自行替换。

8.5 openEuler 操作系统系统命令 yum、cmake 不可用

问题现象

openEuler 20.03、22.03操作系统中安装mxVision软件包导入环境变量后导致系统命令yum、cmake不可用。

原因分析

导入mxVision环境变量后，“/mxVision-*{version}*/opensource/lib”的“libssl.so”与**yum**、**cmake**所依赖的“libssl.so”冲突。（*{version}*为实际安装的mxVision软件版本号。）

解决方案

需要使用**yum**或者**cmake**命令时，将mxVision的“opensource/lib”相关路径从“LD_LIBRARY_PATH”环境变量中暂时移除。

8.6 源码方式安装完 Python 3.9.2 后，pip3 工具无法使用

问题现象

在青松服务器Kylin V10 sp1操作系统中，以源码方式安装完Python 3.9.2后，pip3工具无法使用。

执行过程中出现如下提示。

```
WARNING: pip is configured with locations that require TLS/SSL, however the ssl module in Python is not available.
```

原因分析

系统自带OpenSSL版本过低，不满足pip3工具的需求。

解决方案

手动升级OpenSSL版本，可参考如下方式进行解决。

步骤1 通过OpenSSL官方网站下载最新版本源码包，本文以当前版本源码包“openssl-1.1.1m.tar.gz”为例。

步骤2 将源码包上传至青松服务器Kylin V10 sp1操作系统*$HOME*目录下。

步骤3 解压源码包并访问至解压目录下。

```
tar zxvf openssl-1.1.1m.tar.gz
cd openssl-1.1.1m/
```

步骤4 执行源码编译。

```
./config --prefix=/usr/local/openssl no-zlib
make -j32
make install
```

步骤5 备份原有OpenSSL文件。

```
mv /usr/bin/openssl /usr/bin/openssl.bak
mv /usr/include/openssl/ /usr/include/openssl.bak
```

步骤6 执行以下命令，完成OpenSSL更新。

```
ln -s /usr/local/openssl/include/openssl/ /usr/include/openssl
ln -s /usr/local/openssl/lib/libssl.so.1.1 /usr/local/lib64/libssl.so
ln -s /usr/local/openssl/bin/openssl /usr/bin/openssl
```

----结束

8.7 在服务端接收请求或执行证书导入脚本时，KMC 初始化随机数阻塞

问题现象

在服务端接收请求或执行证书导入脚本时，KMC初始化随机数阻塞但未抛出Error，等待时间不确定且可恢复。

原因分析

“/dev/random”产生随机数发生阻塞。

解决方案

通过添加haveged服务解决随机数阻塞问题，Ubuntu系统可参考以下命令安装使用。

```
apt-get install -y haveged
```

8.8 使用 mxVision 时，出现 can not find the element factory : mxpi_xxxpostprocessor 报错

问题现象

使用mxVision时，出现“can not find the element factory : mxpi_xxxpostprocessor”。

在“/mxVision-*{version}*/opensource/bin”路径下执行**./gst-inspect-1.0 mxpi_xxxpostprocessor(插件名)** 检查插件，发现插件能够正常加载，但运行时仍然报同样的错误信息。

原因分析

Gstreamer的历史缓存没有清除。

解决方案

步骤1 确认环境已安装python3.9。

步骤2 执行**rm ~/.cache/gstreamer-1.0/registry.*{arch}*.bin**（其中*{arch}*根据实际运行环境选择“x86_64”或者“aarch64”）来清除Gstreamer的历史缓存，再运行程序即可。

----结束

8.9 编译时出现 libgfortran.so.4 报错

问题现象

编译时出现依赖不存在的报错**libgfortran.so.4: No such file or directory**。

原因分析

“opensource”文件中的so文件需要依赖“libgfortran.so.4”，在编译环境中找不到。

解决方案

参考以下命令执行，安装gfortran（如果环境中找不到gfortran-4，也可以安装大于或等于4的版本）。

```
apt-get update
apt-get install gfortran-4
```

8.10 在 Kunpeng 服务器中运行多路视频分析，出现 CPU 资源被占满，导致业务堆积、断流的情况

问题现象

在Kunpeng 920 7260服务器中运行多路视频分析，出现CPU资源被占满，导致业务堆积、断流的情况。

原因分析

视频分析业务不要求数据预取，CPU的预取开关打开会造成内存带宽浪费。

解决方案

在BIOS，把CPU的预取开关设置为关闭。

8.11 调用 API 参考（Python）视频解码/编码接口结束时出现 core dumped 现象

问题现象

调用Python视频解码功能，若videoDecoder在全局作用域（非函数内）定义和执行，程序结束出现Segmentation fault (core dumped) 或 Aborted (core dumped)，编码接口同样出现类似情况。

原因分析

由于Pybind问题导致在全局作用域定义videoDecoder无法自动调用析构函数，Python退出解释器时回调线程执行出错。

解决方案

- 方案一：在函数或类的成员方法中定义和运行videoDecoder。
- 方案二：若在全局作用域中定义videoDecoder，则在程序结束时使用del函数删除构造的videoDecoder对象。

8.12 在 Atlas 500 A2 智能小站默认配置下，无法安装 mxVision

问题现象

在Atlas 500 A2 智能小站默认配置下，安装mxVision软件包，发现容量不足，无法完成安装。

原因分析

由于Atlas 500 A2 智能小站默认配置用户区的实际存储空间较小，无法安装mxVision软件包。

解决方案

请参见《**Atlas 500 A2 智能小站 用户指南**》“安装与配置 > 安装硬件选件”章节安装硬件并根据《**Atlas 500 A2 智能边缘管理系统 用户指南**》“Web界面功能介绍 > 管理 > 存储配置 > 本地磁盘管理”章节进行挂载、分区操作，完成扩容后再安装mxVision软件包。

8.13 执行 run.sh 脚本时报错 cannot find factory: mxpi_classpostprocessor

问题描述

进入“MyFirstApp/src”目录，执行编译脚本**./run.sh**时，报错如下内容，说明无法找到模型后处理插件。

```
E20220525 07:10:11.514457 123253 MxsmElement.cpp:413] [6014][element invalid factory] Feature is NULL, can not find the element factory: mxpi_classpostprocessor
E20220525 07:10:11.514672 123253 MxsmElement.cpp:779] factoryName(mxpi_classpostprocessor) is an invalid element factory.
E20220525 07:10:11.514691 123253 MxsmStream.cpp:1853] [6014][element invalid factory] mxpi_classpostprocessor0 is an invalid element of mxpi_classpostprocessor.
E20220525 07:10:11.514716 123253 MxsmStream.cpp:725] [6014][element invalid factory] Creates classification Stream failed.
E20220525 07:10:11.514737 123253 MxStreamManagerDptr.cpp:549] [6014][element invalid factory] create stream(classification) failed.
E20220525 07:10:11.514768 123253 main.cpp:103] [6014][element invalid factory] Failed to create Stream.
```

可能原因

Python版本错误，未正确配置环境变量，导致无法找到正确库。

解决方法

步骤1 使用以下命令检查Python版本，若非返回Python 3.9.2，则需要参照**安装Python3.9.2**进行安装。

```
python3 --version
```

步骤2 将Python 3.9.2安装目录下的“libpython3.9.so.1.0”拷贝至“/usr/lib64/”路径下。

```
mkdir /usr/lib64
cp /usr/local/Python-3.9.2/libpython3.9.so.1.0 /usr/lib64/
```

步骤3 执行以下命令清除Gstreamer历史缓存。

```
rm ~/.cache/gstreamer-1.0/registry.aarch64.bin
```

步骤4 重新运行脚本即可。

```
./run.sh
```

----结束

8.14 尝试打印 Device 侧的 Tensor、Image 内存数据时，发生 coredump

问题现象

当Tensor、Image类数据在Device侧时，尝试调用GetData()接口返回指针并打印指针所指向的数据时，出现coredump提示。

原因分析

Device侧地址空间与Host侧地址空间相互独立，Host侧无法直接访问Device侧数据，具体请参见《CANN 应用软件开发指南 (C&C++)CANN 应用软件开发指南 (C&C++)》。

解决方案

请先使用**Tensor.ToHost()**或**Image.ToHost()**，将Device侧Tensor类或Image类数据转移到Host侧后，再次尝试打印数据操作。

8.15 使用 mxVersion 安装时，出现 libpython3.9.so.1.0 报错

问题现象

使用命令**./Ascend-mindxsdk-mxvision_{version}_linux-{arch}.run --install**安装版本包时，出现如下报错：

libpython3.9.so.1.0: cannot open shared object file: No such file or directory

原因分析

Python3.9需要的so文件依赖libpython3.9.so.1.0，在编译环境中找不到。

解决方案

将Python安装路径下的libpython3.9.so.1.0对象，复制到“/usr/lib”或“/usr/lib64”下。

8.16 mxpi_rtspsrc 拉流插件运行偶现 BUS ERROR：XXX streamming stopped, reason error (-5) 错误

问题现象

pipeline运行过程中，mxpi_rtspsrc插件报错Internal data stream error XXX streaming stopped, reason error (-5)的错误

```
E20231008 17:48:14.074693  6556 MxsmStream.cpp:336] Bus error: Could not write to resource.(../subprojects/gst-plugins-g
ood/gst/rtsp/gstrtspsrc.c(7662): gst_rtspsrc_setup_streams_start (): /GstPipeline:stream15/GstRtspSrcBin:mxpi_rtspsrc15/
GstRTSPSrc:bin_mxpi_rtspsrc_rtspsrc_mxpi_rtspsrc15_0:
Error (500): Internal Server Error).
E20231008 17:48:14.075425  6556 MxsmStream.cpp:336] Bus error: Could not read from resource.(../subprojects/gst-plugins-
good/gst/rtsp/gstrtspsrc.c(5901): gst_rtspsrc_loop_udp (): /GstPipeline:stream15/GstRtspSrcBin:mxpi_rtspsrc15/GstRTSPSrc
:bin_mxpi_rtspsrc_rtspsrc_mxpi_rtspsrc15_0:
Could not receive message. (System error)).
E20231008 17:48:14.075448  6556 MxsmStream.cpp:336] Bus error: Internal data stream error.(../subprojects/gst-plugins-go
od/gst/rtsp/gstrtspsrc.c(6252): gst_rtspsrc_loop (): /GstPipeline:stream15/GstRtspSrcBin:mxpi_rtspsrc15/GstRTSPSrc:bin_m
xpi_rtspsrc_rtspsrc_mxpi_rtspsrc15_0:
streaming stopped, reason error (-5)).
```

原因分析

主要是由于数据源网络连接异常中断、或者是连接时cpu占用较高引起的偶现错误。

解决方案

mxpi_rtspsrc插件具有重连机制，等待30秒会尝试重连，建议等待重连成功即可。

A 安装 7.3.0 版本 gcc

以下步骤请在root用户下执行。

步骤1 下载gcc-7.3.0.tar.gz，下载地址为**https://mirrors.tuna.tsinghua.edu.cn/gnu/gcc/gcc-7.3.0/gcc-7.3.0.tar.gz**。

步骤2 安装gcc时候会占用大量临时空间，所以先执行下面的命令清空“/tmp”目录：

```
rm -rf /tmp/*
```

步骤3 安装依赖（以CentOS和Ubuntu系统为例）。

- CentOS执行如下命令安装。
  ```
  yum install bzip2
  ```
- Ubuntu执行如下命令安装。
  ```
  apt-get install bzip2
  ```

步骤4 编译安装gcc。

1. 进入“gcc-7.3.0.tar.gz”源码包所在目录，解压源码包，命令为：
   ```
   tar -zxvf gcc-7.3.0.tar.gz
   ```
2. 进入解压后的文件夹，执行如下命令下载gcc依赖包：
   ```
   cd gcc-7.3.0
   ./contrib/download_prerequisites
   ```
   如果执行上述命令报错，需要执行如下命令在“gcc-7.3.0/”文件夹下下载好以下依赖包：
   ```
   wget http://gcc.gnu.org/pub/gcc/infrastructure/gmp-6.1.0.tar.bz2
   wget http://gcc.gnu.org/pub/gcc/infrastructure/mpfr-3.1.4.tar.bz2
   wget http://gcc.gnu.org/pub/gcc/infrastructure/mpc-1.0.3.tar.gz
   wget http://gcc.gnu.org/pub/gcc/infrastructure/isl-0.16.1.tar.bz2
   ```
   下载好上述依赖包后，重新执行以下命令：
   ```
   ./contrib/download_prerequisites
   ```
   如果上述命令校验失败，需要确保依赖包为一次性下载成功，无重复下载现象。
3. 执行配置、编译和安装命令：
   ```
   ./configure --enable-languages=c,c++ --disable-multilib --with-system-zlib --prefix=/usr/local/gcc7.3.0
   make -j15    # 通过grep -w processor /proc/cpuinfo|wc -l查看cpu数，示例为15，用户可自行设置相应参数。
   make install
   ```

说明

其中“--prefix”参数用于指定gcc7.3.0安装路径，用户可自行配置，但注意不要配置为“/usr/local”及“/usr”，因为会与系统使用软件源默认安装的gcc相冲突，导致系统原始gcc编译环境被破坏。示例指定为“/usr/local/gcc7.3.0”。

步骤5 配置环境变量（请在实际需要时再进行配置）。

```
export LD_LIBRARY_PATH=/usr/local/gcc7.3.0/lib64:${LD_LIBRARY_PATH}
export CC=/usr/local/gcc7.3.0/bin/gcc
export CXX=/usr/local/gcc7.3.0/bin/g++
export PATH=/usr/local/gcc7.3.0/bin:${PATH}
```

其中“/usr/local/gcc7.3.0”为**步骤4.3**中配置的gcc7.3.0安装路径，请根据实际情况替换。

----结束

B Python 依赖安装参考

介绍

目前mxVision开发套件包依赖Python 3.9，如环境中未配置Python 3.9，可参考本章节步骤安装依赖（以root用户安装Python 3.9.2为例）。

操作步骤

步骤1 安装Python 3.9.2依赖库。

```
apt-get install -y build-essential gcc g++ make cmake zlib1g zlib1g-dev libsqlite3-dev openssl libssl-dev
libffi-dev unzip pciutils net-tools libblas-dev gfortran libblas3 libopenblas-dev
```

步骤2 下载Python 3.9.2源码安装包。

```
wget https://www.python.org/ftp/python/3.9.2/Python-3.9.2.tgz
```

步骤3 解压Python 3.9.2安装包。

```
tar -xzvf Python-3.9.2.tgz
```

步骤4 编译安装Python 3.9.2。

```
cd Python-3.9.2 && ./configure --prefix=/usr/local/python3.9.2 --enable-shared && make -j8 && make install
```

步骤5 拷贝libpython3.9.so.1.0到系统路径。

```
cp /usr/local/python3.9.2/lib/libpython3.9.so.1.0 /usr/lib
```

步骤6 设置环境变量。

```
export LD_LIBRARY_PATH=/usr/local/python3.9.2/lib:$LD_LIBRARY_PATH
export PATH=/usr/local/python3.9.2/bin:$PATH
```

----结束

说明

若源码方式安装完Python 3.9.2后，pip3工具无法使用，请参见**8.6 源码方式安装完Python 3.9.2后，pip3工具无法使用**解决。

C 日志配置文件

日志框架基于glog实现，提供配置文件设置日志基本参数。

日志配置文件（logging.conf）

```
# MindX SDK configuration file

# Program name
# 项目工程名称，任意指定
program_name=mindx_sdk

# log output directory, create logs directory before use
# 日志输出文件夹，若是绝对路径，则会输出在该绝对路径下，该绝对路径只能设置在用户主目录下。若非绝对路径，则会输出在用户主目录下的/log/mindxsdk 与log_dir的拼接路径下。
log_dir=logs

# log file basefile
# 日志文件名前缀
base_filename=mxsdk.log.

# time to buffer the log in seconds，0 means write to file immediately
# 设置日志输出的周期（以秒为单位），0表示立即输出
logbufsecs=0

# will output to file, where level >= global_level,default is 0
# Log level: -1-debug，0-info
# 输出级别大于等于该数值的日志到文件，默认为0
global_level=0

# will output to stderr, where level >= console_level,default is 2
# Log level: -1-debug，0-info, 1-warn, 2-error, 3-fatal
# 输出级别大于等于该数值的日志到stderr，默认为2
console_level=2

# max log size, unit is MB, max_log_size must be an integer, default is 100. 1 <= value <= 1024
# 日志文件size最大值，单位为MB，max_log_size的值必须为大于等于1的正整数，默认值为100
max_log_size=100

# rotate time of log switch: 7 days(default), 1<= value <= 1000
# 日志文件天数回滚开关：默认值7天
# 日志文件天数回滚基于单一进程进行统计（只统计当前进程的日志）；
rotate_day=7

# rotate file number of log switch: 50 (default, For each log level). 1 <= value <= 500
```

```
# 日志文件个数回滚开关：默认值50个（日志各个级别）
# 日志文件个数回滚基于单一进程进行统计（只统计当前进程的日志）；
# 只有在程序正常退出时，才能保证日志文件的数量和rotate_file_number的值一致，用户异常退出程序时，实际日志数量可能会超过rotate_file_number。
rotate_file_number=50

# frequency of printing each log, frequency value must be an integer, defualt value is 1. 1 <= value <= 10000
# 每条日志的打印频率，参数值必须为整型，默认值为1，参数值填写范围在1~10000之间。
flow_control_frequency=1
```

D Metadata proto 文件

说明

“opensource/bin/”目录下的protoc为Google Protobuf的编译工具，若用户在插件中使用了自定义的数据结构，protoc可以将其.proto文件生成相应的接口代码。

Google Protobuf 数据结构定义文件（MxpiDumpData.proto）

```
syntax = "proto3";

package MxTools;

message MxpiDumpData
{
    Buffer buffer = 1;
    repeated MetaData metaData = 2;
}

message Buffer
{
    bytes bufferData = 1;
}

message MetaData
{
    string key = 1;
    string content = 2;
    string protoDataType = 3;
}
```

Google Protobuf 数据结构定义文件（MxpiDataType.proto）

```
syntax = "proto3";

package MxTools;

// 用于存放视频和图像帧，包含帧信息，数据信息。
message MxpiFrame
{
    MxpiFrameInfo frameInfo = 1;
    MxpiVisionList visionList = 2;   // 为了和VPC统一，使用list。
}

message MxpiFrameInfo              // 帧信息
```

```
{
    uint32 channelId = 1;
    uint32 frameId = 2;
    bool isEos = 3;
}

message MxpiVisionList              // 视频、图像数据列表
{
    repeated MxpiVision visionVec = 1;
}

message MxpiVision                      // 视频、图像数据结构
{
    repeated MxpiMetaHeader headerVec = 1; //存放插件信息的vector
    MxpiVisionInfo visionInfo = 2;
    MxpiVisionData visionData = 3;
}

message MxpiMetaHeader              // 数据结构头，用于序列化
{
    string parentName = 1;          //此变量在后续版本中将会被弃用，请使用“dataSource”
    int32 memberId = 2;             //MxpiVisionList中对应存放的MxpiVision索引
    string dataSource = 3;          //依赖数据的索引名,通过该索引获取依赖的元数据
}

enum MxpiMemoryType {
    MXPI_MEMORY_HOST = 0;
    MXPI_MEMORY_DEVICE = 1;
    MXPI_MEMORY_DVPP = 2;
    MXPI_MEMORY_HOST_MALLOC = 3;
    MXPI_MEMORY_HOST_NEW = 4;
}

enum MxpiDataType {                 // 数据类型，包括8位无符号整型类型和32位浮点类型
    MXPI_DATA_TYPE_UINT8 = 0;
    MXPI_DATA_TYPE_FLOAT32 = 1;
}
message MxpiVisionInfo              // 视频、图像描述信息
{
    uint32 format = 1;              //描述图像的格式，具体请参见DvppWrapper.h的
MxbasePixelFormat
    uint32 width = 2;
    uint32 height = 3;
    uint32 widthAligned = 4;
    uint32 heightAligned = 5;
    uint32 resizeType = 6;          //描述缩放类型，具体请参见MxPluginsUtils.h的RESIZETYPE
    float keepAspectRatioScaling = 7; //缩放比例，取值在[1/32, 16]
    repeated MxpiVisionPreProcess preprocessInfo = 8;
}

message MxpiVisionPreProcess        // 图像前处理信息。例如图像宽高、抠图坐标、贴图坐标
{
    uint32 widthSrc = 1;
    uint32 heightSrc = 2;
    uint32 cropLeft = 3;
    uint32 cropRight = 4;
    uint32 cropTop = 5;
    uint32 cropBottom = 6;

    uint32 pasteLeft = 7;
    uint32 pasteRight = 8;
    uint32 pasteTop = 9;
```

```
    uint32 pasteBottom = 10;

    uint32 interpolation = 11;
    string elementName = 12;
}
message MxpiVisionData              // 视频、图像数据内容
{
    uint64 dataPtr = 1;             // 视频、图像内存指针
    int32 dataSize = 2;
    uint32 deviceId = 3;
    MxpiMemoryType memType = 4;
    uint64 freeFunc = 5;            // 视频、图像内存销毁函数
    bytes dataStr = 6;              // bytes数据类型 序列化成JSON时会自动进行base64编码
    MxpiDataType dataType = 7;
    uint64 matPtr = 8;
}
message MxpiObjectList              // 目标列表
{
    repeated MxpiObject objectVec = 1;
}

message MxpiObject                  // 目标数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    float x0 = 2;
    float y0 = 3;
    float x1 = 4;
    float y1 = 5;
    repeated MxpiClass classVec = 6; // 类别信息数据结构，此处“MxpiMetaHeader”无效
    MxpiImageMask imageMask = 7;    // 图像语义分割数据信息
}

message MxpiImageMaskList           // 图像语义分割数据列表
{
    repeated MxpiImageMask imageMaskVec = 1;
}

message MxpiImageMask               // 图像语义分割数据信息。例如数据的类别、形状、类别信息
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated string className = 2;
    repeated int32 shape = 3;
    int32 dataType = 4;
    bytes dataStr = 5;
}
message MxpiClass                   // 类别信息数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    int32 classId = 2;
    string className = 3;
    float confidence = 4;
}

message MxpiClassList               // 类别信息列表
{
    repeated MxpiClass classVec = 1;
}

message MxpiAttributeList           // 属性信息列表
{
    repeated MxpiAttribute attributeVec = 1;
}
```

```
message MxpiAttribute            // 属性信息数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    int32 attrId = 2;
    string attrName = 3;
    string attrValue = 4;
    float confidence = 5;
}

message MxpiTrackLetList
{
    repeated MxpiTrackLet trackLetVec = 1;
}

message MxpiTrackLet
{
    repeated MxpiMetaHeader headerVec = 1;
    uint32 trackId = 2;
    uint32 age = 3;                 // 目标“存活”帧数
    uint32 hits = 4;                // 目标被成功匹配帧数
    int32 trackFlag = 5;            // 状态
}

message MxpiTensorPackageList  // 模型Tensor组合列表
{
    repeated MxpiTensorPackage tensorPackageVec = 1;
}

message MxpiTensorPackage   // 模型Tensor组合数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated MxpiTensor tensorVec = 2;
}

message MxpiTensor       // 模型Tensor数据结构
{
    uint64 tensorDataPtr = 1;             // Tensor内存指针
    int32 tensorDataSize = 2;
    uint32 deviceId = 3;
    MxpiMemoryType memType = 4;
    uint64 freeFunc = 5;            // Tensor内存销毁函数
    repeated int32 tensorShape = 6;       // Tensor维度
    bytes dataStr = 7;
    int32 tensorDataType = 8;
}

message MxpiFeatureVectorList   // 特征向量列表
{
    repeated MxpiFeatureVector featureVec = 1;
}

message MxpiFeatureVector    // 特征向量数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated float featureValues = 2;
}

message MxpiPoseList                        // 人体姿态估计数据列表
{
    repeated MxpiPose poseVec = 1;
}
```

```
message MxpiPose                        // 人体姿态估计数据信息
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated MxpiKeyPoint keyPointVec = 2;
    float score = 3;
}

message MxpiKeyPoint                    // 人体关键点数据信息
{
    float x = 1;
    float y = 2;
    int32 name = 3;
    float score = 4;
}
message MxpiKeyPointAndAngleList        // 关键点和角度信息列表
{
    repeated MxpiKeyPointAndAngle keyPointAndAngleVec = 1;
}

message MxpiKeyPointAndAngle
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated float keyPointsVec = 2;        // 目标对应的五个关键点信息
    float angleYaw = 3;                     // 偏航角
    float anglePitch = 4;                   // 俯仰角
    float angleRoll = 5;                    // 横滚角
}
message MxpiTextObjectList                          // 文本目标数据列表
{
repeated MxpiTextObject objectVec = 1;
}

message MxpiTextObject                              // 文本目标数据信息。例如文本目标框坐标坐
标、置信度、检测文本
{
repeated MxpiMetaHeader headerVec = 1;
float x0 = 2;
float y0 = 3;
float x1 = 4;
float y1 = 5;
float x2 = 6;
float y2 = 7;
float x3 = 8;
float y3 = 9;
float confidence = 10;
string text = 11;
}

message MxpiTextsInfoList                           // 文本生成数据列表
{
repeated MxpiTextsInfo textsInfoVec = 1;
}

message MxpiTextsInfo                               // 文本生成数据信息
{
repeated MxpiMetaHeader headerVec = 1;
repeated string text = 2;
}
message MxpiCustomDataList
{
    repeated MxpiCustomData dataVec = 1;
```

```
}
message MxpiCustomData
{
    repeated MxpiMetaHeader headerVec = 1;
    map<string, string> map = 2;
}
```

OSD protobuf 数据结构定义文件（MxpiOSDType.proto）

```
syntax = "proto3";

package MxTools;

import "MxpiDataType.proto";

message MxpiOsdInstancesList                    //目标或者分类信息Osd列表
{
    repeated MxpiOsdInstances osdInstancesVec = 1;
}

message MxpiOsdInstances                        //Osd属性描述信息数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    repeated MxpiOsdText osdTextVec = 2;        //文字描述
    repeated MxpiOsdLine osdLineVec = 3;        //线条描述
    repeated MxpiOsdRect osdRectVec = 4;        //矩形框描述
    repeated MxpiOsdCircle osdCircleVec = 5;    //画圆描述
}

message MxpiOsdText                             //文字属性数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    string text = 2;                            //要添加的文字内容
    int32 x0 = 3;                               //添加文字原点坐标的X坐标
    int32 y0 = 4;                               //添加文字原点坐标的Y坐标
    int32 fontFace = 5;                         //文字的字体类型
    double fontScale = 6;                       //文字的字体大小
    bool bottomLeftOrigin = 7;                  //取值为true，原点坐标为文字左上角坐标，false为左下角坐
标
    MxpiOsdParams osdParams = 8;    //Osd公有属性实例对象
    bool fixedArea = 9;             //缩放后，Osd属性描述信息是否按照原图等比例缩放
}

message MxpiOsdLine                             //线条属性数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    int32 x0 = 2;                   //线条左上角X坐标
    int32 y0 = 3;                   //线条左上角Y坐标
    int32 x1 = 4;                   //线条右下角X坐标
    int32 y1 = 5;                   //线条右下角Y坐标
    MxpiOsdParams osdParams = 6;    //Osd公有属性实例对象
}

message MxpiOsdRect                             //矩形框属性数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    int32 x0 = 2;                   //矩形框左上角X坐标
    int32 y0 = 3;                   //矩形框左上角Y坐标
    int32 x1 = 4;                   //矩形框右下角X坐标
    int32 y1 = 5;                   //矩形框右下角Y坐标
    MxpiOsdParams osdParams = 6;    //Osd公有属性实例对象
    bool fixedArea = 7;             //缩放后，Osd属性描述信息是否按照原图等比例缩放
```

```
}

message MxpiOsdCircle                          //画圆属性数据结构
{
    repeated MxpiMetaHeader headerVec = 1;
    int32 x0 = 2;                      //画圆左上角X坐标
    int32 y0 = 3;                      //画圆左上角Y坐标
    int32 radius = 4;                    //画圆的半径
    MxpiOsdParams osdParams = 5;      //Osd公有属性实例对象
}

message MxpiOsdParams               //Osd公有属性数据结构
{
    uint32 scalorB = 1;               //颜色B通道值，取值范围0-255
    uint32 scalorG = 2;               //颜色G通道值，取值范围0-255
    uint32 scalorR = 3;               //颜色R通道值，取值范围0-255
    int32 thickness = 4;              //粗细大小
    int32 lineType = 5;               //线条类型
    int32 shift = 6;                  //缩放参数
}
```

E 公网地址说明

MindX SDK的安装包中的网址support.huawei.com和www.huawei.com为华为企业产品支持网站，安装结束后会被清除，并不会访问，不会造成风险。

MindX SDK编译的开源软件中会存在公开网址和邮箱地址，SDK本身不会访问，不会造成风险。

OSD算子编译生成的libcust_aicpu_kernels.so文件，链接了开源三方库，其中包含公网地址（https://github.com、https://developers.google.com），SDK本身不会访问，不会造成风险。

更多公网地址和邮箱地址请参考MindX 5.0.RC3 mxVision公网地址和邮箱地址.xlsx。

F 修订记录

发布日期	修订记录
2023-10-17	第一次正式发布